UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ – UTFPR
CURSO SUPERIOR DE TECNOLOGIA EM ANÁLISE E DESENVOLVIMENTO DE SISTEMAS

ANDRÉ RICARDO GNHOATO

USO DA BIBLIOTECA DPLOT JR COM JAVA E UM COMPARATIVO ENTRE DUAS INTERFACES DE ACESSO: JAVA NATIVE INTERFACE E JAVA NATIVE ACCESS

MEDIANEIRA
2011

ANDRE RICARDO GNHOATO

USO DA BIBLIOTECA DPLOT JR COM JAVA E UM COMPARATIVO ENTRE DUAS INTERFACES DE ACESSO: JAVA NATIVE INTERFACE E JAVA NATIVE ACCESS

Pré-Projeto apresentado à disciplina de Trabalho de Diplomação, do Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas – COADS – da Universidade Tecnológica Federal do Paraná – UTFPR, como requisito parcial para obtenção do título de Tecnólogo.

Orientador: Prof. Paulo Lopes de Menezes, MSc.

MEDIANEIRA

2011

Ministério da Educação
Universidade Tecnológica Federal do Paraná
Diretoria de Graduação e Educação Profissional
Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas

TERMO DE APROVAÇÃO

USO DA BIBLIOTECA DPLOT JR COM JAVA E UM COMPARATIVO ENTRE DUAS INTERFACES DE ACESSO: JAVA NATIVE INTERFACE E JAVA NATIVE ACCESS

Por

André Ricardo Gnhoato

Este Trabalho de Diplomação (TD) foi apresentado às 08:20 h do dia 29 de Novembro de 2011 como requisito parcial para a obtenção do título de Tecnólogo no Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas, da Universidade Tecnológica Federal do Paraná, *Campus* Medianeira. O candidato foi arguido pela Banca Examinadora composta pelos professores abaixo assinados. Após deliberação, a Banca Examinadora considerou o trabalho aprovado.

Prof. MSc. Paulo Lopes de Menezes
UTFPR – *Campus* Medianeira
(Orientador)

Prof. Nelson Miguel Betzek
UTFPR – *Campus* Medianeira
(Convidado)

Prof. M.Eng Juliano R Lamb
UTFPR – *Campus* Medianeira
(Convidado)

Prof. M.Eng. Juliano R Lamb
UTFPR – *Campus* Medianeira
(Responsável pelas atividades de TCC)

A folha de aprovação assinada encontra-se na Coordenação do Curso.

RESUMO

GNHOATO, André Ricardo. Uso da biblioteca DPLOT Jr com Java utilizando *Java Native Interface* (JNI). 2011. Trabalho de Conclusão de Curso, Universidade Tecnológica Federal do Paraná. Medianeira, 2011.

O presente trabalho é um estudo de duas ferramentas para acesso de bibliotecas nativas na plataforma Java. São apresentadas as características de cada uma bem como as suas funcionalidades. No final é apresentado um breve estudo de caso onde a ferramenta destacada como a mais adequada para o propósito do trabalho está aplicada em conjunto com uma biblioteca nativa do sistema operacional *Windows*.

Palavras chave: JNA, JNI, DPLOT, aplicação nativa.

RESUMO EM LINGUA ESTRANGEIRA

GNHOATO, André Ricardo. Use of library DPLOT Jr with java using Java Native Interface (JNI). 2011. Work of Conclusion of Course. Federal Technological University of Paraná. Medianeira, 2011.

This work is a study of two tools to access native libraries on the Java platform. Presents the characteristics of each one and features. At the end is a brief study case where the tool highlighted with better use for the purpose of the work, together with a native library Windows operating system.

Keywords: JNA, JNI, DPLOT, native application.

LISTA DE SIGLAS

API	*Application Programming Interface*
C/C++	*Programming Languages C e C plus plus*
CPU	*Central Processing Unit*
DLL	*Dynamic Language Library*
DPLOT JR	*Biblioteca Nativa para geração de gráficos*
HTML	*HyperText Markup Language*
JDK	*Java Development Kit*
JNA	*Java Native Access*
JNI	*Java Native Interface*
JRE	*Java Runtime Environment*
JVM	*Java Virtual Machine*

LISTA DE FIGURAS

LISTA DE QUADROS

SUMÁRIO

1 INTRODUÇÃO

No desenvolvimento de sistemas no qual envolvem várias plataformas, arquiteturas e diferentes fins para cada tipo de aplicação, encontrar uma solução que atenda todo o escopo do projeto é primordial, porém, não é sempre que a plataforma oferece todos os recursos necessários. Em muitas aplicações é necessário realizar a integração de bibliotecas que implementam tecnologias diferentes e que operam em paradigmas distintos. Nesse trabalho serão abordadas tecnologias que trabalham sobre diferentes implementações, apresentando uma biblioteca nativa do sistema operacional Windows, e as tecnologias voltadas para a utilização dessas bibliotecas em aplicações na plataforma Java.

Em Java para se trabalhar com bibliotecas nativas do sistema como DLL's é necessário fazer uso de recursos, internos ou externos da implementação da JVM. Estes recursos permitem a integração de código escrito na linguagem de programação Java com código escrito em outras linguagens como C e C++. (JNI, 1999).

A biblioteca DPLOT JR 2002, é um software livre destinado a manipular dados bidimensionais e tridimensionais e, por meio de uma *Dynamic Language Library* (DLL) [1], provê mecanismos de comunicação entre processos *Dynamic Data Exchange* (DDE) [2], possibilitando que qualquer programa faça uso de sua interface para gerar gráficos sofisticados (HYDESOFT, 2002).

1.1 OBJETIVO GERAL

Realizar um estudo de caso utilizando a biblioteca DPLOT JR em conjunto de uma aplicação Java desktop, explorando interfaces de acessos a bibliotecas nativas.

---

[1] DLL é a implementação feita pela *Microsoft* para o conceito de bibliotecas compartilhadas nos sistemas operacionais *Microsoft Windows* e OS/2

[2] DDE é uma tecnologia para a comunicação entre múltiplas aplicações executadas em *Microsoft Windows* e OS/2 introduzida pela *Microsoft* em 1987.

1.2 OBJETIVOS ESPECÍFICOS

Os objetivos específicos são:

- Estudar a tecnologia *Java Native Interface* e tecnologias semelhantes.
- Estudar a tecnologia da biblioteca DPLOT.
- Desenvolver uma aplicação para utilizar de forma integrada as tecnologias em questão.

1.3 JUSTIFICATIVA

A plataforma Java é um ambiente de programação que consiste na Java *Virtual Machine* (JVM[3]) e na Java *Application Programming Interface* (API). Aplicações Java, são escritas na linguagem de programação Java, e compilada em um formato de classe binária independente da máquina em que está sendo executada. A classe pode ser executada usando qualquer implementação da maquina virtual. A JVM é um software que é implementado em hardware não virtual em sistemas operacionais, fornece um ambiente no qual *bytecode*[4] Java pode ser executado. É distribuído juntamente com um conjunto de bibliotecas de classe padrão que implementam o Java *Application Programming Interface* (API). As API's apropriadas e empacotados juntos formam o *Java Runtime Environment* (JRE). Qualquer implementação da plataforma Java é garantida para oferecer suporte à linguagem de programação Java, a máquina virtual e a API (JNI, 2011).

O termo *host environment* representa o sistema operacional, um conjunto de bibliotecas nativas e o conjunto de instruções da CPU. Aplicações nativas são escritas na linguagem de programação nativa do sistema, tais como C e C++, compilados em um código binário específico do *host*, e vinculados com as bibliotecas nativas. Aplicações e bibliotecas nativas normalmente são dependentes do ambiente de execução (sistema operacional) específico. Por exemplo, uma aplicação C construída para um sistema operacional normalmente não irá executar em outro *host environment*. A plataforma Java é executada em cima de um sistema operacional. Por exemplo, o JRE (*Java*

[3] JVM significa uma máquina virtual para a plataforma Java.
[4] Código Java compilado, interpretado pela JVM.

Runtime Environment) é um produto da Oracle[5] que oferece suporte aos sistemas operacionais como Solaris, Windows e Linux, oferecendo um conjunto de recursos independente do ambiente em que está sendo executado (JNI, 2011).

Diversas vezes durante a integração de um sistema ou durante o desenvolvimento de alguma aplicação, baseado em diversos sistemas e ferramentas diferentes, depara-se com a necessidade de interagir com programas de terceiros ou com o próprio sistema operacional. O JNI tem como objetivo principal, contornar as restrições impostas pelo uso da JVM. Com ele é possível acessar, desde códigos Java, a bibliotecas desenvolvidas em linguagens com maior suporte a hardware e maior desempenho, como C / C + +, *Assembly* etc.

1.4 ESTRURA DO TRABALHO

O trabalho foi divido em cinco capítulos que abordarão os seguintes temas:

O capítulo dois discorre sobre o JNI e sobre os objetivos do projeto, aborda o mapeamento de tipos e trata as referências. Também é abordado o JNA, uma solução semelhante ao JNI que possui uma implementação diferente. Ainda neste capítulo é tratado sobre os recursos JNA, mapeamento da biblioteca, mapeamento de função, mapeamento de tipos, mapeamento de invocação. Após demonstrar a forma na qual as tecnologias trabalham foi inserido um comparativo entre as duas tecnologias. Em seguida encontram-se informações sobre a biblioteca nativa, uma abordagem sobre os tipos de dados e as principais funções da biblioteca.

O capítulo três trás um estudo de caso envolvendo uma simples aplicação referente ao uso das tecnologias apresentadas no capítulo anterior.

No capítulo quatro são apresentadas às conclusões deste trabalho.

[5] *Oracle Corporation* é uma corporação multinacional americana de tecnologia informática especializada no desenvolvimento e comercialização de sistemas de hardware e software.

2 REVISÃO BIBLIOGRÁFICA

"*Write once, run anywhere"* (Escreva uma vez, rode em qualquer lugar) slogan da linguagem Java que sempre enfatizou a portabilidade da linguagem e suas *APIs*, prometendo que o mesmo código será executado em qualquer plataforma (BARON e MARTINI, 2008). Porém quando se trabalha com aplicações que fazem uso de uma biblioteca específica de um sistema operacional, perde-se essa portabilidade.

No decorrer deste trabalho serão destacadas as tecnologias que fornecem acesso a código nativo sendo possível manter uma conversação com código Java, expondo também algumas falhas e vantagens.

Duas tecnologias serão apresentadas:

- A primeira tecnologia é chamada JNI (*Java Native Interface*). Ela é fornecida por padrão e requer JDK (*Java Development Kit*) para lidar com uma linguagem nativa para fazer chamadas para funções nativas.
- A segunda tecnologia é chamada JNA (*Java Native Access*). É uma API *open source*, que tem a vantagem de abstrair a camada nativa.

2.1 JNI – JAVA NATIVE INTERFACE

O conceito de uma interface de programação que une diferentes linguagens não é novo. Os programas desenvolvidos em C, por exemplo, geralmente podem chamar funções escritas em linguagens como *FORTRAN* e *Assembly*. Da mesma forma, as implementações de linguagens de programação tais como *LISP* e *Smalltalk* suportam uma variedade de interfaces de função externas (JNI, 2011).

O JNI resolve um problema semelhante ao abordado pelos mecanismos de interoperabilidade apoiada por outras linguagens. Há, no entanto, uma diferença significativa entre a JNI e os mecanismos de interoperabilidade usados em muitas outras linguagens. O JNI não foi concebido para uma implementação específica da JVM, pelo contrário, é uma interface nativa que pode ser suportada por todas as implementações da máquina virtual Java (JNI, 2011).

Ela é um recurso que permite tirar proveito da plataforma Java, utilizando códigos escritos em outras linguagens. Como parte da implementação da JVM, o JNI é

uma interface bidirecional que permite que os aplicativos Java possam invocar código nativo e vice-versa. A Figura 1 ilustra o papel do JNI.

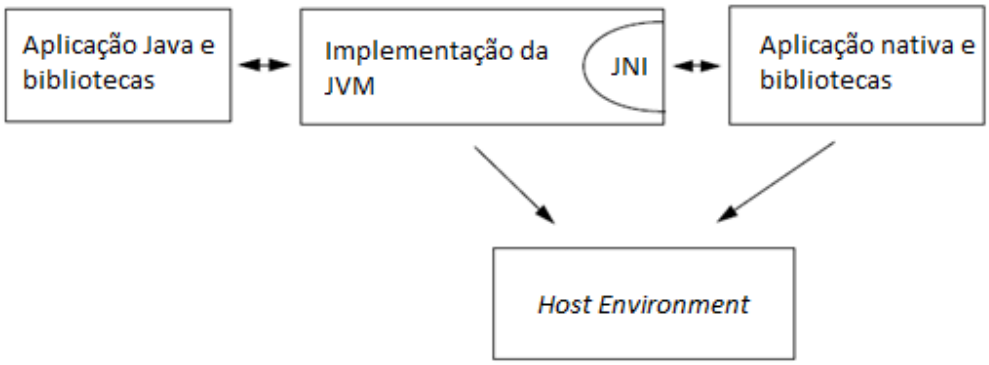

Figura 1- Papel do JNI
Fonte (JNI, 2011)

O JNI é projetado para controlar situações onde é necessário combinar aplicações Java com códigos nativos.

- É possível usar JNI para escrever métodos nativos que permitem que as aplicações Java, chamem funções implementadas em bibliotecas nativas de forma que as aplicações Java possam chamar os métodos nativos da mesma forma que os métodos implementados na própria linguagem de programação Java. Por outro lado, entretanto, os métodos nativos são implementados em outra linguagem e residem nas bibliotecas nativas (JNI, 2011).
- Aplicações nativas podem vincular-se com a biblioteca nativa, permitindo assim o uso da interface de invocação para executar componentes de software escritos em linguagem de programação Java. Por exemplo, um navegador de web escrito em C pode executar *applets*[6] incorporado em uma implementação da JVM (JNI, 2011).

2.1.1 Objetivos do projeto JNI

O objetivo do projeto é garantir que o JNI ofereça compatibilidade binária entre as diferentes implementações da máquina virtual Java em um determinado *host environment*. O binário da mesma biblioteca nativa será executado em diferentes implementações de máquinas virtuais em um determinado *host environment* sem a necessidade de recompilação.

[6] Programa escrito na linguagem de programação Java que pode ser incluído em uma página HTML, da mesma forma que uma imagem é incluída em uma página. (Sun Developer Network, 2011)

O JNI é uma camada de programação que permite que o código Java chame, ou seja chamado pelo código nativo, não sendo necessário adicionar uma biblioteca para a camada JNI uma vez que este programa é fornecido por padrão no JDK, sendo necessário apenas por a biblioteca nativa no *path* do sistema, caso ela já não se encontre (BARON e MARTINI, 2008).

Uma chamada para uma função nativa utilizando JNI não é direta sendo necessário definir um método nativo que atenda a um propósito específico tornando obrigatório passar por um método intermediário que irá incluir esta chamada. (BARON e MARTINI, 2008)

2.1.2 Mapeamento de tipos

Os tipos de argumentos na declaração de métodos nativos têm tipos correspondentes em linguagens de programação nativa, o JNI define um conjunto de tipos C e C++ que correspondem aos tipos na linguagem de programação Java.

Existem duas formas de tipos na linguagem de programação Java: tipos primitivos como *int*, *float* e *char*, e tipos de referência, tais como classes, instâncias e matrizes. Na linguagem de programação Java, *strings* são instâncias da classe *java.lang.String* (JNI, 2011).

O JNI trata tipos primitivos e tipos de referência de forma diferente. O mapeamento de tipos primitivos é simples de forma que o tipo *int*, por exemplo, na linguagem de programação Java equivale ao tipo *jint* em C/C++ (definido no *jni.h* como um inteiro de 32-bit), enquanto o tipo *float* em Java equivale ao tipo *jfloat* em C e C++ (definido no *jni.h* como um número de ponto flutuante de 32 bits) (JNI, 2011).

O JNI passa objetos para métodos nativos como referências opacas. Referências opacas são ponteiros C que se referem a estruturas de dados internas na JVM. O layout exato das estruturas de dados internas, no entanto, é escondida do programador. O código nativo deve manipular os objetos subjacentes via as funções apropriadas do JNI, que estão disponíveis através do ponteiro de interface `JNIEnv`. Por exemplo, o tipo JNI correspondente para *java.lang.String* é *jstring*. O valor exato de uma referência *jstring* é irrelevante para o código nativo (JNI, 2011).

Todas as referências JNI são do tipo *jobject*. Por conveniência e segurança o JNI define um conjunto de tipos de referência que são conceitualmente "subtipos" de

jobject. (A é um subtipo de B e cada instância de A é também uma instância de B). Esses subtipos correspondem a tipos de referência usados com frequência na linguagem de programação Java (JNI, 2011).

2.1.3 Refêrencias locais e globais

O código nativo nunca inspeciona diretamente o conteúdo de um ponteiro de referência opaca. Em vez disso, usa funções para acessar a estrutura de dados apontada por uma referência opaca. Por apenas lidar com referências opacas, não é necessário preocupar-se com layout de objeto interno que é dependente de uma determinada implementação da JVM (JNI, 2011).

O JNI suporta três tipos de referências opacas: referências locais, referências globais, referências globais fracas. Referências locais e globais têm tempo de vida diferente. Referências locais são automaticamente liberadas, enquanto referência global e global fraca permanece válida até que seja liberada pelo programador.

Nem todas as referências podem ser usadas em todos os contextos. É ilegal, por exemplo, usar uma referência local depois que o método nativo criou os retornos de referência (JNI, 2011).

2.1.3.1 Referência local

A maioria das funções JNI criam referências locais. Por exemplo, a função JNI *New-Object* cria uma nova instância e retorna uma referência local para essa instância. A referência local só é válida no contexto dinâmico do método nativo que criá-lo, e só dentro de uma invocação do método nativo. Todas as referências locais criadas durante a execução de um método nativo serão liberadas após o retorno do método. Em consequência não se deve escrever métodos nativos que armazenam uma referência local em uma variável estática e usá-la a mesma referência em invocações subsequentes (JNI, 2011).

2.1.3.2 Referência global

A referência global pode ser usada por várias invocações de um método nativo. Uma referência global pode ser usada em vários segmentos e permanece válida até que

seja liberada pelo programador. Como uma referência local, uma referência global garante que o objeto não referenciado seja eliminado pelo *garbage collector*[7] (JNI, 2011).

Ao contrário de referências locais, que são criados pela maioria das funções JNI, referências globais são criados por apenas uma função JNI *NewGlobalRef*.

2.1.3.3 Referência global fraca

Referência global fraca são criadas usando *NewGlobalWeakRef* e liberadas usando *DeleteGlobalWeakRef*. Como referências globais, referências globais fracas permanecem válidas em chamadas de método nativo e em diferentes segmentos. Ao contrário de referências globais, as referências globais fracas não mantém o objeto subjacente para ser coletado pelo *garbage collector*. Referências globais fracas tornam-se úteis quando uma referência armazenada em *cachê,* pelo código nativo não deve manter o objeto subjacente (JNI, 2011).

2.1.4 Suporte a reflexão

Reflexão geralmente se refere à manipulação de linguagem em nível tempo de execução. A reflexão, por exemplo, permite que se descubra em tempo de execução o nome dos objetos de classe arbitrária e o conjunto de campos e métodos definidos na classe. A reflexão é fornecida no nível da linguagem de programação Java através do pacote *java.lang.reflect*, bem como alguns métodos nas classes *java.lang.Object* e *java.lang.Class*. Embora seja possível sempre chamar a API Java correspondente para realizar operações de reflexão, o JNI fornece as seguintes funções para fazer as operações de reflexão frequentes de código nativo mais eficiente e conveniente: (JNI, 2011).

- `GetSuperclass`: retorna a referência da superclasse de uma classe determinada.
- `IsAssignableFrom`: verifica se as instâncias de uma classe pode ser usada quando as instâncias de outra classe são esperadas.
- `GetObjectClass:` retorna a classe de uma determinada referência do *jobject*.

[7] Forma de gerenciamento automático de memória. O coletor de lixo, ou apenas coletor, tenta recuperar o lixo, ou a memória ocupada por objetos que não estão mais em uso pelo programa. (GC, 2011)

- `IsInstanceOf:` verifica se a referência de *jobject* é uma instância de uma determinada classe.

2.2 JNA – JAVA NATIVE ACCESS

JNA é uma alternativa de acesso mais simples, fornecendo acesso a qualquer sistema de biblioteca compartilhada dinamicamente sem usar JNI. É a biblioteca que vem no arquivo jar do JNA que a torna capaz de lidar com o carregamento de bibliotecas dinâmicas, chamada de funções, definição da estrutura e conversão de tipos, etc. para que todos os passos necessários associados à manipulação de C/C++ para fazer a ligação entre Java e código nativo sejam feita de forma simples (JNA, 2011).

2.2.1 Recursos JNA

Segundo a documentação (JNA, 2011), o JNA oferece dentre outros os seguintes recursos:

- Mapeamento automático de Java para funções nativas, com mapeamentos simples para todos os tipos de dados primitivos;
- Executa na maioria das plataformas que suportam Java;
- Conversão automática entre C e *strings* Java, com codificação/decodificação personalizável;
- Ponteiros de função, (*callbacks* de código nativo para Java) como argumentos e/ou membros de uma *struct*;
- Auto geração de *proxies* Java para ponteiros função nativa;
- Estruturas aninhadas e *arrays*;
- Suporte nativo (32 ou 64 bits, conforme apropriado);
- Aplicações demo;
- Suportado em JVMs 1.4 ou posterior;
- Personalizável de *marshalling/unmarshalling* (argumento e conversões valor de retorno);
- Mapeamento customizável de métodos Java ao nome função nativa e invocação personalizável para simular macros função C;

- Tipo de segurança para ponteiros nativos;
- Mapeamento otimizado direto para aplicações de alto desempenho.

2.2.2 Mapeamento de biblioteca

Quando se determina como compartilhar bibliotecas que detém os métodos que se necessita acessar, cria-se a classe correspondente para essa biblioteca. Por exemplo: O mapeamento para DPLOTLIB.dll em si seria parecido com um dos seguintes:

```
	// Alternativa 1: mapeameto pela interface carregada dinamicamente
	public interface DPLOTLIB extends Library {
		DPLOTLIB INSTANCE = (DPLOTLIB)Native.loadLibrary("dplotlib",
DPLOTLIB.class);
	}


	// Alternativa 2: mapeamento direto
	public class DPLOTLIB {
		static {
			Native.register("dplotlib");
		    }
	}
```

Quadro 1 - Mapeamento

A string passada para o método *Native.loadLibrary(String, Class)* (ou *NativeLibrary.getInstance(String))* é o nome do arquivo da biblioteca compartilhada. Na Tabela 1 é possível ver alguns exemplos de mapeamento de nome de algumas bibliotecas.

Sistema Operacional	**Nome de biblioteca**	**String**
Windows	user32.dll	user32
Linux	libX11.so	X11
Mac OS X	libm.dylib	m
Mac OS X Framework	/System/Library/Frameworks/Carbon.framework/Carbon	Carbon
Any Platform	<current process>	null

Tabela 1 - Bibliotecas nativas
Fonte: (JNA, 2011)

Qualquer biblioteca nativa com um único sistema de arquivos é representada por uma única instância do *NativeLibrary* e obtidos através *NativeLibrary.getInstance(String).* A biblioteca nativa será descarregada quando não for mais referenciado por qualquer código Java.

Se o nome da biblioteca é nulo, seus mapeamentos serão aplicáveis ao processo atual, em vez de uma biblioteca carregada separadamente. Isso pode ajudar a evitar conflitos, se existem várias versões incompatíveis de uma biblioteca disponível.

O caminho de pesquisa para carregar as bibliotecas nativa pode ser modificado por meio do *jna.library.path* e algumas propriedades.(JNA, 2011)

2.2.3 Mapeamento da função

Os nomes das funções são mapeados diretamente de seu nome da interface Java para o nome exportado pela biblioteca nativa. É possível renomear os métodos Java conforme as convenções de codificação Java, fornecendo uma entrada (*Library.OPTION_FUNCTION_MAPPER/FunctionMapper*) nas opções do mapa passado para *Native.loadLibrary()* que mapeia os nomes Java para os nomes nativos. Enquanto isso mantem-se o código Java um pouco mais limpo, o mapeamento de nomes adicionais podem tornar um pouco menos óbvia as funções nativas que estão sendo chamados (JNA 2011).

Uma instância da classe é obtida através da instância *NativeLibrary* correspondente à biblioteca nativa. Esta instância da função trata de gerenciar os paramentos e delegar para a função nativa (JNA 2011).

2.2.4 Mapeamento de tipos

2.2.4.1 Marshalling/Unmarshalling (Java/Native Tipos de conversão)

Tipos Java devem ser escolhidos para coincidir com tipos nativos do mesmo tamanho. Seguem-se os tipos suportados pela biblioteca JNA, em comparação com C.

- *Arrays* Java de tipo primitivo pode ser definido por *buffer*, a fim de aceder a um subconjunto da matriz (mudando o tamanho efetivo).
- *Arrays* Java de tipo primitivo são válidos apenas para utilização no âmbito de uma única chamada.

- Matrizes primitivas e estruturas como membros de uma estrutura são sobrepostas na memória estrutura pai.
- *Bitfields* (valores em bit) deve ser manualmente empacotado em um tipo inteiro.

Todos os outros tipos devem, eventualmente, ser convertido em um dos tipos correspondentes. Métodos com argumentos ou valores de retorno de tipos diferentes desses deve usar tipos decorrentes *NativeMapped* ou o fornecer informações do tipo de conversão para os tipos sem suporte (JNA 2011).

2.2.4.2 Arrays Primitivos

Arrays primitivos Java pode ser usado sempre que uma matriz nativa primitiva é utilizada. Quaisquer alterações feitas pelo código nativo para uma matriz durante uma chamada de função será refletido na matriz Java. Se o código nativo irá utilizar a matriz fora da chamada de função, onde a matriz é fornecida, utiliza-se no lugar *Memory* ou *Buffer* (JNA 2011).

2.2.4.3 Ponteiros

Ponteiros podem ser usados como um tipo opaco a partir do qual outros tipos de dados podem ser extraídos. O tipo *Pointer* é um recuo razoável para qualquer tipo de ponteiro base (incluindo *arrays*). O usuário geralmente não é permitido construir um *Pointer* (JNA 2011).

2.2.4.4 Buffer/ Memory Blocks

Utilizam-se matrizes para representar *buffers* de tipos primitivos passando para uma função para uso apenas durante a invocação da função. Um método nativo não pode retornar uma matriz de Java, já que não há maneira canônica para indicar a duração previsível da matriz retornada. Em vez disso, utiliza-se um dos métodos de matriz de acesso na classe *Pointer*, fornecendo o comprimento da matriz retornada (JNA 2011).

Buffers também podem ser usados como um argumento de entrada para *buffer de memória; buffers* de *bytes* direto muitas vezes podem oferecer um desempenho muito melhor sobre matrizes primitivas. Um ponteiro fornecido pelo código nativo pode ser convertido para um *buffer* chamando *Pointer.getByteBuffer(long, long)* (JNA 2011).

2.2.4.5 Callbacks (Function Pointers)

JNA suporta fornecimento de *callbacks* Java para código nativo. Para utilizar é necessário definir uma interface que estende a interface *Callback*, e definir um único método *callback* com uma assinatura que corresponde o ponteiro de função exigida pelo código nativo. O nome do método pode ser algo diferente de "*callback*" somente se houver apenas um único método na interface que estende ou a classe que implementa *Callback*. Os argumentos e valor de retorno seguem as mesmas regras que para uma chamada de função direta (JNA 2011).

2.2.4.6 Estruturas

A *Structure* Java representa uma *struct* nativa. Por padrão, esse tipo é tratado como um ponteiro para estrutura *(struct* *) no lado nativo quando usado como um parâmetro ou valor de retorno. Quando usado como um campo de estrutura, a estrutura é interpretada como por valor. Para forçar a interpretação complementar, a marcação de interfaces de *Structure.ByValue* e *Structure.ByReference* são fornecidos (JNA 2011).

2.2.5 Mapeamento de Invocação

Às vezes, funções nativas existem apenas como macros C de pré-processador ou como funções *inline*. Caso necessário fazer mais do que simplesmente mudar o nome da função invocada (que pode ser tratada através de mapeamento de função), uma *InvocationMapper* permite reconfigurar arbitrariamente a invocação da função, inclusive mudando o nome do método e reordenação, adição ou remoção de argumentos (JNA 2011).

2.2.6 Dados da Biblioteca Global

O método *NativeLibrary.getGlobalVariableAddress(java.lang.String)* pode ser usado para obter o endereço de variáveis globais como um ponteiro. Métodos de ponteiro podem então ser usados para ler ou escrever o valor apropriado para o tipo de variável (JNA 2011).

2.2.7 Um Proxy para a DLL

Segundo DASGUPTA (2009) o JNA utiliza o padrão *proxy* para esconder a complexidade da integração de código nativo. Ele fornece um *factory method* que os programas Java usam para obter um objeto *proxy* para uma DLL. Os programas podem então invocar as funções da DLL chamando métodos correspondentes do objeto *proxy*. O diagrama de sequência na Figura 2 mostra a criação e o uso de um objeto de *proxy*.

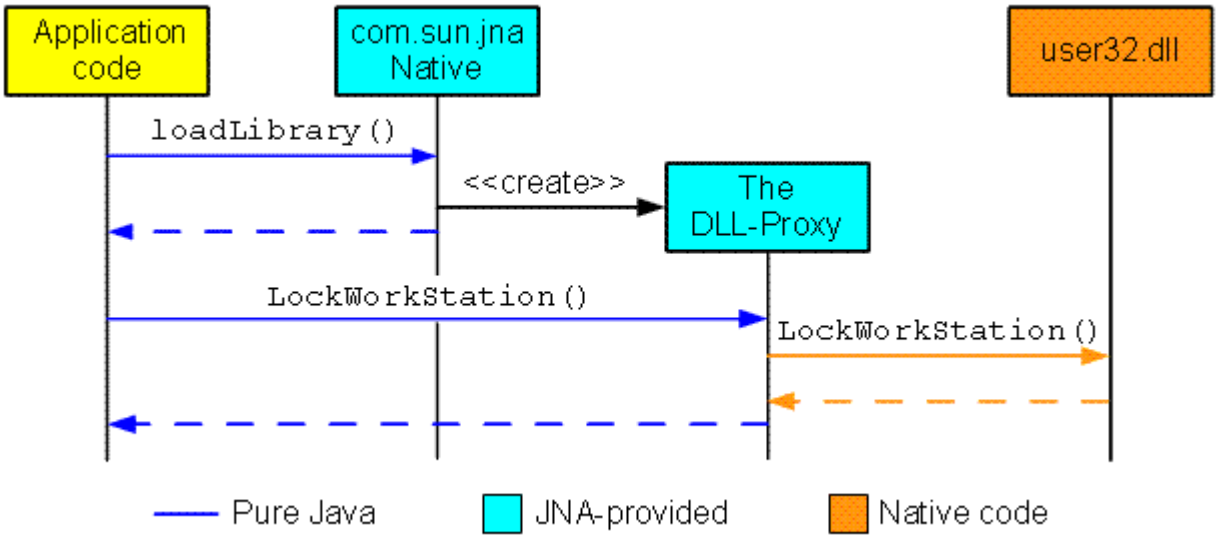

Figura 2 - Criação de um objeto *proxy* Java para uma DLL
Fonte: DASGUPTA, 2009

JNA é responsável por todos os aspectos em tempo de execução, mas requer a ajuda do programador para criar a classe *proxy* Java. Assim, a primeira parte de código necessário é uma interface Java com definições de métodos que correspondem às funções da DLL. Para utilizar corretamente JNA em tempo de execução, a interface deve estender *com.sun.jna.Library (*DASGUPTA, 2009). A interface *proxy* precisa conter declarações para os métodos que a aplicação irá utilizar.

2.3 COMPARATIVO JNI X JNA

No capítulo seguinte serão demonstrados exemplos simples de implementações das duas tecnologias.

2.3.1 Passos para implementação JNI:

- Declarar métodos nativos: definir métodos que permitem que os intermediários possam chamar funções nativas.
- Gerar o cabeçalho C/C++: gerar o arquivo de cabeçalho correspondente às funções nativas.
- Implementar código nativo: Implementação de métodos declarados no cabeçalho do arquivo.
- Compilar e gerar a biblioteca nativa: a construção da DLL cuja funcionalidade é fornecer os serviços definidos pelo cabeçalho.
- Carregar a biblioteca nativa: carregar a biblioteca no programa Java.

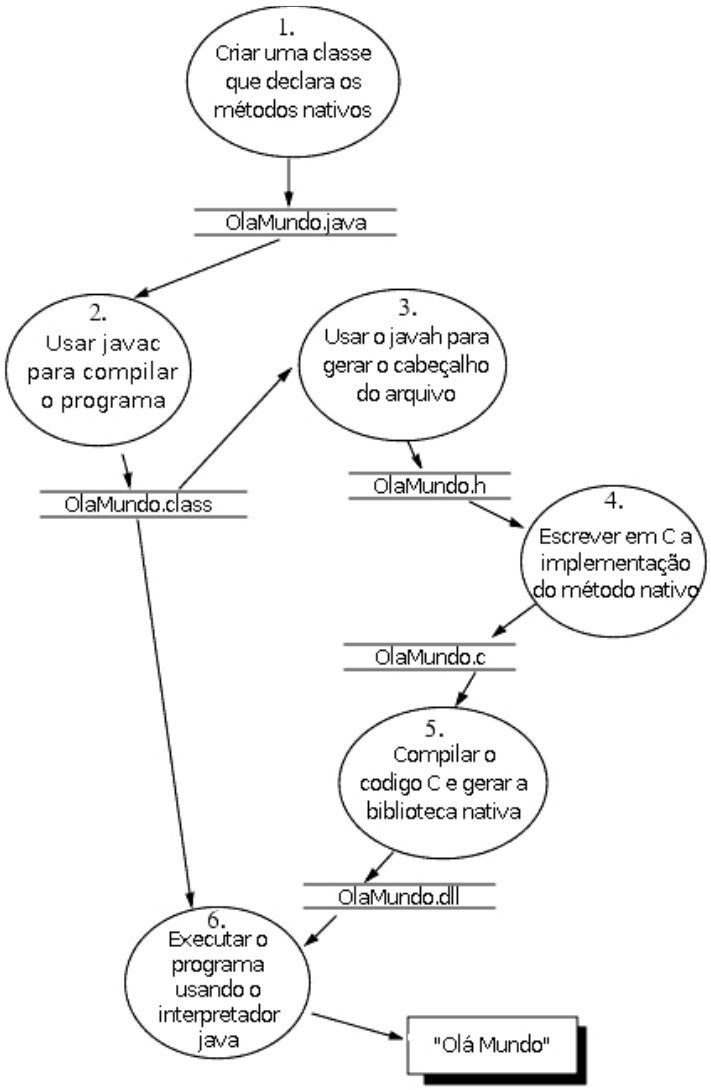

Figura 3 - Passos para Escrever e Rodar o exemplo "OlaMundo"
Fonte: JNI, 2011

2.3.2 Exemplo JNI

O exemplo definido inicia sua implemetação com a declaração dos métodos nativos (passo 1 da Figura 3), ele possui a classe com o nome `OlaMundo` e contém apenas um método, o `imprimir`.

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

Quadro 2 - Declaração do método nativo

A definição da classe `OlaMundo` começa com a declaração do método `imprimir`, seguido por um método main que faz a instância da classe `OlaMundo` e invoca o método nativo `imprimir`. A ultima parte da classe é o inicializador estático responsável por carregar a biblioteca nativa que contém a implementação do método nativo `imprimir`.

Há duas diferenças entre a declaração de um método nativo e os métodos regulares na linguagem de programação Java. A declaração do método nativo deve conter o modificador `native`. O modificador `native` indica que esse método é implementado em outra linguagem. Além disso, a declaração do método é terminada com ponto e virgula, o símbolo finalizador da declaração “()”, porque não há implementação para métodos nativos na classe, ele estão implementados em um arquivo separado (JNI, 2011).

Antes do método nativo `imprimir` poder ser chamado, a biblioteca que o implementa deve ser carregada, no exemplo a biblioteca nativa está sendo carregada no inicializador estático da classe `OlaMundo`. A JVM executa o inicializador estático antes de invocar qualquer método na classe `OlaMundo` garantindo assim que a biblioteca nativa será carregada antes do método nativo `imprimir` ser chamado.

Após definido um método principal `main`, para ser capaz de executar a classe `OlaMundo`. `OlaMundo.main` chama o método nativo `imprimir` da mesma forma como seria chamar um método normal.

`System.loadLibrary` leva um nome da biblioteca, localiza a biblioteca nativa que corresponde a esse nome, e carrega a biblioteca nativa para o aplicativo (JNI, 2011).

Depois de ter definido a classe `OlaMundo`, salve o código fonte em um arquivo chamado `OlaMundo.java`. Em seguida, compile o arquivo de origem usando o compilador `javac` que vem com o JDK ou Java 2 SDK (passo 2 da Figura 3). Na linha de comando rodar:

```
javac OlaMundo.java
```

Quadro 3 – Comando para compilar a classe Java

Este comando irá gerar um arquivo compilado no diretório chamado `OlaMundo.class`.

Em seguida, é utilizado a ferramenta `javah` para gerar um arquivo de cabeçalho JNI que é útil na implementação do método nativo em C (passo 3 da Figura 3).

```
javah-jni OlaMundo
```

Quadro 4 - Comando para gerar o cabeçalho JNI

O nome do arquivo de cabeçalho é o nome da classe com um "`.h`" acrescentada ao final do mesmo. O comando acima gera um arquivo chamado `OlaMundo.h`. A parte mais importante do arquivo de cabeçalho é o protótipo da função `Java_OlaMundo_imprimir`, que é a função C que implementa o método `OlaMundo.imprimir`:

```
JNIEXPORT void JNICALL
Java_OlaMundo_imprimir(JNIEnv *, jobject);
```

Quadro 5 - Conteúdo do cabeçalho JNI

Nota-se que a implementação do método nativo C, aceita dois argumentos mesmo que a declaração correspondente do método nativo não aceite argumentos. O primeiro argumento para cada implementação do método nativo é uma interface de ponteiro `JNIEnv`. O segundo argumento é uma referência para o objeto `OlaMundo` em si (como o tipo de ponteiro "*this*" em Java ou C) (JNI, 2011).

O arquivo de cabeçalho JNI gerado por `javah` ajuda a escrever as implementações C ou C++ para o método nativo. A função a ser escrita deve seguir o

protótipo especificado no arquivo de cabeçalho gerado. É possível implementar o método `OlaMundo.imprimir` em um arquivo C `OlaMundo.c` da seguinte forma (passo 4 da Figura 3):

```
#include <jni.h>
#include <stdio.h>
#include "OlaMundo.h"

JNIEXPORT void JNICALL
Java_HelloWorld_print(JNIEnv *env, jobject obj)
{
    printf("Olá Mundo!\n");
    return;
}
```

Quadro 6 - Implementação C

A implementação deste método nativo usa a função `printf` para exibir a *string* `"Olá Mundo!"` e depois retorna. Como mencionado anteriormente, ambos os argumentos, o ponteiro `JNIEnv` e a referência ao objeto, são ignorados.

O programa C inclui três arquivos de cabeçalho:

- `jni.h` - Este arquivo contém informações do cabeçalho do código nativo necessário para chamar funções JNI. Ao escrever métodos nativos, deve sempre incluir este arquivo em seus arquivos de origem C ou C++.
- `stdio.h` - O trecho de código acima também inclui `stdio.h` porque usa a função `printf`.
- `OlaMundo.h` - O arquivo de cabeçalho gerado usando `javah`. Ele inclui o protótipo C/C++ para a função `Java_OlaMundo_imprimir`.

Após todo o código C necessário estar escrito, é necessário compilar OlaMundo.c e construir esta biblioteca nativa (passo 5 da Figura 3). Cada sistema operacional oferece uma maneira diferente de construir bibliotecas nativas. No *Win32*, o quadro 7 constrói uma biblioteca de vínculo dinâmico (DLL) `OlaMundo.dll` usando o compilador Microsoft® Visual C++[8]:

```
cl -Ic:\java\include -Ic:\java\include\win32 -MD -LD
OlaMundo.c -FeOlaMundo.dll
```

Quadro 7 - Comando para construir a biblioteca nativa

[8] Microsoft Visual C++ é um ambiente de desenvolvimento integrado(IDE), produto comercial da Microsoft para linguagem de programação C, C++ e C++/ CLI. Tem ferramentas para desenvolvimento e depuração de código C++.

A opção `-MD` garante que `OlaMundo.dll` esteja ligada com a biblioteca multitarefa C *Win32*. A opção `-LD` instrui o compilador C para gerar uma DLL em vez de um executável *Win32*. É necessário por a DLL no *path* que refletem a configuração da máquina em questão (JNI, 2011).

Após ter os dois componentes prontos para executar o programa (passo 6 da Figura 3). A classe (`OlaMundo.class`) chama um método nativo, e a biblioteca nativa (`OlaMUndo.dll`) implementa o método nativo. Por ter o método principal a classe `OlaMundo` é possivel executar o programa *Win32* como segue:

```
java HelloWorld
```

Quadro 8 - Comando para executar a classe HelloWorld

Será exibido o seguinte resultado:

```
Olá Mundo!
```

Quadro 9 - Resultado da execução da classe HelloWorld

2.3.3 Sobre o JNI:

Para uma chamada de método simples, é obrigatório seguir um protocolo em cinco etapas:

- Declarar métodos nativos;
- Gerar cabeçalho C/C++;
- Implementar código nativo;
- Compilar e gerar a biblioteca nativa;
- Carregar a biblioteca nativa.

Além disso, deve também considerar a logística:

- Instalação de um compilador C/C++;
- Ter um ambiente de desenvolvimento para C/C++;
- Conhecimento de uma linguagem diferente de Java ao escrever o código acima não é muito complexo.

2.3.4 Passos para implementação JNA

Pelo fato da biblioteca JNA não ser nativa na JVM, é necessário possuir a biblioteca `jna.jar` e adiciona-la ao projeto. Ela se encontra disponível para download no seguinte endereço: https://github.com/twall/jna/raw/3.3.0/jnalib/dist/jna.jar. Após a adição segue-se os seguintes passos:

- Declarar a interface de mapeamento;
- Instanciar a interface dinamicamente;

2.3.5 EXEMPLO JNA

O programa definido para o exemplo será o mesmo com a implementação JNI. Ele carrega a implementação da biblioteca padrão C, e utiliza para chamar a função `printf`.

No Quadro 10 a declaração da interface de mapeamento, ela deve estender a interface `com.sun.jna.Library`. Em tempo de execução essa interface será recuperada. Essa instancia será automaticamente vinculada a bilbioteca nativa e será usada para fazer chamadas diretas a aos métodos nativos. A biblioteca JNA responsabiliza-se por todas as conversões de tipo e por procurar e chamar as funções nativas dinamicamente conforme definido no método Java, assim sendo, elimina a necessidade de escrever qualquer linha de código nativo.

```
import com.sun.jna.Library;
import com.sun.jna.Native;
import com.sun.jna.Platform;

public interface CLibrary extends Library {
   void printf(String format, Object... args);
}

public static void main(String[] args) {
       CLibrary instancia = (CLibrary) Native.loadLibrary(null,
       CLibrary.class);
       instancia.printf("Olá Mundo!");
       for (int i = 0; i < args.length; i++) {
           instancia.printf("Argument : ", i, args[i]);
           }
         }
}
```

Quadro 10 - Declaração da interface de mapeamento

Após ter a declaração da interface basta apenas carregar dinamicamente a biblioteca, no método `main`, é carregado com o `Native.LoadLibrary(CLibrary.class)`, depois de carregado o objeto `instancia` pode chamar o método passando os parametros, no caso uma *string*, após a execução da classe acima será impresso no console java:

```
Olá Mundo!
```

Quadro 11 - Saída do método implementado na biblioteca nativa

2.3.6 Sobre o JNA

Para uma chamada de método simples via JNA, é necessário seguir duas etapas:

- Declarar os métodos nativos em uma interface;
- Instanciar dinamicamente a interface.

Em logística:

- Não há necessidade de instalar um compilador C/C++;
- Não há necessidade de ter um ambiente de desenvolvimento para C/C++;
- Não precisa saber em outra linguagem.

2.4 BIBLIOTECA NATIVA

A biblioteca DPLOT JR é uma extensão do software DPLOT destinado a manipular dados complexos, gerando uma visualização em até três dimensões. O software é de propriedade da companhia HYDESOFT, porém a biblioteca é livre para ser utilizada pelos desenvolvedores (HYDESOFT, 2002).

As funções da DLL DPLOTLIB podem ser chamadas a partir de qualquer linguagem de programação que tenha suporte a chamada a procedimentos externos em DLL's. Por exemplo, programas com código fonte em C, C#, FORTRAN, *Visual Basic*, *Power Basic*, *FreeBASIC* e VB.NET já possuem implementação pronta. Em outras linguagem é necessário adequar-se conforme suas características para fazer a chamada aos procedimentos contidos na biblioteca (HYDESOFT, 2002).

Para cada compilador citado estão inclusos procedimentos capazes de gerar, dentre outras:

- Gráfico com dimensões XY com duas curvas: seno(πx) e cosseno(πx) onde x = 0 para 4;
- Uma gráfico de superfície em três dimensões: z = seno(x)*cosseno(y) onde -3 < x < 3 e -2 < y < 2 dispostos em um *grid* retangular;

2.4.1 Tipos de dados

As funções *DPlot_Plot, DPlot_PlotBitmap, DPlot_PlotToRect* e *DPlot_AddData*, têm como parâmetros de entrada *arrays* de precisão simples e números de ponto flutuante. Em C# e C, estes parâmetros são matrizes do tipo *float*. O DPLOT armazena os dados em matrizes de dupla precisão, assim a precisão dos dados passados para DPLOT usando estas funções serão preservados. Para a maioria das aplicações de precisão simples é suficiente.

Segundo a documentação (HYDESOFT, 2002) que descreve a lista de argumentos das funções DPLOTLIB usando a sintaxe C e tipos de dados, os tipos de dados equivalentes em outras línguas são mostrados na Tabela 2.

Biblioteca DPLOT	Equivalente
**prefixo*	Indica que o endereço do argumento, ao invés do próprio argumento, é passado para a função. Em C# isso é equivalente a um prefixo *ref*.
int	Inteiro assinado de 4 bytes.
float	Ponto flutuante de precisão simples com tamanho de 4-bytes.
double	Número com ponto flutuante de dupla precisão com tamanho de 8 bytes.
DWORD	Inteiro sem assinatura de 4-bytes.
HWND	Identificador para uma janela, inteiro de 4 bytes.
HBITMAP	Identificador para um bitmap, inteiro de 4 bytes.
HENHMETAFILE	Identificador para um metarquivo avançado, inteiro de 4 bytes.
LPSTR	Ponteiro para uma string terminada em *null* de 8-bit do Windows caracteres (ANSI).
DPLOT	A estrutura DPLOT contém informações do formato DPLOT para o gráfico: número de curvas, número de pontos em cada curva, linha e símbolo estilos usados, etc. Esta estrutura é definida no cabeçalho/include arquivo que acompanha a fonte de vários exemplos.

Tabela 2: Tipos de dados nativos equivalentes em outras linguagens

2.4.1.1 Estrutura DPLOT

A estrutura contém informações do formato DPLOT de um gráfico: número de curvas, número de pontos em cada curva, linha e estilos de símbolos usados, é usado em chamadas para as funções: *DPlot_Plot, DPlot_PlotToRect* e *DPlot_PlotBitmap*. Mostrado no Anexo A usando a sintaxe de C padrão.

2.4.2 Principais funções da biblioteca DPLOT Jr

A biblioteca possui várias funções, a seguir serão descritas algumas das principais, imprescindíveis para o uso da biblioteca.

2.4.2.1 DPLOT_PLOT

Esta função permite enviar dados para DPLOT e criar um plano completo com uma única chamada de função. É possivel criar qualquer tipo de gráfico atualmente

suportados pelo DPLOT, incluindo gráficos XY, gráficos de contorno (2D ou 3D) de dados organizados em uma grade retangular, gráficos de contorno (2D ou 3D) de forma aleatória com espaçamento x, y, z e gráficos de pontos de dados de uma dimensão. Declaração da função:

```
int DPlot_Plot(DPLOT *DPlot, float *array1, float *array2, LPSTR commands);
```

Parâmetros:

- `*DPlot`: Endereço de uma estrutura DPLOT. A estrutura contém informações do formato DPLOT para o gráfico: número de curvas, número de pontos em cada curva, linha e símbolo estilos usados, etc.
- `*array1`: Endereço da matriz X de gráficos XY ou uma matriz de 4 elementos usados para as extensões de um gráfico de contorno em uma grade retangular. Não utilizado para gráficos de contorno de pontos aleatórios, gráficos de dispersão 3D.
- `*array2`: Endereço da matriz Y para gráficos XY, Z matriz para gráficos de contorno em uma grade retangular, X, Y, Z (na ordem x (0), y (0), z (0), x (1), y (1), z (1)) para gráficos de contorno de pontos aleatoriamente espaçados e gráficos de dispersão 3D.
- `commands`: Cadeia de caracteres com nenhum ou vários comandos DPlot.

Valores de retorno:

- >0 (maior que zero): Sucesso. O valor de retorno neste caso é o índice do documento criado pela chamada. Este valor será entre 1 e 32. O índice documento pode ser usado em chamadas subsequentes para *DPlot_Command* e *DPlot_Request.*
- 0(zero): Erro genérico tentando se comunicar com DPLOT. Este na maioria das vezes indica que DPLOT está ocupado, por exemplo, uma caixa de diálogo modal estiver aberta, ou que um ou mais comandos no parâmetro comandos é inválido. (Neste último caso, DPLOT irá mostrar a mensagem "Erro ao processar este comando" em uma caixa de dialogo.
- -1 (menos um): Não foi possível encontrar/executar o DPLOT.

- -2 (menos dois): Não foi possível estabelecer uma conexão DDE com DPLOT. Este erro não deve ocorrer.
- -3 (menos três): Tipo inválido *DataFormat* especificado na estrutura DPLOT. Se a estrutura foi preenchida corretamente e obter este erro, é mais provável uma indicação de que a estrutura não está sendo passada corretamente.
- -4 (menos quatro): Número da versão sem suporte na estrutura DPLOT. Os valores válidos são atualmente 2 e 3.

2.4.2.2 DPLOT_PLOTBITMAP

Retorna um identificador para um bitmap HBITMAP. É responsabilidade do aplicativo de chamada para então desenhar o bitmap e excluí-lo quando não for mais necessário. Declaração da função:

```
HBITMAP DPlot_PlotBitmap(DPLOT *DPlot, float *array1, float *array2,
LPSTR commands,int cx, int cy);
```

Parâmetros:

- `*DPlot`: Endereço de uma estrutura DPLOT. A estrutura contém informações do formato DPLOT para o gráfico: número de curvas, número de pontos em cada curva, linha e símbolo estilos usados, etc.
- `*array1`: Endereço da matriz X de gráficos XY ou uma matriz de 4 elementos usados para as extensões de um gráfico de contorno em uma grade retangular. Não utilizado para gráficos de contorno de pontos aleatórios.
- `*array2`: Endereço da matriz Y para gráficos XY, Z matriz para gráficos de contorno em uma grade retangular, X, Y, Z, (na ordem x (0), y (0), z (0), x (1), y (1), z (1),... etc) para gráficos de contorno de pontos aleatoriamente espaçados.
- `commands`: Cadeia de caracteres com nenhum ou vários comandos DPlot.
- `cx, cy`: Largura e altura do bitmap solicitada em pixels.

Valores de retorno:

- 0 (zero): Erro genérico tentando se comunicar com DPlot. Este na maioria das vezes indica que DPlot está ocupado, por exemplo, uma caixa de diálogo modal estiver aberta, ou que um ou mais comandos no parâmetro comandos é inválido.

(Neste último caso, DPLOT irá mostrar a mensagem "Erro ao processar este comando" em uma caixa de dialogo.

- <>0 (diferente de zero): Identificador para um dispositivo bitmap dependente. Esta imagem pode ser desenhado em seu aplicativo com o Windows API funções BitBlt e/ou StretchBlt.

Observações: Se DPlot não está sendo executado quando *DPlot_PlotBitmap* for chamada, esta função irá fechar DPLOT imediatamente antes de retornar para o chamador. Se a aplicação está fazendo várias chamadas para funções *DPlot_PlotBitmap* primeiramente deve-se começar com *DPlot_Start*.

2.4.2.3 DPLOT_GETBITMAP

Esta função recupera um identificador para um *bitmap* HBITMAP de um gráfico determinado, normalmente, mas não necessariamente, um gráfico gerado pelo aplicativo de chamada. Tal como acontece com *DPlot_PlotBitmap*, é responsabilidade do aplicativo de chamada para desenhar o *bitmap* e excluí-lo quando não for mais necessário. Declaração da função:

```
HBITMAP DPlot_GetBitmap(int DocNum, int cx, int cy);
```

Parâmetros:

- `DocNum`: Índice do documento para o documento que você quer uma imagem *bitmap*. Na prática, isso geralmente (embora não necessariamente) é o valor de retorno de uma chamada para DPlot_Plot.
- `cx, cy`: Largura e altura do bitmap, solicitada em pixels.

Valores de retorno:

- 0 (zero): Erro genérico tentando se comunicar com DPlot. Este na maioria das vezes indica que DPlot está ocupado, por exemplo, uma caixa de diálogo modal estiver aberta.
- <>0 (diferente de 0): Identificador para um dispositivo *bitmap* dependente. Esta imagem pode ser desenhado em seu aplicativo com o *Windows* API funções *BitBlt* e/ou *StretchBlt* .

2.4.2.4 DPLOT_COMMAND

Permite-lhe enviar um ou mais comandos para DPLOT. Isso pode ser útil em aplicações em tempo real ou em outras situações em que o conjunto completo de dados para traçar não é conhecido inicialmente. Declaração da função:

```
int DPlot_Command(int DocNum, LPSTR commands);
```

Parametros:

- `DocNum`: Índice do documento retornado por uma chamada anterior para *DPlot_Plot*, ou utilize 0 para enviar a sequência de comandos para a janela do documento ativo no momento. Este formulário (DocNum = 0) só deve ser usado para abrir uma nova janela de documento.
- `commands`: Cadeia de caracteres com nenhum ou vários comandos DPlot. Na versão DPlot Jr, alguns comandos não são suportados pelo DPlot Jr. A cadeia de comando é limitado a 32.768 caracteres e deve ser terminada com *null*.

Valores de retorno:

- >0 (maior que 0): Sucesso. Para DocNum = 0, o valor de retorno é o índice do documento ativo (1-32). Para DocNum> 0, o sucesso é sempre indicado por um valor de retorno de 1.
- 0 (zero): Erro genérico tentando se comunicar com DPlot. Este na maioria das vezes indica que DPlot está ocupado, por exemplo, uma caixa de diálogo modal estiver aberta, ou que um ou mais comandos no parâmetro comandos é inválido. (Neste último caso, DPlot irá mostrar uma caixa de dialogo com "Erro ao processar este comando".
- -1 (menos um): Não foi possível encontrar/executar DPlot.
- -2 (menos dois): Não foi possível estabelecer uma conexão DDE com DPlot. Este erro não deve ocorrer.

Observações: *DPlot_Command* abre uma conversação DDE com DPLOT, envia a cadeia de comando, e depois termina a conversação DDE. Toda vez que uma conversa é aberta, DPLOT inicializa várias variáveis específicas.

O DPLOT Jr não suporta o conjunto completo de comandos incluídos com DPLOT. Em geral, DPLOT Jr não vai abrir um arquivo (sem comando *FileOpen*, por exemplo), nem vai executar muitas das funções de manipulação de dados incluídos na versão completa. Se estiver distribuindo DPLOT Jr com sua própria aplicação, deve sempre verificar o arquivo de ajuda para garantir que DPLOT Jr irá lidar com os comandos desejados.

2.4.2.5 DPLOT_SETERRORMETHOD

Permite controlar como as exceções são retornadas para o usuário/aplicação. Declaração da função:

```
void DPlot_SetErrorMethod(int method);
```

Parâmetros:

- `method`: Código que descreve o relatório de erro.

Valores de retorno:

- 0 (zero): Não há relatos de erro;
- 1 (um): Mensagens de erro relataram o uso de OutputDebugString.
- 2 (dois): Sistema modal de caixa de mensagem.

2.4.2.6 DPLOT_START

Inicia o DPLOT, opcionalmente, em segundo plano. A função também retorna um valor indicando se DPLOT já estava em execução. Declaração da função:

```
HWND DPlot_Start(int Hide, int *was_active);
```

Parâmetros:

- `Hide`: Se diferente de zero, DPLOT será invisível para o usuário. Este é o mesmo estado usado pelo *DPlot_PlotBitmap*, se o DPlot não estiver em execução, quando essas funções são chamadas. Se for 0, DPlot é iniciado normalmente (se ele não estiver em execução).

- `*was_active`: Endereço de um valor inteiro que recebe 1 se DPLOT já estava em execução e 0 caso não. Se esse parâmetro for definido como *null*, nenhum valor é retornado.

  Valores de retorno:

- 0 (zero): Não foi possível iniciar DPlot, ou o número de versão instalado no sistema do usuário está errado. Neste último caso, o usuário visualizará uma caixa de diálogo explicando o problema.
- >0 (maior que 0): Identificador de janela da janela principal da aplicação DPlot.

Observações: Não é obrigatório chamar *DPlot_Start*, uma vez que todas as rotinas que interagem com DPLOT irá iniciá-lo se ele não estiver sendo executado. No entanto, duas rotinas (*DPlot_PlotBitmap e DPlot_PlotToRect*) irão fechar o DPLOT quando terminar.

*DPlot*_Start primeiro verifica os critérios de versão mínima. A versão mínima pode ser definido com uma chamada para *DPlot_MinVersion*. Está função se não for chamado antes da primeira execução, a versão mínima exigida pela bilbioteca DPLOTLIB.DLL (2.0.0.3) é utilizada. Se DPlot Jr está instalado e atende a esse requisito de versão mínima, então DPlot Jr será usado, independentemente da versão completa do DPlot instalado ou em execução. Se nem DPlot Jr nem a versão completa do DPlot cumprir a exigência versão mínima, uma mensagem de erro semelhante à mostrada na figura 4 será apresentada.

Figura 4 - Caixa de Mensagem com informações de erro de versão
Fonte: HYDESOFT, 2009

Caso não queira visualizar esta mensagem de erro, deve-se primeiro chamar *DPlot_MinVersion*. E lidar com a situação dentro do seu próprio programa se essa função retornar 0.

2.4.2.7 DPLOT_STOP

Esta rotina deve ser usado somente se você começar O DPLOT com a função *DPlot_Start* e o parametro *"was_active"* ser retornado como 0, indicando que DPLOT foi iniciado pelo seu aplicativo. Declaração da função

```
void DPlot_Stop();
```

3 ESTUDO DE CASO

Para este experimento foi desenvolvido um protótipo que constitui em uma aplicação desktop java, com a API *Swing*[9]. A aplicação tem como propósito coletar alguns parâmetros e de acordo com os parâmetros e dados de coleta simular a aspersão de um pluviometro, utilizado em sistemas de irrigação de solo, a distribuição da água é representada pelo gráfico. Ela possui também o propósito utilizar algumas das funções disponíveis na biblioteca nativa DPLOT Jr para gerar gráficos. Para o acesso da biblioteca foi utilizado a tecnologia JNA ao invés de JNI, por se mostrar mais vantajosa no propósito deste estudo de caso.

3.1 SOFTWARES UTILIZADOS:

- Eclipse Galileo – Software de código livre desenvolvido em Java para construção de programas de computador, principalmente em linguagem Java. Disponível para download em http://www.eclipse.org/downloads.
- PgAdmin III 1.10.5. – Ferramenta administrativa para banco de dados PostreSql, inteiramente gratuito. Ele permite escrever SQL simples ou complexas e desenvolver banco de dados com as funções do PostgreSQL. Disponível para download em http://www.pgadmin.org/download.
- Para elaboração dos diagramas foi utilizado o Astah Community 6.4.1.

3.2 DESENVOLVIMENTO

O escopo do protótipo de testes, baseia-se na seguinte regra de negócios, o usuário registra um novo projeto, e dentro deste projeto o usuário insere informações. Com esses dados é possivel gerar gráficos, utilizando a biblioteca DPLOT Jr.

[9] API *Swing:* É um conjunto de classes que implementam um conjunto de componentes para a construção de interfaces gráficas (*GUIs*) e adicionando funcionalidade de uma interface visual rica e interatividade para aplicações Java. Os componentes Swing são implementados inteiramente na linguagem de programação Java.

O diagrama de classes foi definido a partir dos casos de uso representados na Figura 5, nele é possível visualizar os principais requisitos que a aplicação proposta deverá cumprir.

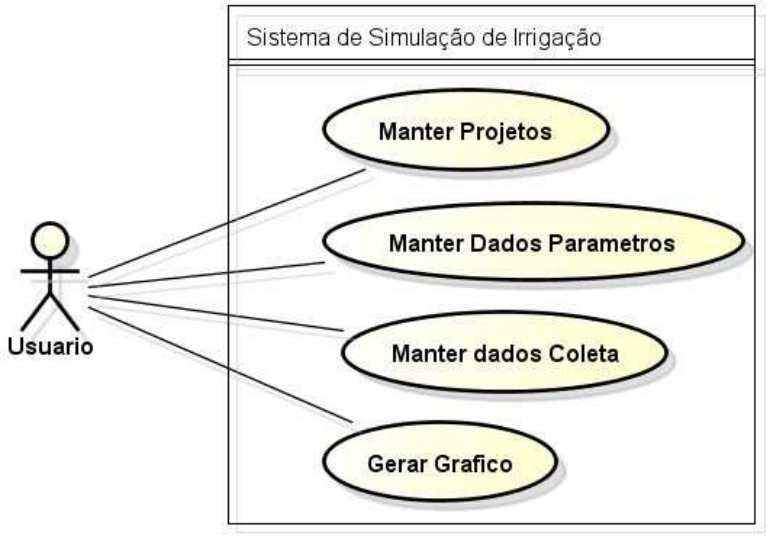

Figura 5 - Casos de Uso da aplicação
Fonte: autoria própria

A figura 6 representa o diagrama de casos de uso básicas que compõe a estrutura do software e suas relações.

Com a biblioteca JNA adicionada no projeto, pois será ela a responsável em intermediar a comunicação com a biblioteca nativa, seguiu-se para o próximo passo, que é a declaração da interface com as funções que irão ser utilizadas pela aplicação:

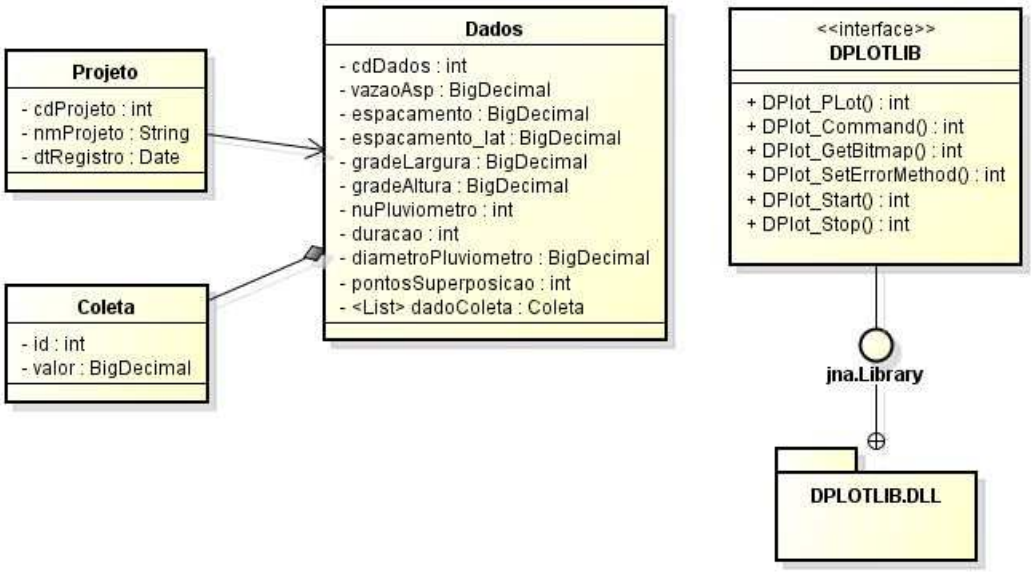

Figura 6 - Diagrama de Classe da aplicação
Fonte: autoria própria

```
package sima.jna;

import com.sun.jna.Library;
import com.sun.jna.platform.win32.WinDef.HBITMAP;
import com.sun.jna.platform.win32.WinDef.HWND;

public interface IDplotLib extends Library {

	int DPlot_Plot(DPLOT dPlot, FloatBuffer array1, FloatBuffer array2
			, String commands);
	int DPlot_Plot8(DPLOT dPlot, double array1, double array2, String
commands);
	int DPlot_Command(int DocNum, String commands);
	HBITMAP DPlot_GetBitmap(int DocNum, int cx, int cy);
	void DPlot_SetErrorMethod(int method);
	HWND DPlot_Start(int Hide, int was_active);
	void DPlot_Stop();
}
```

Quadro 12 - Interface IDplotLib

Para utilizar a interface *IDplotLib* em qualquer parte do código apenas é necessário carregar a biblioteca, o Quadro 12 demonstra como é feito.

```
//carrega a biblioteca que está em c:/windows/32
//ou passa o caminho completo
	System.setProperty("jna.library.path","C:\\Windows\\dplotlib");
	lib = (IDplotLib) Native.loadLibrary("dplotlib",IDplotLib.class);
```

Quadro 13 - Carregamento dinâmico da biblioteca nativa

Com a biblioteca carregada no objeto *lib* possui todos os métodos declarados na interface e pronto para ser utilizado pela aplicação, a Figura 7 demonstra os métodos do objeto.

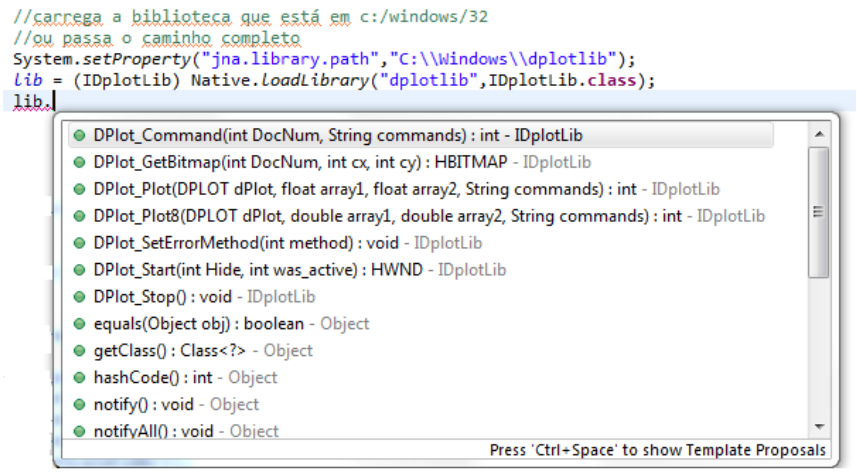

Figura 7 - Métodos disponíveis no objeto *lib*

Fonte: autoria própria

Com a biblioteca carregada é necessário realizar as configurações do gráfico como alimentar com dados e setar algumas configurações específicas da biblioteca nativa. O quadro 13 representa o método responsável para isso, ele é chamado ao abrir a tela de visualização do gráfico.

```
      private void gerarGrafico() {

                    dadosColeta = SimaPrincipal.iDadosColeta
                           .selecionarPorDados(SimaPrincipal.dadosMain);
                    dplot.setVersion(DPLOT_DDE_VERSION.version);
                    dplot.setHwnd(1);
                    dplot.setDataFormat(3);
                    dplot.setScaleCode(1);

                    StringBuilder cmd =  new StringBuilder();


      cmd.append("[Contour3D(1)][ContourGrid(1)][ContourAxes(1)]");
                    cmd.append("[ContourScales(1,1,1)]");

                    float[] floatArray = new float[dadosColeta.size()];

                    for (int i = 0; i < dadosColeta.size(); i++) {
                        Float f = dadosColeta.get(i).getColeta().floatValue();
                        floatArray[i] = (f != null ? f : Float.NaN);
                    }

                    FloatBuffer buff = FloatBuffer.wrap(floatArray);

                    int docNum = lib.DPlot_Plot(dplot, null, buff,
cmd.toString());

                    HBITMAP hBitMap = new HBITMAP();
                    hBitMap = lib.DPlot_GetBitmap(docNum, 600, 400);

                    grafico.setIcon((Icon) hBitMap);
             }
```

Quadro 14 - Método responsável por gerar o gráfico

O método acima carrega os dados em um *FloatBuffer* tipo de dado compatível com JNA, a estrutura DPLOT, está representada pelo objeto dplot, o objeto *lib* é o responsável pela execução da funções nativas. A saída do método acima é representada na Figura 8.

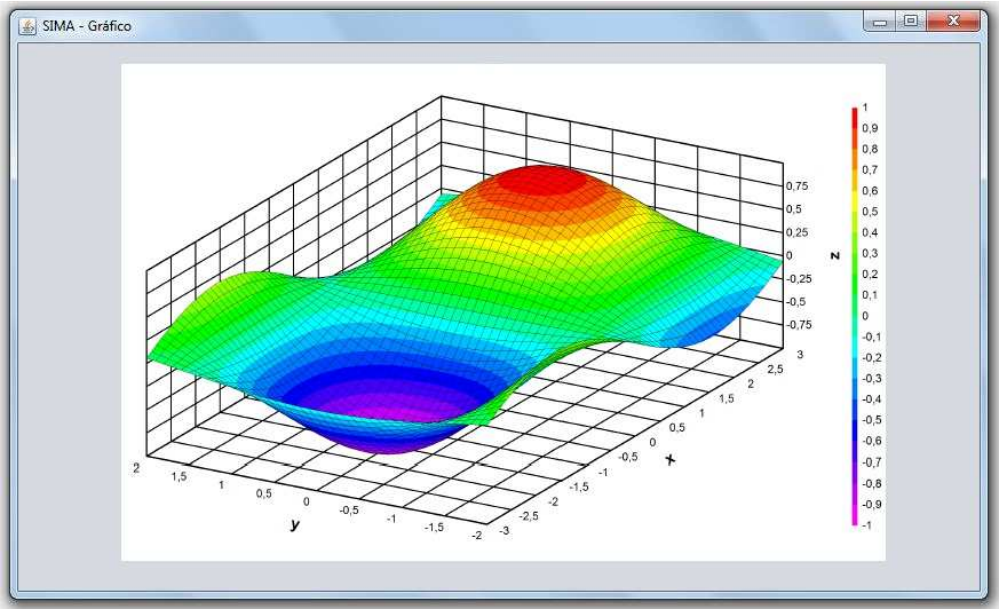

Figura 8 - Representação do gráfico gerado pela biblioteca nativa
Fonte: autoria própria

4 CONSIDERAÇÕES FINAIS

4.1 CONCLUSÃO

Com base na pesquisa bibliográfica, pode-se verificar que:

A solução JNI fornecida pelo JDK tem um nível de dificuldade maior por lidar com código C/C++. Necessita de instalação de ferramentas adicionais, porém oferece uma vantagem quando a biblioteca não é de terceiros, e sim implementada pelo programador. A solução JNA tem a vantagem de estar em uma camada de programação abstrata, o JNA implicitamente atua como uma ponte entre a biblioteca dinâmica e o código Java, mostrando-se favorável quando a biblioteca a ser usada já está implementada por terceiros.

Porém a escolha entre JNI e JNA deve ser feita de acordo com cada caso específico. Necessidades específicas podem ser desenvolvidas com uma solução baseada em JNI. Mas é inegável que JNA simplifica muito as chamadas para funções nativas.

Para o estudo de caso proposto o JNA foi a solução que demonstrou-se superior, foi utilizado e proporcionou um acesso simplificado e eficiente.

4.2 TRABALHOS FUTUROS

Como sugestão de trabalhos futuros fica a ideia de explorar melhor os recursos de acesso do JNI e também do JNA a códigos nativos. Pelo fato do JNA ser um projeto *open source*, é possível participar e contribuir.

5 REFERÊNCIAS BIBLIOGRÁFICAS

BARON. Mickael, MARTINI. Frédéric. Exécuter du code natif en Java: JNI VS JNA. 2008. Disponível em: http://mbaron.developpez.com/javase/jnijna/#LII-4. Acesso em: 10 de Outubro de 2011.

CODE SAMPLES AND APPS – Applets, Sun Developer Network (SDN). Disponível em: http://java.sun.com/applets/ Acessado em: 13 de Agosto de 2011.

DASGUPTA. SANJAY, Simplify Native Code Access with JNA. Novembro 2009. http://today.java.net/article/2009/11/11/simplify-native-code-access-jna

DDE. Dynamic Data Exchange. Disponível em: http://pt.wikipedia.org/wiki/Dynamic_Data_Exchange Acessado em: 13 de Agosto de 2011.

DLL. Dynamic Language Library. Disponível em: http://en.wikipedia.org/wiki/Dynamic-link_library Acessado em: 13 de Agosto de 2011.

GC - WIKIPEDIA. Garbage Collection. Disponível em: http://en.wikipedia.org/wiki/Garbage_collection_(computer_science) Acessado em: 24 de Outubro de 2011.

HYDESOFT COMPUTING. DPLOT Jr - Graph software for scientists e engineers. 2002. Disponível em: http://www.dplot.com/dplotjr_setup.exe. Acesso em: 12 de agosto de 2011.

JAVA NATIVE ACCESS (JNA). Java Native Access API Documentation. 2011. Disponível em: http://jna.java.net/javadoc/overview-summary.html. Acesso em: 20 de Setembro.

JAVA NATIVE INTERFACE (JNI). Programmers Guide and Specification. 1999. Disponível em: http://java.sun.com/docs/books/jni/download/jni.pdf Acesso em: 13 de Agosto de 2011.

ORACLE CORP. Oracle. Disponível em: www.oracle.com Acessado em: 13 de Agosto de 2011.

ANEXO A – Estrutura DPLOT

```
/* Structure used to send data to DPLOT via DDE */

#define DPLOT_DDE_VERSION 4
/* Version 2 of the DPLOT structure is obsolete and not recommended fo
r use unless it is
              imperative that your program be compatible with DPlot ver
sions older than 1.8.
   Version 3 of the DPLOT structure allows up to 100 curves and increa
ses the number of allowable
              characters in the legend from 40 to 80 characters each, n
umber of characters in
              curve labels from 5 to 40 characters each.
   Version 4 of the DPLOT structure adds a 4th title line and increase
s the number of allowable
              characters in the title lines and X, Y axis labels from 8
0 to 200 characters.
              Version 2.1 or later of DPlot or DPlot Jr is required.
*/

#if DPLOT_DDE_VERSION<3

#define MAXC 20

typedef struct tagDPLOT
{
    DWORD Version;           // Caller must set this to DPLOT_DDE_VERS
ION
    DWORD hwnd;              // handle of client application window
                             // (Use DWORD rather than HWND)
    DWORD DataFormat;        // XY pairs, DX and Y, etc.
    DWORD MaxCurves;         // maximum number of curves (must be <= 2
0)
                             // = NX for DataFormat = DATA_3D
                             //   ignore for DataFormat = DATA_3DR
    DWORD MaxPoints;         // maximum number of points/curve
                             // = NY for DataFormat = DATA_3D
                             // = 3 * number of points for DATA_3DR
    DWORD NumCurves;         // actual number of curves, always 1 for
                             //   DATA_3D or DATA_3DR
    DWORD Scale;             // scaling code (Linear, Log, etc.)
    float LegendX;           // left coord of legend, expressed as a r
atio
                             //    of plot size (0->1)
    float LegendY;           // top coord of legend
    DWORD NP[MAXC];          // actual number of points in each curve;
                             //   cannot exceed MaxPoints.

                        // For DATA_3DR files, return number of nodes i
n NP[0]
    DWORD LineType[MAXC];    // line types (see codes below)
    DWORD SymbolType[MAXC];  // symbol types (see codes below)
    DWORD SizeofExtraInfo;   // Extra information following X,Y data
    char  Legend[MAXC+1][40];// Legend[0] is the caption for the legen
d.
                             // Legend[n] is the legend for the n'th c
urve.
    char  Label[MAXC][5];    // Strings displayed beside the last data
 point
                             //   in a curve.
    char  Title[3][80];      // Three title lines.
```

```
    char  XAxis[80];          // X Axis label.
    char  YAxis[80];          // Y Axis label.
} DPLOT;

#else

#define MAXC 100

#if DPLOT_DDE_VERSION==3

typedef struct tagDPLOT
{
    DWORD Version;            // Caller must set this to DPLOT_DDE_VERS
ION
    DWORD hwnd;               // handle of client application window
                              // (Use DWORD rather than HWND)
    DWORD DataFormat;         // XY pairs, DX and Y, etc.
    DWORD MaxCurves;          // maximum number of curves (must be <= 1
00)
                              // = NX for DataFormat = DATA_3D
                              //   ignore for DataFormat = DATA_3DR
    DWORD MaxPoints;          // maximum number of points/curve
                              // = NY for DataFormat = DATA_3D
                              // = 3 * number of points for DATA_3DR
    DWORD NumCurves;          // actual number of curves, always 1 for
                              //   DATA_3D or DATA_3DR
    DWORD Scale;              // scaling code (Linear, Log, etc.)
    float LegendX;            // left coord of legend, expressed as a r
atio
                              //    of plot size (0->1)
    float LegendY;            // top coord of legend
    DWORD NP[MAXC];           // actual number of points in each curve;
                              //   cannot exceed MaxPoints.

                        // For DATA_3DR files, return number of nodes i
n NP[0]
    DWORD LineType[MAXC];     // line types (see codes below)
    DWORD SymbolType[MAXC];   // symbol types (see codes below)
    DWORD SizeofExtraInfo;    // Extra information following X,Y data
    char  Legend[MAXC+1][80];// Legend[0] is the caption for the legen
d.
                              // Legend[n] is the legend for the n'th c
urve.
    char  Label[MAXC][40];    // Strings displayed beside the last data
 point
                              //   in a curve.
    char  Title[3][80];       // Three title lines.
    char  XAxis[80];          // X Axis label.
    char  YAxis[80];          // Y Axis label.
} DPLOT;

#endif
#if DPLOT_DDE_VERSION==4

typedef struct tagDPLOT
{
    DWORD Version;            // Caller must set this to DPLOT_DDE_VERS
ION
    DWORD hwnd;               // handle of client application window
                              // (Use DWORD rather than HWND)
    DWORD DataFormat;         // XY pairs, DX and Y, etc.
    DWORD MaxCurves;          // maximum number of curves (must be <= 1
00)
```

```
                                // = NX for DataFormat = DATA_3D
                                //   ignore for DataFormat = DATA_3DR
    DWORD MaxPoints;            // maximum number of points/curve
                                // = NY for DataFormat = DATA_3D
                                // = 3 * number of points for DATA_3DR
    DWORD NumCurves;            // actual number of curves, always 1 for
                                //   DATA_3D or DATA_3DR
    DWORD Scale;                // scaling code (Linear, Log, etc.)
    float LegendX;              // left coord of legend, expressed as a r
atio
                                //    of plot size (0->1)
    float LegendY;              // top coord of legend
    DWORD NP[MAXC];             // actual number of points in each curve;
                                //   cannot exceed MaxPoints.

                          // For DATA_3DR files, return number of nodes i
n NP[0]
    DWORD LineType[MAXC];     // line types (see codes below)
    DWORD SymbolType[MAXC];   // symbol types (see codes below)
    DWORD SizeofExtraInfo;    // Extra information following X,Y data
    char  Legend[MAXC+1][80];// Legend[0] is the caption for the legen
d.
                                // Legend[n] is the legend for the n'th c
urve.
    char  Label[MAXC][40];    // Strings displayed beside the last data
 point
                                //   in a curve.
    char  Title[4][200];      // Four title lines.
    char  XAxis[200];         // X Axis label.
    char  YAxis[200];         // Y Axis label.
} DPLOT;
#endif
#endif

// DataFormat graph-type and data organization codes
#define DATA_XYXY         0     // One or more sets of X,Y data
#define DATA_DXY          1     // One or more X,Y curves. Constant sp
acing in X and same number of points in all curves.
#define DATA_XYYY         2     // One or more X,Y curves. All curves
have the same X values.
#define DATA_3D           3     // Z values on a rectangular grid
#define DATA_3DR          4     // Random X,Y,Z values
#define DATA_IMAGE        5     // Used only by DPlot -
 there's no way to SEND DPlot an image
#define DATA_1D           6     // One or more groups of Y values.
#define DATA_3DS          7     // 3D scatter plot
#define DATA_4D           8     // 4D surface plot
#define DATA_4DS          9     // 4D scatter plot
// Scale codes:
#define SCALE_LINEARX_LINEARY     1
#define SCALE_LINEARX_LOGY        2
#define SCALE_LOGX_LINEARY        3
#define SCALE_LOGX_LOGY           4
#define SCALE_TRIPARTITE          5
#define SCALE_LINEARX_PROBABILITY 6
#define SCALE_GRAINSIZE_DIST      7
#define SCALE_POLAR               8
#define SCALE_BARCHART            9
#define SCALE_LOGX_PROBABILITY   10
#define SCALE_PROBX_LINEARY      11
#define SCALE_PROBX_LOGY         12
#define SCALE_PROBX_PROBY        13
#define SCALE_TRIANGLE_PLOT      14
#define SCALE_N185               15
#define SCALE_MERCATOR           16
```

```
// Unit-
specific scaling codes ... combine with above values using OR operator
#define UNITS_DEFAULT                   0
#define UNITS_TRIPARTITE_INCHES      UNITS_DEFAULT   // Velocity (Y) i
n inches/sec
#define UNITS_TRIPARTITE_FEET        0x00000100L     //
  feet/sec
#define UNITS_TRIPARTITE_MILLIMETERS 0x00000200L     //
  mm/sec
#define UNITS_TRIPARTITE_CENTIMETERS 0x00000300L     //
  cm/sec
#define UNITS_TRIPARTITE_METERS      0x00000400L     //
  meters/sec
#define UNITS_GRAINSIZE_MILLIMETERS  UNITS_DEFAULT   // Grain sizes (X
) in mm
#define UNITS_GRAINSIZE_INCHES       0x00000100L     //
     inches
#define UNITS_POLAR_RADIANS          UNITS_DEFAULT   // Rotation (X) i
n radians
#define UNITS_POLAR_DEGREES          0x00000100L     //
  degrees
#define UNITS_USERDEFINED            0x00007F00L
// Line styles:
#define LINESTYLE_NONE        0
#define LINESTYLE_SOLID       1
#define LINESTYLE_LONGDASH    2
#define LINESTYLE_DOTTED      3
#define LINESTYLE_DASHDOT     4
#define LINESTYLE_MEDDASH     5
#define LINESTYLE_DASHDOTDOT  6
#define LINESTYLE_DASHDOTDOTDOT 7
// Symbol styles (search DPLOT.HLP for "SymbolType" for other symbol t
ypes)
#define SYMBOLSTYLE_NONE         0
#define SYMBOLSTYLE_DOT          1
#define SYMBOLSTYLE_CROSS        2
#define SYMBOLSTYLE_ASTERISK     3
#define SYMBOLSTYLE_X            4
#define SYMBOLSTYLE_SQUARE       5
#define SYMBOLSTYLE_DIAMOND      6
#define SYMBOLSTYLE_TRIANGLE     7
#define SYMBOLSTYLE_OCTAGON      8
#define SYMBOLSTYLE_ITRIANGLE    9
#define SYMBOLSTYLE_HEXAGON     10
#define SYMBOLSTYLE_PENTAGON    11
#define SYMBOLSTYLE_STAR        12
#define SYMBOLSTYLE_FILL        0x00000100L  // May be combined with s
tyles above
// 0x0220 - 0x02FF: Wingdings font
// 0x0320 - 0x03FF: Wingdings 2 font
// 0x0420 - 0x04FF: Wingdings 3 font
```