Objetos Distribuídos

Celso Gonçalves Júnior

DCC – IMECC – UNICAMP

Departamento de Ciência da Computação
Instituto de Matemática, Estatística e Ciência da Computação
Universidade Estadual de Campinas

Objetos Distribuídos

Celso Gonçalves Júnior

Este exemplar corresponde à redação final da tese corrigida e defendida pelo Sr. Celso Gonçalves Júnior e aprovada pela Comissão Julgadora.

Cidade Universitária Zeferino Vaz, Campinas, 16 de setembro de 1994

Prof. Dr. Rogério Drummond Burnier Pessôa de Mello Filho

Dissertação apresentada ao Instituto de Matemática, Estatística e Ciência da Computação, da Universidade Estadual de Campinas, como requisito parcial para obtenção do título de Mestre em Ciência da Computação.

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP

Gonçalves Junior, Celso

G586o Objetos distribuidos / Celso Gonçalves -- Campinas, [S.P. :s.n.], 1994.

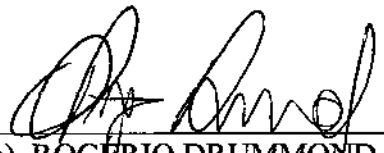
Orientador : Rogerio Drummond Burnier Pessoa de Mello Filho

Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Matemática, Estatística e Ciência da Computação.

1. Programação orientada a objetos. 2. Linguagem de programação (Computadores). 3. Sistemas operacionais distribuídos (Computadores). I. Mello Filho, Rogerio Drummond B. P. de. .II. Universidade Estadual de Campinas. Instituto de Matemática, Estatística e Ciência da Computação. III. Título.

Tese de Mestrado defendida e aprovada em 16 de setembro de 1994

Pela Banca Examinadora composta pelos Profs. Drs.



Prof (a). Dr (a). ROGERIO DRUMMOND BURNIER P. DE MELLO FILHO

Prof (a). Dr (a). RICARDO DE OLIVEIRA ANIDO

Prof (a). Dr (a). ELERI CARDOZO

Dedicado a meus pais

Agradecimentos

A minha família merece todos os agradecimentos possíveis. Meus pais, Celso e Maria do Carmo, deram, sempre que a defesa da tese era adiada, demonstrações de tolerância cada vez mais espetaculares. Minha irmã Estela, que é mestranda na Unicamp com a aplicação costumeira, nunca me censurou pelo desânimo, que atacou agudamente várias vezes.

A minha querida Clara foi, por várias vezes, muito mais do que se espera de uma companheira.

Os irmãos Dindi e Rock Tin Tin, meus dois pequenos, alegres, queridos amigos, não deixam nunca que eu me sinta sozinho — nem que eu queira.

Meus colegas no Laboratório A_Hand me ajudaram muito; meu agradecimento vai em particular para Bill Coutinho de Oliveira, Carlos Alberto Furuti, Cassius Di Cianni, Mauricio Fernández e Wanderley Ceschim.

Há muito tempo que eu e meu caro Alexandre Teles estamos juntos nessa história repleta de computadores. Já são uns 10 anos, durante os quais também estiveram por perto muitas esperanças, dificuldades e felicidades. Pelo jeito, continuaremos assim por um bom tempo.

O meu orientador, Prof. Dr. Rogério Drummond, colaborou principalmente com sua amizade. No mais, entrou no negócio com quase tudo: com o problema, com partes das soluções, com um rascunho do mapa, e com as perguntas certas.

Os Profs. Drs. Ricardo de Olivera Anido e Eleri Cardozo, membros da banca, honraram a defesa da tese com observações importantes para o aperfeiçoamento do trabalho. Agradeço ainda ao Prof. Dr. Hans Liesenberg, membro suplente da banca, pela aceitação do convite.

A Coordenadoria de Aperfeiçoamento de Pessoal de Ensino Superior (CAPES), órgão do Ministério da Educação, e a Pró-Reitoria de Pós-Graduação da Unicamp financiaram com uma bolsa de estudos os dois primeiros anos do meu mestrado. Esse apoio foi essencial para que eu deixasse meu emprego na Indústria e voltasse a estudar.

A minha escola, a Unicamp, é uma extensão da minha casa, e são meus irmãos todos os que têm esse mesmo sentimento.

Muito do que eu sei sobre computadores, e provavelmente muitas das coisas mais interessantes, aprendi apenas ouvindo os Profs. Drs. Nelson Machado, Arthur Catto e Timothy Griffin. Ser aluno deles é uma honra suficiente para mim.

Tenho outros professores (e professoras) a quem admiro muito, sobretudo porque nunca me ensinaram nada sobre computadores. A Profª Dora Fraiman Blatyta, a Profª Drª Victoria Namestnikov El-Murr, e o Dr. Rogério Burnier, cada um à sua particularíssima e especialíssima maneira, me deram o gosto de descobrir muito sobre outras culturas e outros mundos.

Agradeço a Deus por conhecer todas estas pessoas, e pela oportunidade de estar aqui, de aprender e de fazer.

Para alcançar toda beleza,
três coisas me parecem sempre
necessárias: esperança, luta e conquista.

Luis Buñuel

Resumo

A programação orientada a objetos oferece uma sólida base conceitual para o desenvolvimento de sistemas de informação, com ênfase em modularidade, extensibilidade e robustez. Por outro lado, avanços nas tecnologias de *hardware*, arquitetura e meios de comunicação propiciaram uma mudança nos padrões de ambiente de processamento de dados: o modelo de *mainframes* vem sendo substituído pelo modelo distribuído, baseado em redes de estações de trabalho.

A combinação desses dois conceitos, objetos e sistemas distribuídos, oferece a curto prazo perspectivas bastante promissoras. Várias iniciativas visando esse objetivo vêm sendo desenvolvidas ou estão já em uso. Embora esses trabalhos adotem diferentes abordagens, eles procuram basicamente unificar conceitos das duas áreas, estabelecendo um modelo único de programação.

Trabalhos desenvolvidos atualmente no Laboratório de Pesquisas do Projeto A_Hand objetivam a construção de um ambiente de desenvolvimento de *software*, projetado para dar suporte à construção de aplicações de grande magnitude. As linguagens de programação Cm e LegoShell são as ferramentas básicas para programação nesse ambiente. Cm é orientada a objetos, sendo usada para gerar componentes de código reaproveitáveis, que serão combinados através da linguagem gráfica LegoShell. Neste documento é definida uma nova versão da linguagem Cm, que vai atender os requisitos intrínsecos da programação concorrente e distribuída, e que definirá o modelo de aplicações distribuídas no nosso ambiente.

A tarefa de adaptar uma linguagem orientada a objetos, para suprir as necessidades da programação distribuída sem alterar sua semântica, é difícil porque vários aspectos devem ser considerados. A questão principal nos sistemas distribuídos é a transparência de localização; analisamos também soluções relativas a concorrência, exceções e herança. As extensões à linguagem Cm propostas aqui incluem novas construções sintáticas, a definição do conceito de "objeto remoto" e um conjunto de classes e funções pré-definidas.

Abstract

The concepts introduced by the object paradigm embody a complete, sound framework for developing information systems with improved modularity, extendibility and robustness. On the other hand, technological advances in hardware components, computer architecture and communications have caused standards about computing environments to change: the old mainframe model has been steadily shifted in favor of the distributed model, based on workstations networks instead.

Blending these two concepts, objects and distributed systems, offers very promising possibilities in the near future. Indeed, several attempts towards this goal are either under execution or already accomplished. These works, while adopting different approaches and solutions, basically aim at unifying related issues from the two areas within a consistent programming model.

Current efforts at the A_Hand Research Laboratory target the construction of a complex software development environment, planned to support the design and implementation of very large applications. The programming languages Cm and LegoShell are the basic programming tools in this environment. The Cm language is object-oriented, used to produce reusable software components, which can be brought together through the graphical language LegoShell. In this document, we define a new version for the language Cm, in order to meet inherent requirements of concurrent and distributed programming, thus establishing our underlying model for distributed applications programming.

Adapting an existing object-oriented language to address programming issues in a distributed background while preserving its semantic consistency results a hard task, as a number of important aspects must be kept in mind. The main concern in distributed programming systems is location transparency; we have also investigated solutions regarding concurrency, exceptions and inheritance problems. The extensions to the Cm language here proposed include new syntatic constructions, the definition of the "remote object" concept and a set of standard classes and functions.

Índice

Capítulo 3 Sistemas Distribuídos

Capítulo 4 O Ambiente A_Hand

Capítulo 5 Objetos Distribuídos em Cm

Lista de Tabelas

Lista de Figuras

1 Apresentação

Dos diversos instrumentos utilizados pelo homem, o mais espetacular é, sem dúvida, o livro. Os demais são uma extensão de seu corpo. O microscópio, o telescópio são extensões de sua visão: o telefone é a extensão de sua voz; em seguida, temos o arado e a espada, extensões de seu braço. O livro, porém, é outra coisa: o livro é uma extensão da memória e da imaginação.

Jorge Luis Borges

Este é um trabalho que visa dar uma contribuição às pesquisas sendo desenvolvidas para integrar duas grandes áreas da Ciência da Computação: Sistemas Distribuídos e Linguagens de Programação. Aqui é apresentado um modelo de programação distribuída projetado com base em objetos, uma área de pesquisa que tem recebido atenção cada vez maior por parte das comunidades acadêmica e industrial, e que está sendo apontado como o paradigma dominante dos próximos anos.

O presente trabalho, com respeito a seus fundamentos, implementação e objetivos, faz parte dos projetos sendo desenvolvidos no Laboratório A_Hand [Dru87a], do Departamento de Ciência de Computação da Universidade Estadual de Campinas. Este é um grupo de pesquisa e desenvolvimento empenhado em oferecer soluções concretas para o problema de se produzir *software* de qualidade. O objetivo principal desse grupo é implementar um ambiente de desenvolvimento (denominado ambiente A_Hand) de *software* para a construção de sistemas de grande magnitude e complexidade. O trabalho nesse ambiente de desenvolvimento levou a pesquisas nas áreas de linguagens de programação, sistemas distribuídos, hipertextos, interfaces com o usuário e *groupware*.

1.1 Motivação do trabalho

Desde o advento dos computadores digitais, na década de 40, a indústria de informática tem crescido consistentemente ano após ano. Com o aparecimento de computadores pessoais, no início da década de 70, surgiu um formidável mercado consumidor, com diversos campos de atuação. Hoje, pode-se ver a presença do computador em praticamente todos os ramos da atividade intelectual, econômica e artística.

A tecnologia de projeto e produção de computadores reagiu muito rapidamente às demandas do mercado por máquinas mais rápidas e poderosas. A capacidade de processamento e armazenamento das máquinas cresceu num ritmo desconcertante,

sempre acompanhada de redução no preço relativo. Desde o primeiro computador, equipado a válvulas, houve avanços apreciáveis no projeto de processadores, memória, meios de armazenamento e de comunicação. Os avanços continuam se sucedendo, e ninguém tem dúvida de que os computadores que serão vendidos daqui a 10 anos serão muito mais rápidos, poderosos, baratos e fáceis de usar que os computadores de hoje.

Entretanto, parte do potencial dos computadores de hoje espera para ser aproveitada. Isso porque o avanço obtido nas tecnologias de *hardware* não foi acompanhado por um avanço comparável do *software*. Muito embora as tarefas do programador de hoje sejam muito diferentes daquelas dos programadores de 40 anos atrás, a produção de programas continua sendo uma atividade quase artesanal, que tem resistido ao enquadramento a técnicas ou metodologias, e que continua dependendo fundamentalmente da inteligência, da criatividade e da experiência do programador. As razões para isso são diversas, e parecem derivar do aspecto particularíssimo da atividade de programar computadores. Essa atividade é, na sua essência, abstrata demais para que os procedimentos e métodos envolvidos possam ser formalmente descritos e estabelecidos.

Da mesma maneira, os programas, que são o resultado do trabalho dos programadores, também são difíceis de especificar, avaliar e comparar. Verificar se um programa está correto, da mesma maneira como se verifica um circuito elétrico ou uma peça de motor, é praticamente impossível na maioria dos casos. E, no entanto, com tantas incertezas envolvidas, o mercado mundial de *software* é gigantesco, e computadores são usados tanto para tarefas simples como para tarefas importantes (como sistemas bancários e de informação) ou que envolvem vida humana (aplicações médicas, tráfego aéreo, controle de dispositivos militares e nucleares, etc).

A crescente disponibilidade de recursos computacionais e de campos de aplicação dos computadores deixa, portanto, um desafio aos programadores: é preciso modificar o panorama atual para não desperdiçar as oportunidades que se descortinam. Em resumo, é preciso responder às demandas do mercado produzindo *software* de qualidade, flexível e de baixo preço, com gasto aceitável de tempo, recursos materiais e mão-de-obra.

O chamado paradigma de objetos apresenta-se hoje como a tecnologia a ser utilizada para alcançar um ganho significativo nos processos de produção de *software*. Embora o tema ainda não esteja completamente estabelecido, um grande número de experiências acadêmicas e comerciais mostraram-se bem-sucedidas, ou na pior das hipóteses chegaram a resultados que levam a aperfeiçoamentos e extensões ao modelo.

1.2 Descrição do problema

No começo da década de 80, o surgimento das chamadas estações de trabalho, operando em redes locais (LANs), causou uma mudança no padrão de ambiente de processamento de dados. Durante muito tempo, os computadores de grande porte (*mainframes*) eram usados quase que exclusivamente em instituições comerciais, militares e acadêmicas. Esses ambientes, ditos centralizados, eram compostos por

um ou alguns *mainframes*, equipamentos acessórios como controladoras de comunicação, unidades de disco e impressoras, e por um número arbitrário de terminais, usados para entrada de dados, controle, interface das aplicações, etc.

Os chamados computadores pessoais destinavam-se exclusivamente a tarefas simples, como edição de documentos, pequenos bancos de dados, jogos, ou mesmo como terminais de computadores de maior porte. O desempenho desses computadores pessoais era muito modesto, assim como sua capacidade de armazenamento de dados.

As estações de trabalho são, em essência, uma evolução dos computadores pessoais, com capacidade de processamento suficiente para suportar inclusive sistemas multi-usuários, e mostraram-se capazes de desempenhar tarefas de grande magnitude. O uso dessas estações em rede, para compartilhamento de recursos e de carga de processamento, tornou-se possível graças à viabilização dos meios de comunicação de alta velocidade a um custo razoável. Esses ambientes de programação são denominados distribuídos ou de rede.

A possibilidade de integrar vários computadores através de um meio de comunicação com alto desempenho e confiabilidade estimulou, da perspectiva do *software*, a pesquisa de novos modelos de programação, mais adequados para aproveitar a disponibilidade de recursos oferecida pelo modelo distribuído. O trabalho de *software* pode ser basicamente dividido em duas frentes: sistemas operacionais especialmente desenvolvidos para gerenciar uma configuração distribuída de equipamentos; e linguagens de programação adequadas para expressar a natureza intrinsecamente distribuída das novas aplicações.

Uma rede com vários computadores interligados representa um ambiente distribuído do ponto de vista do *hardware*. No que diz respeito ao *software*, um sistema é dito distribuído se apresenta um nível satisfatório de transparência: muito embora esse sistema esteja sendo executado simultaneamente em um conjunto de máquinas, ele oferece seus serviços, aos usuários, como se fosse um sistema centralizado.

Uma grande dificuldade na área de sistemas distribuídos é a necessidade de se integrar máquinas heterogêneas. A diversidade do *hardware* que é produzido é um fato, portanto um sistema não pode confiar em uma arquitetura feita sob medida ou muito específica. Além de integrar diferentes máquinas, o sistema deve ainda esconder essas diferenças dos seus usuários.

Quanto às linguagens de programação, elas tiveram de evoluir para aproveitar melhor a disponibilidade de recursos e a lidar com a complexidade e diversidade das aplicações. Essa diversidade é tamanha que provocou o surgimento de um sem-número de linguagens de programação. Na verdade, surgiram vários paradigmas de programação, cada um derivado de modelos teóricos bem definidos. Dentro de cada um desses paradigmas foram aparecendo várias linguagens, e até mesmo dialetos das linguagens mais bem-sucedidas.

Hoje, para ter possibilidade de sucesso, um sistema deve, ao ser projetado, levar em conta os seguintes aspectos [Dru87b]: interface amigável, dispensando o uso de manual e aumentando a interação com o usuário; portabilidade, tanto entre

máquinas diferentes como entre lançamentos sucessivos de um mesmo fabricante; adaptabilidade, para aumentar sua funcionalidade conforme a evolução do produto; e fácil manutenção, para otimizar o processo de correção de erros detectados por seus usuários.

Os progressos feitos na área de linguagens de programação foram no sentido de aproximar o modelo suportado pela linguagem dos problemas concretos a resolver. As linguagens devem ter um poder de expressão suficiente para descrever os componentes de um problema e os processos envolvidos. O modelo de objetos é considerado a maior evolução nesse sentido desde a programação estruturada. Todavia, ainda não apareceram linguagens ou ferramentas que garantam aos programadores uma base adequada para o desenvolvimento sistemático de programas complexos e confiáveis.

1.3 Organização da tese

Por se tratar de um trabalho na fronteira entre duas áreas (linguagens orientadas a objetos e sistemas distribuídos), é preciso descrever cada uma das partes para depois analisar como a integração entre as duas é possível, interessante de parte a parte e implementável na prática.

O capítulo 2 fala sobre o paradigma ou modelo de objetos. É feita uma descrição do modelo e de suas características fundamentais, e dos aspectos de implementação. Um pequeno *survey* é apresentado, descrevendo algumas linguagens que suportam esse modelo, com comparações entre elas. A seguir apresentamos uma análise crítica do paradigma.

No capítulo 3 é apresentado e discutido o conceito de sistemas distribuídos. São apresentadas linguagens especialmente projetadas para programação em sistemas distribuídos, assim como sistemas distribuídos orientados a objeto. Sistemas já implementados e em uso são descritos e comparados. Finalmente, comenta-se as tentativas de padronização e as tendências de pesquisa na área.

O trabalho do Projeto A_Hand é descrito em detalhe no capítulo 4. A concepção e a filosofia do Projeto, suas realizações e seus objetivos são explicados. As linguagens de programação do ambiente são descritas, assim como as ferramentas que estão relacionadas com esse trabalho. Por fim, este trabalho de mestrado é situado no panorama do Projeto.

O modelo de Objetos Distribuídos é apresentado no capítulo 5. Trata-se do núcleo deste trabalho, e representa a contribuição do autor para o esforço de pesquisa e desenvolvimento do Projeto. O modelo é descrito detalhadamente, assim como sua utilização usando as linguagens de programação do ambiente. Ao final, são descritos e comparados outros trabalhos na área de objetos distribuídos.

No capítulo 6 são apresentadas as conclusões do trabalho, incluindo considerações sobre implementação e extensões futuras.

2 Programação Orientada a Objetos

Destino e vida de leões exige a leonidade que, considerada no tempo, é um leão imortal que se mantém mediante a infinita reposição dos indivíduos, cuja geração e cuja morte formam a força dessa figura imperecível.

Arthur Schopenhauer

Este trabalho de mestrado consiste em uma proposta de extensão da linguagem Cm para torná-la apta à programação distribuída. A linguagem Cm é orientada a objetos, portanto é conveniente uma introdução prévia a esse tema, para extrair elementos necessários à compreensão e análise adequadas da nossa proposta.

Atualmente não há mais dúvida de que a programação orientada a objetos é um tema de grande importância em Ciência da Computação. Esse conceito não é novo: Simula67 [Nyg78] foi a primeira linguagem orientada a objetos, isso em 1967; mas esse estilo de programação só se disseminou depois do aparecimento de Smalltalk [Gol83], em 1980, e de lá para cá tem se tornado cada vez mais popular. A programação orientada a objetos é uma contribuição valiosa para o desenvolvimento da atividade de programar computadores, mas certamente não é uma panacéia; essa opinião está detalhada na seção 2.4 ("Crítica do paradigma de objetos").

Este capítulo se destina a introduzir o paradigma (ou modelo[1]) de objetos, e seus aspectos ou dimensões. É feito um *survey* das linguagens Smalltalk, C++ [Str91] e Eiffel [Mey87], para que haja termos de comparação com o nosso trabalho. Por fim, é feita uma crítica do paradigma.

Nesse capítulo são utilizados irrestritamente muito termos particulares da programação orientada a objetos, como o próprio "objetos", "classes", "herança", "instância", e as conjugações do verbo "instanciar". Isso significa tomar muitas liberdades com a nossa língua, já que muitos desses termos não estão dicionarizados (pelo menos não na acepção em que os utilizamos) ou, pior ainda, são pura e simplesmente tomados de empréstimo da língua inglesa. No apêndice D ("Vocabulário") registramos e comentamos estas e outras "liberdades".

1. Esses dois termos são usados intercambiavelmente em todo o texto.

Finalmente, uma outra observação sobre terminologia: o termo "programação orientada a objetos" aparece nesta dissertação muitas e muitas vezes. Para não cansar o leitor, o termo é substituído pela sigla OOP, do inglês *Object-Oriented Programming*. Essa prática já é mais ou menos difundida na literatura existente, e significa a única concessão (consciente) do autor ao jargão da área.

2.1 Definições

Um *objeto* é um ente composto por um estado e por um comportamento. Esse estado é um conjunto de dados armazenados pelo objeto, e o comportamento é dado por operações que usam esses dados. Dentro da terminologia de OOP, os dados são representados por *variáveis de instância*, e as operações são denominadas *métodos*.

O estado de um objeto é informação encapsulada dentro deste: não pode ser visto de fora do objeto, e só pode ser consultado ou alterado indiretamente, através dos métodos do objeto. Para que um objeto execute um método, ele deve receber uma *mensagem*, à qual reage executando o método solicitado. O conjunto de mensagens às quais um objeto pode responder estabelece a sua *interface*, e corresponde aos métodos que ele oferece.

Uma mensagem é composta de um *seletor* e de uma lista de *parâmetros*. O seletor especifica qual dos métodos do objeto deve ser executado; a lista de parâmetros, que pode ser vazia, é informação adicional exigida pelo método. A recepção de uma mensagem válida faz com que o objeto receptor execute o método pedido e devolva os resultados para o objeto que originou a mensagem.

Objetos com a mesma estrutura e o mesmo comportamento pertencem a uma mesma *classe*. Objetos são gerados a partir de classes; cada objeto gerado é denominado *instância*. Uma classe descreve quais são os dados que devem ser guardados em suas instâncias, qual o conjunto de operações que elas devem oferecer, e como os métodos implementam essas operações. Opcionalmente, uma classe pode declarar dados que são compartilhados por todas as suas instâncias; tais dados são as *variáveis de classe*.

Classes podem ser usadas para gerar novas classes através do mecanismo de *herança*. Por esse mecanismo, uma classe, denominada *subclasse*, herda a estrutura e o comportamento de outra classe, denominada *superclasse*, podendo adicionar novas informações ou métodos, ou redefinir métodos ou dados da superclasse. Quando a herança envolve apenas uma superclasse, temos *herança simples*. É possível que uma classe seja criada herdando de várias superclasses simultaneamente; nesse caso, temos *herança múltipla*.

As linguagens que suportam o conceito de objetos e de herança são denominadas *linguagens orientadas a objetos*; como exemplos, podemos citar Simula, Smalltalk, C++ e Eiffel. As linguagens que suportam objetos mas não suportam herança são ditas *linguagens baseadas em objetos*, como Ada [DoD83] e CLU [Lis77].

2.2 Aspectos do modelo de objetos

Em [Boo94], o chamado *modelo de objetos* é a base conceitual para a análise de tudo que seja relativo a objetos. O modelo é definido por uma série de aspectos, essenciais e acessórios. Os aspectos essenciais estão necessariamente presentes em tudo o que seja relativo a objetos, e são os seguintes:

- abstração
- encapsulamento
- modularidade
- hierarquia

Os aspectos acessórios fazem parte do modelo mas não são fundamentais:

- tipos
- concorrência
- persistência

A compreensão do modelo é essencial para a boa prática de programação orientada usando objetos. Como várias das linguagens ditas orientadas a objetos são baseadas em linguagens de outros paradigmas (e.g., "procedural" ou funcional), um programador que não compreenda adequadamente o modelo de objetos não conseguirá enquadrar os problemas nos termos adequados, e portanto estará se expressando em um modelo menos poderoso que o suportado pela linguagem [Boo94].

2.2.1 Abstração

A *abstração* é um dos mecanismos mais importantes utilizados pela mente humana para compreender o mundo real. Ao analisar um assunto complexo, é natural a tendência de estabelecer semelhanças e diferenças entre entidades, atributos e interações presentes no problema, para que esses elementos possam ser, ainda que informal e incompletamente, descritos, analisados e entendidos. Naturalmente, tal classificação depende do observador, pois ele sabe quais as características que determinam as relações entre os elementos e quais características são irrelevantes para sua análise.

A habilidade de abstrair os elementos essenciais de um problema do mundo real é utilizada quase diariamente pelos programadores. Ao estudar um problema e identificar seus componentes e seus processos, é possível formalizar essas abstrações, usando ferramentas formais como por exemplo linguagens de programação, fluxogramas e diagramas de entidade-relacionamento.

As linguagens de programação são um instrumento formal cuja principal utilidade é permitir a expressão de algoritmos em um nível de abstração mais alto que as linguagens de máquina dos computadores. Os diversos paradigmas — "procedural", funcional, lógico — reúnem linguagens adequadas para abordar determinados tipos de problema.

O paradigma de objetos busca, através de mecanismos como abstração de dados e herança, aproximar as entidades do mundo real de abstrações expressas em uma

linguagem de programação. A idéia básica do uso de objetos é agrupar informações conceitualmente relacionadas juntamente com as operações aplicáveis a essas informações. Esse mecanismo é fácil de ser compreendido e ajuda a modelar os agentes e processos presentes no domínio de um problema.

2.2.2 Encapsulamento

Se a abstração serve para identificar e modelar os componentes de um problema, o *encapsulamento* serve para esconder,[2] daqueles que observam os componentes externamente, os detalhes internos do seu funcionamento. Idealmente, tanto para análise como para construção de um sistema complexo, o comportamento de um elemento do sistema não deve depender da maneira como os demais componentes são estruturados internamente. Tais componentes são descritos por tipos abstratos de dados, uma unidade de encapsulamento na qual o acesso aos dados é feito através de uma interface declarada explicitamente.

Isso leva, na prática, a ver as abstrações de um sistema como sendo compostas de duas partes: uma interface, que descreve aos usuários da abstração como ela pode ser usada; e uma implementação, que especifica concretamente como as operações são executadas. A separação entre uso e implementação, segundo [Lis77], permite que uma abstração seja usada sem que se saiba como é implementada, e que seja implementada sem que se saiba como é usada.

O *princípio de encapsulamento (information hiding)* está descrito em [Par72], e é uma contribuição fundamental para o paradigma, já que propôs a troca da pergunta "como é feito" por "o que é feito e por quem". Dentro do paradigma "procedural", por exemplo, a maneira mais natural de escrever um programa é através da análise *top-down*. Esse método consiste em descrever, no princípio, as funções a serem desempenhadas em termos bastante gerais; e depois, por refinamentos sucessivos, chegar a um nível de detalhamento que possa ser alcançado por uma linguagem de programação. No paradigma de objetos, a atenção se volta para os agentes do problema, e como eles interagem entre si. Ainda segundo [Par72], esse princípio facilita o desenvolvimento de *software* de forma cooperativa e dá maior estabilidade aos programas frente a mudanças de especificação.

A ocultação de informação é inerente à OOP desde que objetos encapsulam dados e operações. O comportamento dos objetos de uma classe pode ser especificado através do *protocolo* da classe, que consiste da descrição das operações públicas implementadas pela classe.

2.2.3 Modularidade

Uma técnica muito usada para lidar com a complexidade de um dado problema é decompô-lo em problemas menores e mais fáceis de serem resolvidos. Há linguagens de programação permitem a decomposição de um programa em módulos,

2. Por causa do verbo "esconder", o termo "ocultação de informação" (do inglês *information hiding*) às vezes é usado no lugar de "encapsulamento".

que podemos definir como trechos completos de código que podem ser compilados e usados em vários contextos. Os módulos são agrupados para formar programas completos; a maneira como os módulos são divididos e implementados afeta diretamente a capacidade de serem utilizados em diferentes contextos.

Naturalmente, dentro de um mesmo módulo é melhor colocar elementos conceitualmente relacionados e que têm interação freqüente entre si. Essa abordagem tende a produzir, na medida do possível, módulos auto-contidos e com poucas dependências de outros módulos. O grau de inter-relação entre os componentes de um módulo é chamado de *coesão*, enquanto que o grau de interação entre módulos é chamado de *acoplamento*.

Linguagens com suporte à programação modular geralmente permitem a separação entre interface e implementação dos módulos, vinculando dessa maneira os conceitos de modularidade e encapsulamento [Boo94]. As linguagens orientadas a objetos vão além, ao usar classes como módulos, já que uma classe agrupa dados e operações conceitualmente relacionados (propiciando alta coesão), e restringe o uso da classe aos componentes da interface (o que tende a diminuir o acoplamento).

2.2.4 Hierarquia

O conceito de *hierarquia* está vinculado ao conceito de abstração. Dado um conjunto de abstrações, é possível que se possa identificar novas abstrações baseando-se em características comuns a um conjunto das abstrações pré-existentes, ou combinando abstrações. A partir dessas novas abstrações, podem ainda surgir outras, e assim por diante, formando uma escala com diversos níveis, representando sucessivos processos de abstração.

Os mecanismos para estabelecer hierarquias de abstrações são bastante usados em áreas como Bancos de Dados e Inteligência Artificial. A cada mecanismo está associado um mecanismo inverso, de modo que podemos agrupá-los em duplas [Tak90]:

- classificação e instanciação
- generalização e especialização
- agregação e decomposição

Classes podem ser definidas de duas maneiras: ou agrupam objetos com estrutura e comportamento semelhantes; ou são como um gabarito[3] usado para criar objetos. No primeiro caso, objetos são agrupados por *classificação*, por compartilharem características determinadas por sua classe. No segundo caso, a criação de objetos é denominada *instanciação*, que significa a obtenção de um novo exemplar (objeto) a partir de um modelo (classe).

Classes são organizadas hierarquicamente através de herança, que define uma relação "*is-a*" entre subclasse e superclasse. Por exemplo: o homem é um primata, e um primata é um mamífero; essas relações podem ser descritas pelas classes

3. Tradução do termo inglês *template*.

Homem, Primata e Mamífero agrupadas em uma cadeia de herança. Uma superclasse representa, em relação às suas subclasses, uma abstração por *generalização*, já que a superclasse fatora semelhanças entre as subclasses, definindo uma abstração mais geral. Inversamente, uma subclasse é obtida por *especialização* de uma superclasse (ou várias superclasses, no caso de herança múltipla), na medida que representa uma abstração com estrutura e comportamento mais específicos e detalhados.

Abstrações complexas podem ser obtidas a partir da combinação de abstrações mais simples. Um compilador, por exemplo, é composto de um analisador léxico, um analisador sintático e um gerador de código; tais partes podem ser vistas como objetos que são usados para compor um objeto mais complexo. A combinação de abstrações é chamada de *agregação*, que estabelece uma relação "*is-part-of*" entre as abstrações. A *decomposição* representa o processo inverso, de obter as abstrações componentes de uma abstração complexa.

2.2.5 Tipos

As linguagens de programação são um mecanismo formal para a descrição de processos do mundo real, e para que o cálculo dos resultados desses processos possa ser efetuado por uma máquina como o computador. No contexto das linguagens, o conceito de tipo é importantíssimo, pois através dele é possível classificar valores de acordo com suas propriedades. Ou, falando mais concretamente, um *tipo* de um objeto especifica o conjunto de valores que esse objeto pode assumir, quais operações podem ser efetuadas sobre esse objeto, e qual a semântica dessas operações.

O processo de examinar um programa para analisar a correção das operações com respeito à compatibilidade de tipos é chamado de *verificação de tipos*. Uma linguagem é denominada *fortemente tipada*[4] se as expressões de um programa válido são consistentes em relação a tipos [Car85]. Eventualmente, pode ser que o tipo de uma expressão não seja conhecido antes da execução do programa (isso é possível graças ao polimorfismo), mas mesmo nesses casos a consistência pode ser assegurada.

O termo *polimorfismo* designa a propriedade de que um determinado nome pode representar valores de diferentes tipos. Em [Car85] o polimorfismo está dividido em dois tipos: universal e *ad hoc*. O tipo universal é representado em linguagens de programação pelo polimorfismo paramétrico[5] e polimorfismo de inclusão (é a forma que toma a redefinição de métodos nas linguagens orientadas a objetos). O tipo *ad hoc* engloba os mecanismos de *overloading* e conversão de tipos.

Em relação a tipos, as linguagens orientadas a objetos variam de extremos como Eiffel, que é fortemente tipada, até Smalltalk, onde a correção das operações é veri-

4. Tradução do termo inglês *strongly typed*. A respeito dessas "traduções", vide comentários no apêndice D ("Vocabulário").

5. O polimorfismo paramétrico também é denominado *generalidade* [Mey86]. Em Ada, por exemplo, funções polimórficas são denominadas funções genéricas.

ficada apenas em tempo de execução. A decisão de adotar uma ou outra abordagem envolve questões como segurança, eficiência e flexibilidade.

2.2.6 Concorrência

Alguns tipos de problema, pela sua própria natureza, são melhor representados por programas atuando separadamente, que cooperam ou competem entre si para atingir um determinado objetivo. Tais problemas podem envolver múltiplos agentes, recursos dispersos geograficamente, ou excesso de carga computacional para um único elemento processador.

O modelo de objetos pode trazer contribuições interessantes para o projeto dessas aplicações. Como esse modelo pretende ser uma aproximação adequada do mundo real, o aspecto da concorrência é importante, já que o mundo real é inerentemente paralelo: num dado momento, vários agentes estão ativos, desempenhando operações por conta própria ou conjuntamente com outros objetos.

Segundo [Weg90], objetos são adequados para expressar concorrência já que sua estrutura representa intuitivamente uma unidade de execução; além do mais, o acoplamento entre objetos é baixo, pois está baseado em troca de mensagens. O estudo de problemas de concorrência baseia-se na noção de *processo*, a unidade de execução usada tradicionalmente.[6] A integração de objetos com concorrência estabelece uma correspondência entre os conceitos de processo e de objeto: tal correspondência implica em analisar, do ponto de vista do modelo de objetos, questões como sincronização e granularidade dos módulos.

2.2.7 Persistência

Durante a execução de programas, os dados estão alocados em memória, e existem por um determinado período de tempo. Esse período de tempo (o chamado "tempo de vida") está, em muitas linguagens, associado ao chamado escopo: dados locais a um procedimento existem durante a execução desse procedimento, e dados globais existem durante toda a execução do programa. Existem ainda dados com tempo de vida muito menor, como resultados temporários de avaliação de expressões e valor de retorno de funções, ou dados com tempo de vida arbitrário, como variáveis criadas e destruídas dinamicamente.

Há informações que são mais duradouras, no sentido de que devem sobreviver à execução dos programas. Para tanto, esses dados devem residir em memória não-volátil, como arquivos em disco. Isso pode ser conseguido em uma linguagem de programação qualquer, se o programador tomar para si a tarefa de guardar os dados que interessam. Todavia isso é muito trabalhoso e propenso a erros, pois é preciso escrever código específico para transformar os objetos em uma representação externa alheia à linguagem, freqüentemente seqüências de caracteres, e vice-versa.

6. Há sistemas em que a unidade de execução é menor que o processo, as chamadas *threads* ou *lightweight process*. Esse assunto é abordado no capítulo 3 ("Sistemas Distribuídos"), seção 3.2 ("Análise de sistemas distribuídos").

Os dados guardados em arquivos não têm tipo,[7] portanto deve-se tomar cuidado com versões dos objetos armazenados.

Em [Weg90] é salientado que o conceito de persistência é relativo; no caso, compara-se a persistência dos dados com a persistência das operações que os utilizam. As funções puras, por exemplo, são operações persistentes sobre dados temporários — seus parâmetros e resultados. Objetos guardam dados e operações com igual persistência. Já transações envolvem dados que persistem a várias operações, e mesmo a *shutdowns* no sistema.

Uma solução mais interessante para dotar os objetos de persistência é o uso de bancos de dados orientados a objetos. Estes são, basicamente, bancos de dados adequados para tratar objetos como dados básicos, o que não pode ser feito convenientemente por outros tipos de bancos de dados. As facilidades oferecidas pelos bancos de dados orientados a objetos incluem suporte a objetos complexos, identidade de objetos, armazenamento de programas e dados, ligação dinâmica e extensibilidade [Har90].

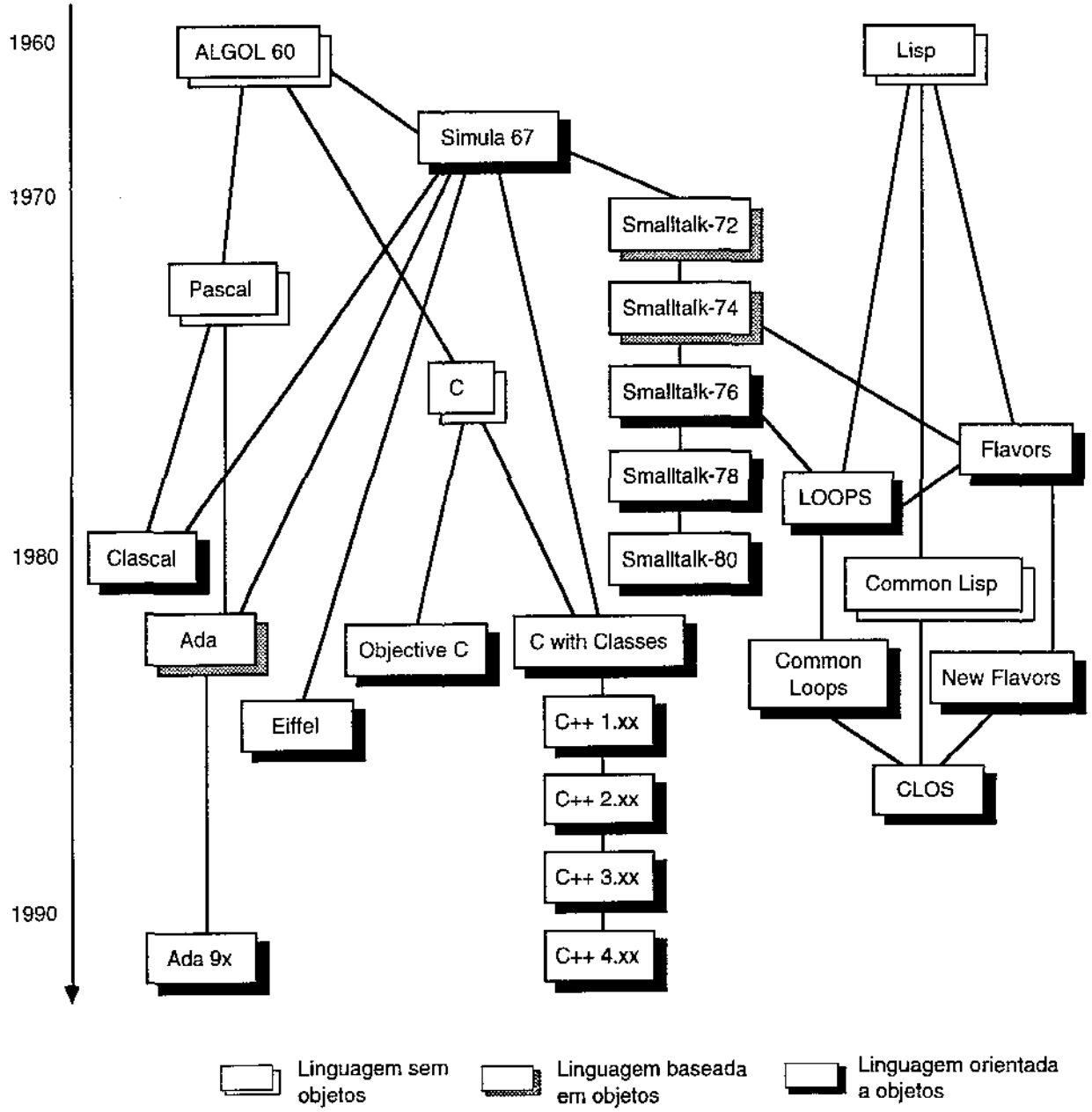
2.3 Linguagens orientadas a objetos

A história das linguagens de programação é, apesar de breve, muito rica, e embora o paradigma de objetos mereça um capítulo à parte, ele deve muito a idéias incorporadas em linguagens fora do paradigma.

O seguinte diagrama, extraído de [Boo94], mostra várias linguagens orientadas a objetos e a sua descendência.

7. Os dados são interpretados por funções que transformam as seqüências de caracteres em objetos.

FIGURA 2-1 Histórico das linguagens orientadas a objetos



A linguagem Algol60 [Nau63] é um importante marco na história das linguagens de programação. A maneira como a linguagem foi definida e apresentada estabeleceu um novo padrão para os trabalhos da área. Foi introduzida, por exemplo, a notação BNF, de tal forma que todas as construções sintáticas da linguagem foram formalmente descritas, juntamente com a semântica associada a essas construções. Pela primeira vez construções como o comando condicional (`if-then-else`) ou

a declaração de um procedimento foram formalizadas dessa maneira. Algol60 teve como sucessora direta Algol68, e influenciou fortemente Pascal e Simula67.

A primeira linguagem orientada a objetos foi Simula67 (ou simplesmente Simula), que introduziu os conceitos de classes, objetos e herança. O fundamental, como lembrado em [Boo94], foi que ela estabeleceu uma disciplina de programação onde as construções da linguagem espelhavam o vocabulário do problema. Embora a importância de Simula seja indiscutível — já que se trata do ancestral comum às linguagens orientadas a objetos — o interesse por ela é mais histórico, pois essa linguagem não teve grande disseminação, e porque o paradigma de objetos somente se consolidou depois do aparecimento de Smalltalk.

A seguir apresentamos um pequeno *survey* das linguagens Smalltalk, C++ e Eiffel, que são bastante representativas do paradigma. Comentamos as características mais importantes dessas linguagens através de exemplos simples. A apresentação desse material é feita no sentido de oferecer subsídios para melhor compreender o nosso ambiente e nossa proposta.

2.3.1 Smalltalk

É a linguagem orientada a objetos por excelência, pois está fundamentada unicamente nesse paradigma. Em Smalltalk todas as informações são consideradas e tratadas como objetos: as estruturas de dados, um arquivo gravado em disco, um *pixel* na tela de um computador, e até mesmo um número inteiro.

A linguagem foi desenvolvida ao longo de uma década, no centro de pesquisa da Xerox em Palo Alto, Califórnia. Foram lançadas versões da linguagem a cada dois anos, começando com Smalltalk–72 e terminando com Smalltalk–80, que é a sua versão definitiva. A linguagem corresponderia à parte de *software* do Dynabook, um ambicioso projeto de computador pessoal.

Segundo [Gol83], Smalltalk é uma visão, pois além de ser uma linguagem é também um poderoso ambiente de programação, composto de uma sofisticada interface homem-máquina, ferramentas como editor, interpretador, *browser* de classes, etc. E não só a linguagem em si teve grande reprecussão, mas também o *look-and-feel* da sua interface influenciou bastante o conceito de interfaces com o usuário, inaugurando um estilo que depois foi seguido por outros ambientes, como a interface do Apple Macintosh e os padrões Open Look e Motif [Boo94].

A terminologia de OOP utilizada hoje foi consagrada por Smalltalk, onde tudo é objeto, e toda interação é baseada em mensagens. Inclusive classes são objetos; a geração de uma instância, por exemplo, é feita em resposta à execução do método new da classe desejada.

Em Smalltalk a herança é simples, i.e., uma subclasse tem apenas uma superclasse. O sistema oferece uma biblioteca de classes cobrindo vários tipos de objetos usados na construção de programas, como estruturas de dados, entrada/saída e objetos da interface com o usuário. Essa biblioteca está organizada, através de herança, como uma hierarquia de classes. Um número inteiro pode ser um objeto da classe *Integer* ou *SmallInteger*, conforme seu tamanho; ambas as classes são subclasses da classe *Number*, que por sua vez é subclasse de *Magnitude*, subclasse de *Object*, a classe

que está no topo da hierarquia. Como tudo é objeto, qualquer classe é, direta ou indiretamente, subclasse de *Object*.

A seguir damos um exemplo de classe Smalltalk, mostrado em [Tak90]:

```
class name
   Stack

superclass
   OrderedCollection

instance methods

   push: anObject
      self add: anObject

   pop
      ^ self size = 0
      thenTrue: [nil]
      elseFalse: [self removeLast]

   isEmpty
      ^ self size = 0

   printOn: aStream
      self do: [ :obj |
         aStream cr; cr.
         obj printOn: aStream ]

   transferTo: aStack
      self isEmpty
         whileFalse: [aStack push: self pop]
```

A classe *Stack* é subclasse de *OrderedCollection*, que está na biblioteca de classes da linguagem. Define cinco métodos: *push*, *pop*, *isEmpty*, *printOn* e *transferTo*. Estes métodos são explicados a seguir.

O método *push* recebe um parâmetro, que é o objeto a ser inserido na pilha. A inserção é feita pelo método *add*, através do envio da mensagem *add: anObject* para a própria pilha, representada pela pseudo-variável *self*. Na especificação do método *push*, o parâmetro *anObject* não está associado a nenhuma classe: de fato, uma mesma pilha pode armazenar objetos de diferentes classes. O método *add* não está definido na classe *Stack*, logo é necessário procurá-lo nas classes que fazem parte da cadeia de herança, começando por *OrderedCollection*. A classe *OrderedCollection* é derivada de *Collection* e implementa uma espécie de *array* dinâmico ordenado pela ordem de inserção e remoção de elementos. O método *add* é herdado de *Collection* e usualmente rebatizado de *addLast*.

A busca de um método pelas superclasses da classe de um objeto termina, na pior das hipóteses, na classe *Object*; se o método finalmente não foi encontrado, isso é um erro, indicando que o objeto recebeu uma mensagem à qual não podia responder. As pseudo-variáveis *self* e *super* servem para referenciar, dentro de um

método, o próprio objeto que executa esse método; a diferença entre as duas está na maneira como é feita a busca do método especificado na mensagem para o próprio objeto. Caso a mensagem seja enviada usando *self*, a busca do método começa na mesma classe onde está o envio da mensagem; se a mensagem foi enviada usando *super*, a busca começa na superclasse.

O método *pop* devolve resultados diferentes dependendo do estado da pilha. Se a pilha está vazia, é devolvido o objeto *nil*, única instância de classe *UndefinedObject*; caso contrário, se a pilha não está vazia, o último objeto inserido será devolvido como resultado da ativação do método *removeLast*.

O método *printOn* imprime o conteúdo da pilha quaisquer que sejam seus componentes. O método *do*, aplicado à variável *self*, causa a iteração sobre todos os elementos da pilha, que são instanciados na variável temporária *obj*. Cada objeto será impresso como resultado da execução do método *printOn*, como definido para objetos da sua classe.

O método *isEmpty* verifica se a pilha está vazia. Esse método ilustra bem o tipo de raciocínio baseado puramente em objetos: o resultado da execução do método *size* é um objeto (número inteiro) que representa o número de elementos da pilha; tal objeto recebe uma mensagem, composta do método = e do parâmetro *0* (objeto representando um número inteiro), sendo que esse método devolve como resultado um objeto da classe *Boolean* (*True* ou *False*). O método =, no caso, é chamado de primitivo, assim como as operações aritméticas; os métodos primitivos são executados diretamente pelo sistema de execução da linguagem.

O método *transferTo* transfere, na ordem inversa, os dados da pilha para outra pilha, passada como parâmetro.

Em Smalltalk as variáveis podem representar objetos de qualquer classe em tempo de execução. Como conseqüência, a determinação de qual método será executado em resposta a uma mensagem depende de qual objeto recebe essa mensagem; determinada a classe desse objeto, a busca do método começa por esse classe, continuando, caso necessário, pelas superclasses ao longo da cadeia de herança, até a classe *Object*. A associação entre uma mensagem e o método a ser executado em resposta é denominada acoplamento mensagem-método, e é absolutamente geral e flexível em Smalltalk.

O ambiente Smalltalk foi portado para uma série de máquinas e sistemas operacionais. Seu uso normalmente depende de *bitmap display* e de um dispositivo de seleção do tipo *mouse*. O código Smalltalk é pré-compilado e o resultado interpretado por uma máquina virtual, composta de um gerenciador de memória, de um interpretador e de um conjunto de funções que implementam os métodos primitivos [Tak90].

2.3.2 C++

A linguagem C++ [Str86, Str91] foi desenvolvida por Bjarne Stroustrup, da AT&T Bell Labs, de onde também surgiram a linguagem C [Ker78] e o sistema Unix [Ker84]. A linguagem é um superconjunto de C, com mecanismos de abstração de

dados elaborados a partir dos conceitos de Simula. O próprio nome C++ indica que a linguagem se propõe a ser uma evolução de C.

A linguagem C não oferece, além da compilação separada, facilidades para a programação modular: um programa é formado por um conjunto de funções, que podem estar distribuídas em diversos arquivos-fonte; tais arquivos são compilados, e o resultado das compilações é ligado para produzir um programa executável. Dentre as funções compiladas, deve existir uma função especial, chamada *main()*, por onde começa a execução do programa. Um programa muito complexo geralmente envolve um grande número de funções, e a manutenção da consistência de seu uso depende de algum tipo de disciplina a ser seguida pelo programador ou pela equipe de programadores.

Já C++ fornece mecanismos de orientação a objetos para possibilitar o desenvolvimento eficiente e sistemático de grandes programas. Os comandos, tipos e operadores de C foram incorporados por C++ sem alterações.[8] Os tipos de C não foram transformados em classes: estas são declaradas explicitamente. A forma de declaração de uma classe é a seguinte:

```
class class_name {
   // declaração dos dados,
   // e dos cabeçalhos dos métodos da classe
};
```

As duas barras ("//") indicam comentário até o fim de linha. Dentro da declaração da classe pode-se especificar quais de seus componentes fazem parte da interface da classe e quais componentes devem ser "escondidos". O corpo dos métodos, em geral, é declarado fora dessa construção, como veremos adiante. Quem vai utilizar uma classe necessita apenas dessa declaração, que pode ser colocada em um arquivo separado (chamado de *header file*), usado através da diretiva `#include`.

Vamos analisar vários exemplos, tomados de [Boo94]. Equipamentos complexos como satélites e naves espaciais podem ser operados remotamente, a partir de um centro de controle. Para que isso seja possível, os dados obtidos por instrumentos e sensores desses equipamentos devem ser enviados para o centro de controle através de telemetria. Tais dados, depois de recebidos, são processados e utilizados para analisar as condições do equipamento e operá-lo adequadamente. Podemos criar uma classe que representa dados genéricos a serem transmitidos:

```
class TelemetryData {

public:
   TelemetryData();
   virtual ~TelemetryData();

   virtual void Transmit();

   Time currentTime() const;
```

8. Aqui estamos nos referindo à linguagem C conforme o padrão ANSI.

```
protected:
   int id;
   Time timeStamp;
};
```

Os componentes da classe estão divididos em duas partes, uma precedida da cláusula `public`, e outra precedida da cláusula `protected`. Os primeiros fazem parte da interface da classe *TelemetryData*; os outros estão encapsulados para os usuários dessa classe, mas estão visíveis para efeito de herança — veremos o efeito disso no próximo exemplo. Além das cláusulas `public` e `protected`, há a cláusula `private`, que encapsula totalmente os componentes, tanto para os usuários de uma classe como para suas eventuais subclasses.

Os métodos com o mesmo nome da classe, *TelemetryData()* e *~TelemetryData()*, são, respectivamente, o construtor e o destrutor da classe; o sinal de '~' sempre precede o destrutor. Esses dois métodos são especiais: o construtor de uma classe é ativado no momento da criação de um objeto, instância dessa classe, e normalmente faz a "inicialização" do objeto. O destrutor é ativado no momento da destruição do objeto, que pode ser de forma explícita através da chamada do destrutor, ou de forma implícita no fim do escopo da variável que representa o objeto. No nosso exemplo, o destrutor é declarado como sendo virtual, indicando que as subclasses de *TelemetryData* eventualmente declararão seus próprios destrutores, e que cada objeto executará o destrutor apropriado, de acordo com sua classe.

O método *transmit()* também é virtual, já que os dados a serem transmitidos dependem, naturalmente, da natureza das informações armazenadas por classes específicas de dados telemétricos (subclasses diretas ou indiretas de `TelemetryData`). O método `currentTime()` permite aos clientes da classe conhecer o valor do dado `timeStamp`, que está encapsulado para esses clientes.

Duas variáveis de instância são declaradas, uma de tipo inteiro (a identificação do dado telemétrico) e outra de tipo `Time` (instante em que o dado foi adquirido). Como visto, os métodos ainda não foram implementados, já que na declaração da classe estão apenas os cabeçalhos dos métodos: para os clientes da classe saber tal informação já é suficiente para usar a classe. Os métodos são implementados da seguinte forma:

```
void TelemetryData::transmit()
{
   // envia a identificação do dado (id)
   // envia o instante de aquisição (timeStamp)
}
```

Um cliente da classe pode usá-la da seguinte maneira:

```
#include "Telemetry.h"
...
TelemetryData telemetry;
...
```

```
telemetry.transmit();
```

Objetos representando dados telemétricos de natureza elétrica podem ser gerados por uma classe resultante da especialização da classe *TelemetryData*.

```
class ElectricalData: public TelemetryData {

public:
   ElectricalData(float v1, float v2,
         float a1, float a2);
   virtual ~ElectricalData();

   virtual void transmit();

   float currentPower() const;

protected:
   float fuelCell1Voltage, fuelCell2Voltage;
   float fuelCell1Amperes, fuelCell2Amperes;
};
```

Essa classe adiciona variáveis de instância (números representando valores de tensão e corrente) e o método *currentPower()*, que usa essas novas informações. A classe *ElectricalData* é herdeira da classe *TelemetryData*: a herança está na 1ª linha da declaração da classe; o qualificador `public` indica que a herança mantém inalterados os atributos de visibilidade dos componentes herdados. É possível, ao ser efetuada a herança, restringir a visibilidade dos dados da superclasse para a subclasse e suas eventuais herdeiras. Para tais casos a herança é feita com qualificadores `protected` ou `private`.

O construtor da nova classe exige quatro parâmetros, que serão atribuídos às novas variáveis de instância no momento da criação de um objeto. Como exemplo de uso desse construtor:

```
ElectricalData electrical( 5.0, -5.0, 3.0, 7.0 );
```

O método *transmit()* pode ser redefinido na classe *ElectricalData*, estendendo a funcionalidade do mesmo método declarado em *TelemetryData*, conforme as novas informações específicas dos dados de natureza elétrica.

```
void ElectricalData::transmit()
{
   TelemetryData::transmit();
   // envia valores de tensão
   // envia valores de corrente
}
```

Nesse exemplo, a execução do método para um objeto da classe *ElectricalData* causa a execução do mesmo método como descrito na classe *TelemetryData*, seguida da parte específica às informações introduzidas pela subclasse.

Em C++ classes definem tipos, e uma hierarquia de classes baseada em herança define uma hierarquia de tipos: uma subclasse, portanto, é um subtipo da sua superclasse. Há regras para a conversão de objetos de acordo com a hierarquia de herança. Vamos exemplificar usando as variáveis *telemetry* e *electrical* declaradas nos exemplos anteriores; a atribuição

```
telemetry = electrical;
```

é legal, já que o tipo da variável *electrical* é subtipo do tipo da variável *telemetry*. Por outro lado, a atribuição

```
electrical = telemetry;
```

é ilegal, pois o tipo da variável *telemetry* é supertipo do tipo da variável *electrical*. Podemos explicar de outra maneira, lembrando que herança equivale a uma relação "*is-a*" da classe herdeira em relação à classe herdada. Portanto, se *ElectricalData* é uma subclasse de *TelemetryData*, dizemos que um objeto da subclasse também é um objeto da superclasse; porém o inverso não é necessariamente verdadeiro.

Dessas situações vêm a necessidade de dispor dos chamados métodos virtuais. Vamos usar a seguinte função como exemplo:

```
void transmitFreshData(TelemetryData& d,
                       const Time& t)
{
   if (d.currentTime() >= t)
      d.transmit();
}
```

Essa é uma função, e não um método, portanto o objeto a ser usado deve ser passado explicitamente como parâmetro. Em virtude do que foi exposto nos parágrafos anteriores sobre supertipos e subtipos, as seguintes chamadas são válidas:

```
transmitFreshData(telemetry, Time(60));
transmitFreshData(electrical, Time(120));
```

Como o método *transmit()* é declarado como virtual, em tempo de execução a chamada de *transmitFreshData()* vai causar a execução desse método conforme a classe do objeto passado como parâmetro.

2.3.3 Eiffel

A linguagem Eiffel [Mey86, Mey87] é contemporânea de C++, foi também projetada dentro do paradigma de objetos, e apresenta muitos aspectos originais. Segundo [Mey87], tais aspectos são em parte derivados da preocupação com questões de engenharia de *software*, pois a linguagem destina-se à produção de grandes sistemas num ambiente industrial.

Eiffel enfatiza principalmente qualidades como reusabilidade — produzir componentes de *software* que podem ser usados em diferentes aplicações — e extensibilidade — no sentido de permitir modificações em sistemas já construídos, sem que sua complexidade seja obstáculo. Outro aspecto relevante é a preocupação com a

correção dos programas escritos em Eiffel, aspecto que será comentado mais adiante.

A unidade básica para a construção de programas em Eiffel é a classe, mas a linguagem utiliza uma terminologia própria para designar outros conceitos da OOP. O conjunto de operações providas por um objeto é composto de *features*,[9] que podem ser de dois tipos: rotinas (funções ou procedimentos, conforme devolvam ou não resultado) e atributos, que correspondem aos dados armazenados nos objetos — as convencionalmente chamadas variáveis de instância. A construção equivalente à chamada de método em C++ foi generalizada na *feature call*, que é sintaticamente idêntica.

Apresentamos a seguir um exemplo de classe em Eiffel, retirado de [Mey87]; os comentários do texto original foram retirados.

```
class ACCOUNT export
   open, deposit, may_withdraw,
   withdraw, balance, owner

feature
   balance: INTEGER;
   minimum_balance: INTEGER is 1000;
   owner: STRING;

open (who:STRING) is
   do
      owner := who
   end;

add (sum:INTEGER) is
   do
      balance := balance + sum
   end;

deposit (sum:INTEGER) is
   require
      sum >= 0
   do
      add (sum)
   ensure
      balance = old balance + sum
   end;

withdraw (sum:INTEGER) is
   require
      sum >= 0;
      sum <= balance - minimum_balance
   do
      add (-sum)
   ensure
```

9. O termo será mantido em inglês, como no original, pois não há tradução satisfatória.

```
      balance = old balance - sum
   end;

may_withdraw (sum:INTEGER) is
   do
      Result := (balance - sum >= minimum_balance)
   end;

Create (initial:INTEGER) is
   require
      initial >= minimum_balance
   do
      balance := initial;
   end;

invariant
   balance >= minimum_balance

end;
```

A classe *ACCOUNT* pode ser usada por outra classe da seguinte forma:

```
acc1: ACCOUNT;
...
acc1.Create(15000);
...
if acc1.may_withdraw(3000) then
   acc1.withdraw(3000);
end;
print(acc1.balance);
```

A variável *acc1*, declarada como sendo do tipo *ACCOUNT*, é denominada entidade. Uma *entidade* é qualquer identificador de um programa que representa um valor em tempo de execução; o termo é geral e designa atributos de classes, parâmetros formais e variáveis locais de rotinas, e a pseudo-variável *Result*, que dentro de uma função representa o valor a ser devolvido [Mey93].

Objetos são criados explicitamente através da *feature Create*, que é usualmente predefinida. No exemplo dado, a classe fornece uma *feature* específica, automaticamente exportada pelo compilador. A entidade *acc1*, antes de representar um objeto da classe *ACCOUNT* pela *feature Create*, é denominada *vazia*.[10]

As *features* exportadas são declaradas no preâmbulo da classe; nesse lugar também são declaradas, através da cláusula `inherit`, as eventuais superclasses das classes formadas por herança — em Eiffel a herança é múltipla. Ao herdar uma classe, a subclasse pode redefinir uma *feature*, mas isso deve constar explicitamente do preâmbulo, através da cláusula `redefine`. Caso a subclasse queira redefinir uma *feature* da superclasse, mas ao mesmo tempo usar o item original, ela deve mudar o

10. Tradução do termo inglês *void*.

nome dessa *feature*, através da cláusula `rename`. Tais procedimentos são necessários porque Eiffel não admite conflito de nomes, uma situação comum na construção de grandes programas na presença de herança múltipla.

Um aspecto muito interessante da construção de *software* que foi incorporado em Eiffel é o chamado *Projeto por Contrato* [Mey93]. De acordo com esse princípio, os objetos componentes de uma aplicação interagem com base em termos pré-estabelecidos, que se assemelham a contratos. Tais termos representam obrigações que uma classe tem para com os seus clientes, e vice-versa; essas obrigações estão explicitamente colocadas nas classes, através de construções sintáticas especiais chamadas asserções. Há três tipos de asserções: *pré-condições* e *pós-condições*, que são relativas a rotinas, e *invariantes de classe* [Mey87].

Uma pré-condição especifica os requisitos necessários à execução de uma rotina, com base nos seus parâmetros e no estado do objeto. A pré-condição é uma obrigação que deve ser cumprida pelo cliente da classe, ao fazer a *feature call*, e representa para o objeto sendo usado uma garantia de proteção contra chamadas inconsistentes. A pré-condição é precedida da palavra reservada `require`; no exemplo dado, a *feature withdraw* exige que o parâmetro seja positivo e que a retirada mantenha o saldo acima do limite mínimo.

A pós-condição de uma *feature* representa a obrigação que o objeto tem, para com o cliente, de produzir um resultado correto em resposta a uma *feature call*, já que o cliente satisfez a pré-condição. A pós-condição é precedida da palavra reservada `ensure`; no exemplo dado, a *feature deposit* garante que, após o depósito, o saldo foi aumentado em quantidade exata à quantia depositada.

A invariante de classe descreve as condições nas quais um objeto dessa classe está em um estado consistente. A invariante deve ser satisfeita imediatamente após a criação do objeto, e depois que toda *feature call* seja completada. Ao final da execução de uma rotina, devem ser satisfeitas tanto a sua pós-condição como a invariante de classe, com o que fica mantida a consistência do objeto. A invariante é especificada através da palavra reservada `invariant`; no exemplo da classe *ACCOUNT*, a invariante determina que o saldo será sempre igual ou maior que o limite mínimo.

As asserções devem ser vistas (e usadas) como um mecanismo para expressar a correção dos programas, a partir de condições que os programadores supõe verdadeiras, e de propriedades que as instâncias de uma classe devem exibir. O uso de asserções é complementado pelo mecanismo de tratamento de exceções, pois uma asserção que não é satisfeita causa a ativação de uma exceção.

O mecanismo de herança preserva o princípio de Projeto por Contrato de duas maneiras. Uma subclasse sempre herda a invariante da superclasse; no caso de herança múltipla, herda todas as invariantes de forma combinada. Além disso, a associação da pré-condição e da pós-condição a uma rotina não é desfeita no caso de redefinição: a subclasse deve prover, para a rotina sendo redefinida, uma pré-condição igual ou mais fraca que a original, e uma pós-condição igual ou mais forte que a original. Através dessa transmissão de asserções, uma classe formada por herança deve honrar as obrigações assumidas pelas suas superclasses.

A implementação de Eiffel usa a geração de código em linguagem C como passo intermediário da compilação [Mey87]. O objetivo dessa abordagem é facilitar a portabilidade do compilador. Eiffel também permite o uso de rotinas escritas em outras linguagens, que depois são ligadas com o resultado da compilação de classes Eiffel. Essas funções são encapsuladas por rotinas Eiffel, com o que são tratadas uniformemente dentro dos programas.

O compilador faz a análise automática das dependências entre classes (classes herdeiras e classes clientes), mantendo versões consistentes das classes alteradas e evitando compilações desnecessárias nos casos em que as alterações não afetam o uso da classe. Em tempo de execução, a criação e destruição de objetos está a cargo de um sistema de suporte da linguagem, responsável pelo gerenciamento de memória e coleta de lixo.[11] O ambiente de programação é complementado por ferramentas de depuração, *profiling* e documentação, e de uma biblioteca de classes que implementam estruturas de dados [Mey87].

2.4 Crítica do paradigma de objetos

Este trabalho de mestrado pretende ser uma contribuição para o campo das Linguagens de Programação, através de uma proposta de extensão da linguagem Cm por mecanismos de programação distribuída. O fato do trabalho estar orientado com o paradigma de objetos significa que acreditamos que esse paradigma é uma base conceitual sólida e coerente para a construção de sistemas e ambientes de programação. Entretanto, há alguns aspectos negativos da OOP, ou pelo menos em algumas de suas implementações, que uma análise equilibrada não pode deixar de destacar.

2.4.1 O paradigma

A primeira convicção que surge, ao buscarmos uma visão crítica do paradigma, é exatamente relacionada a essa visão crítica, ou melhor, à falta dela. Há pesquisadores que estudam e usam o paradigma e não se preocupam em adotar uma postura desapaixonada em relação ao assunto. Dessa forma, ao estudar os trabalhos dessa área, vamos descobrindo um sem-número de vantagens inerentes ao paradigma, sem desvantagens em contrapartida. Isso leva, de forma mais ou menos subliminar, à conclusão de que a OOP é imune a problemas.

O paradigma de objetos, de fato, conseguiu reunir um conjunto de idéias interessantes em um *framework* conciso e elegante. As vantagens relativas à modularidade e reaproveitamento de código vinham aliviar necessidades que se prenunciavam terríveis tempos atrás — a chamada "crise do *software*". O resultado é uma euforia que, apesar do progresso concretamente alcançado pelo paradigma, possivelmente é injustificada. Essa experiência com tecnologias milagrosas não é nova: a mesma coisa aconteceu, em essência, com a Programação Estruturada e a Inteligência Artificial.

11. Tradução do termo inglês *garbage collection*.

Disso tudo, podemos dirigir uma crítica àqueles que transformam o paradigma em uma espécie de religião. Como assinala [Boo94], assim como um homem com um martelo na mão enxerga um prego em tudo que vê, uma pessoa "orientada a objetos" vê todas as coisas do mundo como objetos. Coisas como a eternidade, a cor vermelha e a tristeza certamente não são objetos; ou, usando um exemplo mais concreto [Mad88], essa maneira de pensar faz com que uma expressão simples como

6 + 7

seja interpretada como

6.plus(7)

o que é uma definição nada intuitiva nem tampouco natural de uma operação aritmética.

2.4.2 Eficiência

Os críticos da OOP apresentam também um argumento de ordem prática: programação com objetos representa, via de regra, perda de eficiência. De fato, como a OOP enfatiza a programação modular e reaproveitamento de código, é natural que as classes sejam implementadas sem que se saiba, *a priori*, as aplicações onde serão usadas. Esse aspecto tende a fazer com que as classes sejam muito genéricas e robustas — o que se consegue em detrimento da eficiência.

Ainda sobre eficiência, em [Gut89] é assinalado que linguagens "clássicas" (como C, por exemplo) têm uma semântica muito próxima à estrutura do *hardware* onde são executadas. Isso torna possível otimizar os programas quanto à eficiência, analisando como usar comandos de forma mais eficiente, ou mesmo usando linguagem de máquina se for o caso. As linguagens orientadas a objetos, por outro lado, não têm essa transparência, já que os detalhes de execução estão embutidos em um sistema de execução (máquina virtual, no caso de Smalltalk, ou em bibliotecas de classes e de *run-time*, no caso de C++ e outras).

De fato, o argumento é válido, mas é um tanto especioso por usar o conceito de eficiência de uma maneira muito estreita. Se o poder de expressão de uma linguagem orientada a objetos permite ao programador enquadrar um problema e projetar uma solução mais rapidamente, houve um ganho de eficiência no trabalho do programador. O uso de técnicas como polimorfismo paramétrico e herança facilita o reaproveitamento e a extensão de código existente, reduzindo o esforço de desenvolvimento e de manutenção. Esse último aspecto ganha relevância ao se considerar o grau de correção e adequação aos programas conseguidos. Essa é uma outra acepção de eficiência.

Em [Mad88] é assinalado que a ligação tardia[12] de métodos não implica, necessariamente, em sistemas ineficientes. Em linguagens como C++ e Eiffel, a ligação de métodos é estática por *default*; quando o uso de métodos virtuais for conveniente ou necessário, os programas escritos nessas linguagens terão um *overhead* mínimo,

12. tradução do termo inglês *late binding*.

pois o método correto é buscado em tempo constante. Essa discussão se encerra com a observação [Mad88] de que, se em Smalltalk a busca de um método sempre percorre a hierarquia, às vezes falhando na busca (casos em que é emitida a mensagem de erro *"message not understood"*), isso não se deve ao fato de que Smalltalk é orientada a objetos, mas à maneira como ela foi implementada.

2.4.3 Herança

O mecanismo de herança representa um aspecto fundamental do paradigma, e é uma importante ferramenta conceitual para o efetivo reaproveitamento de código. Entretanto, esse mecanismo é a causa de alguns problemas incômodos.

Em [Sny86] discute-se o potencial conflito entre encapsulamento e herança. Pelo princípio do encapsulamento, os clientes de um módulo podem usá-lo apenas através de sua interface, que é, geralmente, um subconjunto das funções implementadas no módulo. Essa técnica permite separar implementação e uso, já que os clientes enxergam apenas a interface, e portanto o módulo pode ser alterado internamente sem alterar sua funcionalidade visível.

Uma classe pode ser definida herdando a estrutura e o comportamento de classes já desenvolvidas. Por definição, isso inclui a herança das variáveis de instância das superclasses (ou superclasse, no caso de herança simples) e de todas as superclasses ancestrais. Segundo [Tak90], isso viola a premissa de que os dados de um objeto são acessíveis apenas através dos métodos definidos na classe, e portanto Smalltalk, onde uma classe herda toda a estrutura da superclasse, representa um retrocesso em relação a tipos abstratos de dados.

O problema tem duas vertentes, uma para cada sentido da hierarquia de herança. O projetista de uma classe fica preso à sua versão inicial, já que essa classe pode ser usada para definir subclasses e que, ao modificar a superclasse original (por exemplo, suprimindo uma variável de instância), esse projetista pode introduzir erros nas subclasses. Da maneira inversa, uma subclasse pode alterar inadequadamente variáveis de instância que foram declaradas nas superclasses, e assim introduzir erros nos métodos destas.

É sugerido em [Sny86] que uma classe deve oferecer duas diferentes interfaces, uma para as eventuais classes herdeiras e outra para seus clientes, que vão usar essa classe para gerar e usar objetos. Isso faz sentido porque, intuitivamente, a visibilidade que uma classe herdeira tem da classe herdada é maior que a visibilidade de uma classe cliente — que idealmente não deve ter acesso direto às variáveis de instância, por exemplo.

A linguagem C++ implementa essa diferença ao prover um nível de encapsulamento que atua diferentemente para herdeiros e clientes. Além disso, ao herdar uma classe, a subclasse pode restringir (nunca ampliar) as interfaces herdadas, novamente de maneira diferenciada para seus herdeiros e clientes. Quanto à consistência interna dos objetos, que pode ser comprometida pelo manuseio inadequado das variáveis de instância, Eiffel representa um avanço nesse sentido, ao fazer com que uma classe herdeira receba, além da estrutura das superclasses, as suas obrigações, especificadas nas pré e pós-condições dos métodos, e nas invariantes de classe. Dessa forma, o projetista de uma classe pode garantir a consistência interna de suas

instâncias, impondo restrições à implementação de classes herdeiras pela herança das obrigações assumidas pelas classes ancestrais.

Outro aspecto de herança diz respeito à sua implementação. Conforme [Weg90], a questão é a seguinte: o que é herdado, comportamento ou código? A herança de comportamento implica em compartilhamento de código: há apenas uma cópia de uma superclasse, que é utilizada em comum por todas as suas subclasses [Mad88]. Essa abordagem tem impacto na eficiência: quando um objeto recebe uma mensagem, e o método especificado tenha sido herdado por sua classe, a definição do método deve ser procurada pela cadeia de herança. Além disso, segundo [Weg87] o compartilhamento de código inviabiliza linguagens de programação distribuída com herança, já que o compartilhamento é incompatível com a modularidade que deve ser imposta aos objetos distribuídos.[13] A conclusão de [Mad88] sobre o assunto é de que o conceito de herança representa, fundamentalmente, um mecanismo de classificação, e que a maneira como a herança é feita na prática é mera questão de implementação.

Outra questão diz respeito à herança múltipla, que é implementada em algumas linguagens como C++ e Eiffel, ao passo que Smalltalk implementa herança simples. Em [Weg90] é observado que muitas entidades do mundo real podem ser expressas mais naturalmente através de herança múltipla. Por outro lado, em [Mad88] é assinalado que não há um entendimento satisfatório, do ponto de vista teórico, do que seja herança múltipla, e que as implementações que se usam são muito complicadas.

2.4.4 Projeto e desenvolvimento

Além da sua característica principal, de modelar mais facilmente entidades e processos concretos, o paradigma de objetos tem duas características incorporadas nas suas linguagens: desenvolvimento modular e reaproveitamento de código. Mesmo em relação a esses aspectos é possível considerar alguns pontos problemáticos.

Como é o processo de desenvolvimento orientado a objetos? Vamos nos contentar com uma breve receita proposta por [Boo94], que se compõe dos seguintes passos:

- identificar as classes e objetos em um determinado nível de abstração;
- identificar a semântica dessas classes e objetos;
- identificar as relações entre essas classes e objetos;
- implementar essas classes e objetos.

Portanto, o desenvolvimento é baseado em classes, que são projetadas e implementadas por equipes de programadores, as quais interagem para definir as relações entre as classes envolvidas, o que levará à construção de hierarquias de classes. Em princípio a complexidade do problema pode ser adequadamente tratada, já que o

13. É por essa razão que vários autores usam o termo *"Object-based concurrent programming"* ao estudar a programação distribuída e concorrente usando objetos, não só para acomodar linguagens como Ada mas para também destacar esse problema causado pelo mecanismo de herança.

desenvolvimento se faz, quando necessário, em vários níveis de abstração, combinando em níveis as hierarquias correspondentes.

Como estamos falando de grandes sistemas, é inevitável que as hierarquias de classes sejam muito grandes, e essas hierarquias representam o elo entre as equipes de desenvolvimento. Para coordenar esse esforço de desenvolvimento, em [Gut89] observa-se que seria necessário um "czar dos objetos", pela mesma razão que há um administrador de um sistema de banco de dados, por exemplo. Ocorre que as hierarquias de classes não são estáveis, principalmente no começo de um projeto, o que implica na inutilidade desse administrador [Gut89].

Outro aspecto destacado por [Gut89] é a integração de código escrito em várias linguagens. Em muitos sistemas operacionais, por exemplo, pode-se produzir um programa executável combinando funções escritas em linguagens diferentes como C, Pascal e Fortran.[14] Combinar objetos de diferentes linguagens, por outro lado, é assunto difícil de discutir: linguagens como Smalltalk, C++ e CLOS, por exemplo, têm implementações muito diferentes, de modo que essa integração é, na melhor das hipóteses, muito complicada. Os próprios projetistas de linguagens orientadas a objetos reconhecem a necessidade de aproveitar o trabalho feito em outras linguagens mais antigas. Quanto à integração de objetos de diferentes linguagens, tentativas de padronização como CORBA (vide capítulo3, "Sistemas Distribuídos", seção 3.3, "Padronização e tendências") objetivam oferecer uma maneira uniforme de combinar objetos independentemente da linguagem em que foram escritos.

A depuração de programas baseados em objetos é outro problema difícil. Em [Gut89] é reproduzido um comentário atribuído a um grupo de pesquisa em C++ da USENET; diz textualmente: *"It has been discovered that C++ provides a remarkable facility for concealing the trivial details of a program — such as where the bugs are."*[15]

De fato, a dificuldade em depurar este tipo de programa tem duas causas. Primeiro, a análise estática do programa é difícil, já que ao considerar um método de uma classe podemos ser levados a analisar as suas superclasses, tendo necessariamente que conhecer detalhes de como elas foram implementadas, o que contraria violentamente o princípio de encapsulamento. Em segundo lugar, em implementações de linguagens como C++ existe, entre o programa fonte e o programa executável, uma notação intermediária (em C, por exemplo), que pode ser (e é até bom que seja) perfeitamente ilegível. Se a depuração envolver um problema realmente muito difícil (*"a hard bug"*), o programador se verá forçado a saber, por força das circunstâncias, como é o mecanismo de tradução das estruturas do código fonte original para o código intermediário; e isso é muito desagradável. É necessário, portanto, dispor de uma ferramenta de depuração que cuide desses aspectos e cuja interface também seja "orientada a objetos".

14. Não vamos perder tempo discutindo aspectos como modularidade e portabilidade; vamos apenas considerar que é possível fazê-lo.

15. "Descobriu-se que C++ oferece um notável mecanismo para encobrir detalhes triviais de um programa — tais como onde estão os *bugs*".

3 Sistemas Distribuídos

Sinto-me múltiplo. Sou como um quarto com inúmeros espelhos fantásticos que torcem para reflexões falsas uma única anterior realidade que não está em nenhuma e está em todas.

Fernando Pessoa

O ambiente de desenvolvimento proposto pelo Projeto A_Hand objetiva permitir a construção de grandes sistemas de *software* de maneira eficiente e sistemática. Acreditamos que grande parte desses sistemas sejam de natureza distribuída, quer por suas necessidades e características intrínsecas, quer pelas facilidades para esse modelo de programação que são oferecidas por sistemas operacionais da atualidade. Esse tipo de sistema de *software*, composto de um conjunto de programas e recursos compartilhados, e distribuídos por uma rede de máquinas autônomas, recebe a denominação de *aplicação distribuída*. A construção de tais aplicações, feita nesses ambientes e utilizando os modelos e técnicas que lhe são particulares, é o que se chama *programação distribuída*.

A construção dessas aplicações representa uma tarefa de complexidade bem maior que a requerida pelas aplicações não-distribuídas. Em uma rede de computadores não se pode falar, por exemplo, em um "estado da memória", já que as máquinas componentes da rede estão mudando de estado contínua e autonomamente. A comunicação entre as máquinas do ambiente geralmente consome tempo muito grande em comparação com o tempo gasto por uma máquina em um acesso a memória; além do mais, esse tempo de comunicação depende de fatores como a topologia da rede, o volume de tráfego de informações, a distância entre as máquinas comunicantes, etc. O tempo de processamento também é variável, tanto em função da capacidade bruta da máquina como pela sua carga de trabalho em um dado instante.

Todos estes fatores introduzem uma componente de incerteza dentro desses ambientes, o que se reflete no fato de que a ordem em que os eventos se produzem não pode ser prevista, e mesmo a ocorrência desses eventos não pode ser garantida. Os programas que geram e tratam tais eventos devem ser muito robustos e flexíveis, pois a ocorrência de falhas e as possibilidades de combinação dessas pode gerar um sem-número de situações de funcionamento irregular ou de falha irrecuperável (*crash*).

O desenvolvimento e o controle de aplicações distribuídas, portanto, depende fortemente de suporte do ambiente de execução, que deverá ser muito mais elaborado, preciso e robusto que o suporte oferecido por ambientes centralizados ou monoprocessados. Tal suporte inclui aspectos como gerência de recursos, transparência de localização e utilização, comunicação com variáveis níveis de confiabilidade, segurança e serviços. Em um nível de abstração mais alto, há a necessidade de linguagens de programação adequadas, capazes de expressar a natureza distribuída das aplicações. O ambiente de desenvolvimento é complementado por bibliotecas de funções e classes, e ferramentas de documentação, depuração, testes e *profiling*.

Neste capítulo exemplares de sistemas distribuídos são brevemente apresentados, como forma de embasar a discussão que se fará sobre a proposta de objetos distribuídos.

3.1 Definições

Um sistema operacional é um programa que tem basicamente duas funções: controlar os recursos de um computador, quer esses recursos sejam concretos (e.g., memória ou discos), quer sejam abstratos (programas e dados); e oferecer serviços para uso desses recursos por parte dos usuários desse computador. O conjunto desses serviços representa uma abstração dos recursos do computador, oferecendo aos usuários uma "máquina virtual" mais fácil de programar e utilizar.

Os sistemas operacionais evoluíram bastante ao longo da história da computação, para se adaptar às mudanças de *hardware* e *software*. As novas tecnologias de *hardware* trouxeram aumento da velocidade e capacidade de componentes — como processadores, memória, dispositivos de I/O e meios de comunicação —, e o surgimento de novos componentes, como *mouse*, fibra ótica, áudio e vídeo. O seguinte quadro, extraído de [Sha88], ilustra essa evolução.

TABELA 3-1 Recursos de *hardware*

Categoria	Sistemas antigos	Sistemas atuais
Dispositivos periféricos	Cartões perfurados, terminais tty, impressoras de impacto	*Displays* gráficos, *mouse*, impressoras laser
Controle de I/O	Processador central, canais simples	Processadores dedicados e específicos para I/O
Memória secundária	Fitas, *drums*, discos com pouca capacidade	Discos com grande capacidade
Memória principal	Memória com pouca capacidade	Memória virtual e memórias com grande capacidade

TABELA 3-1 Recursos de *hardware*

Categoria	Sistemas antigos	Sistemas atuais
Processador	Processador único	Múltiplos processadores
Sistema	Centralizado	Centralizado: múltiplos processadores com memória compartilhada Distribuído: redes locais e *gateways*.

Os avanços de *software* modificaram o tipo de serviços oferecidos aos projetistas e usuários para o suporte ao desenvolvimento e execução de novas aplicações. Nessa parte podemos citar os protocolos de comunicação, os sistemas de interfaces gráficas para o usuário, aplicativos para desenvolvimento cooperativo, serviço de *mail* e transferência de arquivos, e outros.

3.1.1 Redes de computadores

Uma rede de computadores é um conjunto de computadores autônomos, interligados por um meio de comunicação para a troca de informações. Dentro de uma rede, quaisquer dois componentes podem trocar dados, diretamente ou através de outros componentes. Conforme [Tan81], o fato de que os componentes da rede sejam autônomos é muito importante, já que isso exclui ambientes baseados em relações mestre/escravo: um computador ligado a um conjunto de equipamentos auxiliares, como terminais e impressoras, não é uma rede.

A interligação entre computadores é feita, geralmente, com a intenção de compartilhar dados e carga de processamento, em situações onde os processadores e os dados estão geograficamente separados. O modelo de ambiente de processamento de informações baseado em rede vem se disseminando rapidamente nos últimos anos. Essa disseminação se faz em detrimento dos ambientes ditos centralizados, compostos por um computador com grande poder de processamento (o chamado *mainframe*), gerenciando equipamentos acessórios para entrada e saída de dados (terminais, leitora de cartões, impressoras) e armazenamento de informações (discos e fitas). Segundo [Tan81], o modelo centralizado tem dois defeitos fundamentais: um único computador é responsável por todo o processamento; e os usuários devem "trazer" o trabalho até o computador, quando o mais adequado seria "levar" o computador até onde o trabalho está.

As redes podem ser divididas de acordo com vários fatores, um dos quais é a distância física que separa as máquinas componentes. Essa distância influi diretamente na velocidade e na confiabilidade da transmissão dos dados. O nosso interesse está voltado para as chamadas redes locais, onde a distância entre duas máquinas pode variar (em termos de ordem de grandeza) de 1m até 100m [Tan81]. Redes locais que usam a fibra ótica como meio de transmissão, por exemplo, têm velocidades de

comunicação bastante altas (aproximadamente 10 Mbits/s, como no caso da Ethernet).

3.1.2 Sistemas baseados em rede

O ambiente de processamento de informações preponderante nas universidades, atualmente, são as chamadas redes UNIX. Essas redes são redes locais compostas por uma série de estações de trabalho (ou mesmo *mainframes*) que utilizam UNIX como sistema operacional. Esses componentes estão geralmente organizados em uma rede em forma de barramento, possibilitando que o meio de comunicação permaneça operacional no caso de *crash* em máquinas da rede.

Essas redes disseminaram-se rapidamente na década de 80. O fator determinante para isso foi a viabilização econômica dos meios de transmissão de alto desempenho. Paralelamente, as estações de trabalho tornaram-se mais rápidas, poderosas e versáteis, oferecendo uma fração da capacidade de um *mainframe* por um preço relativo muito menor. Outra qualidade das redes de estações de trabalho é a capacidade de extensão (ou redução) gradativa do sistema, através da adição (ou subtração) de elementos processadores coforme as necessidades. Finalmente, as redes possuem uma capacidade potencial de tolerância a falhas, que pode ser implementada por protocolos e serviços específicos.

Em uma rede UNIX típica, um usuário inicia uma sessão (*login*) em uma das máquinas da rede, mas pode utilizar outras máquinas, com a ajuda de utilitários como `rlogin` ou `rsh`. O sistema de arquivos NFS [Sun88], também amplamente disseminado, permite que todas as máquinas da rede compartilhem um sistema de arquivos global, cujos dados estão fisicamente armazenados em diferentes máquinas da rede.

3.1.3 Sistemas Operacionais Distribuídos

Ao introduzir os conceitos essenciais deste capítulo, é necessário tomar algumas precauções quanto à terminologia. Usaremos o termo "Sistema Distribuído" para representar uma nova classe de sistemas operacionais, e não aplicações de características distribuídas. Um sistema gerenciador de bancos de dados, por exemplo, pode abranger um conjunto de recursos geograficamente distribuídos, mas não é um sistema distribuído na nossa acepção do termo. A melhor maneira de separar esses conceitos é dizer que os sistemas (operacionais) distribuídos são a base ideal para o desenvolvimento e execução de aplicações distribuídas.

O surgimento desses sistemas operacionais distribuídos foi possível graças aos avanços da tecnologia de *hardware*, que se refletiu em crescentes facilidades para a construção e utilização de redes de estações de trabalho a baixo custo. Conforme assinalado em [Che88], a premissa básica dos trabalhos com sistemas distribuídos é a de que um conjunto de estações de trabalho interligadas, quando utilizado corretamente, representa, em relação ao ambiente de programação baseado nos *mainframes*, uma alternativa mais poderosa, extensível e econômica.

A disponibilidade de recursos e as facilidades para interligação continuam crescendo. Uma tendência nítida, conforme [Tan85], é o aumento da capacidade de processamento disponível para os usuários, principalmente com o modelo chamado

processor pool, onde um grande número de processadores é dinamicamente alocado para uso, conforme as necessidades das aplicações. Para um aproveitamento ótimo dessas facilidades, os sistemas operacionais devem gerenciar os recursos de forma automática e transparente, para que os programadores, durante o desenvolvimento de aplicações complexas, não desperdicem tempo lidando com detalhes do ambiente e uso dos recursos, e possam se concentrar na pesquisa de melhores soluções para os problemas essenciais das aplicações. Nesse panorama estão colocados os sistemas distribuídos.

Na definição de [Tan85], um *sistema distribuído* é um sistema que se apresenta aos usuários como um sistema operacional centralizado, mas que, na realidade, tem suas funções executadas por um conjunto de máquinas independentes. Em tais sistemas, a localização dos recursos (dispositivos periféricos, arquivos, etc) é irrelevante para o usuário, que tem a "ilusão" de usar apenas uma máquina que concentra todos os recursos da rede. Isso é possível a partir da idéia de que os recursos são identificados globalmente, independentemente da sua localização, e dessa maneira são utilizados pelas aplicações.

Em um sistema distribuído, o conceito fundamental é *transparência* [Tan85]: isso significa, para o usuário, não saber qual máquina está utilizando, em quais máquinas os arquivos estão fisicamente armazenados, onde os programas estão sendo executados, e assim por diante. Com base nesse princípio, podemos concluir que uma rede UNIX não é um sistema distribuído, pois um usuário executa seus programas na máquina onde iniciou a sessão (*login*), e utiliza outras máquinas para executar programas sempre de forma explícita.

Há muitos exemplos de sistemas distribuídos, na forma de propostas, protótipos ou sistemas já implementados e em uso. Como exemplo podemos citar Amoeba [Mul90], Argus [Lis88], Chorus [Arm89], Emerald [Bla86], Mach [You87], e V[Che88]. Alguns desses sistemas são muito diferentes entre si, refletindo diferentes concepções de projeto, implementação e uso.

Os sistemas distribuídos são bem mais complexos que os sistemas meramente multi-usuários. A integração de uma rede de máquinas autônomas oferecendo um sistema virtualmente centralizado significa integrar, de uma forma coerente, aspectos de comunicação, transparência no uso de recursos, segurança e tratamento de falhas de *hardware*. O atendimento a toda essa série de exigências tem forte impacto no desempenho, sempre citado como problema sério dos sistemas distribuídos [Che88, Hor89, Mul90].

3.2 Análise de sistemas distribuídos

Muitos sistemas distribuídos foram propostos e implementados, desde que surgiu essa linha de pesquisa, aproximadamente no início da década de 80. Não se pode dizer que algum desses sistemas tenha se tornado um exemplar característico ou dominante; entretanto, no conjunto eles estabeleceram uma base conceitual sólida para o desenvolvimento futuro da área. Podemos analisar e comparar diferentes sistemas em relação aos aspectos de gerenciamento de recursos, mecanismos de comunicação, segurança, tolerância a falhas e implementação.

Nesta seção descreveremos e compararemos rapidamente os sistemas Argus, Chorus e Emerald, escolhidos por tratarem-se de exemplos representativos e com repercussão nos meios de pesquisa. Em [Chi91] esses três sistemas, além de outros, são denominados "Sistemas de programação distribuídos baseados em objetos" (DOBPS), por ser resultado da integração dos conceitos de linguagens baseadas no modelo de objetos e sistemas distribuídos. Voltaremos a falar desses sistemas no capítulo 5 ("Objetos Distribuídos em Cm"), seção 5.1 ("Sistemas Distribuídos + Objetos").

3.2.1 Argus

Argus [Lis82, Lis87, Lis88] é uma linguagem de programação e um sistema, integrados para oferecer um ambiente de programação distribuída com grande ênfase na parte de tolerância a falhas. A linguagem, uma extensão de CLU [Lis77], oferece mecanismos para a especificação de módulos que podem ser usados concorrentemente e sem perda de informação no caso de falhas de *hardware*. O sistema foi projetado para atender aos requisitos impostos por programas que manipulam informações distribuídas e que são mantidos em execução por muito tempo, onde é importante a consistência das informações e a capacidade de reconfiguração dinâmica.

Um programa em Argus é executado em uma rede de máquinas ligadas em rede, onde cada máquina é chamada de nó. As redes podem ser locais ou de longa distância; pode haver perda de mensagens ou recepção de mensagens fora de ordem, e problemas no meio de transmissão que podem impedir a comunicação entre nós. Os nós da rede também podem falhar, quando deixam de mandar mensagens para os outros nós.

Dois conceitos introduzidos por Argus são a base para a construção de aplicações distribuídas e tolerantes a falhas: guardiões e ações. Um guardião é um objeto que implementa operações executadas em resposta a requisições enviadas de qualquer ponto da rede. Uma ação (também denominada transação atômica) é uma seqüência de operações que sempre termina com um resultado bem definido, sucesso ou falha. Esses dois conceitos, combinados, suprem as necessidades de autonomia, distribuição, concorrência e consistência, exigidas pela classe de aplicações cuja implementação Argus busca facilitar.

Os guardiões são entidades autônomas e ativas, e controlam o acesso a recursos através de interfaces que são conhecidas e utilizadas por outros guardiões. Uma interface é composta por um conjunto de operações chamadas *handlers*, sendo que a chamada de *handlers* é a única forma de comunicação entre guardiões. Durante uma chamada tanto os parâmetros como o valor de resultado são manipulados através de passagem por valor.

Um guardião reside em um dado nó da rede, no sentido de que é executado e tem seus recursos armazenados nesse nó; eventualmente, ele pode mudar de nó, migrando para outro ponto da rede. Em um nó podem residir vários guardiões simultaneamente. Um programa distribuído em Argus é, portanto, composto por vários guardiões, que residem em diferentes nós da rede.

Em relação a falhas no nó onde reside, um guardião é resiliente: ele sobrevive a essas falhas e pode voltar a executar quando o ambiente estiver normalizado. Os objetos que estão armazenados em um guardião podem ser de dois tipos, estáveis e voláteis. Quando o nó falha, os objetos voláteis são perdidos, mas os objetos estáveis estão guardados em memória secundária, de onde podem ser recuperados quando o guardião se recuperar da falha. Isso garante a consistência dos dados armazenados.

Os guardiões possuem concorrência interna, provendo mecanismos para o acesso disciplinado a recursos utilizados em comum. Num dado momento, várias *threads* podem ser executadas em um guardião: algumas delas ativadas na criação deste (ou durante o processo de recuperação de uma falha) e outras ativadas em resposta à chamada de um *handler*. Os potenciais conflitos causados pelo acesso concorrente a recursos comuns são resolvidos pelo mecanismo de ações.

Uma característica importante de Argus é que a atomicidade é um conceito incorporado à linguagem, com o objetivo de garantir a consistência e disponibilidade dos dados manipulados pelas aplicações. Os guardiões são a unidade de distribuição e encapsulamento da linguagem, ao passo que as ações atômicas são usadas para implementar a atomicidade das operações.

Uma ação atômica (ou simplesmente ação) é uma operação envolvendo objetos que sempre termina completamente, com sucesso ou com falha. Uma ação é indivisível, no sentido de que, durante sua execução, não há possibilidade de que seus resultados sejam comprometidos por interferência de outra ação. Além disso, uma ação pode ser refeita, caso não possa ser completada: em caso de falha, as alterações feitas nos objetos são canceladas, e a ação termina sem efeitos colaterais. Quando a ação termina com sucesso, as alterações feitas são consolidadas e todos os objetos envolvidos estão no estado desejado.

Ações são sincronizadas através de objetos atômicos, objetos cujas funções de manipulação são especiais, no sentido de que essas operações são usadas para sincronizar as ações que as utilizam, e que podem ser canceladas caso a ação termine com falha. Toda operação sobre um objeto atômico é classificada como de leitura ou de escrita, conforme altere ou não o objeto. Cada operação, de leitura ou de escrita, será executada apenas após obter, sobre o objeto, um *lock* correspondente: vários leitores podem ter acesso simultâneo ao mesmo objeto, e um escritor exclui qualquer outra operação concorrente. Alterações feitas nesses objetos são consolidadas apenas se a ação terminar com sucesso.

Ações podem ser encaixadas, com o objetivo de oferecer concorrência dentro de uma ação, e permitir que "sub-ações" sejam executadas sem que uma falha em uma delas cause automaticamente uma falha na "super-ação". Alterações feitas em objetos atômicos dentro das "sub-ações" só são completadas quando a "ação principal" (*top action*) terminar com sucesso.

O sistema Argus foi implementado em uma rede Ethernet de MicroVAX-IIs operando sobre UNIX. Cada guardião é mapeado em um processo UNIX, sendo que o escalonamento de *threads* dentro de um guardião é feito pelo sistema Argus. Conforme [Lis87], evitou-se ao máximo fazer alterações no núcleo do UNIX, o que

significou abrir mão de um controle estrito sobre funções críticas (gerenciamento de memória, e.g), o que teve reflexos na implementação e desempenho do sistema.

3.2.2 Chorus

O sistema Chorus [Arm89, Her89] foi desenvolvido com o objetivo de suportar uma nova geração de sistemas operacionais, com ênfase nos aspectos de distribuição, modularidade e escalabilidade. Esse sistema é um exemplo do modelo de *microkernel*, onde um pequeno núcleo implementa uma máquina virtual, com funções de processamento distribuído, gerenciamento de memória e comunicação no nível mais baixo possível. Acima desse núcleo, são implementados servidores (chamados subsistemas) especializados, que implementam os demais serviços de um sistema operacional.

O ambiente do sistema Chorus é uma rede de máquinas (denominadas *sites*) independentes, que podem ser estações de trabalho e servidores, estações *diskless* ou sistemas dedicados. Cada um desses *sites* engloba um conjunto de recursos: um ou mais processadores, memória e dispositivos de I/O. Em cada um desses *sites* existe um núcleo Chorus, que se comunica com os demais núcleos da rede para troca de informações e serviços básicos.

O núcleo Chorus é composto de um supervisor e de módulos de tempo-real, memória virtual e comunicação. O supervisor atua em contato direto com o *hardware*, portanto é dependente de máquina; sua função é tratar eventos como interrupções e *traps*. O módulo de memória virtual controla a memória do *site* onde está sendo executado, o módulo de tempo-real provê multiprogramação através de *threads*, e o módulo de comunicação provê comunicação transparente entre os núcleos Chorus da rede.

O núcleo Chorus fornece cinco abstrações básicas para os módulos servidores: atores, *threads*, mensagens, portas e grupos.

O ator é a unidade lógica de distribuição e de encapsulamento de recursos. Para cada ator é definido um espaço de memória protegido, onde podem ser executadas *threads* de maneira concorrente. Vários atores podem residir em um mesmo *site*.

A *thread* é a unidade de execução do Chorus, e é composta de informações relativas a contexto de processador (registradores, *program counter*, *stack pointer*, etc) e uma pequena área de dados local. Uma *thread* é criada dentro do contexto de um ator, e compartilha com as demais *threads* desse ator os recursos deste.

As *threads* podem se sincronizar e trocar informações através de mensagens. A comunicação entre *threads* é absolutamente transparente, com interface uniforme, independentemente do fato de as *threads* estarem dentro do mesmo ator, em atores diferentes de um mesmo *site*, ou em atores em diferentes *sites*. Uma mensagem é uma seqüência de *bytes* sem estrutura associada. As mensagens são enviadas não a *threads* diretamente, mas a portas, que são filas de mensagens. Uma porta é associada a um único ator, permitindo que as mensagens enviadas a essa porta sejam consumidas pelas *threads* do ator associado à porta. Uma porta pode migrar de um ator para outro, possibilitando reconfiguração dinâmica do sistema.

O grupo de portas é o mecanismo para comunicação *multicast*. Um grupo representa um conjunto de portas, que recebem transparentemente mensagens enviadas ao grupo. As portas podem ser de diferentes atores, em diferentes *sites*, e são inseridas e retiradas do grupo dinamicamente. Essa facilidade é importante para o oferecimento de serviços mais persistentes, no caso de falhas isoladas em alguns atores.

Os objetos manipulados pelo núcleo Chorus são identificados através de UIs (*Unique Identifiers*), que são únicos na rede. Através desses identificadores o núcleo pode fazer acesso transparente aos recursos distribuídos. Além do UI, os objetos podem ter outras informações associadas, como as *capabilities*, usadas para proteger esses objetos de acessos indevidos ou não autorizados.

A filosofia representada pelo *microkernel* é oferecer um conjunto de serviços básicos a partir do qual diferentes sistemas operacionais podem ser construídos. Isso é possível porque um *microkernel* típico oferece serviços muito primitivos, e a interface para as aplicações deve ser oferecida por subsistemas dedicados. Além do mais, dada a natureza distribuída do *microkernel*, os sistemas construídos com base nesses núcleos podem usufruir de vantagens como comunicação transparente, programação em tempo real e concorrência através de *multithreading*.

O sistema Chorus comprovou essa concepção com a implementação de um sistema UNIX através de subsistemas. Essa implementação é o Chorus/MiX, que adiciona a uma interface UNIX padrão facilidades de programação em tempo real e *multithreading* [Her89]. Os diferentes serviços do UNIX são oferecidos através de servidores dedicados: gerenciador de processos, de arquivos, de dispositivos, de *pipe* e de *sockets*. Um processo UNIX é implementado como um ator, e controlado pelo gerenciador de processos.

Segundo [Arm89], essa abordagem para a construção de sistemas operacionais tem reflexos bastante positivos na modularidade e na manutenção desses sistemas. Um *X-Terminal*, por exemplo, pode utilizar uma configuração do UNIX sem os gerenciadores de *pipe* e de arquivos, pois esses não são necessários para ele. Dessa forma, o sistema pode ser usado sempre com a funcionalidade realmente necessária, ao contrário das implementações atuais, onde mais e mais funções foram, ao longo dos anos, adicionadas a um núcleo monolítico, com muitas interdependências entre as suas partes. Um sistema UNIX semelhante ao Chorus/MiX é o OSF/1 [Var94], construído sobre o *microkernel* Mach [You87].

3.2.3 Emerald

Emerald [Bla86, Raj91] é um sistema e uma linguagem, desenvolvidos com base em objetos, utilizados para a construção de aplicações distribuídas. Tem um modelo uniforme de objetos, que são usados através de uma interface comum independentemente de sua localização e granularidade. O ambiente para execução do sistema é uma rede de máquinas autônomas comunicando-se através de trocas de mensagens. Uma aplicação em Emerald é composta por um conjunto de objetos cooperantes, que residem em nós da rede. Um nó é uma abstração de uma máquina, podendo existir vários deles em uma mesma máquina.

O conceito de objetos é central em Emerald: todas as entidades são objetos, desde números inteiros até arquivos ou diretórios. Um objeto é composto por um nome, um conjunto de dados armazenados, um conjunto de operações (privadas ou exportadas) e um processo opcional.

O nome do objeto é uma identificação única na rede, permitindo que esse objeto seja utilizado independentemente da sua localização. Os objetos podem migrar de um ponto a outro da rede, conforme necessário, com pleno suporte do sistema. Desse modo, para a construção de aplicações, todo o sistema distribuído pode ser considerado como uma única máquina. Caso o objeto tenha um processo, ele é ativo, e esse processo é executado em paralelo com as ativações de suas operações; caso contrário, ele é passivo, entrando em execução apenas para atender a uma ativação de uma operação. Pode haver concorrência dentro de um objeto, pela execução simultânea de suas operações; o acesso concorrente aos dados do objeto é controlado através de monitores [Hoa74, Raj91].

Em Emerald não há classes: objetos novos são obtidos a partir de *construtores de objetos*, expressões cujo resultado é um objeto. O construtor especifica quais os dados armazenados no objeto, as operações oferecidas (nome, tipo dos parâmetros, tipo do resultado e implementação) e atributos do objeto (instruções para o compilador e sistema de execução). Dessa forma, tais construtores cumprem funções das classes em outras linguagens: geram novos objetos, descrevem a representação dos objetos e a implementação das operações oferecidas [Raj91].

Emerald é fortemente tipada, sendo que a verificação de tipos está baseada nos conceitos de *tipos abstratos* e *conformidade de tipos*. Um objeto pode implementar as operações previstas por diferentes tipos abstratos, e um tipo abstrato pode ser implementado por diferentes objetos [Bla86]. Isso faz com que objetos possam ser utilizados no lugar de outros objetos através de interfaces compartilhadas (ao contrário do compartilhamento de código baseado em herança, que não existe em Emerald). A verificação de tipos é normalmente feita estaticamente, pelo compilador, mas se necessário pode ser postergada até a fase de execução, desde que tais casos sejam explicitamente especificados pelo programador.

O maior objetivo do modelo de objetos de Emerald é a completa uniformidade no tratamento de objetos, que apresentam a mesma semântica independentemente da sua implementação pelo sistema. Essa característica da linguagem permite que os objetos sejam usados em diferentes situações sem necessidade de modificação de código, cabendo ao compilador dar suporte transparente às aplicações nesse sentido. As diferentes maneiras de implementação dos objetos são denominadas *estilos de implementação*, que são os seguintes:

- objetos globais: são a unidade de distribuição das aplicações, podem migrar de um nó para outro e são visíveis por todo o sistema, podendo ser utilizados por outros objetos;
- objetos locais: existem dentro do contexto de outro objeto, e são invisíveis fora deste. Mudam de nó apenas quando acompanhando a migração do objeto que os contém;
- objetos diretos: são objetos locais, com a diferença de que seus dados e operações são incorporados ao objeto que os declara. É o caso de objetos muito pequenos, como dados de tipos primitivos [Bla86].

O compilador decide qual o estilo de implementação é o mais adequado para um determinado objeto com base nas informações disponíveis na fase de compilação. Tal escolha leva em conta as possibilidades para o uso desse objeto, eficiência e custo de invocação de operações.

A interação entre objetos que fazem parte de uma aplicação em Emerald é independente da localização desses objetos. Cabe ao sistema de suporte à execução determinar, a partir de uma invocação, onde está o objeto que está sendo invocado e completar a execução da operação solicitada. Entretanto, uma aplicação pode decidir, de acordo com as suas necessidades, qual a melhor configuração dos objetos dentro da rede, visando vantagens como: proximidade de algum recurso, divisão de carga de processamento, interação intensa entre objetos e outras. Portanto, a linguagem oferece operações relativas à localização dos objetos:

- *locate*, para determinar onde está um objeto;
- *fix*, para manter um objeto onde ele está;
- *unfix*, para liberar um objeto para migração; e
- *move*, para efetivamente mover um objeto entre os nós da rede.

Um objeto também pode migrar se for passado como parâmetro de uma invocação. Isso porque o objeto que está sendo chamado, ao usar os objetos passados como parâmetro, na realidade está usando referências a esses objetos, pois eles não são passados por valor. Assim, ao fazer, por sua vez, invocações a esses objetos, o objeto executando a invocação pode fazer muitas outras invocações remotas, o que poderia ser evitado se todos esses objetos estivessem no mesmo nó [Bla86]. A idéia, portanto, é "levar" os parâmetros para junto do objeto chamado, de forma a evitar um aumento muito grande no número de invocações remotas. Essa migração, entretanto, depende de alguns fatores, como o tamanho do objeto parâmetro, o número de chamadas que ele pode estar atendendo nesse momento (objetos podem migrar mesmo enquanto respondem a chamadas) e como ele será usado no objeto chamado. Depois de migrar, o objeto parâmetro pode permanecer no seu novo nó ou retornar ao nó onde estava — esses mecanismos de passagem de parâmetros são denominados *call-by-move* e *call-by-visit*, respectivamente.

O sistema está implementado em uma rede de MicroVAXIIs, operando sob UNIX e conectados via Ethernet. Dentro de um objeto, os diferentes fluxos de controle são implementados como *lightweight processes*, permitindo baixo custo de escalonamento e troca de contexto pelo sistema de suporte à execução. Todos os objetos e processos de um nó compartilham, junto com o sistema de suporte, o mesmo espaço de endereçamento. O compilador gera código nativo da máquina, que é carregado em um "mundo de objetos Emerald", que é o ambiente de execução das aplicações [Bla86]. Outra característica do sistema é uma forte integração entre o compilador e o sistema de execução.

3.3 Padronização

Com a disseminação de sistemas de programação distribuída surgiu, como conseqüência, a preocupação de integrar aplicações baseadas nos diferentes sistemas. O desenvolvimento de aplicações nesses novos ambientes tem expectativas crescentes

de interoperabilidade, que envolva arquiteturas, redes e/ou linguagens e modelos de programação heterogêneos.

As tentativas de padronização na área de sistemas distribuídos objetivam integrar as diferentes abordagens e implementações através de interfaces que devem ser respeitadas por sistemas que se pretendem compatíveis. Tais interfaces são utilizadas para acesso a serviços mais elaborados como servidor de nomes, RPC e sistemas de arquivos distribuídos.

Há também um esforço de padronização em um nível de abstração mais alto, envolvendo as linguagens de programação. Nesse ponto, o modelo de programação orientada a objetos deve ser confirmado como o modelo universal, já que objetos são uma unidade natural para a distribuição de dados e de funcionalidade. Diferentes aplicações poderão compartilhar serviços e informações através de objetos usados em comum. As aplicações devem basear-se em mecanismos oferecidos pelo ambiente de programação para localizar objetos e solicitar a execução de operações.

3.3.1 Sistemas distribuídos abertos

As *tecnologias abertas* objetivam a integração, dentro de um mesmo sistema ou produto, de componentes de diferentes fabricantes. Isso é possível porque as partes relacionadas são compatíveis e portanto podem ser utilizadas em conjunto. Essa compatibilidade é conseguida com a adoção, por parte dos fabricantes, de padrões estabelecidos, que determinam as características que devem ser respeitadas pelos componentes do sistema. Tais padrões são abertos, i.e., são conhecidos publicamente, permitindo a quem quer que seja fabricar componentes ou produtos de acordo com esses padrões.

Em contrapartida, as *tecnologias proprietárias* são geradas e utilizadas por um único fabricante, sem preocupação com a cooperação de outros. Quando essas tecnologias são estabelecidas, podem ser repassadas a outros fabricantes, que normalmente pagam *royalties* pela sua exploração.

Na computação, durante um bom tempo houve certa competição entre tecnologias abertas e proprietárias, com relação a componentes eletrônicos (processadores, barramentos), redes (protocolos, serviços) e *software* (sistemas operacionais, bancos de dados, linguagens). Atualmente, essa questão não faz muito sentido, porque o mundo está cada vez mais integrado, e o compartilhamento de dados e serviços é, a um só tempo, um desejo e uma necessidade. Uma rede como a Internet, por exemplo, realiza diariamente milhares de transferências de informações através do mundo, coisa difícil de prever 20 anos atrás.

A área de sistemas distribuídos está em franca expansão, nas comunidades acadêmicas e industriais, introduzindo os sistemas que deverão substituir os da geração atual. Os esforços no sentido de padronizar a área buscam estabelecer uma base para a efetiva integração de diferentes sistemas no futuro. Aqui apresentaremos dois padrões propostos, o DCE/OSF [OSF90] e o CORBA/OMG [OMG91, OMG92], comentando os aspectos envolvidos nessas propostas e o impacto que devem ter na computação distribuída.

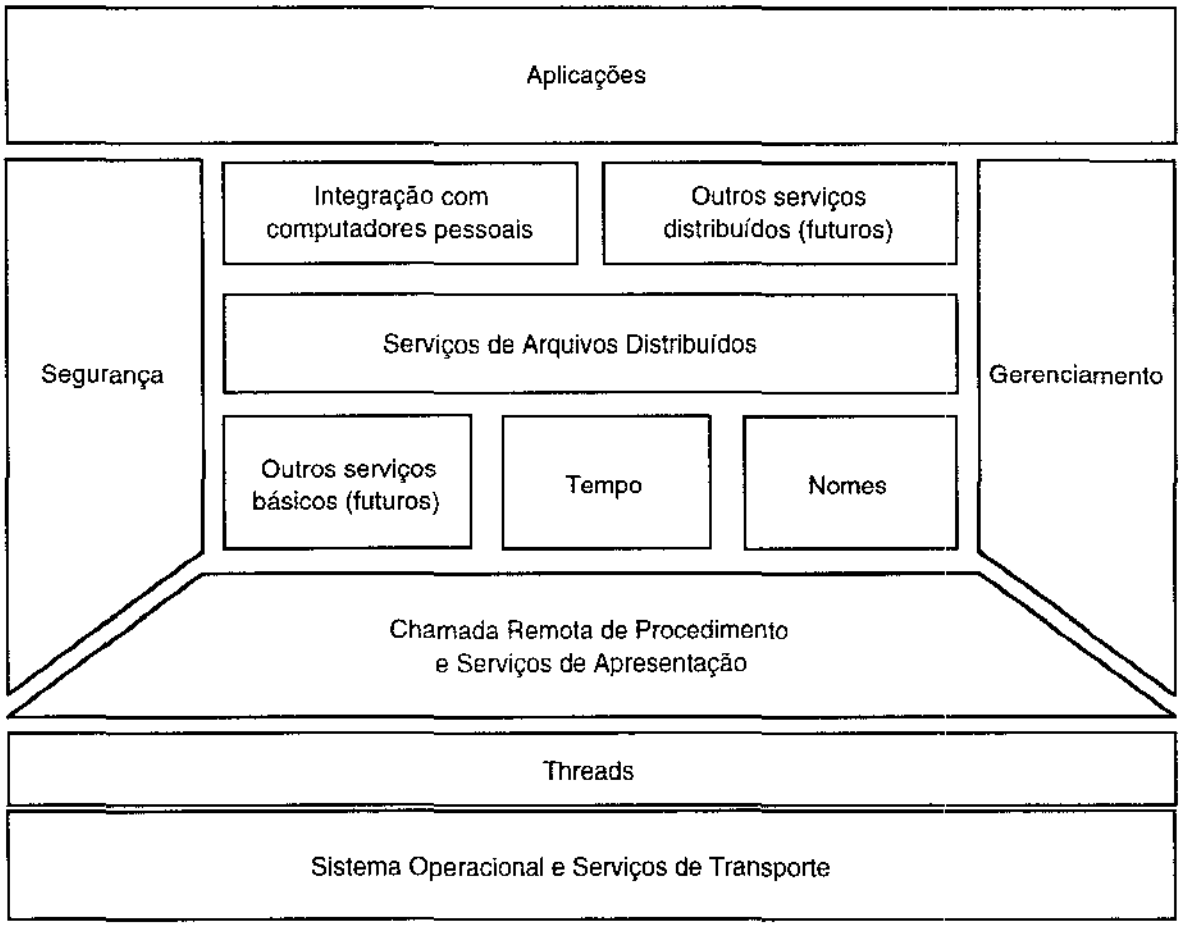
3.3.2 DCE — Distributed Computing Environment

O DCE [OSF90] é um padrão proposto pela Open Software Foundation (OSF) para ambientes de programação distribuída. Esse ambiente padrão fornece uma série de serviços, que suprem as necessidades para o desenvolvimento e execução de aplicações distribuídas. O padrão é independente de sistema operacional e de rede de comunicação, o que implica possibilidade de integração de novas aplicações com os ambientes de programação atuais.

O DCE foi desenvolvido a partir da premissa de que era necessário oferecer técnicas para a completa interoperabilidade entre sistemas heterogêneos. O processo seguido para obter um padrão foi a integração de uma série de tecnologias, desenvolvidas em separado por diferentes fabricantes. Tais tecnologias representam o "estado da arte" na área, porém representam, quando tomadas isoladamente, uma solução apenas parcial para a implantação de um ambiente inteiramente distribuído. O resultado da integração dessas tecnologias é uma base poderosa, coerente e aberta para o desenvolvimento de aplicações distribuídas.

Os serviços providos pelos ambientes padrão estão organizados conforme mostrado no diagrama da figura 3-1 (extraído de [OSF90]). A seguir descrevemos rapidamente esses serviços. Na figura há partes reservadas a tecnologias que podem ser incorporadas no futuro.

FIGURA 3-1 Arquitetura do DCE



Esses serviços estão divididos em duas categorias: os serviços distribuídos básicos e os serviços de compartilhamento de dados. Os serviços distribuídos básicos são as ferramentas usadas pelos projetistas para o desenvolvimento de serviços mais elaborados, e são os seguintes:

- Chamada remota de procedimento:[1] torna possível a ativação de procedimentos em sistemas remotos. Esse tipo de chamada é uma extensão do modelo de chamada de procedimento para um equivalente distribuído, possibilitando o desenvolvimento de aplicações distribuídas conceitualmente tão simples como programas monolíticos. A implementação desse serviço está dividida em duas partes: um conjunto de funções que implementa o protocolo de RPC, com

1. Do termo inglês *Remote Procedure Call*; também é usada a sigla RPC.

ênfase na portabilidade e independência de rede de comunicação; e um tradutor, baseado em uma linguagem de especificação de tipos, responsável pela geração de *stub procedures* a partir da descrição da interface do servidor.

- Nomes: implementa um modelo de nomes uniforme dentro do ambiente, permitindo que as aplicações obtenham, pelo nome, acesso aos recursos disponíveis (servidores, arquivos, impressoras, etc), sem conhecer a sua localização. O serviço oferece facilidades de replicação de dados, *caching* de nomes e segurança no tráfego de informações.
- Tempo: oferece uma referência de tempo comum no sistema para determinar a ordem de eventos e a sua duração. Os relógios presentes na rede são mantidos sincronizados, por *software*, com alto grau de precisão. O mecanismo utilizado é tolerante a falhas e cobre tanto redes locais como redes de longa distância.
- *Threads*: oferece facilidades para programação concorrente através de múltiplos fluxos de execução dentro de um processo. A execução paralela de várias atividades tende a diminuir o tempo em que os processadores permanecem ociosos (devido a operações de entrada e saída, e.g.), resultando em um melhor aproveitamento de CPU. Esse serviço também permite a construção de servidores multitarefa.
- Segurança: mecanismos de autenticação e autorização, integrados a outros serviços como chamada remota de procedimento (para garantir a inviolabilidade dos dados transmitidos) e sistema de arquivos distribuído (controlar o acesso dos usuários aos recursos). A administração de contas de usuário é integrada ao serviço de nomes, através de um banco de dados distribuído, que mantém a consistência e a proteção dessas informações. O uso de um mecanismo uniforme para as contas de usuário permite o uso de nome e senha únicos por usuário, em toda a rede e para todos os serviços.

Os serviços de compartilhamento de dados são implementados a partir dos serviços distribuídos básicos, e são os seguintes:

- Sistema de arquivos distribuído: consiste na integração dos sistemas de arquivos das máquinas da rede, criando um espaço de nomes uniforme e transparência de localização. O serviço provê mecanismos para a recuperação de arquivos após falha de um nó da rede, e replicação transparente de dados para melhorar o tempo de acesso e a disponibilidade das informações.
- Suporte a estações *diskless*: objetiva a utilização efetiva de estações *diskless* sem comprometimento da eficiência da comunicação em rede.
- Integração com computadores pessoais: serviços que permitem a usuários de *mainframes* ou de computadores pessoais utilizar recursos e aplicações de um ambiente distribuído, incluindo transferência e impressão de arquivos.

Os serviços do DCE são escritos em linguagem C, obedecendo aos padrões POSIX e X/Open para a especificação de interfaces. Tais implementações são portáveis para vários sistemas operacionais, e o ambiente pode ser baseado em diferentes tipos de rede, como TCP/IP e X.25.

3.3.3 CORBA — Common Object Request Broker Architecture

O CORBA [OMG91, OMG92] foi proposto pela Object Management Group (OMG), uma organização com mais 300 membros, com o objetivo de estabelecer

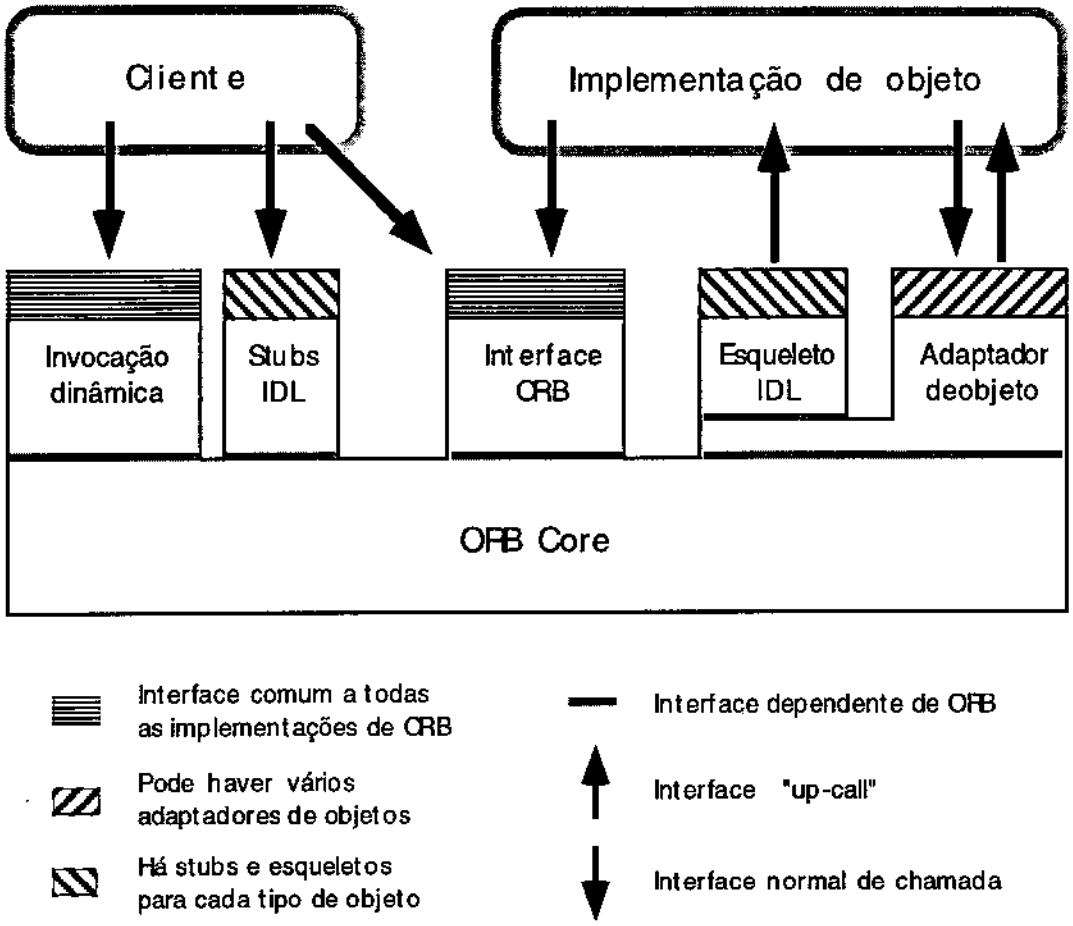
uma base para a integração de aplicações distribuídas baseadas em objetos. O padrão especifica os mecanismos utilizados para a comunicação transparente entre objetos envolvendo diferentes plataformas, sistemas e linguagens. Diferentemente do DCE, o CORBA é composto apenas de especificações, não de *software*.

O padrão é dividido basicamente em cinco partes: o núcleo ORB (*ORB Core*), uma linguagem universal para definir interfaces (IDL — *Interface Definition Language*), uma interface para permitir a chamada dinâmica de serviços (DII — *Dynamic Invocation Interface*), um banco de declarações de interfaces (IR — *Interface Repository*) e módulos para adaptar diferentes modelos de implementação de objetos (OA — *Object Adapters*). Em conjunto, esses componentes possibilitam a integração de sistemas baseados em objetos desenvolvidos com alto grau de heterogeneidade.

O núcleo do padrão é o ORB (*Object Request Broker*), componente de *software* que provê os serviços para a interação transparente entre objetos. De acordo com a nomenclatura da OMG, um sistema de programação baseado em objetos oferece serviços a clientes, onde um cliente é qualquer entidade que pode pedir a execução de determinado serviço. Um objeto é uma entidade com identidade única, e encapsula recursos, acessíveis através de serviços. Um cliente solicita a execução de um serviço através de uma requisição: a função do ORB é encontrar o objeto que deve responder a uma requisição e transportar os dados envolvidos. A maneira como os clientes e objetos interagem através do ORB é independente da localização desses, da linguagem de programação em que estão implementados, e de qual mecanismo de comunicação fazem uso.

Um ambiente de programação que queira compartilhar serviços através do padrão CORBA deve implementar o seu próprio ORB, com todos os seus componentes e interfaces especificadas. Tais implementações podem variar bastante, dependendo das características dos ambientes onde estão baseadas; isso não é importante, desde que as interfaces sejam respeitadas. A estrutura de um ORB, conforme diagrama extraído de [OMG91], é mostrada na figura 3-2.

FIGURA 3-2 Estrutura e interfaces de um *Object Request Broker*



A implementação de um objeto guarda os dados daquela instância e o código para executar os métodos. Os objetivos são especificados através de uma referência, que é utilizada pelo ORB para localizar o destinatário de uma requisição. Um cliente pode executar serviços de um determinado objeto desde que tenha uma referência para ele.

A linguagem de definição de interfaces (IDL) define os tipos dos objetos através da descrição de suas interfaces. Uma interface é um conjunto de operações, identificadas por seu nome e lista de parâmetros. O objetivo dessa linguagem é permitir que um cliente utilize as operações oferecidas por um determinado objeto conhecendo apenas sua interface.

Um cliente envia uma requisição a um determinado objeto através de funções *stubs* ou através da interface de invocação dinâmica. As funções *stubs* servem como intermediário entre o mecanismo de ativação de métodos do cliente e as funções do

ORB que fazem a transmissão de requisições. Os *stubs* são específicos para um determinado tipo de uma interface. Já o módulo de invocação permite que a chamada seja especificada em tempo de execução, quando o cliente monta uma requisição com a referência do objeto que vai executar a operação, qual a operação a ser executada, e a lista de parâmetros com os tipos correspondentes. A descrição da interface do objeto pode ser obtida através de uma consulta ao banco de interfaces, ou de qualquer outra forma disponível em tempo de execução.

Os adaptadores de objeto são utilizados para acomodar diferentes implementações de objetos, mantendo a representação comum do ORB genérica, ao tratar a heterogeneidade de implementações de maneira particular. Entre as funções de um adaptador, estão a geração e interpretação de referências, a chamada de métodos (através dos esqueletos IDL), e a ativação e desativação de implementações.

Finalmente, o módulo de interface ORB é composto de um conjunto de funções que dão acesso direto a serviços do ORB. A maioria dos objetos componentes das aplicações vai interagir com o ORB indiretamente, através dos módulos descritos anteriormente. As funções dessa interface devem ser necessariamente oferecidas por todas as implementações de ORB.

4 O Ambiente A_Hand

Se criarmos um universo, que ele não seja abstrato ou vago, mas que represente, concretamente, coisas reconhecíveis. Vamos construir um universo bidimensional, a partir de um número infinitamente grande de componentes, todos idênticos porém distintamente reconhecíveis. Poderia ser um universo de pedras, estrelas, plantas, animais ou pessoas.

Maurits C. Escher

Este trabalho de mestrado faz parte do projeto de desenvolvimento do ambiente A_Hand, em andamento no Departamento de Ciência da Computação da Unicamp. O Laboratório A_Hand reúne um grupo de trabalho voltado para a pesquisa de soluções para a produção de *software*, estimulada pela demanda por sistemas cada vez maiores e mais complexos.

As atividades do A_Hand geraram resultados em um grande número de áreas, como ambientes de desenvolvimento, linguagens de programação, hipertextos, interfaces e *groupware*. Todas essas linhas de pesquisa estão baseadas em premissas comuns, como por exemplo orientação a objetos e programação distribuída.

4.1 Histórico

O Projeto A_Hand foi criado em 1986, como parte de um projeto mais abrangente, o AIDSH (Ambiente Integrado de Desenvolvimento de *Software* e *Hardware*), que tinha por objetivo oferecer um ambiente de ponta para o desenvolvimento de projetos em computação. O A_Hand corresponde à parte específica de desenvolvimento de *software*.

O nome A_Hand significa Ambiente de Desenvolvimento de *Software* Baseado em Hierarquias de Abstração em Níveis Diferenciados. A principal característica desse ambiente é que os objetos por ele manipulados podem ser compostos de objetos mais simples, que por sua vez também podem ser objetos compostos e assim por diante. Um objeto complexo representa então uma hierarquia de objetos, com vários níveis de abstração; dessa estrutura vem o nome A_Hand.

Esse grupo de pesquisa é composto por professores doutores, assistentes de pesquisa, mestrandos e graduandos do Departamento de Ciência da Computação da Unicamp.

4.2 Objetivos

O Projeto A_Hand propõe, como seu próprio nome diz, um ambiente de desenvolvimento de *software* para a construção de sistemas de grande magnitude e complexidade. Esse ambiente é composto de um núcleo conceitualmente poderoso (hierarquias de abstração) e um conjunto de ferramentas que auxiliam na produção de *software* em todas as suas fases.

As tarefas envolvidas na produção de um programa, depois de sua especificação, são a edição, compilação/interpretação e depuração. Essas tarefas são em grande parte monótonas e repetitivas, consumindo grande quantidade de tempo em comparação ao tempo gasto no projeto e especificação do programa. Além disso, há outras tarefas, como testes, documentação, controle de versões e *back-up*, que muitas vezes são consideradas pouco importantes e, devido à falta de tempo e de ferramentas adequadas, são deixadas de lado pelo programador. Tal postura tem efeitos no produto final, podendo ficar comprometidas tanto a sua qualidade (funcionalidade exigida + desempenho + robustez) como o futuro do produto (adição de novas *features* + esforço de manutenção).

Um ambiente de desenvolvimento que permitisse ao programador concentrar-se unicamente nas questões importantes de um programa ou sistema, liberando-o de tarefas secundárias, aumentaria em muito a produtividade dos programadores, com reflexos na qualidade do produto final e no tempo de desenvolvimento.

O ambiente A_Hand tenta atender aos requisitos de produção de *software* em larga escala ao enfatizar os seguintes aspectos [Dru87a]:

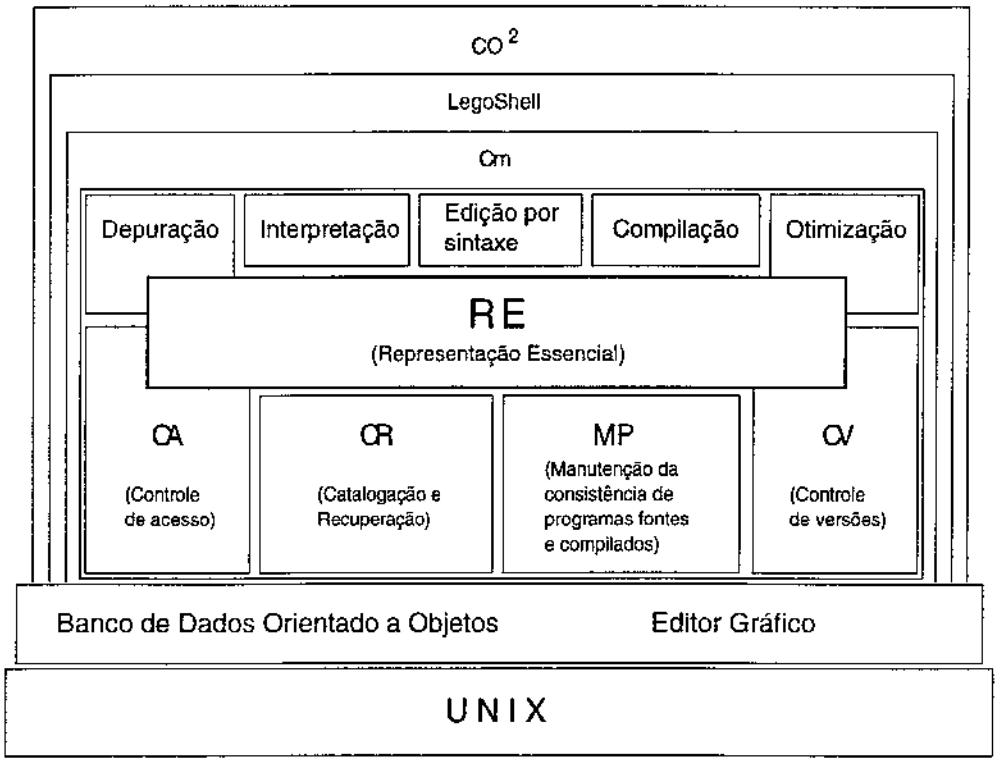
- sistemas incompletos: durante o ciclo de desenvolvimento de um sistema, várias de suas partes são construídas em paralelo, e necessitam, na medida do possível, ser especificadas, implementadas e testadas separadamente.
- desenvolvimento distribuído: um sistema complexo geralmente está a cargo de uma equipe de programadores, que utilizam ferramentas específicas para acompanhar o andamento do trabalho e facilitar a integração das partes desenvolvidas paralelamente.
- reutilização de código: a possibilidade de usar partes de um sistema em outros sistemas novos é fundamental para a economia de esforço de desenvolvimento; a reutilização de código depende de linguagens de programação adequadas e políticas de gerenciamento de versões e documentação.
- interfaces uniformes: as diversas ferramentas do ambiente devem interagir com o usuário de forma regular e uniforme, a fim de aumentar a produtividade do programador e diminuir o tempo de treinamento em novas ferramentas.

4.3 Estrutura

É importante frisar que o ambiente proposto pelo A_Hand não está completamente implementado. Portanto, algumas partes do ambiente são descritas com base no que já foi feito e que se encontra disponível; outras partes são descritas da forma como foram planejadas até agora.

O ambiente A_Hand pode ser conceitualmente descrito pelo seguinte diagrama:

FIGURA 4-1 Estrutura do ambiente A_Hand



Os programas serão manipulados pelo ambiente através de sua Representação Essencial (RE). O código-fonte de um programa nada mais é que a "decompilação" de sua RE, que é basicamente uma árvore sintática acrescida de atributos.

Os editores a serem usados serão sensíveis à sintaxe da linguagem na qual foi escrito um programa, ao contrário dos editores comuns, que exibem um programa como uma seqüência de caracteres. Com um editor sensível à sintaxe, é possível visualizar um programa como um conjunto de blocos ou partes inter-relacionadas. Em C, por exemplo, a visão macroscópica de um programa mostra uma seqüência de declarações, seguida de uma seqüência de funções; uma função é composta de declarações e de um bloco, que por sua vez é composto de blocos menores, e assim por diante.

As tarefas de catalogação e recuperação (CR), controle de versões (CV), controle de acesso (CA) e manutenção da consistência de programas fontes e compilados (MP) estão a cargo do ambiente. Praticamente todas essas operações operam sobre a RE dos programas. As ferramentas envolvidas apresentarão uma interface uniforme para as diversas linguagens suportadas pelo ambiente.

A linguagem de programação Cm [Dru88, Sil88, Fur91, Tel93] é a linguagem básica do ambiente. É orientada a objetos, e tem funcionalidade igual ou superior a C++, com a qual apresenta muitas semelhanças. As linguagens CO^2 e LegoShell [Dru89] são as linguagens de comandos do ambiente. A linguagem CO^2 é uma *shell* textual, apta para o desenvolvimento de protótipos. A LegoShell é uma *shell* gráfica, onde é possível especificar computações através da conexão entre programas, arquivos e dispositivos periféricos.

O ambiente A_Hand é baseado no sistema operacional Unix, pelas facilidades que este apresenta para o desenvolvimento de *software* e pela sua disponibilidade nas comunidades acadêmica e industrial.

4.4 A Linguagem Cm

O ambiente A_Hand destina-se à produção de sistemas de *software* grandes e complexos. Tais sistemas, para serem projetados e construídos dentro de metas de prazo e qualidade satisfatórias, não podem prescindir do desenvolvimento distribuído e do uso intensivo de ferramentas específicas e elaboradas. Além do mais, tais sistemas devem levar em conta aspectos como estruturação modular e reutilização de código, eficiência e portabilidade.

As linguagens disponíveis à época do início do Projeto A_Hand, como C [Ker78], C++ [Str86], Modula-2 [Wir85] e Ada [DoD83], não atendiam plenamente a estas exigências [Tel93]. Foi então projetada a linguagem Cm, para ser a linguagem de programação básica do ambiente.

A linguagem Cm é derivada de C com respeito a comandos, operadores e expressões. Foram corrigidas ou eliminadas certas particularidades de C, como a sintaxe das declarações, deficiências em relação a tipos e o uso do pré-processador. A primeira versão da linguagem, apresentada em [Dru88, Sil88, Fur91], acrescenta a C as seguintes características:

- estrutura de tipos uniforme, que permite verificação rigorosa e estática de tipos em qualquer expressão de um programa, e regularidade no tratamento de tipos primitivos e derivados.
- mecanismos de abstração de dados para aumentar o poder de expressão da linguagem, como módulos, tipos abstratos de dados, polimorfismo e herança.

A versão corrente da linguagem [Tel93] apresenta novos recursos, como construtores e destrutores de classes e objetos, sobrecarga de operadores e funções, alterações sintáticas menores e tratamento de exceções.

4.4.1 Classes

A linguagem Cm suporta classes como um novo construtor de tipos. O uso de classes introduz na linguagem programação modular e orientada a objetos. Uma classe define constantes, tipos, variáveis e funções: cada um desses componentes pode ser exportado, e assim fazer parte da interface da classe. Uma classe está descrita em um único arquivo de código-fonte, sendo, portanto, uma unidade de

compilação. Não há classes aninhadas nem funções declaradas fora de classes. Uma classe define um tipo, e variáveis desse tipo são objetos, instâncias dessa classe.

As classes podem ser parametrizadas, tanto por valores como por tipos. Classes com parâmetros são denominadas meta-classes, pois definem classes de classes, conforme a instanciação dos seus parâmetros. Usando classes com tipos como parâmetros temos polimorfismo paramétrico na linguagem, como mostrado no exemplo abaixo:

```
class Stack <type T; int SIZE>

T[size] st;
int top = 0;

export void push (T x)
{
  st[top++] = x;
}

export T pop()
{
  return st[--top];
}
```

A classe *Stack* define um número ilimitado de classes, pois o tipo *T* pode ser instanciado com qualquer tipo da linguagem, primitivo ou derivado.

`Stack<int,500>`	pilha de 500 números inteiros
`Stack<char*,200>`	pilha de 200 cadeias de caracteres.
`Stack<Stack<int,100>,10>`	pilha de 10 pilhas de 100 números inteiros.

Uma classe pode ter parâmetros com valor *default*, que são usados quando o parâmetro correspondente é omitido quando do uso da classe. No caso da classe *Stack*, ela poderia ser declarada como:

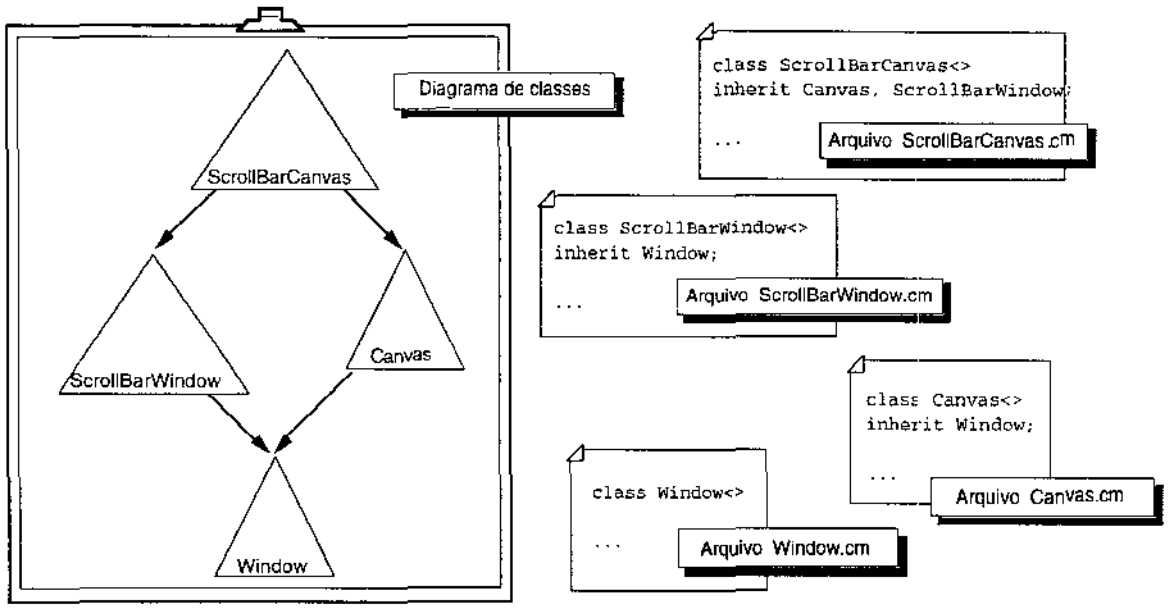
```
Stack<type T = int; int size = 1024>
```

E então todas as seguintes classes são válidas e equivalentes

```
Stack<int,1024>, Stack<int>, Stack<>  e Stack.
```

Novas classes podem ser criadas a partir de modificações e extensões a classes já existentes, através do mecanismo de herança. A linguagem Cm suporta herança múltipla, como ilustrado pelo exemplo seguinte:

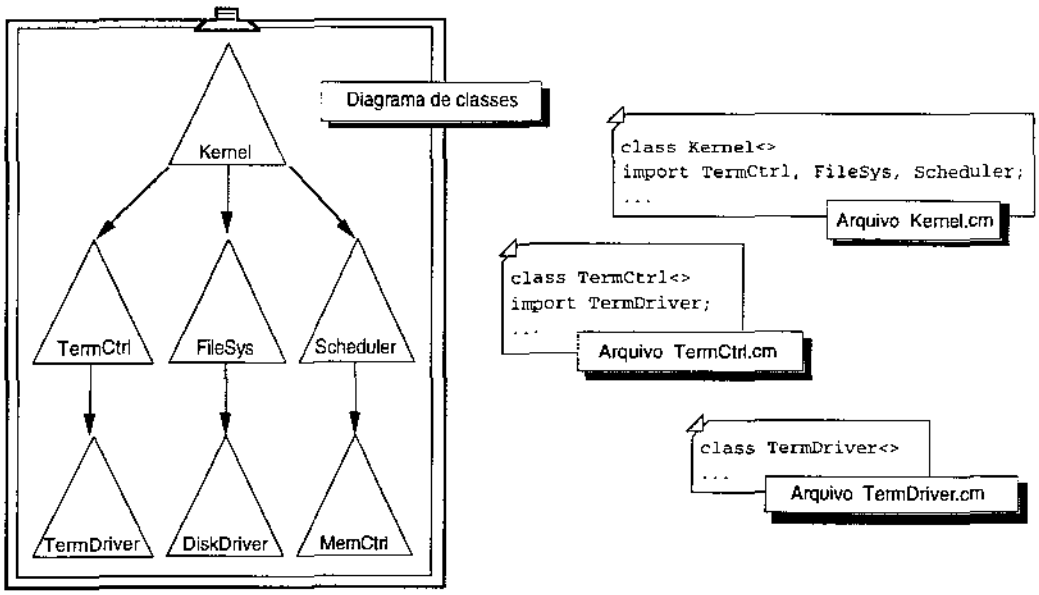
FIGURA 4-2 Exemplo de herança



Para o compartilhamento de funcionalidade, Cm também dispõe do mecanismo de importação de classes. Uma classe pode importar várias outras, declarando com isso o uso dos tipos definidos por essas classes. Dada uma variável cujo tipo é uma classe importada, os componentes dessa classe disponíveis para uso são aqueles que constam da interface da classe importada, isto é, os componentes exportados.

Conceitualmente, uma classe que herde ou importe outras classes representa, em relação às classes herdadas e importadas, o resultado de operações de especialização e agregação, respectivamente. Tal classe depende conceitualmente das classes herdadas e importadas, pois uma mudança em alguma dessas classes provavelmente se reflete na classe dependente. Essas relações definem uma hierarquia de classes, onde a classe que depende das demais, direta ou indiretamente, representa o topo dessa hierarquia.

FIGURA 4-3 Exemplo de importação de classes



4.4.2 Sistema de tipos

Cm tem alta uniformidade de tipos, tratando igualmente tipos primitivos e tipos derivados. Para qualquer tipo possível da linguagem, estão definidas as operações de atribuição (operador =), igualdade (operador ==), desigualdade (operador !=), criação e destruição dinâmica de objetos (funções `new` e `release`, respectivamente). Além disso, valores de qualquer tipo podem ser passados como parâmetro e retornados por funções.

Tanto funções como operadores podem sofrer sobrecarga. Podem ser definidas várias funções com o mesmo nome, cada uma operando sobre tipos diferentes. Isso é diferente de polimorfismo paramétrico, já que o código de cada uma dessas funções é específico para cada tipo envolvido. Os operadores, ao sofrerem sobrecarga, não podem ter alteradas nem sua precedência nem o número e ordem de seus operandos.

A seguinte tabela exemplifica a sobrecarga de funções. A função *print()* é definida mais de uma vez, com diferentes parâmetros, e com comportamento diferente em cada um desses casos.

TABELA 4-1 Sobrecarga de funções

```
void print (char* str)          print ("Hello World!\n");
{ ... }
void print (int i)              print (25 * 25 );
{ ... }
void print (float f)            print ( 3.14159 / 2.7182818 );
{ ... }
```

A seguir temos um exemplo de sobrecarga de operador.

```
class complex<>
float re, im;

constructor (float r=0, i=1)
{ re = r; im = i; }

export operator+ (complex x)
{
   return complex( re + x.re, im + x.im);
}

Class usecomplex<>
import complex;
// declaração de objetos da classe complex
complex c1, c2, c3;
// exemplo de utilização
c1 = c2 + c3;
```

A sintaxe de declaração de tipos foi alterada para facilitar a legibilidade e fornecer uma forma canônica para a definição de tipos. Os construtores de tipo são associativos à esquerda e a leitura se faz sempre da direita para a esquerda.

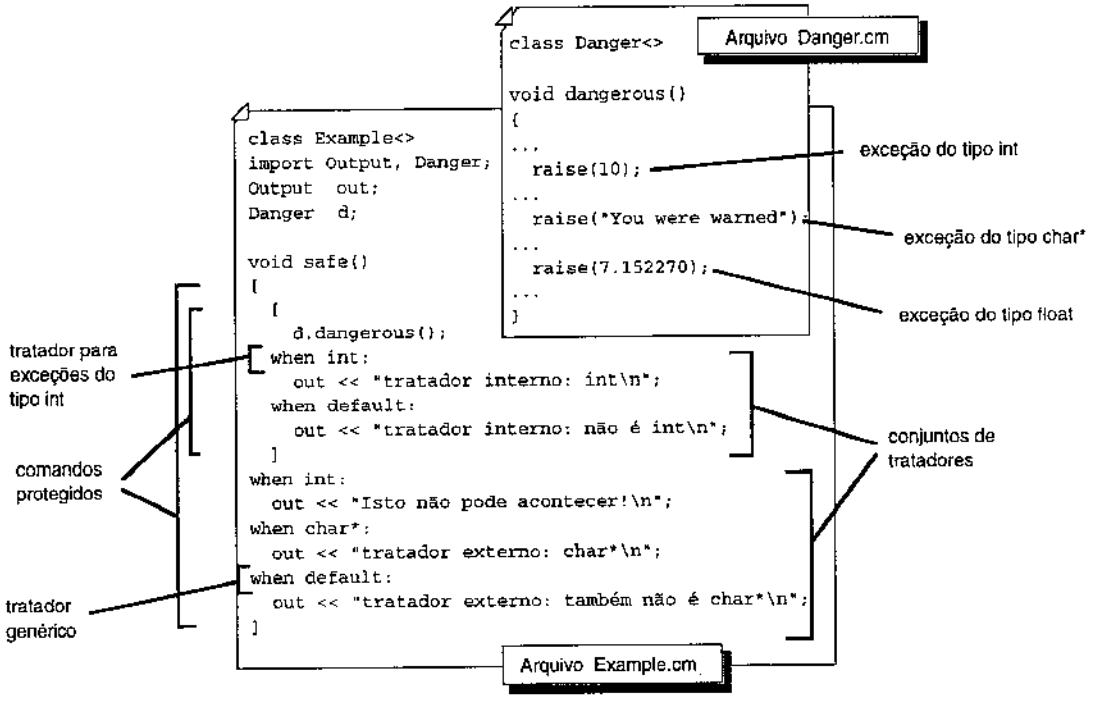
`int*  api`	*api* é apontador para tipo `int`
`int[5]*  apvi`	*apvi* é apontador para tipo vetor de 5 elementos de tipo `int`
`int*[10]  vapi`	*vapi* é vetor de 10 elementos do tipo apontador para tipo `int`
`int()*  apfi`	*apfi* é apontador para função que retorna tipo `int`, sem parâmetos
`int*(float)*  afapi`	*afapi* é apontador para função que retorna tipo apontador para tipo `int`, com um parâmetro do tipo `float`.

4.4.3 Exceções

O mecanismo de tratamento de exceções permite a execução de código previsto pelo programador para lidar com o aparecimento de situações anormais ou especiais durante a execução do programa. O mecanismo implementado em Cm é baseado no modelo de terminação, semelhante ao da linguagem C++ [Koe90].

Em Cm, exceções são identificadas por seu tipo. A especificação de tratadores se dá através de comandos protegidos, que são blocos de comandos delimitados por abre e fecha-colchetes, '[' e ']', ao invés de abre e fecha-chaves, '{' e '}'. Um comando protegido especifica um escopo para um conjunto de tratadores a serem usados caso uma exceção associada a um deles ocorra naquele contexto. Quando o fluxo de execução chega a um comando protegido, o conjunto de tratadores associados é ativado, permanecendo ativo até que o fluxo de execução deixe o comando protegido.

FIGURA 4-4 Propagação e tratamento de exceções



Caso não seja encontrado um tratador apropriado, o sistema de execução oferece um tratador genérico que captura a exceção. O procedimento *default*, executado por esse tratador, é abortar o programa e apresentar uma mensagem de erro indicando as circunstâncias em que a exceção foi gerada.

4.4.4 Cm Distribuído

A proposta inicial do Cm previa que a entrada/saída e concorrência não seriam parte da linguagem [Dru87a]. Entrada e saída, por exemplo, seriam implementadas através de portas[1] de comunicação, com a classe pré-definida `Port`; eventualmente, seria também oferecida uma classe `File`, com funções de entrada/saída semelhantes às de C. Atualmente, as funções de entrada e saída estão a cargo das classes `Input` e `Output`, respectivamente. O uso de portas de comunicação, assim como mecanismos para criação de objetos distribuídos e controle de concorrência, será descrito como parte do modelo de Objetos Distribuídos, no capítulo 5. Esse modelo significa uma mudança sensível na linguagem, que passará a chamar-se Cm Distribuído. Nessa nova linguagem, uma aplicação típica é composta de vários objetos cooperantes, que podem estar situados em diferentes máquinas de uma rede.

4.5 O Sistema OMNI

O sistema OMNI [Dru92] oferece facilidades para a criação de processos distribuídos e comunicação entre esses processos. Seu objetivo é facilitar a construção de aplicações distribuídas com uniformidade, transparência e segurança. O ambiente de uso do OMNI é uma rede de máquinas heterogêneas que utilizam Unix como sistema operacional.

O sistema é composto de uma biblioteca de funções escritas em C, e de um conjunto de *daemons* sendo executados em cada uma das máquinas integradas pelo sistema. O projeto do OMNI teve como importante preocupação sua portabilidade, razão pela qual sua implementação não faz nenhuma modificação no Unix propriamente dito: o OMNI opera sobre o Unix, oferecendo, através de uma camada de *software*, uma interface de sistema funcionalmente distribuído.

O sistema OMNI é a base das aplicações e ferramentas distribuídas do A_Hand, portanto sua funcionalidade é bem geral. Como exemplo de projetos a serem implementados usando OMNI, podemos citar o Phone+ (sistema de teleconferência) e o Sistrac (Sistema de Suporte ao Trabalho Cooperativo). As linguagens de programação do A_Hand (Cm, CO^2 e LegoShell) usarão recursos de programação distribuída fornecido por um módulo chamado *Run Time System*, que tem por função servir de interface entre essas linguagens e o OMNI.

4.5.1 Módulos do sistema OMNI

O sistema OMNI lida, basicamente, com as seguintes abstrações: processos OMNI, identificações OMNI, portas, conectores especiais e mensagens. Existem outras abstrações menos importantes, como contextos, processos comuns, grupos de processos e grupos de portas. O sistema fornece funções para criação, uso e destruição desses recursos. As abstrações mais importantes são definidas assim:

1. O termo "porta" é utilizado no lugar do original inglês port. No apêndice D ("Vocabulário"), discutimos o uso desses termos.

- processo OMNI: processo tradicional do Unix com estruturas adicionais para gerenciamento pelo OMNI.
- identificação OMNI: identifica univocamente cada recurso do OMNI.
- portas: meio de comunicação entre processos.
- conectores especiais: objetos especiais para concentração e disseminação de dados.
- mensagens: unidade de informação trocada entre processos através de portas.

Cada uma dessas abstrações é implementada em uma parte do OMNI. Funcionalmente, o sistema é dividido em quatro módulos: Servidor de Nomes, Gerenciador de Processos, Portas Conectáveis e Portas Não-Conectáveis. A seguir esses módulos são explicados.

4.5.1.1 Servidor de Nomes

Objetos que representam recursos do OMNI, tais como processos e portas, têm uma identificação usada para distingui-los dos demais objetos do sistema. Essa identificação, conhecida como OMNIid, é única no espaço e no tempo, ou seja, ela identifica univocamente um objeto no sistema, e mesmo após a destruição desse objeto sua OMNIid não será reutilizada.

O Servidor de Nomes permite associar nomes simbólicos a seqüências arbitrárias de *bytes*, que num caso particular podem ser a OMNIid de um objeto; nesse caso, dizemos que tal objeto foi cadastrado com um nome simbólico. O Servidor de Nomes fornece funções para busca de objetos usando um nome simbólico como argumento; uma vez cadastrado, um nome simbólico pode ser localizado de qualquer ponto da rede. Dessa maneira, é possível que um objeto de determinada aplicação possa ser compartilhado por outras aplicações através de um nome simbólico conhecido.

Um nome simbólico pode possuir três atributos: visibilidade, unicidade e permissões de acesso. O atributo de visibilidade especifica se um nome é visível apenas na máquina onde foi cadastrado ou se é visível através de toda a rede. A unicidade de um nome determina se ele deve ser único em sua máquina, único em toda a rede, ou se ele pode ter homônimos. As permissões de acesso se aplicam a um nome simbólico de maneira semelhante às permissões de um arquivo Unix, pois associados a um nome estão o usuário que o cadastrou e o grupo desse usuário; baseando-se nessas informações, e nas permissões de acesso especificadas, pode-se permitir ou negar acesso a uma informação cadastrada no Servidor.

Existem espaços de nomes independentes, denominados contextos. Nomes idênticos cadastrados em diferentes contextos não entram em conflito. A utilização de contextos é útil para não limitar a escolha de nomes para as aplicações e permitir que diferentes aplicações compartilhem de um espaço de nomes mais geral.

Quando um nome simbólico é cadastrado, através da função pertinente, o par (nome simbólico, informação associada) é guardado pelo Servidor em uma estrutura de dados local à máquina. Uma consulta a um nome local é uma pesquisa, feita pelo Servidor, nessa estrutura. Se o nome é global, e não é encontrado na máquina do processo que fez a consulta, o *daemon* do Servidor dessa máquina comunica aos demais *daemons* (situados em outras máquinas) que está procurando

um nome simbólico global, e passa esse nome a eles. Se o nome é encontrado, a informação associada é repassada ao processo (que fez a consulta) e colocada em um *cache* de nomes externos. A consistência das informações armazenadas nesse *cache* não é feita pelo Servidor, e deve ser incumbência da aplicação.

4.5.1.2 Gerenciador de Processos

Através das funções desse módulo pode-se criar processos distribuídos pela rede. A relação de processo pai e processo filho, como definida no Unix, é preservada pelo OMNI. Os processos criados pelo Gerenciador podem trocar sinais (como KILL, CONT, HUP, etc) transparentemente, não importando se estão em máquinas distintas.

Um processo é criado pela primitiva *om_createProc()*, que retorna a OMNIid do processo criado. A partir de uma máquina da rede, é possível criar processos em qualquer outra máquina. A função *om_createProc()* aceita como parâmetros informações adicionais para a criação de um processo: parâmetros de linha de comando, variáveis de ambiente, redirecionamento de descritores de arquivo, grupo e sessão do novo processo, diretório corrente, máscara de sinais, etc.

As demais funções do Gerenciador são equivalentes distribuídos de funções do Unix para tratamento de processos. A função *om_kill()*, por exemplo, envia um sinal para um processo usando como argumento uma OMNIid, ao invés de um Pid, para especificar o processo destinatário. Outras funções do Gerenciador são *om_wait()*, *om_signal()*, *om_getpid()* e *om_ptrace()*.

4.5.1.3 Módulo de Portas Conectáveis

Este módulo, juntamente com o módulo de Portas Não-Conectáveis, implementa as portas de comunicação do OMNI. As portas são filas de mensagens trocadas entre processos distribuídos. Uma mensagem é uma seqüência de *bytes* enviada de um processo a outro, cujo tamanho e conteúdo depende de quem as usa. Processos gerenciados pelo OMNI podem se comunicar através de sinais e de mensagens; para os usuários das linguagens do ambiente (Cm, CO^2 e LegoShell), entretanto, a comunicação deve se restringir a mensagens, já que essas linguagens implementarão portas de comunicação como objetos. Os programas desenvolvidos nessas linguagens também farão uso de sinais, mas isso será feito para fins de controle e de maneira transparente para o programador.

As portas conectáveis levam esse nome porque seu uso depende de uma conexão prévia. Uma conexão estabelece um canal de comunicação entre duas portas: através desse canal os processos donos das portas trocam mensagens. As conexões podem ser feitas e desfeitas dinamicamente, conforme as necessidades dos processos envolvidos.

Toda porta de comunicação, e em particular as conectáveis, possui uma OMNIid, e está vinculada ao processo que a criou; assim, não é possível para uma porta "migrar" de um processo para outro. As portas conectáveis têm ainda os seguintes atributos que regulam o seu uso:

- direção: uma porta conectável pode ser de entrada, de saída, ou de entrada e saída simultaneamente.
- semântica: é a estrutura dos dados transmitidos ou recebidos pela porta. Os valores possíveis desse atributo são: stream, tamanho fixo, caractere separador, tamanho variável e semântica herdada.
- nome simbólico: atributo opcional, é o nome com o qual uma porta pode ser cadastrada no Servidor de Nomes.

A interface provida pelo OMNI para o uso de uma porta lembra as funções básicas de entrada/saída do Unix. Uma porta é usada como um descritor de arquivo; a função para envio de dados é *om_write()*, e a função para recepção é *om_read()*. Mensagens também podem ser recebidas assincronamente, através de uma função tratadora.

A conexão entre duas portas pode ser simétrica ou assimétrica. No caso simétrico, além dos processos donos das portas há a necessidade de um terceiro processo, que executa a função *om_connect()* para estabelecer a conexão. No caso assimétrico, um processo dono de uma porta toma a iniciativa de conectar-se a outro processo. Em qualquer um dos casos é permitido a qualquer um dos processos conectados tomar a iniciativa de desfazer a conexão, através de uma chamada a *om_disconnect()*.

4.5.1.4 Portas Não-Conectáveis

Essas portas são mais apropriadas para programação num estilo cliente-servidor, pois não existe uma vinculação fixa entre quem envia e quem recebe uma mensagem. Na verdade, nem há possibilidade de conexão, pois só existem portas não-conectáveis de entrada. Para enviar uma mensagem a uma porta não conectável basta usar a função *om_send()*, passando como parâmetro, além da mensagem, a OMNIid da porta destino.

No processo dono da porta não-conectável, as mensagens podem ser recebidas de maneira síncrona, através da função *om_receive()*, ou assíncrona, através do registro de uma função tratadora de mensagens associada àquela porta. Se a mensagem recebida causar uma resposta, o processo que originou a mensagem deve ter enviado, como parte da mensagem, a OMNIid de uma porta sua (também não-conectável) que deve receber a resposta. Esse "protocolo" depende dos programas envolvidos, pois não há suporte do OMNI para tal.

Portas não-conectáveis podem ser agrupadas, criando uma nova "porta" que corresponde a um grupo. Uma mensagem pode ser enviada a um grupo exatamente da mesma maneira que é enviada a uma porta. A semântica de um grupo é determinada no momento da sua criação e especifica o efeito do recebimento de uma mensagem para o grupo. Se o grupo tem semântica *broadcast*, a mensagem é repassada para todas as portas do grupo; se a semântica é *mailbox*, a mensagem é enviada para somente uma das portas do grupo. Portas podem entrar e sair de um grupo dinamicamente, e podem fazer parte de vários grupos simultaneamente. Portas de processos sendo executados em máquinas diferentes podem estar em um mesmo grupo.

4.5.1.5 Conectores especiais

Embora os conectores especiais sejam implementados pelo módulo de portas conectáveis, eles serão descritos aqui, separadamente, em virtude da sua importância.

Os conectores especiais fazem parte da linguagem LegoShell (seção 4.6). Os programas escritos em LegoShell, chamados de computações, são especificados como um conjunto de programas, arquivos e dispositivos periféricos conectados entre si através de portas. Os chamados conectores especiais são usados para concentrar e distribuir mensagens: eles possuem um número arbitrário de portas, usadas para receber e enviar mensagens, além de um *buffer* para armazenamento temporário de mensagens. Há dois tipos de conectores especiais:

- conector estrela: todas os dados recebidos pelas portas de entrada são replicados e enviados a todas as portas de saída.
- conector *mailbox*: cada mensagem recebida em uma porta de entrada é enviada para somente uma porta de saída.

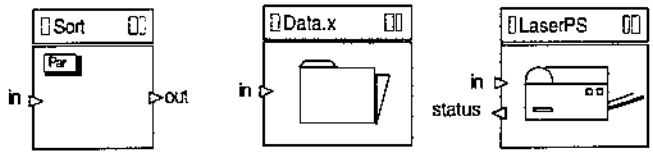
4.6 A Linguagem LegoShell

A LegoShell [Dru89, Ari91] é, além de uma nova linguagem, um novo conceito de interface entre sistema operacional e usuário [Dru87a]. Esta nova interface privilegia a interação com o usuário, apresentando dados, programas e operações através de ícones. A LegoShell será, dentro do ambiente A_Hand, a visão oferecida aos usuários pelo ambiente.

Para o desenvolvimento de programas, o usuário dispõe de um conjunto de objetos básicos, como arquivos, programas, dispositivos periféricos e peças para conexão entre esses objetos. Uma conexão entre dois componentes da LegoShell especifica um canal para o fluxo de dados entre os componentes conectados. Ao compor vários objetos, o resultado são objetos mais elaborados; nesse aspecto a construção de programas assemelha-se ao jogo de crianças Lego, que deu nome à linguagem. Programas feitos na LegoShell são chamados computações.

Um programa define portas de entrada e de saída, conforme sejam suas necessidades de trocas de informação com o mundo externo. Um arquivo terá associada a ele uma porta de entrada ou de saída, quando for utilizado para operações de escrita ou leitura, respectivamente. Os dispositivos periféricos têm portas de entrada e saída conforme sua funcionalidade.

FIGURA 4-5 Exemplo de componentes da LegoShell



O botão à esquerda do nome serve para transformar a "caixinha" em um ícone. Os botões à direita do nome são usados para iniciar/encerrar e suspender/continuar a execução desse componente. O botão `Par`, colocado debaixo do nome, permite a especificação de parâmetros para esses componentes. As portas de comunicação são representadas por triângulos colocados na borda das "caixinhas".

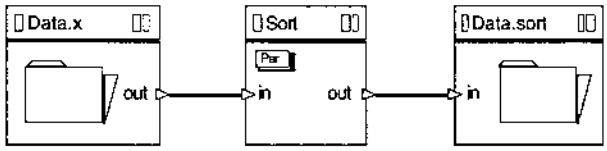
A conexão entre os objetos não é determinada por estes, mas sim por quem está editando o programa (um usuário ou outro programa). Portanto, um componente não sabe a quem ele está ligado, e nem precisa saber, pois a conexão se dá através de portas, que são uma espécie de referência ao outro participante da conexão. Essa abordagem é semelhante aos descritores padrão dos processos Unix: um programa, ao ser executado, não sabe se vai escrever seus resultados na tela do terminal, ou em um arquivo, ou em um *pipe*. Assim, os componentes da LegoShell podem ser usados em diferentes contextos, possibilitando sua fácil reutilização.

A conexão entre duas portas define um canal de comunicação, por onde fluirão dados produzidos pelo objeto dono da porta de saída e consumidos pelo processo dono da porta de entrada. Essa abordagem constitui uma extensão natural do paradigma de comunicação do Unix (*Stream-Oriented Communication*).

A manipulação desses objetos através da LegoShell faz lembrar a construção de sistemas eletrônicos, pois esses objetos de *software* têm características semelhantes aos objetos de *hardware* usados nesses circuitos, os chips eletrônicos: funcionalidade encapsulada, interface bem definida e alto grau de qualidade. A partir de uma linguagem como essa, os programadores podem pensar em construir um *software* que seja totalmente "plugável", fácil de ser entendido e usado.

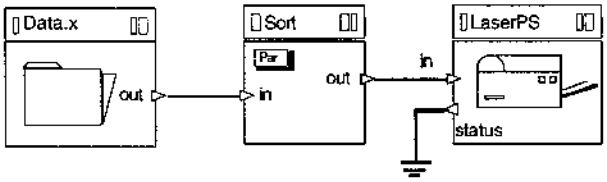
A conexão entre objetos se faz através de portas desses objetos que sejam compatíveis entre si. Um exemplo muito simples de computação é o seguinte:

FIGURA 4-6 Computação que ordena um arquivo, com o resultado em outro arquivo



Para destacar o aspecto de reusabilidade, podemos usar o arquivo `Data.x` e o programa `Sort` em outra computação muito semelhante:

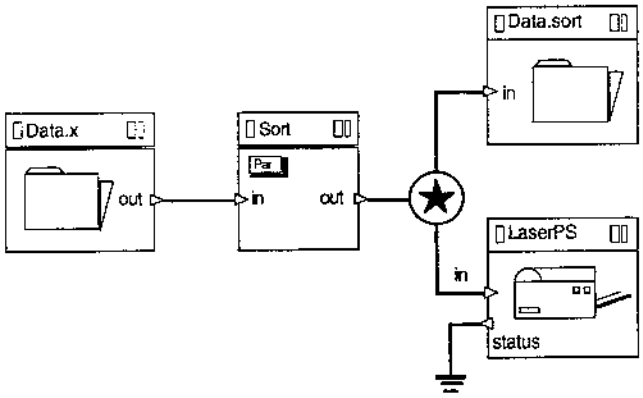
FIGURA 4-7 Computação que ordena um arquivo, com o resultado na impressora



Portas cuja ligação não é necessária para os resultados do programa — como a porta `status` da impressora do exemplo — são conectadas a um componente especial, denominado "buraco negro", representado na ilustração pelo sinal de "terra". Conectado a uma porta de entrada, ele envia um sinal de EOS (*End of stream*), indicando ausência de dados; conectado a uma porta de saída, ele destrói todas as mensagens enviadas por essa porta. Conceitualmente ele age como o dispositivo `/dev/null` do sistema Unix, e serve para "fechar" uma computação.

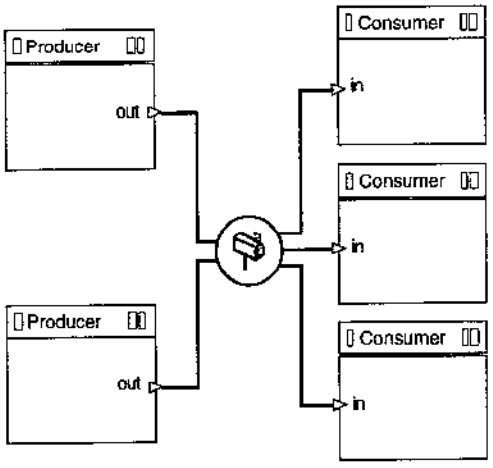
Os chamados conectores especiais permitem a construção de programas onde o fluxo de dados é multi-dimensional, devido à concentração e disseminação de dados. Os conectores especiais são o conector estrela e o conector *mailbox*. Eles são mostrados abaixo:

FIGURA 4-8 Computação com conector estrela



O conector estrela replica os dados recebidos nas portas de entrada e os envia para todas as portas de saída. No caso, o resultado do programa `Sort` será enviado tanto para o arquivo `Data.sort` como para a impressora.

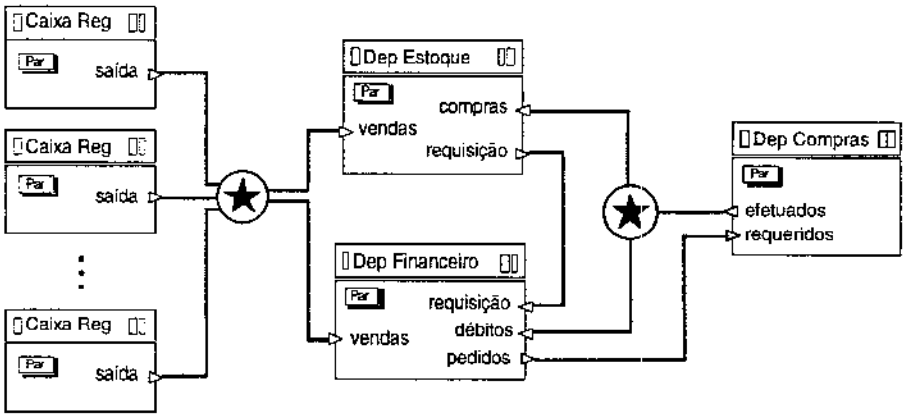
FIGURA 4-9 Computação com conector *mailbox*



O conector *mailbox* coleta os dados das portas de entrada e os distribui às portas de saída sob demanda; um dado recebido pelo conector é enviado para apenas uma porta de saída. No caso, um dado gerado em um dos programas produtores será lido por apenas um dos programas consumidores.

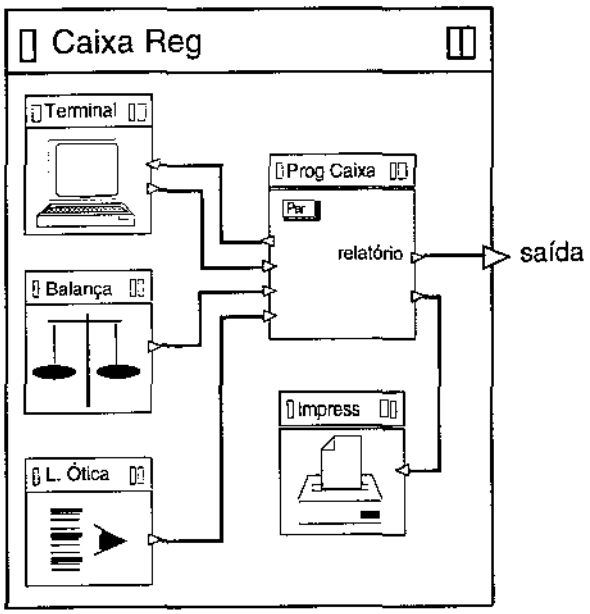
Com esses elementos básicos (programas, arquivos/dispositivos e conectores especiais) é possível compor computações bastante complexas.

FIGURA 4-10 Computação que implementa um supermercado



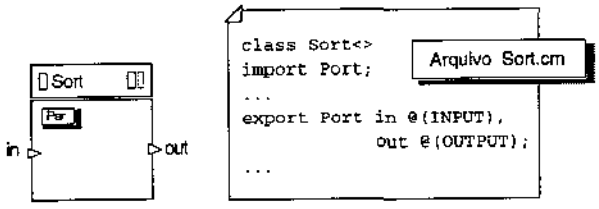
Uma vez construída, uma computação pode ser encapsulada e usada como componente simples para compor outras computações. Essa operação de encapsulamento é uma forma de abstração. As portas de uma abstração serão as portas de seus componentes que não tiverem sido conectadas. A porta de uma abstração é apenas uma referência, que permite o efetivo encapsulamento da porta real. Do exemplo do supermercado, na figura anterior, podemos editar um componente e verificar que ele não é um programa, e sim uma abstração.

FIGURA 4-11 Expansão de um componente



Juntamente com arquivos e dispositivos periféricos, os programas são os componentes básicos das computações. Esses programas são escritos em Cm, e a edição de um objeto desses na LegoShell deve mostrar o arquivo fonte da classe do objeto. Usando o programa *Sort* (figura 4-6) como exemplo:

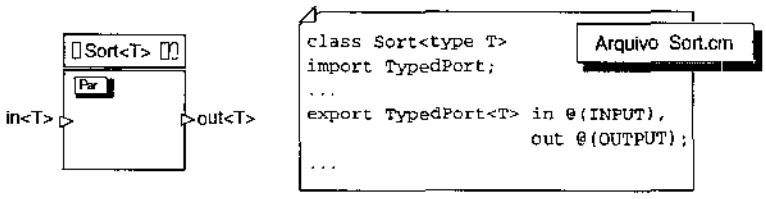
FIGURA 4-12 Programa em Cm como componente da LegoShell



O botão *Par* permite a especificação dos parâmetros desse componente. No caso de programas em Cm, há os parâmetros de classe (no caso das meta-classes, uma lista de parâmetros delimitada por < e >) e parâmetros de execução, que serão passados para o construtor do objeto (no caso específico de programa Cm) ou para uma lista de parâmetros de "linha de comando", como os parâmetros *argc* e *argv* usados em programas C.

As portas de comunicação podem ser tipadas,[2] usando a classe *TypedPort*, uma especialização da classe *Port*. Uma porta tipada tem um tipo associado, e as operações de entrada e saída são executadas atomicamente sobre dados desse tipo. Para o uso de portas tipadas é necessário assegurar a compatibilidade dos tipos das portas que participam da conexão; esse aspecto está descrito na seção seção 5.5.2, página 5-34.

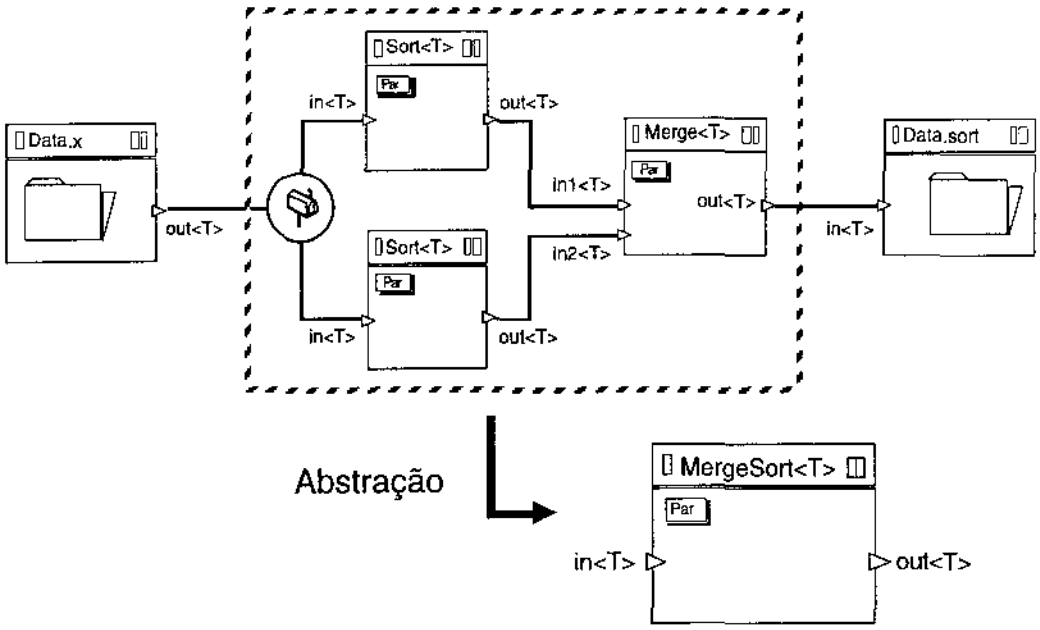
FIGURA 4-13 Programa Cm com portas tipadas



A edição de uma computação pode envolver meta-classes, e os parâmetros das classes podem ser referências a parâmetros da abstração. Portanto, é possível construir computações polimórficas, preservando na LegoShell o polimorfismo paramétrico da linguagem Cm.

2. Usamos "portas tipadas" no lugar do termo inglês *typed port*; o apêndice D ("Vocabulário") apresenta uma discussão sobre o uso desses termos.

FIGURA 4-14 Exemplo de computação polimórfica



A LegoShell também será utilizada para acompanhar e controlar a execução das computações. Durante a execução dos componentes da computação, a LegoShell dispõe de mecanismos para visualizar dinamicamente a evolução e as condições de execução, como por exemplo tráfego de informações, dados armazenados dentro dos arquivos, carga dos conectores especiais, etc.

As abstrações da LegoShell também incorporarão o mecanismo de tratamento de exceções. Isso porque, para um componente que faz parte de uma abstração, o seu contexto de execução é a própria abstração. Quando dizemos que um programa é posto para funcionar, através de um comando de uma *shell*, por exemplo, se esse programa gerar uma exceção não tratada ele vai, em último caso, sinalizar o problema para a *shell* e encerrar sua execução. Se um programa estiver encapsulado dentro de uma abstração, nesse caso a exceção propagada para fora do programa passa a ser de responsabilidade da abstração; deve-se notar que, para o programa, o contexto é irrelevante.

A LegoShell, pela sua generalidade, é implementada por um sistema muito complexo. Ele é basicamente dividido em três partes: o editor, o sistema de execução e o gerador de código. O editor é usado para editar computações, verificar a consistência das computações, fornecer parâmetros, executar computações, mostrar resultados, etc.A interface com o usuário permite selecionar ícones com o *mouse*, fazer conexões envolvendo portas e conectores, abstrair e expandir computações, guardar computações em disco, etc.

O gerador de código é usado para criar, a partir de uma computação, um programa que pode ser compilado e executado. Isso seria feito através do percurso da estru-

tura de dados que representa a computação. Como estruturas de controle como `if` e `while` inexistem na LegoShell (que é basicamente uma elaborada linguagem de configuração e visualização), os programas produzidos pelo gerador de código seriam bastante simples: uma série de declarações de objetos, correspondentes aos objetos da computação; e a conexão entre esses objetos através de funções pré-definidas na linguagem alvo.

O sistema de execução da LegoShell é a interface da linguagem com o ambiente de execução, baseado no modelo de objetos distribuídos (vide capítulo 5). Essa interface é específica para a LegoShell, e fornece funções básicas para verificação de tipos, configuração e execução de computações.

4.7 O Run Time System

O sistema OMNI é a base das ferramentas das aplicações distribuídas desenvolvidas no Projeto A_Hand. As linguagens Cm, CO^2 e LegoShell, projetadas para escrever programas de natureza inerentemente distribuídas, terão que utilizar, para tanto, as estruturas de dados e funções do OMNI para mover as informações "daqui pra lá".

Os recursos oferecidos pelo OMNI, todavia, estão em um nível de abstração incompatível com as construções das linguagens citadas. O OMNI tem funcionalidade muito geral, e o uso das suas funções pode levar ao exame de muitos detalhes que podem ser irrelevantes para um grande número de aplicações. Portanto, é necessário estabelecer um nível de abstração acima do OMNI, para preencher o *gap* semântico entre este e as linguagens de programação usuárias dos seus serviços.

Essa camada é o chamado *Run Time System*, também denominado, nesta dissertação, pelo termo mais genérico "Sistema de suporte à execução". Esse sistema tem por objetivo oferecer uma interface uniforme do "sistema distribuído" provido pelo OMNI às aplicações construídas dentro do sistema. No que diz respeito às linguagens, o *Run Time System* é genérico, oferecendo um modelo básico de objetos e comunicação. Entre as funções do *Run Time System* podemos citar:

- criação e destruição de objetos distribuídos
- encapsulamento de arquivos, dispositivos e conectores especiais
- criação e destruição de portas de comunicação
- funções para comunicação com portas
- conexão entre objetos
- representação universal de tipos
- propagação de exceções inter-processos
- funções para uso de *threads*
- mecanismos de controle de concorrência

As funções do *Run Time System* devem estabelecer uma interface clara entre o modelo de objetos distribuídos e as camadas que se utilizam dele, isto é, aplicações em geral e os compiladores/interpretadores das linguagens Cm, CO^2 e LegoShell.

Acima do *Run Time System* serão implementados módulos específicos para suporte de cada uma das linguagens.

A proposta original deste trabalho de mestrado [Gon92] previa a especificação completa do *Run Time System* e sua implementação. Entretanto, como explicado nas conclusões desta dissertação (vide capítulo 6), a completa concepção de um modelo de programação distribuída para a linguagem Cm tornou-se o objetivo desta tese. Esse modelo fundamentará a especificação do *Run Time System* e definirá a forma de construção de aplicações distribuídas no nosso ambiente. Dessa forma, as diferentes partes do sistema de suporte serão detalhadas à medida que o modelo proposto for apresentado.

5 Objetos Distribuídos em Cm

Todos os seres que têm alguma coisa em comum procuram os seus semelhantes. (...) Igualmente todo o ser que participa da comum natureza inteligente esforça-se por reunir-se ao que lhe é aparentado, e mais ainda. Com efeito, quanto mais um ser é superior aos outros mais está pronto a misturar-se e unir-se com o seu semelhante. (...) Entre os seres ainda superiores, mesmo se estão distantes, forma-se uma espécie de união, como por exemplo entre os astros. Igualmente, o esforço para se elevar a um nível superior é capaz de produzir simpatia entre os seres, apesar da distância.

Marco Aurélio

O termo "Objetos Distribuídos" representa, da melhor maneira possível, a síntese deste trabalho: uma integração das facilidades oferecidas pelos sistemas distribuídos com o modelo de programação orientada a objetos. Esse é um campo de pesquisa muito concorrido na atualidade, com perspectivas muito promissoras para áreas como ambientes de desenvolvimento de *software*, simulação, bancos de dados e *groupware*.

Este capítulo contém os resultados que este trabalho de mestrado alcançou, e que representam a contribuição do autor para a pesquisa na área de linguagens orientadas a objetos. Tais resultados consistem, basicamente, em definir o modelo de programação distribuída a ser incorporado à linguagem Cm. Além disso, as modificações que deverão ser feitas na linguagem (e conseqüentemente em seu compilador), assim como as funções que devem ser oferecidas pelo sistema de suporte à execução, são apresentadas juntamente com algumas considerações quanto à sua implementação. Ao final do capítulo, comentamos outras iniciativas de pesquisa relativas a objetos distribuídos.

Os conceitos expostos neste capítulo são acompanhados de ilustrações, com representação de classes, processos de compilação e execução, objetos e aplicações. As figuras utilizadas estão explicadas no apêndice B ("Notação").

5.1 Sistemas Distribuídos + Objetos

O artigo [Hor89] tem, sem considerar outras qualidades, um título bastante feliz: *"Is Object Orientation a Good Thing for Distributed Systems?"*. Esse título retrata o estado de espírito de muitos pesquisadores na área, que embora respondam "sim" à pergunta, não podem deixar de considerar os vários problemas envolvidos na tentativa de integrar sistemas distribuídos à programação orientada a objetos. Nesse artigo, por exemplo, são abordados aspectos como transparência, ativação de méto-

dos, objetos passivos e ativos, persistência, controle de concorrência, tolerância a falhas e desempenho.

A programação com objetos depende de uma linguagem que suporte o paradigma, e, opcionalmente, de um ambiente de programação que ofereça objetos como recursos básicos do sistema. O termo *Sistema de Programação Baseado em Objetos* é apresentado em [Chi91] como sendo o ambiente que fornece essa facilidade; se, além disso, o sistema oferece um ambiente distribuído para a construção de programas, então ele é chamado *Sistema de Programação Distribuído Baseado em Objetos*. Esses sistemas apresentam características muito atraentes, das quais destacamos:

- distribuição: composição de um ambiente de programação integrado a partir de um conjunto de máquinas autônomas e interligadas;
- transparência: usuários e aplicações devem utilizar objetos através de uma interface uniforme, independentemente da localização dos mesmos dentro do sistema;
- tolerância a falhas: problemas isolados em nós da rede ou em aplicações não comprometem irremediavelmente o sistema;
- concorrência: objetos podem ser executados simultaneamente sempre que houver disponibilidade de processadores, aproveitando o paralelismo tanto entre objetos como pelo atendimento de múltiplas requisições dentro de objetos.

De um ponto de vista conceitual, a programação distribuída é uma contribuição importante para a OOP, porque aumenta o seu poder de modelagem. Objetos devem representar entidades do mundo real, e esse mundo é inerentemente paralelo: os agentes que dele participam estão agindo simultaneamente, em contato com o ambiente e com outros agentes. Um objeto, ao representar uma entidade real, deve simular seu comportamento, o que inclui a capacidade de agir autonomamente. E de um ponto de vista prático, aplicações baseadas em objetos podem melhorar seu desempenho em ambientes distribuídos pelo melhor aproveitamento de elementos processadores.

Muitas linguagens de programação orientadas a objetos não permitem programação distribuída ou concorrente. Entre essas linguagens incluem-se exemplares importantíssimos, como Smalltalk, C++ e Eiffel. Pesquisas nessa área seguiram, no que se refere ao projeto e implementação de linguagens, duas abordagens: o desenvolvimento de linguagens novas (e.g., Emerald); e a extensão de linguagens já existentes. Mais adiante, na seção 5.7, página 5-51, comentaremos essas iniciativas.

5.1.1 Como modificar as linguagens de programação?

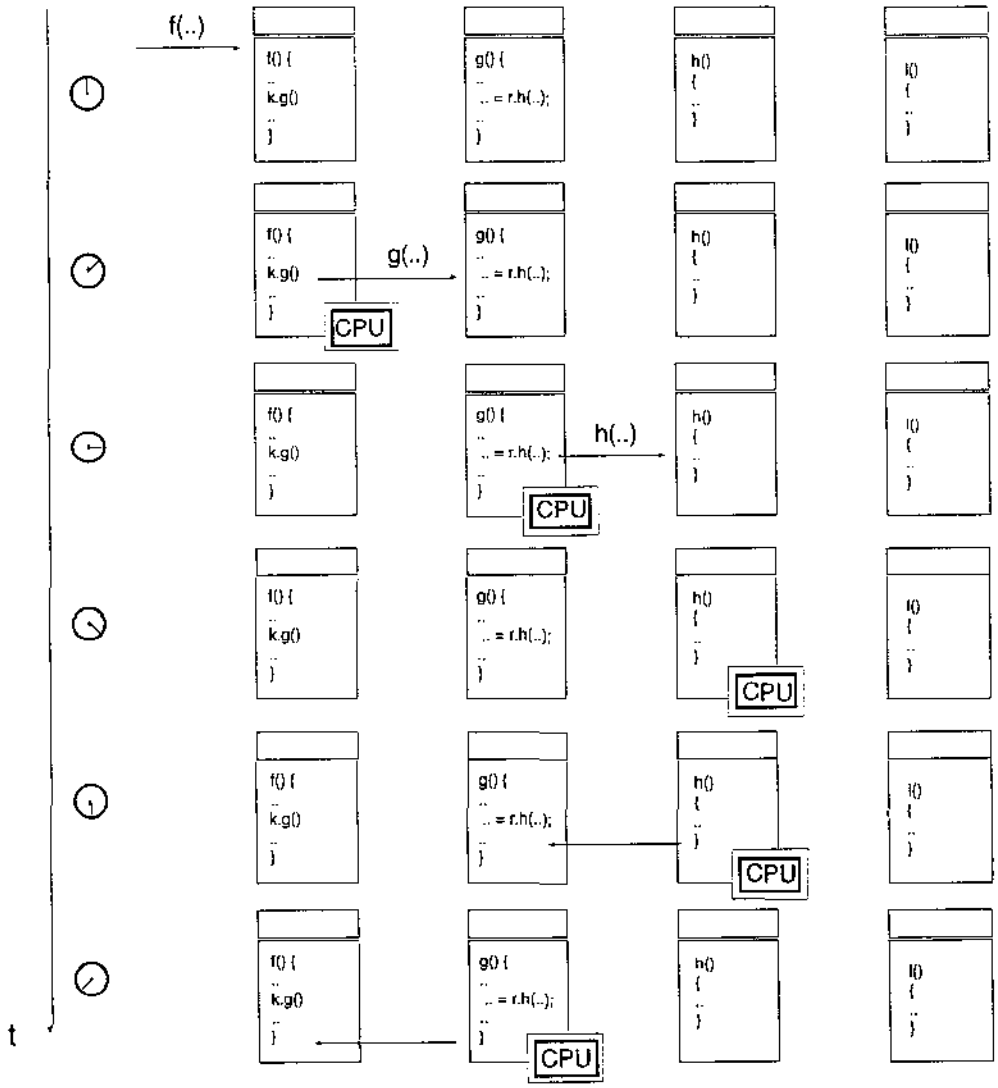
Este trabalho descreve a definição de um mecanismo de objetos distribuídos a ser implementado em uma linguagem de programação orientada a objetos que, atualmente, é baseada no modelo seqüencial de execução. Essa alteração da linguagem depende de extensões à linguagem para expressar a natureza distribuída de programas, e de suporte mínimo do ambiente de execução para criação e comunicação entre objetos distribuídos. A base para o nosso trabalho é o sistema OMNI,[1] sobre plataforma UNIX.

5.1.1.1 Modelos de execução de objetos

Um objeto é composto de dados privados e de operações que usam esses dados. Ao receber uma mensagem, o objeto deve executar a operação pedida na mensagem, através de computações sobre seus próprios dados e, opcionalmente, requisitando operações a outros objetos. Um objeto, ao executar uma operação, pode dar início a uma seqüência de chamadas de métodos entre objetos, semelhante a uma seqüência de chamada de funções dentro de um programa. O objeto que recebeu uma mensagem mais recentemente está executando, enquanto que o objeto que enviou essa mensagem está bloqueado à espera de uma resposta; assim estão também todos os demais objetos da seqüência de chamadas.

1. Para mais detalhes, consultar a seção 4.5 ("O Sistema OMNI").

FIGURA 5-1 Objetos sendo executados de forma seqüencial



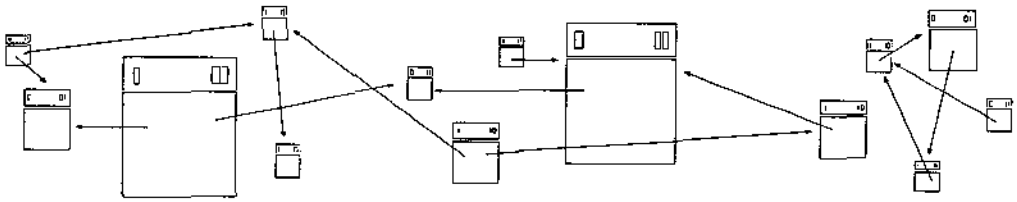
Nesse modelo, um objeto não decide executar operações autonomamente, e sim apenas para responder a mensagens. Tais objetos são, portanto, de natureza passiva. Conforme [Bal89], pode-se passar de um modelo de execução seqüencial de objetos para um modelo paralelo das seguintes maneiras:

- um objeto pode estar ativo sem ter recebido uma mensagem;
- um objeto pode continuar ativo após devolver o resultado de uma mensagem;
- um objeto pode enviar uma mesma mensagem a mais de um objeto simultaneamente;

- um objeto pode continuar ativo enquanto espera pelo resultado de uma mensagem.

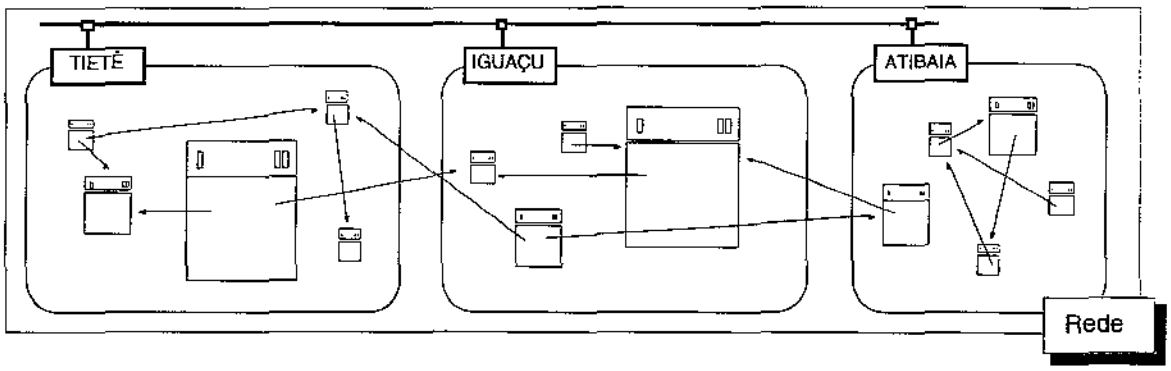
Num modelo de execução paralelo, vários objetos podem estar ativos e várias mensagens podem ser enviadas num mesmo momento. O desempenho das aplicações é melhorado através do paralelismo na execução dos vários objetos, que ficam bloqueados apenas por necessidades de sincronização próprias da aplicação, e não por uma limitação imposta pelo modelo de execução.

FIGURA 5-2 Objetos sendo executados de forma paralela



Em um modelo distribuído, o conceito de transparência passa a ser importante, possibilitando que os objetos sejam utilizados de maneira uniforme independentemente de seu estado ou de sua localização em um dado momento.

FIGURA 5-3 Objetos sendo executados de forma distribuída



A conseqüência da adoção de um modelo de programação distribuída, para um projetista de uma linguagem de programação, leva a duas perguntas: como definir uma nova linguagem de programação? Ou então, como modificar uma linguagem de programação, orientada a objetos e baseada no modelo seqüencial de execução,

para permitir a construção de programas compostos por objetos executados em um ambiente distribuído?

O nosso trabalho utiliza como base a linguagem Cm, que já foi definida e implementada [Dru88, Sil88, Fur91, Tel93]. Portanto, o que nos interessa é como estendê-la para permitir programação distribuída, de maneira a aumentar seu poder de expressão e, ao mesmo tempo, preservar suas características.

5.1.2 Requisitos para modificar uma linguagem

Quando se fala em ambientes de programação distribuída, o conceito fundamental é transparência. Nesses ambientes, os recursos são usados sem que seus usuários estejam conscientes de como esses recursos estão divididos entre as partes do sistema. Em uma aplicação composta por objetos, a interação entre os objetos (que encapsulam os recursos) se dá através do envio de mensagens (mecanismo para a chamada de métodos), portanto é nesse aspecto que a transparência deve ser considerada.

Exemplificando, para quem usa um objeto *obj*, cuja classe é *A*, com um método *m()* exportado, a ativação desse método com uma chamada da forma

obj.m();

deve ser feita sem levar em consideração onde está o objeto representado pela variável *obj*. Essa é a forma tradicional de chamada de métodos, que deve ser usada, com a mesma sintaxe, para o modelo distribuído.

A semântica de uma chamada dessa forma é alterada no sentido de que o objeto "alvo" pode estar em outro ponto do sistema distribuído, em relação ao objeto que faz a chamada. Não há perda de transparência porque a sintaxe é idêntica, assim como o efeito da chamada (discutiremos isso em detalhes mais adiante). Pode haver diferença na declaração desses objetos (como é o caso desta proposta), já que no momento da declaração é importante destacar a diferença entre objetos que serão utilizados de forma seqüencial e os objetos que serão utilizados de forma concorrente ou distribuída.

Em [Mey93], é apresentada uma série de requisitos a serem cumpridos para estender uma linguagem orientada a objetos para expressar computação distribuída; destacamos os seguintes:

- extensão mínima: as alterações feitas na linguagem não podem levar a um resultado final muito complexo ou muito diferente da linguagem original;
- compatibilidade com técnicas de OOP: mecanismos de programação como herança e polimorfismo não devem ser restringidos no contexto distribuído;
- possibilidade de prova de programas: a necessidade de provar a correção de programas é muito mais complexa e muito mais importante no caso distribuído;
- reusabilidade de *software* não concorrente: classes já escritas, testadas e depuradas devem ser reaproveitadas com esforço mínimo de adaptação.

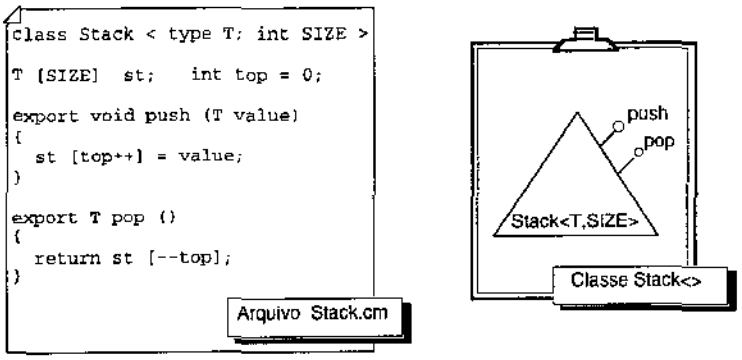
5.2 Proposta de Objetos Distribuídos em Cm

O Cm é a linguagem de programação básica do Projeto A_Hand, e nela será incorporado o conceito de objetos distribuídos [Dru93, Dru94]. As outras linguagens, CO^2 e LegoShell, vão permitir a construção de aplicações distribuídas utilizando os mecanismos básicos de suporte ao uso de objetos distribuídos em Cm. Esses mecanismo básicos são: criação de objetos distribuídos, comunicação transparente entre objetos, representação universal de tipos, propagação de exceções, portas de comunicação e concorrência dentro de objetos. Cada um destes tópicos é discutido em detalhe nas próximas seções.

5.2.1 Exemplos

As extensões sendo propostas serão exemplificadas através de pequenos trechos de código e de diagramas representando objetos e suas interações. Na maior parte dos casos, será usada a classe *Stack*, usada para gerar pilhas simples. Apresentamos essa classe aqui e, onde necessário, ela será reapresentada com eventuais modificações.

FIGURA 5-4 A classe *Stack<>*



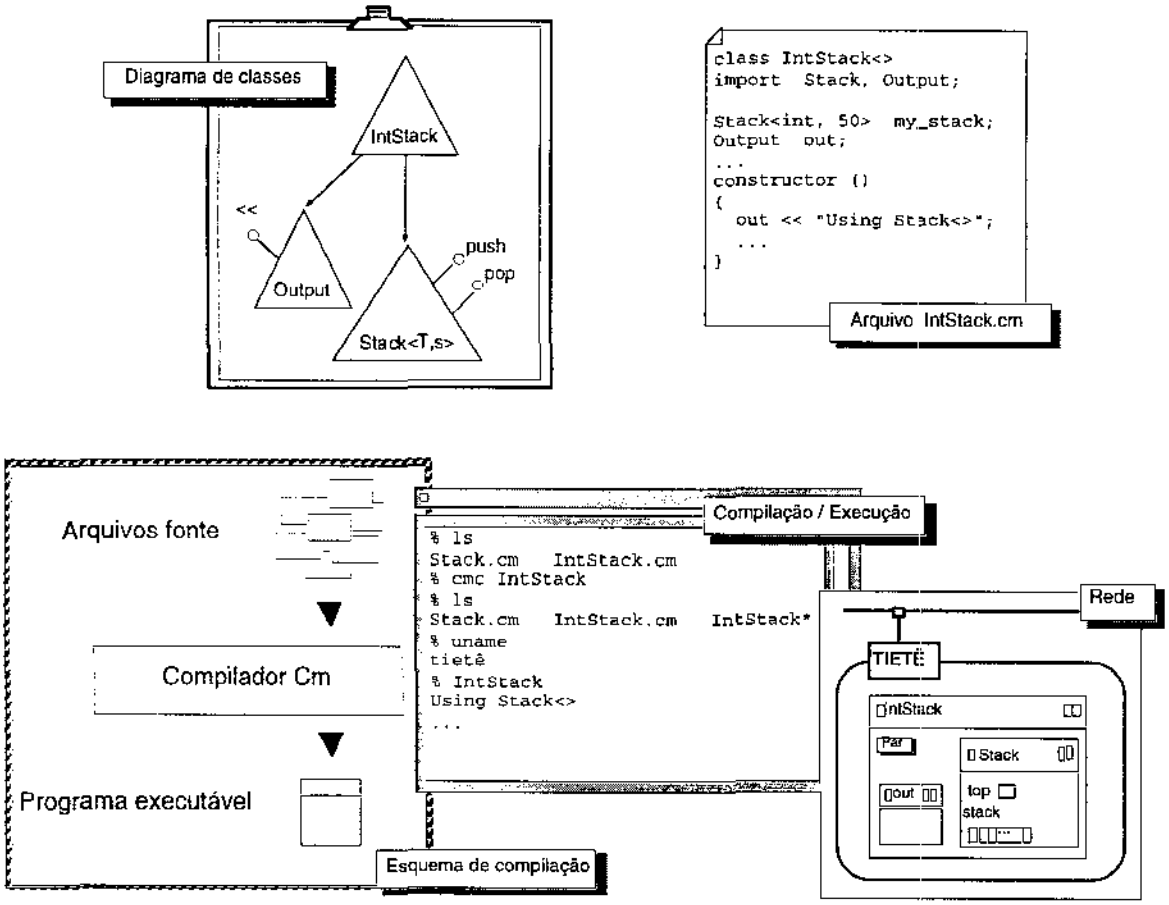
5.2.2 Programando em Cm

Um programa Cm pode ser descrito como uma hierarquia de classes; a compilação da classe que está no topo da hierarquia (que pode ser chamada de classe principal) causa a compilação de todas as outras classes da hierarquia,[1] das quais a classe principal depende, direta ou indiretamente, por importação ou herança. O resultado da compilação é um programa executável, cujo ponto de entrada é a ativação do

1. O compilador Cm faz, automaticamente, uma análise das classes envolvidas para verificar se existe, para cada uma delas, uma versão já compilada; isso evita compilação desnecessária de classes. A frase do texto omite essa explicação a bem da clareza.

construtor da classe principal, seguida da execução do método *main()* do objeto construído. A partir daí, o fluxo de controle passa pelos objetos contidos no programa.

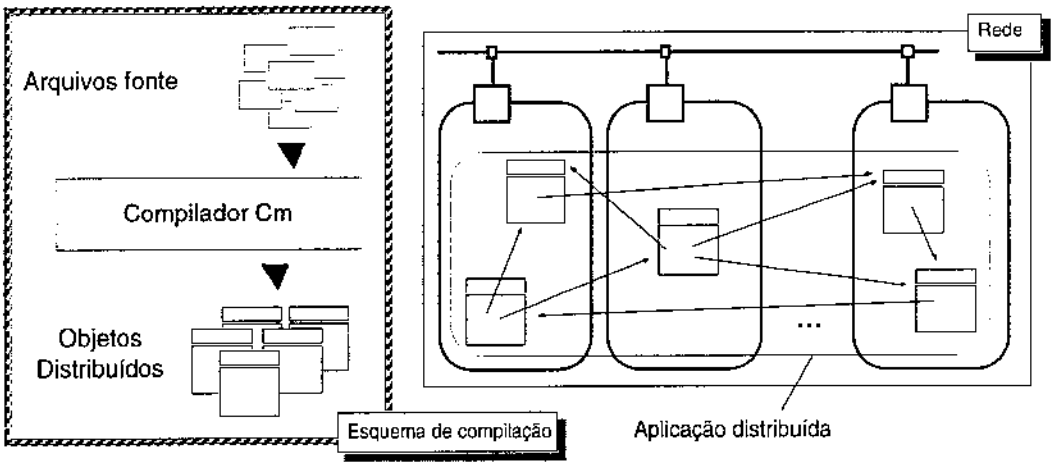
FIGURA 5-5 Mecanismo de construção de programa em Cm



Como a execução é estritamente seqüencial, todos os objetos são servidos por um único fluxo de controle, e os dados e código dos objetos estão agrupados dentro de um único programa executável.

Um programa composto de objetos distribuídos deverá ser representado, em tempo de execução, por vários objetos, que se comunicam para pedir a execução de métodos e receber os resultados correspondentes. Tais programas são denominados genericamente de aplicações distribuídas baseadas em objetos, ou simplesmente aplicações distribuídas.

FIGURA 5-6 Mecanismo de construção de aplicações distribuídas



Uma aplicação distribuída típica em "Cm Distribuído" é, portanto, composta de um conjunto de objetos executando autonomamente, e solicitando uns aos outros a execução de operações. Tais operações estão implementadas nos métodos do objeto, que na declaração da classe podem ser colocados à disposição de outros objetos. Esses métodos fazem parte da interface da classe, e nós os denominamos *serviços*. Um objeto que oferece serviços é um *servidor*, e um objeto que utilizar esses serviços é chamado *cliente*. Uma característica importante do modelo de programação distribuída da linguagem Cm é que não há diferença conceitual entre servidores e clientes: qualquer objeto que ofereça operações através de sua interface é um servidor, e qualquer objeto que utilize serviços de outros é um cliente.

5.2.3 Objetos Remotos

Programação distribuída em Cm é possível através de *objetos remotos*. Um objeto é dito remoto se ele tem um contexto de execução separado do contexto do objeto que o criou. Esse contexto de execução é o conjunto de informações referentes à execução de um objeto dentro do sistema: é composto essencialmente pelo espaço de endereçamento associado à execução do objeto, ou seja, onde estão guardados seus dados e código.

Definição 5.1. Um objeto é *remoto* se tem um contexto de execução próprio, separado do contexto do objeto que o criou e do contexto de seus objetos clientes.

A primeira e mais importante implicação do fato de que os objetos remotos são executados em contextos diferentes é que esses objetos podem ser executados em paralelo. As aplicações distribuídas são feitas pela criação e composição de objetos remotos, a partir da criação de um "objeto principal". Objetos remotos podem criar

outros objetos remotos, estabelecendo relações de "paternidade" semelhantes àquelas existentes entre processos UNIX.

Definição 5.2. Um objeto que cria um objeto remoto é denominado *objeto pai*, e o objeto criado é denominado *objeto filho*.

Se um objeto cria um objeto remoto na mesma máquina, o fato de que ambos estejam sendo executados pela mesma unidade processadora significa apenas uma limitação do paralelismo (que é inerente aos objetos remotos) imposta pela indisponibilidade de recursos (no caso, processadores). Esses dois objetos são considerados, um em relação ou outro, objetos remotos, mesmo sendo executados na mesma máquina, pois o conceito é baseado no contexto de execução, que é uma abstração da máquina real.

Um objeto remoto em Cm é criado por declaração, assim como os objetos comuns, que não são remotos. A distinção entre um e outro caso é dado pela palavra reservada `remote`, que é aplicada à classe do objeto indicando que se trata da declaração de um objeto remoto.

Exemplo 5.1. Suponhamos uma classe *C*, que importa uma classe *S*. Um objeto da classe *C* cria um objeto remoto da classe *S* através da declaração

```
S remote obj;
```

após o que os objetos pai e filho são executado em contextos diferentes.

Podemos ilustrar melhor a definição usando a classe *Stack*, mostrada na figura 5-4, e a classe *FloatStack* definida abaixo.

FIGURA 5-7 Classe *FloatStack<>*

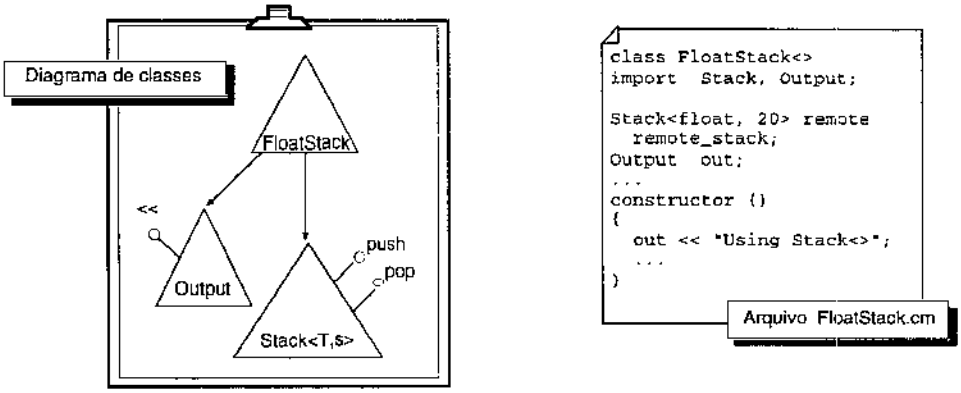
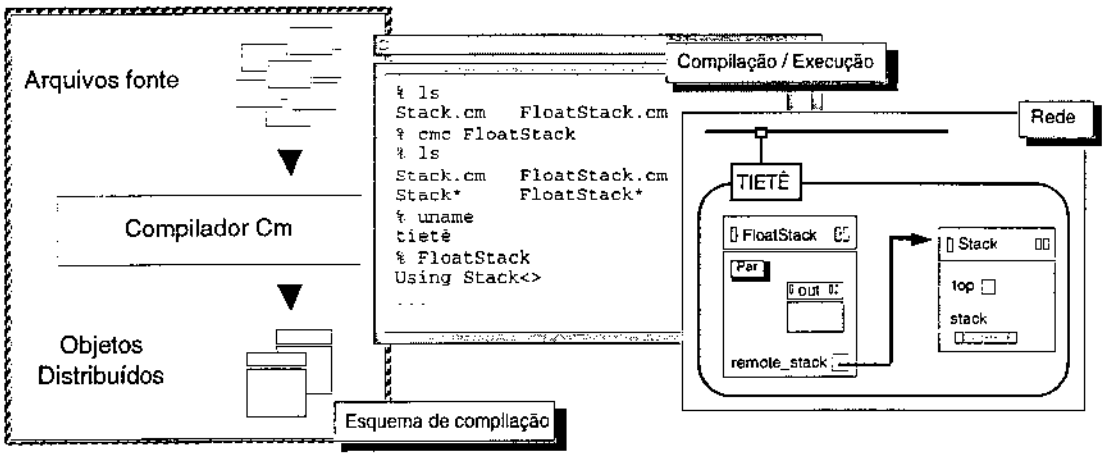


FIGURA 5-8 Exemplo de criação de um objeto remoto



(Fim do exemplo 5.1.)

Outra decorrência importante da programação com objetos remotos é a de que um objeto remoto pode ser usado por outros objetos além do seu objeto pai. Isso porque, no exemplo 5.1, o identificador *obj* é apenas um símbolo local do objeto pai, ao passo que o objeto remoto que ele representa é uma entidade que existe fora do contexto do objeto pai. A idéia é poder associar, dentro de outros objetos, identificadores locais a objetos remotos já existentes. Os objetos remotos têm uma identificação única, independentemente de seus atributos ou estrutura e de sua localização; tal identificação distingue esse objeto de todos os demais objetos remotos. Mais adiante, na seção 5.2.5 ("Objetos remotos compartilhados"), mostraremos como um objeto remoto pode ser utilizado por vários objetos.

Do ponto de vista semântico, a palavra reservada `remote` é um construtor de tipos, aplicável apenas a tipos classe: não há sentido em declarar uma variável inteira como um objeto remoto. Como os demais construtores de tipos Cm, aplica-se da direita para a esquerda: embora a declaração "remote S obj" pareça mais adequada do ponto de vista da sintaxe da língua inglesa, a forma correta é o "adjetivo" (`remote`) após o "complemento" (nome da classe).

Um objeto remoto, portanto, é de tipo incompatível com o tipo de um objeto da mesma classe que não seja remoto. Não pode ser feita passagem de parâmetros, atribuição ou comparação envolvendo objetos remotos e objetos não remotos, mesmo que eles sejam da mesma classe.

Depois que um objeto remoto é criado, os componentes da sua interface (variáveis e métodos exportados) estão disponíveis para uso. Conforme dito anteriormente, a sintaxe para uso dos elementos da interface de um objeto remoto é a mesma que para os demais objetos.

Exemplo 5.2. Vamos supor as seguintes declarações na classe *S*:

```
class S<>
...
export int int_var;
...
export void void_f() { ... }
...
export int int_f() { ... }
```

Na classe C, esses símbolos exportados são usados normalmente

```
class C<>
import ... S, ...;
...
S remote obj;
int j,k;
...
j = obj.int_var ++;
obj.void_f();
k = j + obj.int_f();
...
```

(Fim do exemplo 5.2.)

As operações sobre as variáveis da interface são leitura, atribuição, os operadores previstos pela linguagem Cm (comparação, operações aritméticas, etc.) e os operadores superpostos porventura definidos e aplicáveis a essas variáveis. É proibido, entretanto, o uso do operador & (endereço), já que isso implicaria manipular, dentro de um objeto, um endereço pertencente a outro contexto de execução; por extensão, essas variáveis também não podem ser usadas como parâmetros passados por referência em chamadas de métodos.

A utilização de uma variável presente na interface de um objeto remoto será, muito provavelmente, bastante ineficiente, já que isso representa informação sendo transferida de um contexto para outro. A simples leitura de uma variável pode significar a transmissão de duas mensagens pela rede, o que exige do programador o uso muito criterioso dessa alternativa. De qualquer forma, a disponibilidade de variáveis nas interfaces de objetos é prática condenável em todo caso, pois isso viola o princípio de encapsulamento.

Definição 5.3. Um *método remoto* é um método a ser executado em um objeto remoto; um pedido para a execução de um tal método é denominado *ativação* ou *chamada remota de método.*

Uma chamada remota é sempre síncrona, a exemplo de uma chamada não remota. Essa regra é válida mesmo que o método remoto não devolva nenhum resultado (método de tipo `void`, ou "função" utilizada como "procedimento"). Embora o mecanismo de chamada assíncrona seja mais geral, há razões que nos fizeram optar pela variante síncrona.

Em primeiro lugar vem a questão de prova formal de programas, e mais abstratamente uma forma de raciocinar a respeito da execução e depuração. A chamada assíncrona leva à seguinte situação: no ponto imediatamente após a chamada remota, não se pode definir um predicado que represente o estado da computação naquele ponto, já que o objeto pode estar executando a chamada, ou já tê-la completado ou nem mesmo tê-la recebido. Dessa forma, se durante uma seqüência de chamadas endereçadas a um mesmo objeto remoto, acontecer uma falha envolvendo esse objeto (*crash* do objeto ou da máquina, problema no meio de comunicação, etc.) não há como saber quais chamadas foram completadas e quais ficaram pendentes. Em um objeto distribuído, essa pode ser uma situação bastante comum.

Nessa mesma linha de raciocínio, a variante síncrona preserva a semântica da chamada de método, bloqueando o objeto chamador independentemente do *status* do objeto chamado (remoto ou não). Como o uso de exceções é possível no caso de chamadas remotas,[1] isso é necessário mesmo que o método remoto não devolva nenhum resultado, pois ele sempre pode "devolver" uma exceção. No caso de chamadas assíncronas, seria preciso deter o fluxo de execução quando esse estivesse saindo de um comando protegido onde está a chamada remota, porque senão uma exceção no objeto remoto cairia num contexto errado.

A variante assíncrona provavelmente exigiria alguma nova construção sintática, diferente da chamada síncrona, ou então o uso de objetos de uma classe pré-definida, para tratar as chamadas como dados. Isso traria mais complexidade para a linguagem e, mais importante, violaria o princípio de transparência.

A chamada síncrona significa, em teoria, limitação do paralelismo, já que não permite que o fluxo de execução prossiga após o envio da mensagem ao objeto servidor, tornando o objeto cliente inativo até o recebimento da resposta. Entretanto, na nossa proposta está prevista concorrência dentro de objetos, através da execução de vários fluxos de execução simultâneos; dessa forma, um fluxo de execução que faz uma chamada remota é bloqueado, liberando o processador para outro fluxo de execução do mesmo objeto ou de outro objeto. A concorrência dentro de objetos em Cm está apresentada na seção 5.6.

A declaração de objetos remotos também pode se valer de construtores, para determinar o estado inicial do objeto após sua criação.

Exemplo 5.3. Considerando o exemplo 5.2, podemos ter as seguintes declarações de objetos remotos

```
S remote obj_1 @(10), obj_2 @("plop"), obj_3 @(128, '\n');
```

que são válidas se a classe *S* oferecer os três construtores com os parâmetros mostrados. (Fim do exemplo 5.3.)

1. Assunto discutido na seção 5.3 ("Exceções em objetos distribuídos")

Uma classe, depois de sofrer a aplicação do construtor de tipos `remote`, é um tipo Cm válido, que pode ser usado como outro tipo qualquer, como componente de estruturas ou elemento de vetor.

Exemplo 5.4. Utilizando a mesma classe *S* dos exemplos anteriores

```
S remote [] array_S = { @(0), @(1), @(2) };

struct {
   S remote field_S;
   int q;
} str_S_int;

type remoteS = S remote;

S remote ftS ( ... )
{ ... }
```

(Fim do exemplo 5.4.)

Objetos remotos também podem ser criados dinamicamente, a partir da função new, e depois destruídos com a função `release`.

Exemplo 5.5. Utilizando a mesma classe *S* dos exemplos anteriores

```
S remote * ptr_S;
...
ptr_S = new (S remote) @(17, '$');
...
release ptr_S;
```

(Fim do exemplo 5.5.)

5.2.4 Objetos passivos e objetos ativos

Um objeto é composto por um conjunto de dados e por operações que usam esses dados. Como já vimos anteriormente, num modelo seqüencial os objetos estão à espera de alguma mensagem, para então executar uma operação. Já no modelo paralelo, os objetos podem estar, a qualquer momento, executando alguma tarefa, sem que isso seja resultado da recepção de uma mensagem.

Vamos supor uma classe que implementa uma pilha. Tipicamente essa classe exporta duas funções, *push()* e *pop()*, para inserir um elemento no topo da pilha e para retirar da pilha o último elemento inserido, respectivamente. Os elementos são guardados em uma estrutura de dados, como um vetor ou uma lista, e há um apontador que indica onde está o topo da pilha. As funções de inserção e retirada são, meramente, inserir ou retirar um elemento dessa estrutura e atualizar o apontador de topo de pilha. Para tal objeto, ser executado de forma seqüencial ou paralela não faz diferença, pois não há o que fazer além das operações de inserção e retirada: nos intervalos entre chamadas, o objeto está ocioso.

Vamos supor, por outro lado, a implementação de uma classe que manipula números primos. Essa classe oferece duas funções: uma para verificar se um determinado número, passado como parâmetro, é ou não primo; e outra função que devolve o i-ésimo número primo, a partir de um parâmetro de valor i. Um objeto dessa classe seria usado, por exemplo, em uma aplicação de criptografia, onde geralmente os número manipulados são muito grandes (da ordem de 10^{100} ou 10^{200}) . Testar se um número muito grande é ou não primo, e gerar primos muito grandes, são ambas tarefas computacionalmente caras, pois envolvem grande número de cálculos. As funções fornecidas por essa classe, portanto, podem consumir muito tempo de processamento, às custas das aplicações que as utilizam.

Um objeto dessa classe poderia se beneficiar de um modelo paralelo de execução. Durante os intervalos entre chamadas, esse objeto permaneceria ativo, calculando números primos em seqüência e armazenando-os em uma estrutura de dados para consulta rápida. Em determinado momento, vamos supor que tenha sido calculado o k-ésimo primo, cujo valor é N_k: se é feita uma consulta para saber se um número N é primo, a resposta pode ser bastante rápida se $N < N_k$, pois nesse caso basta fazer uma busca na relação de primos já calculados; caso contrário, a consulta deve esperar até que seja calculado um primo maior ou igual a N.

Um objeto da classe pilha, portanto, é apenas uma estrutura de dados, que pode ser usada por vários objetos para troca de informações. Já um objeto da classe que fornece operações com números primos representa uma parte ativa de uma aplicação: mesmo que não haja demanda num dado momento, ele está continuamente produzindo resultados, que estão prontos para serem usados e que também são úteis para produzir novos resultados. Baseados na natureza e no comportamento desses objetos, podemos dizer que uma pilha é um objeto *passivo*, e que um servidor de números primos é um objeto *ativo*.

Definição 5.4. Um objeto é *ativo* se ele possui o método *main()*, e é *passivo* no caso contrário.

Quando um objeto remoto é criado, em primeiro lugar é executado seu construtor, e nesse momento seu objeto pai está bloqueado. Depois de executado o construtor, o objeto pai é liberado; se o objeto filho possui o método *main()*, este é posto para executar, sem que haja necessidade de uma chamada explícita no objeto pai. Na verdade, os clientes de um objeto (e entre eles o próprio objeto pai) não precisam, em princípio, saber se esse objeto é ativo ou passivo.

5.2.5 Objetos remotos compartilhados

As aplicações distribuídas escritas em Cm são compostas por um conjunto de objetos cooperantes. Um destes objetos vê seus servidores como objetos remotos, mesmo que não seja o pai de nenhum deles. Isso é possível já que a linguagem Cm permite associar uma variável dentro de um objeto a um objeto remoto já existente.

As informações sobre um objeto remoto estão guardadas no seu objeto pai, como efeito de sua criação. Para que esse objeto remoto possa ser usado por outros objetos, além do pai, é necessário que esses clientes tenham uma referência para ele, que deve ser fornecida pelo objeto pai.

Uma maneira de tornar um objeto conhecido fora do objeto pai é associá-lo, no instante da sua criação, a um nome simbólico. Essa operação é suportada pelo Servidor de Nomes que faz parte do ambiente de execução.[1]

Exemplo 5.6. A declaração

```
S remote obj_S as "Object Class S";
```

causa o cadastramento, no Servidor de Nomes, do objeto remoto criado, com o nome simbólico "Object Class S". A partir desse momento, outros objetos podem ter acesso ao objeto recém-criado, a partir de uma consulta ao Servidor de Nomes, como será exemplificado mais adiante. (Fim do exemplo 5.6.)

Na criação de um objeto remoto é possível, ainda, dar atributos de visibilidade e unicidade ao nome simbólico associado, e especificar a máquina da rede onde o objeto será criado e executado. Tais atributos alteram os procedimentos de cadastramento e busca de nomes no Servidor. A destruição do objeto[2] causa o cancelamento do seu nome simbólico.

Exemplo 5.7. As seguintes declarações ilustram a criação de objetos remotos com nomes simbólicos e atributos

```
S remote obj1_S as "Object 1 Class S" global unique;
...
S remote obj2_S at "tietê";
...
S remote obj3_S as "Object 3 class S" at "atibaia" unique;
```

O objeto representado por *obj1_S* é criado na mesma máquina do objeto pai, com nome simbólico "Object 1 class S"; esse nome simbólico é único e global, o que significa que a criação do objeto só é bem sucedida se não houver nada registrado com esse mesmo nome em toda a rede, e após a declaração não haverá, até que o objeto seja destruído, nada com esse mesmo nome.

O objeto representado por *obj2_S* é criado na máquina Tietê, sem nome simbólico; e o objeto representado pela variável *obj3_S* é criado na máquina Atibaia, com nome simbólico "Object 3 class S", que é único nessa máquina. (Fim do exemplo 5.7.)

Um objeto pode utilizar como servidor um objeto remoto criado por um terceiro objeto. Para tanto, deve declarar uma variável que é uma espécie de referência para um objeto remoto, e mais tarde associar essa variável ao objeto desejado. A declaração dessa variável não pode ser feita usando o construtor de tipos `remote`, pois isso causaria a criação de um novo objeto remoto. O construtor de tipos `unattached`, utilizado em lugar de `remote`, tem o efeito de criar uma variável do tipo do objeto remoto desejado mas que inicialmente tem valor indefinido.

1. Para detalhes vide subseção 4.5.1.1 ("Servidor de Nomes").
2. Ver detalhes na seção 5.2.6 ("Tempo de vida de objetos remotos").

Exemplo 5.8. Dentro de uma classe P, a declaração

```
S unattached my_obj_S;
```

causa a criação da variável *my_obj_S*, cujo tipo é a classe *S*, e cujo valor é indefido. (Fim do exemplo 5.8.)

Como não há a criação de um objeto, não faz sentido o uso de construtores. Uma variável declarada com o construtor `unattached` não pode ser utilizada enquanto não for associada a um objeto remoto. Para que essa associação seja feita, o objeto onde a variável foi declarada deve obter uma referência para o objeto remoto que se quer utilizar. Essa referência pode ser, por exemplo, o resultado de uma consulta ao Servidor de Nomes, se o objeto remoto foi criado com nome simbólico. A associação é feita pela função *attach()*, que é pré-definida na linguagem.

Exemplo 5.9. Usando os exemplos 5.6 e 5.8, o comando

```
attach (my_obj_S, "Object Class S");
```

associa, à variável *my_obj_S*, o objeto cadastrado no Servidor de Nomes com o nome simbólico "Object Class S". Depois de feita essa associação, *my_obj_S* passa a representar um objeto remoto da classe *S*, e está disponível para ser usado. (Fim do exemplo 5.9.)

A função *attach()* ainda oferece duas outras maneiras de se fazer a associação, que diferem pelo tipo do 2° parâmetro, a referência ao objeto remoto. As alternativas para esse 2° parâmetro são: a identificação do objeto remoto atribuída pelo sistema de suporte à execução; e uma variável ou parâmetro declarados com o construtor `remote`.

Exemplo 5.10. As declarações

```
S unattached my_obj1_S, my_obj2_S;
```

criam variáveis que serão associadas a objetos remotos conforme os comandos

```
...
ObjectId oid;
...
oid = ...   // obtém a identificação de um objeto remoto
...
attach (my_obj1_S, oid);
...
void f( ..., S remote par_S, ...)
{
   ...
   attach (my_obj2_S, par_S);
   ...
```

sendo que variável *oid* é do tipo pré-definido *ObjectId* e armazena a identificação única de um objeto remoto. Esse tipo pré-definido, e muitos outros, estão no

Apêndice C ("Alterações na Linguagem"), onde são apresentadas, mais formalmente, as modificações da linguagem propostas por este trabalho. (Fim do exemplo 5.10.)

No caso da associação feita através de um nome simbólico ou de uma *ObjectId*, é preciso verificar se o objeto remoto a ser associado e a variável declarada como `unattached` têm tipos compatíveis, já que nomes simbólicos e OMNIIds podem ser usados para recuperar objetos de qualquer tipo. Essa verificação está a cargo do sistema de suporte da linguagem, que usa uma representação universal de tipos do ambiente. Essa questão é abordada na seção 5.5.2.3 ("Associação de objetos remotos utilizando o Servidor de Nomes").

A associação de variáveis (declaradas como `unattached`) com objetos remotos compatíveis também será possível através da LegoShell. Podemos considerar uma associação destas como um procedimento de configuração do objeto, e nesse particular a LegoShell deverá oferecer mais recursos que os disponíveis na linguagem Cm. A configuração de objetos pela LegoShell é baseada na conexão de portas de comunicação, mas a linguagem deverá, a partir dessa proposta, incorporar também o modelo de objetos distribuídos.

Uma vez que uma variável esteja associada a um objeto remoto, essa associação pode ser desfeita através da função *detach()*, para ser refeita depois, com algum outro objeto remoto.

Exemplo 5.11. O comando

```
detach (my_obj_S);
```

faz com que a variável *my_obj_S* tenha novamente valor indefinido; até que uma nova associação seja feita, através da função *attach()*, ela não pode ser utilizada (Fim do exemplo 5.11.)

5.2.6 Tempo de vida de objetos remotos

Os objetos remotos são criados e mantidos dentro de um contexto de execução separado do contexto do objeto pai, onde são representados por identificadores locais. Os objetos ficam à espera de mensagens (e também executando o método *main()* caso sejam ativos), até o momento da sua destruição. Um objeto é criado por uma declaração dentro do objeto pai, e seu tempo de vida está, em princípio, vinculado ao tempo de vida do identificador correspondente no objeto pai.[1]

Exemplo 5.12. Supondo os seguintes trechos de código da classe *M*

```
class M<>
```

1. Nesse caso, não consideramos objetos remotos criados e destruídos dinamicamente, através das funções `new` e `release`, porque nesses casos o tempo de vida desses objetos é inteiramente arbitrário, dependendo de como se conduz a execução do programa. Discutimos isso alguns parágrafos mais adiante.

```
import ...S, ...;
...
S remote obj_S_level0;
...
void f(...)
{
S remote obj_S_level1;
...
   if (...) {
      S remote obj_S_level2;
      ...
   }
   ...
}
...
```

os objetos remotos criados têm diferentes tempos de vida. O objeto representado por *obj_S_level0*, por exemplo, tem tempo de vida praticamente igual ao de seu objeto pai, já que a declaração dessa variável está no escopo da classe *M*. Cada ativação do método *f* causará a criação de um objeto remoto representado pela variável *obj_S_level1*; tais objetos serão destruídos ao fim da execução do método. E um objeto remoto representado por *obj_S_level2* será criado sempre que o fluxo de execução passar pela primeira alternativa do comando `if`, e será destruído ao fim desse bloco. (Fim do exemplo 5.12.)

Entretanto, pode ser interessante que um objeto remoto seja criado e tenha um tempo de vida maior que previsto no caso geral, ou seja, que seu tempo de vida seja desvinculado do tempo de vida do identificador cuja declaração causou sua criação. Tal objeto remoto poderia, inclusive, sobreviver à destruição de seu objeto pai; nesse caso, como esse objeto poderia viver por tempo indeterminado, ele só pode ser destruído explicitamente, por algum outro objeto ou por si próprio.

É natural que seja o objeto pai quem decida se um objeto filho deve ou não ser "mais durável", pois se o filho deve sobreviver à morte do pai, é necessário que este, "ainda em vida", deixe a outros objetos pelo menos uma referência para o filho. Do contrário, o identificador local ao objeto pai que representa o filho será a única maneira de referenciar este, e a destruição do pai causará a perda dessa informação. A criação de um objeto remoto que tenha um tempo de vida desvinculado do objeto pai é feita com o construtor de tipo `autonomous`.

Definição 5.5. Um objeto remoto é denominado *autônomo* se tiver tempo de vida arbitrário, obtido através de criação com o construtor de tipos `autonomous`.

O construtor de tipo `autonomous` contém, implicitamente, um construtor de tipo `remote`, já que um objeto só pode ser autônomo se for remoto. Isso torna redundante a aplicação consecutiva dos dois construtores de tipo.

Exemplo 5.13. A seguinte declaração

```
S autonomous obj_S;
```

cria um objeto remoto da classe S cujo tempo de vida é indeterminado. Esse objeto só poderá ser destruído explicitamente, já que não existe mais vínculo entre o seu tempo de vida e o da variável *obj_S*. (Fim do exemplo 5.13.)

No caso de objetos ativos, o término normal do método *main()* não implica na destruição automática do objeto; a partir desse momento, ele continua disponível para continuar executando métodos a pedido dos objetos clientes. Isso é uma importante diferença entre processos e objetos,[1] já que um processo pode ser pensado, *grosso modo*, como um "objeto com apenas um método", e que o término desse "método" significa, simplesmente, o término do processo. Já um objeto provê várias operações, e diferentes execuções de cada uma podem ser independentes enquanto o objeto existir. Caso um objeto decida, por conta própria, que não "quer", não "deve" ou não "pode" mais existir, ele pode se destruir explicitamente, por uma chamada para a função *kill()*.

Objetos remotos podem ser destruídos através da função *kill()*, que é pré-definida na linguagem. Como parâmetro, a função aceita um objeto remoto, que será destruído se o objeto que chamar a função tiver permissão para tal. Um objeto tem permissão para destruir outro se ambos pertencem à mesma "árvore genealógica" e o objeto a ser destruído é "descendente" do outro (já que o pai do objeto a ser destruído pode não existir mais). A chamada de *kill()* sem parâmetros causa a destruição do próprio objeto chamador.

No caso da criação dinâmica de objetos remotos, estes também têm um tempo de vida indeterminado, pois são criados com a função new e destruídos com a função `release`, e o instante da execução dessas funções depende do comportamento do programa. Suponhamos que, para um objeto remoto criado dinamicamente, a função `release` não tenha sido aplicada, e o escopo correspondente à variável que representava esse objeto já esteja encerrado. Essa situação mais parece um erro e menos uma tentativa de criar um objeto autônomo. Portanto, é necessário evitar que os objetos remotos criados dinamicamente vivam mais tempo que o devido por causa de um erro de programa; eles devem ser destruídos assim que tal situação seja detectada. Isso pode ser feito com a manutenção, no objeto pai, de uma lista com todos os objetos filhos criados e não destruídos; no momento da destruição do objeto pai, todos os filhos recebem um aviso e destróem-se automaticamente. A exceção são os objetos autônomos, que continuam vivos.

5.2.7 Variáveis de classe

Os objetos em Cm possuem dados que são próprios a cada instância, as chamadas *variáveis de instância*, e possuem também dados compartilhados por todas as instâncias de uma classe, as chamadas *variáveis de classe*. Dentro de uma classe Cm, uma variável declarada com a classe de armazenamento `static`, fora de um método, é uma variável de classe.

No caso da programação com objetos remotos, as variáveis de classe são um problema sério, pois significam compartilhamento de informações num contexto dis-

1. A propósito, consulte-se a definição 5.6, seção 5.6.

tribuído. Suponhamos uma aplicação composta de vários objetos remotos de uma mesma classe, filhos de um mesmo objeto pai, e cada um sendo executado em uma máquina diferente da rede. Suponhamos que esses objetos remotos tenham variáveis de classe: essas variáveis representam informação sendo compartilhada por todos esses objetos remotos. De imediato temos um problema de eficiência: como utilizar essas informações com um tempo de espera aceitável? Há um outro problema, que é manter a consistência dessas informações, que são compartilhadas pelos objetos remotos. E se são criados objetos da mesma classe e não remotos, dentro do mesmo objeto pai: as instâncias remotas compartilham os mesmos dados das instâncias não remotas?

Sobre esse assunto vale a pena entender como o compilador implementa as variáveis de classe, pois isso vai nos possibilitar entender como as variáveis de classe são utilizadas e como podemos procurar uma solução a respeito. Como já foi dito, o compilador de Cm produz, a partir de uma hierarquia de classes, um único programa executável. Há um passo intermediário do compilador, que consiste em gerar, como código objeto, arquivos em C, que por sua vez serão compilados e ligados para compor o programa executável.

Na notação intermediária em C, as variáveis de classe são mapeadas em variáveis globais, já que são compartilhadas pelas instâncias da classe (as variáveis de instância são guardadas em estruturas). Dentro dos métodos da classe, qualquer operação (leitura, atribuição, operações aritméticas, etc.) sobre uma variável de classe é traduzida pela aplicação da operação sobre a variável global que representa a variável de classe. As variáveis de classe começam a existir com a execução do programa, e deixam de existir com o fim do programa: precedem, então, a criação de qualquer instância da classe, e continuam a existir depois da destruição da última instância da classe.

Vamos imaginar o mesmo procedimento no caso de objetos distribuídos: o "lugar" mais adequado para guardar essas variáveis parece ser o objeto pai, de forma que as variáveis de classe fiquem agrupadas em um único lugar, e que existam antes e depois da criação de qualquer instância de objeto (remoto ou não) a partir desse objeto pai. Não obstante as dificuldades para uso dessas variáveis por parte dos objetos remotos (pelas razões citadas anteriormente), há um problema conceitual: os objetos remotos definem seus próprios contextos de execução, separados do contexto do objeto pai. Tais variáveis de classe estão no contexto do objeto pai, de forma que não estão disponíveis para nenhum objeto remoto que seja seu filho.

Na realidade, todas essas dificuldades originam-se de um mesmo problema: o conflito entre distribuição e compartilhamento de dados [Weg87]. Há vários estudos propondo, inclusive, linguagens para programação distribuída com suporte a objetos mas não à herança, já que esta pode ser implementada por compartilhamento de código, o que seria incompatível com um ambiente distribuído. No caso da linguagem Cm, a herança é resolvida em tempo de compilação, e o código dos métodos das superclasses são replicados nas subclasses, de modo que não há compartilhamento de código em tempo de execução. O problema das variáveis de classe, entretanto, permanece; para resolvê-lo, é necessário suprimir a possibilidade de compartilhamento de dados nesses casos.

A solução estabelecida é a seguinte: objetos remotos têm suas próprias variáveis de classe, pertencentes ao seu contexto de execução. Um contexto de execução próprio, separado do contexto do objeto pai, é a propriedade que define um objeto remoto, e portanto ela deve prevalecer em qualquer caso. Informações a ser compartilhadas por objetos remotos "irmãos" devem estar armazenadas em objetos (também remotos) devidamente criados com esse objetivo, e visíveis a todos que os utilizem com essa finalidade.

5.2.8 Comunicação entre objetos distribuídos

Dentro de uma aplicação distribuída, os diferentes objetos que a compõem solicitam uns aos outros a ativação de métodos através do envio de mensagens. A fim de preservar a transparência do ambiente, uma ativação de método de um objeto remoto deve ter a mesma forma, o mesmo comportamento e os mesmos efeitos de uma chamada de um método não remoto. Naturalmente a comunicação entre objetos remotos é um pouco mais demorada, pela necessidade de transmissão das mensagens pela rede e mais propensa a erros, pela possibilidade de que algum nó da rede fique temporariamente inativo ou sem comunicação, tornando inacessíveis os objetos que estão lá. Essas situações, no entanto, são devidas a falhas no ambiente, e fogem ao controle do programador ou da aplicação.

Toda chamada de método remoto deve ser intermediada pelo sistema OMNI, pois representa informação sendo transferida entre diferentes espaços de endereçamento, mesmo que os dois objetos em questão (o cliente, que fez a chamada, e o objeto remoto chamado) estejam na mesma máquina.

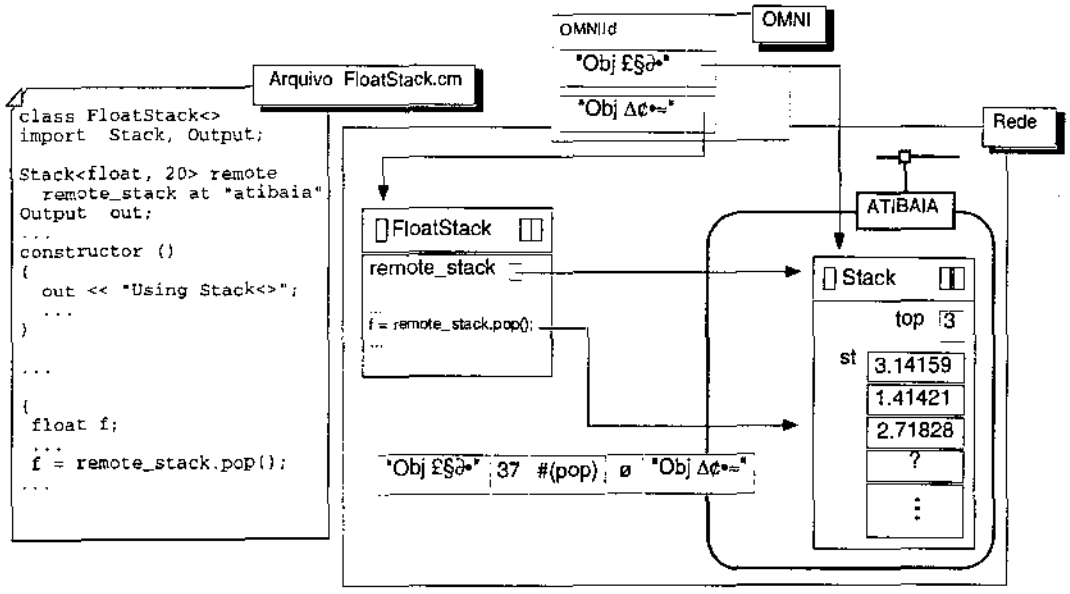
O uso de métodos e variáveis da interface da classe, por um objeto remoto, é transformado pelo compilador em montagem de mensagens equivalentes e em chamadas para funções de comunicação. Tais mensagens são geradas, enviadas, recebidas e interpretadas pelo sistema de suporte à execução da linguagem.

Uma chamada de método como *obj.f()* leva à construção de uma mensagem com 5 campos:

- a identificação do objeto *obj*;
- um número que identifica a chamada no cliente;
- a identificação do método *f()*;
- lista de parâmetros do método (vazia nesse caso);
- identificação do objeto que está fazendo a chamada (o cliente).

Exemplo 5.14. Usando uma nova versão da classe *FloatStack*, podemos ilustrar o formato de uma mensagem que corresponde a uma ativação de método em um objeto remoto.

FIGURA 5-9 Chamada de um método remoto



(Fim do exemplo 5.14.)

5.2.8.1 Tratamento da lista de parâmetros

O conteúdo da lista de parâmetros pode ser extenso, dependendo do tipo de serviço executado pelo método. Na maior parte dos casos dos métodos com parâmetros, eles ocuparão um espaço maior (ou muito maior) que o restante da mensagem enviada ao objeto alvo, já que não há restrição quanto aos tipos de dados que podem ser trocados entre objetos situados remotamente (estruturas, *strings*, vetores, etc).

Torna-se necessário, portanto, planejar com cuidado o mecanismo de montagem das chamadas, de forma a permitir que a transformação dos parâmetros em *bytes* (para transmissão pela rede) e a operação inversa, no objeto alvo, impliquem em um *overhead* aceitável para uma chamada remota. Na nossa proposta, esse trabalho está a cargo do sistema Linear [Sou92], apresentado a seguir.

Elementos de uma estrutura complexa (que envolva apontadores e registros, e.g.) são alocados individualmente, conforme são necessários; o resultado é que a estrutura fica distribuída em memória. Para transmissão pela rede, a estrutura deve ser *linearizada*, ou seja, transformada em uma seqüência de *bytes*. . O processo inverso, denominado *deslinearização*, corresponde a produzir, a partir de uma seqüência de caracteres, uma estrutura em memória equivalente à original.

Muitos sistemas de linearização são baseados no padrão XDR [Sun88], utilizado no mecanismo de RPC da Sun. Este padrão estabelece uma linguagem universal de definição de tipos, utilizada para definição das interfaces dos programas servidores,

e uma representação binária desses tipos independente de linguagem, sistema operacional e arquitetura de *hardware*, para livre transmissão entre programas. A linguagem de definição de tipos do XDR é muito semelhante à definição de tipos na linguagem C. Para cada tipo de dados processado, são geradas funções *stubs* específicas, utilizadas nos processos de linearização/deslinearização a cada envio e recebimento de chamada que inclua um dado tipo. Essas funções são compiladas juntamente com o código dos programas clientes e servidores. A abordagem do XDR tem alguns inconvenientes: o uso de uma representação universal implica no aumento do tamanho da representação (superconjunto de todas as representações possíveis) e os dados a serem transmitidos são limitados pelas funções geradas. Outro aspecto importante é que as implementações de XDR não tratam adequadamente estruturas que contenham auto-referências (uma lista circular é o exemplo mais simples).

O Linear adota uma abordagem diferente do sistema XDR. As informações sobre os tipos são armazenadas em uma tabela, e são interpretadas por duas funções universais de linearização e deslinearização. Dessa forma, não se adiciona uma quantidade grande de código às aplicações, ao mesmo tempo em que não se limita *a priori* os tipos de dados que podem ser trocados.

Para integração com o compilador da versão distribuída da linguagem Cm, a tabela de tipos é meramente um subconjunto da tabela de símbolos de uma classe sendo compilada. Esse trabalho também deverá ser feito para a transmissão de dados através de portas, conforme se verá mais adiante, na seção 5.4.2 ("Portas Tipadas").

5.2.8.2 Tratamento de chamadas de métodos

Quando a mensagem chega ao objeto remoto, ela é atendida por um tratador genérico, denominado *dispatcher*, que faz parte do sistema de suporte à execução. Esse tratador recebe e interpreta todas as mensagens enviadas a um objeto. No caso, essa mensagem corresponde a uma ativação de método; o tratador chama uma função intermediária que retira da mensagem os parâmetros e faz a chamada ao método indicado, passando os parâmetros correspondentes. Quando termina a execução do método, o resultado é devolvido para a função intermediária que monta a mensagem de resposta, que a seguir é enviada para o objeto cliente.

No objeto cliente, a mensagem é recebida pelo tratador genérico, que verifica tratar-se de uma resposta a uma ativação de método pedida por esse objeto. Essa resposta pode ser de dois tipos: um resultado normal ou uma exceção. No caso de um resultado normal, ele é usado no lugar da expressão que representa a chamada. No caso em que a resposta é uma exceção, isso significa que a chamada do método remoto causou uma exceção no servidor, e que essa exceção não foi tratada lá; portanto ela é devolvida ao objeto cliente para tratamento adequado. A propagação de exceções entre objetos distribuídos é apresentada na seção 5.3 ("Exceções em objetos distribuídos").

Se houver algum problema na transmissão da chamada do método, na execução do método ou na devolução da resposta, é provável que a chamada não possa ser completada, ocorrendo um *time-out* no cliente. Nessa situação, é gerada uma exceção, de um tipo pré-definido pelo sistema de suporte e com um valor dependendo do

contexto da chamada. Se por acaso a resposta à chamada remota chegar depois disso, ela é descartada.

5.3 Exceções em objetos distribuídos

As exceções são um mecanismo oferecido pelas linguagens de programação para tratar, de maneira estruturada e uniforme, situações especiais ou anormais que surgem durante a execução dos programas.

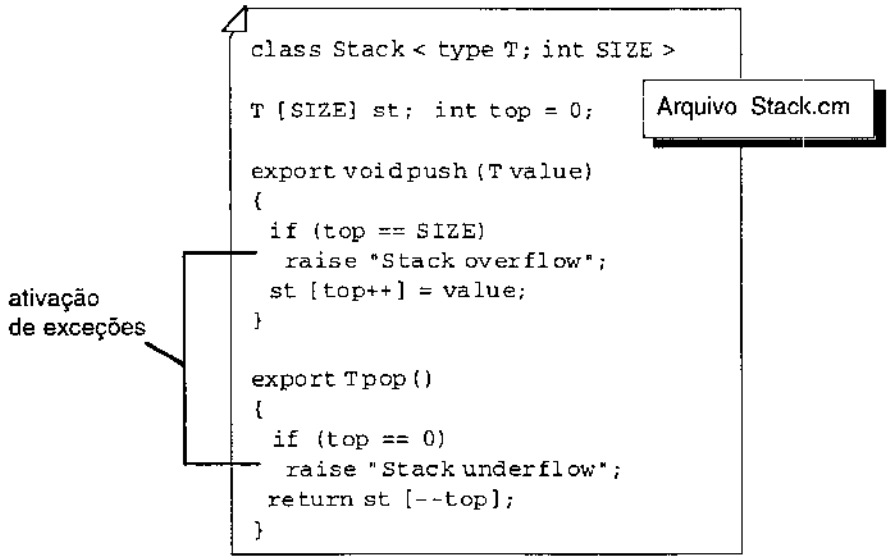
O mecanismo de tratamento de exceções da linguagem Cm está baseado nos chamados *comandos protegidos* [Tel93]. A ocorrência de uma exceção dispara a busca de um tratador; essa busca é feita no sentido inverso da seqüência da chamada de métodos e da entrada em blocos de programa. No momento em que uma exceção é sinalizada, ela pode transportar um valor, que é uma informação de contexto, útil para o tratador; o tipo desse valor identifica essa exceção, e é usado para encontrar o tratador adequado. O mecanismo de tratamento de exceções está descrito na subseção 4.4.3 ("Exceções").

Na extensão da linguagem Cm para programação distribuída, o mecanismo de tratamento de exceções será preservado, e sua semântica e implementação serão estendidas para o contexto distribuído. O princípio básico a ser seguido é o da transparência, no sentido de obter comportamento uniforme para objetos remotos e não remotos.

Quando um objeto faz uma chamada a um método remoto, e essa chamada causa uma exceção no objeto remoto, caso essa exceção não seja tratada lá ela passa a ser de responsabilidade do objeto chamador. O sistema de suporte da linguagem se encarrega de fazer transparentemente a propagação da exceção entre os objetos e a reativação da mesma no contexto do objeto chamador.

Exemplo 5.15. Vamos usar uma implementação da classe *Stack* que usa exceções para sinalizar erro.

FIGURA 5-10 Classe *Stack* utilizando exceções



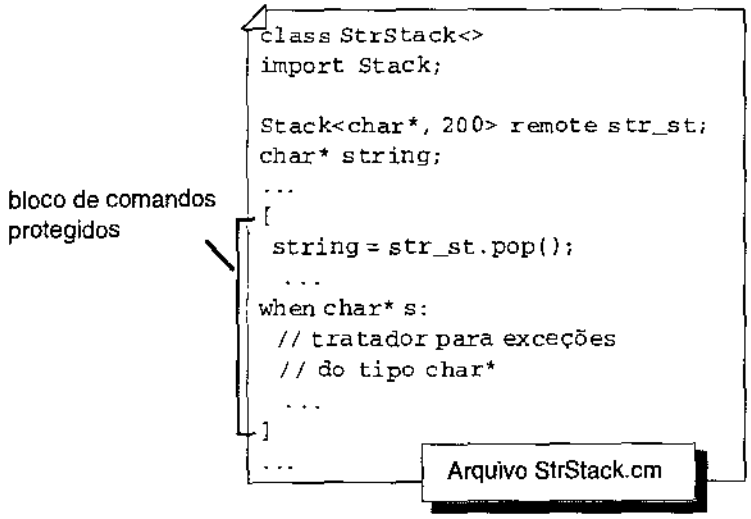
Um objeto dessa classe vai gerar exceções em dois casos: uma tentativa de inserção na pilha cheia e uma tentativa de retirada de uma pilha vazia. (Fim do exemplo 5.15.)

Objetos clientes devem prever a possibilidade de ocorrência de exceções nos seus servidores e, portanto, as chamadas remotas devem estar contidas em comandos protegidos, para detectar e tratar convenientemente essas situações especiais. Isso implica em conhecer quais exceções podem ser geradas por objetos de uma classe, aí incluídas exceções geradas por servidores de objetos dessa classe e que não são tratadas. Na realidade, essas exceções pertencem à interface da classe, pois o resultado de um método pode ser um valor do seu tipo ou pode ser uma exceção, e isso é visível externamente aos objetos.

Nesse particular, a linguagem Cm não oferece suporte a essa "documentação" de exceções, como de resto outras linguagens, como por exemplo C++. Em Argus [Lis88], os procedimentos indicam explicitamente, além do tipo de retorno, as exceções que podem gerar, definindo assim a sua interface de forma completa. O ideal seria que as classes Cm pudessem oferecer, como informação disponível às classes clientes, informações sobre as exceções que podem ser geradas por seus métodos. Não sabemos ainda se essas informações deverão ser colocadas explicitamente no código fonte, através de alguma nova construção sintática, ou se isso poderia ser implementado pelo ambiente de suporte à execução da linguagem. Essa questão, embora importante, não está todavia nos objetivos desse trabalho, já que trata de um problema inerente à linguagem Cm (dentro ou fora do contexto da programação distribuída).

Exemplo 5.16. Utilizando essa nova versão da classe *Stack*, vamos criar uma classe cliente que prevê a ocorrência de exceções durante a execução de métodos de *Stack*, e providencia tratadores para essa exceções.

FIGURA 5-11 Exemplo de cliente que prevê exceções no servidor



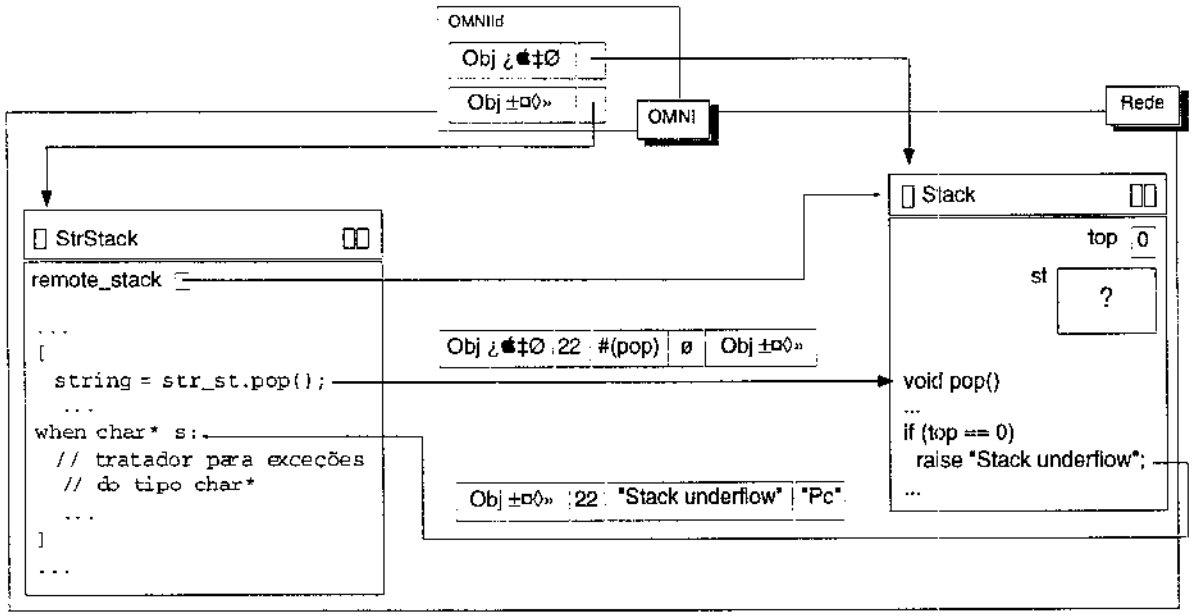
A chamada *str_st.pop(..)* vai ficar bloqueada até que seja recebida uma resposta ou ocorra um *time-out*. Caso a pilha esteja cheia, essa chamada causa no objeto remoto uma exceção do tipo *char**, cujo valor é `"Stack underflow"`. Essa exceção é de responsabilidade do objeto chamador da classe *StrStack*, pois não há comando protegido dentro do método *pop()*. Essa exceção é devolvida ao objeto chamador como resultado da ativação do método, e sinalizada no ponto da chamada. (Fim do exemplo 5.16.)

A informação transportada por uma exceção é o valor a ela atribuído no momento da sinalização, e o seu tipo é determinado pelo tipo da expressão que produziu o valor da exceção. Quando uma exceção deixa um objeto, para ser tratada em outro lugar, essa informação de contexto deve ser colocada dentro da mensagem, que será entregue ao objeto chamador. Uma mensagem de exceção tem 4 campos:

- a identificação do objeto chamador, a quem a exceção é endereçada;
- o número da chamada que originou a exceção;
- o valor transportado pela exceção, linearizado;
- o tipo da exceção, representado por sua assinatura.

Exemplo 5.17. Um exemplo de mensagem sinalizando uma condição excepcional na classe *Stack* é mostrada a seguir.

FIGURA 5-12 Propagação de exceção entre objetos distribuídos



Os campos mostrados na mensagem de exceção representam as informações relevantes para efeito de explicação; naturalmente deve haver, no cabeçalho das mensagens trocadas entre os objetos, informações sobre o tipo da mensagem (no caso mostrado acima, isso serve para distinguir uma mensagem que representaria o resultado da execução normal do método remoto de uma mensagem representando uma exceção). A *string* "Stack underflow" é o valor da exceção, indicando uma tentativa de execução do método *pop()* em uma pilha vazia, e a *string* "Pc" é a assinatura[1] do tipo da exceção (`char*`). (Fim do exemplo 5.17)

A chegada de uma requisição para a execução de um método, em um objeto remoto, causa o estabelecimento de um contexto fictício, que oferece apenas um tratador para capturar exceções causadas por essa ativação de método, que tenham sido sinalizadas e não tratadas dentro de objeto. Quando acionado, esse tratador monta e envia a mensagem devida ao objeto chamador, representando a exceção e seu valor e tipo associados.

A mensagem representando a exceção será recebida pelo *dispatcher* do objeto chamador, que a repassará a uma função que extrairá o valor e o tipo da exceção. O valor obtido e a assinatura do tipo serão então utilizados para sinalizar a exceção no ponto da chamada remota.

1. As assinaturas são representação de tipos em tempo de execução, consulte-se a seção 5.5.2 ("Verificação de compatibilidade").

5.4 Portas de Comunicação

A proposta original de programação distribuída em Cm está baseada na linguagem LegoShell, e o modelo de programação subjacente. Por esse modelo, os programas escritos em Cm dispõem de portas de comunicação para receber e enviar dados entre o objeto e o mundo externo. Cabe à LegoShell fazer a composição desses objetos a partir de conexões envolvendo as portas de comunicações dos objetos. Para maiores detalhes vide seção 4.6 ("A Linguagem LegoShell").

Os programas em Cm são, dessa maneira, módulos de *software* auto-contidos, e as computações em LegoShell representam a combinação desses módulos para compor um sistema de maior complexidade. Essa forma de programar corresponde a uma separação entre a tarefa de escrever pequenos programas (*programming-in-the-small*) e a de escrever sistemas completos a partir de um número arbitrário de pequenos programas (*programming-in-the-large*) [DeR76].

O modelo de objetos distribuídos representa um aumento do poder de expressão da linguagem Cm, de forma a permitir a construção de programas cuja complexidade os colocava no escopo da LegoShell. Basicamente, o modelo de objetos distribuídos prevê outra forma de comunicação entre objetos, a chamada remota de métodos,[1] que não estava prevista na proposta original da LegoShell. As portas de comunicação constituem-se, portanto, em outro modelo de comunicação, e serão implementadas na linguagem como proposto originalmente.

5.4.1 Portas em Cm

Dentro de programas Cm, as portas de comunicação são objetos da classe *Port*. Essa classe é pré-definida na linguagem: seus métodos são implementados diretamente pelo sistema de suporte à execução. As portas declaradas por um objeto são utilizadas para trocar informações entre esse objeto e os objetos donos de outras portas, às quais as suas portas serão conectadas. Nesse sentido, as portas são objetos especiais, que recebem um tratamento especial, pois de certa forma são criadas e manipuladas antes que os objetos comecem realmente sua execução.

Toda classe que declara portas deve importar a classe *Port* (ou a classe *TypedPort*, no caso de portas tipadas). A declaração da interface dessas classes é apresentada, com comentários, no apêndice C ("Alterações na Linguagem").

Exemplo 5.18. Vamos apresentar um exemplo muito simples, composto por três objetos remotos.

```
class X<>
import Port, Y, Z;

Y remote obj_Y;
Z remote obj_Z;
```

1. Para detalhes, consulte-se a seção 5.2.3 ("Objetos Remotos").

```
...
connect (obj_Y.out, obj_Z.in);
...
```

Um objeto da classe *X* cria dois objetos remotos, um da classe *Y* e outro da classe *Z*, e realiza a conexão entre as portas de comunicação declaradas por esses objetos.

```
class Y<>
import Port;

Port output @ (Port::OUTPUT);

...
output << ":-)";
...
```

Um objeto da classe *Y* possui uma porta de saída, e faz uma operação de escrita através do operador "<<".

```
class Z<>
import Port;

Port input @ (Port::INPUT);

...
char* str;
input >> str;
...
```

Um objeto da classe *Z* possui uma porta de entrada, e faz uma operação de entrada com o operador ">>". (Fim do exemplo 5.18.)

No exemplo mostrado acima, os operadores "<<" e ">>" foram redefinidos para aplicar sobre as portas operaçoes de *write()* e *read()*, respectivamente. Essas operações são síncronas, i.e., os dados só serão escritos e lidos quando essas funções forem executadas: no caso de uma porta de leitura, pode acontecer que haja uma fila de mensagens, à espera de leitura. Pode-se especificar, no sistema OMNI, que a chegada de uma mensagem em uma porta seja imediatamente sinalizada ao dono da porta, mas não há uma construção em Cm para explorar essa possibilidade (uma abordagem possível seria utilizar exceções).

O sistema OMNI oferece portas não conectáveis, úteis quando um servidor tem um número arbitrário de clientes, enviando mensagens em ordem aleatória. Na proposta original deste trabalho de mestrado, essas portas seriam usadas para o *bind* dinâmico entre clientes e servidores. Nos clientes, chamadas a *serviços* seriam convertidas em mensagens endereçadas a uma porta não conectável; no servidor, dono da porta não conectável, a chegada de uma mensagem ativaria a execução do serviço correspondente. Com o modelo de objetos distribuídos, entretanto, esse mecanismo não é mais necessário, já que a associação entre clientes e servidores não se faz mais por serviços, e sim entre os próprios objetos. Vale dizer, portanto, que o modelo de objetos distribuídos engloba o modelo da proposta original.

5.4.2 Portas Tipadas

As portas tipadas têm basicamente, funcionalidade equivalente às portas comuns, com a diferença que elas têm um tipo associado, e as operações de entrada e saída operam sobre dados desse tipo. Durante a configuração de aplicações utilizando essas portas, há ainda a necessidade de fazer a verificação de compatibilidade entre portas que se pretende conectar; em tempo de execução, deve-se fazer a conversão desses dados em seqüências de *bytes* e vice-versa (operações de linearização e deslinearização, respectivamente).

Essas portas são objetos da classe *TypedPort*, que é parametrizada pelo tipo da porta. As operações de *read()* e *write()* (e os correspondentes operadores redefinidos) são como no caso de portas simples, com a diferença de que os elementos transferidos são dados do tipo da porta, ao invés de *bytes*.

Exemplo 5.19. Vamos fazer um exemplo semelhante ao de portas simples. Será necessário definir apenas o tipo dos dados transmitidos.

```
class Date<>

int dia, mes, ano;

// métodos da classe Date
...
```

As outras classes componentes do exemplo são as seguintes:

```
class TX<>
import TypedPort, TY, TZ;

TY remote obj_TY;
TZ remote obj_TZ;

...
connect (obj_TY.out, obj_TZ.in);
...
```

```
class TY<>
import TypedPort, Date;

TypedPort<Date> output @ (TypedPort::OUTPUT);
Date d;
...
output << d;      // d armazena uma data válida
...
```

```
class TZ<>
import TypedPort, Date;
```

```
TypedPort<Date> input @ (TypedPort::INPUT);
Date input_date;
...
input >> input_date;
...
```

(Fim do exemplo 5.19.)

O tipo dos dados enviados por portas tipadas pode ser arbitrariamente complexo, qualquer tipo que possa ser especificado em Cm. Os algoritmos de serialização e desserialização tratam adequadamente estruturas auto-referenciáveis (listas ligadas, por exemplo), mesmo casos particularmente difíceis (quando vários apontadores referenciam partes não disjuntas de uma mesma estrutura). Tais algoritmos (que compõem o sistema Linear [Sou92]) estão embutidos no sistema de suporte à execução, e são executados de forma transparente pelas funções de escrita e leitura.

A verificação da compatibilidade que deve ser feita no momento da conexão de duas portas tipadas envolve a representação de tipos do ambiente. Eventualmente, tais informações devem ser manipuladas em tempo de execução, já que portas de comunicação podem ser criadas dinamicamente. Essa questão é descrita na próxima seção.

5.5 Representação de tipos

As aplicações construídas no ambiente A_HAND usam uma representação universal de tipos, garantindo total compatibilidade entre seus componentes. Para grandes programas, cuja produção o nosso ambiente procura facilitar e sistematizar, a capacidade de verificação de tipos é importante para detectar erros, antes da fase de execução. Eventualmente, será necessário fazer verificação de tipos em tempo de execução.

A representação de tipos é baseada no sistema de tipos da linguagem Cm, o que abrange os tipos de todos os objetos que podem ser manipulados pelo ambiente. Uma importante característica desse sistema de tipos é a presença de polimorfismo: esse aspecto será explorado ao máximo na construção de aplicações, e não representa obstáculo para os mecanismos de verificação de compatibilidade de tipos.

A compatibilidade em ambientes que respeitam o padrão CORBA, por exemplo, está baseada na definição de interfaces utilizando-se a linguagem IDL.[1] Com base em interfaces definidas nessa linguagem, um cliente pode solicitar serviços de um objeto sem conhecer outras informações a seu respeito (linguagem de programação em que está escrito, e.g.). Para o nosso ambiente está prevista a compatibilidade com essa linguagem, dentro de um esforço mais abrangente, para permitir que as aplicações distribuídas possam combinar objetos de outros ambientes (esse assunto está exposto no capítulo 6, onde estão as conclusões do trabalho).

1. Para maiores informações vide a subseção 3.3.3 ("CORBA — Common Object Request Broker Architecture")

5.5.1 O sistema de tipos de Cm

A linguagem Cm suporta um pequeno conjunto de tipos padrão, que são pré-definidos, e um conjunto de construtores de tipo, utilizados para definir tipos mais complexos. É também possível a definição de sinônimos para tipos e definição de novos tipos por nomeação.

Os tipos padrão são basicamente herdados de C: *char*, *short*, *int*, *long*, *float*, *double* e *void*. Os construtores de tipo de Cm são: apontador (`*`), referência (`&`), *array* (`[]`), função (`()`), estruturas ou agregados (`struct`), uniões ou agregados justapostos (`union`), enumerado (`enum`) e classe (`class`). O tipo referência e o tipo classe não existem em C; os demais são bem conhecidos. O tipo classe é o mais complexo de Cm, e introduz na linguagem abstração de dados, programação modular e orientação a objetos.

Do ponto de vista sintático, as declarações em Cm diferem bastante de C, já que os construtores de tipos não se aplicam aos identificadores, e sim aos tipos mais simples sendo compostos. Essa representação permite que a declaração de tipos seja lida da direita para a esquerda, pela aplicação sucessiva dos construtores de tipo a tipos cada vez mais simples. Essa representação é facilmente mapeada na assinatura de tipo, descrita na próxima seção.

A compatibilidade de tipos em Cm é definida por regras que permitem a utilização de dados de tipos diferentes em operações de atribuição e passagem de parâmetros. Vamos descrever sucintamente essas regras, visando a discussão contida nas próximas seções; a definição detalhada delas está em [Tel93].

Se o domínio de um tipo está propriamente contido no domínio de um outro tipo, o valor de um dado do primeiro tipo pode ser transformado, por *promoção*, para um valor equivalente do segundo tipo. A operação inversa nem sempre é possível, pois existem valores de um domínio que não pertencem ao outro; nesses casos, ocorre uma *projeção* de valores de um tipo no outro, com conseqüente perda de informação.

A compatibilidade de tipos pode ocorrer em três níveis: equivalência, generalização e conversão. Esse níveis dão a medida da diferença entre os tipos envolvidos no teste de compatibilidade. Há equivalência quando os tipos confrontados são o mesmo tipo, um é sinônimo do outro ou ambos são sinônimos de um mesmo tipo. Na generalização, os tipos são diferentes, mas um pode ser promovido para o outro, que deve ser considerado genérico pela linguagem. Já a conversão ocorre se a equivalência e a generalização não podem ser aplicadas, mas um tipo puder ser promovido ou projetado no outro. Na impossibilidade de qualquer uma dessas alternativas, diz-se que os tipos comparados são incompatíveis.

Para os diversos construtores de tipos da linguagem, temos as seguintes normas para a compatibilidade:

- tipos padrão
  - i. os tipos *char*, *unsigned char*, *short*, *unsigned short*, *unsigned int* e enumerados são compatíveis, por generalização, com o tipo *int*;
  - ii. da mesma forma, o tipo *float* é compatível com o tipo *double*;

iii. enumerados nomeados são incompatíveis entre si; e

iv. o tipo *void* é incompatível com qualquer outro;

- tipo apontador

  i. qualquer tipo apontador é compatível por generalização com o tipo apontador para *void*; e

  ii. apontadores para tipos equivalentes são compatíveis por equivalência;

- a compatibilidade entre *arrays* segue as regras da compatibilidade entre apontadores; a compatibilidade por equivalência exige dimensões iguais e elementos equivalentes;
- dois tipos função são equivalentes se

  i. seus tipos de retorno são equivalentes;

  ii. o número de parâmetros é o mesmo e são dois a dois equivalentes; e

  iii. ambas são automáticas (classe de armazenamente `auto`) ou nenhuma delas o é.[1]

- o tipo referência não afeta a compatibilidade, esta é verificada com base nos tipos referenciados;
- dois tipos agregados são equivalentes se

  i. ambos são nomeados, têm o mesmo nome ou são sinônimos;

  ii. seus membros, considerados dois a dois por ordem de declaração, têm o mesmo nome e o mesmo tamanho em bits (no caso de campos); e

  iii. são ambos estruturas, uniões ou uniões discriminadas.

  e se satisfazem ii e iii mas não i, são compatíveis por generalização se o segundo tipo for anônimo ou por conversão se o primeiro for anônimo.

- dois tipos classe são equivalentes se seus nomes são idênticos e, no caso de classes parametrizadas, os parâmetros são dois a dois equivalentes; um tipo classe é compatível com um segundo tipo, por conversão, se este é equivalente a uma das classes base do primeiro, em qualquer nível da hierarquia de herança.

5.5.2 Verificação de compatibilidade

Durante a fase de compilação, a compatibilidade de tipos é decidida por algoritmos baseados na representação do compilador para os tipos da linguagem. O que nos interessa é o que ocorre depois disso, quando a fase de compilação e ligação estiver concluída e os objetos estiverem sendo executados. Nesse momento, a informação dos tipos dos objetos deve ser suscetível de recuperação, pois pode haver a necessidade de verificar a consistência de operações dependentes de valores obtidos durante a execução das aplicações.

A representação de tipos em tempo de execução é necessária, pelo menos, nas seguintes situações:

- conexão entre portas tipadas de comunicação;
- propagação de exceções entre objetos remotos; e

1. Essa regra se refere à "presença" ou não do parâmetro implícito *self* (apontador para o objeto corrente) na lista de parâmetros da função.

- associação de objetos remotos utilizando o Servidor de Nomes.

Nesses casos, os objetos sendo manipulados devem ser comparados com base em uma notação que represente seu tipo. A representação utilizada é a chamada *assinatura do tipo*, e consiste de uma cadeia de caracteres que permite identificar univocamente o tipo envolvido. Essa representação está proposta em [Koe90], e já é utilizada pelo mecanismo de tratamento de exceções da linguagem Cm. A assinatura de um tipo é composta de uma seqüência de caracteres, codificados para representar tipos padrão ou construtores de tipo, de tal forma que a aplicação dos construtores na ordem em que aparecem na assinatura resulta, ao final, na forma canônica do tipo representado.

A seguir consideramos cada uma dessas três alternativas de verificação de compatibilidade de tipos em tempo de execução. Interessa conhecer os procedimentos utilizados para lidar com todas as possibilidades que podem acontecer em cada caso, e como a informação do tipo é utilizada.

5.5.2.1 Conexão de portas tipadas de comunicação

Um objeto que declara portas (tipadas ou não) de comunicação deve levar, embutido no seu código executável, uma tabela que contém descritores de portas. Nesses descritores estão armazenadas informações usadas pelo compilador (e.g., porta de entrada ou de saída, tipo do dados que é transmitido pela porta) como informações próprias de tempo de execução (e.g., porta conectada ou não, estado de operações de E/S).

No caso de portas tipadas, há uma coluna com a assinatura do tipo da porta, que descreve univocamante esse tipo. Durante a operação de conexão que envolve essa porta, o sistema de suporte extrai essa informação do código do objeto, e função genérica para comparação de assinaturas de tipo pode ser aplicada. Ao mesmo tempo que se faz essa verificação, outras também são necessárias, sendo a porta tipada ou não: por exemplo, é preciso que a uma porta de saída conecte-se uma de entrada, as portas devem ter semânticas compatíveis.[1] Essa é a situação mais simples que pode ocorrer durante a conexão.

Conexões mais complexas são possíveis em dois casos, que vamos descrever apenas brevemente pois ainda são objeto de estudo. O primeiro deles diz respeito a versões de classes. Na assinatura de tipos, uma classe é representada por seu nome, que se supõe seja único no ambiente; o compilador também procura as classes (código fonte) por nome.

Em um ambiente de desenvolvimento com grande número de usuários, entretanto, é natural que usuários utilizem classes diferentes mas representadas pelo mesmo nome. Pode acontecer que essas diferenças sejam mínimas, dependentes das necessidades de cada um, ou que sejam muito grandes e que as classes tenham características completamente diferentes, i.e., o mesmo nome seria apenas uma

1. Para maiores detalhes consulte-se a seção 4.5.1.3 ("Módulo de Portas Conectáveis").

coincidência. Um conjunto de classes utilizado por um determinado usuário seria, portanto, uma visão que ele tem do ambiente, que pode ser inteiramente diversa das visões de outros usuários, trabalhando em outros projetos.

Para que não haja tais conflitos de nomes, um mecanismo de gerenciamento de versões de classes (assim como de outros documentos do sistema) deve estar disponível, de modo que todas as ferramentas do ambiente (e não apenas o compilador ou o módulo de portas, responsável pela operação de execução) trabalhem sempre sobre as versões corretas. O mecanismo proposto em [Vic89, Vic90] considera as necessidades do Projeto A_Hand nesse particular, e deve ser implementado para oferecer as facilidades descritas.

Outra situação mais complicada é quando a conexão envolve as chamadas computações polimórficas,[1] que envolvem instâncias de classes que ainda não tiveram parâmetros de tipo instanciados. Nesse caso, a assinatura de tipos deve ser capaz de representar essa informação, e o algoritmo de verificação de compatibilidade terá de tratar situações muito mais complexas, que podem envolver unificações de tipos. Essa facilidade será necessária na LegoShell, para a configuração de computações polimórficas.

5.5.2.2 Propagação de exceções entre objetos remotos

A propagação de exceções entre objetos distribuídos usa a mesma representação da assinatura de tipo que já é usada pela linguagem Cm. Não há, entretanto, diferença conceitual entre objetos num contexto distribuído e no contexto atual. Entretanto, aqui também se aplica a discussão sobre versões de classes, conforme exposto no tópico anterior.

5.5.2.3 Associação de objetos remotos utilizando o Servidor de Nomes

O Servidor de Nomes será utilizado para armazenar informações no formato (nome simbólico, identidade de objeto remoto). Uma entrada dessa forma será colocada no SN no momento da criação de um objeto remoto com nome simbólico, e será removida quando esse objeto for destruído. Essa informação estará disponível para consulta por parte de objetos remotos que queiram uma referência para o objeto cadastrado com um determinado nome.

A função *attach()* usa, como segundo parâmetro, uma referência para um objeto remoto, que pode ser tanto um nome simbólico como uma OMNIId (de alguma maneira disponível para o programa).[2] Nessas duas informações, inexiste a noção de tipo, portanto a execução de *attach()* deve proceder à pesquisa do tipo do objeto que está sendo associado, e compará-lo com o tipo do identificador declarado com o construtor `unattached`. Ambas as informações, porém, dão acesso ao objeto, e neste está contida a informação sobre o seu tipo. Naturalmente, aqui também há o problema de versões de classes.

1. Sobre esse particular ver a seção 4.6 ("A Linguagem LegoShell").
2. A função *attach()* está descrita no apêndice C ("Alterações na Linguagem").

5.6 Concorrência

A efetiva exploração do paralelismo depende da divisão de um problema em um conjunto de partes que podem ser executadas em separado e simultaneamente, visando o mínimo de interdependência entre elas e o máximo proveito dos recursos compartilhados. Essa tarefa será muito difícil, e o resultado pouco eficiente, se não existirem, dentro da linguagem de programação utilizada, mecanismos adequados para expressar esse particionamento de atividades.

Para resolver tais problemas, a programação concorrente orientada a objetos estuda como construir programas compostos por uma coleção de objetos, a partir da implementação de mecanismos de programação distribuída e concorrente nas linguagens orientadas a objetos. Objetos podem representar naturalmente essas entidades paralelas já que, por definição, apresentam propriedades de encapsulamento e autonomia. É necessário analisar alguns aspectos dessa integração de concorrência com objetos, particularmente a incorporação de conceitos bastante estabelecidos (como processos, sincronização, exclusão mútua, etc.) dentro das técnicas de programação características do paradigma de objetos.

5.6.1 Definições

O estudo da programação concorrente no contexto de objetos implica em algumas alterações de terminologia. Em especial, o termo processo será utilizado de maneira diferente da sua acepção mais comum, da área de sistemas operacionais.

Definição 5.6. Um *processo* tem uma interface composta por várias operações por ele executadas, e contém um ou mais fluxos de execução, ativos ou suspensos num determinado momento.

De acordo com essa definição (de [Weg87]), os objetos que fazem parte de uma aplicação distribuída são processos. É importante destacar que esse termo está sendo usado com uma nova acepção, próxima à definição de objeto.

Os *fluxos de execução* são também denominados *lightweight processes* ou *threads* (o termo que usaremos mais freqüentemente a partir de agora). São unidades de execução com granularidade menor que os processos (na acepção tradicional destes). O contexto de execução de uma *thread* é pequeno, composto de informações muito básicas, como valores dos registradores de máquina (e outras informações críticas) e uma pequena área de dados locais. As *threads* contidas dentro de um objeto são invisíveis fora deste e compartilham os dados "globais" desse objeto: as variáveis de instância, variáveis de classe e variáveis estáticas. As variáveis locais, declaradas dentro dos métodos e restritas a esse escopo, existem em instâncias particulares a cada *thread*.

Objetos comunicam-se através de mensagens e suas informações internas estão encapsuladas, podendo ser consultadas/alteradas por outros objetos apenas indiretamente, através das funções definidas nas interfaces. A sincronização entre objetos está, dessa maneira, baseada em troca de mensagens. As *threads* que executam dentro de um objeto, por outro lado, compartilham o contexto de execução desse objeto, como explicado anteriormente. Portanto, os mecanismos de controle de

concorrência mais naturais para uso das *threads* são aqueles baseados em memória compartilhada (semáforos, monitores, etc.), já que as *threads* representam, nesse aspecto, o mesmo papel dos processos (na acepção dos sistemas operacionais).

Com relação ao nível de concorrência interna, processos são classificados em *seqüenciais*, *quase-concorrentes* e *concorrentes* [Weg90]. Nos processos seqüenciais, há no máximo uma *thread*, de modo que praticamente não há concorrência dentro do processo; um exemplo é o das *tasks* em Ada [DoD83], cuja execução pode ser suspensa apenas para sincronização e troca de dados com outras *tasks* (*rendezvous*). Processos quase-concorrentes apresentam concorrência pela execução combinada de várias *threads*, com a limitação de haver apenas uma delas ativa num dado momento. Monitores [Hoa74] são exemplos de processos quase-concorrentes, já que apenas uma *thread* está executando comandos dentro do monitor, enquanto as demais estão suspensas, ou na entrada do monitor ou por efeito de chamadas às funções *wait()* e *signal()*.

A presença de múltiplas *threads* ativas caracteriza os processos concorrentes, como é o caso de Argus.[1] Para a execução de operações dentro de um guardião, *threads* são ativadas dinamicamente, havendo necessidade de sincronização e serialização apenas quando são feitos acessos a dados compartilhados. Os objetos em Cm terão, da mesma forma que os guardiães Argus, várias *threads* ativas a um mesmo tempo, como resultado do atendimento simultâneo de requisições de serviços, e portanto podem ser classificados como processos concorrentes. Esse é o assunto da próxima subseção.

5.6.2 Modelo de Concorrência em Cm

A concorrência entre objetos, na versão distribuída da linguagem Cm, é feita através de objetos remotos [Dru94, Gon94]. Esses objetos são unidades autônomas de execução e distribuição, interagindo periodicamente com outros objetos através de chamadas remotas de métodos. Os objetos podem, além disso, apresentar concorrência internamente, para possibilitar a execução de várias tarefas simultaneamente. De maneira geral, contudo, podemos dizer que um objeto pode apresentar concorrência interna por decisão de implementação, e que esta decisão não diz respeito aos clientes de tais objetos.

Objetos que não são remotos, i.e., que são criados e executados dentro do contexto de algum objeto, podem ser de natureza passiva ou ativa. Os objetos do tipo passivo têm seu código incorporado ao objeto que os criou, e executam apenas chamadas de métodos; nos intervalos entre chamadas, permanecem sem executar nenhuma tarefa.

Os objetos ativos, depois de criados, passam a executar uma seqüência de comandos particular, ao mesmo tempo em que continuam atendendo chamadas de métodos, quando solicitados. Esses comandos particulares são, basicamente, tarefas necessárias à manutenção do estado interno do objeto, e estão contidas dentro do método *main()*, que distingue objetos ativos dos passivos. A execução dessas tarefas

1. Para maiores detalhes, consulte a subseção 3.2.1 ("Argus").

é intercalada, no tempo, com a execução dos métodos. A tais objetos é associada uma *thread* para executar o método *main()*, no instante da sua criação. A declaração desses objetos especifica essa condição através da palavra reservada `thread`.

Exemplo 5.20. Dada uma classe *A* que define objetos ativos

```
class A<>
...
main()
{
   ..    // tarefas particulares do objeto
}
```

tais objetos são usados dentro de uma classe P da forma

```
class P<>
import .., A, ..

A thread obj_A;
...
```

o que causa a execução do método *main()* do objeto *obj_A* em paralelo com as *threads* do objeto da classe *P*. (Fim do exemplo 5.20.)

Com o construtor de tipos `thread`, ficam definidas as diferentes maneiras pelas quais um objeto em Cm pode ser executado. Um objeto remoto tem um contexto de execução próprio, que não está incluído em nenhum outro contexto; dessa forma, os objetos remotos são sempre visíveis a partir dos outros objetos. Os objetos declarados com o construtor `thread` são uma unidade de execução autônoma, mas estão incluídos dentro do objeto que os declarou, e são portanto invisíveis externamente. E os demais objetos, que são declarados apenas com o nome da classe, têm seus dados e códigos incorporados ao objeto que os declarou. Essas alternativas correspondem aos objetos globais, locais e diretos de Emerald.[1]

Até aqui o paralelismo tem granularidade equivalente a objetos. Outra forma de concorrência, com granularidade mais fina, é a *thread*, que implementa execução concorrente de métodos. Como objetos remotos podem ser compartilhados por vários clientes, pode ser que, num determinado momento, pedidos para a execução de métodos sejam atendidos pelo servidor enquanto um ou mais métodos estão sendo executados. Nossa proposta prevê a definição de objetos com concorrência interna pelo atendimento simultâneo de métodos, como pode-se ver mais adiante, na seção 5.6.3.2 ("Regiões Críticas Condicionais em Cm").

Os métodos têm acesso tanto às próprias variáveis locais como também aos dados do objeto — as variáveis de instância, variáveis de classe e variáveis estáticas. Todos estes dados são compartilhados pelas *threads* que forem ativadas para o atendimento de pedidos de execução de métodos. Mais adiante, veremos como é disci-

1. Ver detalhes na subseção 3.2.3, página 3-9.

plinado o acesso das *threads* aos dados do objeto, para manter seu estado interno coerente.

5.6.3 Controle de Concorrência

A concorrência entre *threads*, resultado do atendimento simultâneo de métodos por um objeto, deve ser disciplinada de forma a garantir a consistência do estado interno do objeto. Como as *threads* compartilham dados do objeto onde estão, o controle de concorrência deve ser baseado em memória compartilhada. Esse controle, em Cm, é feito por um mecanismo baseado em regiões críticas condicionais [Hoa72]. Nesta subseção apresentaremos a forma original da regiões críticas condicionais,[1] seu formato em Cm (com extensões), a análise semântica desse mecanismo e aspectos de implementação.

5.6.3.1 Proposta Original de Regiões Críticas Condicionais

Nesta seção, vamos apresentar as CCRs como definidas em [Hoa72], portanto usaremos a terminologia original, com a acepção tradicional do termo *processo*.

Para o uso de regiões críticas condicionais, variáveis compartilhadas entre processos são associadas a um *recurso*. Usando a notação de [And83], um recurso r contendo variáveis $v_1, \ldots, v_n$ pode ser declarado como

```
resource r: v1, ..., vn
```

Um recurso define um conjunto de dados que serão compartilhados entre vários processos. As variáveis $v_1, \ldots, v_n$ só poderão ser utilizadas dentro de regiões críticas condicionais relativas ao recurso r. A execução paralela de processos $P_1, P_2, \ldots, P_n$, com respeito à utilização de um recurso r em comum, é declarada com a seguinte construção:

```
r: {P1 // P2 // .. // Pn}
```

Uma *região crítica condicional* especifica um trecho de código, dentro de um processo P_i, que é executada em exclusão mútua com os demais processos em relação ao recurso r. Uma CCR é definida pelo comando `with`, que tem o seguinte formato

```
with r when B do S
```

onde B é uma condição lógica e S é um comando. A avaliação de B e a execução de S não são interrompidas por outros processos que usam o mesmo recurso r. A condição B é garantidamente verdadeira quando S inicia sua execução.

Dados dois processos P_a e P_b executando concorrentemente, e compartilhando dados de um recurso r, a não-interferência entre esses dois processos é garantida pelas seguintes regras:

1. Para representar esse termo usaremos muitas vezes a sigla CCR, do inglês *Conditional Critical Region*.

1. Os dados do recurso *r* podem ser utilizados apenas dentro de regiões críticas condicionais relativas a ele;
2. Qualquer dado compartilhado pelos dois processos pertence ao recurso *r*.

Conforme [And83], a CCR combina, em um único mecanismo, os dois aspectos da sincronização entre processos: a *exclusão mútua* (execução exclusiva da CCR) e a *condição de sincronização* (avaliação da condição lógica).

5.6.3.2 Regiões Críticas Condicionais em Cm

As regiões críticas condicionais em Cm são mais genéricas, no sentido de relaxar várias regras contidas na sua forma original — por essa razão são denominadas regiões críticas condicionais estendidas. Há também outra diferença, já que em Cm esse mecanismo diz respeito à concorrência entre *threads*, e não entre processos. Mais adiante detalharemos as extensões feitas e quais as implicações decorrentes.

A notação de uma região crítica condicional estendida é a seguinte

```
with <expr-reg> [ when <cond_sinc> ] do <comando>
```

onde *expr_reg* é uma expressão de região, *cond_sinc* é uma condição lógica e *comando* é um comando, possivelmente composto. As expressões *expr_reg* e *cond_sinc* são referidas, em conjunto, como a *guarda* da CCR; e o comando é a região crítica propriamente dita.

A expressão de região envolve variáveis de região, operadores (&& e =) e constantes inteiras. Uma variável de região é definida com o tipo *region*. Seu valor inicial é chamado de *cardinalidade* da exclusão mútua associada a essa variável, i.e., qual o número máximo de *threads* permitidas dentro de regiões críticas protegidas por essa variável. O valor *default* da cardinalidade de uma variável de região é 1; quando o valor usado na definição da variável é 0, indica um número arbitrariamente grande de *threads*, i.e., cardinalidade infinita.

A condição lógica (expressão precedida da palavra reservada when), que também é chamada *condição de sincronização*, é usada para testar se a região crítica pode ser executada pela *thread* que avalia a guarda naquele momento. A guarda só é avaliada depois de garantida a exclusão mútua em relação às regiões especificadas. Se o resultado da expressão é *true*, então é permitida àquela *thread* a execução do comando; se o resultado é *false*, a *thread* é bloqueada, à espera de uma nova oportunidade para testar a guarda da CCR, e eventualmente executá-la. Neste último caso, a *thread* libera o acesso exclusivo às regiões especificadas na guarda.

Definição 5.7. Um objeto em Cm permite atendimento e execução simultânea de métodos se sua classe for declarada com o termo `threaded`. Esses objetos são denominados *servidores multitarefa*.

Exemplo 5.21. Vamos utilizar como exemplo o problema clássico de leitores e escritores. Um determinado dado é compartilhado por objetos leitores e objetos escritores, sendo que um número fixo de leitores podem atuar concorrentemente, ao passo que um escritor exclui qualquer outro leitor ou escritor. Podemos implementar isso em Cm com a declaração da seguinte classe:

```
threaded class
   Readers_Writers <type T; int NREADERS>

T data;
region rr = NREADERS;

export void write (T x)
{
   with rr = 1 do
      data = x;
}

export T read()
{
   with rr do
      return(data);
}
```

A palavra reservada `threaded` indica que as instâncias da classe *Readers_Writers* suportam execução concorrente de métodos. Cada chamada a *read()* e *write()*, originada de um cliente dessa instância, causa a ativação de uma *thread* com o fim exclusivo de executar o método com os parâmetros oferecidos, e devolver o resultado esperado. (Fim do exemplo 5.21.)

A capacidade de executar simultaneamente métodos transmite-se por herança, pelo menos para os métodos herdados. Entretanto, isso só tem efeito quando a classe herdeira também é declarada como `threaded`, e o controle de concorrência sobre um mesmo conjunto de dados deve ser feito com base nas mesmas variáveis de região. Caso a classe a ser herdada declare algum método como sendo virtual, é mais interessante que essa classe seja declarada como `threaded`, pois nesse caso o método, como implementado pelas subclasses, poderá ser ou não executado concorrentemente com outros métodos dependendo apenas se a subclasse é `threaded` ou não.

Exemplo 5.22. Suponhamos que a classe *A*, que implementa os métodos *m1()* e *m2()*, é herdada pela classe *B*, que por sua vez redefine o método *m2()* e define o método *m3()*. Temos então quatro alternativas:

- *A* e *B* não são declaradas como `threaded`: nenhum dos métodos pode ser executado simultaneamente com os outros, tanto para objetos da classe *A* como para objetos da classe *B*;
- *A* é declarada como `threaded`, mas *B* não: para objetos da classe *A*, os métodos *m1()* e *m2()* podem ser executados concorrentemente, e para objetos da classe *B*, qualquer método é executado em exclusão mútua;
- *B* é declarada como *threaded*, mas *A* não: para objetos da classe *B*, os métodos *m2()* e *m3()* podem ser executados concorrentemente, e *m1()* é executado em exclusão mútua; para objetos da classe *A*, os dois métodos são executados em exclusão mútua;
- *A* e *B* são declarados como `threaded`: para objetos de qualquer das classes, quaisquer métodos podem ser executados concorrentemente.

Na classe *B*, caso a nova versão do método *m2()* ou o método *m3()* usem dados que estão protegidos contra acesso indevido nos métodos *m1()* e *m2()* da classe *A*, deve-se proteger estes dados com base nas mesmas variáveis de região utilizadas na classe *A*, que são herdadas para a subclasse. (Fim do exemplo 5.22).

5.6.3.3 Semântica das Regiões Críticas Condicionais Estendidas

Variáveis de região são objetos da classe *region*, pré-definida na linguagem. Tais variáveis recebem, no momento da sua definição, um valor inicial, denominado *cardinalidade*. A cardinalidade de uma variável de região denota o número máximo de *threads* que podem ser executadas concorrentemente em regiões críticas guardadas por aquela variável. Fazendo uma analogia: cada CCR, para ser executada por uma *thread*, requer que esta tenha "chaves", que estão depositadas em "caixinhas" (as variáveis de região). A avaliação da expressão de região é uma requisição dessas "chaves": quando a avaliação é bem-sucedida, a *thread* começa a executar a região crítica, levando consigo as "chaves" que usou para entrar. As "chaves" são devolvidas para as "caixinhas" quando a *thread* terminar a execução da região crítica e sair da CCR.

As variáveis de região são necessariamente "globais" dentro do objeto (i.e., são declaradas no maior escopo léxico da classe), já que representam informação a ser compartilhada por todas as *threads* criadas dentro do objeto.

A expressão de região envolve variáveis de região, expressões inteiras, operadores de conjunção (`&&`) e atribuição (=). Essas operações têm a seguinte semântica:

- `&&` combina, em uma expressão de região, mais de uma variável de região. Esse operador é útil para evitar a construção de comandos `with` encadeados quando isso não for estritamente necessário.
- = atribui temporariamente uma nova cardinalidade a uma variável de região, durante a execução da CCR. A cardinalidade efetiva da variável de região será o menor valor dentre o valor inicial (atribuído no momento da declaração da variável de região) e as cardinalidades temporárias associadas por todas as CCRs em execução.

Exemplo 5.23. Supondo as seguintes declarações

```
region r1 = 1, r2 = 2, r3 = 3;
region[10] rv;
region* rp;
```

As seguintes expressões de região têm o correspondente significado (vamos omitir a condição de sincronização por clareza):

```
with r1 && r2 do …
```

acesso deve ser garantido por *r1* e *r2*, com cardinalidades iguais a 1 e 2, respectivamente.

```
with r3 = 2 && *rp do …
```

acesso deve ser garantido por *r3* (com cardinalidade restrita a 2 durante as execução da CCR) e por **rp* (a cardinalidade foi atribuída no momento da criação da variável de região com a função new).

```
with rv[j] = 3 do ...
```

o acesso é garantido por *rv [j]*, com cardinalidade 1. O valor temporário de 3 para *rv [j]* não faz efeito porque a cardinalidade original da variável é 1; a redefinição de cardinalidade só tem efeito, portanto, se representar restrição de concorrência. (Fim do exemplo 5.23.)

Uma *thread* executa uma CCR após satisfazer a expressão que limita o grau de concorrência (expressão de região) e a condição de sincronização. Dentro da CCR, a *thread* leva consigo as "chaves" que usou para entrar; pode ser possível, portanto, entrar novamente em outra CCR, cuja guarda especifique variáveis de região para as quais a *thread* já tem uma "chave".

Exemplo 5.24. Considerando o seguinte trecho de código:

```
region r = 5;
...                    // 1
with r = a {
   ...                 // 2
   with r = b {
      ...              // 3
   }
...
}
...
```

Os valores da cardinalidade da variável r, à medida que esse trecho vai sendo executado, é a seguinte: em (1), *r = 5*; em (2), *r = min(5, a)*; e em (3), *r = min(5, a, b)*. A entrada em CCRs encadeadas, contendo uma mesma variável de região, não implica na necessidade de múltiplas "chaves", ou seja, a *thread* pode entrar em todas as regiões para as quais já tiver as "chaves". (Fim do exemplo 5.24.)

Há dois casos especiais que têm representações especiais da CCR. A expressão de região `all` representa, para todas as variáveis de região definidas dentro do objeto, uma redefinição temporária da cardinalidade para 1. O outro caso especial são regiões críticas que devem ser executadas de forma atômica: essas CCRs não têm expressão de região.

Exemplo 5.25. A CCR

```
with all when i > j do
   something_important();
```

é executada em exclusão mútua com todas as demais CCRs do objeto. Já a CCR

```
with when i == j do
   something_critical();
```

é executada da seguinte maneira: todas as demais *threads* do objeto são suspensas, a CCR é executada, e as demais *threads* são então reativadas. Obseve-se a ausência de expressão de região: pode ser que alguma *thread* do objeto esteja executando uma CCR, mas isso não é problema, o importante é que estarão todas suspensas enquanto esta CCR não for terminada. (Fim do exemplo 5.25.)

A execução de uma região crítica de forma atômica corresponde a um nível privilegiado de execução, concedido temporariamente à *thread*. Esse nível de execução é o mesmo do sistema de suporte, que manipula dados críticos do objeto e portanto não pode sofrer interferências durante essas operações.

5.6.3.4 Diferenças entre a proposta original e as CCRs estendidas

Na proposta original, o comando `with` especifica uma CCR relativa a um recurso, que é um conjunto de dados compartilhados. Um recurso é definido formalmente, e se assemelha a um registro com vários campos. Variáveis pertencentes a um recurso não podem ser utilizadas fora de regiões críticas relativas a esse recurso, e dados não podem ser compartilhados se não pertencerem a nenhum recurso.

As principais diferenças entre a notação original e a nossa proposta para Cm são:

- não existe a definição de recurso, e sim de variáveis de região, e não há vínculo formal entre essas variáveis e os dados compartilhados entre as *threads*;
- o nível de concorrência dentro de uma CCR em Cm pode ser arbitrário, ao passo que, na proposta original, a execução de uma CCR relativa a um recurso é sempre em exclusão mútua com outras CCRs do mesmo recurso (i.e., cardinalidade fixa em 1);
- o nível de concorrência pode ser alterado dinamicamente, através da redefinição temporária da cardinalidade das variáveis de região;
- a exclusão com base em mais de uma variável de região pode ser feita em uma única CCR através do operador `&&`, e na proposta original a exclusão com base em mais de um recurso é feita com comandos `with` encaixados.

De um ponto de vista teórico, a proposta apresentada representa um mecanismo mais flexível, já que não há vinculação explícita entre as variáveis de região e os dados que devem ser protegidos pelas CCRs[1] e também porque as CCRs podem apresentar um nível arbitrário de concorrência. A possibilidade de alteração dinâmica da cardinalidade das variáveis de região simplifica a resolução de certos problemas.

Do ponto de vista prático, o maior impacto dessas diferenças está na estratégia de teste das guardas, que deve está a cargo do mecanismo de controle de concorrência. A estratégia de teste da proposta original é bastante simples: quando um processo deixa uma CCR, os processos que estão esperando nas guardas relativas àquele

1. Quer dizer, deve haver uma relação, implicitamente estabelecida pelo programador, entre uma variável de região e um conjunto de dados, que são alterados de forma consistente dentro de regiões críticas protegidas por essa variável. O que não há é uma maneira explícita, formal e passível de verificação dessa correspondência pelo compilador.

recurso são testados. Esse teste é suficiente porque a condição de sincronização usa apenas variáveis do recurso, e portanto passa de *false* para *true* apenas dentro de CCRs.

As CCRs em Cm, no entanto, são mais genéricas que a proposta inicial, e apresentam as seguintes características:

- dentro de uma CCR pode haver várias *threads*, tantas quantas permitidas pelas cardinalidades variáveis de região da guarda;
- não há relação explícita ou formal entre variáveis de região e os dados que estão protegidos dentro das regiões críticas;
- as condições de sincronização envolvem variáveis que não estão necessariamente protegidas por regiões críticas, podendo passar de *true* para *false* tanto dentro como fora das regiões.

Essas diferenças tornam mais complexo o mecanismo de escalonamento de *threads*, principalmentre porque não há vinculação entre as variáveis de regiões e os dados manipulados dentro das CCRs ou envolvidos nas condições de sincronização. Mais adiante, na subseção 5.6.3.7 ("Implementação"), comentamos mais aspectos desse problema.

5.6.3.5 Semântica das condições de sincronização

A condição de sincronização, parte opcional da região crítica condicional, representa um pré-requisito para a execução da região crítica. O momento em que a guarda da CCR é avaliada não é sabido de antemão, nem quantas vezes a avaliação é feita. Se houver, na condição de sincronização, efeitos colaterais, o resultado da execução da CCR pode variar em função do número de vezes que a guarda da CCR foi avaliada. Isso é indesejável, potencialmente perigoso, e portanto seu uso é desaconselhável.

Como determinar se uma expressão tem efeitos colaterais? Em muitos casos, principalmente se há chamadas de função envolvidas, não há como afirmar conclusivamente, já que isso depende da execução do programa. Uma análise exaustiva desse tipo de situação é muito difícil ou mesmo impraticável, principalmente levando em conta a presença de apontadores.

A análise de uma expressão, à procura de efeitos colaterais, deve levar em conta as seguintes possibilidades:

- operações de atribuição, pré/pós incremento/decremento;
- chamada para funções com parâmetros passados por referência, e alteração desses parâmetros dentro das funções;
- chamada para funções que alteram dados compartilhados pelas *threads*, ou seja, variáveis de instância, variáveis de classe e variáveis estáticas.

A primeira alternativa é simples de ser verificada, o que não ocorre com as duas restantes. Pode haver chamadas para outras funções, chamadas envolvendo apontadores para funções, e chamadas para métodos virtuais. Proibir a chamada de funções é muito restritivo, já que isso também excluiria funções *inline* e operadores superpostos (*overloaded*). Entretanto,o compilador de Cm pode, ao analisar uma CCR, gerar

mensagens de advertência caso seja detectada a certeza ou possibilidade de efeito colateral.

5.6.3.6 Exceções

Na versão distribuída da linguagem Cm, a semântica das exceções será estendida para o caso distribuído. Dados dois objetos distribuídos, um servidor e um cliente, se durante a execução do serviço ocorrer uma exceção que não puder ser devidamente tratada pelo servidor, ela será transparentemente propagada para o cliente, como "resultado" pela execução do serviço. Para mais detalhes consulte a subseção 4.4.3 ("Exceções") e a seção 5.3 ("Exceções em objetos distribuídos").

O mecanismo de tratamento de exceções da linguagem Cm está baseado nas macros *setjmp()* e *longjmp()* da biblioteca padrão C; com a implementação de concorrência na linguagem, deverão ser utilizadas as funções *sigsetjmp()* e *siglongjmp()*, que operam corretamente em conjunto com os sinais do Unix [Tel93]. A transferência do fluxo de execução, no caso de uma exceção, se faz considerando a seqüência de chamadas da *thread* que executou o comando causador da exceção; não há, portanto, risco de interferência entre as *threads* de um mesmo objeto.

O mecanismo de tratamento de exceções também será integrado às regiões críticas condicionais. Isso é importante pois em aplicações distribuídas deve-se sempre ter em mente as inúmeras possibilidades de falhas no sistema, e uma postura muito rígida por parte dos mecanismo de suporte pode comprometer a flexibilidade e robustez das aplicações. No caso, as CCRs poderão gerar exceções quando ocorrerem situações em que o funcionamento normal das *threads* estiver comprometido. Podemos citar as seguintes situações:

- no caso mais simples, quando a avaliação da guarda (que pode conter chamadas para funções) gerar uma exceção, ou quando a execução da região crítica gerar uma exceção;
- quando uma determinada *thread* estiver bloqueada por mais tempo que um valor pré-definido;
- quando a avaliação da condição de sincronização der resultado *false*, e a expressão envolver termos constantes, ou variáveis locais, de maneira que o resultado não possa ser diferente de *false* em qualquer outra avaliação.

5.6.3.7 Implementação

O mecanismo de controle de concorrência do Cm faz parte do ambiente de suporte de execução da linguagem, e será implementado de forma integrada às demais funcionalidades como criação/destruição de objetos distribuídos, ativação remota de métodos e tratamento de exceções. O mecanismo está embutido na linguagem, com construções sintáticas próprias, o que implica em mudanças no compilador; tais mudanças, porém, são pequenas, pois se trata apenas de mais um comando.

Todavia, o compilador terá um papel muito mais importante: fazer uma análise estática das CCRs de uma classe, estabelecendo relações de dependência entre variáveis de região e variáveis do objeto que são compartilhadas pelas *threads*. O objetivo dessa análise é otimizar, na medida do possível, o trabalho a ser feito pelo sistema de suporte em tempo de execução, oferecendo informações úteis para o escalonamento de *threads* e os testes das guardas de CCRs.

Além disso, outro trabalho que deve ser feito pelo compilador é detectar casos de terminação anormal de uma região crítica, i.e., casos em que a execução de uma *thread* é desviada para fora da região crítica. Isso acontece com o comando `raise` (ativação e reativação de exceções), `break` e `continue` (transferência de controle para fora dos comandos `do`, `for`, `while` e `switch`), e `return` (retorno de função). Já existe, dentro do compilador, procedimentos específicos para essas situações, pois a entrada em um bloco de comandos pode causar efeitos colaterais (alocação de memória dinâmica, e. g.) com a criação de objetos, efeitos que devem ser desfeitos ao término, normal ou não, da execução do bloco correspondente.

As funções básicas requeridas pelo mecanismo de controle de concorrência do Cm — como criação, destruição e escalonamento de *threads* — ainda não estão inteiramente definidas. No entanto, essa interface não será sofisticada, pois o programador não poderá influir na execução das *threads*; além disso, é necessário facilitar o porte do ambiente de execução da linguagem para outros sistemas operacionais. A implementação do protótipo da versão distribuída do Cm será feita com base no sistema Solaris (versões mais recentes do SunOS [Sun93]). No futuro, nosso objetivo é utilizar o serviço de *threads* proposto pelo padrão OSF/DCE [OSF90].[1]

Teste das guardas

As guardas das CCRs determinam, de maneira estática, as condições que devem ser satisfeitas antes da execução de regiões críticas. O objetivo da guarda é duplo: garantir acesso controlado e limitado a um conjunto de dados protegidos, e ordenar no tempo a execução de regiões críticas, com base na condição de sincronização. Estabelecidas estas restrições, não se pode prever a ordem exata em que as *threads* vão ser executadas, a partir do recebimento de requisições de serviços nesse objeto.

Cabe ao ambiente de execução arbitrar o comportamento das *threads*. Isso significa decidir quais *threads* vão executar regiões críticas, e em que momentos a entrada de *threads* em regiões críticas deve ser testada. Pretender que as *threads* concorram em igualdade absoluta de condições (i.e., que o mecanismo de controle de concorrência seja *justo*) é coisa muito difícil de obter. Não se pode fazer nenhuma suposição quanto às suas velocidades relativas de execução, portanto umas terão acesso a recursos mais rapidamente e mais freqüentemente que outras. Além disso, uma CCR que exija exclusão mútua com relação a um grande número de variáveis de região ou uma condição de sincronização muito forte vai penalizar as *threads* que a executam com um tempo de espera (provavelmente) maior que a média. Uma análise muito criteriosa dessas situações implicaria em algoritmos complexos e demorados, às custas do tempo de processamento que deveria estar disponível para as *threads*.

É necessário, de qualquer maneira, definir os momentos em que se faz o teste de guarda das *threads* bloqueadas. As alternativas mais naturais são:

- quando uma *thread* atinge uma CCR;

1. Considerações sobre implementação estão no capítulo 6 ("Conclusões").

- quando uma *thread* deixa uma CCR.

Essas duas situações são as mais adequadas porque, nesses momentos, o controle é transferido para o sistema de suporte à execução, que vai executar os procedimentos necessários em relação ao futuro imediato da *thread* que estava ativa, e executar as alterações necessárias no estado das variáveis de região.

Quando uma *thread* deixa uma CCR, as *threads* que estão bloqueadas em variáveis de região da guarda dessa CCR são candidatas naturais ao teste da guarda. Isso porque a *thread* que saiu representa uma "vaga" que pode ser preenchida por alguma ou algumas *threads* bloqueadas. O preenchimento da vaga não é certo, pois uma *thread* pode estar bloqueada em outras variáveis de região, ou na condição de sincronização.

Se não há *threads* que podem se aproveitar do término de uma CCR, deve-se passar à análise de *threads* bloqueadas em outras variáveis de região, já que pode haver algumas delas bloqueadas exclusivamente na condição de sincronização, que podem mudar de estado em qualquer ponto do programa. O tempo de espera das *threads* bloqueadas deve ser levado em conta na análise dessas outras variáveis de região.

Regiões críticas especiais

Para uma dada classe C é criada, dentro das instâncias dessa classe, uma variável de região especial, denotada por R_c, de tal forma que a cardinalidade de R_c é uma constante K ($K > 1$ e dependente de implementação) caso a classe C seja declarada como `threaded`, e $K = 1$ no caso contrário.

Para regiões críticas que devem ser executadas atomicamente, será criada uma variável de região especial, aqui denominada de R_{atomic}, que controla o teste de guardas e execução atômica de *threads* dentro dessas regiões.

Estruturas de dados

As guardas são transformadas, pelo compilador, em estruturas de dados a serem usadas por um avaliador de expressões do sistema de suporte, para o teste das mesmas em tempo de execução. Esse avaliador supre todas as operações da linguagem Cm, necessárias para a avaliação das condições de sincronização.

As variáveis de região estão mapeadas em estruturas manipuladas pelo sistema de suporte a execução. Para cada variável de região, há uma lista de *threads* bloqueadas e uma lista de *threads* executando CCRs guardadas pela variável. As *threads* estarão, nestas listas, ordenadas por tempo de espera, o que implica em uma política FIFO no caso de guardas sem condição de sincronização. As variáveis de regiões especiais, que são a variável de região da classe e a variável de região das CCRs que pedem execução atômica, também estão representadas dessa forma, mas não podem ser usadas por programa.

Análise de CCRs pelo compilador

O compilador de Cm colabora com o mecanismo de controle de concorrência fazendo uma análise estática das CCRs, com respeito às variáveis de região presentes nas guardas e as variáveis do objeto que são alteradas dentro das regiões críticas. O objetivo dessas análises é detectar potenciais situações de erro (efeitos

colaterais nas condições de sincronizacão, e. g.) e fornecer informações para otimizar os procedimentos de escalonamento das *threads*, em tempo de execução.

Definição 5.8. Dada uma região crítica R, o conjunto $K(R)$ representa as variáveis de região que controlam a execução de R, ou seja, K(R) = {v | v é uma variável de região presente na guarda de R}.

É importante lembrar que variáveis de região podem fazer parte de estruturas, serem usadas através de apontadores ou como elementos de um vetor. Nesse último caso, por exemplo, $K(R)$ contém o nome do vetor, ou seja, todos os seus elementos, porque não há como saber, antes da fase de execução, qual variável de região será realmente usada.

Definição 5.9. Duas CCRs são *competidoras* se as suas execuções dependem de um conjunto comum de variáveis de região. Dadas duas CCRs R_a e R_b, elas são competidores se $K(R_a)$ e $K(R_b)$ têm interseção não vazia.

Se não há competição por variáveis de região nem por dados comuns do objeto, então as regiões são "indiferentes" entre si.

Definição 5.10. Dada uma CCR R, o conjunto $W(R)$ representa todos os dados do objeto que são ou que podem ser alterados pela execução de R (inclusive a guarda, caso haja efeitos colaterais).

A interferência indevida pode ocorrer durante a execução de CCRs que utilizam dados globais em comum mas que têm o acesso garantido por variáveis de região diferentes, e que portanto não serão executadas (a não ser dependendo de exclusão pela condições de sincronização, hipótese que não pode ser verificada praticamente pelo compilador) em exclusão mútua.

Definição 5.11. Duas CCRs são *conflitantes* se utilizam dados comuns do objeto mas não têm variáveis de região em comum. Assim, duas CCRs R_a e R_b são conflitantes se a intersecção de $K(R_a)$ e $K(R_b)$ é vazia e $W(R_a)$ e $W(R_b)$ têm intersecção não vazia.

O compilador pode auxiliar na detecção de potenciais problemas de interferência indevida entre *threads* através de mensagens de advertência na presença de CCRs conflitantes. Vale ressaltar que todas essas considerações dependem da decisão de vincular ou não, através de algum mecanismo sintático, as regiões críticas condicionais a conjunto de dados que devem ser protegidos. Esta proposta, levando em conta que é difícil determinar, *a priori*, se a nossa abordagem é mais vantajosa que a proposta original, decide adotar a alternativa mais genérica.

Considerações finais

Em [Hoa74] há um conjunto de princípios que o autor prescreve para a implementação de um mecanismo de concorrência específico (monitores). Na nossa opinião, tais princípios são genéricos o bastante para uma discussão genérica sobre o tema, e

são excelentes políticas para conseguir um bom resultado com qualquer mecanismo de sincronização. Destacamos entre eles:

- não procurar soluções sempre ótimas, tentar simplesmente evitar que sejam sistematicamente péssimas;
- manter os tempos de espera baixos e aproximados entre si;
- evitar prioridades fixas, objetivando sempre que os processos envolvidos, em geral, estejam progredindo em sua execução;
- manter o comportamento do sistema satisfatório em condições limite (*race conditions*); e
- definir regras para o uso correto e equilibrado do mecanismo, assumindo que os programadores as respeitarão.

5.7 Análise comparativa do nosso modelo

O modelo de objetos distribuídos apresentado nesse capítulo, e proposto para a linguagem Cm, é um ponto de partida para possibilitar, dentro do ambiente A_Hand, o desenvolvimento sistemático de aplicações com pleno suporte de mecanismos de programação distribuída. Depois que esse modelo estiver implementado, em uma primeira versão distribuída da linguagem Cm, esperamos obter um razoável *feedback* para considerar modificações e extensões a esse modelo. Por esse *feedback* entendemos a concepção, desenvolvimento e uso intensivo de aplicações ou ferramentas compostas por grande número de objetos distribuídos (dezenas ou mesmo centenas deles).

Ao formalizar esta proposta de extensão da linguagem, é conveniente, entretanto, situar o nosso trabalho junto a outros trabalhos da área, que estejam propostos ou já implementados. Para isso faremos uma análise comparativa, entre o nosso trabalho e esses outros, considerando uma série de aspectos relacionados a linguagens de programação orientadas a objetos e a sistemas distribuídos.

5.7.1 Propostas de Programação Distribuídas Orientada a Objetos

Além dos sistemas Argus, Chorus e Emerald, comentados na seção 3.2 ("Análise de sistemas distribuídos"), existem outras propostas de ambientes de programação distribuída baseados no conceito de objetos. Dentre essas propostas, vamos analisar algumas que consistem, basicamente, na extensão de linguagens de programação orientadas a objetos já implementadas e utilizadas; afinal, é isso que estamos fazendo com relação a Cm. Consideramos três propostas: COOL (Chorus Object-Oriented Layer), baseado no sistema Chorus e independente de linguagem; e Separate Entities e Eiffel//, baseadas na linguagem Eiffel.

5.7.1.1 COOL

O projeto COOL [Hab90, Lea93] busca a implementação de um sistema distribuído orientado a objetos. Nesse sistema, um modelo genérico de objetos é suportado, de forma que não há um compromisso com uma linguagem de progranação específica; os artigos analisados, entretanto, descrevem o mapeamento das estruturas do COOL em C++.

A idéia fundamental do COOL é aproximar as abstrações oferecidas pelo Chorus dos mecanismos de programação embutidos nas linguagens orientadas a objetos. Isso significa implementar, acima do Chorus, mecanismos de suporte genérico à programação com objetos que possam ser eficientemente utilizados pelas linguagens de programação. Tais mecanismos são, por exemplo, a criação e destruição de objetos, chamada transparente de métodos e persistência de objetos.

O sistema COOL é implementado através de camadas de *software*, utilizando o Chorus como fornecedor de serviços básicos. São três camadas: a base (denominada COOL-base), o *Run-time System* genérico (GRT) e a camada dependente de linguagem de programação. Vamos descrever a estrutura e a funcionalidade dessas camadas, e a máquina virtual que oferecem para o nível superior.

A camada base atua diretamente sobre o núcleo Chorus, estendendo-o de forma a ser um *microkernel* para sistemas orientados a objetos. Essa camada implementa serviços mais elaborados a partir das primitivas do núcleo Chorus, e oferece esses serviços através de uma interface composta por *system calls*. Esse serviços são, basicamente, mecanismos de mapeamento de objetos em memória, transmissão de mensagens e um sistema de execução baseado nas *threads* Chorus.

Com relação ao gerenciamento de memória, são introduzidas duas abstrações: *clusters* e espaços de contextos. Um *cluster* define onde objetos vão residir, e consiste de um conjunto de regiões de memória que são dinamicamente mapeados em contextos. Um contexto é o mesmo que espaço de endereçamento, e corresponde a um *actor*, uma das abstrações do núcleo Chorus.[1] Um espaço de contexto compreende vários contextos, que podem estar divididos entre vários *sites*. O mapeamento de um *cluster* em um espaço de contexto é o mapeamento de cada componente do *cluster* em todos os contextos desse espaço. Um *cluster* é, além disso, a unidade de persistência de dados do sistema, e a camada base controla a transição transparente de suas informações entre memória principal e secundária. Um espaço de contextos representa, assim, um espaço de endereçamento virtual e distribuído.

A camada de *Run-time* genérica (GRT) implementa a abstração de objeto, e as correspondentes funções de criação, chamada transparente de métodos e associação com os espaços de contextos. Objetos são identificados de duas formas: uma referência *domain-wide*, que é única, global e persistente; e uma referência de linguagem, válida dentro do contexto onde o objeto está mapeado num dado momento.

Nessa camada também são definidos dois novos conceitos: atividade e *job*. Uma atividade é uma *thread* criada dentro de um objeto, podendo passar para outros objetos, situados em outras máquinas, através de chamadas de métodos. Um *job* é definido como um conjunto de contextos, e tipicamente representa uma aplicação distribuída. Os contextos de um *job* vão servir para o mapeamento de *clusters*, cada um podendo conter vários objetos relacionados, e com várias atividades em execução simultaneamente. Para o controle de concorrência, o GRT fornece mecanismos tanto para sincronização entre atividades em um mesmo contexto (semá-

1. Mais detalhes consulte-se a seção 3.2.2 ("Chorus").

foros e *locks* de leitura/escrita) como para sincronização de objetos (algoritmos distribuídos baseados em *tokens*).

A ultima camada do COOL é dependente de linguagem, permitindo suporte adequado a aplicações com objetos desenvolvidos, por exemplo, em C++, Eiffel e outras. Os objetos, como implementados pelo GRT, são genéricos, e portanto cabe a essa camada mapear o modelo de objetos específico da linguagem, com sua semântica associada, no modelo do GRT. São usados pré-processadores para gerar *stub procedures*, cuja função é fazer a conversão das chamadas de métodos para chamadas de funções do GRT, e fornecer as funções *up-call* para acesso do GRT a informações dos objetos.

Em [Lea93], é citado um pré-processador chamado COOL++, que suporta uma extensão da linguagem C++ padrão, e que processa código a ser executado no ambiente provido pelo GRT. Os autores não detalham essa extensão (citam apenas duas palavras reservadas) nem dão exemplos do seu uso. O COOL++ gera classes especiais que serão usadas para instanciar os chamados objetos de interface, usados para mapear chamadas de métodos em ativação remotas ou locais, conforme a situação do objeto servidor.

5.7.1.2 Separate Entities

Essa proposta de Meyer está exposta em [Mey93] e considera uma extensão da linguagem Eiffel para programação concorrente e distribuída. Diferentemente da abordagem do COOL, essa proposta está baseada em uma análise cuidadosa da programação concorrente do ponto de vista das linguagens orientadas a objetos e da engenharia de *software*. Não há, por exemplo, uma implementação do modelo proposto e nem comentários visando essa implementação. Entretanto, o artigo é excelente, e pensamos que irá se tornar uma referência clássica nessa área, pela abordagem segura das questões envolvidas e pela elegância das soluções. Nessa seção, usaremos o termo "Entidades Separadas" para designar as "Separate Entities" do texto original.

A preocupação essencial da proposta é alterar, da maneira mais simples possível, uma linguagem de programação orientada a objetos, para atender aos requisitos da programação concorrente e distribuída. Depois de destacar as semelhanças entre processos e objetos, e a conveniência da integração entre esses dois conceitos, o autor assinala que as tentativas já feitas nesse sentido são muito complexas, ressentindo-se inclusive de uma melhor compreensão teórica do assunto. Antes de desenvolver uma solução, são apresentados vários requisitos a serem considerados para a extensão da linguagem.

A modificação da linguagem deve ser a mínima possível, dos pontos de vista sintático e semântico. Isso é necessário para que não haja redundância ou conflito entre as técnicas da programação concorrente e as técnicas da OOP. Além disso, as técnicas da OOP (herança, polimorfismo, classes abstratas) devem ser utilizadas sem restrições, pois são ferramentas conceituais poderosas para a concepção e desenvolvimento de *software*.

Do ponto de vista da engenharia de *software*, vários aspectos devem ser considerados para que o desenvolvimento de aplicações concorrentes e distribuídas possa ser feito de forma sistemática e segura. Entre esses aspectos temos a necessidade de

axiomas para prova formal da correção de programas, a utilização de bibliotecas de classes que implementam diferentes estilos de programação concorrente, e a reusabilidade de *software* não-concorrente.

Essa proposta (Entidades Separadas) depende da redefinição, por parte do autor, de um conceito fundamental, o processador. A identificação pura e simples de processo e objeto leva, na sua opinião, a soluções muito complexas (por questões como controle de concorrência, e.g.) e à perda do poder de expressão e versatilidade dos objetos. Os processadores servirão, portanto, para definir a diferença fundamental entre a programação seqüencial e a programação concorrente. Um *processador* é definido como um fluxo virtual de controle, que executa operações de forma seqüencial em um ou mais objetos. Na prática, um processador pode ser mapeado em uma CPU (de uma máquina mono ou multiprocessada), em um processo Unix ou em um *lightweight process*.

Dessa definição vêm as *entidades separadas*, o conceito que baseia a proposta. Uma entidade é *separada* se ela, ou sua classe, for declarada com a cláusula `separate`. Usamos o seguinte trecho de [Mey93] para exemplo (os comentários são nossos):

```
x: SOME_TYPE;              -- x é uma entidade comum
y: separate  SOME_TYPE;    -- y é uma entidade separada
```

a diferença entre os dois casos é que o objeto representado por y, depois de criado, será executado em outro processador, diferente do processador onde está o objeto que contém essa declaração. Objetos referenciados por entidades separadas são denominados *objetos separados*, e uma chamada para um objeto separado é uma *chamada separada*. Essa é a única extensão sintática prevista (a palavra reservada `separate`): a *feature call* continua com o mesmo formato, quer o objeto servidor esteja separado ou não. A diferença no efeito da chamada é que, para objetos cliente e servidor executados no mesmo processador, a chamada é sempre bloqueante; no caso de objetos separados, há um paralelismo potencial entre eles, já que estão em processadores diferentes.

Para garantir o acesso correto a um objeto separado, toda chamada a métodos desse objeto (chamadas separadas) deve estar dentro de uma rotina, sendo que o objeto separado em questão deve ser um parâmetro dessa rotina. No momento em que essa rotina é chamada, o objeto separado é "reservado", de forma que durante a execução da rotina o cliente tem acesso exclusivo ao objeto, sem preocupar-se portanto com a possibilidade de acessos concorrentes de outros clientes, o que poderia afetar a consistência do estado do objeto separado. Portanto, uma chamada a uma rotina cujos parâmetros representam objetos separados pode bloquear até que o acesso reservado a esses objetos esteja garantido.

As chamadas separadas são, no caso geral, assíncronas, permitindo que o objeto cliente e o objeto chamado executem em paralelo. Essa regra tem duas exceções: se a chamada tiver parâmetros que representam objetos locais, e se a chamada especifica uma função ou atributo do objeto separado. No primeiro caso, o objeto local referenciado pelo parâmetro será representado, no objeto separado, também como um objeto separado; assim, a chamada "reserva" esse objeto para o objeto separado, de forma que o cliente fica bloqueado esperando pelo término da chamada e a con-

seqüente liberação do objeto. No segundo caso, a função ou atributo é, tipicamente, uma consulta ao estado do objeto, que pode depender da prévia execução das rotinas chamadas anteriormente, na ordem em que foram chamados (isso deve ser garantido para preservar a semântica); portanto, a chamada bloqueia até a devolução do resultado, o que significa que não há, nesse momento, nenhuma chamada separada pendente ou incompleta para aquele objeto.

O mecanismo de "reserva" de objetos separados pode ser alterado para permitir "mensagens urgentes", casos em que um determinado cliente, de alta prioridade, não pode esperar pela liberação de um objeto separado. Nesses casos, a classe *CONCURRENCY* fornece rotinas para modificar os procedimentos normais de atendimento de chamadas. Esses mecanismos especiais estão integrados ao mecanismos de exceções para garantir a semântica do Projeto por Contrato:[1] um cliente pode receber uma exceção sinalizando o término anormal de uma operação, quando na realidade o objeto separado precisou atender a uma requisição urgente. Como o mecanismo de tratamento de exceções da linguagem é muito simples, procedimentos mais sofisticados estão disponíveis na classe *EXCEPTIONS*.

5.7.1.3 Eiffel//

Essa proposta de [Car93] é bem mais pragmática que a de "Separate Entites", e na verdade ela já tem uma implementação em ambiente Unix. Essa extensão não depende de nenhuma alteração na linguagem propriamente dita (não há novas palavras reservadas ou novas construções sintáticas), e sua semântica está baseada apenas em classes pré-definidas e no sistema de suporte a execução.

O termo *processo* é definido como uma instância de uma classe herdeira, direta ou indireta, da classe *PROCESS*. Essa classe oferece vários métodos para controlar o recebimento e execução de requisição de serviços. A rotina *Live*, definida nessa classe, é posta para execução no momento da criação de um processo, e define uma seqüência de operações que devem ser executadas enquanto o processo existir. Um processo termina pelo fim da execução de *Live* ou quando ele não puder mais ser utilizado por nenhum outro processo (procedimento de *garbage collection*).

Em Eiffel//, os processos são objetos, mas o contrário não é necessariamente verdadeiro. Há uma clara distinção entre objetos ativos (processos) e objetos passivos: os primeiros executam ações desde o momento de sua criação, e os outros esperam por chamadas para a execução de métodos. Um objeto passivo é de acesso exclusivo ao processo que o declarou, pois ele pertence ao contexto desse processo, e é invisível fora dele. Dessa forma, objetos podem ser diretamente compartilhados por vários processos apenas se forem também declarados como processos. A distinção entre processos (objetos ativos) e objetos comuns (passivos) está na herança da classe *PROCESS*.

A comunicação entre processos é unificada com o mecanismo de *feature call*. A chamada *obj.m()*, caso *obj* seja um processo, representa tanto uma chamada de uma rotina como uma transferência de dados e controle entre os contextos de dois pro-

1. Para mais detalher ver seção 2.3.3 ("Eiffel").

cessos. Parâmetros da chamada que sejam processos são representados por uma referência para os mesmos, ao passo que objetos comuns são copiados.

A semântica de uma *feature call* de um processo é composta de duas partes: *transmissão síncrona* e *serviço assíncrono*. Quando a chamada é feita, é gerada uma exceção no processo servidor, interrompendo sua execução: nesse momento, um objeto especial representando a chamada é copiado do cliente para o processo chamado. Esse objeto especial especifica a rotina e os parâmetros correspondentes que fazem parte da chamada. Executado esse pequeno protocolo (a "transmissão síncrona"), tanto o cliente como o processo chamado retomam sua execução.

Depois que a requisição foi recebida pelo processo, ela será tratada quando isso for possível ou conveniente ("serviço assíncrono"). O método *Live* obedece a uma política FIFO para o atendimento das requisições, e uma série de facilidades adicionais são oferecidas pela classe *PROCESS*, como por exemplo consulta e manipulação da lista de requisições, e funções de temporização. Está implícito que os processos são seqüenciais, i.e., têm apenas um fluxo de execução (*thread*).

Como a *feature call* é assíncrona, há necessidade de definir um mecanismo de sincronização entre os processos. O autor define o mecanismo como *wait-by-necessity*: um processo pode ser bloqueado apenas quando quiser utilizar o resultado de uma chamada que ainda não foi completada. As chamadas são modeladas como objetos, e quando um objeto representa uma chamada não terminada ele é denominado um *awaited object*. Tal objeto pode ser usado em uma operação de atribuição, e passado como parâmetro de uma rotina, tudo isso enquanto a chamada se processa; quando o valor desse objeto (i.e., o resultado da chamada) for realmente utilizado, então o processo é bloqueado. A utilização de tal mecanismo é automática, dispensando qualquer construção sintática especial.

A implementação de Eiffel// mapeia cada processo (objeto de classe herdeira de *PROCESS*) em um processo Unix, e a comunicação entre processos é feita através de *sockets*. Facilidades de *multithreading* podem ser usadas para diminuir a granularidade de concorrência das aplicações.

5.7.1.4 Comparação das propostas

Em [Hab90], os sistemas distribuídos orientados a objetos são divididos em dois grupos: os sistemas que oferecem um modelo uniforme de objetos e sistemas que oferecem um modelo não uniforme. A diferença em que se baseia essa definição é a maneira como objetos são utilizados em função de suas propriedades, como escopo, granularidade, ou o fato de ser ativo ou passivo.

Nos sistemas com modelo de objetos uniforme (Emerald é citado como exemplo), a interface para a utilização de objetos independe das suas características, o sistema é integrado a uma linguagem de programação orientada a objetos, e é possível definir objetos com granularidade bastante fina. Já nos sistemas com modelo não uniforme (Argus, e.g.), há diferenças entre os chamados *objetos de sistema* e os *objetos de linguagem*: os primeiros são ativos, têm visibilidade global e alta granularidade (geralmente definem um espaço de endereçamento); os segundos são passivos, têm escopo local e média/baixa granularidade — estruturas de dados, basicamente.

Podemos analisar a inserção na nossa proposta nesse panorama, embora Cm seja uma linguagem de programação e não um sistema. Teremos um sistema distribuído e orientado a objetos quando esta proposta estiver implementada e quando a LegoShell for a interface de programação oferecida para o usuário. Esse ambiente oferecerá um modelo de objetos uniforme, no sentido de que não haverá diferença entre "programação de sistema" e "programação de aplicações". As linguagens Cm e LegoShell utilizarão o mesmo modelo de objetos, e os programas escritos em uma ou outra linguagem se combinarão conforme o modelo de Hierarquias de Abstração que fundamenta o ambiente A_Hand.

A seguir comparamos a nossa proposta com todas as outras abordagens descritas nesta dissertação, com respeito a uma série de aspectos. Essas outras abordagens são: os sistemas distribuídos apresentados na seção 3.2 ("Análise de sistemas distribuídos"), i.e., Argus, Chorus e Emerald; e as propostas de programação distribuída orientada a objetos, descritas nas seções precedentes — COOL, Separate Entities e Eiffel//.

Semântica

A nossa maior preocupação neste trabalho foi elaborar um modelo de programação distribuída que respeitasse as características da linguagem Cm e se enquadrasse na filosofia de desenvolvimento do ambiente subjacente. Não era nosso objetivo construir uma nova linguagem de programação "a partir do zero", e nem incluir no Cm toda e qualquer extensão, mecanismo ou facilidade que julgássemos útil ou interessante.

Essa preocupação deslocou, paulatinamente, o foco da nossa atenção: do trabalho de implementação desse modelo para a sua completa especificação. Isso ocorreu na medida em que ficou claro que é um trabalho muito difícil estabelecer esse modelo, respeitando ao mesmo tempo a herança representada pela linguagem Cm e produzindo um resultado final coerente e viável.

Acreditamos que o modelo apresentado, e as conseqüentes extensões sugeridas, trata adequadamente o problema da programação distribuída orientada a objetos. As extensões sintáticas apresentadas têm complexidade e escopo bem limitados, e os conceitos do paradigma de objetos são usados no contexto distribuído com uma semântica consistente. De negativo podemos citar as variáveis de classe e os construtores de objetos.

Quanto às variáveis de classe, conforme destacado na seção 5.2.7, o problema é fundamental e não admite soluções simples — se é que admite alguma solução. Embora essas variáveis sejam um meio muito conveniente de compartilhar informação, isso deveria ser feito sempre através de objetos criados explicitamente para esse fim. Na linguagem Cm, as variáveis de instâncias estão "dentro" dos objetos, mas as variáveis de classe não, pertencem a todos e pertencem a nenhum; em C++ o problema é ainda mais sério, já que pode haver variáveis fora de classes. Variáveis de classe em Smalltalk são mais naturais porque as classes também são objetos, e porque toda interação entre objetos está contida no universo representado pela máquina virtual Smalltalk.

O problema dos construtores de objetos, na nossa opinião, é que eles limitam a versatilidade da linguagem, são difíceis de entender e não oferecem nenhuma vanta-

gem em contrapartida. Objetos devem ser criados de forma explícita, através de um método pré-definido, como ocorre em Smalltalk e Eiffel, de forma desvinculada da declaração do identificador que vai representar o objeto. No caso da versão distribuída do Cm, os construtores de objetos forçaram diretamente a necessidade do construtor de tipos `unattached`, e em menor extensão o construtor de tipos `autonomous`. Sem construtores de objetos, talvez apenas o construtor de tipos `remote` seria sucifiente para declarar objetos distribuídos, como na proposta Separate Entities (o correspondente é a cláusula `separate`).

Transparência

As linguagens e sistemas orientados a objetos são a ferramenta básica para a construção e execução de aplicações distribuídas, e nessas aplicações buscamos transparência de localização, de acesso, de falhas e de concorrência. Cabe aos sistemas e às linguagens implementar todos esses aspectos da transparência, e dela também se beneficiar sempre que possível, pelas vantagens que ela representa nos ambientes distribuídos.

O sistema (e a linguagem) Emerald tem um forte compromisso com a transparência na utilização e localização dos objetos, independentemente de suas propriedades. Há funções para lidar explicitamente com a localização dos objetos quando isso for interessante para a aplicação, mas o uso desses objetos é sempre uniforme. Essas facilidades também serão incorporadas no nosso ambiente, mas acreditamos que essa discussão está além do escopo deste trabalho.

O sistema COOL se beneficia da transparência implementada pelo Chorus no nível do sistema operacional, mas não pudemos avaliar, pelo material bibliográfico disponível, como esse aspecto se reflete nas linguagens de programação suportadas pelo ambiente (i.e., qual a interface de programação oferecida).

A proposta Separate Entities trata muito adequadamente, a nosso ver, a diferença entre objetos envolvidos em atividades seqüenciais e objetos de atividades concorrentes. A definição de *processador* permite separar o conceito de objetos separados, embutido na linguagem, da forma como serão implementados na prática. A nossa proposta tem uma abordagem muito semelhante (embora menos genérica), baseada na coincidência de opiniões sobre essa diferença. Conforme também assinalado em [Mey93], julgamos que a separação entre objetos remotos e não remotos (que ocorre na declaração dos mesmos, e não no uso) não significa perda de transparência, e é essencial para estender a semântica do uso de objetos para o contexto distribuído.

Em outros aspectos, porém, a proposta Separate Entities está mais afinada com a proposta Eiffel//, principalmente na decisão de suportar a forma assíncrona de chamada de métodos. Acreditamos que a forma síncrona, apesar de ser menos geral e menos flexível, é semanticamente mais consistente e transparente.

Tolerância a falhas

Nossa proposta não apresenta nenhuma extensão da linguagem Cm para atender esse aspecto, muito embora ele seja de importância central no estudo de ambientes de programação distribuídos. Não há dúvida de que o modelo representado por Argus é o mais poderoso e o que mais tem apresentado, conforme assinala [Lis88], resultados convincentes nessa importante área. De fato, a incorporação, na lingua-

gem de programação, do conceito de *ação atômica*, com pleno suporte do sistema para o seu uso, oferece uma base sólida para o desenvolvimento sistemático de aplicações tolerantes a falhas. Na falta de mecanismos adequados à implementação de consistência e disponibilidade das informações, a dificuldade para a construção dessas aplicações recai inteiramente sobre o programador.

No sistema COOL, a persistência de objetos é obtida através da persistência de *clusters*. Depois de criados, os objetos têm tempo de vida independente de quem os criou. A transferência de um objeto de um ponto a outro na rede, ou para memória secundária, é transparentemente gerenciada pelo sistema. Não há, entretanto, um mecanismo elaborado como em Argus, para garantir a consistência dos objetos durante a execução de operações com os mesmos.

No extremo oposto a Argus está o sistema Emerald, onde os objetos de um mesmo *site* compartilham o mesmo espaço de endereçamento. Esse procedimento tem efeito positivo no desempenho das aplicações, mas não permite nenhum esquema de segurança contra falhas com respeito a violações no acesso à memória. Portanto, um problema em um objeto pode comprometer irremediavelmente todos os demais objetos do mesmo *site* [Chin91].

Concorrência

Dentre os sistemas e linguagens estudados, há diferenças na forma como tratam o aspecto de concorrência. Posto que há concorrência entre objetos em todos os exemplos estudados, as diferenças estão na adoção (ou não) de concorrência dentro de objetos. Segundo a classificação de [Weg87], vamos analisar os modelos com processos seqüenciais, e depois os modelos com processos concorrentes (não há casos com processos quase-concorrentes).

As propostas Separate Entities e Eiffel// suportam processos seqüenciais, com apenas um fluxo de execução. Em Separate Entities, cada objeto separado, antes de ser usado por outro objeto, deve ser "reservado" por este e, além disso, deve estar ocioso. E em Eiffel// as requisições para um processo são enfileiradas e atendidas uma por vez. Na exposição dessa proposta [Car93], o autor sugere — sem desenvolver a idéia — que o seu modelo não impede a implantação de facilidades de *multithreading*, para permitir uma melhor distribuição da granularidade dos objetos.

Objetos em Argus, COOL e Emerald são concorrentes, pois podem atender e executar, simultaneamente, várias requisições de serviços. Em COOL a concorrência está baseada nas *threads* Chorus, e há um conjunto de primitivas para fazer o controle de concorrência, quer baseado em memória compartilhada (semáforos, *mutexes* e *locks*), quer baseado em algoritmos de *tokens* distribuídos. Em Emerald, a sincronização entre *threads* é obtida através de monitores, que são construções primitivas da linguagem. Em Argus, o controle de concorrência está integrado com o mecanismo de transações atômicas (há contenção de *threads* quando estas tentam operações sobre um mesmo dado compartilhado), além de primitivas para exclusão mútua simples.

A nossa proposta estabelece processos concorrentes, pela possibilidade de execução simultânea de métodos, e objetos ativos contidos dentro de outros objetos. A adoção ou não de concorrência para um determinado objeto depende da

declaração da sua classe. A sincronização entre *threads* é feita através de um único mecanismo, as regiões críticas condicionais estendidas. Esse mecanismo, tal como proposto, é de alto nível de abstração (não há como o programador interferir no escalonamento das atividades) e tem grande flexibilidade. Além disso, estamos trabalhando na definição de axiomas que nos permitam provar formalmente a correção de *software* concorrente e distribuído em Cm.

Desempenho

Conforme [Hor89], tanto as linguagens como os sistemas distribuídos orientados a objetos têm problemas crônicos quanto a desempenho. No caso dos exemplares de linguagens e sistemas apresentados nesta dissertação, o estudo da bibliografia revela uma preocupação recorrente com esse aspecto, principalmente no caso de COOL e Argus. Os números apresentados, contudo, não nos servem para nenhum tipo de comparação, pois refletem medidas relativas a diferentes situações ou procedimentos, obtidas em diferentes ambientes.

Quanto à nossa proposta, ela será implementada sobre plataforma Unix padrão, i.e., sem nenhum componente (*hardware*, meio de comunicação ou sistema operacional) modificado conforme alguma necessidade específica nossa. O fato de abrirmos mão do controle estrito sobre operações críticas do ponto de vista de desempenho (gerenciamento de memória virtual, protocolos de comunicação, escalonamento de processos, etc.) pode significar um espaço de manobras pequeno para que possamos melhorar o desempenho das aplicações no nosso ambiente. Por outro lado, a adoção de ambientes padrão, não específicos, é fundamental dentro da nossa filosofia de desenvolvimento.

De qualquer forma, não é possível adiantar qualquer consideração a respeito. Há ainda um extenso trabalho de implementação a ser feito, no sentido de alterar a linguagem e prover mecanismos básicos de programação distribuída com objetos no nosso ambiente. Somente depois da implantação da primeira versão distribuída da linguagem Cm é que teremos dados concretos para analisar a nossa proposta sobre esse aspecto.

6 Conclusões

Toda arte e toda indagação, assim como toda ação e todo propósito, visam a algum bem; por isto foi dito acertadamente que o bem é aquilo a que todas as coisam visam.

Aristóteles

Foi apresentado o modelo de programação distribuída a ser implementado na linguagem Cm. O objetivo é adaptar a linguagem à construção de aplicações distribuídas, compostas por uma coleção de objetos, com ganhos em modularidade, transparência e portabilidade.

Atualmente, essa área de pesquisa está recebendo grande atenção, embora as bases teóricas desse campo já estejam há muito estabelecidas (por exemplo, as linguagens DP [Han78] e CSP [Hoa78]). Acreditamos que este trabalho de mestrado está sincronizado com esforços acadêmicos atuais, e esperamos dispor de um protótipo da versão distribuída da linguagem (o que inclui alterar o compilador com as mudanças sugeridas e implementar o sistema de suporte à execução) em um prazo razoavelmente curto.

A seguir, apresentamos alguns comentários relativos ao trabalho no seu conjunto. Analisamos também alguns aspectos (macroscópicos) de implementação, para dar uma medida do esforço que será necessário para concluir esse protótipo. Finalmente, destacaremos alguns tópicos de pesquisa e tendências da área, os quais gostaríamos de incluir em nossos esforços futuros.

6.1 Proposta original

A construção de aplicações distribuídas dentro do ambiente A_Hand depende de dois esforços complementares: o sistema OMNI, como mecanismo básico de suporte à programação distribuída; e as linguagens Cm e LegoShell, com construções para a especificação de aplicações distribuídas. As linguagens utilizam os recursos do OMNI através de uma camada de *software* denominada *Run Time System* (também chamado de "sistema de suporte à execução") da linguagem Cm.

Este trabalho de mestrado foi proposto originalmente em [Gon92], e o resultado final, apresentado nessa dissertação, tem diferenças bastante importantes, tanto na

especificação das modificações da linguagem como na implementação destas. Tais diferenças afetaram o desenvolvimento do trabalho, que tornou-se mais abrangente, ao custo de um prazo maior do que o previsto.

A base da proposta original era a definição e implementação do *Run Time System* da linguagem Cm, que serviria como interface entre o sistema OMNI e as linguagens Cm e LegoShell. Esse objetivo não mudou. O trabalho proposto foi baseado, principalmente, na linguagem LegoShell, onde a forma de comunicação entre objetos são conexões envolvendo portas de comunicação. A função principal do *Run Time System* seria mapear as operações suportadas pelas portas de comunicação nas funções de mais baixo nível oferecidas pelo OMNI. Dentro de programas Cm, as portas são objetos da classe *Port* (ou *TypedPort*, se forem portas tipadas), que é pré-definida na linguagem.

O conceito de objetos distribuídos não existia dentro da linguagem Cm. Uma aplicação distribuída, composta por vários objetos executados em diferentes máquinas da rede, deveria ser implementada como uma computação LegoShell. A comunicação entre os objetos se faria utilizando exclusivamente as portas de comunicação, e a ligação entre clientes e servidores se faria pela configuração da computação na LegoShell, ou através do Servidor de Nomes.

Basicamente, a noção de objetos distribuídos deslocou o foco do problema da LegoShell para o Cm, de modo que a construção de aplicações distribuídas completas possa ser feita também com programas Cm. A ligação entre clientes e servidores pode ser direta e estática, através da criação de objetos remotos (construtor de tipos `remote`), ou feita dinamicamente por programa ou pela LegoShell, através da declaração de referências para objetos remotos (construtor de tipos `unattached`).

Não resta dúvida que a programação baseada no modelo de objetos distribuídos em Cm é muito mais flexível e poderosa que a baseada apenas em portas de comunicação, mesmo porque aquela inclui esta (conforme a nossa proposta). De qualquer forma, a comunicação através de portas é ortogonal à comunicação por chamada remota de métodos. Ainda não avaliamos se os dois modelos de comunicação são mesmo necessários: esta conclusão depende de uma análise cuidadosa da melhor adequação dos dois mecanismos em função das necessidades das aplicações.

6.2 Implementação

O término desse trabalho de mestrado conclui a fase de definição da versão distribuída da linguagem Cm. Essa definição deverá ser apresentada formalmente em um documento separado, com a especificação completa e detalhada das alterações que a linguagem Cm deve sofrer (até mesmo no seu nome, se for o caso). Completada a especificação, terá início a implementação do *Run Time System* e as mudanças necessárias no compilador.

A base para a implementação da versão distribuída da linguagem Cm será reconsiderada. O sistema OMNI será usado para a construção do protótipo (da versão distribuída da linguagem Cm), e depois a própria base do OMNI deverá mudar. Atualmente, os esforços nesse sentido buscam a adesão aos padrões OSF/DCE

[OSF90] e OMG/CORBA [OMG92], como forma de facilitar a portabilidade do ambiente e de permitir a livre interoperabilidade com outros ambientes baseados em objetos. Essa proposta é a base do projeto de pesquisa ASAP, a ser financiado pelo Protem/CNPq, que envolve o DCC-UNICAMP (através do Laboratório A_Hand), o LCMI/UFSC, o DEE/UFF e o DEE/UFC.

Nesse projeto, uma camada denominada Objetos Distribuídos (OD) vai implementar, basicamente, funções para a criação e destruição de objetos, comunicação transparente através da ativação remota de métodos e portas de comunicação, e propagação de exceções. Tais funções serão baseadas nos serviços oferecidos pelo DCE. Também estão previstos módulos com funções para programação tolerante a falhas (TF) e programação tempo real (TR). Essas funcionalidades deverão, num passo seguinte, ser incorporadas dentro da linguagem Cm.

Acima da camada de Objetos Distribuídos, há a Linguagem de Especificação de Interfaces (LEI), que servirá de interface geral entre aplicações baseadas em objetos. O conjunto DCE + OD + TR + TF + LEI deve ser implementado como um ORB — para detalhes consulte a seção 3.3.3 ("CORBA — Common Object Request Broker Architecture"). Isso possibilitará a cooperação de objetos de outros ambientes, baseados em outros ORBs. Como módulos usuários desse ORB, serão implementadas as linguagens de programação (Cm e LegoShell no nosso caso).

Nesse novo contexto, o trabalho de implementação deve ser composto dos seguintes passos:

- definição das atribuições da camada de Objetos Distribuídos;
- estudo da incorporação de mecanismos de programação tolerante a falhas e programação tempo real dentro da linguagem Cm;
- integração das funções da camada OD com o padrão OSF/DCE;
- integração das linguagens de programação com o modelo de objetos estabelecido pelo padrão OMG/CORBA.

6.3 Extensões futuras

Depois de implementada a versão distribuída da linguagem Cm, acreditamos que os programadores terão a seu dispor uma excelente ferramenta para o desenvolvimento de aplicações distribuídas com grande eficiência, flexibilidade e robustez. A opção pelos padrões OSF/DCE e OMG/CORBA é importante na medida em que permite a essas aplicações "ganhar o resto do mundo", tornando-se mais poderosas e produtivas. Aí inclui-se também a possibilidade de direcionar melhor nosso esforço de desenvolvimento, pelo aproveitamento de peças de *software* que estejam disponíveis comercialmente e que possam ser absorvidas dentro da nossa necessidade.

A imensa quantidade de trabalho representada pela implementação da especificação de linguagem como está neste trabalho de mestrado inibe, até certo tempo, a pretensão de propor outras extensões. Antes de tudo, é necessário verificar, na prática, a viabilidade e a adequação das mudanças sendo propostas. Entretanto, é possível citar algumas áreas que merecem pesquisa.

Em um aspecto mais teórico, seria interessante estudar mais a fundo a incompatibilidade entre o mecanismo de herança e a programação concorrente e distribuída (problema destacado, entre outros, por [Weg87]). O problema é sério, já que herança é um mecanismo muito poderoso de modelagem e reutilização de código. As soluções propostas, que incluem até mesmo o abandono puro e simples de classes e herança (como por exemplo Emerald e linguagens baseadas em protótipos), não têm obtido consenso.

Um aspecto que certamente merece atenção é a programação tolerante a falhas utilizando objetos. Conforme [Lis88], apenas o sistema Argus tem apresentado uma proposta com resultados consistentes nesse aspecto. Essa área contempla vários tópicos de pesquisa, como migração de objetos, ações atômicas, e integração com bancos de dados orientados a objetos.

Finalmente, há a questão do desenvolvimento de ferramentas para auxílio no desenvolvimento de aplicações distribuídas em larga escala. Particularmente importante é o aspecto de depuração de aplicações compostas de objetos distribuídos, o que deverá exigir um suporte do sistema de execução muito elaborado, flexível e preciso. Nesse sentido, está em estudo o uso da LegoShell, que foi projetada como uma linguagem de especificação de programas, como um ambiente integrado de configuração, monitoração, *profiling* e depuração de aplicações distribuídas. Isso significa incorporar, na LegoShell, o conceito de objetos distribuídos e comunicação através de chamada remota de métodos, e fazer a integração da linguagem com o sistema de suporte do Cm Distribuído.

6.4 Considerações finais

O término deste trabalho deixa muitas certezas, e também muitos pontos de interrogação. De certa forma, é difícil, nesse momento, analisar as conseqüências e o desenvolvimento posterior do trabalho de implementação baseado nessa definição.

Este trabalho chega ao fim com dois aspectos negativos: o tempo que ele consumiu e a falta de implementação. Quanto ao tempo necessário para terminá-lo, foi muito longo. Para isso concorreram vários fatores, principalmente a complexidade que o trabalho tomou, e que fugiu à nossa previsão. E a falta de implementação, de um protótipo que seja, deve-se também à complexidade do resultado final do trabalho; nesse sentido, a atidude mais responsável é completar, no maior grau de detalhe possível, a especificação da linguagem e do sistema de suporte. O autor, todavia, não abre mão da responsabilidade por esse atraso e por essa falta.

De positivo, há aspectos que podem ser considerados algo subjetivos, mas que para o autor são bastante concretos. Esta dissertação está fundamentada em um trabalho sólido, paciente, articulado de pesquisa e entendimento do paradigma de objetos, o que abre uma perspectiva otimista para o futuro próximo de seu desenvolvimento.

E outros aspectos, como companheirismo, trabalho, reconhecimento, a possibilidade de pensar e realizar dentro de uma comunidade de pessoas notáveis por seu conhecimento e capacidade, são coisas que o autor guarda para si, porque são a experiência que se leva. Para o leitor, se oferece este trabalho, que tem um pouco disso tudo.

A Referências Bibliográficas

Este apêndice contém as referências bibliográficas utilizadas nesta dissertação. Material produzido dentro do âmbito do Projeto A_Hand, seja relatório técnico, artigo publicado, proposta de tese ou tese de mestrado, está agrupado na primeira seção; as demais referências estão na seção seguinte.

A.1 Publicações do Projeto A_HAND

1. **[Ari91]** **Editor Topológico para a Linguagem de Especificação de Computações LegoShell**
Hernán P. Arias
Tese de Mestrado, DCC-IMECC-UNICAMP, janeiro de 1991.

2. **[Dru87a]** **A_HAND: Ambiente de Desenvolvimento de Software Baseado em Hierarquias de Abstração em Níveis Diferenciados**
Rogério Drummond e Hans K. E. Liesenberg
IV Encontro de Trabalhos do Projeto ETHOS, Petrópolis, RJ, abril 1987.
Revisto e reimpresso como relatório técnico, Projeto A_HAND, outubro 1987.

3. **[Dru87b]** **Requisitos para um ambiente de desenvolvimento de PROGRAMAS**
Rogério Drummond e Hans K. E. Liesenberg
I Encontro IBM de Ciência e Tecnologia em Informática, Rio de Janeiro, RJ, novembro 1987.

4. **[Dru88]** **Manual de referência da Linguagem Cm (versão preliminar)**
Rogério Drummond e Fábio Q. B. da Silva
Projeto A_HAND, DCC-IMECC-UNICAMP, março 1988.

5. **[Dru89]** **LegoShell Linguagem de Computações**
Rogério Drummond
III Simpósio Brasileiro de Engenharia de Software, Recife (PE), 1989.

6. **[Dru92]** **OMNI – Sistema de suporte a aplicações distribuídas**
Rogério Drummond e Cassius Di Cianni
Anais do VI Simpósio Brasileiro de Engenharia de Software, pp 309–324, novembro 1992.

7. **[Dru93]** **Aspectos da Implementação de Objetos Distribuídos**
Rogério Drummond, Celso Gonçalves Jr e Alexandre P. Teles
Anais do XI Simpósio Brasileiro de Redes de Computadores, pp 139–152, maio 1993.

8. **[Dru94]** **Objetos Distribuídos em Cm**
Rogério Drummond e Celso Gonçalves Jr.
Anais do XII Simpósio Brasileiro de Redes de Computadores, pp. 188–201, maio 1994.

9. **[Fur91]** **Um compilador para uma linguagem de programação orientada a objetos**
Carlos A. Furuti
Tese de mestrado, DCC-IMECC-UNICAMP, julho 1991.

10. **[Gon92]** **Suporte para Programação em Sistemas Distribuídos**
Celso Gonçalves Jr. e Rogério Drummond
Proposta de Tese de Mestrado, DCC-IMECC-Unicamp, maio 1992.

11. **[Gon94]** **Controle de Concorrência em Objetos**
Celso Gonçalves Jr., Bill Coutinho e Rogério Drummond
Relatório Técnico, Laboratório A_Hand, DCC-IMECC-UNICAMP, abril 1994.
Trabalho submetido para publicação.

12. **[Har90]** **Utilização de um Banco de Dados Orientado a Objetos em um Ambiente de Desenvolvimento de Software**
Carmen S. Hara
Tese de mestrado, DCC-IMECC-UNICAMP, setembro 1990.

13. **[Sil88]** **Programação em Cm**
Fábio Q. B. da Silva, Hans K. E. Liesenberg e Rogério Drummond
Anais do XV Semish, pp 101–102, Rio de Janeiro, RJ, julho 1988.

14. **[Sou92]** **Marcelo H. de Souza**
Relatório Parcial de Projeto de Bolsa de Iniciação Tecnológica e Industrial:
Sistema Linear.
Relatório Técnico, Laboratório A_Hand, DCC-IMECC-UNICAMP, agosto 1992.

15. **[Tel93]** **A linguagem de programação Cm (versão 2.0x)**
Alexandre P. Teles
Tese de mestrado, DCC-IMECC-UNICAMP, outubro 1993.

16. **[Vic89]** **Mecanismo de Gerenciamento de Versões e Configurações do A_HAND**
Eliane Z. Victorelli, Geovane C. Magalhães e Rogério Drummond
Revista Brasileira de Computação, 5 (2), pp. 3–9, dezembro de 1989.

17. **[Vic90]** **Mecanismo de Gerenciamento de Versões do A_HAND**
Eliane Z. Victorelli
Tese de Mestrado, DCC-IMECC-UNICAMP, dezembro de 1990.

A.2 Publicações diversas

18. **[And83]** **Concepts and Notations for Concurrent Programming**
Gregory R. Andrews e Fred B. Schneider
ACM Computing Surveys, 15 (1), pp. 3–43, março 1983.

19. **[Arm89]** **Revolution 89 or "Distributing UNIX Brings it Back to its Original Virtues"**
François Armand, Michel Gien, Frédéric Herrmann e Marc Rozier
Chorus Systèmes, CS/TR–89–36.1, agosto 1989.

20. **[Bal89]** **Programming Languages for Distributing Computing Systems**
Henry E. Bal, Jennifer G. Steiner e Andrew S. Tanenbaum
ACM Computing Surveys, 21 (3), setembro 1989.

21. **[Bla86]** **Object Structure in the Emerald System**
Andrew Black, Norman Hutchinson, Eric Jul, Henry Levy
ACM SIGPLAN OOPSLA'86 Proceedings, pp. 78–86, setembro 1986.

22. **[Boo94]** **Object Oriented Design with Applications, 2nd edition.**
Grady Booch
The Benjamin/Cummings, Redwood City (CA), 1994.

23. **[Car85]** **On Understanding Types, Data Abstraction, and Polymorphism**
Luca Cardelli e Peter Wegner
ACM Computing Surveys, 17 (4), pp. 471–522, dezembro 1985.

24. **[Car93]** **Toward a Method of Object-Oriented Concurrent Programming**
Denis Caromel
Communications of the ACM, 36 (9), pp 90–102, setembro 1993.

25. **[Che88]** **The V Distributed System**
David Cheriton
Communications of the ACM, 31 (3), pp. 314–333, março 1988.

26. **[Chi91]** **Distributed Object-Based Programming Systems**
Roger S. Chin e Samuel T. Chanson
ACM Computing Surveys, 23 (1), março 1991.

27. **[DeR76]** **Programming-in-the-Small Versus Programming-in-the-Large**
Frank DeRemer e Hans H. Kron
IEEE Transactions on Software Engineering, SE–2, nº 2, junho 1976.

28. **[DoD83]** **Reference manual for the Ada programming language**
ANSI/MIL-STD-1815A
Department of Defense, Washington (DC), janeiro 1983.

29. **[Gol83]** **Smalltalk-80: The Language and Its Implementation**
Adele Goldberg e David Robson
Addison-Wesley, Reading (MA), 1983.

30. **[Gut89]** **Are the Emperor's New Clothes Object Oriented?**
Scott Guthery
Dr. Dobb's Journal, pp. 80–86, dezembro 1989.

31. **[Hab90]** **COOL: Kernel Support for Object-Oriented Environments**
Sabine Habert, Laurence Mosseri e Vadim Abrossimov
ACM Sigplan Notices ECOOP/OOPSLA'90 Proceedings, pp 269–276, outubro 1990.

32. **[Han78]** **Distributed Processes, A Concurrent Programming Concept**
Per Brinch Hansen
Communications of the ACM, 21 (11), pp 934–941, novembro 1978.

33. **[Her89]** **Multi-threaded UNIX Processes in CHORUS/MIX**
Frédéric Herrmann, François Armand e Marc Rozier
Chorus systèmes, CS/TR–89–37.2, junho 1989.

34. **[Hoa72]** **Towards a Theory of Parallel Programming**
Charles A. R. Hoare
Operating Systems Techniques
Academic Press, New York (NY), 1972.

35. **[Hoa74]** **Monitors: An Operating System Structuring Concept**
Charles A. R. Hoare
Communications of the ACM, 17 (10), pp 549–557, outubro 1974.

36. **[Hoa78]** **Communicating Sequential Processes**
Charles A. R. Hoare
Communications of the ACM, 21 (8), pp 666–677, agosto 1978.

37. **[Hor89]** **Is Object Orientation a Good Thing for Distributed Systems?**
Chris Horn
Lecture Notes on Computer Science nº 433, pp. 60–74, Springer-Verlag, Berlin

38. **[Ker78]** **The C Programming language**
Brian W. Kernighan e Dennis M. Ritchie
Prentice–Hall, Englewood Cliffs (NJ), 1978.

39. **[Ker84]** **The UNIX Programming Environment**
Brian W. Kernighan e Robert Pike
Prentice-Hall, Englewood Cliffs (NJ), 1984.

40. **[Koe90]** **Exception Handling for C++**
Andrew Koenig e Bjarne Stroustrup
Journal of Object Oriented Programming, pp. 16–33, julho/agosto 1990.

41. **[Lea93]** **COOL: System Support for Distributed Programming**
Rodger Lea, Christian Jacquemot e Eric Pillevesse
Communications of the ACM, 36 (9), pp 37–46, setembro 1993.

42. **[Lis77]** **Abstraction Mechanism in CLU**
Barbara Liskov, A. Snyder, R. Atkinson, C. Schaffert
Communications of the ACM, 20 (8), pp 564–576, agosto 1977.

43. **[Lis82]** **Guardians and Actions: Linguistic Support for Robust, Distributed Programs**
Barbara Liskov, Robert Scheiffler
Proceedings of the 9th Symposium on Principles of Programming Languages, pp. 7–19, janeiro 1982.

44. **[Lis87]** **Implementation of Argus**
Barbara Liskov, Dorothy Curtis, Paul Johnson, Robert Scheifler
Proceedings of the 11th Symposium on Operating Systems Principles, pp. 111–122, novembro 1987.

45. **[Lis88]** **Distributed Programming in Argus**
Barbara Liskov
Communications of the ACM, 31 (3), pp. 300–312, março 1988.

46. **[Mad88]** **What object-oriented programming may be — and what it does not have to be**
Ole L. Madsen e Birger Møller-Pedersen
Lecture Notes on Computer Science nº 322, pp. 1–20, Springer-Verlag, Berlin

47. **[Mey86]** **Genericity versus Inheritance**
Bertrand Meyer
ACM SIGPLAN OOPSLA'86 Proceedings, pp. 391–405, setembro 1986.

48. **[Mey87]** **Eiffel: Programming for Reusability and Extendibility**
Bertrand Meyer
Sigplan Notices, 22 (2), fevereiro 1987.

49. **[Mey93]** **Systematic Concurrent Object-Oriented Programming**
Bertrand Meyer
Communications of the ACM, 36 (9), pp 56–80, setembro 1993.

50. **[Mul90]** **Amoeba A Distributed Operating System for the 1990s**
Sape J. Mullender, Guido van Rossum, Andrew S. Tanenbaum, Robbert van Renesse, Hans van Staveren
Computer, pp. 44–53, maio 1990.

51. **[Nau63]** **Report on the Algorithmic Language Algol60**
Peter Naur, Editor
Communications of the ACM, 6 (1), pp. 1–17, janeiro 1963.

52. **[Nyg78]** **The Development of the SIMULA Languages**
Kristen Nygaard e Ole-Johan Dahl
History of Programming Languages Conference
SIGPLAN Notices, 13 (8), agosto 1978.

53. **[OMG91]** **The Common Object Request Broker: Architecture and Specification**
OMG Document Number 91.12.1.
John Wiley & Sons Inc., New York (NY), 1991.

54. **[OMG92]** **Object Management Architecture Guide**
OMG TC Document 92.11.1.
John Wiley & Sons Inc., New York (NY), 1992.

55. **[OSF90]** **OSF Distributed Computing Environment Rationale**
Open Software Foundation, maio de 1990.

56. **[Par72]** **On the Criteria To Be Used in Decomposing Systems into Modules**
Dave L. Parnas
Communications of the ACM, 15 (12), pp. 1053–1058, dezembro 1972.

57. **[Raj91]** **Emerald: A General-Purpose Programming Language**
Rajendra K. Raj, Ewan Tempero, Henry M. Levy, Andrew P. Black, Norman C. Hutchinson e Eric Jul
Software — Practice & Experience, 21 (1), pp 91–118, janeiro 1991.

58. **[Sha88]** **Special Section on Operating Systems**
Alan Shaw (guest editor)
Communications of the ACM, 31 (3), pp. 255–257, março 1988.

59. **[Sny86]** **Encapsulation and Inheritance in Object-Oriented Programming Languages**
Alan Snyder
ACM SIGPLAN Notices OOPSLA'86 Proceedings, pp. 38–45, 1986.

60. **[Str86]** **The C++ Programming Language**
Bjarne Stroustrup
Addison-Wesley, Reading (MA), 1986.

61. **[Str91]** **The C++ Programming Language, 2nd Ed.**
Bjarne Stroustrup
Addison-Wesley, Reading (MA), 1991.

62. **[Sun88]** **Network Programming Guide**
Sun Microsystems, Mountain View (CA), maio 1988.

63. **[Sun93]** **SunOS 5.2 System Services**
Sun Microsystems, Mountain View (CA), maio 1993.

64. **[Tak90]** **Programação Orientada a Objetos**
Tadao Takahashi, Hans K. E. Liesenberg e Daniel T. Xavier
VII Escola de Computação, IME-USP, São Paulo (SP), julho 1990.

65. **[Tan81]** **Computer Networks**
Andrew S. Tanenbaum
Prentice-Hall, Englewood Cliffs (NJ), 1981.

66. **[Tan85]** **Distributed Operating Systems**
Andrew S. Tanenbaum, Robbert Van Renesse
ACM Computing Surveys, 17 (4), pp 419–470, dezembro 1985.

67. **[Var94]** **Small Kernels Hit It Big**
Peter D. Varhol
BYTE, 19 (1), pp. 119–128, janeiro 1994.

68. **[Weg87]** **Dimensions of Object-Based Language Design**
Peter Wegner
ACM SIGPLAN Notices OOPSLA'87 Proceedings, pp. 168–182, 1987.

69. **[Weg90]** **Concepts and Paradigms of Object-Oriented Programming**
Peter Wegner
OOPS Messenger, 1 (1), pp 7–87, agosto 1990.

70. **[Wir85]** **Programming in Modula-2, 3rd editon**
Niklaus Wirth
Springer-Verlag, New York(NY), 1985.

71. **[You87]** **The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System**
Michael Young, Avadis Tevanian, Richard Rashid, David Golub, Jeffrey Eppinger, Jonathan Chew, William Bolosky, David Black, Robert Baron
Carnegie-Mellon University, Technical Report CMU–CS–87–155, agosto 1987.

B Convenções

As convenções tipográficas e simbólicas utilizadas nesta dissertação estão apresentadas neste apêndice.

B.1 Convenções tipográficas

Código fonte é apresentado em parágrafos com margem maior, e com fonte Courier 9 pontos.

```
x = y + z++;
obj.f(x);
while (a && (x < y))
   obj.g(x++);
```

Os identificadores de linguagens de programação — o que inclui nome de variáveis, tipos, funções, arquivos e outros —, quando dentro do texto, são destacados em formato itálico. Exemplo: no trecho de código mostrado acima, temos as variáveis *x* e *z*, e a chamada *obj.f(x)*.

Já palavras reservadas das linguagens, assim como símbolos terminais tomados separadamente, estão em fonte Courier, do mesmo tamanho do texto. Como exemplo, dizemos que a chamada *obj.g(x++)* está dentro de um comando `while`, e que a operação de conjunção lógica se faz pelo operador `'&&'`.

Construções sintáticas são representadas por produções na notação BNF, com tipos e formatos particulares. Exemplos dessas construções estão no apêndice C ("Linguagem"). Uma produção BNF é representada por um símbolo não-terminal, seguido de uma seta ('->') que indica "é substituído por", e a seguir uma seqüência de símbolos, terminais e não-terminais. Os símbolos terminais estão em fonte Times, em negrito; os símbolos não-terminais estão em fonte Helvetica, e símbolos reservados da notação BNF estão em fonte Helvetica, em negrito. Exemplo:

```
while_stmt    ->   while ( expr ) stmt
```

Produções que definem símbolos que podem ser substituídos por mais de um seqüência de itens empregam 'I' para indicar essas alternativas. Por exemplo,

```
stmt          ->   if_stmt
                |  while_stmt
                |  do_stmt
```

Se as alternativas de substituição de um símbolo são seqüências de um item apenas (tipicamente um símbolo terminal), pode-se usar uma notação abreviada, como mostrado abaixo

```
stg_class     ->   one of    auto static extern
```

Símbolos que são opcionais têm o subscrito 'opt'; esses símbolos geralmente possuem uma alternativa de substituição vazia, que é representada pelo símbolo especial 'ε'. Exemplificando:

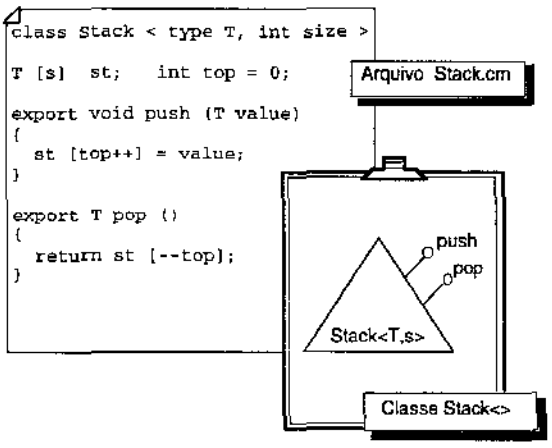
```
if_stmt       ->   if ( expr ) stmt else_part_opt

else_part     ->   else stmt
                |  ε
```

B.2 Convenções de ilustrações

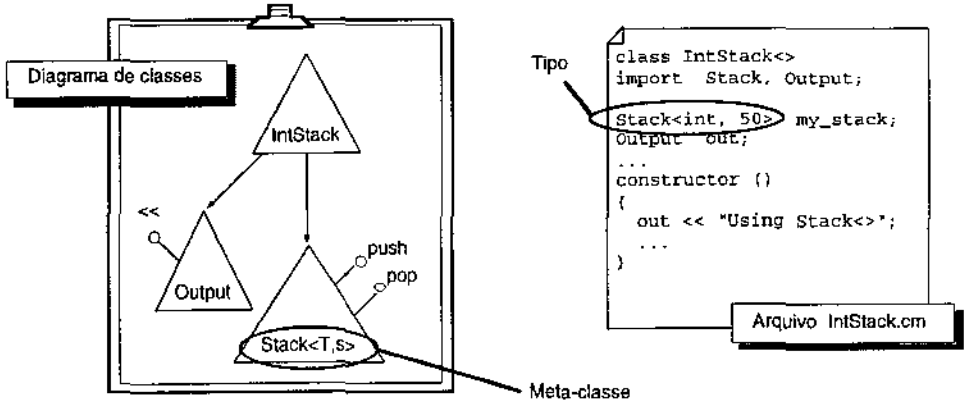
A dissertação faz uso de muitos desenhos para representar objetos, diagramas, etc. Classes são representadas de duas formas: por texto em fonte Courier colocado em uma "folha de papel", rotulada pelo nome do arquivo onde está guardada a classe; e por um triângulo, rotulado pelo nome da classe. Nesse triângulo são mostrados os símbolos exportados da classe, através de hastes colocadas nos lados do triângulo.

FIGURA B-1 Desenho representando uma classe



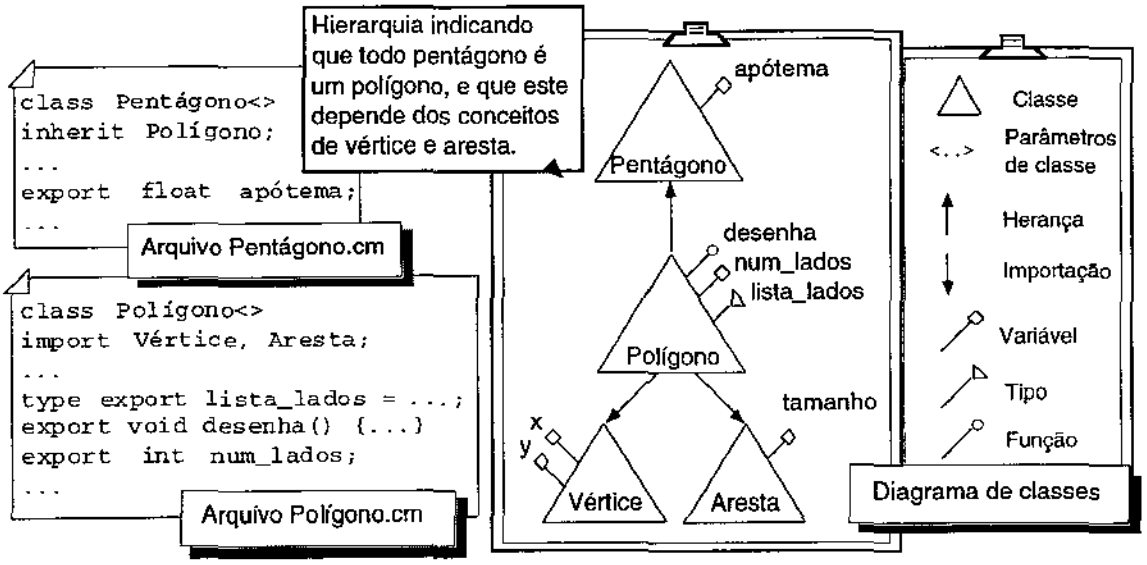
As classes representadas por triângulos podem ser combinadas em diagramas de classe, onde pode-se especificar hierarquias baseadas em importações e herança. A relação de importação é representada por uma seta apontando para baixo, e a relação de herança é representada por uma seta apontando para cima.

FIGURA B-2 Exemplo simples de diagrama de classes



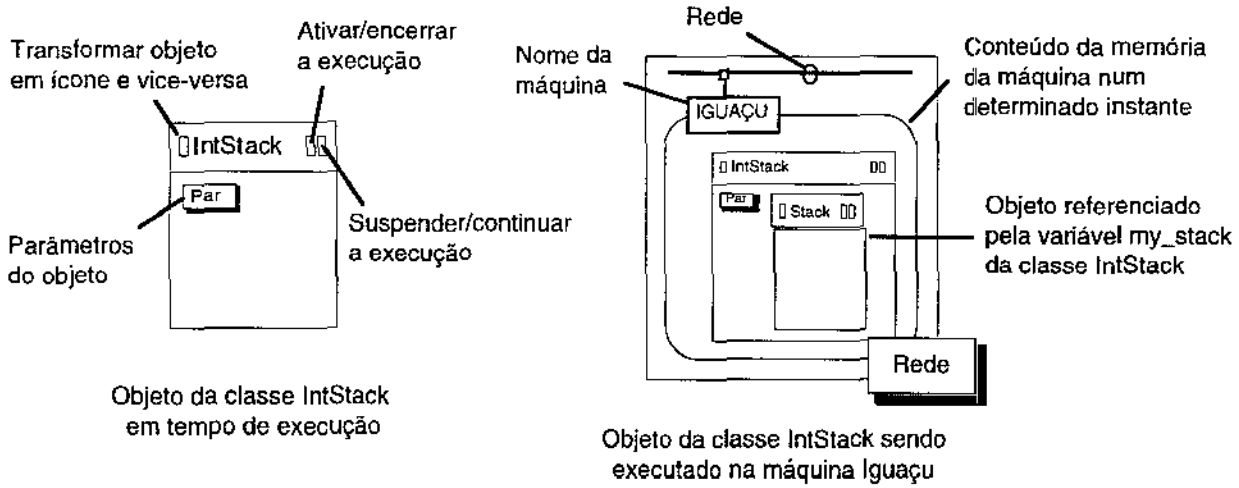
Os símbolos exportados pelas classes que aparecem no diagrama de classes são diferenciados pelo formato das hastes acopladas ao triângulo, de acordo com sua categoria: quadrados para variáveis, triângulos para tipos e círculos para métodos.

FIGURA B-3 Desenho representando uma hierarquia de classes



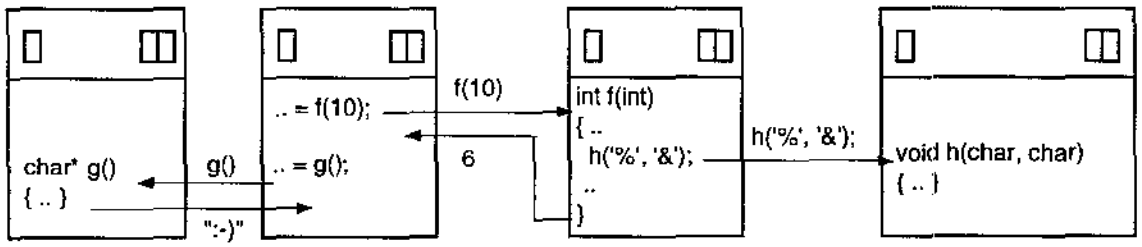
Objetos podem ser representados em tempo de execução, relacionados a outros objetos ou a seu *site* de execução. Essa notação é útil para compreender como se faz a interação entre objetos dinamicamente.

FIGURA B-4 Representação de objetos em tempo de execução



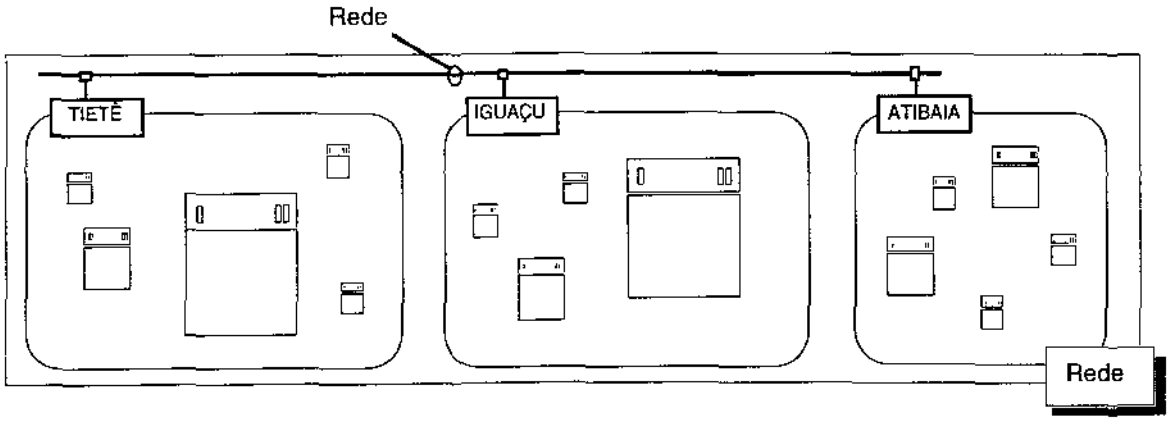
As mensagens enviadas entre os objetos, para a execução de métodos, são representadas por setas rotuladas pelo nome do método e a lista de parâmetros, que é opcional; o resultado da execução do método também é simbolizado por uma seta, rotulada pelo valor desse resultado que é devolvido ao objeto cliente.

FIGURA B-5 Envio de mensagens para execução e resultado de métodos



Um ambiente típico de *hardware* distribuído é representado por uma coleção de máquinas, conectadas a um meio de transmissão disposto em uma topologia de barramento.

FIGURA B-6 Exemplo de rede de máquinas executando objetos



Rede composta da máquinas Tietê, Iguaçu e Atibaia

C Alterações na Linguagem

Neste apêndice são apresentadas formalmente as modificações e extensões propostas para a implementação da versão distribuída da linguagem Cm. Esse material está agrupado em três categorias:

- alterações na sintaxe da linguagem;
- definição de classes pré-definidas;
- definição de funções pré-definidas.

Alterações da sintaxe da linguagem estão expressas na notação BNF, e as classes e funções pré-definidas estão escritas em Cm.

C.1 Alterações na sintaxe de Cm

São basicamente três alterações: os construtores de tipos que são usados para declarar objetos não comuns; a modificação dos construtores de objetos remotos para declaração de nome simbólico e local de execução desses objetos; e a sintaxe das regiões críticas condicionais estendidas.

Os construtores de tipos a que nos referimos foram expostos na seção 5.2 ("Proposta de Objetos Distribuídos em Cm"). Tais construtores aplicam-se apenas a tipos classe, pois as unidades de distribuição das aplicações são os objetos remotos. O resultado da aplicação de algum desses construtores sobre uma classe é um tipo que pode ser usado para declarar parâmetros, elementos de vetores, campos de estruturas, etc. Exemplificando: utilizando uma classe *C*, as declarações abaixo são ou não válidas dependendo da aplicação do construtor de tipos `remote`.

```
C remote obj_c;                      // 1. correto
int remote x;                        // 2. errado!
C<10> remote * ptr_obj_c;            // 3. correto
C * remote rem_ptr_obj_c;            // 4. errado!
C<20> remote [200] * ptr_arr_obj_c;  // 5. correto
```

A declaração 2 está errada porque não faz sentido declarar um número inteiro como um objeto remoto. A declaração 3 está correta já que a variável *ptr_obj_c* é local, e o seu conteúdo é que referencia um objeto remoto. A declaração 4 está incorreta porque a variável *rem_ptr_obj_c* não pode ser remota. Portanto, esses construtores devem aparecer apenas imediatamente após um tipo classe. Na gramática da linguagem Cm [Teles93], a produção

```
type_name0  ->  type_id
            |   std_ty_spec
            |   class_inst_name
```

será modificada para aceitar os construtores de tipos desejados. A modificação consiste em colocar, depois do símbolo *class_inst_name* (3ª alternativa), o símbolo *class_cat*, definido abaixo:

```
class_cat   ->  one of   thread remote unattached autonomous
```

A outra alteração diz respeito ao formato dos construtores de objetos. Esse formato deve ser estendido para permitir a associação de um nome simbólico, com atributos de unicidade e escopo, e de um local de execução específico para o objeto remoto sendo declarado. A produção da gramática do Cm usada para analisar um construtor é a seguinte

```
class_init_opt ->  @ ( actual_par_list_opt )
               |   ε
```

Essa produção é opcional porque a declaração de um objeto pode ser implícita, se for usado o construtor *default*. Ela cobre os seguintes casos, exemplificando com uma classe C:

```
C<15> obj1_c;
C obj2_c @('x');
C arr_obj[5] = { @('!'), @('@'), @('#'), @('$'), @('%') };
C * ptr_obj_c;
...
ptr_obj_c = new ( C<20> @('w') );
```

A modificação do formato do construtor é bem isolada, já que consiste em acrescentar, no construtor da declaração do objeto remoto, o nome simbólico, atributos de unicidade e escopo, e o local de execução, sempre que esses itens existirem. Essa informação a ser acrescentada é representada pela inclusão do símbolo *rem_obj_spec*, definido abaixo:

```
rem_obj_spec ->  as obj_name site_spec_opt scope_spec_opt uniq_spec_opt
             |   site_spec
             |   ε

site_spec    ->  at site_name

scope_spec   ->  one of   local global

uniq_spec    ->  one of   unique not_unique
```

Os símbolos *obj_name* e *site_name*, nome simbólico de um objeto remoto e nome de um *site* para execução, respectivamente, podem ser substituídos por literais ou constantes tipo *string*, como pode-se ver nos exemplos mostrados na seção 5.2.

O símbolo *class_init_opt* é modificado da seguinte forma:

```
class_init_opt ->  @ ( actual_par_list_opt ) rem_obj_spec_opt
                |  rem_obj_spec_opt
```

Uma região crítica condicional estendida é um comando, precedido pela guarda da CCR. Na gramática da linguagem Cm, a produção que descreve um comando é a seguinte:

```
stmt        ->  e_stmt
            |   compound_stms
            |   do_stmt
            |   break_stmt
            |   continue_stmt
            |   return_stms
            |   goto_stmt
            |   raise_stmt
            |   null_stmt
            |   while_stmt
            |   for_stmt
            |   ifelse_stmt
            |   if_stmt
            |   switch_stmt
            |   label_stmt
            |   decl_stmt
```

Como novo comando, o símbolo *ccr_stmt* deve ser incluído entre as alternativas para substituir o símbolo *stmt*, e a sua definição é a seguinte:

```
ccr_stmt    ->  with expr_reg sync_cond do stmt

expr_reg    ->  expr_reg && expr_reg
            |   expr_reg = expr
            |   ( expr_reg )
            |   all
            |   ε
sync_cond   ->  when expr
            |   ε
```

O símbolo *expr* representa uma expressão válida em Cm. Na primeira produção, a expressão após o `switch` é opcional para as CCRs que devem executar de forma atômica. O símbolo *sync_cond* é opcional para o caso das CCRs sem condição de sincronização. Para maiores detalhes consulte-se a seção 5.6.3 ("Controle de Concorrência").

C.2 Classes pré-definidas

O uso de portas de comunicação está baseado nas classes *Port* e *TypedPort*, que são pré-definidas na linguagem, i.e., seus métodos são implementados pelo sistema de suporte à execução. Os objetos que usam portas de comunicação devem importar a classe *Port* (ou a classe *TypedPort*, conforme o caso), e declarar essas portas con-

forme os exemplos mostrados na seção 5.4 ("Portas de Comunicação"). A declaração da classe *Port*, apenas com os símbolos exportados (dos métodos mostram-se apenas os *prototypes*), é a seguinte:

```
class Port<>


type export OMNIId = // representação da OMNIid

type export Uniqueness = enum { UNIQUE, NOT_UNIQUE };

type export Scope = enum { LOCAL, GLOBAL };

type export PortCat =
   enum { INPUT, OUTPUT, INPUT_OUTPUT };

type export PortSem =
   enum { STREAM, BLOCK, BLOCK_W_SEP, ATOMIC, TO_INHERIT };

type export PortSetUp =
   enum { AUTO, BY_REQUEST };

type export PortError =
   enum {
      CANNOT_CREATE_PORT,
      SYMBOLIC_NAME_CONFLICT,
      PORT_NOT_PREPARED,
      PORT_NOT_CONNECTED,
      PORT_NOT_DISCONNECTED,
      PORT_NOT_OPENED,
      PORT_NOT_CLOSED,
      SEMANTICS_MISMATCH,
      BAD_PARAMETERS,
      INVALID_CONNECTION,
      INPUT_QUEUE_OVERFLOW,
      OUTPUT_QUEUE_OVERFLOW,
      LOST_SEPARATOR,
      BLOCK_OVERFLOW,
      NO_ERROR
   };

type export PortStatus =
   enum {   CREATED, PREPARED, OPENED, CLOSED,
            CONNECTED, DISCONNECTED, WRONG };

export constructor (
   PortCat cat       = INPUT_OUTPUT;
   char* symb_name   = NIL;
   PortSem sem       = STREAM;
   PortSetUp setup   = AUTO )

export constructor (
   PortCat cat       = INPUT_OUTPUT;
   char* symb_name   = NIL;
   PortSem sem       = BLOCK;
```

```
    int bsize         = 1024;
    PortSetUp setup   = AUTO )

export constructor (
    PortCat cat       = INPUT_OUTPUT;
    char* symb_name   = NIL;
    PortSem sem       = BLOCK_W_SEP;
    char chsep        = '\0';
    PortSetUp setup   = AUTO )

export int open();
export int close();

export int connect();
export int disconnected();

export int read(char* buf);
export int read(char* buf, int n);
export int read_blocking(char* buf);
export int read_blocking(char* buf, int n);
export int operator>>(char* buf);

export int write(char* buf);
export int write(char* buf, int n);
export int write_blocking(char* buf);
export int write_blocking(char* buf, int n);
export int operator<<(char* buf);

export int input_count();
export int output_count();

export int is_created();
export int is_prepared();
export int is_opened();
export int is_closed();

export int is_EOS();

export PortError get_error();
export void clear_error();

export PortStatus get_status();

export PortError register_name(
    char* name;
    Uniqueness u;
    Scope s );

export PortError cancel_name(char* name);

/* Fim da classe Port */
```

A classe *TypedPort* é uma especialização da classe *Port*, já que oferece as mesmas funções, modificadas para operar sobre valores do tipo passado como parâmetro da

classe. Por exemplo: o método *write()* da classe *Port* usa como primeiro parâmetro um *buffer* de caracteres, que são os dados a serem transmitidos; na caso de portas tipadas, esse parâmetro deve ser um *buffer* de valores do tipo da porta. Omitimos a redefinição desses métodos pois nenhuma informação relevante seria acrescentada.

```
class TypedPort<type T>
inherit Port;

type export TypedPortError =
   enum {
      TYPE_MISMATCH,
      SERIALIZE_ERROR,
      DESERIALIZE_ERROR
   };

// redefinição de métodos por causa do tipo T
...
/* Fim da classe TypedPort */
```

C.3 Funções pré-definidas

As funções pré-definidas da linguagem implementam operações básicas relativas a portas de comunicação e a objetos remotos. No caso de portas de comunicação, são as funções *connect()* e *disconnect()*, usadas para fazer a conexão e desconexão de portas, respectivamente. Para usar essas funções não é necessário importar ou herdar nenhuma classe.

```
PortError connect(Port, Port);
PortError disconnect(Port);
```

As funções pré-definidas de operações com objetos remotos são: *attach()*, para associar um identificador declarado como `unattached` com uma referência para um objeto remoto já existente; *detach()*, para desfazer essa associação; e *kill()*, para destruir explicitamente um objeto remoto.

```
void attach(/* tipo de objeto unattached */, char *);
void attach(/* tipo de objeto unattached */, Port::OMNIId);
void attach(/* tipo de objeto unattached */,
        /* tipo de objeto remoto* /);
void detach();
void kill(/* tipo de objeto remoto */);
void kill(); // sem parâmetros indica "suicídio"
```

D Vocabulário

Nesta dissertação são usadas muitas palavras não encontradas em dicionários de português. São termos próprios da área de computação, amplamente utilizados pela nossa comunidade e formados, na sua maioria, por uma adaptação de termos em inglês, que é a língua universal nesse campo. Em alguns casos, o termo em inglês, como utilizado, também não consta dos dicionários em inglês, já que tratam-se de acepções tão novas quanto a ciência da computação. Neste apêndice vamos falar sobre essas adaptações, mais particularmente dos termos mais importantes, observando a sua origem e de que forma poderíamos justificar sua utilização em português.

Este apêndice reúne material de caráter meramente expositivo, sem nenhuma pretensão de analisar detalhadamente o assunto. O autor limita-se a anotar o uso desses termos, mostrar suas (prováveis) origens e enquadrá-los nas acepções pretendidas.

Durante a nossa argumentação, pode ser que uma palavra esteja presente em uma sentença ora no seu sentido mais comum, ora no sentido em que a empregamos na área. Para que não haja confusão, as palavras em itálico representam um termo de computação, e a mesma palavra em estilo comum representa seu sentido original.

O termo mais importante é *objeto*. No capítulo 2 temos uma definição formal de objeto, e talvez possamos incorporar essa nova acepção no nosso vocabulário. O verbete sobre objeto que consideramos [Aur86] é o seguinte:

> **objeto**. [Do lat. *objectu*, part. de *objicere*, 'por, lançar diante', 'expor'.] *S. m.* **1.** Tudo que é apreendido pelo conhecimento, que não é o sujeito do conhecimento. **2.** Tudo que é manipulável e/ou manufaturável. **3.** Tudo que é perceptível por qualquer dos sentidos. **4.** Coisa, peça, artigo de compra e venda: *objeto barato*. **5.** Matéria, assunto: *o objeto de uma ciência, de um estudo*. **6.** Agente; motivo, causa: *objeto de discórdia*. **7.** O ponto de convergência de uma atividade; mira, desígnio: *A filosofia hindu era o objeto de suas meditações*. **8.** Mira, fim, propósito, intento; objetivo: *Tem como objeto formar-se*. **9.** *Filos.* O que é conhecido, pensado ou representado, em oposição ao ato de conhecer, pensar ou repre-

> sentar. **10.** *Filos.* O que se apresenta à percepção com um caráter fixo e estável. [Cf. (nesta acepç.) *sujeito* (11).] **11.** *Ópt.* Fonte de luz ou corpo iluminado cuja imagem se pode formar num sistema óptico. **12.** *Jur.* Aquilo sobre que incide um direito, obrigação, faculdade, norma de procedimento, proibição, etc.

Acreditamos que o significado de objeto, como usado no contexto da computação, é uma acepção bastante feliz desse termo, já que os *objetos* pretendem capturar atributos e procedimentos de entidades reais. Uma *variável*, por exemplo, é um termo de origem matemática, e usada como uma abstração de um valor armazenado na memória do computador e codificado segundo uma regra de representação de dados. O termo *objeto*, por outro lado, implica em uma representação mais concreta, mais tangível das informações manipuladas.

Objetos são definidos de duas formas: dados + funções associadas; e *instâncias* de classes. Vamos supor que já temos uma acepção correta de instância (o que veremos mais adiante) e classe, de forma que podemos usar o termo objeto conforme uma nova acepção:

> **objeto.** (..) **13.** *Inf.* Agregado de dados e operações associadas; exemplar, instância de uma classe.

O termo "instância" tem as seguintes acepções [Aur86]:

> **instância.** [Do lat. *instantia*] *S. f.* **1.** Qualidade do que é instante. **2.** Pedido ou solicitação instante, insistente: "Uma vez satisfeitas as instâncias da carne ou o orgulho pessoal, chega a saciedade." (João Gaspar Simões, *O Mistério da Poesia*, p. 230) **3.** Pedido urgente e repetido. **4.** *Jur.* Jurisdição; foro (ô). **5.** *Jur.* Série de atos dum processo, desde a sua apresentação e um juiz ou tribunal até a sentença decisória. **6.** *Jur.* Ordem ou grau da hierarquia jurídica. **7.** *Psican.* Segundo Freud [v. *freudiano*], cada uma das diferentes partes do psiquismo considerado como elemento dinâmico. [V. *id*, *ego* e *superego*]

Como utilizado em computação, e mais especificamente em aspectos relativos as paradigma de objetos, a acepção, conforme trecho extraído de [Oxf71], é a seguinte:

> **Instance**. (..) (4) in med. Schol. L. an objection to a general statement, an instance to the contrary, transl, Gr. ενστασισ (..) III In Scholastic Logic, and derived senses **5.** A case adduced in objection to or disproof of a universal assertion (= med. L. *instantia*, Gr. ενστασισ..) (..) **6.** A fact or example brought forward in support of a general assertion or an argument, or in illustration of a general truth. Hence, any thing, person, or circumstance, illustrating or exemplifying something of a more general character; a case, an illustrative example. Also, in broader sense, a case occurring, a recurring occasion. (..)

O verbo inglês *to instance*, a partir do qual inventamos o verbo "instanciar", barbarismo provavelmente recente, também segundo [Oxf71] tem o seguinte significado:

> **Instance**. (..) **II 2.** *intr.* To cite an instance, to adduce an example in illustration or proof. Const. *in* (the example adduced), rarely *upon* (the matter illustrated); (..) **3.** trans. To illustrate, prove, or show, by means of an instance; to exemplify; to exhibit. (..) **4.** To cite as an instance or example, to mention in illustration. (..)

Conforme a definição do substantivo inglês *instance*, essa acepção vem do termo grego *enstasis*, através do termo latino *instantia*, em acepção datada da Idade Média, próximo portanto do fim do latim como língua falada. O seguinte verbete, retirado de [Oxf82], dá para a palavra latina *instantia* aproximadamente a mesma acepção que temos hoje para o termo português "instância".

> **instantia** ~ae, f. [INSTANS + -IA]
> 1. The fact of being present or impending; immediate applicability.
> 2. Earnestness, insistence, importunity, urgency; (mental) concentration, application.

Por outro lado, o termo grego *enstasis* é definido em [Bai50] como:

> ενστασισ, εωσ (η) [α] 1 action de se mettre à, *d'ou* action de se déposer dans, *en parl. de pierres, de calculs*, (..) || 2 direction (d'une enterprise, d'une vie, *etc.*), genre de vie (..) || 3 action de se dresser contre, *d'ou* opposition, résistence, action de presser un adversaire, objection à un argument, (..)

O termo grego sugere, na terceira acepção, uma idéia contrária, uma contestação a uma afirmação ou argumento, e se essa acepção foi incorporada ao léxico dos religiosos da Idade Média, deve ter-se prestado a muitas e ácidas disputas sobre assuntos da Fé. O significado em inglês confirma isso e vai além, estabelecendo o termo como prova concreta da validade de uma afirmação e como exemplo de um conceito mais geral e abrangente. Dessa forma, podemos acrescentar uma nova acepção do termo português "instância" com base nessas origens do termo inglês *instance*, e no sentido em que é utilizado na computação.

> **instância.** (..) **8.** *Inf.* Exemplar de uma classe, objeto; agregado de informações gerado segundo uma descrição abstrata de propriedades e operações associadas.

Quanto ao verbo "instanciar", determinar sua "origem" é muito mais complexo, já que implicaria em saber se um verbo correspondente em latim poderia ser formado (ou se se formou, realmente, na época) dessa acepção de "instância", ou se o verbo português poderia se formar diretamente a partir do substantivo *instância* (na nova acepção) ou ainda a partir do termo original grego. O verbo *instanciar*, de qualquer maneira, já foi incorporado ao nosso vocabulário, e uma vez que aceitemos como correta a acepção de *instância* proposta, o barbarismo fica mais ou menos justificado. Da mesma forma, usamos *instanciação* como ato ou efeito de *instanciar*.

Depois de definidos *objeto* e *instância*, seria o caso de completar um pequeno léxico da programação orientada a objetos, o que incluiria *classe*, *método*, *mensagem*, *herança*, *subclasse*, *superclasse*, e outros. Ocorre que, na computação, utilizamos esses termos de forma perfeitamente correta conforme suas definições, e novas acepções vão servir apenas para colocá-los formalmente no vocabulário da Língua. Vamos deixar isso de lado para analisar outro problema.

Um termo que certamente merece análise "porta tipada", que como "tradução" do termo inglês *typed port* é um barbarismo absolutamente condenável, e causa estranheza mesmo nos ouvidos de quem lida com ele de forma rotineira. Aqui é o caso de examinar a possibilidade de se usar "porta" nesse contexto, além do verbo "tipar". (Outro uso do verbo "tipar" é na expressão "linguagem fortemente tipada".)

A definição do termo *port* em inglês, segundo [Web91], são duas: a primeira e mais imediata, dá o significado de "porto", onde os navios ficam estacionados. A segunda, mais interessante, é esta:

> [2] **port** n [ME *porte*, fr. MF, gate, door. fr L *porta* passage, gate; akin to L *portus* port] (bef. 12c) 1 chiefly Scot: GATE 2 a: an opening for intake or exhaust of a fluid esp. in a valve seat or valve face b: the area of opening in a cylinder face of a passageway for the working fluid in an engine; *also*: such a passageway c: a place of access to a system 3 a: an opening in a ship's side to admit light or air or to load cargo b *archaic*: the cover for a porthole 4: a hole in an armored vehicle or fortification through which guns may be fired.

O sentido que nos parece melhor é "um lugar de acesso a um sistema": se considerarmos um *objeto* como um "sistema", então as "portas" dão acesso a ele; desse modo, o uso do termo *porta de comunicação* parece justificado.

Quanto ao verbo "tipar", precisamos novamente pesquisar as origens do verbo inglês *to type*. Antes disso, vamos verificar o significado de tipo conforme [Aur86]:

> **tipo**[1]. [Do gr. *typos*, 'cunho, molde, sinal'.] *S. m.* **1.** Aquilo que inspira fé como modelo. **2.** Coisa que reúne em si os caracteres distintivos de uma classe; símbolo: "Estêvão era o tipo do rapaz sério." (Machado de Assis, *Contos Fluminenses*, p. 86.) **3.** Exemplar, modelo. (..) **9.** *Tip.* Paralelepípedo de metal fundido (ou de madeira, nos grandes corpos), cujo olho, convenientemente entintado, imprime determinada letra ou sinal. [V. *linha-bloco*] **10.** *Tip.* Letra impressa, resultante de composição tipográfica ou de fotocomposição. [Sin.: *caráter, letra, letra de imprensa, letra de fôrma, letra redonda.*]

Em português temos o verbo "tipificar", termo jurídico, que conforme a definição de [Aur86] não se enquadra exatamente nas situações em que usamos "tipar":

> **tipificar.** [De *tip(o)*-[2] + *-ficar.*] *V. t. d.* e *p.* Tornar(-se) típico; caracterizar(-se).

Usamos "tipar" no sentido de atribuir um tipo a, determinar o tipo de, ou dotar de tipos. Na medicina (mais propriamente, no linguajar laboratorial) usa-se o termo "tipagem" (de uma amostra de sangue, e.g.), termo que seria deverbal de "tipar".

O verbo inglês *to type* tem, segundo [Web91], o seguinte significado:

> [2]**type** *vb* **typed**; **tip•ing** *vt* (1596) 1: to represent beforehand as a type: PREFIGURE 2 a: to produce a copy of b: to represent in terms of typical characteristics: TIPIFY 3: TYPEWRITE; *also*: KEYBOARD 4: to identify as belonging to a type: as a: to determine the natural type of (as a blood sample) b: TYPECAST – *vi*: TYPEWRITE (...)

O termo em latim também deriva diretamente do original grego, conforme [Oxf82]:

> **typus (typos)** -i, *m.* [Gk. τυποσ]
> 1 A bas-relief.
> 2 A surveyor's ground-plan.

Segundo [Bai50], o significado do original grego correspondente a "tipo", e o verbo que denota a ação de um produzir um "tipo", são as seguintes:

τιπος, ου (ο) [υ] **I** coup **II** marque imprimée par un coup, empreinte (en creux *ou* en relif), *particul.*: **1** marque (d'un coup, d'une blessure, *etc.*), marque des dents, marque des clous, marque des coups || **2** trace de pas || **3** empreinte de la monnaie, du fer rouge, d'un sceau **4** caractères gravés, signes d'écriture (..)

τυποω – ω [υ] **1** marquer d'une empreinte, *d'ou* cacheter, sceller || **2** figurer, former, façonner d'aprés un modèle, façonner (..) prendre une forme, un caractère déterminé (..)

No original grego, assim como em latim, o termo "tipo" representa uma marca, um sinal causado com um golpe, e essa é a acepção de "tipo" do vocabulário dos impressores. Fazer conjecturas sobre a origem da acepção mais comum do termo "tipo" (categoria, classe) não deve nos levar a lugar nenhum. É possível supor, entretanto, que o verbo inglês *to type* (no sentido usado na computação) derivou diretamente do substantivo *type*, já que o verbo grego não oferece a acepção desejada. Se isso for verdade, podemos considerar que o termo "linguagem fortemente tipada" é tão correto (ou incorreto, depende do ponto de vista) quanto o termo inglês "strongly typed language".

Para escrever esse apêndice, foram utilizadas as seguintes fontes bibliográficas:

- Novo Dicionário da Língua Portuguesa (código Aur86)
  Aurélio B. de H. Ferreira
  Editora Nova Fronteira, Rio de Janeiro (RJ), 1986.
- Dictionnaire Grec Français (código Bai50)
  A. Bailly
  Librarie Hachette, Paris, 1950.
- The Compact Edition of the Oxford English Dictionary (código Oxf71)
  Oxford University Press, Oxford, 1971.
- Oxford Latin Dictionary (código Oxf82)
  Oxford University Press, Oxford, 1982.
- Webster's Ninth New Collegiate Dictionary (código Web91)
  Merrian-Webster Inc., 1991