UNIVERSIDADE FEDERAL DE SANTA CATARINA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Clayton Bonelli

INTEGRAÇÃO DE SISTEMAS LEGADOS COM A TECNOLOGIA DE OBJETOS DISTRIBUÍDOS

Dissertação submetida à Universidade Federal de Santa Catarina como parte dos requisitos para a obtenção do grau de Mestre em Ciência da Computação

Prof. Dr. João Bosco Mangueira Sobral

Florianópolis, Maio de 2001

Integração de Sistemas Legados com a Tecnologia de Objetos Distribuídos

Clayton Bonelli

Esta Dissertação foi julgada adequada para a obtenção do título de Mestre em Ciência da Computação Área de Concentração Sistemas de Computação e aprovada em sua forma final pelo Programa de Pós-Graduação em Ciência da Computação.

João Bosco Mangueira Sobral, Dr.

Fernando Álvaro O. Gauthier

Banca Examinadora

Hugo Fuks, Ph.D.

Rosvelter João Coelho da Costa, Dr.

Luiz Carlos Zancanella, Dr.

Daniela Barreiro Claro, Msc.

Este Trabalho é para todos os meus entes queridos

que colaboraram com a realização

deste, incentivando-me.

AGRADECIMENTOS

À UFSC,
pelo curso de graduação e pós-graduação em Ciências da Computação, responsável por todo o meu embasamento teórico.

Ao meu orientador, Prof. Dr. João Bosco M. Sobral,
por todos os momentos que se fez presente.

Aos Professores e Colegas do Curso de Mestrado em Ciências da Computação,
pela presença em todos os momentos como amigos e profissionais.

A minha mãe, irmã, irmão, namorada e demais entes queridos,
por fazerem parte da minha história de vida.

A minha amiga Angelita Marçal Flores,
Pelo companheirismo e ajuda com a formatação deste trabalho.

SUMÁRIO

RESUMO

Nos últimos quarenta anos de história da computação, foram escritos muitos aplicativos, ao redor do mundo. Muitos destes podem ainda estar sendo utilizados. Com o surgimento de novas tecnologias, em especial os objetos distribuídos, as empresas começaram a desejar que seus sistemas legados fossem integrados a esta tecnologia. Entretanto, muitas das linguagens utilizadas na construção destes sistemas legados, não permitem a utilização de construções orientadas ao objeto, nem reconhecem objetos distribuídos como objetos CORBA. Através de técnicas que utilizam DLL's e programas residentes, os sistema legados poderão não apenas ser integrados a infra-estrutura CORBA, como também poderão estar disponíveis para execução em redes, além de serem usados dentro de uma arquitetura cliente-servidor de N camadas.

ABSTRACT

For the last forty years int the Computer Science history, have been written so many applications all over the world. Many of these ones can have been still used. With new technologies, mainly with distributed objects, most of business started to desire that their legacy systems were integrated to such technologies. However, many of these languages, which were used in developing these systems do not allow that they can develop new technologies towards object oriented,neither recognize distributed objects such as CORBA objects. Through the techniques that using DLL's and resident programs, the legacy systems can not only be integrated to CORBA infra-structure but also, can be available to execute it through a network. Besides this it can be used in a client-server architecture of N-tier.

1. INTRODUÇÃO

Nas últimas décadas muitos softwares foram construídos ao redor do mundo. Muitos destes sistemas, podem ainda estar em funcionamento nas mais diversas empresas. Estes aplicativos tem como característica principal o fato de : a) continuarem ainda em funcionamento; b) o custo de manutenção, corretiva ou de ampliação, ser muito alto para o cliente. Para estes aplicativos foi dado a denominação de sistemas legados.

Em todo este período, no qual muitos aplicativos foram construídos. Uma área que também teve sua contribuição, Foi o de *softwares* desenvolvidos para funcionarem no sistema operacional DOS. Tais sistemas legados, de modo geral, utilizavam-se de linguagens como Pascal, Clipper, C, Fotran, Assembly, entre outras.

Os empresários que ainda utilizam em seus negócios aplicativos feitos para DOS, e construídos a muito tempo, perceberam que para manterem-se competitivos precisariam que seus sistemas de computação evoluíssem e adotassem tecnologias mais modernas, como a Internet e os objetos distribuídos.

Entretanto, os aplicativos construídos a muito tempo, ou que utilizam linguagens com várias décadas de idade, e que não evoluíram tecnicamente, podem não estar capacitados para trabalharem com outras tecnologias, por uma simples limitação técnica da própria linguagem.

O objetivo deste trabalho é mostrar ao leitor uma técnica conhecida como *wrapper* cujo propósito é criar uma camada de software "ao redor" dos sistemas legados permitindo, desta forma, que estes aplicativos façam uso de objetos distribuídos e da Internet. Com a utilização dos *wrappers*, sistemas legados mesmo que tenham sido

construídos usando o paradigma estruturado, podem ser utilizados como se eles fossem orientados ao objeto. De forma semelhante, mesmo que as linguagens usadas na construção dos sistemas legados não possuam comandos próprios para uso com objetos distribuídos, ou a Internet, será através dos *wrappers* que tal limitação deixará de existir.

Ao se realizar uma pesquisa, é possível encontrar artigos que versam a respeito de sistemas legados. Lendo tais artigos, entretanto, é perceptível que o enfoque dado, para o reaproveitamento dos sistemas legados, na grande maioria, diz respeito aos aplicativos desenvolvidos para *mainframes*, ou então, para sistemas operacionais multi-usuário, como é o caso do Unix. O motivou a realização deste trabalho foi a carência de artigos que tratem do reaproveitamento de código de sistemas legados, desenvolvidos especificamente para o DOS.

Dos artigos encontrados, destaca-se *Wrapping Legacy Code with Visual Basic 4.0 Objects* [Tom Gordon] e *Object Management Group & Its Developments* [Chan].

Tom Gordon, com o seu artigo *Wrapping Legacy Code with Visual Basic 4.0 Objects*, relata como uma aplicação legada, escrita em linguagens, como C, COBOL ou Fortran, podem ser reutilizadas, por programas escritos em Visual Basic. Este artigo demonstra que, através da criação de arquivos, no formato DLL, é possível a perfeita integração com o Visual Basic. Sobre os pontos fracos, estão: a necessidade da existência de todos os fontes do código legado, caso contrário o arquivo DLL não poderá ser criado; a necessidade da utilização de uma ferramenta, que crie arquivos no formato DLL de 32 bits, a partir dos códigos fontes do sistema legado; a linguagem Visual Basic, não é uma linguagem multiplataforma, como é o caso do Java.

Francis Chan, com o seu artigo *Object Management Group & its Developments*, mostra a infra-estrutura oferecida pelo CORBA e como, através do uso de um arquivo IDL, é possível criar *wrappers* para reutilização de componentes legados. Este artigo deu a idéia de utilizar o CORBA, junto com o Java, para permitir uma independência de plataforma, e para criar *wrappers*, para os sistemas legados.

Os capítulos deste trabalho estão divididos da seguinte forma: no capítulo 2 é mostrado o conceito de *wrappers*; no capítulo 3 é descrito a técnica conhecida como *wrapper* de tela; no capítulo 4 é descrito a técnica conhecida como *wrapper* de banco de dados; no capítulo 5 é descrito a técnica conhecida como *wrapper* de código; no

capítulo 6 é apresentado o protocolo de comunicação *Java Native Interface*; no capítulo 7 é mostrado um estudo de caso onde foi implementado a técnica de *wrapper* de tela junto com a linguagem Clipper; no capítulo 8 será mostrado um estudo de caso com exemplos da técnica de *wrapper* de código para a linguagem Pascal; no capítulo 9 é apresentado propostas para futuros trabalhos; e, no capítulo 10, as conclusões.

2. *WRAPPER* : O INVÓLUCRO PARA SISTEMAS LEGADOS

Nos últimos 50 anos de história da computação, foram construídos diversos tipos de aplicativos de software. Destes, é possível que muitos ainda estejam sendo utilizados em diversas empresas ao redor do mundo. É possível imaginar que aplicativos com mais de 40 anos de idade sejam difíceis de encontrar ainda em funcionamento, mas esta dificuldade não ocorre com aplicativos construídos mais recentemente, por exemplo, nas décadas de 80 e/ou 90.

Para estes aplicativos ainda em funcionamento nas empresas, é dado o nome de Sistemas Legados. Dentre estes, muitos dos aplicativos desenvolvidos foram construídos, especificamente, para o sistema operacional DOS. Não é difícil de lembrar como a computação pessoal evoluiu a partir do início da década de 80, e como o DOS se tornou quase que um padrão nos microcomputadores.

Os sistemas legados em funcionamento no DOS, de modo geral, foram escritos em linguagens como o Pascal, o Clipper, o Fortran, entre outras. Tais linguagens tem em comum o uso do paradigma estruturado.

Quando o assunto é a integração de sistemas legados com objetos distribuídos, deve ser considerado, que a linguagem utilizada na construção do aplicativo legado, pode não ter nenhum tipo de sintaxe que seja compatível com a programação orientada ao objeto. Neste caso, criar os objetos distribuídos na própria linguagem de construção do sistema legado, se torna uma tarefa impossível. O mesmo vale para a Internet, pois a linguagem pode não ter nenhuma construção sintática que possibilite a ligação com a rede mundial de computadores.

Um desenvolvedor de software pode imaginar que para resolver o problema da limitação técnica da linguagem, bastaria rescrever o sistema legado, utilizando uma linguagem mais moderna. Mas esta decisão deve levar em conta os seguintes aspectos:

- Tempo de desenvolvimento: um sistema legado composto de dezenas de milhares de linhas de código, para ser reescrito, pode resultar em um esforço de programação que pode levar meses, ou até anos, para o seu término. Talvez, o tempo necessário para conclusão desta tarefa seja longo demais para atender às necessidades do cliente. É importante lembrar que muitos clientes podem necessitar de uma solução rápida, caso contrário correm o risco de empresas concorrentes tirá-los do mercado.

- Custo financeiro: o custo do projeto é influenciado por diversos fatores, dentre os quais: número de profissionais que compõe a equipe; complexidade técnica inerente à reescritura do sistema legado; tempo necessário para a conclusão da tarefa; aprendizado das regras de negócio da empresa; entendimento do funcionamento do sistema legado; estudo de arquivos fontes e de outras documentações. Estes fatores podem significar para o cliente um custo financeiro superior ao custo original do sistema legado, isto é, o custo que o sistema legado teve quando o mesmo foi implantado para cliente.

- Regras de negócio: todo software quando é construído possui muito mais que um amontoado de linhas de código escritas em uma linguagem qualquer. As linhas de código, além de solucionar um determinado problema, precisam necessariamente preservar as regras que definem o negócio da empresa. Toda empresa possui as suas próprias regras e um software para funcionar corretamente deve preservá-las. As regras de negócio devem preservar as operações, relacionamentos e a estrutura de negócio existente na empresa (Wijegunaratne, 1998). Quando um aplicativo é reescrito, tais regras devem ser mantidas a qualquer custo. Para isto, algumas vezes o desenvolvedor terá que estudar o funcionamento da empresa, entrevistar os usuários, estudar a documentação/manuais/arquivos fontes, bancos de dados e outras informações, necessárias para a preservação das regras de negócio. O desenvolvedor poderia pensar que, se uma reescrita está sendo feita, a preservação das regras do negócio ocorre de forma automática, à medida que os arquivos fontes estão sendo traduzidos para a nova

linguagem. Este pensamento, por parte do desenvolvedor, serve como ponto de partida para a discussão sobre a ausência dos fontes.

- Ausência dos fontes: um sistema legado, independente da época do seu desenvolvimento, pode não ter mais disponível os seus fontes. Muitos sistemas são adquiridos como pacotes de empresas que desenvolvem aplicativos, que de modo geral, disponibilizavam para os clientes apenas os arquivos binários. Outras situações ocorrem quando sistemas legados são desenvolvidos especificamente para os clientes onde os fontes não necessariamente estão disponíveis. Quando os arquivos fontes não podem ser usados, será muito difícil realizar a reescrita do sistema legado. Na verdade, ao invés de uma reescrita, a equipe de desenvolvimento deverá refazer o aplicativo, para isto, o processo de reengenharia terá de ser executado. Devido à reengenharia, a equipe de desenvolvimento poderá precisar: entrevistar os usuários, contactar a equipe de desenvolvimento original do aplicativo legado, estudar os manuais, estudar os arquivos de help, estudar os dicionários de dados, enfim, tudo o que puder ajudar a reconstruir o sistema legado. Presume-se que, se um sistema legado levou anos para ficar consistente, o mesmo período de tempo seria necessário para a sua reescrita. Muitas vezes o cliente depende do software para permanecer competitivo, e nenhuma empresa irá investir na reescrita, se ao final desta tarefa, o aplicativo reescrito possuir inconsistências. Um aplicativo com incorreções pode levar o cliente a perder espaço de mercado, e nenhuma empresa irá investir se este risco existir (Brodie, 1995).

- Formato dos arquivos do banco de dados: muitos sistemas legados utilizam bancos de dados para armazenamento de informações. Estes bancos de dados podem estar em vários formatos, desde simples sistemas de arquivos, até complexos Sistemas Gerenciadores de Base de Dados. A equipe de desenvolvimento deverá conhecer profundamente cada formato de banco de dados utilizado, cada tabela, cada campo existente nas tabelas, cada regra de atualização, e assim por diante. Todas estas características, inerentes à forma pela qual os dados são armazenados, atualizados e recuperados, tem de ser muito bem conhecidas pela equipe de desenvolvimento.

Após o término da análise do sistema legado, a equipe de desenvolvimento poderá concluir que uma reescrita do aplicativo é inviável. Neste caso, a utilização de wrappers deverá ser considerada como uma solução ao problema. Segundo a definição existente

em (Mowbray, 1995), um *wrapper* é uma camada de software colocada entre o sistema legado e outros softwares externos, com o objetivo de estabelecer a comunicação entre o aplicativo legado e os softwares externos.

A técnica de wrapper corresponde a uma camada de software criada para encapsular o sistema legado de tal forma que o mesmo passa a ser tratado pelos demais softwares externos como sendo orientado ao objeto, mesmo que o sistema legado não tenha sido construído usando este paradigma. Isto é possível porque o wrapper é construído usando o paradigma orientado ao objeto e, do ponto de vista de outros aplicativos, para acessar o sistema legado, a única forma será via wrapper, através de seus métodos e atributos. Toda funcionalidade existente dentro do aplicativo legado, e que deve ser executada por outros softwares externos, deverá possuir um mapeamento correspondente a um método, definido dentro da própria classe de wrapper. Este mapeamento é feito da seguinte forma: um software externo qualquer, através do wrapper, solicita a execução de um método. Este método, internamente, enviará comandos para o sistema legado, que será o real responsável pela execução da tarefa solicitada. Por exemplo, considere que o sistema legado tem a tarefa de calcular um saldo. Esta tarefa de calcular o saldo será mapeada, na classe de wrapper, através da criação de um método chamado, por exemplo, "getSaldo". Este método será a única forma que softwares externos terão de obter o saldo existente dentro do sistema legado. A Figura 2.1 ilustra este exemplo:

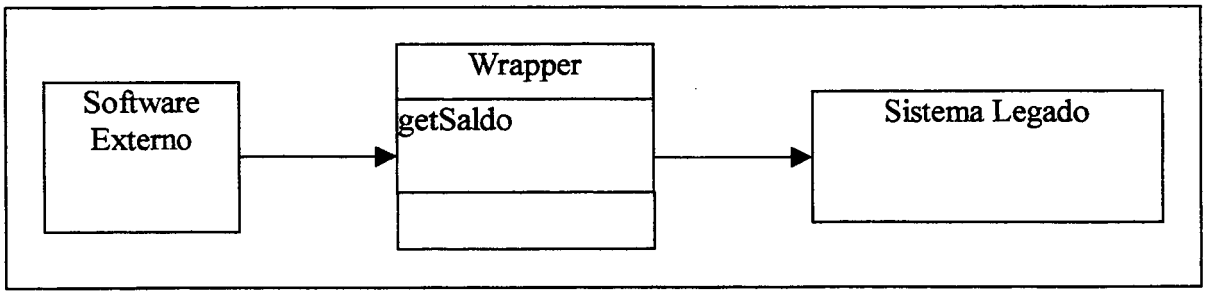

Figura 2.1 – Exemplo de mapeamento do wrapper

2.1. Arquitetura do Wrapper

O wrapper é usado para que tarefas existentes dentro do sistema legado possam estar passíveis de acesso por outros softwares externos. Este acesso se dá através da

ativação de um ou mais métodos, definidos dentro da classe de wrapper. Neste trabalho, o wrapper será dividido de acordo com a Figura 2.2.:

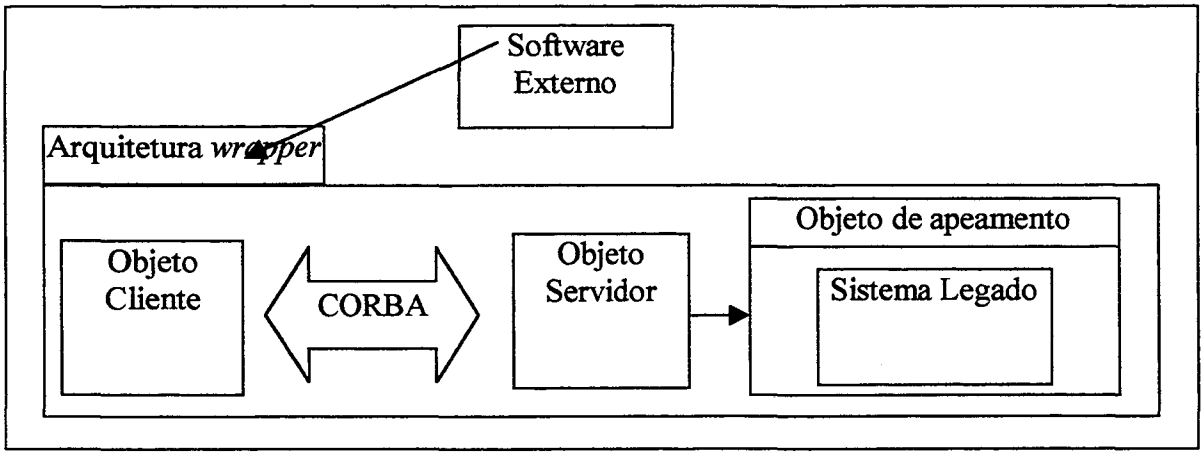

Figura 2.2 – Arquitetura dos Wrappers

O objeto-cliente: através de instâncias desta classe, os softwares externos farão solicitações de serviços pela invocação de métodos existentes dentro deste objeto. Além dos serviços oferecidos pelo sistema legado, o objeto-cliente possui o código necessário para a comunicação com o CORBA.

O objeto-servidor: este objeto recebe solicitações enviadas pelo objeto-cliente, pedindo que determinados métodos sejam executados, e as encaminha para o objeto de mapeamento. O objeto-servidor, além dos serviços oferecidos pelo sistema legado, possui o código necessário para a comunicação com o CORBA.

O objeto de mapeamento: esta classe é responsável pelo mapeamento com o sistema legado. Os métodos definidos dentro desta classe, apenas farão chamadas para os serviços existentes dentro do sistema legado. Sua finalidade é criar uma separação entre os códigos, necessária para o mapeamento com o sistema legado, e os códigos específicos da comunicação com o CORBA. É através desta separação que o desenvolvedor, caso resolva no futuro utilizar alguma outra tecnologia de objetos distribuídos, poderá fazê-lo, sem que a classe de wrapper necessite ser alterada. Apenas as classes de comunicação vão precisar sofrer modificações.

2.2. Tipos de Wrapper

Os três tipos existentes são: Wrapper de Tela, Wrapper de Banco de Dados e Wrapper de Código. Cada um destes tipos serão tratados neste capítulo.

Quando a tecnologia de objetos distribuídos deve ser incorporada a um sistema legado, existem três tipos de enfoques que podem ser adotados: wrapper de tela; wrapper de banco de dados e wrapper de código. O uso de um (ou mais) destes enfoques, dependerá de diversos fatores, como custo financeiro, tempo para conclusão, complexidade do aplicativo legado, disponibilidade de fontes/documentações, entre outros. Qualquer um dos tipos de wrappers possui um nível diferente de complexidade e pode ser usado de forma separada ou em conjunto.

3. *WRAPPER* DE TELA

Um dos enfoques que possui menor grau de complexidade, se comparado aos demais tipos de *wrappers*, é o chamado *wrapper* de tela, O fato de possuir um grau menor de complexidade, se deve ao fato de a equipe de desenvolvimento não precisar ter um conhecimento em profundidade, do funcionamento do sistema legado e/ou suas estruturas internas. O *wrapper* de tela também é conhecido como "simulação do pressionamento de teclas".

3.1 Simulação do pressionamento de teclas

Em uma simulação de pressionamento de teclas, o *wrapper* envia ao aplicativo legado (uma seqüência de teclas) para que o sistema legado execute uma determinada tarefa. A execução desta tarefa ocorre da mesma forma, como se o usuário tivesse digitando, ele próprio, as teclas na frente de seu computador pessoal.

A figura 3.1, corresponde a uma tela de entrada de dados de um hipotético sistema legado. Para que um usuário consiga chegar até ela, é necessário selecionar as opções de um hipotético menu de escolhas:

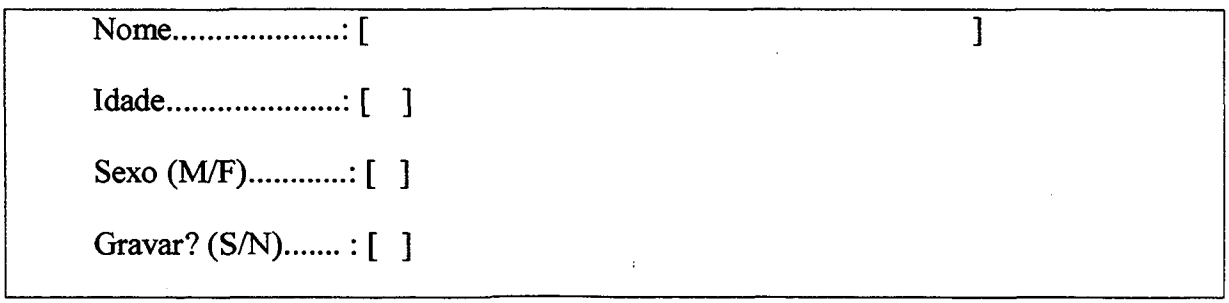

Figura 3.1 – Tela de entrada de dados

Caso um *wrapper* de tela seja utilizado, este deverá enviar para o sistema legado a mesma seqüência de teclas, para que a funcionalidade oferecida pela tela ilustrada na figura 3.1, seja obtida.

Pode-se destacar os seguintes benefícios deste tipo de *wrapper*:

- a rápida construção do *wrapper*, sem que a equipe de desenvolvimento tenha de se inteirar completamente das partes mais intrínsecas do sistema legado. A expectativa é que o custo do projeto seja reduzido, e o tempo para o seu término. Este é, portanto, um dos benefícios mais claros deste tipo de *wrapper*;

- o wrapper de tela pode ser utilizado mesmo que não estejam disponíveis os fontes do aplicativo legado;

- a consistência da lógica de negócio, mantida pelo aplicativo legado, será preservada, uma vez que o wrapper de tela simplesmente solicita um serviço que já está implementado pelo sistema legado. Como o software legado já mantém a lógica de negócio coesa, também esta lógica será mantida, mesmo que a solicitação do serviço parta do wrapper;

- outro benefício é que as antigas telas de entrada de dados, baseada em caracter, poderão receber uma aparência mais moderna, utilizando interfaces gráficas;

- o sistema legado será utilizado por outros softwares externos como se ele fosse orientado ao objeto, mesmo que o aplicativo legado não tenha sido construído usando este paradigma. A figura 3.2 ilustra uma classe que possui um método para incluir uma pessoa em um sistema legado.

```
public class WrapperTela {
   public native void incluiNovaPessoa( String nome, int idade, char sexo );
   static{ System.loadLibrary( "wrapperTela" ) };
}
```

Figura 3.2 – Exemplo de classe para wrapper de tela

Na figura 3.2, o *wrapper* possui um método público no qual é responsável pela inclusão, hipotética, das informações de uma pessoa no sistema legado. Tal método internamente irá transformar as informações passadas como parâmetro, em uma seqüência de teclas, sendo que esta seqüência será enviada para o sistema legado. Do

ponto de vista do usuário da classe *wrapper*, o serviço solicitado nada mais é que a execução de um método. Não interessa, ao usuário, que internamente a implementação do método seja feita através do envio de seqüências de teclas para o aplicativo legado. Para o usuário, apenas importa que o serviço seja realizado e não quem, efetivamente, executa o serviço.

No se que se refere às desvantagens, pode-se destacar os seguintes aspectos:

- O *wrapper* de tela, após ter sido construído, terá uma ligação explícita com os campos das telas de entrada de dados do aplicativo legado. Isto que dizer que qualquer alteração na ordem dos campos de leitura da tela do sistema legado, ou mesmo de qualquer menu de opção (que leva à execução da tela de entrada de dados), fará com que o *wrapper* de tela seja afetado. Caso seja feita alguma manutenção no sistema legado que resulte na alteração de qualquer tela de entrada de dados, o *wrapper* de tela também deverá ser alterado, do contrário ele deixará de funcionar.

- As soluções possíveis de serem implementadas pelo wrapper de tela, estão limitadas às funções que a tela do sistema legado executa. No caso de uma regra de negócio precisar ser alterada, esta não poderá ser feita através do *wrapper*.

- O caminho de comunicação entre o *wrapper* de tela e o aplicativo legado é sempre de uma via, isto é, o *wrapper* de tela pode enviar comandos para o sistema legado, mas o contrário não é possível.

3.1.1. Simulando o pressionamento de teclas no DOS

As telas de entrada de dados do sistema legado possuem campos de leitura, os quais esperam receber informações digitadas, através do teclado, pelo usuário. A forma que o *wrapper* envia esta seqüência de teclas, para os campos de leitura, deve ser feito de tal forma que o software legado "pense" que exista um usuário teclando as informações. Para que o *wrapper* envie esta seqüência de teclas, o *wrapper* poderia usar o enfoque de escrever todos os códigos das teclas dentro de um arquivo texto. De posse deste arquivo texto, o *wrapper* poderia redirecionar a entrada de dados, fazendo com que esta entrada fosse feita a partir do arquivo, e não mais do teclado. Com este

redirecionamento, o aplicativo legado aceitaria a entrada das informações não mais do teclado, o que é o *default*, mas através do arquivo texto. Na verdade, o aplicativo legado, nem mesmo saberia que as informações estão vindo de um arquivo. Para ele, os campos de leitura apenas fariam a captura de informações de um dispositivo de entrada de dados. Não interessa se este dispositivo é o teclado, ou é um arquivo.

O DOS define um grupo de dispositivos que funcionam tanto para entrada como para saída. Para cada um destes dispositivos, o DOS possui um *handle default*, o qual é criado quando o sistema operacional é inicializado. A tabela 3.1 mostra estes dispositivos e os *handles* associados:

Dispositivo	Função	Handle
COM	Entrada Saída	0 – Teclado 1 – Tela
AUX	Porta Serial	3
PRN	Impressora	4

Tabela 3.1 - Dispositivos padrão no DOS

O nosso interesse está no dispositivo COM, o qual tem o *handle* zero(0) para o teclado. Para entender o uso deste *handle*, e como o teclado pode ser redirecionado no DOS, é necessário utilizar as interrupções do DOS. Os programas escritos para funcionar no DOS, independente da linguagem utilizada, acabam sempre tendo que usar as interrupções de teclado para permitir a leitura de informações. O DOS utiliza a interrupção 21h e a função 3Fh para permitir a leitura do teclado (Tischer, 1992). Esta leitura deve ocorrer a partir do dispositivo padrão de entrada, no caso o teclado, o qual possui o *handle* de número zero(0), ou então a leitura deverá ocorrer a partir de um outro *handle*, que represente um arquivo previamente aberto.

O DOS pode ter o dispositivo padrão de entrada de dados redirecionado, através da alteração do *handle* a ser enviado para a função 3Fh, da interrupção 21h (Tischer, 1992). Quando os programas realizam as leituras, eles o fazem sempre com o *handle* zero(0) que corresponde ao teclado. Como não há nenhuma interrupção, seja DOS ou mesmo no nível da BIOS, que faça o *handle* zero(0) representar um outro dispositivo (como um arquivo) ficará difícil executar este redirecionamento. A única forma do aplicativo legado aceitar o redirecionamento, seria se ele já fosse construído de forma a permitir este redirecionamento. O normal é encontrar aplicativos legados onde a fonte

da entrada de dados é apenas o teclado. A solução encontrada para resolver este problema são os programas residentes.

De acordo com Yallouz (1991), um programa residente: "É um programa que fica na memória (residente) ao mesmo tempo que outros programas são executados.". Com esta técnica, é possível construir um programa residente; deixá-lo ativo ao mesmo tempo que o sistema legado e fazer com que este programa residente envie os códigos das teclas para o buffer do teclado. O aplicativo legado irá capturar qualquer caracter que exista no buffer do teclado. Este buffer pode ser preenchido, através da digitação do usuário, ou mesmo ser preenchido pelo programa residente. O programa residente faria a emulação da digitação do teclado, através da colocação dos caracteres, ou código das teclas, diretamente dentro do buffer de teclado. Como os campos de leitura do aplicativo legado apenas verificam se há alguma tecla dentro do buffer de teclado, não importando quem (ou como) esta tecla foi parar dentro do buffer, o aplicativo irá realizar normalmente a leitura das informações.

O buffer do teclado pode ser manipulado simplesmente através da interrupção 16h da BIOS (Tischer, 1992). Através do uso desta interrupção, o programa residente poderá colocar todas as teclas que forem necessárias, dentro do buffer de teclado, para posterior captura pelas telas do sistema legado.

Os programas residentes, para serem ativados, necessitam de um conjunto de chamadas especiais. Entretanto, como regra geral, basicamente é necessário escolher uma das interrupções de hardware existentes no computador, e associar a esta interrupção um trecho de código, de tal forma que no momento em que esta interrupção de hardware ocorrer, este trecho de código seja executado (Yallouz, 1991).

Existe na BIOS uma área de memória, chamada de "vetor de interrupções", onde é armazenado o endereço de cada rotina, interna à própria BIOS, responsável por tratar a interrupção correspondente, quando a mesma é executada. Desta forma, por exemplo, quando se digita uma tecla qualquer no teclado, é executado uma interrupção de hardware de número 09h. A esta rotina está associado um determinado programa, da própria BIOS, que faz o tratamento da INT 09h. O que um programa residente faz é, temporariamente, definir uma nova rotina para realizar o tratamento de uma interrupção qualquer. Para isto, é necessário se utilizar de comandos especiais que permitam alterar

o vetor de interrupções, mudando a rotina de tratamento original da BIOS, para uma outra rotina. Após a alteração do vetor de interrupções, é importante que o novo tratador de interrupção continue a executar o antigo tratador, aquele definido pela BIOS. Caso contrário, o novo tratador teria de refazer todos os códigos existentes dentro da rotina tratadora original do BIOS.

Para utilizar um único programa residente, que possa funcionar com qualquer conjunto de teclas, o arquivo texto se torna indispensável. Caso o arquivo texto não fosse usado, seria necessário construir um programa residente para cada conjunto de teclas a ser enviado para o aplicativo legado. Por exemplo, se houvesse duas telas no sistema legado, uma para cadastrar o endereço de um cliente, e outra para cadastrar o saldo de uma conta bancária, seria necessário a existência de dois programas residentes: um para enviar a seqüência de teclas para o cadastro do endereço, e outro programa residente para enviar a seqüência de teclas para o cadastro do saldo bancário. Com o uso do arquivo texto, somente será necessário um único programa residente. Este programa apenas lerá o conteúdo dos arquivos texto e, independente de seu conteúdo, o programa residente enviará os códigos das teclas ali existentes para o buffer do teclado.

3.2 Usando o wrapper em um ambiente distribuído

Para a utilização do wrapper de tela em um ambiente distribuído, é necessário criar um conjunto de objetos, para servir às requisições dos clientes e um conjunto de objetos para responder a estas requisições. Estes objetos são encarregados de estabelecer a comunicação entre os clientes e os servidores. O objeto servidor, por sua vez, irá ativar um determinado método, definido dentro de uma classe de mapeamento. Esta, por sua vez, será a verdadeira responsável pela comunicação com o sistema legado.

A equipe de desenvolvimento pode acreditar que não existe a necessidade de uma classe de mapeamento, mas este pensamento deve ser reavaliado. Se a comunicação com o sistema legado fosse colocada diretamente dentro do objeto-cliente, isto acarretaria em uma carga desnecessária no objeto-cliente. Cada objeto-cliente teria de possuir internamente, além dos códigos necessários para estabelecer as comunicações CORBA, os códigos exigidos para a comunicação com o sistema legado.

Além disso, se existisse a necessidade de uma alteração do código de mapeamento, esta alteração teria de ser propagada entre todas as máquinas que se utilizam deste objeto-cliente, o que no mundo da WEB, pode significar a alteração de centenas, ou até milhares, de equipamentos.

Fazer o mapeamento diretamente no objeto-servidor deve ser uma decisão de projeto das classes, mas segundo Ambler (1998), as classes devem ter uma única e clara função. Se for necessário uma classe para cuidar de questões relacionadas à comunicação com objetos remotos, então esta classe deverá ter apenas esta função. Questões relativas ao funcionamento do mapeamento com o sistema legado, devem ser deixadas para uma outra classe. A classe de mapeamento deve se preocupar apenas com questões diretamente pertinentes à comunicação com o sistema legado, como a execução do programa residente e a montagem do arquivo texto, com os códigos das teclas.

Imaginando a existência de um serviço no sistema legado para o armazenamento de informações de uma pessoa, onde este serviço precisa ser ativado remotamente por um cliente, para se conseguir esta ativação remota, é necessário criar uma estrutura de objetos que permita tal comunicação. Por isso, é necessário escrever alguns arquivos fontes, usando a linguagem Java e a linguagem IDL, onde neste último, seria colocado a definição de todos os serviços a serem invocados remotamente. Os demais arquivos-fontes a serem criados, além do arquivo IDL, correspondem aos arquivos contendo as definições da classe-cliente e da classe-servidor.

Como exemplo, a figura 3.3 ilustra o código de um arquivo IDL, contendo a definição de um método, hipotético, correspondente ao serviço de armazenamento de informações de uma pessoa. Maiores detalhes sobre o arquivo IDL podem ser obtidos em Orfali (1998).

```
module idlWrapperTela{
interface iWrapperTela{
void incluiPessoa( in string codigo, in string nome, in string endereco );
};
};
```

Figura 3.3 – Exemplo de código IDL

No que se refere ao objeto-cliente, esta classe deve possuir, além dos comandos específicos para colocar o CORBA em funcionamento, o método "incluiPessoa()", previamente definido no arquivo IDL. A implementação do método "incluiPessoa()" do objeto-cliente consistirá apenas de uma única linha de código, correspondendo a chamada de um método, de mesmo nome, existente na classe-servidor.

Para a criação do objeto-cliente, antes de mais nada é necessário gerar um arquivo Java a partir do arquivo IDL, contendo os códigos de ativação para o CORBA. Isto é feito através do utilitário idl2java (Orfali, 1998). A figura 3.4 mostra a listagem completa do objeto-cliente:

```
public class objClienteRemoto{
  private idlWrapperTela.iWrapperTela objeto;
  objClienteRemoto( String[] args ){
    try{
      org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init( args, null );
      objeto = idlWrapperTela.iWrapperTelaHelper.bind( orb, "clayton" );
    }
    catch( org.omg.CORBA.SystemException e ){
      System.err.println( e );
    }
  }
  public void incluiPessoa( String codigo, String nome, String endereco ){
    objeto.incluiPessoa( codigo, nome, endereco );
  }
}
```

Figura 3.4 – Listagem do objeto-cliente

Na implementação do método "incluiPessoa()", note que existe apenas a solicitação da execução do serviço, de mesmo nome, existente no objeto-servidor, aqui representado pela variável chamada "objeto".

O objeto-servidor é responsável pela execução dos serviços solicitados pelo objeto-cliente. É importante lembrar ao leitor que o objeto-cliente solicita a execução de um determinado método, existente dentro do objeto-servidor, no exemplo chamado de "incluiPessoa()". Este método, no objeto-servidor, deverá internamente solicitar a execução de um outro método, desta vez, existente dentro da classe de mapeamento.

A figura 3.5, mostra a listagem do objeto-servidor.

```
import wrapperTela;
public class objServidor extends idlWrapperTela._iWrapperTelaImplBase{
```

```
public objServidor(java.lang.String name){
super(name);
}
public objServidor(){
super();
}
public void incluiPessoa( String codigo, String nome, String endereco ){
wrapperTela wt = new wrapperTela();
wt.incluiPessoa( codigo, nome, endereco );
}
}
```

Figura 3.5 – Listagem do objeto-servidor

Na figura 3.5, a implementação do método "incluiPessoa()", faz a solicitação de execução do método, de mesmo nome, definido dentro da classe "wrapperTela". Para isto é utilizado a variável "wt", que representa uma instância da classe de mapeamento. É esta classe de mapeamento que, efetivamente, executará o programa residente e fará a comunicação com o sistema legado.

No capítulo 7 será mostrado um exemplo de classe de mapeamento com o código necessário para executar o programa residente e estabelecer a comunicação com um hipotético sistema legado.

4. *WRAPPER* DE BANCO DE DADOS

O *wrapper* de banco de dados necessita de uma fase maior de estudos por parte da equipe de desenvolvimento. Este estudo é necessário porque o banco de dados legado possui conjuntos de regras de atualização que preservam a lógica de negócio da empresa. São estas regras de negócio que precisam ser entendidas e, principalmente, preservadas pela equipe de desenvolvimento. O resultado deste estudo, envolverá a criação de uma camada de *software* que permita que programas externos possam se comunicar com o banco de dados legado. Esta camada de *software* é necessária porque permite uma independência dos programas externos da lógica de negócios existente no banco de dados legado. Tal camada de software é denominado de "*wrapper* de banco de dados" e a figura 4.1 mostra como o *wrapper* de banco de dados se comunica diretamente com o banco de dados legado:

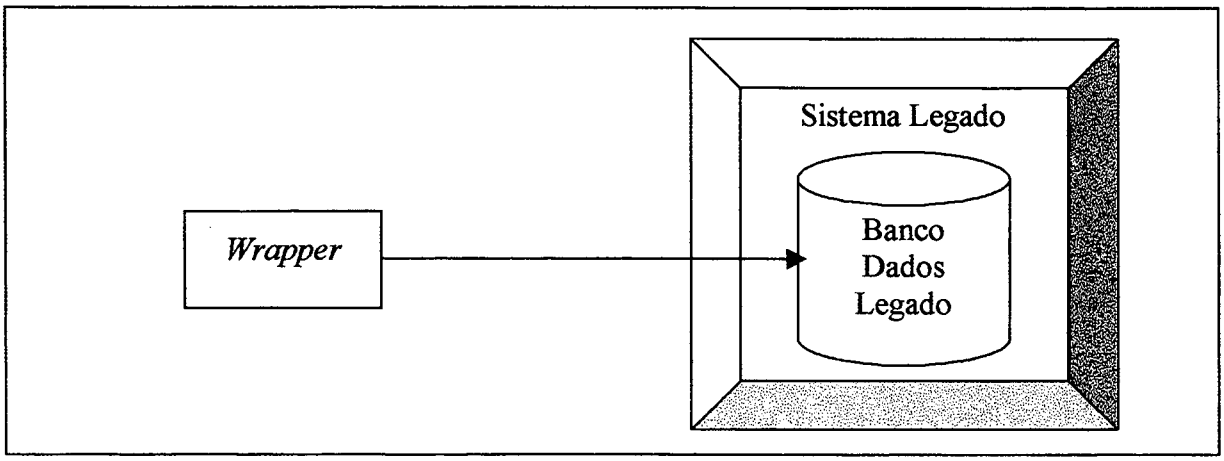

Figura 4.1 – Ligação do *wrapper* com o banco de dados legado

As informações digitadas em um sistema legado, normalmente são armazenadas em algum tipo de meio físico. Dentre os meios físicos de armazenamento, destacam-se

os sistema de arquivos e os bancos de dados. Os bancos de dados podem se utilizar de diferentes modelos, como o relacional, o hierárquico ou o modelo em rede (Date, 1984).

Da mesma forma que os programas fontes devem se manter consistentes às regras de negócio da empresa, o mesmo deve ser assegurado pelo banco de dados. É crucial conhecer a estrutura física de cada tabela, cada campo existente em cada tabela, seus nomes e tipos, enfim, cada detalhe interno do banco de dados legado.

4.1 Gateways

Quando o *wrapper* de banco de dados estiver sendo construído, é necessário descobrir como, efetivamente, o *wrapper* irá se comunicar com o banco de dados. Mesmo que o sistema legado trabalhe com sistemas de arquivos, ou mesmo com SGBDs, o *wrapper* terá de manipular os dados da base de dados legada. Existem diferentes fornecedores, os quais possuem produtos com características internas completamente diferentes entre si. Por exemplo, um banco de dados COBOL, internamente, pode ser completamente diferente de um banco de dados ORACLE. Produtos como estes, devem ser manipulados pelo *wrapper,* mas as diferenças internas de cada produto devem ser resolvidas pelo *wrapper* de uma forma padrão, sem que seja necessário a construção de *wrappers* diferentes, próprios para tipos de SGBD. Um fator que favorece esta padronização, são os programas fornecidos pelos próprios fabricantes dos bancos de dados. Estes programas foram feitos especificamente para outros softwares se comunicarem com os dados armazenados dentro do banco de dados, sendo denominadas de *gateway*, porque estabelecem um mecanismo e comunicação entre um programa externo e o banco de dados, executando traduções, entre as solicitações dos programas externos, e as formas de acesso internas do SGBD (Ganti, 1995). A figura 4.2 ilustra a utilização dos *gateways*, junto ao *wrapper* de banco de dados:

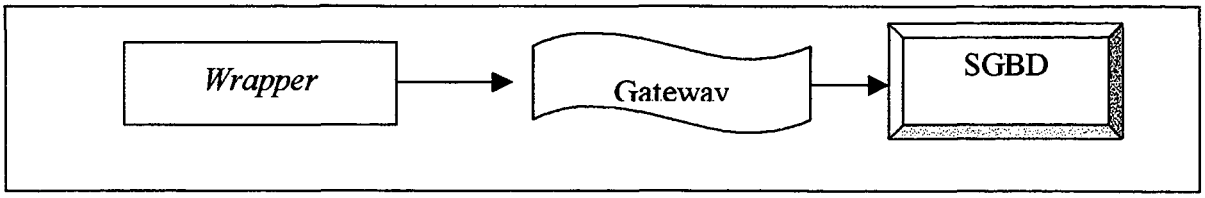

Figura 4.2 – Uso de *gateway* junto ao *wrapper* de banco de dados

4.1.1 ODBC

O ODBC é um *gateway* criado pela Microsoft, para que a equipe de desenvolvimento fique livre das particularidades de cada banco de dados. Desta forma, é possível padronizar os acessos aos bancos de dados, uma vez que o ODBC internamente se preocupa em como, um determinado comando, solicitado pelo usuário, será executado dentro de um SGBD qualquer. Como disse Funes (1999): "...um programa escrito numa linguagem (C. C++, Java, Visual Basic) faz chamadas a ODBC utilizando um conjunto de funcionalidades padrão; por sua vez a ODBC utiliza o *driver* (programa escrito pelo fabricante) para acessar o SGBD."

4.1.2 JDBC-ODBC

Apenas o uso do ODBC não é suficiente para permitir que *wrappers* em Java acessem diferentes SGBDs. Por isso, a Sun, criadora do Java, definiu um *gateway* feito especificamente para que os programas escritos em Java, possam acessar os bancos de dados. Um ponto importante é que o JDBC-ODBC, para funcionar, precisa que o ODBC esteja instalado. O JDBC-ODBC é ele próprio, um *gateway* para o ODBC. Na figura 4.3 é mostrado como o JDBC-ODBC é usado (em conjunto ao ODBC, para acesso ao SGBD):

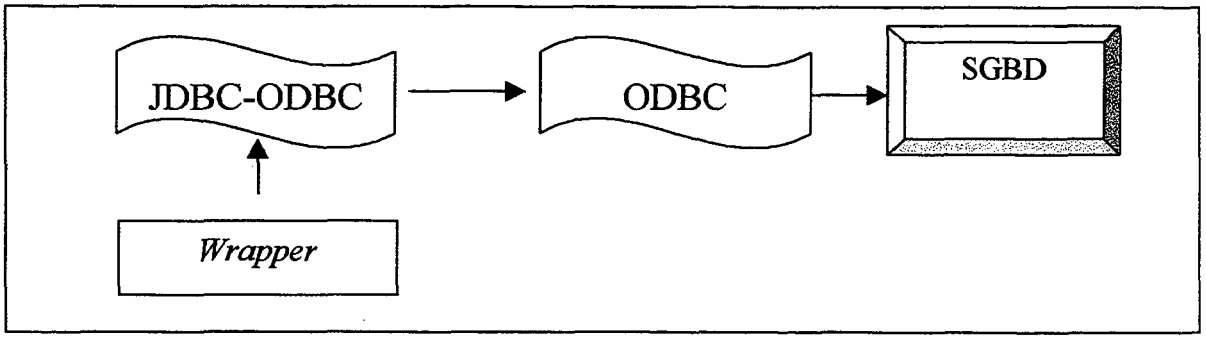

Figura 4.3 – Uso do JDBC-ODBC com o ODBC

4.2 Usando o wrapper em um ambiente distribuído

A utilização do wrapper em um ambiente distribuído, implica na criação de um conjunto de classes, para acesso aos dados legados. A partir destas classes, como já foi mostrado anteriormente, será gerado uma instância para o objeto-cliente, uma instância do objeto-servidor, e uma instância para o objeto-mapeamento. O objeto-cliente será responsável pelas solicitações feitas pelos softwares externos, o objeto-servidor será responsável por atender as solicitações feitas pelo objeto-cliente e, finalmente, o objeto-mapeamento será responsável pelos comandos para manipulação das informações armazenadas no banco de dados legado. O JDBC-ODBC, de posse destes comandos, se encarregará de enviá-los para o SGBD.

O passo inicial, para escrever as classes de comunicação CORBA é a definição do arquivo IDL contendo as especificações dos métodos a serem ativados remotamente. Estes métodos devem corresponder às mesmas definições de métodos existentes na classe de mapeamento. Como ilustração, a figura 4.4 apresenta um arquivo IDL, contendo três(3) métodos hipotéticos, para inclusão, alteração e exclusão de informações de uma pessoa, no banco de dados legado:

```
module idlWrapperBancoDados{
  interface iWrapperBancoDados{
    void inclui( in string codigo, in string nome, in string endereco );
    void altera( in string codigo, in string nome, in string endereco );
    void exclui( in string codigo );
  };
};
```

Figura 4.4 – IDL para acesso ao banco de dados legado

O objeto-cliente contém os comandos de comunicação necessários pelo CORBA, além dos mesmos métodos definidos no arquivo IDL. A implementação dos métodos do objeto-cliente, consistirá apenas de uma única linha de código, correspondendo a chamada de um outro método, de mesmo nome, existente na classe-servidor.

Da mesma forma como já foi citado anteriormente, o utilitário idl2java se encarrega de gerar, de forma automática, os arquivos necessários para a comunicação com o CORBA. A equipe de desenvolvimento terá a incumbência, de posse destes arquivos, de escrever a implementação dos métodos públicos, que serão ativados remotamente. Na figura 4.5 é apresentado a listagem do objeto-cliente:

```
public class objClienteRemoto{
  private idlWrapperBancoDados.iWrapperBancoDados objeto;
  objClienteRemoto( String[] args ){
   try{
    org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init( args, null );
    objeto = idlWrapperBancoDados.iWrapperBancoDadosHelper.bind( orb,
"clayton" );
   }
   catch( org.omg.CORBA.SystemException e ){
    System.err.println( e );
   }
  }
  public void inclui( String codigo, String nome, String endereco ){
   objeto.inclui( codigo, nome, endereco );
  }
  public void altera( String codigo, String nome, String endereco ){
   objeto.altera( codigo, nome, endereco );
  }
  public void exclui( String codigo ){
   objeto.exclui( codigo );
  }
 }
```

Figura 4.5 – Listagem do objeto-cliente do wrapper de banco de dados

Observando a implementação dos métodos "inclui()", "altera()" e "exclui()", pode-se notar que existe apenas a solicitação da execução do serviço, de mesmo nome, existente no objeto-servidor, aqui representado pela variável chamada "objeto".

O objeto-servidor será responsável por instanciar a classe de mapeamento a qual, na realidade, é quem executará os serviços solicitados pelo objeto-cliente. No exemplo, ilustrado na figura 4.5, os serviços que o objeto-cliente pode solicitar foram chamados de "inclui()", "altera()" e "exclui()".

Abaixo, na figura 4.6, é apresentado a listagem do objeto-servidor:

```
import wrapperBancoDados;
public class objServidor extends
idlWrapperBancoDados._iWrapperBancoDadosImplBase{
  public objServidor(java.lang.String name){
   super(name);
  }
  public objServidor(){
   super();
  }
  public void inclui( String codigo, String nome, String endereco ){
   wrapperBancoDados wbd = new wrapperBancoDados();
```

```
    wbd.inclui( codigo, nome, endereco );
  }
  public void altera( String codigo, String nome, String endereco ){
    wrapperBancoDados wbd = new wrapperBancoDados();
    wbd.altera( codigo, nome, endereco );
  }
  public void exclui( String codigo ){
    wrapperBancoDados wbd = new wrapperBancoDados();
    wbd.exclui( codigo );
  }
}
```

Figura 4.6 – A listagem do objeto-servidor

Na figura 4.6, a implementação dos métodos "inclui()", "altera()" e "exclui()", fazem a execução dos serviços, de mesmos nome, definidos dentro da classe "wrapperBancoDados". Para isto é utilizado a variável "wbd", que representa uma instância da classe de mapeamento. É esta classe de mapeamento que, efetivamente, executará os comandos de manipulação do banco de dados legado.

5. *WRAPPER* DE CÓDIGO

Neste capítulo abordaremos o que é chamado de *wrapper* de código. O objetivo deste tipo de *wrapper* é construir uma camada de software, que possibilite a reutilização dos programas legados. Em outras palavras, se o sistema legado for composto de partes de software, seja na forma de arquivos fontes, arquivos binários, ou ambos, o *wrapper* terá que encontrar alguma forma de interação.

5.1. Reaproveitamento de código

Para a interação do *wrapper* com os programas legados, é necessário possibilitar o reaproveitamento do código legado. Código este, que por já estar em uso dentro do sistema legado, possui as regras de negócio da empresa já codificadas. Se uma determinada funcionalidade já existe dentro do aplicativo legado, e se esta funcionalidade já foi codificada dentro de uma ou mais rotinas do software legado, o caminho mais lógico seria promover a reutilização destas rotinas. Esta atitude permitirá ganhos, não apenas no tempo de desenvolvimento, pois se a rotina está pronta e mostrou a sua funcionalidade, então não será necessário desperdiçar horas preciosas tendo de "reinventar a roda". Outro ganho importante são as regras de negócio da empresa. Por já estarem codificadas, dentro do aplicativo legado, e havendo o reaproveitamento de código, então certamente as regras de negócio serão preservadas. O ponto principal, quando *wrappers* de código estão sendo construídos, é a equipe de desenvolvimento conhecer detalhadamente as rotinas do software legado.

É necessário que os módulos do sistema legado possuam uma interface, pela qual os serviços do aplicativo legado possam ser reaproveitados pelo *wrapper*. Essa interface é materializada na forma de uma API. Esta API pode estar na forma de arquivos TPU, DLL, LIB, OBJ, entre outros.

5.2. Gerenciando o *wrapper* de código

O *wrapper* de código será construído usando a linguagem Java. No entanto, o aplicativo legado pode ter sido escrito em qualquer outra linguagem. As rotinas existentes dentro do sistema legado devem ser usadas, de alguma forma, pelo *wrapper*, independente dos fontes legados, ou arquivos binários, estarem disponíveis. É aí que se encontra a grande dificuldade na construção do *wrapper* de código. A linguagem Java se comunica apenas com a linguagem C/C++ e com nenhuma outra. Desta forma, à primeira vista, pode parecer impossível que a linguagem Java execute rotinas escritas em Pascal, Assembly ou Fortran.

Diversas técnicas podem ser usadas para resolver o principal problema inerente à construção dos *wrappers* de código: o interfaceamento entre a classe Java e as rotinas escritas em linguagens, muitas vezes, "incompatíveis" com o Java. Algumas destas técnicas foram desenvolvidas por outros autores, enquanto outras foram criadas especificamente para este trabalho. A utilização de uma ou mais das técnicas deve ser considerada mediante os fatores descritos a seguir:

- Reescrever os módulos: a técnica de reescrever os módulos deve ser muito bem pensada, pois o custo de tempo e o custo financeiro pode ser enorme, muitas vezes inviabilizando esta tarefa. Mas, caso ela seja usada, podem ser adotadas as técnicas descritas em (Brodie, 1995) e que são conhecidas pelos nome de "*Chicken Little*" e "*Cold Turkey*".

- Disponibilidade dos arquivos fontes/binários: a existência dos fontes/binários irá determinar se um *wrapper* poderá ser construído ou não. Se os módulos do sistema legado não estiverem disponíveis, seja na forma de arquivos fontes ou mesmo de arquivos binários, a equipe de desenvolvimento terá de voltar a pensar apenas nas outras duas categorias de *wrappers*: tela e banco de dados.

• A linguagem utilizada na construção dos módulos: é importantíssimo determinar qual foi a linguagem na qual o módulo legado foi inicialmente escrito, mesmo que os fontes não estejam disponíveis, mesmo que apenas os arquivos binários sejam acessíveis, é necessário determinar qual foi a linguagem inicialmente usada para construí-lo. Esta necessidade implicará em como o Java irá se comunicar com o módulo legado.

5.3. Técnicas de reaproveitamento de código

O uso de uma ou mais técnicas será ditada pelo tipo de linguagem usada na construção do módulo legado, e se os fontes/binários, estão disponíveis. Com relação à existência apenas dos binários, independente da linguagem, é crucial para utilização de um *wrapper* que os nomes das rotinas, parâmetros e o contexto na qual elas devem ser usadas, sejam conhecidas. Caso contrário, a sua reutilização se tornará inviável.

5.3.1. Chicken Little

Esta técnica diz respeito a reescrever partes do sistema legado, sendo que estas partes devem ser, preferencialmente, as menores possíveis para que as regras de negócio sejam mantidas. Inicialmente, apenas partes pequenas devem ser reescritas e, gradativamente, estas partes devem ser aumentadas até que o módulo (ou mesmo todo o sistema legado) tenha sido substituído por um novo aplicativo. Uma vez que a equipe irá se concentrar em pequenas partes do aplicativo, poucos profissionais precisarão ser usados para trabalhar nesta tarefa. Com isso, o custo de tempo e o custo financeiro tenderá a ser menor, mesmo que esta tendência não seja uma regra geral.

5.3.2. Cold Turkey

Esta técnica parte da idéia que o sistema legado inteiro deva ser rescrito, e então colocado em funcionamento para o usuário. É claro que um enfoque como este poderá

levar o sistema, após a sua reescrita, apresentar problemas que façam com que os serviços antes perfeitamente funcionais apresentem falhas. Isto poderá acarretar perdas para a empresa cliente.

5.3.3. *Wrappers* para outras linguagens

A ligação de uma classe Java, com *softwares* escritos em outras linguagens, pode não ser uma tarefa das mais simples de ser realizada. Principalmente, porque nem todas as linguagens se comunicam diretamente com o Java e também porque nem sempre os arquivos fontes estão disponíveis. A seguir, serão mostrados alguns exemplos utilizando diversas linguagens para que o leitor tenha uma idéia sobre como construir *wrappers* de código. Como regra geral, considere que será necessário estudar cada caso separadamente. Mesmo que existam diversos módulos do sistema legado, escritos usando a mesma linguagem, isto não quer dizer que o trabalho será menor. A técnica de *wrapping* pode ser conhecida pela equipe de desenvolvimento, mas o entendimento sobre as regras de negócio e, principalmente, sobre as rotinas existentes dentro de cada módulo (sua forma de chamada, seus parâmetros e sua finalidade) podem variar muito de um aplicativo legado para outro. A tarefa será ainda mais complicada no caso de o código fonte não estar disponível ou a documentação do sistema legado for de baixa qualidade. Outro ponto importante que a equipe de desenvolvimento tem que ter sempre em mente é que o módulo legado, caso seja escrito em uma linguagem diferente do Java, somente será interfaceado com o *wrapper* Java, através do protocolo de comunicação JNI, o que será abordado no capítulo seguinte. Este protocolo faz a ligação do Java com a linguagem C/C++. Por isso, é necessário a construção de um programa em C/C++ e, a partir deste programa, estabelecer a comunicação com os módulos do sistema legado. Em outras palavras, o grau de conexão que uma classe Java possui com outras linguagens está diretamente ligado à capacidade que os programas escritos em C/C++ tem de se comunicar com rotinas escritas em outros tipos de linguagens.

5.3.3.1 *Wrappers* para módulos C/C++

Um módulo legado, que tenha sido escrito na linguagem C/C++ poderá ser interfaceado diretamente com o *wrapper* Java, isto porque existe um recurso conhecido como JNI que permite que programas em Java utilizem rotinas escritas em C/C++. Todavia, a equipe de desenvolvimento poderá passar por dificuldades, uma vez que nem sempre os arquivos fontes do módulo escrito na linguagem C/C++ podem estar disponíveis para utilização junto ao *wrapper*. Para tratar deste caso mais detalhadamente, serão mostradas algumas situações e suas possíveis soluções.

- Disponibilidade dos fontes C/C++ : o sistema legado pode ter sido desenvolvido para um ambiente de 16 bits, como o DOS. Por isso, para que a equipe de desenvolvimento possa usar as rotinas existentes dentro do módulo C/C++, é necessário portá-lo para um ambiente, no mínimo, de 32 bits. Esta tarefa seria muito simples se todos os fontes necessários para o funcionamento do módulo estivessem disponíveis. Se isto ocorrer, bastará utilizar algum compilador de 32 bits (ou superior, por exemplo o Visual C++) e compilar os fontes para esta nova arquitetura.

- Disponibilidade dos arquivos fontes e binários: Existe a situação onde os fontes do módulo C/C++ utilizam rotinas pertencentes à bibliotecas de terceiros. Geralmente estas bibliotecas estão no formato binário, como arquivos OBJ ou LIBs e usando a arquitetura de 16 bits. Nesta situação, é provável que a compilação usando um compilador de 32 bits não seja possível, justamente pela presença destes arquivos binários de 16 bits. Se este for o caso, será necessário criar artifícios para que o módulo em C/C++ possa ser conectado ao *wrapper* Java. Este artifício tem a seguinte idéia básica:

1. Crie um programa executável utilizando um compilador de 16 bits e faça a ligação deste executável ao módulo C/C++, cujas rotinas são de interesse para o *wrapper* Java;

2. Dentro deste arquivo executável, faça a leitura de um arquivo texto, o qual terá uma estrutura conhecida, tanto pelo executável como pelo *wrapper* Java;

3. O arquivo texto conterá, dentro de si, dois tipos de informações: o nome de uma das rotinas existentes dentro do módulo C/C++ e os parâmetros necessários para o seu perfeito funcionamento;

4. O programa executável fará a leitura deste arquivo texto, obtendo assim o nome da rotina a ser executada e os parâmetros necessários, sendo que após esta leitura, a rotina desejada será ativada;

5. A montagem do arquivo texto será feita pelo módulo JNI e não pelo *wrapper* Java, isto para que este *wrapper* possa ficar isolado dos detalhes de implementação - necessários para a ativação das rotinas existentes dentro do módulo C/C++. A figura 5.1 ilustra esta técnica:

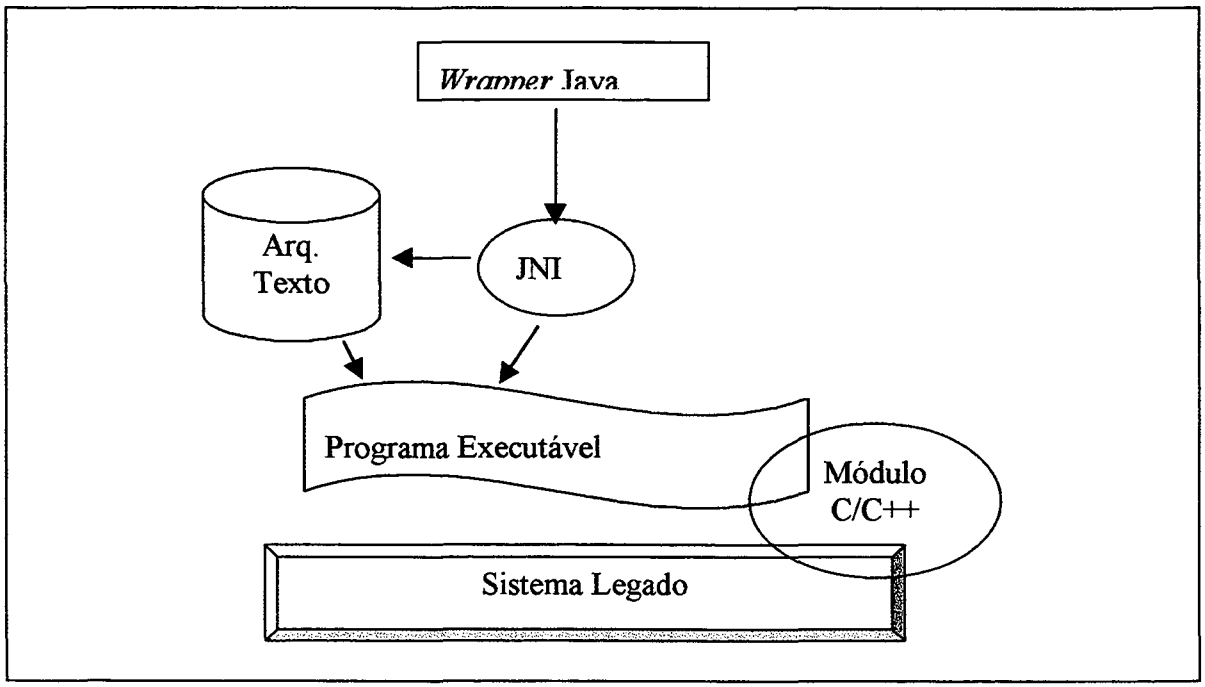

Figura 5.1 – Ligação do *Wrapper* com módulos C/C++

O leitor pode estar se perguntando para que serve o arquivo texto gerado pelo módulo JNI. Afinal, tanto o JNI como o módulo C/C++ do sistema legado estão escritos na mesma linguagem. Por isso, um pensamento que poderia surgir na equipe de desenvolvimento seria passar as informações necessárias para a ativação das rotinas do sistema legado diretamente através da memória RAM. Esta poderia ser uma solução, se não fosse um pequeno detalhe: não existe garantia que o uso de memória seja possível, devido às arquiteturas de 16 bits e de 32 bits. Nestes tipos de arquiteturas, a forma de endereçamento de memória são diferentes. Com isso, o uso de espaços de memória para compartilhar informações pode não ser possível. O arquivo texto serve, então, como uma ponte de comunicação, independente da forma pela qual os ambientes Java, JNI, ou do módulo legado tratem a memória RAM. O armazenamento e recuperação de

informações, no arquivo texto, usará sempre a mesma abordagem, independente da arquitetura utilizada. O arquivo texto também tem uma utilidade adicional: permitir uma comunicação de duas vias entre o *wrapper* e o sistema legado. O arquivo texto pode ser usado para indicar qual rotina deverá ser executada pelo módulo legado; e também pode ser usado para armazenar qualquer tipo de resposta gerada pelo módulo legado.

O arquivo executável pode ser visto, pela equipe de desenvolvimento, como algo desnecessário. Já foi mostrado a importância do arquivo texto e que o arquivo executável serve apenas para ler o conteúdo do arquivo texto e também para solicitar a execução das rotinas do módulo legado. Se a tarefa fosse apenas de ler o arquivo texto, isto poderia ser feito sem o uso do arquivo executável, mas se os serviços desejados existirem apenas dentro de arquivos binários, então não existirá outra alternativa: o arquivo executável terá de ser usado. Mesmo que os fontes estejam disponíveis, ainda assim, é uma boa idéia usar o arquivo executável. Do contrário, as rotinas existentes dentro do módulo C/C++ teriam de ser alteradas para que a leitura do arquivo texto seja feita diretamente dentro destas rotinas. Isto pode não parecer um grande problema, mas qualquer alteração, por menor que seja, quando é feita em um *software* em funcionamento, sempre há a possibilidade de falhas acontecerem. O uso do arquivo executável impede que tais falhas ocorram em rotinas já funcionais.

- Disponibilidade dos arquivos binários: Nesta situação, não existe outra alternativa além de usar o arquivo texto e o arquivo executável, discutido na seção anterior.

5.3.3.2. *Wrappers* para módulos Pascal

A linguagem Pascal, apresenta um problema para a conexão com o *wrapper* Java: tanto a linguagem Java como a linguagem C/C++ não se comunicam diretamente com a linguagem Pascal. Por isso, será necessário criar artifícios. A conexão entre o *wrapper* e o sistema legado, escrito em Pascal, será feita através de um módulo escrito usando o protocolo JNI; este, por sinal, escrito em C/C++. Isto ocasiona uma mistura de linguagens: onde uma estrutura de objetos distribuídos, escrita em Java, se comunicará com rotinas definidas em um módulo JNI, escrito em C/C++, para que serviços

existentes em um módulo do sistema legado, escrito em Pascal, seja executado. A figura 5.2 mostra este tipo de conexão entre as linguagens:

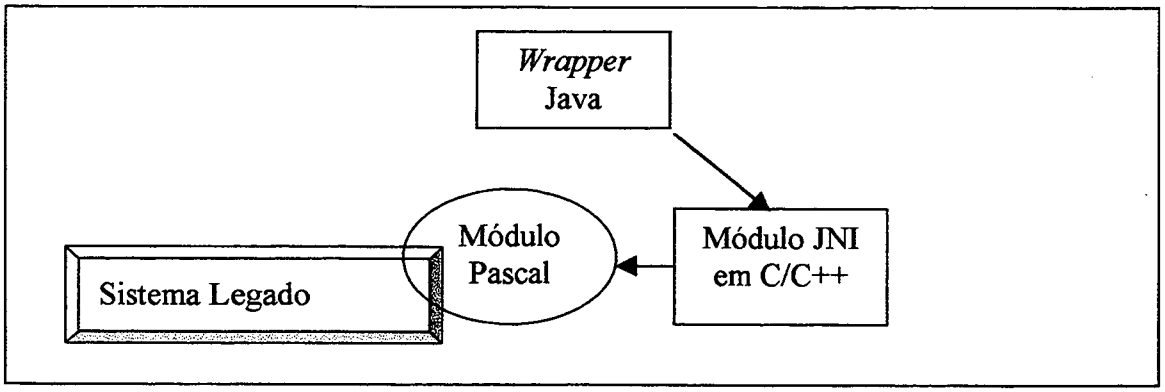

Figura 5.2 – Conexão entre o *wrapper* Java e o módulo Pascal

- Disponibilidade dos fontes Pascal: O mecanismo para resolver o problema de conexão entre o *wrapper* Java e o módulo Pascal, quando os fontes estiverem disponíveis, será o mesmo utilizado na seção 3.3.3.3.1, ou seja, criar um arquivo texto e um arquivo executável para estabelecer a ligação entre o *wrapper* Java e o módulo Pascal.

- Disponibilidade dos arquivos fontes e binários: Neste caso, deve ser usada a mesma solução já descrita na seção 3.3.3.3.1.

- Disponibilidade dos arquivos binários: O arquivos binários do Pascal, geralmente estão no formato TPU. Este é o formato usado para os módulos compilados através do compilador Turbo Pascal, e estes formatos não podem ser ligados com o Java, tampouco com o C/C++, e também não podem ser transformados em DLLs (usando o ambiente Delphi). Por isso, com todos estas negativas, se faz necessário usar a mesma solução da criação do arquivo executável e do arquivo texto, descrito na seção 3.3.3.3.1.

Cabe aqui comentar que os arquivos C/C++ podem ser ligados a arquivos no formato OBJ. Além disso, há um programa chamado "tpu2obj.exe" que promete transformar um arquivo binário, no formato TPU, para o formato OBJ. Após estudar este utilitário, que é gratuito na Internet, foi descoberto que o programa não possui nenhum tipo de suporte dado pela empresa Borland. Tampouco existe documentação completa, apenas um pequeno manual de referência. Também foi verificado que os

arquivos no formato TPU somente podem ser convertidos para OBJ se estes arquivos TPU tiverem sido gerados pelo compilador Turbo Pascal, versão 7.0. Isto limita a sua utilização, uma vez que sistemas legados mais antigos provavelmente foram compilados em versões do Turbo Pascal, anteriores à versão 7.0.

- Gerando arquivos DLL a partir do Pascal: Caso a equipe de desenvolvimento tenha a sua disposição todos os fontes na linguagem Pascal, independente do sistema legado ter sido feito para a arquitetura de 16 bits ou 32 bits, poderá ser utilizada de uma outra técnica que evitaria a utilização do arquivo texto e do arquivo executável. É importante, entretanto, que nenhuma rotina, feita por terceiros, ou mesmo pertencente à alguma biblioteca própria do compilador Pascal, sejam utilizados a não ser que os fontes destas rotinas também estejam disponíveis. Isto se deve porque a técnica envolve compilar os fontes Pascal em um formato condizente com a arquitetura de 32 bits. A presença de arquivos binários de 16 bits poderia inviabilizar a compilação dos fontes para a arquitetura de 32 bits. A técnica a ser usada compreende a geração de um arquivo no formato DLL através de um compilador como, por exemplo, o Delphi - sendo que este arquivo DLL, posteriormente, será ligado ao módulo JNI e este ligado diretamente ao *wrapper* Java.

5.3.3.3. *Wrappers* para módulos Clipper

- Disponibilidade dos fontes Clipper: O Clipper não é compatível com o Java, mas com a linguagem C/C++ (o Clipper possui um certo grau de compatibilidade). Através desta compatibilidade, é possível ao Clipper executar rotinas escritas em C/C++. Isto é possível porque o Clipper possui uma API chamada de "*Extended System*". Esta API permite uma comunicação de apenas uma direção, isto é, chamar uma rotina C/C++ de um programa Clipper, mas o inverso não é possível. Este tipo de limitação, a saber, a comunicação de apenas uma via, impede estabelecer um ligação direta entre um módulo JNI e as rotinas escritas em Clipper. A solução geral será construir um arquivo executável, em Clipper e, através de um arquivo texto, estabelecer a comunicação entre o módulo Java e o executável Clipper.

- Disponibilidade dos arquivos fontes e binários: Neste caso, deve ser usada a mesma solução já descrita na seção 3.3.3.3.1.

- Disponibilidade dos arquivos binários: Após a compilação de um módulo em Clipper, é gerado um arquivo binário no formato OBJ. O compilador C/C++ permite que arquivos neste formato sejam usados durante o processo de ligação. No entanto, o formato dos arquivos OBJ que o C/C++ utiliza é diferente dos arquivos OBJ gerados após a compilação dos fontes Clipper. Desta forma, não é possível usar diretamente arquivos OBJ nos programas escritos em C/C++. Outro detalhe que impediria a utilização dos arquivos OBJ, é o fato de o compilador Clipper Ter sido feito para ambientes de 16 bits e, consequentemente, os arquivos OBJ tem o seu código binário gerado para 16 bits (o que por si só já seria um empecilho para a utilização junto aos compiladores C/C++ de 32 bits). Por isso, a solução encontrada passa novamente pela geração do arquivo executável e pelo arquivo texto a ser usado como forma de comunicação entre o módulo JNI e o módulo Clipper.

- Gerando arquivos DLL a partir do Clipper: A mesma solução usada para módulos escritos na linguagem Pascal, no qual um compilador de 32 bits é usado para construir arquivos no formato DLL, para que, desta forma, os arquivos executáveis não mais precisem ser utilizados, não funcionam para o Clipper (mesmo que todos os fontes estivessem presentes). É possível citar dois(2) compiladores para a geração dos arquivos no formato DLL, a partir de programas Clipper: "*Blinker*" [Blinkinc] e o "*VisualObjects*" [VO]. O problema não são os compiladores, mas sim o formato que estes arquivos DLL possuem após a compilação. Estes arquivos não são compatíveis com os compiladores C/C++, por isso, a sua utilização se torna impossível.

Um problema que impede a utilização das DLLs e dos arquivos OBJs, pelos programas C/C++, é que a linguagem Clipper é fracamente tipada. Isto quer dizer que os tipos das variáveis, usadas pelos programas Clipper, somente são conhecidos durante a execução do programa. Os arquivos executáveis escritos em Clipper, bem como os arquivos DLL e OBJ, internamente, já estão preparados para este tipo de tipagem fraca, o que não ocorre para os programas C/C++. Os programas C/C++ não possuem a mesma estrutura para gerenciamento das variáveis do Clipper, e por isso não conseguem trabalhar com a tipagem fraca do Clipper. O resultado é que arquivos OBJ/DLL, mesmo

gerados através de compiladores Clipper de 32 bits, continuam se utilizando da tipagem fraca, o que impede o seu uso junto a programas C/C++. Um detalhe que cabe mencionar, é que a partir das versões Clipper 5.3 e VisualObjects, já é possível definir tipos para cada variável Clipper, como ocorre com linguagens fortemente tipadas como o Pascal ou o C/C++. No entanto, a grande maioria de sistemas legados, escritos em Clipper, foram feitos quando a linguagem Clipper não possuía a tipagem forte. É bom notar que seria possível converter os programas Clipper, com tipagem fraca, para programas contendo uma tipagem forte. O problema é que seria necessário estudar cada variável utilizada dentro dos fontes Clipper, e determinar cada tipo utilizado, e então definir estas variáveis com o tipo de dados correto. Um exemplo deste fato é uma variável que durante toda vida útil do sistema assume apenas valores numéricos inteiros, poderia ser definida como sendo deste tipo. É claro que existiriam situações onde uma mesma variável Clipper, poderia assumir diversos tipos durante a execução do sistema. Em situações assim, você teria de alterar o sistema definindo diversas variáveis diferentes, cada uma com o seu tipo de dados. Por exemplo, uma variável, digamos chamar-se "X", que recebe-se valores numéricos e caracter, poderia ser trocada por duas outras variáveis, digamos "nX" e "cX", as quais respectivamente seriam usadas para armazenar os valores numéricos e os valores caracter. Note que independente da situação você teria de alterar o programa fonte Clipper, seja para definir o tipo correto de dado das variáveis, seja para substituir variáveis para garantir a unicidade de tipos (como o exemplo das variáveis "nX" e "cX"). É importante lembrar que, sempre que um software é alterado, sempre existe a possibilidade de algum trecho de sistema legado, passar a funcionar incorretamente.

5.3.3.4. *Wrappers* para módulos de outras linguagens

O leitor já pode ter percebido que, independente da linguagem utilizada para construir o sistema legado, é possível empregar a mesma solução, ou seja, criar um arquivo executável para acessar as rotinas do módulo legado, e um arquivo texto para definir a comunicação entre o *wrapper* Java e o módulo legado. É claro que outras soluções podem ser usadas, mas isto irá depender da possibilidade da linguagem poder se comunicar com o Java ou com o módulo JNI (escrito em C/C++). Antes de sair

escrevendo qualquer linha de código é necessário determinar o grau de ligação que a linguagem do sistema legado possui com o Java e/ou com a linguagem C/C++. Depois de compreender exatamente o grau de interfaceamento que uma determinada linguagem possui, então a equipe de desenvolvimento poderá saber como estabelecer a conexão entre o módulo legado e o *wrapper*.

5.3.4. Usando tradutores de código

Existem na WEB, diversos programas que executam, com diferentes níveis de sucesso, a tradução das linhas de código de uma linguagem para outra. Por exemplo, o programa "Pas2C" [Mpsinc], o qual converte código Pascal para a linguagem C. Também é possível citar o programa "FlagShip" [Fship] que transforma linhas escritas em Clipper para a linguagem C. Apesar de existirem tais tradutores não é recomendado o seu uso. Não apenas porque não é garantido que todas as linhas serão perfeitamente traduzidas, uma vez que podem existir construções de comandos que não possuam correspondência na linguagem C/C++. Por exemplo, os comandos "SELECT", "SKIP 0", "@ GET..." entre outros da linguagem Clipper, podem não possuir uma correspondência perfeita em C/C++. Da mesma forma poderiam existir comandos de bibliotecas de terceiros que não podem ser traduzidos para o C/C++, pelo simples fato de estes mesmos comandos não estarem implementados na linguagem C/C++. Outro fator, que faz com que não seja recomendado, utilizar tais programas tradutores, é que a legibilidade, resultante após a tradução, pode não ser da melhor qualidade. Lembre que a função destes aplicativos, é de traduzir os fontes de uma linguagem para outra. Manter a legibilidade do código, após a tradução, é outra história. Por isso deve ser considerado se utilizar estes tradutores realmente irá valer a pena.

No capítulo 8 será mostrado um estudo de caso, enfocando exclusivamente o *wrapper* de código. Neste estudo de caso, será abordado a ligação do Java, com a linguagem Pascal..

6. A INTERFACE NATIVA DO JAVA

Este capítulo trata da interface nativa do Java, sem a qual a comunicação entre o Java e a linguagem C seria impossível. O *Java Native Interface* [JNI 97], consiste de uma interface que permite aos programadores Java ativar rotinas escritas originariamente em C. É, através desta interface, que as aplicações legadas podem ser acessadas a partir de uma classe Java. Tanto faz que o *software* legado tenha sido escrito usando o paradigma estruturado ou o paradigma orientado ao objeto - o JNI permitirá ao Java executar rotinas existentes dentro destes *softwares* legados.

Desde o JDK1.1, a Sun notou a real necessidade de permitir que o Java seja conectado com outras linguagens, apesar do termo "outras linguagens" não ser bem apropriado, uma vez que o JNI permite a conexão do Java apenas com a linguagem C. Para que o Java se "conecte" com linguagens diferentes do C, é necessário que seja criado antes um programa em C.

O JNI deve ser usado sempre que a performance de determinadas partes de uma aplicação Java seje crucial ou quando a linguagem Java não possuir algum tipo de recurso (por exemplo, ativar interrupções de *hardware*), ou então, quando for necessário criar uma interface entre o Java e uma aplicação legada.

O Java roda em cima da sua máquina virtual (JVM), o que garante a independência dos aplicativos Java das características próprias do *hardware* ou do sistema operacional. Com isso, é possível ter classes escritas em Java independentes de plataforma (Gordon, 1998).

O problema com o JVM é que ele não foi feito para permitir uma excelente performance uma vez que, para isso, é necessário que o código use de características próprias do *hardware*.

6.1. Construindo um programa usando JNI

O programa a ser usado para ilustrar o uso do JNI mostrará na tela a mensagem "Alô mundo". Um exemplo tão simples, como a mera exibição de um texto na tela, mesmo sendo uma tarefa trivial em Java, não é tão simples em JNI. Uma vez que é necessário, além de uma classe Java, um programa escrito na linguagem C. Para este exemplo, para a exibição da mensagem "Alô Mundo" na tela, será construído um programa em C, que fará a exibição da mensagem, e uma classe em Java, que executará este programa.

Para que o código escrito em C seja executado a partir de uma classe Java, é necessário que alguns passos sejam seguidos, os quais são:

- 1° Passo: declarar o método nativo em Java: o passo inicial para qualquer interface entre o Java e o C, é a construção de uma classe, contendo um ou mais métodos abstratos. Estes métodos abstratos são conhecidos como métodos nativos, uma vez que a sua real implementação não será feita em Java, mas através de código em C. Em outras palavras, todo método nativo é um método abstrato Java, cuja implementação é feita em C.

Para a definição deste método abstrato é necessário a utilização da palavra reservada *native*, existente na linguagem Java. Por exemplo, a seguinte linha de código será usada para a definição do método "imprime()", o qual será responsável pela impressão, na tela, da mensagem "Alô Mundo".

```
private native void imprime();
```

O método "imprime" foi definido como privativo, apenas porque, neste exemplo, não existe a necessidade de outro tipo de visibilidade, como protegida ou pública, embora os métodos nativos possam ter qualquer um dos tipos de visibilidade normalmente usados na definição de métodos Java. Outro ponto de interesse é a

obrigatoriedade do uso da palavra reservada *native*. Também é necessário que os métodos nativos sejam sempre terminados com um ponto-e-vírgula (;).

Uma vez que a real implementação do método nativo é feita em C, o código para ser executado, a partir de uma classe Java, deve ser colocado dentro de um arquivo de ligação dinâmica, e este arquivo deverá ser ligado ao Java. Esta ligação é feita através da linha de comando mostrada a seguir:

```
static { System.loadLibrary( "AloMundo" ); }
```

No código acima, será carregado o arquivo de ligação dinâmica de nome "AloMundo.dll". De acordo (Gordon, 1998) a chamada para "System.loadLibrary()" deve ser colocada dentro de um bloco *static*, para que seja assegurado a carga da biblioteca. Esta carga será feita obrigatoriamente na inicialização da classe. A figura 6.1 ilustra a classe Java com a definição do método nativo é mostrado a seguir:

```
class AloMundo {
  private native void imprime();
  public static void main( String[] args ) {
  new AloMundo().imprime();
  }
static { System.loadLibrary( "AloMundo" ); }
}
```

Figura 6.1 – Classe Java com a definição método nativo "AloMundo"

- 2° Passo: compilar a classe Java: a compilação da classe Java é feita de forma absolutamente normal, usando o compilador "javac". Portanto, salve o arquivo AloMundo.java e faça a compilação do mesmo.

- 3° Passo: criar o arquivo de *header* da função nativa na linguagem C: para usar o JNI, é necessário escrever rotinas em C, mas antes disto, é necessário criar um arquivo de *header,* para que a perfeita ligação entre o Java e o C possa ser obtida. O JNI possui definições que devem ser usadas para que os tipos de dados existentes na linguagem Java possam ser mapeados para os tipos de dados da linguagem C. Estas definições podem parecer um tanto complexas de serem entendidas e escritas, à primeira vista, mas com tempo e prática, elas irão ser gradativamente compreendidas. Um ponto favorável é que o JDK possui uma ferramenta que facilita a construção dos arquivos de *header*. Através desta ferramenta, os arquivos de *header* podem ser automaticamente construídos. Esta ferramenta é denominada de "javah" e deve receber, como entrada, o

arquivo compilado "AloMundo.class". Sua utilização é feita mediante a execução do seguinte comando:

javah –jni AloMundo

Após o uso da ferramenta "javah", será criado, automaticamente, o arquivo de *header* chamado "AloMundo.h":

JNIEXPORT void JNICALL Java_AloMundo_imprime (JNIEnv *, jobject);

No arquivo "AloMundo.h", as palavras "JNIEXPORT" e "JNICALL" definem as convenções de chamada entre o Java e o C e são geradas automaticamente pela ferramenta "javah". Caso o leitor deseje, poderá procurar maiores informações em (Gordon, 1998) e (Liang, 1999). Por outro lado, os parâmetros "JNIEnv" e "jobject" não fazem parte da definição do método nativo na classe Java e sempre estarão presentes em qualquer método nativo, tendo o método nativo parâmetros ou não. Os parâmetros tem o seguinte significado:

a) "JNIEnv *" => corresponde a um ponteiro para a instância do JVM que está executando a classe Java;

b) "jobject" => corresponde a instância da classe Java propriamente dita.

Outro detalhe diz respeito ao nome do método, gerado pela ferramenta "javah", para o arquivo de *header*. O método na classe Java tem o nome de "imprime()", mas no arquivo de *header*, o nome do método passou a ser "Java_AloMundo_imprime". Isto é um padrão usado pela ferramenta "javah". O nome de qualquer método existente em uma classe Java, que for gerado automaticamente pela ferramenta "javah" para formar o arquivo de *header*, terá o seu nome sempre modificado de tal forma que o novo nome, no arquivo de *header*, será formado pela palavra "Java" concatenada com o nome da classe onde o método nativo foi definido e, finalmente, concatenado com o nome do método nativo propriamente dito.

- 4° Passo: escrever a implementação da função nativa: Uma vez que o arquivo de *header* tenha sido construído, já pode ser iniciado a escrita do método nativo em C. É importante notar que o cabeçalho da função nativa deve coincidir perfeitamente com a definição existente no arquivo de *header*.

- 5° Passo: compilar o fonte C e criar o arquivo DLL: O próximo passo compreende a criação da biblioteca de ligação dinâmica. O comando necessário para a criação deste arquivo pode ser dado na linha de comando, apesar de ser meio longo. O comando é mostrado em seguida:

cl AloMundo.c -FeAloMundo.dll -MD -LD -Ic:\vc98\include -Ic:\jdk1.2.1\include -Ic:\jdk1.2.1\include\win32 -link c:\vc98\lib\msvcrt.lib c:\vc98\lib\oldnames.lib c:\vc98\lib\kernel32.lib

- 6° Passo: rodar o programa em Java: A parte final consiste, em simplesmente executar o programa Java.

6.2 Mapeamento dos tipos de dados Java para C

Para a realização do mapeamento entre o Java e o C, é necessário verificar se o tipo a ser mapeado são primitivos ou referencias, uma vez que JNI trata estes tipos de forma diferente. Quando for tipos primitivos, o mapeamento é mais simples, uma vez que o JNI possui quase que uma correspondência direta entre o tipo Java e o correspondente tipo em C. Mas quando o assunto for o mapeamento de referências, o mapeamento deverá ser feito de forma mais cuidadosa. Para maiores informações consultar [Liang 1999] e [Gordon 1998].

6.2.1 Mapeamento de tipos primitivos do Java

Os tipos primitivos do Java possuem uma correspondência quase que direta para os tipos existentes no C. O JNI possui um arquivo, denominado "jni.h", o qual define todos os tipos que devem ser usados neste mapeamento.

6.2.1.1 Retornando valores do C

Para ilustrar, a figura 6.2 mostra uma classe Java, onde é definido um método nativo, indicando que um valor do tipo inteiro, será retornado do C.

```
class Exemplo1 {
private native int getValorInt();
public static void main( String[] args ) {
System.out.println( new Exemplo1().getValorInt() );
}
static { System.loadLibrary( "Exemplo1" ); }
}
```

Figura 6.2 – Classe Java com retorno de valor do tipo inteiro

Após a criação da classe, mostrada na figura 6.2, é necessário criar o arquivo de *header*, e o programa fonte em C. A listagem do programa fonte em C, é mostrado na figura 6.3.

```
#include <stdio.h>
#include "Exemplo1.h"
JNIEXPORT jint JNICALL Java_Exemplo1_getValorInt (JNIEnv *env, jobject thisObj){
  return( 1234 );
}
```

Figura 6.3 – Retorno de valor do tipo inteiro do C

O próximo passo será construir a biblioteca, isto é, o arquivo de ligação dinâmica, e executar o programa Java.

6.2.1.2 Enviando valores para o C

No próximo exemplo, será enviado um número, de uma classe Java para um método nativo implementado em C. A rotina, escrita na linguagem C, irá mostra na tela este valor. Para este exemplo será usado um parâmetro. A figura 6.4 ilustra a classe Java, e o método nativo que envia parâmetros para o programa em C.

```
class Exemplo2 {
private native void imprime( int valor );
```

```
public static void main( String[] args ) {
int x = 666;
new Exemplo2().imprime( x );
}
static { System.loadLibrary( "Exemplo2" ); }
}
```

Figura 6.4 – Classe Java com envio de parâmetros

Na figura 6.4, o exemplo mostra que será enviado, como parâmetro, para o método nativo um número inteiro. O método nativo terá, por sua vez, como função mostrar na tela este número.

A figura 6.5, mostra a listagem do programa C, que implementa o método nativo "imprime()" definido na classe Java.

```
#include <stdio.h>
#include "Exemplo2.h"
JNIEXPORT void JNICALL Java_Exemplo2_imprime(JNIEnv *env, jobject thisObj,
jint valor){
  printf( "%d", valor );
  return;
}
```

Figura 6.5 – Método nativo em C recebendo parâmetro

6.2.2 Mapeamento de tipos referência do Java

O leitor, acostumado a trabalhar com a linguagem Java, sabe que além dos tipos primitivos, existem as referências, que correspondem aos tipos *String*, *Array*, e outros derivados direta, ou indiretamente de *Object*. Estes tipos tem um forma de mapeamento com o C, mas o JNI não fornece um mapeamento tão direto, como ocorre com os tipos primitivos. Quando o assunto envolver tipos referência, será necessário utilizar de rotinas próprias do JNI, para permitir que o C possa acessar valores dos tipos de referência do Java. Lembrando que uma referência nada mais é que a instância de um objeto, isto significa que este objeto ocupa um espaço de memória, gerenciado pelo Java, onde tal espaço de memória não é acessível diretamente pelos programas em C. Para permitir que os valores existentes no espaço de endereçamento de memória do Java possam ser acessados por um programa escrito em C, foram criadas rotinas explícitas

que recuperam os valores da "memória" do Java e os tornam disponíveis para uso pelos programas em C.

6.2.2.1 Mapeamento de *strings* Java

O tipo *String* do Java é, na verdade, uma referência a um objeto que representa uma seqüência de caracteres. Por ser uma instância, uma *String* não pode ser usada diretamente dentro de um programa C, por dois motivos: o primeiro é que um programa C não tem acesso direto ao espaço de memória gerenciado pelo Java; o segundo é que uma *String* Java, internamente, não tem necessariamente a mesma representação de uma seqüência de caracteres C. Por estes dois motivos, para que um programa em C possa usar facilmente um valor do tipo *String*, é necessário utilizar algumas funções definidas dentro do JNI, que permitem o correto uso de referências aos objetos *String* existentes no Java. Isto significa que existe a necessidade de, dentro de um programa em C, termos acesso a um conjunto de rotinas que foram definidas dentro do JNI. É neste momento, que será usado o parâmetro "JNIEnv".

O parâmetro "JNIEnv", é usado para permitir que um programa escrito em C, possa usar de rotinas definidas dentro do JNI. Segundo Gordon (1998), o parâmetro "JNIEnv" é um ponteiro para outro ponteiro, o qual por sua vez aponta para uma tabela de ponteiros de funções. Cada um destes ponteiros de funções corresponde a uma função JNI. É através desta tabela de ponteiros que é possível ao C acessar informações definidas dentro do espaço de endereçamento existente no Java.

A seguir será mostrado exemplos de métodos nativos que enviam valores *String* do Java para o C, e vice-versa, quando uma seqüência de caracteres do C é retornada para o Java.

No figura 6.6, é mostrado como é possível enviar um valor do tipo *String* do Java para C. Uma vez dentro da rotina escrita na linguagem C, será feito a exibição na tela do conteúdo do parâmetro *String*, recebido como parâmetro do Java:

```
class Exemplo3 {
private native void imprime( String s );
public static void main( String[] args ) {
```

```
String x = "Clayton Bonelli";
new Exemplo3().imprime( x );
}
static { System.loadLibrary( "Exemplo3" ); }
}
```

Figura 6.6 – Método nativo, enviando parâmetro *String*

O código em C, que é responsável pela impressão do parâmetro na tela, é mostrado na figura 6.7:

```
#include <stdio.h>
#include "Exemplo3.h"
JNIEXPORT void JNICALL Java_Exemplo3_imprime(JNIEnv *env, jobject thisObj,
jstring s){
  char buf[ 128 ];
  int tam;
  tam = (*env)->GetStringLength( env, s );
  (*env)->GetStringUTFRegion( env, s, 0, tam, buf );
  printf( "%s", buf );
  return;
}
```

Figura 6.7 – Recebendo parâmetro *String*

O leitor poderá notar que foram usadas duas funções definidas dentro do JNI. Estas funções são necessárias para copiar os caracteres existentes dentro da *String* passada pelo Java para uma forma de caracteres aceita pelo C. Um detalhe que deve ser mencionado e que foi citado por Liang (1999) é que o Java utiliza *Strings* no formato Unicode, enquanto que o C utiliza *Strings* no formato UTF-8. Por esta razão, é necessário alguma forma de traduzir o formato Unicode para UTF-8. Esta tarefa é feita pela função do JNI chamada de "GetStringUTFRegion". Para que esta função JNI funcione corretamente, é necessário determinar, de antemão, a quantidade total de caracteres existentes na *String*, passada pelo Java. Para que esta informação seja corretamente obtida, é usado uma outra rotina, a qual retornará à quantidade de caracteres exata que a *String* Java possui. Esta rotina é chamada de "GetStringLength"

Da mesma forma como foi mostrado anteriormente, para que uma *String* seja retornada para o Java, é necessário que funções especiais do JNI sejam utilizadas. Na figura 6.8 é mostrado a classe Java, a ser usada para definir o método nativo:

```
class Exemplo4 {
private native String getString();
public static void main( String[] args ) {
System.out.println( new Exemplo4().getString() );
}
static { System.loadLibrary( "Exemplo4" ); }
}
```

Figura 6.8 – Recebendo *String* do C

Será necessário, dentro do programa C, usar algumas rotinas especiais para converter *Strings* no formato C (UTF-8), para o formato do Java (Unicode). O programa em C, contendo as rotinas que fazem esta conversão são mostradas na listagem 6.9:

```
#include <stdio.h>
#include "Exemplo4.h"
JNIEXPORT jstring JNICALL Java_Exemplo4_getString(JNIEnv *env, jobject
thisObj){
  char buf[ 128 ];
  strcpy( buf, "Clayton Bonelli" );
  return (*env)->NewStringUTF( env, buf );
}
```

Figura 6.9 – Retornando String para o Java

No código da figura 6.9, é utilizado a função do JNI chamada de "NewStringUTF", a qual é responsável pela criação de uma "String" no formato Java. A sua função é transformar uma "String" UTF-8 para o formato Unicode.

6.2.2.2 Mapeamento de *arrays*

O JNI trata os *Arrays* de tipos primitivos, forma diferente dos *Arrays* de objetos. Os *Arrays* de tipos primitivos são aqueles *arrays* formados por valores dos tipos

primitivos do Java, como *int*, *char*. *float*, etc. Os *Arrays* de objetos são aqueles cujos valores são referências para qualquer instância, direta ou indireta, da classe Java *Object*.

Os *Arrays* de tipo primitivos, não tem uma forma direta de acesso, dentro de um programa C, como ocorre com os tipos primitivos. Ou seja, um valor *int* no Java possui o correspondente *jint* no C e o seu manuseio dentro de um programa em C ocorre exatamente da mesma forma como ocorre com um valor do tipo *int* da linguagem C. Os *Arrays* de tipos primitivos, para serem acessados dentro de um programa em C, devem se utilizar de funções especiais existentes no JNI, para que os valores possam ser manuseados. Da mesma forma, como os valores do tipo *String* são acessados nos programas C, através de rotinas como "GetStringUTFRegion", os *Arrays* também possuem rotinas definidas especificamente para manusear seus elementos.

Na figura 6.10 é mostrado a listagem de uma classe em Java, que monta um *Array* de valores inteiros e, através de um método nativo, envia este *Array*, para um programa em C:

```
class Exemplo5 {
private native int getSoma( int[] vetor );
public static void main( String[] args ) {
int[] v = new int[ 5 ];
v[ 0 ] = 10;
v[ 1 ] = 20;
v[ 2 ] = 30;
v[ 3 ] = 40;
v[ 4 ] = 50;
System.out.println( new Exemplo5().getSoma( v ) );
}
static { System.loadLibrary( "Exemplo5" ); }
}
```

Figura 6.10 – Envio de um Array para um programa C

Na figura 6.11, é mostrado o programa em C que faz o acesso aos elementos do Array, definido na classe Java, mostrada na figura 6.10:

```
#include <stdio.h>
#include "Exemplo5.h"
JNIEXPORT jint JNICALL Java_Exemplo5_getSoma(JNIEnv *env, jobject thisObj,
jintArray vet){
```

```
jint *buf, soma = 0, i, tam;
buf = (*env)->GetIntArrayElements( env, vet, NULL );
if ( buf == NULL ){
  return 0;
}
tam = (*env)->GetArrayLength( env, vet );
for ( i = 0; i < tam; i++ ){
  soma = soma + buf[ i ];
}
(*env)->ReleaseIntArrayElements( env, vet, buf, 0 );
return soma;
}
```

Figura 6.11 – Programa em C para acessar os elementos de um *Array*

No código da figura 6.11, existem duas funções para manipular o *Array* de inteiros. A primeira destas funções, é chamada de "GetIntArrayElements" e tem a função de obter um *pointer* para o vetor de números inteiros definidos no Java. A próxima função, "ReleaseIntArrayElements", tem a tarefa de liberar o espaço de memória alocado, dentro do espaço de endereçamento do C pela função "GetIntArrayElements". Note que a memória que for liberada não afeta o espaço de memória do Java.

Os tipos referência, diferentemente dos arrays de tipos primitivos, possuem como valores, referências a objetos. Por objetos, entendemos os valores do tipos *String*, *Array* e toda instância derivada, direta ou indiretamente da classe Java *Object*. Por este motivo, da mesma forma como os *Arrays* de tipos primitivos possuem funções JNI, definidas especificamente para acessar os valores existentes em cada posição de um vetor, os *Arrays* de objetos (ou tipos referência) possuem funções feitas exclusivamente para que os métodos nativos, escritos na linguagem C, possam acessar os objetos Java. Caso o leitor deseje obter informações mais detalhadas, consultar Liang (1999) e Gordon (1998).

Na figura 6.12, será criado um vetor no Java contendo *Strings* e este vetor será enviado para um método nativo, para que o conteúdo deste vetor seja exibido na tela. Esta exibição será feita dentro do método nativo.

```
class Exemplo6 {
private native void imprime( String[] vetor );
public static void main( String[] args ) {
String[] v = new String[ 4 ];
v[ 0 ] = "isto";
v[ 1 ] = "eh";
v[ 2 ] = "um";
v[ 3 ] = "exemplo";
new Exemplo6().imprime( v );
}
static { System.loadLibrary( "Exemplo6" ); }
}
```

Figura 6.12 – Definição de um *Array* de referências para *Strings*

A figura 6.13, ilustra a implementação do método nativo, na linguagem C, o qual recebendo o vetor como parâmetro, exibirá o seu conteúdo na tela.

```
#include <stdio.h>
#include "Exemplo6.h"
JNIEXPORT void JNICALL Java_Exemplo6_imprime(JNIEnv *env, jobject thisObj,
jobjectArray vet){
  char buf[ 128 ];
  jint tam, tot, i;
  jstring s;
  tot = (*env)->GetArrayLength( env, vet );
  for ( i = 0; i < tot; i++ ){
    s = (jstring) (*env)->GetObjectArrayElement( env, vet, i );
    tam = (*env)->GetStringLength( env, s );
    (*env)->GetStringUTFRegion( env, s, 0, tam, buf );
    printf( "%s\n", buf );
    (*env)->DeleteLocalRef( env, s );
  }
  return;
}
```

Figura 6.13 – Método nativo para *Arrays* de referências

6.2.2.3 Mapeamento de objetos

Escrever sobre "objetos" pode parecer um tanto redundante, uma vez que instâncias da classe *String*, bem como as instâncias de *Array*, são todos objetos. Mas

esta seção é dedicada a uma outra questão. Caso, em um programa Java, seja criada uma instância de uma classe qualquer, como fazer para acessar os atributos e métodos existentes neste objeto, de dentro de um programa C? O objetivo desta seção é responder a esta questão.

De acordo com os conceitos da teoria da orientação ao objeto, uma classe possui atributos de classe e atributos de instância. De acordo com Amber (1998): "Os Atributos de Instância, geralmente chamados apenas de atributos, são grupos de informações aplicáveis a um único objeto (instância). Os Atributos de Classe, por outro lado, são grupos de informações aplicáveis às classes como um todo." Com este conceito em mente e percebendo que os atributos possuem diferenças entre si, já é possível imaginar que o JNI, para permitir que os programas C acessem os atributos de um objeto qualquer, possui um conjunto específico de funções JNI, construídas para permitir o acesso aos atributos de classe e um outro conjunto de funções JNI, que permitem acessar os atributos de instância.

Para ilustrar a forma pela qual um atributo de instância pode ser acessado, de dentro de um método nativo em C, será usado a classe definida na listagem mostrada na figura 6.14:

```
class UmaClasse{
public int numero = 666;
}
class Exemplo7 {
private native void imprime( UmaClasse obj );
public static void main( String[] args ) {
new Exemplo7().imprime( new UmaClasse() );
}
static { System.loadLibrary( "Exemplo7" ); }
}
```

Figura 6.14 – Classe para acesso de atributo de instância

No exemplo ilustrado pela figura 6.14, foi criado uma classe cujo nome é "UmaClasse", na qual se define um atributo de instância chamado "numero". Quando o programa Java for iniciado, será criado uma instância da classe "UmaClasse" e esta instância será enviada como parâmetro para o método nativo chamado "imprime()". A implementação do método nativo é ilustrada pela figura 6.15:

```
#include <stdio.h>
#include "Exemplo7.h"
JNIEXPORT void JNICALL Java_Exemplo7_imprime(JNIEnv *env, jobject thisObj,
jobject obj){
 jfieldID atributo;
 jint numero;
 jclass umaClasse;
 umaClasse = (*env)->GetObjectClass( env, obj );
 atributo = (*env)->GetFieldID( env, umaClasse, "numero", "I" );
 if ( atributo == NULL ){
  return;
 }
 numero = (*env)->GetIntField( env, obj, atributo );
 printf( "%d", numero );
 return;
}
```

Figura 6.15 – Método nativo para acesso ao atributo de instância

Note que na listagem da figura 6.15, há dois(2) parâmetros do tipo "jobject" os quais são "ThisObj" e "obj". Estes parâmetros, respectivamente, representam a instância da classe "Exemplo7", que é o programa Java que está em execução, e o último parâmetro, que representa uma instância da classe "UmaClasse". O "jobject" é um tipo que permite, aos programas C, acessar qualquer tipo de instância de objetos definida pelo Java. Para acessar o conteúdo do atributo de instância "numero", é necessário determinar a classe do objeto, que chegou ao método nativo "imprime()", através do parâmetro "jobject". O JNI define uma rotina que executa a tarefa de determinar a classe do parâmetro "jobject". Esta rotina é chamada de "GetObjectClass".

Após determinar a classe do objeto, é necessário obter o identificador do atributo desejado. Este "identificador" é necessário para que o conteúdo do atributo possa ser "capturado" pelo programa em C. Esta tarefa é feita por uma rotina do JNI chamada de "GetFieldID". Feito isto, para se obter uma cópia do conteúdo do atributo de instância, bastará utilizar uma outra função JNI chamada de "GetIntField".

A forma de acesso aos atributos de classe é similar ao acesso feito aos atributos de instância, na qual rotinas específicas definidas pelo JNI foram elaboradas para acesso

aos conteúdos dos atributos. O programa mostrado na listagem 6.16 mostra a definição do atributo de classe:

```
class UmaClasse{
public static int numero = 666;
}
class Exemplo8 {
private native void imprime( UmaClasse obj );
public static void main( String[] args ) {
new Exemplo8().imprime( new UmaClasse() );
}
static { System.loadLibrary( "Exemplo8" ); }
}
```

Figura 6.16 – definição do atributo de classe

Os métodos para acesso aos atributos de classe são os mesmos dos que já apresentados para acesso aos atributos de instância. A única diferença é que o nome do método, para acesso aos atributos de classe, deverá conter a palavra "Static". Por isso, para não repetir as explicações dadas anteriormente, a figura 6.17 mostra a implementação do método nativo na linguagem C. O leitor facilmente entenderá o seu funcionamento, uma vez que a explicação destas rotinas é análoga, àquela apresentada para as rotina que fazem o acesso aos atributos de instância.

```
#include <stdio.h>
#include "Exemplo8.h"
JNIEXPORT void JNICALL Java_Exemplo8_imprime(JNIEnv *env, jobject thisObj,
jobject obj){
 jfieldID atributo;
 jint numero;
 jclass umaClasse;
 umaClasse = (*env)->GetObjectClass( env, obj );
 atributo = (*env)->GetStaticFieldID( env, umaClasse, "numero", "I" );
 if ( atributo == NULL ){
  return;
 }
 numero = (*env)->GetStaticIntField( env, obj, atributo );
 printf( "%d", numero );
 return;
}
```

Figura 6.17 – Método nativo para acesso ao atributo de classe

Da mesma forma que os atributos, os métodos também são passíveis de serem acessados de dentro de um método nativo implementado através de um programa escrito na linguagem C. A forma de acesso aos métodos também possui a mesma distinção que os atributos. Ou seja, existem os métodos de classe e existem os métodos de instância que, segundo Ambler (1998): "Os Métodos de Instância são aplicáveis a objetos individuais, modificando e acessando os atributos de instâncias. Os Métodos de Classe, como você deve imaginar, são aplicáveis às classes."

A figura 6.18, ilustra a classe Java, onde é definido um método de instância, o qual será ativado de dentro de um programa nativo, escrito em C:

```
class UmaClasse{
public void imprime(){
System.out.println( "Clayton Bonelli" );
}
}
class Exemplo9 {
private native void executa( UmaClasse obj );
public static void main( String[] args ) {
new Exemplo9().executa( new UmaClasse() );
}
static { System.loadLibrary( "Exemplo9" ); }
}
```

Figura 6.18 – Classe Java com método de instância

A figura 6.19, ilustra a implementação do método nativo em C. Nesta listagem, as rotinas principais, são denominadas de "GetMethodID" e "CallVoidMethod", que possuem, respectivamente, a função de obter uma referência ao método Java e de solicitar a execução do método Java.

```
#include <stdio.h>
#include "Exemplo9.h"
JNIEXPORT void JNICALL Java_Exemplo9_executa(JNIEnv *env, jobject thisObj,
jobject obj){
```

```
    jmethodID metodo;
    jclass umaClasse;
    umaClasse = (*env)->GetObjectClass( env, obj );
    metodo = (*env)->GetMethodID( env, umaClasse, "imprime", "()V" );
    if ( metodo == NULL ){
      return;
    }
    (*env)->CallVoidMethod( env, obj, metodo );
    return;
}
```

Figura 6.19 – Método nativo para execução de método de instância

Ativar um método de classe, é muito similar à execução de métodos de instância, já que ambos necessitam de rotinas JNI específicas. As rotinas que permitem a execução dos métodos de classe possuem, praticamente, o mesmo nome das rotinas usadas para a execução dos métodos de instância. A única diferença entre uma rotina e outra é que as rotinas que permitem acesso aos métodos de classe possuem a palavra "Static" no seu nome. Por causa desta similaridade, não será mostrado nenhum programa-exemplo a respeito de como ativar métodos de classe.

Nos capítulos 7 e 8, serão mostrados exemplos da utilização do JNI com a linguagem Java e C, para estabelecer a conexão entre o Java e a linguagem Pascal, e o Java com a linguagem Clipper.

7. ESTUDO DE CASO: SIMULANDO AS TECLAS

7.1 O problema

Uma das formas que podemos utilizar para criar um *wrapper* para um sistema legado se dá através da simulação do pressionamento de teclas. Uma grande vantagem desta técnica é a maior velocidade de conversão do sistema legado, uma vez que na prática nenhuma conversão de código-fonte ocorre. O que acontece é que o *wrapper* envia os comandos que farão um determinado serviço, que já é oferecido pelo sistema legado, ser executado. Note, que o *software* legado continuará em funcionamento, realizando todas as tarefas as quais ele já executava. A grande diferença é que estes mesmos serviços serão solicitados por um cliente remoto. O *wrapper* enviará os códigos das teclas que são necessárias para que o *software* execute o serviço solicitado. Com isso, a equipe apenas terá de determinar qual a seqüência de teclas que será necessária para a perfeita execução do serviço. Assim, não existirá a necessidade de um maior entendimento, por parte da equipe, de como as sub-rotinas legadas funcionam, tampouco como o sistema de arquivos deve trabalhar. Toda a lógica de negócio codificada dentro do aplicativo legado continuará sendo executada pelo próprio sistema legado, sem que a equipe precise se preocupar em estudar, muitas vezes, centenas de milhares de linhas de código para permitir a integração/conversão do software legado com o *wrapper*.

A técnica também é conhecida como emulação de terminal nos sistemas multi usuários. Infelizmente, nem todos os *softwares* legados foram feitos para o ambiente multi usuário. Uma grande quantidade de *softwares* legados foram escritos para funcionarem de forma mono-usuário ou então para rodarem junto de uma rede. Estes

sistemas estão hoje em dia em pleno funcionamento. Dentre estes sistemas legados temos, principalmente aqui no Brasil, uma grande parcela de *softwares* escritos para o sistema operacional DOS. Este tipo de sistema legado tem um desafio maior para a equipe que irá construir o *wrapper*, pois a emulação de terminal que poderia ser uma tarefa relativamente simples para alguns sistemas operacionais multi usuários (tipo Unix), se mostra um problema quando o sistema operacional é mono-usuário. Lembramos que nos sistemas operacionais multi usuários os *softwares* já foram feitos para operar junto a terminais (o próprio sistema operacional já trabalha com terminais). É por isso que a criação de um emulador de terminais se torna um problema para os ambientes mono-usuários ou mesmo para os sistemas legados que rodam em rede, pois tais ambientes não foram feitos para serem usados com terminais. Mesmo que no DOS um *software* possa permitir o uso de terminais, este é um caso especial, pois, neste caso, o sistema legado teria de ter sido construído para permitir tal característica. Mas este tipo de *software* não é uma regra e, portanto, não será discutido neste trabalho.

7.2. O *software* para estudo

Neste trabalho, vamos usar um *software* que foi construído especialmente para demonstrar a técnica. Por isso, o *software* não tem necessariamente uma função, no sentido em que ele resolve qualquer problema. O sistema apenas apresenta alguns menus, campos de edição e mensagens. Muitos dos menus nem mesmo realizam qualquer tarefa, pois como já foi mencionado, o objetivo é mostrar a técnica que empregaremos para a simulação de digitação do teclado. Para facilitar o entendimento por parte do leitor, foi construído um sistema bem simples, com poucas linhas de código, utilizando a linguagem Clipper versão 5.2d, pertencente a empresa Computer Associates. Note, entretanto, que não é objetivo deste trabalho ensinar a linguagem, os comandos, ou o ambiente de compilação / execução da linguagem Clipper.

O programa exemplo, o qual corresponde a um hipotético sistema legado e que será usado para descrever a técnica para a simulação do pressionamento de teclas, foi escrito na linguagem Clipper e a sua listagem se encontra no anexo 1.

Para dar ao leitor uma idéia melhor do software, abaixo segue a imagem capturada da tela do sistema legado, na qual um usuário normalmente entraria para realizar a inclusão das informações pessoais de um indivíduo qualquer. Tais informações correspondem ao nome, idade e endereço de uma pessoa, e normalmente, o usuário deste *software* legado usaria uma determinada seqüência de opções de menu para chegar até o ponto onde tais informações pessoais poderiam ser digitadas. A figura 7.1 mostra a imagem da tela do software legado hipotético:

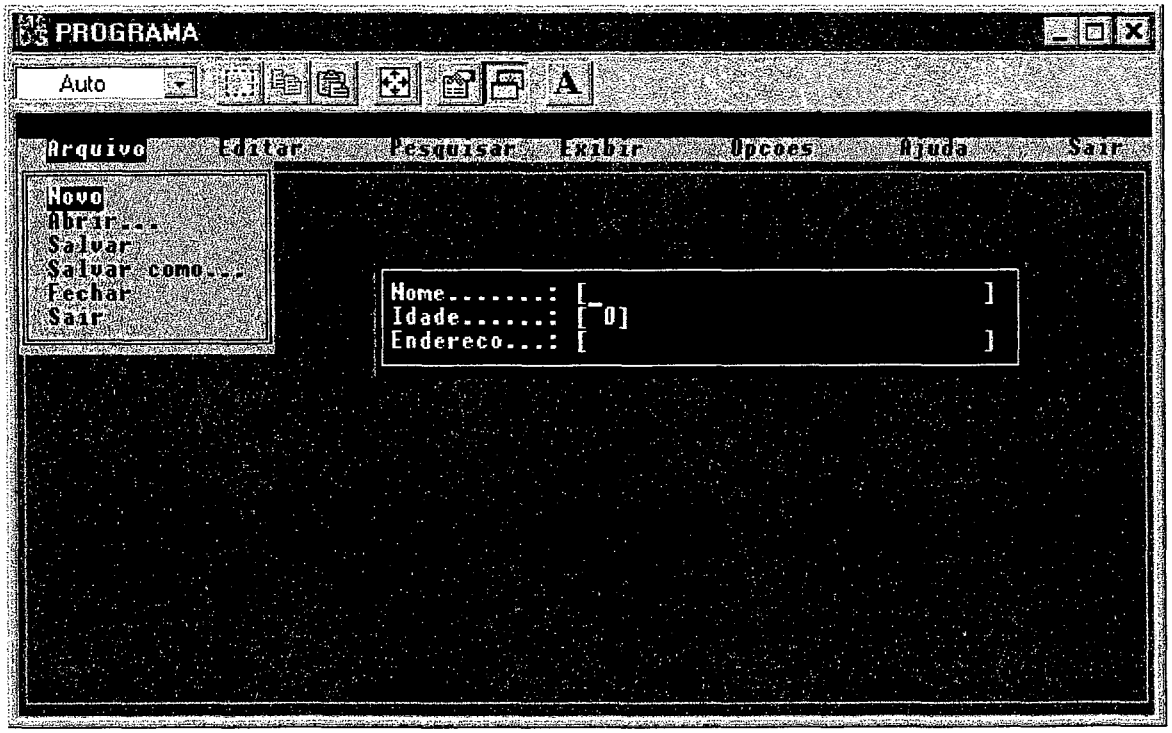

Figura 7.1 – Exemplo da tela de um hipotético sistema legado

Pela imagem mostrada na figura 7.1, pode ser visto que o usuário deste hipotético sistema legado, para conseguir entrar na janela de digitação de dados pessoais, teria de passar pelas opções de menu: "Arquivo | Novo". Após as informações terem sido digitadas, tais dados seriam incluídos em um hipotético banco de dados, o qual para este trabalho não será explicado por não comprometer a simulação.

7.3 Vantagens e desvantagens da utilização

A idéia principal por detrás da técnica de emulação de terminal, ou a simulação do Pressionamento das teclas, é obter a mesma funcionalidade de um determinado serviço

como se um usuário estivesse em frente a um teclado, operando o sistema normalmente com a digitação das informações necessárias para a concretização de um serviço qualquer. Uma vantagem da utilização desta técnica, é que do sistema legado, não é necessário a existência do código-fonte do *software*. Isto significa que a utilização desta técnica pode ser feita independente dos fontes estarem disponíveis ou não. Outro ponto favorável é que a equipe, responsável pela integração dos sistema legado com o *wrapper*, não precisará estudar a funcionalidade do *software* ou os seus fontes, nem mesmo as regras de atualização do banco de dados. Isto porque todo o trabalho será feito pelo sistema legado que ainda funciona perfeitamente. A equipe apenas terá que determinar qual a seqüência de teclas devem ser pressionadas para que o serviço solicitado pelo *wrapper* seja obtido. Com isto, a equipe ganhará um tempo muito grande, uma vez que não será necessário nenhum tipo de conversão, ou construção de *wrapper* auxiliares, ou de uso de outros ambientes, por exemplo Delphi ou Visual Objects, como já foram vistos em outras técnicas.

No que se refere às desvantagens, o uso da emulação de terminal, deixa a equipe limitada a apenas os serviços que o sistema legado já oferece. Qualquer ampliação ou mudança da funcionalidade não poderá ser obtida, a não ser que a mesma seja desenvolvida, isto é, programada no sistema legado ou então no *wrapper*. Mas, para que isto ocorra, será necessário que a equipe estude os fontes e/ou sistemas de arquivos do *software* legado para conseguir os resultados desejados. Infelizmente, quanto maior o sistema legado, maior será o tempo, e também os custos, necessários para implementar a nova funcionalidade. Este estudo não apenas deve se preocupar com o novo serviço, mas deve considerar o impacto que esta nova funcionalidade acarretará nos serviços que atualmente o sistema legado já possui. Sem estas considerações, ou melhor, sem este estudo detalhado, é bem provável que a(s) funcionalidade(s) possam gerar algum tipo de inconsistência no *software* legado, muitas vezes, em serviços já existentes. Dependendo, será necessário usar uma, ou mais, das técnicas já descritas.

Outra desvantagem é que o *wrapper* ficará profundamente ligado à interface do sistema legado. Isto é, para que uma tarefa seja executada é necessário que a seqüência de teclas seja enviada para o *software* legado, em uma ordem fixa. Mas e se, por algum motivo, o sistema legado for alterado de forma que os menus que levam até um determinado serviço (ou então os campos de edição que proporcionam a execução de

uma tarefa) forem alterados no próprio software legado? Como ficará o *wrapper*? Simplesmente deixará de funcionar. Com isso, após uma mudança, o *wrapper* terá de ser novamente reformulado para que a funcionalidade do serviço oferecido pelo sistema legado possa ser oferecido novamente pelo *wrapper*. Esta reformulação envolve, necessariamente, redefinir a seqüência de teclas que devem ser enviadas para o *software* legado, para que um determinado serviço possa voltar a ser corretamente oferecido.

7.4. A técnica

A técnica consiste, antes de qualquer coisa, determinar a seqüência de teclas que devem ser enviadas para todos os serviços que o software legado oferece para que os mesmos possam ser também oferecidos pelo *wrapper*. Independente da quantidade de comandos de PgDn, PgUp, Left-Arrow, Up-Arrow e demais teclas que forem necessárias para que um determinado serviço seja executado, todas estas teclas precisam ser detalhadamente conhecidas, pois somente assim o *wrapper* poderá ser construído.

Uma vez que as teclas tenham sido determinadas, o *wrapper* deverá ser construído com todos os serviço que o *software* legado dispõe. Com isso, será possível executar remotamente uma determinada tarefa que de outro modo apenas poderia ser executada localmente, ou seja, com o usuário sentado a frente do teclado e digitando os comandos.

De acordo com a técnica a ser utilizada, as teclas serão armazenadas em um arquivo texto para que, posteriormente, este arquivo seja lido e os códigos das teclas lá existentes possam ser enviados para o sistema legado.

Um problema que passamos quando da definição desta técnica foi o de como fazer para o *software* legado ler o conteúdo do arquivo. Seria simples, caso a equipe tenha os fontes. A partir deste ponto, então, poderia ser estudado toda a listagem do *software* legado e a mesma poderia ser alterada para que as leituras que normalmente são feitas pelo teclado possam ser alteradas, para que as informações venham de um arquivo texto, previamente preparado. A idéia parece boa, à primeira vista, mas temos alguns problemas:

1. A equipe terá de estudar os fontes, o que poderá levar meses ou até anos, dependendo da quantidade de sistemas legados envolvidos e o grau de integração entre eles;

2. Sempre que um *software* é alterado, existirá a possibilidade de algum engano ser feito e o sistema poderá não funcionar corretamente. Isto será muito perigoso, caso o software legado seja crítico e não possa ficar inconsistente como, por exemplo, um sistema que faça a dosagem de medicamentos em um paciente durante uma operação;

3. Nem sempre os fontes estão disponíveis. Muitas vezes, do sistema legado, existem apenas o executável. Por isso, pode parecer uma tarefa impossível conseguir alterar um determinado código sem que os fontes do sistema legado estejam disponíveis.

Devido a estes problemas, foi decidido que a solução deve ser alcançada sem que seja necessário alterar nenhuma linha de código do sistema legado. Na realidade, vamos considerar que nem mesmo temos os fontes do *software* legado. Para isto, vamos nos utilizar de um programa residente (TSR) que será disparado pelo *wrapper*. A função deste programa residente será ler o arquivo texto e colocar no *buffer* do teclado cada uma das teclas que o sistema legado espera para executar uma determinada tarefa. Uma vez que as teclas são colocadas no *buffer* do teclado, para o sistema legado, será como se algum usuário estivesse na frente do teclado fazendo a entrada de dados, através de uma simples digitação. O sistema legado não saberá diferenciar se a informação foi introduzida por um usuário, ao pressionar uma tecla, ou se a mesma foi inserida dentro do buffer do teclado, via um programa residente. Este programa residente será escrito usando a linguagem Turbo Pascal e será colocado em funcionamento, através do *wrapper*, imediatamente antes do sistema legado ser colocado em funcionamento e logo após o arquivo texto ter sido preparado.

7.4.1 Um modelo da técnica em funcionamento

A figura 7.2 ilustra o funcionamento da técnica. Após a figura, será mostrado como a solicitação, feita por um cliente, será atendida pelo *wrapper*:

1. O objeto-cliente faz a solicitação de um serviço qualquer;

2. Através do ORB, o serviço é localizado no objeto-servidor;

3. O objeto-servidor, cria uma instância da classe de mapeamento Java, para que o serviço seja executado;

4. O objeto de mapeamento, solicita, através JNI, que um determinado método seja executado;

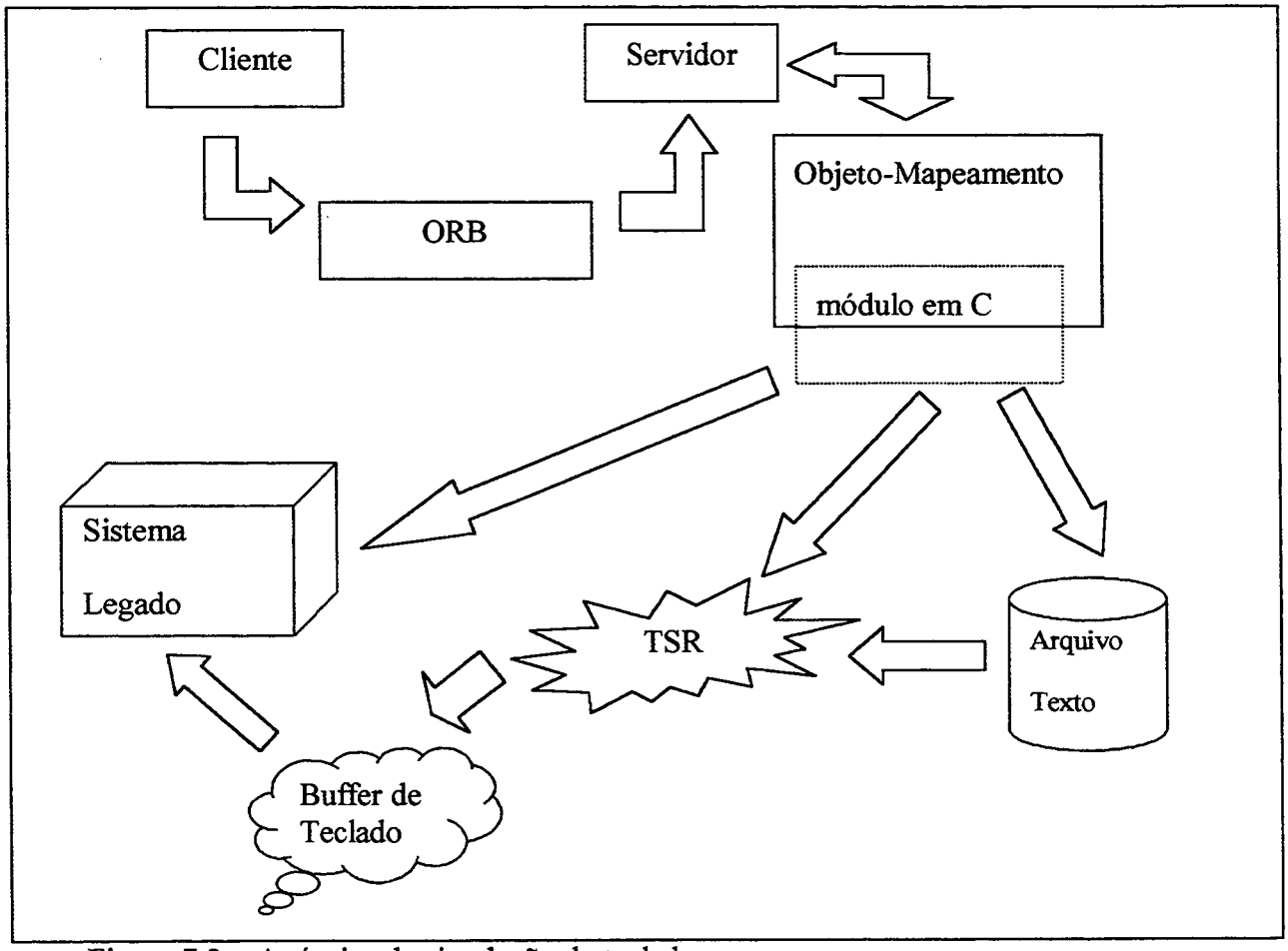

Figura 7.2 – A técnica da simulação do teclado

5. Dentro do arquivo em C, o método cuja execução foi solicitada pelo objeto de mapeamento, cria um arquivo texto contendo todas as seqüências de teclas necessárias para que o sistema legado execute o serviço desejado. O código destas teclas será armazenado dentro deste arquivo texto;

6. É colocado em funcionamento, pelo módulo C, um *software* residente cuja função é continuamente ler o conteúdo do arquivo texto, enquanto lá existirem códigos de teclas;

7. O sistema legado é ativado imediatamente após o programa residente ter sido colocado em funcionamento;

8. O código de cada tecla lida de dentro do arquivo texto, pelo programa residente, será inserido dentro do *buffer* de teclado, para posterior utilização pelo sistema legado;

9. O *software* legado fará a leitura normalmente como sempre o fez e, pelo fato de o *buffer* de teclado estar sendo alimentado constantemente pelo programa residente, o sistema legado obterá as teclas, como se algum usuário as tivesse digitado;

10. Uma vez que o serviço tenha sido executado, o sistema legado será encerrado, já que dentro do arquivo texto também serão colocadas as seqüências das teclas necessárias para terminar a execução do sistema legado.

7.5 A classe de mapeamento

A classe de mapeamento, escrita em Java, não realiza praticamente nenhuma função, além de carregar a biblioteca contendo o módulo C e disponibilizar o método que executará o serviço existente no sistema legado. Mas mesmo assim, para um melhor entendimento, é mostrado na figura 7.3, a listagem da classe de mapeamento:

```
public class TWrapper2SimulacaoTeclas{
  public native void incluiPessoa(String nome, int idade, String endereco);
  static{System.loadLibrary("Wrapper2SimulacaoTeclas_1");}
}
```

Figura 7.3 – A listagem da classe de mapeamento

7.6. O módulo C

Após o código da classe de mapeamento ter sido escrito, é necessário escrever o código do módulo C, o qual é responsável por gerar o arquivo texto contendo os códigos

das teclas a serem usadas pelo sistema legado para a realização do serviço desejado. Além da preparação do arquivo texto, será o módulo C o responsável por colocar em funcionamento o programa residente e também por ativar o próprio sistema legado. Como já foi dito, será o sistema legado que efetivamente irá executar o serviço solicitado pelo Cliente remoto.

Para iniciar a construção do módulo C, é necessário construir o arquivo de cabeçalho contendo a interface dos métodos nativos existentes dentro da classe Java "Twrapper2SimulacaoTeclas", mostrado na figura 7.3. acima.

A geração deste arquivo de cabeçalho, felizmente, é feita automaticamente após a classe de mapeamento Java ter sido compilada através do comando "javac". O comando necessário para gerar o arquivo de cabeçalho é o seguinte:

javah –jni Twrapper2SimulacaoTeclas

Após o comando acima ter sido executado com sucesso, o leitor poderá ver o arquivo de cabeçalho "Twrapper2SimulacaoTeclas.h" a ser usado pelo módulo C.

7.6.1 A listagem do módulo C

No anexo 2, o leitor poderá ver a listagem do módulo C, o qual realiza as seguintes tarefas: gerar o arquivo contendo os códigos das teclas (chamado "teclado.txt"); colocar em funcionamento o programa residente (chamado "relogio.exe"); colocar em funcionamento o sistema legado (chamado "programa.exe").

Na listagem existente no anexo 2, o leitor poderá observar os trechos de código que armazenam, no arquivo texto, a seqüência de teclas responsável por simular o pressionamento da tecla "ENTER". A figura 7.4 ilustra este trecho de código:

```
buffer[ 0 ] = 13;  // codigo da tecla ENTER
buffer[ 1 ] = 48;
buffer[ 2 ] = 13;  // codigo da tecla ENTER
buffer[ 3 ] = 48;
fwrite( &buffer, 4, 1, stream );
```

Figura 7.4 – Armazenamento da tecla ENTER no arquivo texto

O trecho de código, mostrado na figura 7.5, é responsável por transformar as informações vindas do Java para as estruturas de dados aceitas pela linguagem C:

```
zeraVetor( buffer, sizeof( buffer ) );
tam = (*env)->GetStringLength(env, nome );
(*env)->GetStringUTFRegion(env, nome, 0, tam, buffer);
buffer[ strlen( buffer ) ] = 13; // codigo da tecla ENTER
buffer[ strlen( buffer )+1 ] = 48;
// coloca a "idade" na variavel "buffer"
strcat( buffer, fcvt( idade, 0, &decimal, &sinal ) );
// coloca o "endereco" dentro da variavel "buffer"
zeraVetor( buffer2, sizeof( buffer2 ) );
tam = (*env)->GetStringLength(env, endereco );
(*env)->GetStringUTFRegion(env, endereco, 0, tam, buffer2);
strcat( buffer, buffer2 );
buffer[ strlen( buffer ) ] = 13; // codigo da tecla ENTER
buffer[ strlen( buffer )+1 ] = 48;
```

Figura 7.5 – Transformação de tipos Java para tipos C

Após as informações vindas do Java terem sido convertidas para as variáveis da linguagem C, que na listagem tem o nome de "buffer", será necessário armazenar tais informações no arquivo texto. Esta tarefa é executada pelo trecho de código mostrado na figura 7.6.:

```
zeraVetor( buffer2, sizeof( buffer2 ) );
j = 0;
for ( i = 0; i<strlen( buffer ); i++ ){
  buffer2[ j ] = buffer[ i ];
  buffer2[ j+1 ] = 48;
  j = j + 2;
}
```

Figura 7.6 – Armazenando informações no arquivo texto

Após as informações terem sido armazenadas no arquivo texto, será necessário garantir que o sistema legado, após a execução do serviço, encerre o seu funcionamento. Para isto, será necessário enviar também para o arquivo texto uma seqüência de teclas "ESC" que permitam o término da execução do sistema legado. O armazenamento desta seqüência de teclas, no arquivo texto, é mostrado no trecho de código ilustrado pela figura 7.7.:

```
zeraVetor( buffer, sizeof( buffer ) );
buffer[ 0 ] = 27; // codigo da tecla ESC
```

```
buffer[ 1 ] = 48;
buffer[ 2 ] = 27; // codigo da tecla ESC
buffer[ 3 ] = 48;
buffer[ 4 ] = 27; // codigo da tecla ESC
buffer[ 5 ] = 48;
buffer[ 6 ] = 27; // codigo da tecla ESC
buffer[ 7 ] = 48;
fwrite( &buffer, 8, 1, stream );
```

Figura 7.7 – Armazenando a tecla ESC no arquivo texto

Finalmente, após o arquivo texto ter sido preparado, falta apenas colocar em funcionamento o programa residente e, em seguida, o próprio sistema legado. Para isto, basta a execução dos seguintes comandos:

```
system( "relogio.exe" );

system( "programa.exe" );
```

7.7 O programa residente

O programa residente foi escrito usando a linguagem Turbo Pascal da empresa Borland, versão 7.0, mas o mesmo poderia ser compilado usando outras versões do compilador. A listagem do programa residente pode ser vista no anexo3. Este programa residente foi denominado de "Relógio.pas" pelo fato do mesmo redirecionar a interrupção do relógio do computador, através da interrupção "Int 08h".

Na listagem mostrada no anexo 3, é importante notar o trecho de código que faz o redirecionamento da interrupção relógio, int 08h, por uma nova rotina, a qual foi chamada de "nova_int_08h". Este trecho de código é mostrado na figura 7.8:

```
{troca a interupcao de teclado}
SwapVectors;
getIntVec( $08, velha_int_08h );
setIntVec( $08, @nova_int_08h );
```

Figura 7.8 – Redirecionando a interrupção de teclado

Outro trecho importante a ser ressaltado, localizado dentro da nova rotina de interrupção do teclado, é que a rotina antiga (antes do redirecionamento) continue sendo chamada, pois do contrário, os efeitos podem ser imprevisíveis. Esta chamada da antiga interrupção foi feita através da inclusão de código de máquina, através do comando

"inline" logo na entrada da rotina "nova_int_08h". Basicamente, este código de máquina executa o comando *Assembly* "call", no endereço existente dentro da variável "velha_int08h". Esta chamada da antiga interrupção é mostrada em seguida:

```
{chama a antiga int 08h}
Inline( $9C/$FF/$1E/velha_int_08h );
```

A interrupção do teclado ocorre, muitas vezes, enquanto o sistema legado está em funcionamento. Acontece que o *software* legado deverá recuperar teclas do *buffer* do teclado, o que não necessariamente ocorre em um intervalo de tempo sincronizado com a execução da interrupção de teclado. Isto é, a rotina "nova_int_08h" poderá ser chamada diversas vezes antes que o sistema legado tenha tido tempo de recuperar uma única tecla de dentro do *buffer* do teclado. Para evitar isto, foi criado uma espécie de sincronismo. Logo após rotina "nova_int_08h" ter chamado a antiga interrupção do teclado, será verificado se o *buffer* do teclado está vazio. Caso isto seja verdade, será colocado dentro do buffer de teclado uma nova tecla, a qual será obtida através da leitura do arquivo texto preparado pelo módulo C. Com isso, é esperado que o *buffer* de teclado não sofra um *overflow*, pois apenas uma nova tecla será inserida no mesmo quando o *buffer* estiver vazio, o que irá ocorrer sempre que a tecla que ali existia, tenha sido retirado pelo *software* legado.

Para verificar se o *buffer* de teclado está vazio, com o intuito de uma nova tecla poder ser colocada dentro do *buffer*, é feito através do trecho de código mostrado em seguida:

```
if not Keypressed then
begin
{demais comandos}
end;{if}
```

A leitura do arquivo texto é feita de acordo com os códigos mostrados na figura 7.9. Convém notar que no layout definido para o arquivo texto, pelo módulo C, as teclas existentes estão agrupadas sempre em pares. O primeiro byte usado representa o código da tecla e o segundo byte representa o código especial, gerado pelas teclas PgDn, PgUp, etc., que são chamadas de *scancode*. Abaixo, seguem os comandos responsáveis por

esta leitura. Pode ser notado que para garantir a perfeita leitura das teclas deste arquivo, alguns testes foram feitos. Estes testes servem para garantir que o arquivo exista e que os códigos das teclas estejam agrupados corretamente, de acordo com o layout esperado.

```
Assign( arq, 'teclado.txt' );
  {$I-}
  Reset( arq );
  {$I+}
  {somente se o arquivo existe}
  if IOResult = 0 then
  begin
    {examina a variavel de controle de leitura do arquivo}
    if posicao < FileSize( arq ) then
    begin
      {posiciona no proximo byte}
      Seek( arq, posicao );
      {verifico se ainda existe alguma tecla para ser lida}
      if not Eof( arq ) then
      begin
        Read( arq, asciicode );
        {por seguranca testo novamente se nao alcancou o eof}
        if not Eof ( arq ) then
        begin
          Read( arq, scancode );
          {envia o(s) caracter(es) para o buffer do teclado.}
          asm
            mov    ah,05h
            mov    ch,scancode
            mov    cl,asciicode
            int    16h
          end;
        end;{if}
      end;{if}
      {incrementa a posicao para a proxima leitura}
      Inc( posicao,2 );
    end;{if}
    {fecha o arquivo}
    Close( arq );
  end
  else
    writeln( 'arquivo "teclado.txt" nao encontrato' );
end;{if}
```

Figura 7.9 – Leitura do arquivo texto pelo programa residente

Para encerrar, a última parte a ser explicada é a efetiva colocação das teclas dentro do *buffer* de teclado. Isto é feito através de uma interrupção da BIOS, chamada de Int

16h, a qual é responsável por incluir, no *buffer* de teclado, uma determinada tecla. O código responsável pela execução desta interrupção da BIOS, foi escrito em linguagem Assembly e o seu fonte é mostrado na figura 7.10:

```
mov   ah,05h
mov   ch,scancode
mov   cl,asciicode
int   16h
```

Figura 7.10 – Armazenamento de teclas no *buffer* do teclado

A listagem da figura 7.10, mesmo representando um código em linguagem Assembly, está inserida dentro do programa TSR Turbo Pascal. Isto é obtido graças a uma facilidade que a linguagem Turbo Pascal possui que permite a inclusão de código Assembly diretamente dentro de código Pascal.

7.8 A resposta do sistema legado

O TSR após terminar a sua tarefa, não retornará ao objeto de mapeamento nenhum tipo de resposta. O caminho de comunicação entre o objeto de mapeamento e o sistema legado possui somente uma direção. Caso o sistema legado escrito em Clipper tivesse alguma resposta a ser passada para o objeto de mapeamento, como uma mensagem indicando o sucesso ou fracasso da solicitação do serviço, esta resposta não teria como ser enviada para o objeto de mapeamento. Cabe ao objeto de mapeamento considerar que o serviço foi concluído com sucesso.

8. ESTUDO DE CASO: REAPROVEITANDO CÓDIGO PASCAL

A linguagem Pascal foi criada por Nicklaus Wirth em 1968. Desde então, ela se tornou muito conhecida, principalmente entre os meios acadêmicos. Devido a sua grande utilização nas universidades, muitas pessoas começaram a sua carreira profissional escrevendo programas usando esta linguagem. Por isso, é natural que muitos *softwares*, tendo uso comercial ou apenas educacional, tenham sido desenvolvidos através do uso do Pascal. Uma versão do Pascal amplamente utilizada foi desenvolvida pela empresa Borland, sob o nome de Turbo Pascal. Devido ao amplo uso desta versão, ela foi escolhida para demostrar as técnicas de interfaceamento de código discutido nesta sessão.

8.1 O problema

Quando nos deparamos com um código legado escrito em Turbo Pascal, podemos encontrar situações onde os programas fontes podem ou não estar disponíveis. O fato do código fonte não estar disponível se explica muitas vezes pela idade do programa fonte que, com o passar do tempo, poderia ter sido perdido ou mesmo pelo uso de bibliotecas de terceiros que não forneceram os fontes. Em situações como esta, faz-se necessário definir técnicas que auxiliem no aproveitamento deste código legado.

8.2. As técnicas

As técnicas para reutilização dos programas escritos em Turbo Pascal serão determinadas muito em função do tipo de código que temos disponível, isto é, se temos os fontes ou se temos apenas os programas compilados (TPUs). Quando temos os fontes, podemos usar técnicas, desde escrever todos fontes Turbo Pascal em outra linguagem como o Java ou mesmo a linguagem C/C++. No caso de reescrever os fontes, podemos ser auxiliados com o uso de um programa tradutor. Outra técnica seria a compilação dos fontes usando uma ferramenta que faça a geração de código DLL ou de código LIB. Mas, quando não mais estão disponíveis os fontes, isto é, quando existem apenas os arquivos compilados (TPUs), ainda é possível reaproveitar este código compilado, entretanto é necessário usar de técnicas especiais, como será explicado neste capítulo.

8.2.1 Usando a linha de comando para ativar código em arquivo binário

Quando uma biblioteca de sub-rotinas escritas em Turbo Pascal é compilada, a mesma gera um arquivo com a extensão TPU. Este arquivo contém o código binário decorrente do processo de compilação que o Turbo Pascal realiza nas suas bibliotecas (Units no jargão do Turbo Pascal).

Quando o arquivo fonte não está mais disponível, independente do motivo que levou a sua não mais existência, temos que, de alguma forma, usar apenas o arquivo TPU. Ocorre que o Java se comunica apenas com a linguagem C/C++ (através do JNI), e não com a linguagem Turbo Pascal. Com isso, temos que criar algum mecanismo que faça com que a linguagem C/C++ se comunique com o Turbo Pascal e, consequentemente, permita ao Java se aproveitar deste código legado.

Um problema que enfrentamos é que a linguagem C/C++ não pode ser ligada com arquivos TPU. Para resolver isso, devemos usar um pequeno artifício: criar um módulo em Turbo Pascal (desde que seja utilizado uma versão compatível com o arquivo TPU) que permita, via linha de comando, ativar as sub-rotinas existentes dentro do arquivo TPU. Este módulo será construído na forma de um arquivo executável, o qual será ativado pelo programa C.

Antes de construir o módulo em Turbo Pascal, é extremamente necessário que as assinaturas das sub-rotinas existentes dentro do arquivo TPU sejam perfeitamente conhecidas, isto é, a parte pública tem de ser completamente conhecida por qualquer programa que pretenda se utilizar desta Unit (o arquivo TPU), caso contrário, a sua utilização se tornaria impossível.

Na figura 8.1, é colocado a parte pública de uma Unit escrita em Turbo Pascal, contendo as sub-rotinas que serão chamadas de uma classe Java. Notem que a Unit da figura 8.1 tem a única intenção de servir como fonte didática da explicação desta técnica. Por isso, não é interesse deste trabalho mostrar arquivos em Turbo Pascal que possuam uma grande complexidade, justamente porque o objetivo é facilitar o entendimento por parte do leitor.

```
unit rotpas;
interface
const
  MAX_NOTAS = 100;
type
  {define um vetor para armazenar as notas de um aluno hipotético}
  TVetorNotas = array[ 0..MAX_NOTAS-1 ] of Real;
  {retorna o somatório de dois números inteiros}
  function somaDoisInteiros( A, B : Integer ) : LongInt;
  {faz a inversão de uma string qualquer}
  function inverteString( S : string ) : string;
  {retorna a media de um aluno hipotético dado um vetor com suas notas}
  function calculaMedia( var Notas : TVetorNotas; Qtde : Integer ) : Real;
implementation
end.
```

Figura 8.1 – A parte pública de uma Unit Pascal

Após o leitor examinar o código acima, verá que nele existem três sub-rotinas públicas dentro da Unit sendo que estas serão chamadas pelo código Java através do JNI e da linguagem C/C++. Outro ponto importante é que cada sub-rotina terá associado um índice único e exclusivo, que será usado pelo módulo C, e também pelo módulo Turbo Pascal, para ativar as sub-rotinas existentes nesta Unit.

Como já mencionado anteriormente, a técnica a ser utilizada consiste em construir um módulo em Turbo Pascal usando uma versão do compilador que permita chamar o arquivo TPU. Este módulo corresponderá a um programa executável (EXE) que será chamado pelo módulo C. Na chamada a este executável, o módulo C deverá enviar

parâmetros na linha de comando do DOS que serão usados pelo módulo Turbo Pascal para ativar a sub-rotina desejada. Dentre os parâmetros enviados, estão: o nome de um arquivo a ser usado para retorno da resposta das sub-rotinas; o índice associado à sub-rotina (existente dentro do arquivo TPU); e os demais parâmetros exclusivamente necessários para a chamada da sub-rotina.

No parágrafo acima, foi comentado sobre o nome de um arquivo texto para armazenar as respostas das sub-rotinas. A utilização deste arquivo texto se explica pelo fato da impossibilidade de um programa escrito em C chamar diretamente sub-rotinas escritas em Turbo Pascal. Esta impossibilidade se dá no nível interno da construção dos próprios compiladores e do código nativo gerado por eles. Com isso, temos diferenças no nível de segmentos de pilha, segmentos de código, segmentos de dados, etc. Além do fato do compilador Turbo Pascal gerar código em 16 Bits e o compilador C em uso (MS Visual C++) gerar código em 32 bits. Com tudo isso (mais o fato que as formas de chamadas de sub-rotinas em C, bem como o mecanismo de retorno destas sub-rotinas sejam completamente diferentes do Turbo Pascal) se torna necessário que a comunicação seja feita usando um canal comum a ambas as linguagens. Por isso, foi determinado que o módulo C chamaria o módulo Turbo Pascal, sendo que este último ativaria uma das sub-rotinas existentes dentro do arquivo TPU. Assim, o resultado desta ativação seria armazenado pelo próprio módulo Turbo Pascal, dentro de um arquivo texto. O nome deste arquivo texto seria enviado, pelo módulo C, como um dos parâmetros na linha de comando. O módulo C deveria, após o módulo Turbo Pascal ter terminado a sua execução, ler o arquivo texto e devolver o seu conteúdo para o programa Java.

Outro ponto que deve ser destacado é que será definido um índice para cada uma das sub-rotinas públicas existentes na Unit Turbo Pascal. Este índice será um número a ser enviado pela linha de comando de ativação do módulo Turbo Pascal, o qual será usado para selecionar qual a sub-rotina deverá ser executada. O módulo C deverá conhecer qual índice corresponde a qual sub-rotina, caso contrário um código errado poderia ser ativado e efeitos indesejados poderiam ocorrer. No arquivo TPU, as sub-rotinas tiveram associados os índices mostrados na tabela 8.1:

SubRotina na Unit Turbo Pascal	Índice Associado
SomaDoisInteiros	0
InverteString	1
CalculaMedia	2

Tabela 8.1 – Índices das sub-rotinas Pascal

Para permitir um melhor entendimento por parte do leitor, é colocado a seguir uma simulação demostrando como um objeto-cliente faz a solicitação de um serviço qualquer existente dentro do TPU, e como esta solicitação seria atendida:

1. O objeto-cliente faz a solicitação de um serviço qualquer através da chamada de um método existente em um objeto-servidor;

2. Através do ORB o objeto-servidor é localizado;

3. O objeto-servidor chama um método específico, o qual efetivamente executará o serviço existente dentro de um objeto-mapeamento Java. Este objeto-mapeamento foi instanciado pelo próprio objeto-servidor;

4. O objeto-mapeamento Java executa um determinado método "native". É este método que, via JNI, chamará uma sub-rotina escrita em C;

5. O módulo C então montará a linha de comando, a qual conterá o nome do executável, o nome do arquivo texto a ser usado para armazenar a resposta, o índice associado a sub-rotina existente dentro do TPU (que executará o serviço desejado) e, finalmente, os demais parâmetros necessários para a execução da sub-rotina;

6. O módulo Turbo Pascal iniciará a sua execução obtendo os parâmetros que lhe foram passados pela linha de comando, ou sejam: o nome do arquivo texto, o número (índice) da sub-rotina que o módulo Turbo Pascal deverá ativar e os demais parâmetros necessários para a perfeita execução da sub-rotina;

7. O módulo Turbo Pascal, de posse do índice passado na linha de comando, irá chamar a sub-rotina existente dentro do arquivo TPU, enviando os demais parâmetros obtidos da linha de comando para a sub-rotina;

8. A sub-rotina será executada e a resposta será devolvida para o módulo Turbo Pascal;

9. O módulo Turbo Pascal pegará a resposta da sub-rotina e a armazenará dentro do arquivo texto, terminando a sua execução. Com isso o controle de execução retornará ao módulo C;

10. O módulo C, após o término da execução do módulo Turbo Pascal, fará a leitura do arquivo texto, o qual conterá as respostas geradas pela execução do serviço desejado;

11. De posse das respostas, o módulo C fará as conversões de tipo necessárias para transformar o tipo da resposta, de C para um tipo JNI. Com isso, a execução da sub-rotina C terminará e a resposta será retornada para o objeto de mapeamento Java;

12. O objeto-mapeamento Java passará a resposta obtida do módulo C para o objeto-servidor;

13. O objeto-servidor, via ORB, finalmente devolverá esta resposta para o cliente remoto.

A figura 8.2 ilustra as camadas de *software* envolvidas na chamada das sub-rotinas existentes dentro da Unit TPU. A intenção da figura é mostrar ao leitor a quantidade de camadas de *softwares* que devem cooperar para que uma solicitação feita por um Cliente remoto seja atendida por um Servidor.

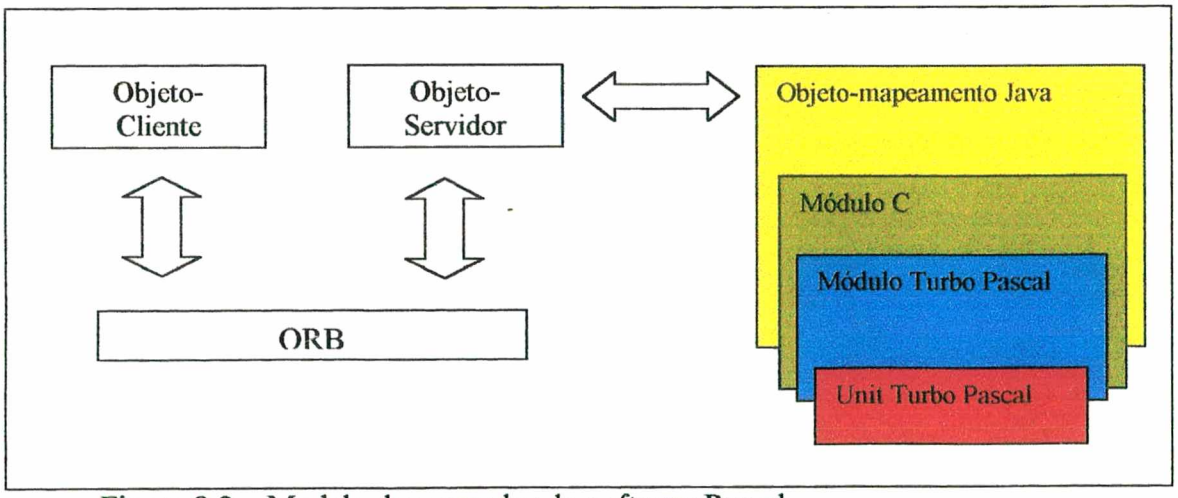

Figura 8.2 – Modelo das camadas de *software* Pascal

No anexo 4, o leitor poderá encontrar a listagem do módulo Turbo Pascal. Este módulo permitirá que as rotinas existentes dentro do arquivo binário TPU possam ser usadas pelo objeto de mapeamento, escrito em Java.

No código fonte do programa mostrado no anexo 4, pode ser visto a sub-rotina denominada "execRotina", a qual é responsável pela ligação entre o índice (o número da sub-rotina já discutido anteriormente), enviado pelo módulo C, e a correspondente sub-rotina do arquivo TPU. Na figura 8.3 é mostrado o código da sub-rotina "execRotina":

```
procedure execRotina;
begin
 if ParamStr( 2 ) = '0' then
  execSomaDoisInteiros
 else if ParamStr( 2 ) = '1' then
  execInverteString
 else if ParamStr( 2 ) = '2' then
  execCalculaMedia
 else
  Writeln( 'Numero da subrotina não encontrado...' );
end;
```

Figura 8.3 – Listagem da rotina de ligação com o arquivo TPU

Analisando o trecho de código da figura 8.3, é possível notar que, dependendo do parâmetro enviado na linha de comando para o executável Wrappas1.exe, uma determinada sub-rotina será executada.

Esta sub-rotina tem por finalidade retirar os demais parâmetros da linha de comando, chamar a sub-rotina da Unit Turbo Pascal que efetivamente executará o serviço desejado e, finalmente, armazenar no arquivo texto o resultado da execução deste serviço.

Como ilustração, na figura 8.4, é mostrado o código da sub-rotina "execSomaDoisInteiros", a qual é responsável pela chamada da sub-rotina "somaDoisInteiros", existente dentro do arquivo TPU.

No trecho de código da figura 8.4, pode ser notado, que são retirados dois (2) parâmetros da linha de comando, respectivamente: "ParamStr(3)" e "ParamStr(4)". Estes parâmetros são convertidos para valores numéricos (uma vez que estes são retirados da linha de comando como valores literais) e então a sub-rotina do arquivo TPU é chamada com estes valores. Após a execução da sub-rotina, o resultado desta será armazenado no arquivo texto.

Quando o módulo Turbo Pascal estiver construído, já poderá ser construído a classe Java e, em seguida, já poderá ser construído o módulo C. A classe Java é muito

simples e é composta de um método native (no linguajar do JNI) para cada sub-rotina existente na Unit Turbo Pascal.

```
Procedure execSomaDoisInteiros;
var
  a, b, erro : Integer;
begin
  if ParamCount <> 4 then
    Writeln( 'Qtde de parâmetros incorretos para executar a sub-rotina' )
  else
  begin
    {pega os parâmetros passados na ativação do programa e os transforma
     para números inteiros}
    Val( ParamStr( 3 ), a, erro );
    Val( ParamStr( 4 ), b, erro );
    {executa a sub-rotina e grava o resultado no arquivo texto}
    Writeln( arq, rotpas.somaDoisInteiros( a, b ) );
  end;{if}
end;
```

Figura 8.4 – Ativando o código existente no arquivo TPU

Após o código Java ter sido construído, será possível ver a chamada das sub-rotinas como se as mesmas fossem as chamadas de métodos de uma classe. Em outras palavras: damos uma "cara" orientada a objetos para rotinas criadas com o paradigma estruturado.

Na figura 8.5, é mostrado o código fonte da classe de mapeamento Java que permite o acesso as sub-rotinas Turbo Pascal:

```
Public class TWrapper2UnitRotPas{
  // retorna o somatório de dois números inteiros
  public native int somaDoisInteiros( int A, int B );
  // faz a inversão de uma string qualquer
  public native String inverteString( String S );
  // retorna a media de um aluno hipotético dado um vetor com suas notas
  public native double calculaMedia( double[] Notas, int Qtde );
  static{ System.loadLibrary("Wrapper2UnitRotPas_1"); }
}
```

Figura 8.5 – Classe de mapeamento Java

Uma vez que o código em Java tenha sido construído, é necessário começar a criação do arquivo DLL "*Wrapper*2UnitRotPas_1". Este arquivo DLL possui o código fonte escrito em C. Notem que é aqui que entra o módulo C, discutido desde o início

deste capítulo. Este módulo é que fará a efetiva chamada para o executável Turbo Pascal.

Uma vez que a classe Java tenha sido compilada, e desde que nenhum erro seja encontrado, o arquivo T*Wrapper*2UnitRotPas.class será criado. Uma vez isto feito, é necessário escrever o módulo C. Mas antes disto, é necessário criar o arquivo de cabeçalho T*Wrapper*2UnitRotPas.h, que será usado pelo próprio módulo C. Para realizar a criação deste arquivo de cabeçalho é necessário a execução do seguinte comando:

javah –jni T*Wrapper*2UnitRotPas

A listagem do arquivo *Wrapper*2UnitRotPas_1.c, que corresponde ao módulo C, pode ser vista no anexo 5. Após análise do código existente no anexo 5, o leitor poderá verificar que foi criada uma sub-rotina para cada um dos códigos "native", que foram definidos na classe "T*Wrapper*2UnitRotPas.java".

Cada uma das sub-rotinas existente no módulo C tem como função chamar o executável Wrappas1.exe enviando, na linha de comando, os parâmetros necessários para a execução do serviço desejado (serviço este existente dentro do arquivo TPU) e, após o módulo Turbo Pascal ter encerrado a sua execução, o módulo C irá ler o arquivo texto, o qual tem o resultado da execução do serviço. O conteúdo deste arquivo será convertido para o tipo JNI esperado pelo Java e retornado para o objeto de mapeamento Java.

No anexo 5, a parte de código responsável pela montagem da linha de comando e a correspondente ativação do executável Wrappas1.exe é mostrada na figura 8.6:

```
strcpy( buffer, "wrappas1.exe resposta.txt 0 " );
 strcat( buffer, fcvt( A, 0, &decimal, &sinal ) );
 strcat( buffer, " " );
 strcat( buffer, fcvt( B, 0, &decimal, &sinal ) );
 system( buffer );
```

Figura 8.6 – A montagem da linha de comando

O módulo C deverá ler o arquivo texto, montado inicialmente pelo módulo Turbo Pascal. Esta leitura é ilustrada na figura 8.7.

```
stream = fopen( "resposta.txt", "r" );
fseek( stream, 0, SEEK_SET );
fread( &buffer, sizeof( buffer ), 1, stream );
fclose( stream );
```

Figura 8.7 – Leitura do arquivo texto

Neste ponto falta apenas gerar a DLL, com base no módulo C discutido anteriormente. Para a criação deste arquivo DLL é necessário disparar o seguinte comando de compilação existente dentro do MS Visual C++:

cl *Wrapper*2UnitRotPas_1.c -Fe*Wrapper*2UnitRotPas_1.dll -MD -LD -Ic:\arquiv~1\micros~2\vc98\include -Id:\compiler\jdk1.2.1\include -Id:\compiler\jdk1.2.1\include\win32 -link c:\arquiv~1\micros~2\vc98\lib\msvcrt.lib c:\arquiv~1\micros~2\vc98\lib\oldnames.lib c:\arquiv~1\micros~2\vc98\lib\kernel32.lib c:\arquiv~1\micros~2\vc98\lib\uuid.lib

8.2.2 Traduzindo o código Turbo Pascal para outras linguagens

Uma forma de aproveitar o código legado escrito em Turbo Pascal seria traduzir, ou melhor, escrever todas as sub-rotinas Pascal para alguma outra linguagem. Este enfoque tem três grandes desvantagens: 1) o tempo necessário para a tradução; 2) o perfeito entendimento do código legado; 3) funcionalidades não existentes na nova linguagem. O leitor que deseja realizar a tradução do sistema legado em Turbo Pascal para uma outra linguagem deverá analisar bem se este trabalho é realmente necessário e se a relação custo X benefício é vantajosa para o cliente que, no final das contas, irá pagar pelo novo *software*. A seguir, é discutido sobre cada uma das desvantagens citadas anteriormente, para que o leitor possa melhor tomar a decisão.

- O tempo necessário para a tradução: o tempo necessário para terminar a tarefa de traduzir o código Turbo Pascal para outra linguagem pode ser demasiadamente grande, uma vez que pode ser necessário traduzir centenas de milhares de linhas de código. Com isso, além do tempo necessário para tradução ser muito grande, os custos decorrentes deste processo (afinal os profissionais que executam este serviço tem de ser pagos) podem ser altíssimos, principalmente se este processo de tradução se estender por meses ou até mesmo por anos.

- O perfeito entendimento do código legado: os profissionais que irão realizar esta (árdua) tarefa de tradução serão obrigados a entender profundamente o *software* legado antes mesmo de começar a fase de tradução. Somente assim eles poderão garantir que; após o processo de tradução, os serviços oferecidos pelo sistema legado continuarão a ser oferecidos pelo novo *software*. Infelizmente, muitas vezes, as coisas não saem exatamente como se quer. Após um processo de tradução, envolvendo vários profissionais, meses de trabalho e milhares de linhas de código, não existe garantia que logo após o *software* legado ter sido traduzido ele começará a funcionar perfeitamente, tal qual era o sistema escrito em Turbo Pascal.

É relevante lembrar que, muitas vezes, principalmente em sistemas mais antigos, a única fonte de documentação existente se restringe apenas ao próprio código fonte. Dependendo da idade do sistema legado, podem não existir mais nenhum tipo de documentação, arquivos de help ou, até mesmo os profissionais que inicialmente desenvolveram o sistema legado (ou mesmo aqueles que realizavam a manutenção) podem não mais ser encontrados. Infelizmente, nunca foi uma prática muito comum de programadores documentar os programas fontes, tampouco era hábito comum escrever código de forma legível, principalmente em sistemas legados mais antigos. Com isso, pode se tornar difícil para a equipe de tradução entender partes, ou até mesmo todo o sistema legado. Além disso, quando o código não é perfeitamente entendido, a sua tradução se torna mais complexa. Com isso, o sistema traduzido poderá apresentar incorreções semânticas (lógicas) que poderão comprometer a sua funcionalidade.

- Funcionalidades não existentes na nova linguagem: muitos sistemas legados escritos em Turbo Pascal podem se utilizar de bibliotecas (TPUs), desenvolvidas por terceiros, as quais não necessariamente deixam disponíveis os fontes. Neste caso, o uso da técnica descrita no capítulo 7 se torna indispensável. Outra questão é que o sistema legado pode possuir características ou comando próprios da linguagem Turbo Pascal que não existem implementados na nova linguagem (por exemplo, o comando "Delay" do Turbo Pascal). Neste caso, este comando inexistente deverá ser implementado na nova linguagem. Com isso, o trabalho de tradução poderá se tornar mais difícil, uma vez que o perfeito entendimento desta funcionalidade inexistente deverá ser obtida, caso contrário a sua tradução não se dará de forma perfeita e, consequentemente, a tradução

do sistema legado como um todo, não alcançará o objetivo primordial que é deixar o novo sistema (traduzido) funcionando tal como ele era em Turbo Pascal.

8.2.2.1 Usando um tradutor de código Turbo Pascal

O leitor que desejar traduzir o código Turbo Pascal para alguma outra linguagem poderá escolher ser auxiliado por alguma ferramenta automatizada de tradução. Estes tipos de ferramentas têm como entrada um programa escrito em Turbo Pascal e a saída será a tradução deste código para a linguagem desejada. Recomenda-se que o leitor tenha muito cuidado com a utilização deste tipo de ferramenta. Mesmo que o código possa ser traduzido 100%, o que nem sempre é verdade (leia o tópico "Funcionalidade não existentes na nova linguagem"), ainda há a possibilidade do código traduzido pelo *software* conversor se mostrar ilegível ou, na melhor das hipóteses, com pouca legibilidade. Para estas ferramentas, o resultado final corresponde apenas ao software traduzido, muitas vezes sem que existam preocupações com a legibilidade do código traduzido. Desta forma, futuras manutenções podem se tornar difíceis de serem realizadas. Numa situação mais comum, os conversores de código, ou tradutores se preferirem, nem sempre conseguem realizar uma tradução completa. Nestes casos, a equipe responsável pela conversão do código escrito em Turbo Pascal, terá que verificar os trechos de código que não puderam ser traduzidas e realizar manualmente as conversões ou adaptações que forem necessárias. O problema que surge, como já mencionado, é que após a tradução feita pelo *software* conversor, a legibilidade pode ser pouca. Neste caso, a equipe poderá não apenas ter de estudar o programa em Turbo Pascal para entender qual trecho não foi traduzido, como terá de ler um código de pouca legibilidade, para compreender, em qual parte a intervenção manual a ser feita pela equipe de tradução, terá de ocorrer. Isto pode se mostrar uma tarefa bem árdua e custosa, tanto em questões de tempo, como de dinheiro. Por isso, o uso de uma ferramenta que promete a realização de conversões automáticas deve ser muito bem estudada.

8.2.3 Chamando código Turbo Pascal existente em um arquivo no formato DLL

Este tipo de formato permite uma maior facilidade, do ponto de vista de ligação com o módulo C, uma vez que a linguagem C/C++ poderá mais facilmente se comunicar com as sub-rotinas escritas em Turbo Pascal. É verdade que o compilador C não pode se ligar diretamente com arquivos DLL, mas facilmente poderemos converter o arquivo DLL para o formato LIB, o qual permite a ligação direta com a linguagem C/C++. Tal conversão poderá ser facilmente obtida através do uso de um programa gratuito chamado IMPLIB32 [Imp32], cuja função é pegar um arquivo DLL e gerar um outro arquivo no formato LIB. Outra vantagem é que não existirá a necessidade da criação de um *wrapper* Pascal, como foi visto anteriormente. Com isso, a quantidade de camadas de *software* necessárias para a execução do serviço Java será reduzida, aumentando com isso a simplicidade da solução, bem como a velocidade da execução.

Um ponto importante que deve ser observado é que todos os fontes do módulo Pascal, pertencente ao sistema legado, devem estar disponíveis. Nenhum arquivo binário, seja no formato TPU, seja no formato OBJ, poderá ser usado a não ser que estes arquivos binários possuam similares na arquitetura de 32 bits. Lembre-se que os programas em Pascal normalmente foram feitos para operar em 16 bits e a geração de DLLs para trabalhar em conjunto com o objeto de mapeamento Java obriga a utilização de arquivos na arquitetura de 32 bits.

Conforme pode ser visto na figura 8.8, a conversão das sub-rotinas Turbo Pascal em uma DLL, através do ambiente Delphi, diminui a quantidade de camadas de *software* necessárias para que um objeto-cliente possa executar uma determinada funcionalidade, existentes no arquivo Turbo Pascal.

O compilador Turbo Pascal não permite a criação de DLLs. Por isso, é necessário usar um outro artifício: usar um outro ambiente que permita a criação de arquivos DLLs e que também possa aceitar arquivos escritos em Turbo Pascal.

O ambiente que escolhido para a criação da DLL pertence a empresa Borland e é chamado de Delphi, e neste trabalho foi usado a versão 4.0. É importante notar que esta solução apenas poderá ser obtida caso o fonte da Unit Turbo pascal esteja disponível, uma vez que o ambiente Delphi não aceita o formato TPU, típico do compilador Turbo Pascal.

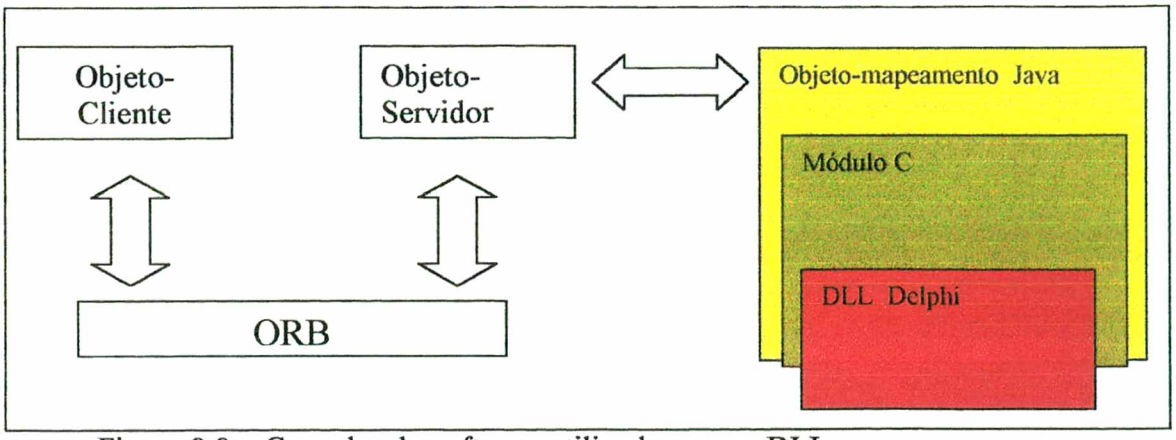

Figura 8.8 – Camadas de *software* utilizadas com a DLL

O código em Turbo Pascal que será convertido para uma DLL pode ser encontrado no anexo 6. O leitor interessado deverá se dirigir até este anexo, para ter um melhor entendimento do arquivo fonte Pascal.

Dentro do ambiente Delphi, será necessário criar uma Library (arquivo para gerar DLL, no linguajar Delphi), contendo uma sub-rotina de mapeamento para cada uma das sub-rotinas existentes dentro da Unit Turbo Pascal que desejamos exportar. Este mapeamento é necessário porque a construção da DLL Delphi exige que determinados tipos sejam usados, bem como determinadas diretivas sejam usadas na assinatura das funções públicas. O código fonte da DLL Delphi é mostrado no anexo 7. O leitor interessado deverá se dirigir até o anexo 7, para obter maiores informações.

No código existente no anexo 7, é possível notar que todas as sub-rotinas que são exportadas pela Unit Turbo Pascal recebem uma "casca" de uma outra sub-rotina para que determinados tipos e diretivas sejam usadas. Na figura 8.9, pode ser visto um trecho de código, retirado do anexo 7, que faz a ativação de uma rotina existente dentro da Unit Turba Pascal:

```
function DLL_inverteString( S : PChar ) : PChar; stdcall;
var
  C : array[ 0..255 ] of Char;
begin
  {coloca dentro do vetor "C" a resposta da execução da sub-rotina}
  StrPCopy( C, inverteString( StrPas( S ) ) );
  {retorna a resposta}
  Result := C;
end;
```

Figura 8.9 – Ativando uma rotina da Unit Turbo Pascal

Notem que, no trecho de código mostrado na figura 8.9, foram usados algumas sub-rotinas especiais próprias do ambiente Delphi para realizar as conversões entre os tipos da DLL e os tipos Turbo Pascal. Estas sub-rotinas são: "STRPAS" e "STRPCOPY". Notem também o uso da diretiva "STDCALL" logo após a assinatura da sub-rotina "DLL_INVERTESTRING". Todas estas conversões e diretivas são necessárias para a perfeita criação da DLL, caso contrário, o programa IMPLIB32 não poderia realizar a conversão do arquivo DLL para o formato LIB e, consequentemente, o módulo C não iria poder se utilizar das sub-rotinas da Unit TPU (depois, é claro, de convertidas para o formato LIB).

Uma vez que o arquivo DLL tenha sido gerado pelo Delphi, é possível fazer a criação do arquivo LIB. Para isto, é necessário utilizar o programa IMPLIB32. Este *software* recebe com entrada um arquivo DLL e gera como saída o mesmo arquivo, só que no formato LIB. Para realizar a geração do arquivo LIB, basta dar o comando abaixo (Consideraremos que o arquivo DLL gerado pelo ambiente Delphi recebeu o nome de Project1.Dll):

IMPLIB32 project1

Após o comando acima ter sido executado, o programa IMPLIB32 irá criar um arquivo no formato LIB chamado PROJECT1.LIB, baseado no arquivo Project1.Dll, o qual contém as sub-rotinas públicas da Unit Turbo Pascal. De posse deste arquivo LIB, será necessário criar o arquivo na linguagem C/C++. De posse deste arquivo em C/C++, deverá ser utilizado o compilador "cl", ligando o arquivo LIB através do parâmetro "link" e então será criado o arquivo no formato DLL. Este arquivo poderá então ser usado pelo objeto de mapeamento Java.

9. TRABALHOS FUTUROS

Durante o estudo deste trabalho, até mesmo durante o desenvolvimento dos *wrappers*, foi verificado que alguns tipos de sistemas e algumas técnicas para a comunicação entre o *wrapper* e os sistemas legados poderiam ser implementadas. A seguir, são propostos alguns trabalhos complementares a este projeto inicial.

Sistemas legados em ambientes proprietários – Alguns sistemas legados foram construídos dentro de ambientes próprios (que rodam em cima do DOS), por exemplo: os ambientes da linguagem MUMPS. Tais ambientes contém os arquivos fontes e os dados, dentro de uma estrutura proprietária, a qual não permite um acesso direto. O único arquivo existente é o que coloca em funcionamento o próprio ambiente MUMPS. Desta forma, uma estrutura de *wrappers* teria que, de alguma forma, conseguir interagir com os dados e os fontes existentes dentro do ambiente MUMPS, enquanto este ambiente estiver em funcionamento.

Utilizar a memória RAM para comunicação – Na maioria dos *wrappers* apresentados neste trabalho, o elo de comunicação entre as linguagens se deu através de um arquivo texto. Uma forma alternativa que poderia ser estudada seria enviar e receber informações entre os módulos C e os módulos das outras linguagens (exemplo: Java, C/C++, Turbo Pascal, Clipper, entre outras) que utilizam a memória RAM. Com isso, as diferenças de como as linguagens manipulam a memória e como a arquitetura 16 bits e a arquitetura 32 bits acessam os blocos de memória devem ser, de alguma forma, resolvidas.

Ferramenta para forçar a tipagem forte de arquivos fontes Clipper – Construção de uma ferramenta que analise os arquivos fontes de programas Clipper e faça a troca

das variáveis usadas nos programas, por outras variáveis, de tal forma que seja forçado a tipagem forte nos arquivos fontes do Clipper.

10. CONCLUSÕES

Integrar um sistema legado, com objetos distribuídos, pode ser necessário que alterações sejam realizadas nos aplicativos legados. Tais alterações podem se tornar inviáveis para o cliente, por dois motivos: quando o tempo necessário para a realização das alterações for demasiadamente longo ou quando o custo financeiro necessário para a concretização das tarefas for excessivamente grande. Quando um sistema legado levou, por exemplo, anos para ser inicialmente desenvolvido, não é admissível, para o cliente, que o mesmo tempo seja necessário para a execução das alterações. Da mesma forma, se o aplicativo legado custou, na época de seu desenvolvimento, centenas de milhares/milhões de dólares, não é possível que um valor similar seja desembolsado, novamente pelo cliente. Ambos os temores - tempo de desenvolvimento e custo financeiro - deve ser eliminado da cabeça (e do bolso) do cliente através do uso das técnicas descritas neste trabalho. Tais técnicas foram rotuladas como sendo três tipos de *wrappers*: *Wrapper* de Tela, *Wrapper* de Banco de Dados e *Wrapper* de Código.

Com estas técnicas é perfeitamente possível, independente da linguagem utilizada, banco de dados (ou forma de entrada de dados) efetivarem as alterações sem que, para isso, o cliente seja onerado com uma longa espera, ou mesmo com um grande custo. Os aplicativos legados que seguem o padrão EXE, para arquivos executáveis, feitos para funcionar no sistema operacional DOS (podendo ter ou não disponíveis os arquivos fontes) são candidatos diretos para a utilização das técnicas de *wrapper*. Com o *Wrapper* de Tela e com a utilização de programas residentes, é possível criar mecanismos que simulem o pressionamento de teclas de tal forma que o aplicativo legado será executado como se um usuário estivesse digitando no teclado. Com o *Wrapper* de Banco de Dados é possível acessar o sistema de arquivos, ou SGBD, do

sistema legado. Através do *Wrapper* de Código é possível que programas escritos na linguagem Java possam ativar rotinas escritas em outras linguagens, mesmo que, a princípio, estas linguagens não sejam diretamente integradas com o Java, por exemplo, Java e Pascal.

Quando um ou mais *wrappers* forem construídos, é possível criar classes de objetos distribuídos, usando a tecnologia CORBA, a fim de permitir que tais objetos distribuídos façam acesso aos *wrappers*, que por sua vez também não deixam de serem classes. Desta forma, sistemas legados que tenham passado pelo processo de *wrapping* podem ser integrados à tecnologia de objetos distribuídos, além de, por causa dos *wrappers*, os aplicativos legados passam a se comportar como *softwares* orientadas ao objetos, mesmo que os sistemas legados não tenham sido escritos usando este paradigma.

Com os objetos distribuídos, um objeto cliente remoto poderá fazer solicitação de tarefas que serão executadas pelo sistema legado. A infra-estrutura oferecida pelo CORBA poderá ser usada para habilitar a objetos clientes acesso a serviços de objetos servidores, mesmo que seja desconhecido o local onde estes objetos servidores estejam fisicamente localizados.

A utilização dos *wrappers* é uma grande técnica, pois permite que aplicativos legados, independente da linguagem, banco de dados e da época em que os sistemas legados tenham sido construídos, possam vir a se beneficiar dos recursos oferecidos pelos objetos distribuídos.

11. ANEXOS

Anexo 1 – Sistema legado hipotético (usado no capítulo 7)

```
#include "set.ch"
#include "box.ch"
#include "inkey.ch"
#define TRUE .T.
#define FALSE .F.
LOCAL nOpcao := 1, cCor
//
cCor := SETCOLOR( "W/W" )
@ 1, 0 CLEAR TO 1, MAXCOL()
SETCOLOR( "W/B,W+/N" )
@ 2, 0 CLEAR TO MAXROW(), MAXCOL()
@ 2, 0, MAXROW(), MAXCOL() BOX B_SINGLE
//
SET WRAP ON
SET DELIMITERS ON
SET DELIMITERS TO "[]"
//
DO WHILE TRUE
 SETCOLOR( "N/W,W+/N" )
 @ 1, 02 PROMPT "Arquivo"
 @ 1, 14 PROMPT "Editar"
 @ 1, 26 PROMPT "Pesquisar"
 @ 1, 38 PROMPT "Exibir"
 @ 1, 50 PROMPT "Opcoes"
 @ 1, 62 PROMPT "Ajuda"
 @ 1, 74 PROMPT "Sair"
 MENU TO nOpcao
 //
 DO CASE
 CASE nOpcao == 0
  EXIT
 CASE nOpcao == 1
  IF !MenuArquivo()
   EXIT
  ENDIF
 CASE nOpcao == 2
 CASE nOpcao == 3
 CASE nOpcao == 4
 CASE nOpcao == 5
 CASE nOpcao == 6
 CASE nOpcao == 7
  EXIT
 ENDCASE
ENDDO
//
SETCOLOR( cCor )
```

```
CLEAR SCREEN
RETURN
//
*****************************************************************************
***
STATIC FUNCTION MenuArquivo()
 LOCAL nOpcao := 0, bResposta := TRUE, cTela, cCor
 //
 cTela := SAVESCREEN( 2, 0, 9, 17 )
 cCor := SETCOLOR( "W/W" )
 @ 2, 0 CLEAR TO 9, 17
 SETCOLOR( "N/W,N/W" )
 @ 2, 0, 9, 17 BOX B_DOUBLE
 DO WHILE TRUE
  SETCOLOR( "N/W,W+/N" )
  @ 3, 2 PROMPT "Novo"
  @ 4, 2 PROMPT "Abrir..."
  @ 5, 2 PROMPT "Salvar"
  @ 6, 2 PROMPT "Salvar como..."
  @ 7, 2 PROMPT "Fechar"
  @ 8, 2 PROMPT "Sair"
  MENU TO nOpcao
  //
  DO CASE
  CASE nOpcao == 0
   EXIT
  CASE nOpcao == 1
   MenuNovo()
  CASE nOpcao == 2
  CASE nOpcao == 3
  CASE nOpcao == 4
  CASE nOpcao == 5
  CASE nOpcao == 6
   bResposta := FALSE
   EXIT
  ENDCASE
 ENDDO
 //
 SETCOLOR( cCor )
 RESTSCREEN( 2, 0, 9, 17, cTela )
RETURN bResposta
//
*****************************************************************************
***
STATIC PROCEDURE MenuNovo()
 LOCAL cTela, cCor, cNome := REPLICATE( " ", 28 )
 LOCAL nIdade := 0, cEndereco := REPLICATE( " ", 28 )
 LOCAL cPicNome := REPLICATE( "X", 28 )
```

```
 LOCAL cPicEndereco := REPLICATE( "X", 28 )
 //
 cTela := SAVESCREEN( 6, 25, 10, 70 )
 cCor := SETCOLOR( "W+/N" )
 @ 6, 25 CLEAR TO 10, 70
 @ 6, 25, 10, 70 BOX B_SINGLE
 DO WHILE TRUE
  @ 7, 26 SAY "Nome.....:" GET cNome PICTURE cPicNome
  @ 8, 26 SAY "Idade.....:" GET nIdade PICTURE "99"
  @ 9, 26 SAY "Endereco...:" GET cEndereco PICTURE cPicEndereco
  READ
  IF LASTKEY() == K_ESC
   EXIT
  ELSE
   MostraMsg()
  ENDIF
 ENDDO
 //
 SETCOLOR( cCor )
 RESTSCREEN( 6, 25, 10, 70, cTela )
RETURN
//
*****************************************************************************
***
STATIC PROCEDURE MostraMsg()
 LOCAL cTela, cCor
 //
 cTela := SAVESCREEN( 12, 35, 17, 55 )
 cCor := SETCOLOR( "W/W" )
 @ 12, 35 CLEAR TO 17, 55
 SETCOLOR( "N/W" )
 @ 12, 35, 17, 55 BOX B_DOUBLE
 @ 14, 40 SAY "Incluido..."
 SETCOLOR( "R/R" )
 @ 16, 43 CLEAR TO 16, 48
 SETCOLOR( "N/R" )
 @ 16, 45 SAY "Ok"
 INKEY( 0 )
 //
 SETCOLOR( cCor )
 RESTSCREEN( 12, 35, 17, 55, cTela )
RETURN
```

Anexo 2 – O módulo C (usado no capítulo 7)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "TWrapper2SimulacaoTeclas.h"

// ********************************************************************
void zeraVetor( char* S, int n ){
 int i;
 for ( i=0; i<n; i++ )
   S[ i ] = 0;
}

// ********************************************************************
JNIEXPORT void JNICALL Java_TWrapper2SimulacaoTeclas_incluiPessoa
 (JNIEnv* env, jobject thisObj, jstring nome, jint idade, jstring endereco){
 char buffer[ 256 ], buffer2[ 256 ];
 int tam, i, j, decimal, sinal;
 FILE *stream;

 // zera o buffer
 zeraVetor( buffer, sizeof( buffer ) );
 // grava as informacoes no arquivo texto
 stream = fopen( "teclado.txt", "w" );
 fseek( stream, 0, SEEK_SET );
 // coloca os comandos para ativar as opcoes de menu
 buffer[ 0 ] = 13;  // codigo da tecla ENTER
 buffer[ 1 ] = 48;
 buffer[ 2 ] = 13;  // codigo da tecla ENTER
 buffer[ 3 ] = 48;
 fwrite( &buffer, 4, 1, stream );
 // coloca o "nome" dentro da variavel "buffer"
 zeraVetor( buffer, sizeof( buffer ) );
 tam = (*env)->GetStringLength(env, nome );
 (*env)->GetStringUTFRegion(env, nome, 0, tam, buffer);
 buffer[ strlen( buffer ) ] = 13; // codigo da tecla ENTER
 buffer[ strlen( buffer )+1 ] = 48;
 // coloca a "idade" na variavel "buffer"
 strcat( buffer, fcvt( idade, 0, &decimal, &sinal ) );
 // coloca o "endereco" dentro da variavel "buffer"
 zeraVetor( buffer2, sizeof( buffer2 ) );
 tam = (*env)->GetStringLength(env, endereco );
 (*env)->GetStringUTFRegion(env, endereco, 0, tam, buffer2);
 strcat( buffer, buffer2 );
 buffer[ strlen( buffer ) ] = 13; // codigo da tecla ENTER
 buffer[ strlen( buffer )+1 ] = 48;
```

```
// prepara o "buffer2" para armazenar as teclas
zeraVetor( buffer2, sizeof( buffer2 ) );
j = 0;
for ( i = 0; i<strlen( buffer ); i++ ){
  buffer2[ j ] = buffer[ i ];
  buffer2[ j+1 ] = 48;
  j = j + 2;
}
// escreve no arquivo
fwrite( &buffer2, strlen( buffer )*2, 1, stream );
// coloca no arquivo os comandos para sair do sistema legado
zeraVetor( buffer, sizeof( buffer ) );
buffer[ 0 ] = 27; // codigo da tecla ESC
buffer[ 1 ] = 48;
buffer[ 2 ] = 27; // codigo da tecla ESC
buffer[ 3 ] = 48;
buffer[ 4 ] = 27; // codigo da tecla ESC
buffer[ 5 ] = 48;
buffer[ 6 ] = 27; // codigo da tecla ESC
buffer[ 7 ] = 48;
fwrite( &buffer, 8, 1, stream );
// fecha o arquivo texto
fclose( stream );

// coloca o programa residente em funcionamento
system( "relogio.exe" );
// coloca o sistema legado em funcionamento
system( "programa.exe" );
}
```

Anexo 3 – O programa residente (usado no capítulo 7)

```
program relogio;
{$M 1024, 0, 0}
uses
  dos,crt;
var
  velha_int_08h : Pointer;
  contador : LongInt Absolute $0000:$046C;
  posicao : LongInt;

{**************************************************************************
********}
procedure nova_int_08h; interrupt;
var
  asciicode, scancode : Char;
  arq : file of Char;
begin
  {chama a antiga int 08h}
  Inline( $9C/$FF/$1E/velha_int_08h );
  {devido a execucao da interrupco do teclado (Int08h) ocorrer
   varias vezes, enquanto uma operacao de leitura/abertura de arquivo,
   ou mesmo durante a execucao do preenchimento do buffer do teclado,
   coloco este teste para que o codigo somente seja executado quando
   a operacao de leitura do arquivo tenha terminado}
  if not Keypressed then
  begin
    {abre o arquivo contendo os codigos a serem
     posteriormente colocados no buffer do teclado}
    Assign( arq, 'teclado.txt' );
    {$I-}
    Reset( arq );
    {$I+}
    {somente se o arquivo existe}
    if IOResult = 0 then
    begin
      {examina a variavel de controle de leitura do arquivo}
      if posicao < FileSize( arq ) then
      begin
        {posiciona no proximo byte}
        Seek( arq, posicao );
        {verifico se ainda existe alguma tecla para ser lida}
        if not Eof( arq ) then
        begin
          Read( arq, asciicode );
          {por seguranca testo novamente se nao alcancou o eof}
          if not Eof ( arq ) then
          begin
```

```
        Read( arq, scancode );
        {envia o(s) caracter(es) para o buffer do teclado.}
        asm
          mov    ah,05h
          mov    ch,scancode
          mov    cl,asciicode
          int    16h
        end;
       end;{if}
      end;{if}
      {incrementa a posicao para a proxima leitura}
      Inc( posicao,2 );
    end;{if}
    {fecha o arquivo}
    Close( arq );
   end
   else
    writeln( 'arquivo "teclado.txt" nao encontrato' );
 end;{if}
end;
begin
 {zera as variaveis de controle}
 posicao := 0;

 {troca a interupcao de teclado}
 SwapVectors;
 getIntVec( $08, velha_int_08h );
 setIntVec( $08, @nova_int_08h );
 {mantem o programa residente}
 Keep( 0 );
end.
```

Anexo 4 – O módulo Pascal (usado no capítulo 8)

```
program wrappas1;
uses
 rotpas;
var
 arq : Text;
{executa a sub-rotina "somaDoisNumeros"}
procedure execSomaDoisInteiros;
var
 a, b, erro : Integer;
begin
 if ParamCount <> 4 then
  Writeln( 'Qtde de parâmetros incorretos para executar a sub-rotina' )
 else
 begin
  {pega os parâmetros passados na ativação do programa e os transforma
   para números inteiros}
  Val( ParamStr( 3 ), a, erro );
  Val( ParamStr( 4 ), b, erro );
  {executa a sub-rotina e grava o resultado no arquivo texto}
  Writeln( arq, rotpas.somaDoisInteiros( a, b ) );
 end;{if}
end;
{executa a sub-rotina "inverteString"}
procedure execInverteString;
begin
 if ParamCount <> 3 then
  Writeln( 'Qtde de parâmetros incorretos para executar a sub-rotina' )
 else
 begin
  {pega a string passada na ativação do programa e faz a inversão. O
   resultado, no caso a string invertida, será gravada no arquivo texto}
  Writeln( arq, rotpas.inverteString( ParamStr( 3 ) ) );
 end;{if}
end;
{executa a sub-rotina "calculaMedia"}
procedure execCalculaMedia;
var
 vet : TVetorNotas;
 erro, qtde, i : Integer;
begin
 if ParamCount < 2 then
  Writeln( 'Qtde de parâmetros incorretos para executar a sub-rotina' )
 else
 begin
  {determina a qtde de notas passadas na ativação do programa}
  qtde := ParamCount - 2;
```

```
    {preenche o vetor com as notas passadas na ativação}
    for i := 0 to qtde-1 do
      Val( ParamStr( 2+i+1 ), vet[ i ], erro );
    {executa a sub-rotina e grava o resultado no arquivo texto}
    Writeln( arq, rotpas.calculaMedia( vet, qtde ):0:2 );
  end;{if}
end;
{executa a rotina solicitada pelo usuário na forma de um numero}
procedure execRotina;
begin
  if ParamStr( 2 ) = '0' then
    execSomaDoisInteiros
  else if ParamStr( 2 ) = '1' then
    execInverteString
  else if ParamStr( 2 ) = '2' then
    execCalculaMedia
  else
    Writeln( 'Numero da Sub-Rotina não encontrado...' );
end;
begin
  {eh necessário que seja passado parâmetros na linha de comando}
  if ParamCount < 2 then
    Writeln( 'uso : WRAPPAS1 <arquivo> <rotina> [param 1, param 2,...,param n]' )
  else
  begin
    {abre o arquivo enviado como parâmetro na chamada do programa}
    Assign( arq, ParamStr( 1 ) );
    Rewrite( arq );
    {executa a sub-rotina solicitada}
    execRotina;
    Close( arq );
  end;{if}
end.
```

Anexo 5 – O módulo C (usado no capítulo 8)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "TWrapper2UnitRotPas.h"

// ************************************************************************
void zeraVetor( char* S, int n ){
  int i;
  for ( i=0; i<n; i++ )
    S[ i ] = 0;
}

// ************************************************************************
JNIEXPORT jint JNICALL Java_TWrapper2UnitRotPas_somaDoisInteiros
  (JNIEnv* env, jobject thisObj, jint A, jint B){
  jint resposta = 0;
  int decimal, sinal;
  char buffer[128];
  FILE *stream;
  // zera o buffer
  zeraVetor( buffer, sizeof( buffer ) );

  // monta a linha de comando
  strcpy( buffer, "wrappas1.exe resposta.txt 0 " );
  strcat( buffer, fcvt( A, 0, &decimal, &sinal ) );
  strcat( buffer, " " );
  strcat( buffer, fcvt( B, 0, &decimal, &sinal ) );
  // executa o programa Wrapper, em Pascal, para permitir
  // acesso as sub-rotinas da Unit Rotpas.tpu
  system( buffer );
  // zero o buffer para armazenar o resultado do arquivo
  zeraVetor( buffer, sizeof( buffer ) );
  // como resultado da execucao da sub-rotina eh colocado dentro
  // do arquivo "resposta.txt"
  // faco a leitura deste arquivo
  stream = fopen( "resposta.txt", "r" );
  fseek( stream, 0, SEEK_SET );
  fread( &buffer, sizeof( buffer ), 1, stream );
  fclose( stream );
  // transforma o conteudo do buffer para o tipo correto
  resposta = atoi( buffer );

  return( resposta );
}

// ************************************************************************
```

```
JNIEXPORT jstring JNICALL Java_TWrapper2UnitRotPas_inverteString
  (JNIEnv* env, jobject thisObj, jstring S){
  char buffer[ 256 ], dest[ 256 ];
  int tam;
  FILE *stream;
  // zera o buffer
  zeraVetor( buffer, sizeof( buffer ) );
  // pega o tamanho da string enviada pelo Java
  tam = (*env)->GetStringLength(env, S );
  // copia a String Java para o vetor "buffer"
  (*env)->GetStringUTFRegion(env, S, 0, tam, buffer);
  // monta a linha de comando
  strcpy( dest, "wrappas1.exe resposta.txt 1 " );
  strcat( dest, buffer );
  // executa o programa Wrapper, em Pascal, para permitir
  // acesso as sub-rotinas da Unit Rotpas.tpu
  system( dest );
  // zero o buffer para armazenar o resultado do arquivo
  zeraVetor( buffer, sizeof( buffer ) );
  // como resultado da execucao da sub-rotina eh colocado dentro
  // do arquivo "resposta.txt"
  // faco a leitura deste arquivo
  stream = fopen( "resposta.txt", "r" );
  fseek( stream, 0, SEEK_SET );
  fread( &buffer, sizeof( buffer ), 1, stream );
  fclose( stream );
  // retorna a string invertida para o java
  return( (*env)->NewStringUTF(env,buffer) );
}

// ************************************************************************
JNIEXPORT jdouble JNICALL Java_TWrapper2UnitRotPas_calculaMedia
  (JNIEnv* env, jobject thisObj, jdoubleArray Notas, jint Qtde){
  jdouble resposta = 0;
  jdouble vetor[ 100 ];
  int decimal, sinal;
  char buffer[256], tmp[ 20 ], d1, d2;
  int i;
  FILE *stream;
  // zera o buffer
  zeraVetor( buffer, sizeof( buffer ) );
  zeraVetor( tmp, sizeof( tmp ) );
  // coloca as notas do Java para dentro do vetor
  (*env)->GetDoubleArrayRegion( env, Notas, 0, Qtde, vetor );
  // monta a linha de comando
  strcpy( buffer, "wrappas1.exe resposta.txt 2" );

  for ( i = 0; i<Qtde; i++ ){
```

```
  strcat( buffer, " " );
  // transforma o numero em string
  strcpy( tmp, fcvt( vetor[ i ], 2, &decimal, &sinal ) );
  // copia os digitos da parte decimal
  d1 = tmp[ strlen( tmp )-2 ];
  d2 = tmp[ strlen( tmp )-1 ];
  // coloca o "." no valor convertido
  tmp[ strlen( tmp )-2 ] = '.';
  tmp[ strlen( tmp )-1 ] = d1;
  tmp[ strlen( tmp ) ] = d2;
  tmp[ strlen( tmp )+1 ] = 0;
  // concatena o numero na linha de comando
  strcat( buffer, tmp );
 }
 // executa o programa Wrapper, em Pascal, para permitir
 // acesso as sub-rotinas da Unit Rotpas.tpu
 system( buffer );
 // zero o buffer para armazenar o resultado do arquivo
 zeraVetor( buffer, sizeof( buffer ) );
 // como resultado da execucao da sub-rotina eh colocado dentro
 // do arquivo "resposta.txt"
 // faco a leitura deste arquivo
 stream = fopen( "resposta.txt", "r" );
 fseek( stream, 0, SEEK_SET );
 fread( &buffer, sizeof( buffer ), 1, stream );
 fclose( stream );
 // transforma o conteudo do buffer para o tipo correto
 resposta = atof( buffer );
 return( resposta );
}
```

Anexo 6 – A Unit Turbo Pascal (usada no capítulo 8)

```
unit rotpas;
interface
const
  MAX_NOTAS = 100;
type
  {define um vetor para armazenar as notas de um aluno hipotético}
  TVetorNotas = array[ 0..MAX_NOTAS-1 ] of Real;
  {retorna o somatório de dois números inteiros}
  function somaDoisInteiros( A, B : Integer ) : LongInt;
  {faz a inversão de uma string qualquer}
  function inverteString( S : string ) : string;
  {retorna a media de um aluno hipotético dado um vetor com suas notas}
  function calculaMedia( var Notas : TVetorNotas; Qtde : Integer ) : Real;
implementation
const
  NULO = '';
function somaDoisInteiros( A, B : Integer ) : LongInt;
begin
  somaDoisInteiros := A + B;
end;

function inverteString( S : string ) : string;
var
  tam, i : Integer;
  resposta : string;
begin
  tam := Length( S );
  resposta := NULO;
  for i := tam downto 1 do
    resposta := resposta + S[ i ];
  {}
  inverteString := resposta;
end;

function calculaMedia( var Notas : TVetorNotas; Qtde : Integer ) : Real;
var
  soma : Real;
  i : Integer;
begin
  soma := 0;
  for i := 0 to Qtde-1 do
    soma := soma + Notas[ i ];
  {}
  if Qtde > 0 then
    calculaMedia := soma / Qtde
```

```
  else
    calculaMedia := 0.0;
end;
end.
```

Anexo 7 – O módulo DLL (usado no capítulo 8)

```
library Project1;
uses
  SysUtils,
  Classes,
  rotpas in 'Rotpas.pas';

function DLL_somaDoisInteiros( A, B : Integer ) : LongInt; stdcall;
begin
  Result := somaDoisInteiros( A, B );
end;

function DLL_inverteString( S : PChar ) : PChar; stdcall;
var
  C : array[ 0..255 ] of Char;
begin
  {coloca dentro do vetor "C" a resposta da execução da sub-rotina}
  StrPCopy( C, inverteString( StrPas( S ) ) );
  {retorna a resposta}
  Result := C;
end;

function DLL_calculaMedia( var Notas : TVetorNotas; Qtde : Integer ) : Extended;
stdcall;
begin
  Result := calculaMedia( Notas, Qtde );
end;

exports
  DLL_somaDoisInteiros index 1,
  DLL_inverteString index 2,
  DLL_calculaMedia index 3;
begin
end.
```

12. LISTA DE ABREVIATURAS E SIGLAS

API	*Application Program Interface*. Conjunto de rotinas que permitem acesso a algum tipo de funcionalidade de um software ou hardware.
AUX	Dispositivo auxiliar padrão no DOS.
BIOS	*Basic Input Output System*. Conjunto de rotinas que realiza uma interface entre o sistema operacional e o hardware do computador.
CORBA	*Commom Object Request Broker Architecture*. Infraestrutura responsável pela tecnologia de objetos distribuídos possibilitando a criação de arquiteturas de software em N camadas.
CON	Dispositivo de entrada/saída padrão no DOS.
DLL	*Dynamic Link Library*. Arquivos carregados em tempo de execução. Estes tipos de arquivos surgiram inicialmente com o MS-Windows.
DOS	*Disk Operating System*. Sistema Operacional de maior utilização nos equipamentos padrão IBM-PC.
EXE	Formato de arquivos executáveis.
IDL	*Interface Definition Language*. Linguagem usada para definir interfaces, a serem usadas pela infraestrutura CORBA.
JDBC-ODBC	*Java Database Conectivity-Open Database Conectivity*. Mecanismo criado pela empresa Sun para acesso a banco de dados através do uso do padrão ODBC.

JDK	*Java Development Kit.*
JNI	*Java Native Interface.* Conjunto de rotinas que permitem aos programas escritos em Java executar rotinas escritas na linguagem C/C++.
JVM	*Java Virtual Machine.* Camada de software responsável pela execução de programas escritos na linguagem Java.
LAN	*Local Area Network.* Rede local de computadores.
LIB	Abreviatura de *Library*, que corresponde a uma extensão de arquivos usada por algumas linguagens como o C/C++.
ODBC	*Open Database Conectivity.* Mecanismo criado pela empresa Microsoft para enviar comandos SQL para banco de dados relacionais.
OBJ	Formato de arquivos após o processo de compilação de alguns tipos de linguagens, como C/C++, Clipper, Assembly, entre outras.
ORB	*Object Request Broker.* Padrão de utilização de objetos distribuídos usado pelo CORBA.
PRN	Dispositivo de impressora padrão no DOS.
RAM	*Random Access Memory.* Memória primária do computador.
SGBD	Sistemas Gerenciadores de Bases de Dados. Sosftwares responsáveis pelo armazenamento e recuperação de informações em um banco de dados.
TPU	*Turbo Pascal Unit.* Formato de arquivo gerado após a compilação, através do compilador Turbo Pascal, de fontes escritos usando o modelo conhecido pelos programadores Tutbo Pascal como tendo o nome de Units.
TSR	*Terminate and Stay Resident.* Programas residentes que são executados através do compartilhamento de uso da CPU com outros programas.
WEB	Sigla que representa a Internet.

13. GLOSSÁRIO

Array	Tipo de dado existente nas linguagens de programação.
Assembly	Linguagem de computador mais próxima da linguagem de máquina.
Blinker	Software criado pela empresa Blink com a finalidade de criar arquivos no formato DLL para uso com a linguagem Clipper.
Buffer	Área de memória para armazenamento temporário de informações.
Clipper	Linguagem de computador de alto nível, que combina, linguagem de programação, com linguagem de manipulação de banco de dados.
Cobol	Linguagem de computador de alto nível.
Delphi	Ambiente de desenvolvimento criado pela empresa Borland.
Extended System	Conjunto de rotinas, criadas pelo Clipper, para permitir a comunicação entre módulos Clipper e módulos C
Gateway	Conjunto de rotinas usadas para estabelecer a comunicação entre dois outros sistemas de software.
Handle	Manipulador de dispositivo.
Java	Nome da linguagem de computador criada pela empresa Sun.
Oracle	Nome de um Sistema Gerenciador de Banco de Dados.
String	Tipo de dado existente nas linguagens de programação.
Turbo Pascal	Linguagem de computador de alto nível.
Unit	Nome da pela linguagem Turbo Pascal para as suas bibliotecas de funções.
VisualObjects	Linguagem de computador compatível com o MS-Windows correspondente a próxima evolução da linguagem Clipper.
Wrapper	Camada de software usada para "empacotar" uma determinada biblioteca, aplicativo, API, ou qualquer tipo de software, dando a sensação que o software "empacotado" é orientado ao objeto, mesmo que este software tenha sido escrito usando este paradigma estruturado.
Wrapping	Processo de criar o *wrapper* para um sistema legado.

14. REFERÊNCIAS BIBLIOGRÁFICAS

AMBLER, Scott W. Análise e Projeto Orientados a Objeto: Seu Guia para Desenvolver Sistemas Robustos com Tecnologia de Objetos. IBPI Press, 1998. Volume II.

BLINKINC. http://www.blinkinc.com, consulta feita em 09/09/2001

BORLAND, Turbo Assembler: User's Guide. Borland Internacional, 1989.

BRODIE, Michael L.; Stonebraker, Michael. Migrating Legacy Systems: Gateways, Interfaces & The Incremental Approach. Morgan Kaufmann Publishers, 1995.

CHAN. http://www-cad.eecs.berkeley.edu/~naji/litsearch/omg_ics.html, consulta feita em 05/10/2000

COULOURIS, George; Dollimore, Jean; Kindberg, Tim. Distributed Systems: Concepts and Design. Addison-Wesley, 1996. Second Edition.

DATE, C. J. Introdução a Sistemas de Bancos de Dados Editora Campus, 1984.

FSHIP. http://www.fship.com, consulta feita em 20/03/2001

FUNES, Manuel. Java e Banco de Dados. Brasport, 1999.

GANTI, Narsim; Brayman, William. The Transition of Legacy Systems to a Distributed Architecture. Wiley-QEDe, 1995.

GORDON, Rob. Essential. JNI: Java Native Interface. Prentice-Hall, 1998.

IMP32. http://www.geocities.com/SiliconValley/5806/implib32.htm, consulta feita em 20/03/2001

JNI 97. http://java.sun.com/products/jdk/1.2/docs/guide/jni/spec/jniTOC.doc.html, consulta feita em 20/07/2000

LEMAY, Laura; Perkins, Charles L. Aprenda Java em 21 Dias. Editora Campus, 1997.

LIANG, Sheng. The Java Native Interface: Programmer's Guide and Specification. Addison-Wesley, 1999.

MARTIN, James; Odell, J. James. Análise E Projeto Orientados A Objeto. Makron Books, 1995.

MOWBRAY, Thomas J.; Zahavi, Ron. The Essential CORBA: Systems Integration UsingDistributed Objects. John Wiley & Sons Inc., 1995.

MPSINC. http://www.mpsinc.com, consulta feita em 09/09/2001

NANTUCKET, Clipper 5.0 Referência. Nantucket Corporation, 1990.

ORFALI, Robert; Harkey, Dan. Client/Server Programming with Java and CORBA. John Wiley & Sons Inc., 1998. Second Edition.

PAGE-JONES, Meilir. O Que Todo Programador Deveria Saber Sobre Projeto Orientado A Objeto. Makron Books, 1999.

PTKA, Richard L.; Morgenthal JP.; Forge, Simon. Manager's Guide to Distributed Environments: From Legacy to Living Systems. Wiley Computer Publishing, 1999.

RUMBAUGH, James; Blaha, Michael; Premerlani, Willian; et al. Modelagem E Projetos Baseados Em Objetos. Editora Campus, 1991.

TISCHER, Michael.. PC Intern System Programming – The Encyclopedia of DOS Programming Know How. Data Becker, 1992.

TOM GORDON. http://msdn.microsoft.com/LIBRARY/ BACKGRND/ HTML/MSDN_WRAPPING.HTM, consulta feita em 05/06/2000

VO. http://www.cavo.com, consulta feita em 09/09/2001

WARREN, Ian.. The Renaissance of Legacy Systems: Method Suport for Software-System Evolution. Springer, 1999.

WIJEGUNARATNE, Indrajit; Fernandez, George. Distributed Applications Engineering: Building New Application and Managing Legacy Applications with Distributed Technologies. Springer, 1998.

WUTKA, Mark. Java. Técnicas Profissionais. Berkeley, 1997.

YALLOUZ, Carlos. Programas Residentes No IBM PC. Livros Técnicos e Científicos Editora, 1991.