Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Departamento de Ciência da Computação

ORIENTAÇÃO A OBJETO IMPLEMENTADO EM LABVIEW

Por
Vitor Rodrigues Miranda

Projeto da Monografia de Final de Curso

Prof. Roberto da Silva Bigonha

Belo Horizonte, fevereiro de 2012

Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Especialização em informática: Ênfase: Engenharia de Software

ORIENTAÇÃO A OBJETO IMPLEMENTADO EM LABVIEW

Por

Vitor Rodrigues Miranda

Monografia de final de curso

Prof. Dr. Roberto da Silva Bigonha
Orientador

Belo Horizonte, Fevereiro de 2012

VITOR RODRIGUES MIRANDA

ORIENTAÇÃO A OBJETO IMPLEMENTADO EM LABVIEW

Monografia apresentada ao Curso de Especialização em Informática do Departamento de Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais, como requisito parcial para a obtenção do certificado de Especialista em Informática

Área de concentração: Engenharia de Software

Orientador: Prof. Dr. Roberto da Silva Bigonha

Belo Horizonte, Fevereiro de 2012

AGRADECIMENTOS

Em primeiro lugar agradeço a Deus o dom supremo da salvação, motivo pelo qual faço tudo com alegria

Agradeço aos meus pais o apoio e incentivo

E finalmente ao meu orientador pela sua disposição em me ajudar e orientar

RESUMO

O objetivo desse trabalho é apresentar os conceitos e estruturas fundamentais da linguagem de programação gráfica LabVIEW e, baseado nesses conceitos, introduzir a noção de orientação a objeto para essa linguagem. LabVIEW é uma linguagem baseada em diagrama de blocos, na qual o código é representado por elementos gráficos ao invés de linhas de texto. As funções são representadas por caixas e as variáveis por fios. As caixas possuem conectores onde esses fios são ligados, fazendo analogia à passagem de parâmetros para as funções. O LabVIEW utiliza a paradigma de programação *dataflow*, em que o código não tem sua ordem de execução determinada pela sequência em que aparece no programa mas sim pela sequência com que os dados ficam disponíveis na entrada das funções. Ela tem em engenheiros e pesquisadores o seu principal público alvo, pois permite que profissionais sem profundos conhecimentos de engenharia da computação desenvolvam programas elaborados e eficientes. Nesse trabalho apresentamos alguns exemplos e comparações de pequenos trechos de programas com linguagens estruturadas como JAVA e C++. Mostraremos também a forma de se construir um programa orientado a objeto a partir dessa linguagem. Ao fim do trabalho demostramos a aplicação dos padrões de projeto *singleton* e *factory method*.

Palavras-chave: *Orientação a objeto, linguagem gráfica, dataflow*

ABSTRACT

The objective of this paper is to present the concepts and fundamental structures of the LabVIEW programming language and, based in this concepts, introduce the language´s object-oriented features. LabVIEW is a block diagram based language, in which the code is represented by graphical elements instead of code lines. The functions are represented by boxes and the variables by wires. The boxes have connectors in which the wires are connected, representing the input arguments of a function. LabVIEW uses the dataflow paradigm, in which the order of execution of the code is not defined by the order in which it is written but by the order in which data becomes available along the code´s execution. This language is mainly directed to be used by researchers and engineers. Because of its simple representation and concepts it allows professionals with no deep knowledge in science computing to develop high complexity efficient applications. In this paper we are going to show some examples and comparisons between code portions written in LabVIEW and JAVA or C++. We will also show the way of building an object-oriented code from this language. At the end of the paper we will show how to implement the *singleton* and *factory method* design patherns.

Keywords: Object orientation, graphical language, dataflow

LISTA DE FIGURAS

LISTA DE SIGLAS

LABVIEW	Laboratory Virtual Instrument Engineering Workbench
CLP	Controlador Lógico Programável
VHDL	VHSIC Hardware Description Language
FPGA	Field Programable Gate Array
OO	Orientação a Objeto
NI	National Instruments
GPIB	General Purpose Interface Bus
IEEE	Institute of Electrical and Electronics Engineers

SUMARIO

1. INTRODUÇÃO

O conceito de orientação a objeto é largamente utilizado pela grande maioria das linguagens de programação modernas. Isso se dá pelo fato de um código orientado a objeto ser muito mais fácil de estender e manter, além de permitir abstrações bem diretas com situações do mundo real, facilitando a construção de programas. Com o avanço dos sistemas computadorizados, algumas empresas começaram a desenvolver linguagens de programação próprias, voltadas para os sistemas por elas fabricados, como é o caso da linguagem LADDER, usada para a programação de CLP e VHDL, usada para a programação de FPGA. Os dois exemplos citados acima são de equipamentos usados em automação industrial.

A empresa norte americana National Instruments criou a linguagem LabVIEW, que era inicialmente dedicada para a programação de sistemas de coleta de dados. Essa linguagem preza pela simplicidade e intuição, por ser voltada para pessoas sem formação em programação. Por esse motivo LabVIEW é uma linguagem gráfica, ou seja, o programa é feito de acordo com o modelo de fluxo de dados, interligando os blocos de funções por fios para descrever a lógica do programa (Camargo, 2011). Essa linguagem foi sendo aperfeiçoada ao longo do tempo, incorporando os conceitos de linguagens mais difundidas como JAVA e C++ e passou a ser considerada uma linguagem de programação plena. Em 2006 foi introduzido o conceito de orientação a objeto, o que se mostrou um desafio. Os engenheiros de software tiveram que encontrar uma forma coerente para representar todos os elementos de uma classe por meio de fios e blocos.

O objetivo deste trabalho é discorrer sobre como e por que a OO foi aplicada na linguagem e quais as considerações tiveram de ser feitas para que a OO se tornasse viável, tal como as vantagens e desvantagens de se utilizar orientação a objeto em uma linguagem baseada em fluxo de dados. Iremos abordar inicialmente alguns conceitos básicos da linguagem LabVIEW para que o restante do trabalho se faça compreensível. Em seguida iremos abordar alguns conceitos de orientação a objeto por objetivo de familiarizar o leitor com o domínio de discussão. Por fim realizaremos uma comparação entre a linguagem LabVIEW e a linguagem JAVA para esclarecer ainda mais o funcionamento da OO, tendo como base uma linguagem mais usual.

2. A HISTÓRIA DO LABVIEW

No início dos anos oitenta, um dos fundadores da National Instruments, Jeff Kodosky, vislumbrou a possibilidade de criar um ambiente de desenvolvimento voltado para engenheiros e cientistas. A ideia era fazer uma linguagem que representasse para a engenharia o que a planilha de cálculo representa para a área financeira (Camargo, 2011). Para tanto, Kodosky imaginou um ambiente de programação totalmente gráfico. Na época, apenas o Macintosh da Apple oferecia tal recurso. E assim nasceu, em 1986, o LabVIEW 1.0, exclusivamente para Macintosh. Foram cerca de dois anos da concepção até chegar ao produto final. Nascia a primeira versão do *Laboratory Virtual Instrument Engineering Workbench*, o LabVIEW.

Nessa época basicamente a NI era uma fornecedora de interfaces IEEE 488, ou GPIB. A ideia de Kodosky era de transformar um computador (inicialmente os Macs e mais tarde os PCs) em um instrumento de medição virtual, que fosse configurável. Dessa forma o computador poderia se transformar em um multímetro, osciloscópio, gerador de funções e realizar os mais diversos tipos de cálculo com o sinal, dependendo da necessidade do engenheiro ou cientista. Associando o LabVIEW e as interfaces IEEE 488, era possível criar um instrumento bastante flexível, do ponto de vista da Interface de Usuário, Análises disponíveis e apresentação dos dados. Mais tarde, com o advento das placas de aquisição de dados da própria National Instruments , essa configuração ficou ainda mais flexível e totalmente independente dos chamados Instrumentos Tradicionais. Entrava em cena a Instrumentação Virtual. Hoje a instrumentação virtual é utilizada na grande maioria das grandes indústrias de tecnologia, automotiva, mecânica, energia entre outros. A ferramenta possibilita o desenvolvimento rápido de programas que realizam as mais diversas tarefas para a engenharia. Como exemplo podemos citar a determinação de esforços mecânicos de uma estrutura com o uso de extensômetros, que são sensores que medem a deformação de um material através da variação da resistência elétrica medida entre seus terminais. Os dados desses sensores podem ser coletados e analisados por um software escrito em LabVIEW. Outro bom exemplo seria a determinação do comportamento vibracional dessa mesma estrutura através do uso de acelerômetros, que são sensores que medem a intensidade de vibrações mecânicas através da variação da tensão elétrica entre seus terminais, ligados a um computador executando um software escrito em LabVIEW.

Com o passar do tempo o LabVIEW passou a incorporar outros conceitos de programação e hoje é considerada uma linguagem de programação de uso geral, pois com

essa linguagem podemos desenvolver qualquer tipo de software e não somente aqueles usados na instrumentação virtual. O fabricante do sistema possui hoje escritórios em todo o mundo e é o líder na distribuição de hardware para aquisição de dados no Brasil.

3. O QUE É O LABVIEW

3.1. **Conceito**

LabVIEW é um ambiente de programação gráfica voltada a engenheiros e cientistas para o desenvolvimento de sistemas de medição, controle e teste. Em contraste às linguagens de programação baseadas em texto, em que instruções determinam a execução do programa, o LabVIEW utiliza programação baseada em fluxo de dados, em que o fluxo de dados determina a ordem de execução.

Os programas escritos nesse ambiente possuem obrigatoriamente uma interface de usuário e um diagrama de blocos. A interface de usuário é conhecida como “painel frontal”. O painel frontal é construído utilizando objetos de uma biblioteca de componentes, que são acessados a partir de um menu *drag and drop*. Esses componentes podem ser customizados pelo programador, porém não com a mesma flexibilidade oferecida por uma linguagem tradicional como JAVA ou C++. O diagrama de blocos é o código do programa em si. Nessa tela o programador interliga os componentes de forma a criar uma lógica de execução. O diagrama também é construído a partir de bibliotecas de funções que também são acessadas através de um menu *drag and drop*. Cada elemento presente no painel frontal possui um terminal no diagrama de blocos de forma que todas as variáveis destinadas para aquele elemento do diagrama de blocos serão exibidas no seu objeto correspondente no painel frontal. No universo do LabVIEW isso é representado por fios que interligam os objetos. A analogia é imediata. Mostraremos um exemplo que apresenta de forma bem simples a formatação da linguagem e explica de forma bem objetiva o conceito por trás dessa linguagem de programação.

A figura abaixo mostra o painel frontal de uma função que, dependendo da opção que o usuário seleciona no controle “Enum opção”, realiza as operações de soma ou subtração entre os valores dos controles “valor 1” e “valor 2” e exibe o resultado no mostrador “resultado”.

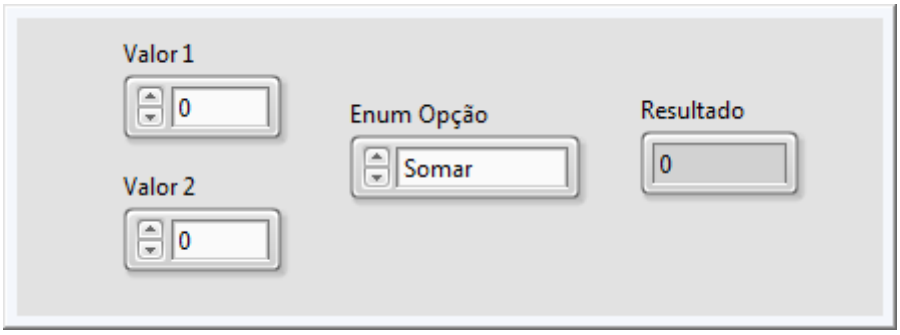

Figura 1 - Painel frontal

As duas figuras abaixo mostram os dois estados que o bloco diagrama da função pode assumir dependendo da opção escolhida pelo usuário:

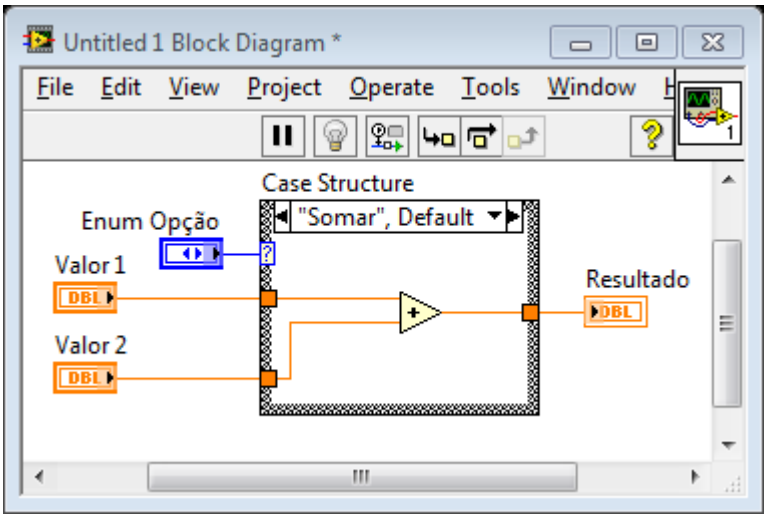

Figura 2 - Bloco diagrama case somar

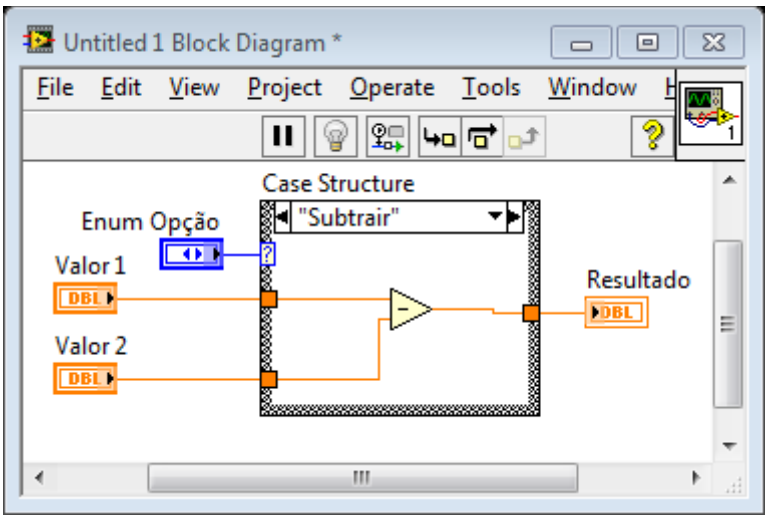

Figura 3 - Bloco diagrama case subtrair

Reparamos que ao redor dos blocos de soma e subtração existe uma estrutura retangular de nome *"case structure"*. Esse retângulo é análogo ao comando "if" das linguagens tradicionais. "Seu modo de operação depende da variável conectada ao terminal "*?*" na borda da estrutura. A estrutura "case" do LabVIEW pode conter vários frames, sendo que a estrutura executa apenas o frame correspondente ao valor da variável conectada no terminal de seleção *"?"* e caso a variável não contenha um valor definido em nenhum dos frames a estrutura executa o frame *"default"*. Dessa forma essa estrutura exerce ao mesmo tempo o papel da estrutura *"if"* e da estrutura "switch case" das linguagens tradicionais.

Observe agora o exemplo abaixo. Ele mostra um programa que calcula as raízes de uma equação de segundo grau através da fórmula de Bhaskara:

$$ax^2 + bx + c \quad \rightarrow \quad x_{1,2} = \frac{-b \mp \sqrt{(b^2 - 4ac)}}{2a}$$

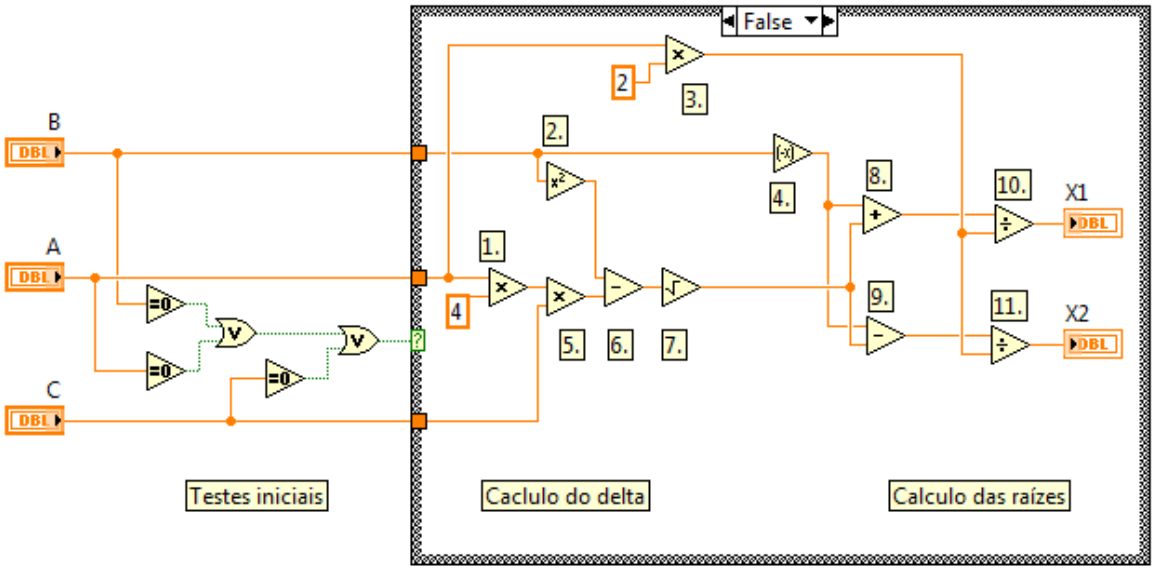

Figura 4 - Equação do segundo grau

Uma das principais características da linguagem LabVIEW é que ela tem a capacidade de executar vários processos em paralelo pelo fato de adotar o paradigma do fluxo de dados. Sob a ótica desse paradigma um bloco se encontra apto à execução quando todos os seus dados de entrada estiverem disponíveis. Dessa forma, o que determina a ordem de execução não é a ordem na qual os blocos estão dispostos no diagrama de blocos e sim a própria ordem de cálculo do programa. Voltaremos a falar do paradigma do fluxo de dados adiante.

No exemplo acima os terminais “A”, “B” e “C” representam as variáveis que contém os valores dos coeficientes da equação. O primeiro bloco de comandos sendo executado é o bloco identificado no código como “testes iniciais”. Ele verifica se existe algum coeficiente com o valor “0”. Observe que todos os três testes realizados pela função “=0” são executados simultaneamente, uma vez que tanto “A”, quanto “B”, quanto “C” estão disponíveis na entrada dessa função no início da execução do programa. Em seguida, caso nenhum dos coeficientes seja igual a “0” o programa inicia o cálculo das raízes. Seguindo o mesmo princípio de disponibilidade de valores nas entradas das funções, percebemos que as operações identificadas pelos números “1”, “2”, “3” e “4” também ocorrem ao mesmo tempo, pois, quando o fluxo de dados atinge a estrutura de seleção (retângulo que envolve o cálculo da fórmula) “A”, quanto “B”, quanto “C” estão disponíveis na entrada. Após a execução dessas primeiras operações, os blocos de comando que dependiam dos resultados por eles gerados são executados. Nesse caso temos somente a operação “5”, que dependia do resultado de “1” para ser executada. Em seguida executa-se a operação “6” que depende do resultado de “5” e “2”. Depois a operação “7”, que depende de “6”. As operações “8” e “9” são executadas paralelamente, assim como “10” e “11”. O resultado do cálculo das raízes é então exibido nos mostradores *X1* e *X2*. Mais detalhes sobre o paralelismo em LabVIEW serão abordados adiante.

3.2. Representação dos tipos de dados

Cada elemento do painel frontal possui seu terminal correspondente no bloco diagrama. Os valores são representados por fios, que correspondem às variáveis do código. Cada tipo de variável (*double, float, int, string*, etc) é representado por um fio diferente. No exemplo mostrado observamos que a variável *double* possui um fio fino alaranjado e o valor numérico inteiro fornecido pelo controle *Enum* possui um fio fino azul. Vetores possuem um fio grosso na cor dos elementos constituintes, matrizes possuem fios duplos e assim por diante. É importante que esses fios sejam distintos porque, como dito anteriormente, seu formato e suas cores identificam o tipo de variável que ele representa. A figura abaixo mostra alguns exemplos de fios.

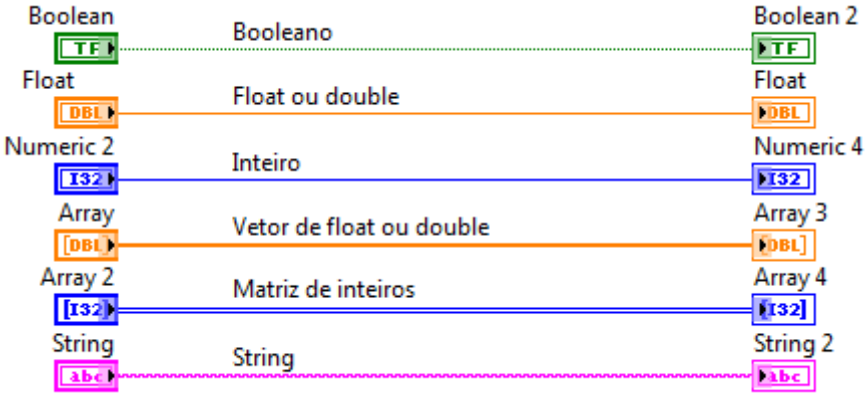

Figura 5 - diferentes variáveis

Existem funções específicas para se criar variáveis de cada tipo de dado. A figura abaixo mostra como são criados os vetores e matrizes de "n" dimensões a partir de valores individuais.

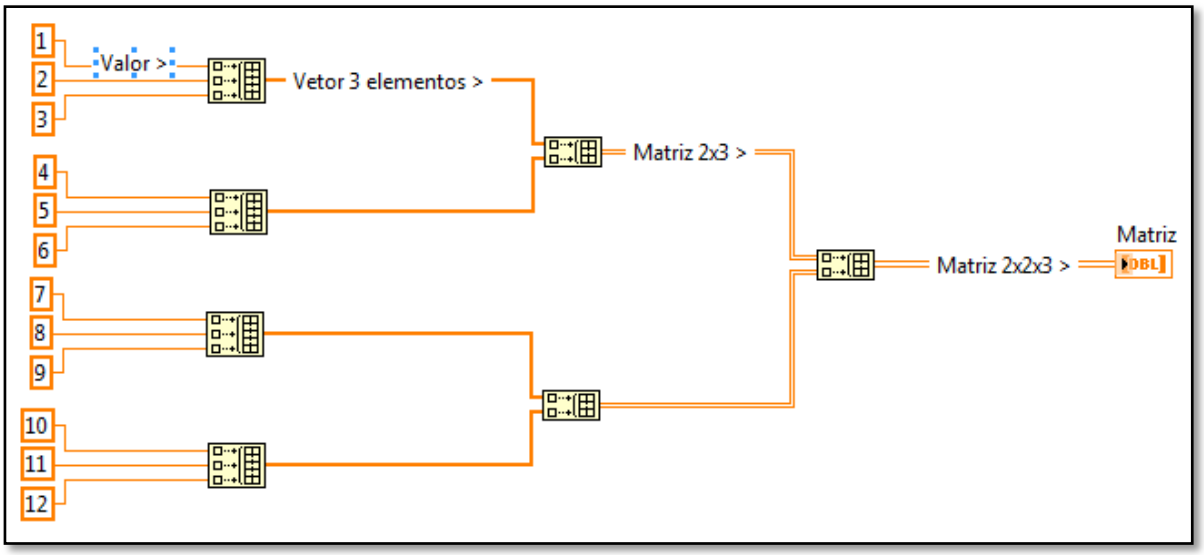

Figura 6 - Vetores e matrizes

Como dito anteriormente, vemos que o padrão do fio que representa a variável se altera de acordo com a quantidade de dimensões que ele contém, ficando cada vez mais espesso. A criação de variáveis multidimensionais é feita através do bloco "*build array*". Essa função recebe várias entradas e as agrupa em uma única saída, colocando cada uma das variáveis recebidas na entrada em uma posição da variável multidimensional resultante. No

exemplo vemos que se fornecermos como entrada três constantes a função nos retorna um vetor com três posições. Se fornecermos como argumentos dois vetores com três posições a função nos retorna uma matriz com duas linhas e três colunas e assim por diante. Não existe limite para o número de dimensões que as variáveis podem atingir. A figura abaixo mostra o resultado das múltiplas concatenações. O primeiro indicador mostra o vetor resultante da primeira chamada da função. O segundo indicador mostra o resultado da concatenação dos dois primeiros vetores formados. O terceiro indicador mostra a matriz tridimensional resultante da concatenação das duas matrizes 2 x 3 geradas no exemplo. Observe que o primeiro índice desse indicador possui o valor 1. Isso significa que ele está mostrando a segunda posição da dimensão de maior grau da matriz.

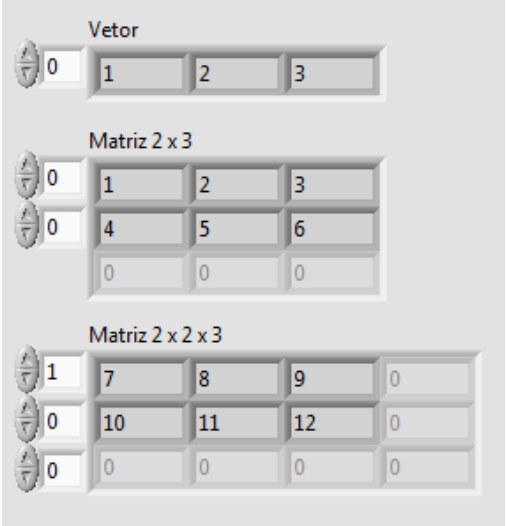

Figura 7 - Variáveis multidimensionais

A linguagem também possui outros tipos de dados variáveis definidas pelo usuário. Em outras linguagens essas estruturas são denominadas *"structs"* ou *"records"*, mas no LabVIEW esse tipo de dado é chamado de cluster (National Instruments, 2005). Um cluster é uma forma de encapsulamento de variáveis de tipos diferentes em uma mesma estrutura. A linguagem permite que o usuário crie e salve esses clusters, criando um novo tipo de dados. Assim como em C++, quando o usuário altera a definição principal da variável, todas as instâncias daquela variável presentes no código também são alteradas. Existem basicamente duas formas de se criar um cluster. A primeira delas é a partir do painel frontal e a segunda a partir do diagrama de blocos. Quando criamos o cluster a partir do painel frontal ele se torna um novo tipo de dado, como acabamos de explicar. Se criarmos o cluster a partir do diagrama de blocos ele fica definido localmente. Normalmente a declaração de clusters pelo bloco diagrama é usado somente quando se deseja diminuir a quantidade de fios na tela de trabalho.

A figura abaixo mostra um cluster declarado partir do painel frontal. Observe que o cluster nada mais é do que uma caixa na qual são colocados quantos controles o usuário desejar. Esse tipo de declaração é o que define um novo tipo de variável. Se o usuário acrescentar algum controle dentro de “cluster” todas as instâncias desse controle serão alteradas ao longo do código:

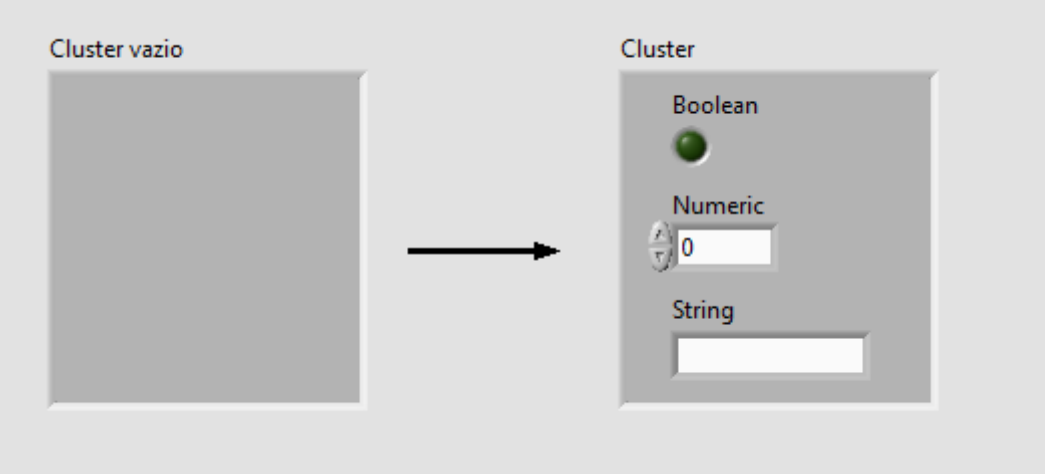

Figura 8 - Construindo um cluster

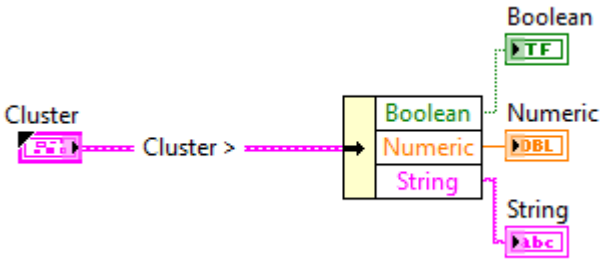

Figura 9 - Cluster no bloco diagrama

A figura acima mostra, no diagrama de blocos o terminal da estrutura criada. Observe que seu terminal possui um detalhe preto no canto superior esquerdo. Isso é o indicativo que aquele terminal pertence a um tipo de dado definido pelo usuário e que ele possui uma declaração padrão, de forma que se sua declaração padrão for alterada ele também será. A função *“unbundle by name”* se encarrega de dar acesso individual a cada um dos controles colocados dentro do cluster.

A figura a seguir mostra o processo de criação a acesso de um cluster a partir do bloco diagrama:

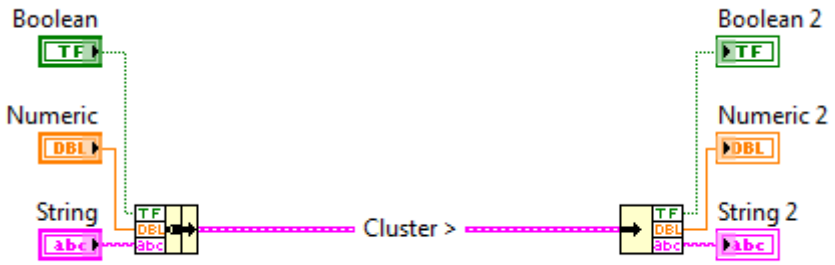

Figura 10 - Cluster a partir do diagrama de blocos

Nesse caso o cluster foi usado somente para transportar dados de um lado para o outro do diagrama. Quando esse procedimento é adotado nenhum novo tipo de dado é definido, o cluster tem efeito local. Para isso basta que o usuário conecte os controles desejados nos terminais da função *"bundle*. Para se ter acesso aos controles individuais do cluster basta conectar o fio (que representa a variável criada) no terminal de entrada da função *"unbundle"* que é mostrada no lado direito do diagrama.

3.3. Funções

Como o LabVIEW é uma linguagem naturalmente baseada em fluxo de dados, não abordaremos as funções como sendo métodos ainda, porque de fato elas podem ser usadas sob qualquer conjunto de dados que seja compatível com as entradas da função, não tendo necessariamente que pertencer a nenhuma classe.

É muito comum, principalmente durante o aprendizado de programação fazermos analogias de funções com caixas, que recebem certo tipo de dado na entrada, realiza operações com aquele dado e retorna certo tipo de saída. No caso da nossa linguagem isso é literalmente o que ocorre porem em LabVIEW chamamos as funções de *subvis* (sub virtual instruments) (National Instruments, 2006). Qualquer diagrama LabVIEW pode ser transformado em uma *subvi*. Quando isso ocorre essa *subvi* é salva em um arquivo separado e pode ter várias instâncias em qualquer parte do código, inclusive dentro de outras *subvis*. Assim como qualquer outro elemento do código LabVIEW uma *subvi* executa sempre que todas os seus argumentos de entrada estão disponíveis em seus terminais, sendo que o fluxo de execução é sempre da esquerda para a direita.

Quando transformamos um diagrama em uma *subvi* é necessário que se defina os argumentos de entrada da função, da mesma forma como são definidos os argumentos de

entrada em um cabeçalho de uma função em JAVA por exemplo. A diferença é que não é necessário explicitar qual o tipo de variável sendo recebida, uma vez que, em função do terminal a ele ligado, o LabVIEW define automaticamente o tipo das variáveis de entrada. A sequência de figuras a seguir demonstra o processo de criação de uma *subvi*.

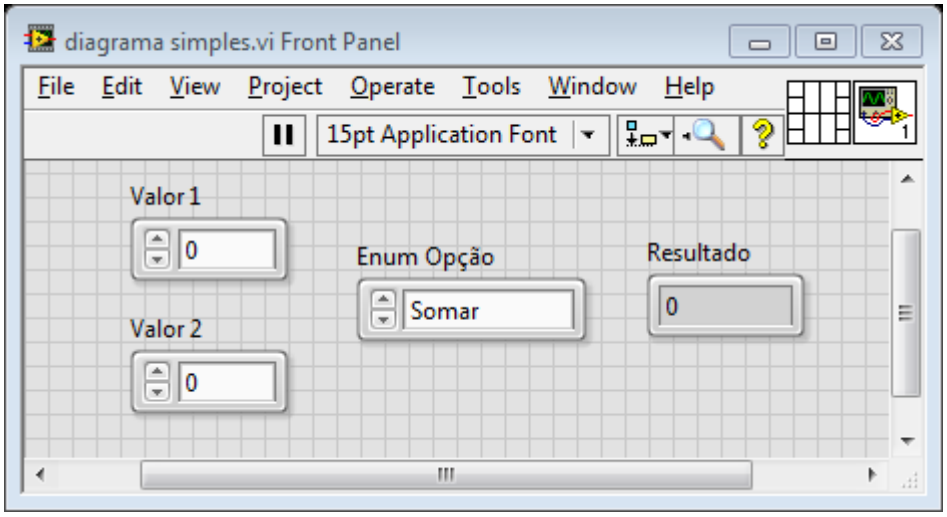

Figura 11 - Painel frontal de uma função qualquer

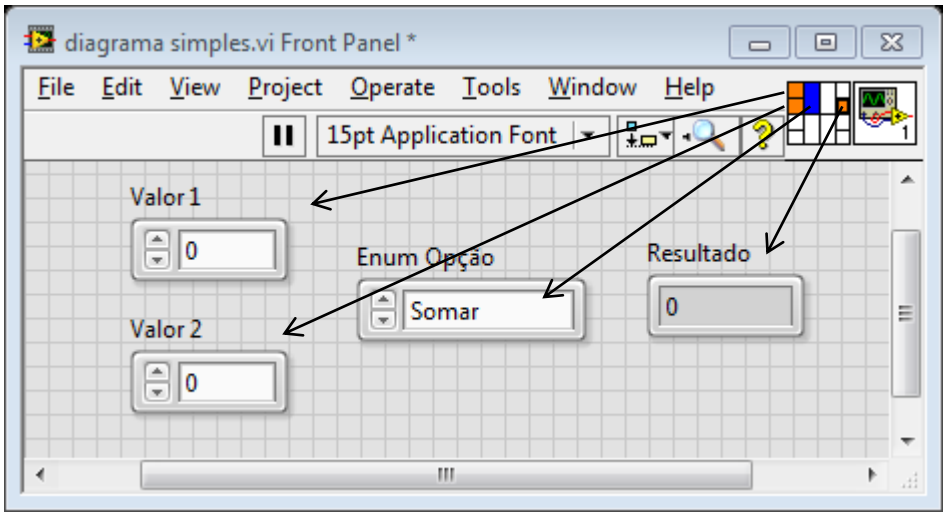

Figura 12 - Definição dos terminais de entrada e saída

No exemplo acima vemos que para transformar qualquer diagrama em uma função basta conectar os terminais. A partir de então esse diagrama passa a ser uma função de nome *"diagrama simples.vi"* Repare que a cor dos terminais associados apresenta a cor que define o tipo de dados de cada controle, que no caso *de "valor 1", "valor 2"* e *"Resultado"* é laranja, que representa o tipo de dados *"double"*. Já a cor do terminal associado ao *"enum opção"* é azul, que representa o tipo de dados *"int"*.

A figura abaixo mostra como a *subvi "diagrama simples.vi"* seria usada dentro de um outro diagrama.

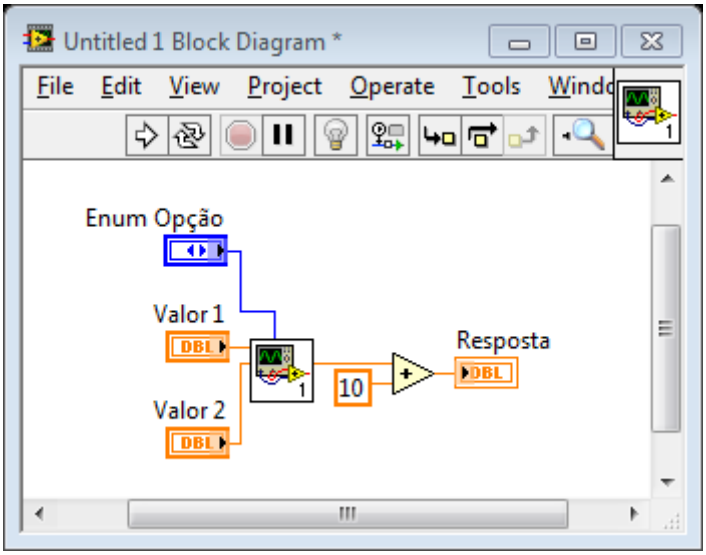

Figura 13 - Uso de subvis

Cada *subvi* criada possui uma ilustração que pode ser definida pelo usuário. O LabVIEW possui uma ferramenta rápida de edição de ícones que permite criar ícones diferenciados para cada função criada, de forma a facilitar a sua identificação dentro de outros diagramas.

3.4. Estruturas de repetição

A linguagem LabVIEW também possui todas as estruturas de repetição mais difundidas como os laços "*while*", "*for*" e *"do while"*. Os exemplos a seguir apresentam essas estruturas:

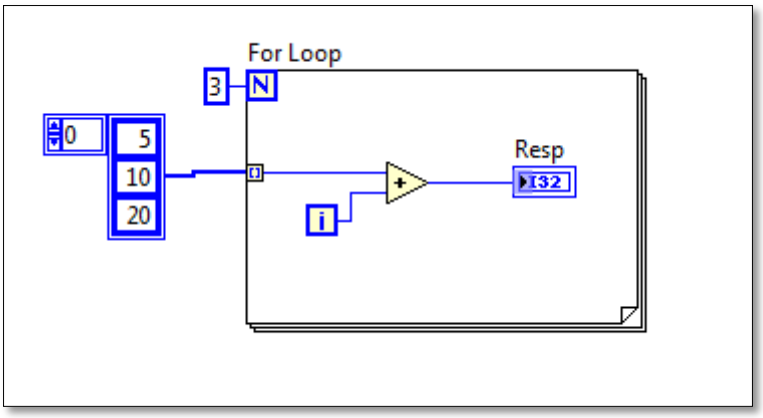

Figura 14 - laço "for"

A figura acima apresenta o laço de repetição *"for"*. O valor ligado ao bloco *"N"* determina o número de repetições do laço e o valor retornado pelo bloco *"i"* corresponde à ordem da iteração atual. Esse pequeno exemplo possui um vetor com três constantes e um laço que executa três vezes, somando o valor de *"i"* em cada um dos elementos do vetor e mostrando a resposta no painel frontal através do mostrador *"resp"*. Observe que na parte externa do loop o fio azul é espesso, o que representa um vetor e que na parte interna do loop o fio azul é mais fino, indicando um único valor. O código JAVA correspondente ao exemplo acima seria:

```
Int a[] = {5, 10, 20};
for (int i=0; i<3; i++)
      resp = i + a[i];
      Sistem.out.println("resp = "+ resp);
end
```

Observe agora o exemplo utilizando o laço *"while"*:

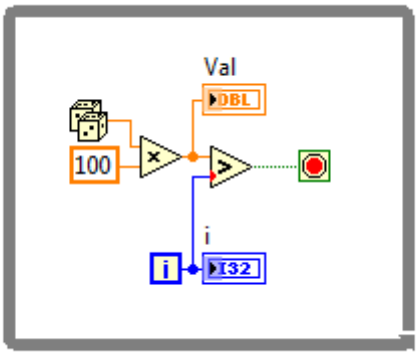

Figura 15 - laço while

A estrutura *"while"*, diferente da estrutura *"for"* não possui o terminal *"N"*. Ao invés disso ela possui um terminal *"stop"*, representado no diagrama pela caixa com o octógono vermelho em seu interior. Todo o código dentro do frame é executado *n* vezes até que um valor booleano *"true"* seja enviado para o terminal *stop*. O código JAVA equivalente ao exemplo mostrado seria:

```
Random random = new Random();
      double x;
      int i = 0;
            do
                  x = random.nextdouble() * 100;
                  Sistem.out.println(" x = " + x);
                  i++;
                  Sistem.out.println(" I = " + i);
            while (x > i);
```

3.5. Biblioteca de funções da linguagem

Podemos observar que em muitos casos o código gerado na linguagem LabVIEW é mais sucinto e de interpretação mais direta que o código em uma linguagem baseada em linha de comando. Essa talvez seja a sua maior vantagem em relação a linguagens baseadas em linhas de comando. O fato de cada uma das funções do programa possuir um ícone que remete para o que cada função realiza faz com que a linguagem se torne fácil de aprender e simples de se programar. Além disso, todas as funções (usaremos o termo funções por não estarmos fazendo ainda uma abordagem sob o ponto de vista da OO) e comandos da linguagem estão

agrupados por área de atuação e basta que o usuário acesse o menu de funções para que ele possa escolher cada uma delas. Existem milhares de funções e comandos disponíveis e o usuário ainda pode criar as suas próprias funções baseadas nas funções e comandos existentes ou pode importar funções escritas em outas linguagens a partir de uma DLL. Quando isso ocorre, a função importada aparece no bloco diagrama em forma de uma caixa na qual o número de terminais de entrada é igual ao número de argumentos da função e o número de terminais de saída é igual ao número de retornos.

As figuras abaixo mostram como o usuário acessa as funções a partir do bloco diagrama e a partir do painel frontal. Dependendo da tela na qual o programador está trabalhando o conteúdo do menu se alterna entre componentes do bloco diagrama e componentes do painel frontal.

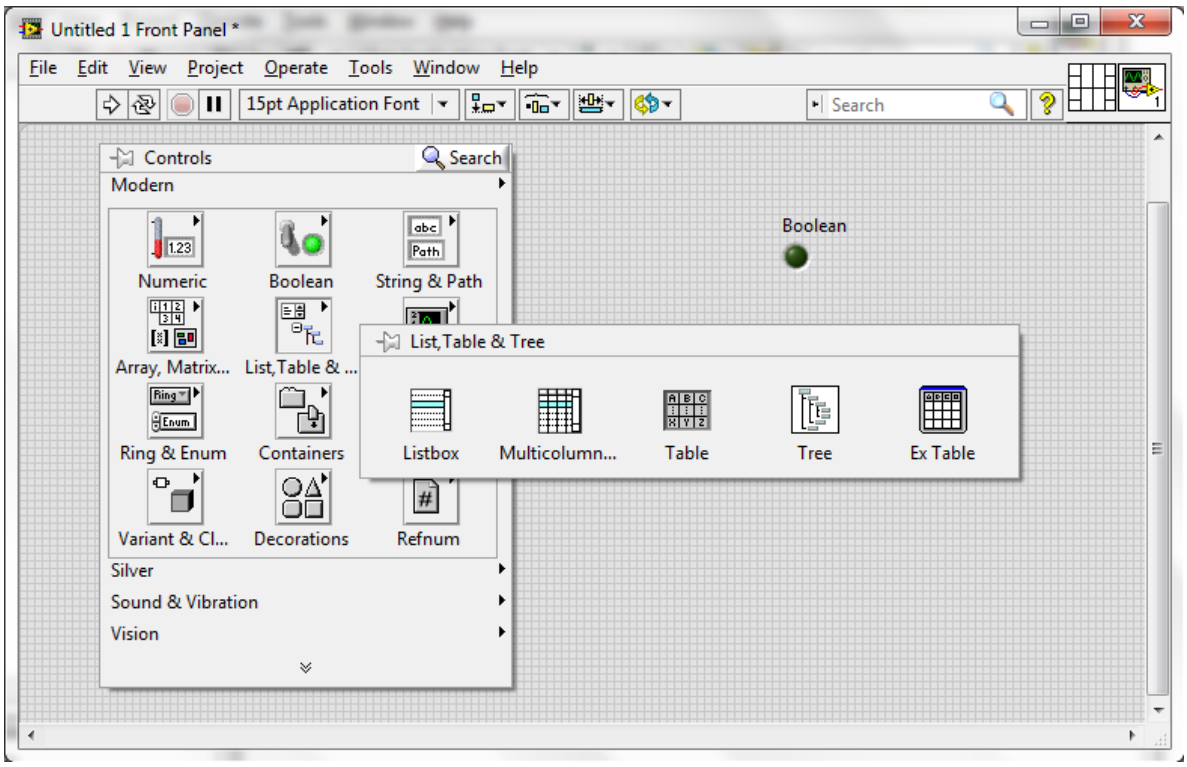

Figura 16 - Menu de componentes do painel frontal

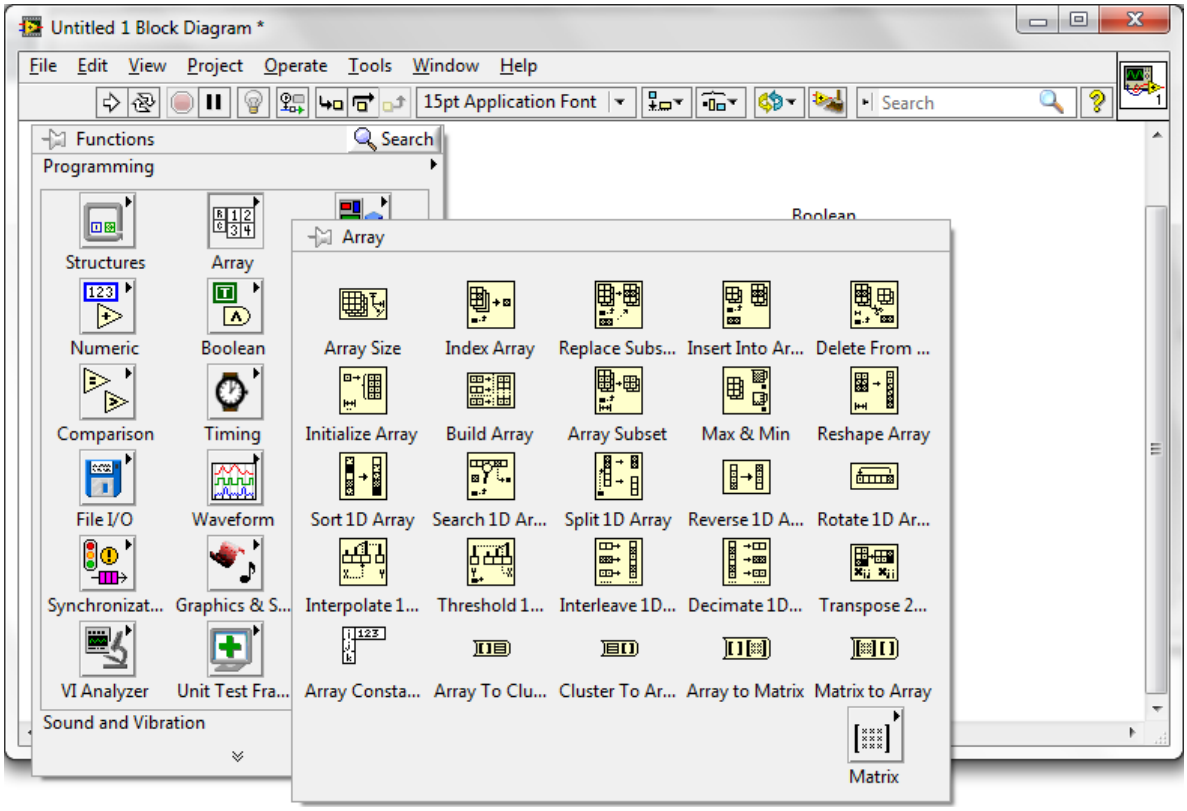

Figura 17 - Menu de componentes do bloco diagrama

Atualmente essa linguagem está se difundindo no Brasil no meio industrial e acadêmico. Cada vez mais pessoas se interessam em aprender a programação gráfica que utiliza o paradigma do fluxo de dados muito por causa da sua agilidade, simplicidade e intuição. A transição do raciocínio de uma linguagem estruturada para uma linguagem baseada em fluxo de dados costuma ser um pouco estafante pelo fato desse tipo de linguagem se basear em um paradigma diferente. Apesar de parecer trivial a linguagem permite a concepção de programas complexos, que utilizam threads concorrentes e conceitos mais avançados de programação. Não é nosso objetivo com essa breve explicação nos aprofundar nos conceitos da ferramenta e sim apresentar uma visão geral para que o leitor possa se situar no contexto do assunto a ser abordado. Para entender o nível de complexidade que um algoritmo em LabVIEW pode atingir observe a figura abaixo. Ela mostra o trecho de um programa usado para fazer coleta, gravação e análise em tempo real de dados de vibração colhidos a partir de um acelerômetro. Esse código executa dois processos em paralelo e processa uma grande quantidade de dados por segundo.

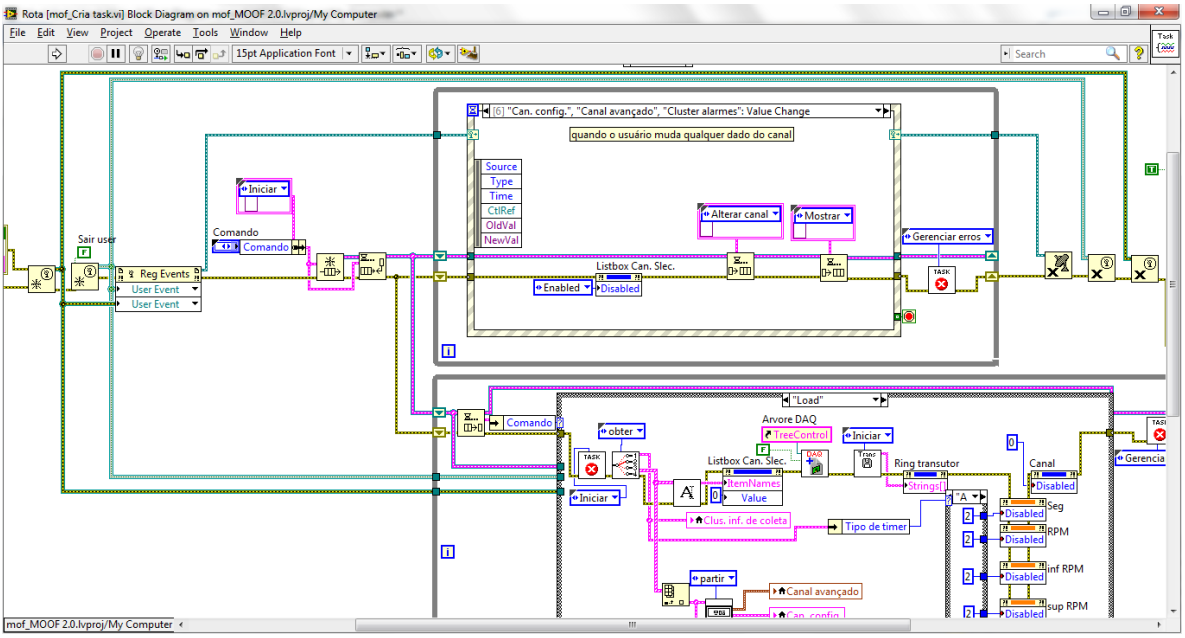

Figura 18 - Código complexo

4. O PARADIGMA DO FLUXO DE DADOS

Para falar sobre o paradigma da orientação a objeto devemos primeiro entender o paradigma do fluxo de dados, que é naturalmente implementado pelo LabVIEW devido a sua natureza gráfica.

O maior benefício de se usar uma linguagem gráfica para resolver problemas de engenharia é o fato de a codificação ser análoga à tarefa de desenhar um diagrama em um papel. Os processadores de vários núcleos que surgiram recentemente tornam o paradigma do fluxo de dados uma escolha mais atraente devido a sua facilidade de expressar processos paralelos. A natureza sequencial do LabVIEW torna fácil para o programador e para o compilador entenderem que sempre que existir uma ramificação em um dos fios do diagrama, ou uma sequência paralela significa que aquelas threads podem ser executadas em paralelo. Nos termos da ciência da computação isso é chamado de “paralelismo implícito”, porque não é necessário que o programador torne explicito que aqueles processos devem correr em paralelo. É exatamente esses paralelismos implícitos que o compilador da linguagem utiliza para tornar o código mais eficiente. Testes iniciais de aplicações comuns escritos em LabVIEW mostram que as aplicações atingem naturalmente uma performance de 25 a 30% melhor pelo simples fato dos códigos apresentarem um paralelismo natural (National Instruments, 2008).

O modelo de fluxo de dados contrasta com o modelo de controle implementado por linguagens como JAVA e C. Devido a abordagem sequencial *top-down* (abordagem na qual o código executa seguindo a ordem na qual as linhas foram escritas), programas escritos em linguagens como as mencionadas anteriormente possuem certas limitações naturais quando são executados por processadores com vários núcleos. Sabemos, no entanto, que tanto JAVA quanto C também programam de forma eficiente, sob o ponto de vista da execução da aplicação, a programação paralela. No entanto isso tem um alto custo para o programador, que deve conhecer a fundo a linguagem e que irá gerar um código bem mais complicado e de acompanhamento mais trabalhoso. Em C, por exemplo, o programador deve gerenciar a sincronização através de travas, e outras técnicas avançadas de programação. Quando o número de *threads* paralelas se torna alto a ponto de ficar difícil de acompanhar, armadilhas como *deadlocks*, contenção de memória, *race conditions* e ineficiência podem surgir.

No modelo *dataflow*, quando um bloco do código recebe todas as suas entradas especificadas ele executa e gera os dados de saída e passa esses dados para o próximo bloco de código na sequência do *dataflow*. O fluxo de dados através dos nodos determina a ordem de execução do código, como já demonstrado nos exemplos anteriores. Abaixo podemos ver um exemplo de código que executa dois processos em paralelo. O loop *"while"* na parte superior do diagrama monitora os eventos gerados pela interface de usuário (observe a estrutura de eventos dentro do loop) enquanto que o loop inferior trata de ler as portas do hardware e gravar os dados em arquivo. O simples fato de um loop estar colocado sobre o outro já diz ao compilador que os processos são paralelos. Caso o programa esteja sendo executado em um computador com vários núcleos de processamento o LabVIEW endereça automaticamente uma *thread* para cada núcleo. Caso esteja instalado em uma máquina com processador *multicore*, o LabVIEW tem a capacidade de endereçar trechos de código que podem ser executados em paralelo para processadores diferentes. Se voltarmos no exemplo da formula de Bhaskara, mostrado no início desse texto vamos perceber que existem várias operações que podem ser executadas em paralelo.

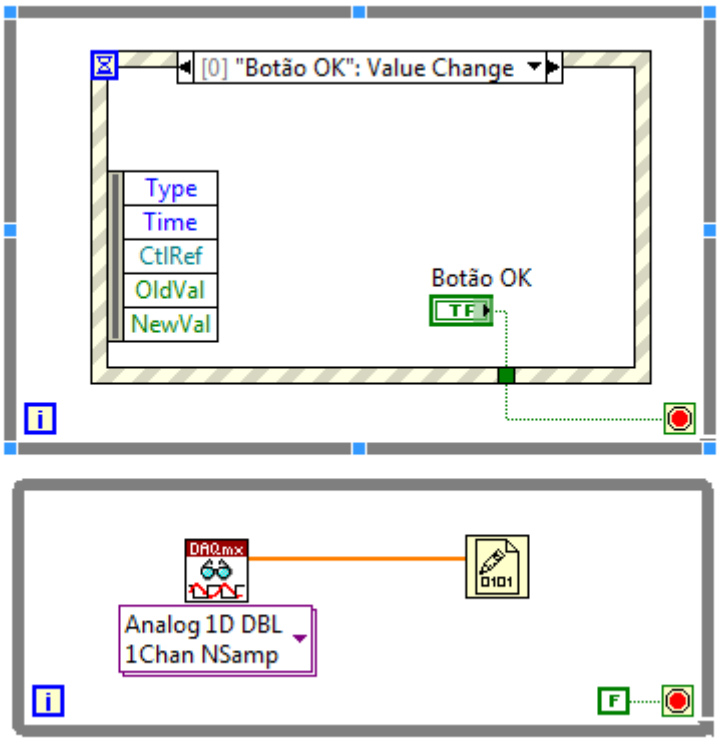

Figura 19 - Threads paralelas

Para que esse mesmo código fosse executado em C ou JAVA várias linhas de código adicionais teriam que ser geradas para gerenciar o paralelismo. Além disso um código escrito baseado no paradigma do fluxo de dados é mais fácil de se documentar, entender e depurar, pois não existem atalhos pelo programa. Um programador que domina a linguagem não encontra grandes problemas para rapidamente desvendar o funcionamento de um código.

5. O PARADIGMA DA ORIENTAÇÃO A OBJETO

A programação orientada a objeto consiste em abordar um problema lançando mão da análise das entidades e seus relacionamentos dentro do contexto do problema. Orientação a objetos é um termo que descreve uma série de técnicas para estruturar soluções para problemas computacionais. Trata-se de um paradigma de programação no qual um programa é estruturado em objetos, e que enfatiza os aspectos abstração, encapsulamento, polimorfismo e herança. A seguir falaremos a respeito de cada uma delas.

5.1. Abstração

A abstração é a prática de se referir a uma entidade através de suas características que são relevantes para o negócio do problema. Esse conceito é mais bem explicado através de um exemplo: Suponhamos que um carro está para entrar em um estacionamento particular. Cada veículo presente no estacionamento deve ser identificado para que o funcionário saiba quanto tempo cada carro permaneceu no estabelecimento e assim possa aplicar a taxa devida. Para isso ele possui um bloco de anotações onde é anotada a placa e o horário de entrada de cada veículo. Observe nesse exemplo que o carro está sendo representado pela sua placa, ou seja, se o funcionário conhece uma característica (ou atributo) do carro, que nesse caso é a placa, ele consegue identificar todo o veículo. Sendo assim a placa do carro é a abstração usada no problema. É importante ressaltar que a característica ou características (ou atributos) escolhidas para ser a abstração do objeto devem ser relevantes para o domínio do problema. Várias outras características poderiam ter sido usadas para se referir ao veículo como cor, número de portas, ano de fabricação, marca, distância entre eixos, número de cilindros no motor entre outras, mas observe que nenhuma delas é relevante para o domínio do problema, que nesse caso é identificar um carro no estacionamento. Sob o ponto de vista da programação fazemos o mesmo tipo de analogia, definindo os atributos das classes geradas de forma coerente e relevante para o domínio de negócio.

5.2. Encapsulamento

Encapsular significa separar o programa em partes, as mais isoladas possíveis. A ideia é tornar o software mais flexível, fácil de modificar e de criar novas implementações. Uma boa forma de se explicar o conceito de encapsulamento é através de um exemplo: imagineos um motorista. Para que ele consiga dirigir é necessário que ele conheça apenas os pontos de interface do carro, que são justamente seus controles e indicadores. Em outras palavras ele precisa saber que o pedal do acelerador faz o carro se mover, o pedal de freio faz o carro parar e que o volante muda sua direção. Os indicadores do carro seriam os mostradores do painel de instrumentos, que retornam ao motorista o resultado das ações por ele tomadas sobre os controles. Nesse caso se ele aperta o acelerador o velocímetro retorna um dado de aumento na velocidade e se ele pisa no freio o velocímetro mostra que a velocidade diminuiu. O motorista

não precisa saber o funcionamento do carro para dirigi-lo. Ele não precisa saber que ao apertar o acelerador a borboleta do coletor de entrada libera um volume maior de ar ao mesmo tempo em que a injeção eletrônica aumenta a quantidade de combustível que entra na câmara de combustão, levando a uma explosão com maior poder calorífico causando um aumento na velocidade angular do eixo virabrequim que por sua vez traciona as rodas fazendo com que a velocidade do carro aumente. Todos esses detalhes de funcionamento estão encapsulados para que o usuário não precise se preocupar com eles. O mesmo ocorre com a orientação a objeto. Os comandos internos dos métodos se encontram encapsulados, sendo a sua interface a única parte do código que precisa ser conhecida pelo usuário. Dessa forma o código de um método pode ser alterado sem que haja necessidade de se alterar sua implementação no corpo principal do programa. Trazendo essa realidade para o exemplo citado, se um mecânico trocar todo o motor de um carro o motorista continua sendo capaz de dirigir o veículo, uma vez que sua interface não mudou.

O encapsulamento proporciona grandes vantagens para o código pois quebra o problema principal entre vários outros problemas menores e mais amplos, facilitando a tarefa do programador. É importante mencionar que o paradigma do fluxo de dados também programa os aspectos da abstração, através do uso das *subvis*, mencionadas anteriormente.

5.3. Polimorfismo

Polimorfismo é o princípio pelo qual duas ou mais classes derivadas de uma mesma superclasse podem invocar métodos que têm a mesma identificação (assinatura) mas comportamentos distintos, especializados para cada classe derivada, usando para tanto uma referência a um objeto do tipo da superclasse. A decisão sobre qual o método que deve ser selecionado, de acordo com o tipo da classe derivada, é tomada em tempo de execução. Em outras palavras podemos dizer que o comportamento de um método polimórfico muda de acordo com o tipo de objeto sobre o qual o método foi invocado. Isso permite uma maior agilidade durante a programação, além de simplificar a vida do programador, pois diminui a quantidade de informação que ele precisa gerenciar.

Essa característica da orientação a objeto permite um código mais versátil e fácil de manter. O exemplo abaixo mostra um pequeno exemplo de uma aplicação de polimorfismo em JAVA.

```
public interface Carro{
   public void acelerar();
}
public Ferrari implements Carro{
   public void acelerar(){
      System.out.println("Ferrari acelerando...");
   }
}
public Fusca implemets Carro{
   public void acelerar(){
      System.out.println("Fusca tentando acelerar...");
   }
}
Carro c = new Ferrari();
c.acelerar();
c = new Fusca();
c.acelerar();
```

O exemplo acima mostra de forma simples e direta o polimorfismo. A primeira chamada do comando "*c.acelerar();"* retorna "Ferrari acelerando", pois nesse momento a variável "c" se refere a um objeto do tipo "Ferrari". A segunda chamada da função retorna "Fusca tentando acelerar", pois nesse instante a variável "c" aponta para um objeto do tipo "Fusca". Vemos que dependendo do tipo de objeto passado como parâmetro o método *"acelerar"* apresenta retornos diferentes.

5.4. Herança

Herança é um conceito que permite que as classes de um programa compartilhem atributos e métodos, como forma de diminuir a quantidade de código gerado e organizar o programa hierarquicamente. Quando uma classe herda parâmetros e atributos de outra ela passa a ser considerada pelo programa como uma filha dessa primeira classe. É o conceito chamado de *"é um"*. Uma forma simples de explicar o conceito do "é um" é através de um

exemplo. Se a estrutura hierárquica das pessoas que convivem em uma universidade fosse separada em classes poderíamos ter a seguinte estrutura:

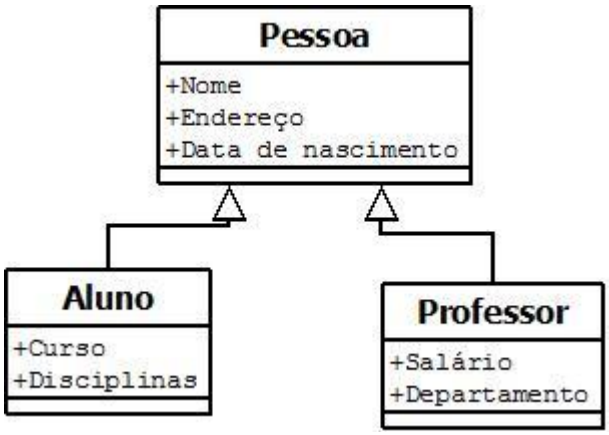

Figura 20 – Herança

Evidente que a estrutura organizacional de uma universidade não se resume a alunos e professores, porém a figura acima cumpre o papel de explicar o conceito de herança. Ambos, alunos e professores são pessoas, sendo assim possuem atributos em comum como nome, endereço e data de nascimento. Antes de serem alunos ou professores os seres humanos que convivem na universidade são pessoas e possuem várias características em comum. Entretanto existem especializações de pessoas, que no contexto da universidade, sob o ponto de vista em questão, se dividem entre alunos e professores. Os alunos possuem características que são próprias só de alunos, como o curso que estudam e as disciplinas que cursam. No caso dos professores não faz sentido falar em termos de curso que estudam ou disciplinas nas quais estão matriculados, porque esses atributos não se aplicam. Sendo assim, no exemplo citado, os professores possuem os atributos “salário” e “departamento”. O fato de um professor ter um salário e um departamento não afeta o fato de ele ter um nome, sendo assim esse atributo, assim como no caso do aluno, continua fazendo parte da sua lista de atributos.

Assim como acontece com os atributos, classes que herdam de outras classes também herdam os seus métodos. Métodos são ações que podem ser tomadas sobre algum objeto. No nosso exemplo poderíamos tem um método de nome “caminhar” pertencente à classe “pessoa”. Ora, a forma de caminhar de um professor é a mesma forma de caminhar de um aluno, sendo assim, invocando-se o método “caminhar” passando como parâmetro um objeto da classe “professor” ou da “classe” aluno resultaria na chamada do mesmo código pertencente à classe “pessoa”. Temos aqui uma herança de método.

A orientação a objeto define classes e métodos que buscam analogia com as entidades e ações relacionadas ao problema em questão. A orientação a objeto define um paradigma de

programação mais elaborado que o fluxo de dados. Na prática percebemos que o paradigma de fluxo de dados também oferece estruturas de dados bem elaborados, permitindo a construção de códigos robustos e fáceis de manter e compreender. Mesmo assim, para programas com mais funções e mais complexos a orientação a objeto se mostra mais adequada.

6. CONSIDERAÇÕES PARA A ORIENTAÇÃO A OBJETO EM LABVIEW

6.1. Porque orientação a objeto para LabVIEW

Antes de começar a falar sobre orientação a objeto para LabVIEW devemos primeiro entender o porque de se implementar a orientação a objeto em uma linguagem de fluxo de dados. Como dito na introdução, inicialmente o LabVIEW tinha mais características de uma ferramenta de design do que uma linguagem de programação (National Instruments, 2009). Com o passar do tempo a ferramenta foi evoluindo até se tornar uma linguagem de programação plena. Como a linguagem foi inicialmente concebida para o uso de pessoas sem muita intimidade com a programação nos seus detalhes mais profundos, os fabricantes da ferramenta buscaram uma forma de fazer com que os conceitos pudessem ser aplicados da forma mais simples possível. Dessa forma uma pessoa com pouca intimidade com a engenharia de software seria capaz de gerar um programa orientado a objeto sem se perder. Sendo assim, uma pessoa já acostumada com os conceitos tradicionais da orientação a objeto pode estranhar em um primeiro momento as considerações feitas para tornar a implantação do OO na linguagem LabVIEW possível.

A orientação a objeto tem se mostrado em vários aspectos superior às linguagens procedurais, tendo sido escolhida como base em várias linguagens de programação. Além dessas características a OO encoraja o desenvolvimento de interfaces mais coerentes e diminui o grau de acoplamento entre as várias partes de um código, sendo mais fácil de estender e depurar.

6.2. Características OO em LabVIEW

Cada linguagem orientada a objeto implanta o conceito OO de forma distinta. Entre as várias linguagens orientadas a objeto existentes atualmente podemos citar C++, JAVA, Smaltalk, .NET, Pascal, C#, Delphi, VB , Eifell, Ruby, Python, Perl, entre outras. Entretanto o paradigma OO não é tratado da mesma forma em todas elas. A tabela abaixo mostra uma comparação entre algumas das linguagens citadas acima. Observe que cada linguagem implanta a orientação a objeto de forma distinta e essa forma de implantação está diretamente ligada com a visão dos criadores da linguagem e o ramo do universo da programação que a linguagem pretende atender.

	Eiffel	Smalltalk	Ruby	Java	C#	C++	Python	Perl	Visual Basic
Object-Orientation	Pure	Pure	Pure	Hybrid	Hybrid	Hybrid / Multi-Paradigm	Hybrid	Add-On / Hybrid	Partial Support
Static / Dynamic Typing	Static	Dynamic	Dynamic	Static	Static	Static	Dynamic	Dynamic	Static
Generic Classes	Yes	N/A	N/A	No	No	Yes	N/A	N/A	No
Inheritance	Multiple	Single	Single class, multiple "mixins"	Single class, multiple interfaces	Single class, multiple interfaces	Multiple	Multiple	Multiple	None
Feature Renaming	Yes	No	Yes	No	No	No	No	No	No
Method Overloading	No	No	No	Yes	Yes	Yes	No	No	No
Operator Overloading	Yes	Yes?	Yes	No	Yes	Yes	Yes	Yes	No
Higher Order Functions	Agents (with version 5)	Blocks	Blocks	No	No	No	Lambda Expressions	Yes (???)	No
Lexical Closures	Yes (inline agents)	Yes (blocks)	Yes (blocks)	No	No	No	Yes (since 2.1)	Yes	No
Garbage Collection	Mark and Sweep or Generational	Mark and Sweep or Generational	Mark and Sweep	Mark and Sweep or Generational	Mark and Sweep or Generational	None	Reference Counting	Reference Counting	Reference Counting
Uniform Access	Yes	N/A	Yes	No	No	No	No	No	Yes
Class Variables / Methods	No	Yes	Yes	Yes	Yes	Yes	No	No	No
Reflection	Yes (as of version 5)	Yes	Yes	Yes	Yes	No	Yes	Yes?	No
Access Control	Selective Export	Protected Data, Public Methods	public, protected, private	public, protected, "package", private	public, protected, private, internal, protected internal	public, protected, private, "friends"	Name Mangling	None	public, private
Design by Contract	Yes	No	Add-on	No	No	No	No	No	No
Multithreading	Implementation-Dependent	Implementation-Dependent	Yes	Yes	Yes	Libraries	Yes	No	No
Regular Expressions	No	No	Built-in	Standard Library	Standard Library	No	Standard Library	Built-in	No
Pointer Arithmetic	No	No	No	No	Yes	Yes	No	No	No

Tabela 1 - Particularidades das linguagens

Assim como cada linguagem possui suas considerações, os criadores do LabVIEW também fizeram várias considerações em relação a orientação a objeto.

Em LabVIEW o conceito OO foi baseado em duas prerrogativas: encapsulamento e herança. Para os criadores da linguagem o usuário deve ser capaz de criar um cluster de dados (como explicado no final da seção “representação dos tipos de dados”) e fazer com que aquele cluster se torne um tipo de dado definido pelo usuário com acesso exclusivo por algumas funções definidas. Também deve ser capaz de criar uma classe a partir de outra previamente definida e fazer com que ela herde os seus métodos e atributos.

6.3. Passagem de parâmetros

O LabVIEW utiliza a paradigma *dataflow*. Como mostrado anteriormente as variáveis na nossa linguagem são representada por fios. Quando o programador cria uma ramificação

desse fio o valor daquela variável é duplicado na memória. Sendo assim as variáveis utilizam a sintaxe “por valor”. Entretanto existe outro tipo de variável na linguagem que é chamado de referência. As variáveis ou fios do tipo “referência” são análogos a ponteiros. Elas contém apenas o endereço da memória onde o valor por eles referenciado pode ser encontrado. Quando um fio do tipo “referência” é ramificado a linguagem não cria uma nova área de memória para aquele controle, duplica somente o valor do ponteiro. Nesse caso a linguagem está utilizando a sintaxe “por referência”.

Consideremos a linguagem C++. Existem duas formas de se passar um objeto para um método podendo ser por valor ou por referência. O programador deve sempre estar atento para a forma de passar os objetos que ele está utilizando. Em linguagens como JAVA e C# a única forma de se passar um objeto para um método é por referência. Isso significa que todos os métodos estão sempre se dirigindo ao objeto original na mesma região da memória. Para fazer uma cópia do objeto o programador deve utilizar métodos explícitos. Tendo como pano de fundo essas comparações podemos discutir para o caso do LabVIEW qual seria a melhor forma de aplicar a passagem de objetos aos métodos.

A nossa linguagem utiliza primariamente o paradigma do *dataflow*. Esse estilo de representação do código permite que o programador ramifique vários fios a partir de um fio principal, criando várias cópias do valor por ele representado na memória. Múltiplos valores permitem que várias *threads* executem paralelamente, sem a preocupação de que uma *thread* irá interferir na execução da outra. Se tivermos os valores representados por seus ponteiros, todas as *threads* estarão se referindo a uma mesma posição na memória, criando a possibilidade de interferência entre *threads*. Nesse caso o programador teria de se responsabilizar por prevenir comportamentos anormais, como *race conditions*, por exemplo. Como discutido anteriormente, o LabVIEW se propõe a ser uma linguagem voltada para aquisição e processamento de dados em tempo real. Para que isso se tornasse possível o sistema deveria ter fortes vocações para trabalhar com processamento paralelo, favorecendo a execução de múltiplas *threads* simultâneas. Sendo assim a linguagem implanta somente a sintaxe “por valor” para passar objetos como argumentos para os métodos. Isso faz com que o programador tenha menos trabalho e o programa se torne menos suscetível a falhas devido a acessos concorrentes a mesmas regiões da memória. Portanto, sempre que existir uma bifurcação de algum fio que representa um objeto este será duplicado, da forma como o compilador julgar mais apropriada (National Instruments, 2009)

6.4. LabVIEW Object, o ancestral de todas as classes

LabVIEW Object é o nome da classe ancestral de todas as classes criadas na linguagem. Em outras palavras todos herdam de *LabVIEW Object*, da mesma forma que em JAVA todas as classes herdam de *java.object.lang* por exemplo. Uma classe ancestral comum a todos os objetos possibilita a criação de funções que podem operar com qualquer classe da mesma forma. Sendo assim funções como *build array* e *bundle by name*, que são escritas para receber objetos do tipo *LabVIEW Object* mostradas anteriormente, possam trabalhar formando vetores com elementos de várias classes por exemplo.

6.5. Tipo de dados

LabVIEW possui somente dados do tipo *private*, ou seja, os atributos das classes são visíveis apenas para suas respectivas classes. Somente os métodos podem ser públicos ou privados (National Instruments, 2009). O maior motivo para que todos os tipos de atributos de classes LabVIEW fossem privados é que a própria estrutura do sistema induzisse o usuário a escolher a arquitetura mais robusta, mesmo que ele não seja um especialista em software. Isso é uma tentativa de diminuir a quantidade de conceitos avançados de programação que o usuário deve dominar para conseguir desenvolver aplicações eficientes e robustas. Limitando o acesso das variáveis pelos respectivos métodos de suas classes simplifica o processo de depuração do código, pois concentra todas as possibilidades de alteração do valor daquela variável em um único local conhecido. Isso também cria um ambiente onde lógicas de validação dos dados da variável possam ser implantados. O fato de a linguagem não dar suporte a tipos de dados protegidos ou públicos elimina um conceito potencialmente confuso, e direciona os usuários a escolherem uma arquitetura melhor. Um lado negativo dessa prerrogativa é que se cria a necessidade de construir métodos para leitura e escrita de cada um dos parâmetros das classes, o que muitas vezes pode ser uma tarefa exaustiva. Sendo assim a versão 8.5 do LabVIEW introduziu uma ferramenta especializada em criar métodos para acesso das variáveis de forma automática. Não entraremos em maiores detalhes a respeito dessa ferramenta.

6.6. Métodos construtores

Um ponto que alguns programadores com experiência em outros tipos de linguagem certamente estranham quando entram em contato com a orientação a objeto para LabVIEW é justamente a forma como os criadores abordaram esse conceito. LabVIEW não implanta métodos construtores de objetos. Analisando mais profundamente a função dos métodos construtores vemos que eles são usados para desempenhar uma ou mais das seguintes tarefas:

- Selecionar os valores iniciais de um objeto
- Calcular valores iniciais de um objeto baseado em parâmetros de entrada
- Reservar recursos de sistema usados pelo objeto tais como memória, canais de comunicação, hardware entre outros.

O conjunto de atributos de uma classe em LabVIEW nada mais é do que um *cluster*, como mostrado anteriormente. Por default a linguagem tem a habilidade de definir valores iniciais para cada elemento desse cluster. No instante em que o usuário cria o cluster e o salva em arquivo, os valores contidos em seus indicadores serão considerados os valores padrão daquela variável. Sendo assim, todas as vezes que um controle daquele tipo for invocado no código ele já virá preenchido com seus valores padrão. A orientação a objeto tirou proveito dessa característica para resolver a questão da inicialização de variáveis.

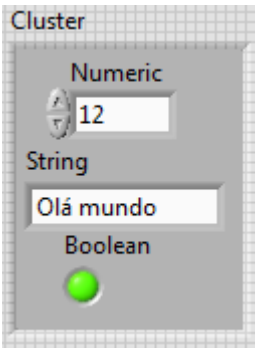

Figura 21 - Cluster inicializado com valores padrão

Consideremos um exemplo similar em C++:

```
Class Coisa{
      private:
            int mx;
            double mz;
            std::string ms;
      public:
            Coisa(): mx(1), mz(2,5), ms(“abc”){}  //construtor 1
            Coisa(int x){                         //construtor 2
```

```
mx = x;
mz = (x>1)? 3.5: 2.5;
ms = "def";
```

No caso mostrado acima, o primeiro método construtor simplesmente inicializa as variáveis. O mesmo ocorre com o exemplo mostrado em LabVIEW. Já o segundo método construtor não possui suporte direto em LabVIEW. Para que um objeto tenha seus parâmetros iniciados com valores que precisam de um algoritmo para serem definidos, o programador deve criar um método que recebe parâmetros externos e o objeto inicializado e então alterar os valores iniciais.

6.7. Métodos destrutores

Para falar de métodos destrutores primeiro devemos abordar o porquê de eles existirem. Para isso faremos uma comparação com a linguagem C++. Nessa linguagem o ciclo de vida de uma variável inteira começa no momento em que ela é declarada e se encerra no momento em que o retorno da função em que ela foi criada é executado. No caso do LabVIEW o ciclo de vida de uma variável não pode ser determinado diretamente, pois a variável não é extinta no momento em que o fio que a representa termina e sim a partir do momento em que aquela variável não é mais necessária no programa. Essa é uma tarefa que cabe ao compilador. Sendo assim concluímos que as variáveis em LabVIEW não possuem um ciclo de vida "espacial", que é aquele em que a variável é declarada em um ponto do código e destruída em outro ponto, e sim um ciclo de vida "temporal". Quando um fio se encontra ligado a um terminal de indicador no painel frontal, o valor daquela variável persiste até mesmo depois que a função termina a sua execução. Cada vez que um fio representando um objeto é bifurcado uma nova cópia daquele objeto é criado na memória. Fica sempre a cargo de o compilador decidir quando uma variável ou objeto deve deixar a memória e liberar os recursos para ele reservados. Sendo assim o LabVIEW não possui funções destrutoras para objetos.

7. CONSTRUINDO UMA APLICAÇÃO OO EM LABIVEW

7.1. Criando uma classe

Para explicar como se construir um programa orientado a objeto em LabVIEW devemos primeiro entender o conceito de projeto. Um projeto LabVIEW basicamente é um arquivo que condensa todas as funções de um determinado programa em um único lugar. Como cada função ou *vi* é escrito em um arquivo diferente o usuário poderia se perder entre tantos arquivos. O arquivo de projeto contém os paths de todas as *vis* usadas em um programa e permite ao usuário invocar qualquer função e fazer qualquer tipo de associação entre esses arquivos. Em outras palavras o projeto é um mapa geral da aplicação. A figura abaixo apresenta uma janela de exploração de um projeto.

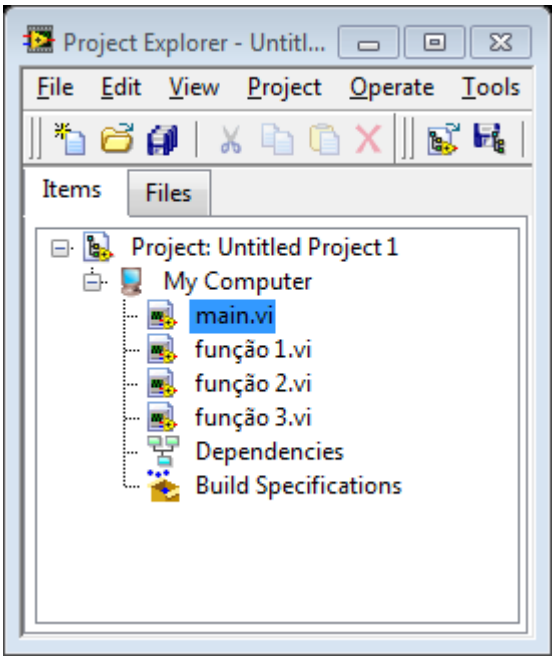

Figura 22 - Project explorer

Cada uma das funções salvas no projeto está armazenada em algum local do disco. Como essas funções compõe uma aplicação, os caminhos de todas as funções estão salvos no arquivo de projeto. Observe que todas elas estão colocadas abaixo de *my computer*, indicando que todas elas estão localizadas em algum local do disco rígido do computador. Os arquivos que compõe uma aplicação também podem estar localizados em outros computadores ou até

mesmo em outros tipos de equipamentos, como CLP´s (Controlador Lógico Programável). Nesses casos os arquivos apareceriam abaixo de algum outro domínio dentro da árvore de projeto.

Cada classe em LabVIEW também é definida em um arquivo separado. A classe também é criada a partir da árvore de projeto. Observe a figura abaixo:

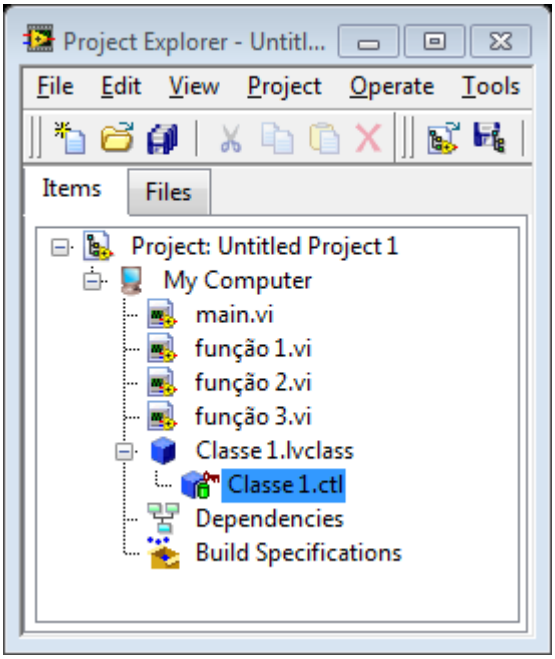

Figura 23 - Representação de uma classe

Assim como no caso das funções os endereços das classes de um projeto salvas em disco são armazenadas no arquivo de projeto. Observe que a classe criada para o nosso exemplo *Classe 1.lvclass* possui um item logo abaixo, chamado *Classe1.ctl*. Esse item é justamente o conjunto de atributos da classe e é criado automaticamente quando a classe é definida (Labview, 2011). No momento de sua criação ele não apresenta nenhum conteúdo. Como já mencionado anteriormente as classes em LabVIEW possuem apenas atributos privados. Para definir os atributos o usuário deve editar o arquivo *Classe 1.ctl*. Ao abrir esse arquivo a seguinte tela é exibida:

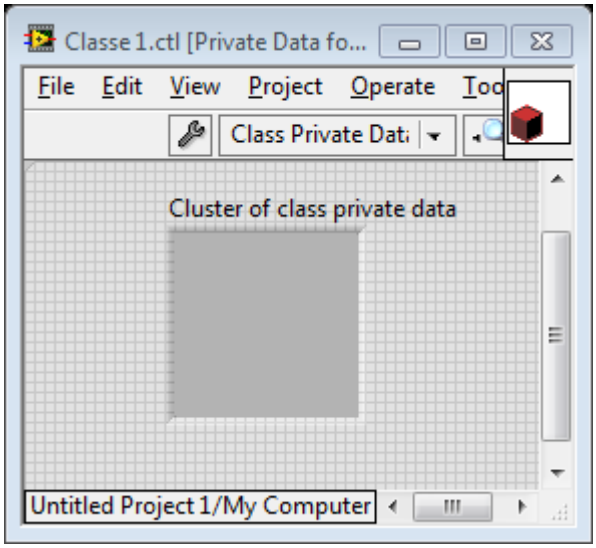

Figura 24 - Dados privados

Observe que essa “caixa” nada mais é do que um cluster vazio. Para definir os atributos da classe basta que o programador preencha o cluster com os atributos. Para isso ele simplesmente acessa ao menu de controles e os coloca dentro do cluster. Em seguida os valores default dos controles devem ser preenchidos:

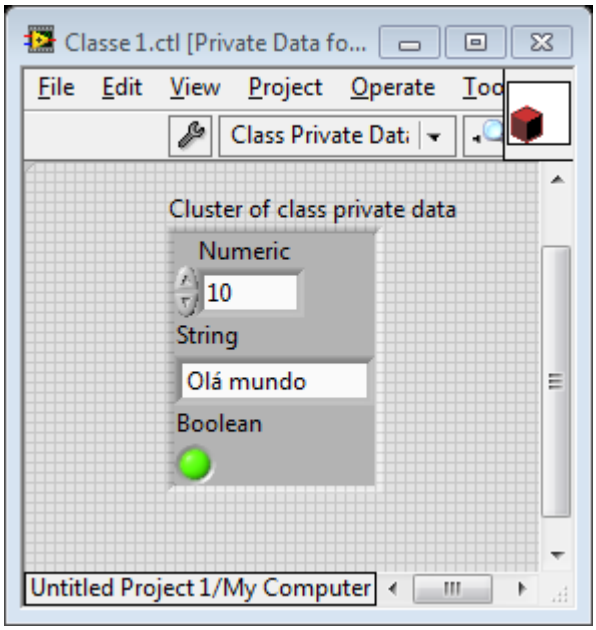

Figura 25 - Dados privados

No exemplo mostrado o usuário definiu uma variável numérica, uma variável do tipo *string* e uma do tipo booleana. Nesse instante temos declarada a classe *Classe 1*. Em JAVA essa mesma classe seria declarada da seguinte forma:

```
class Classe_1{
```

```
        private int numeric;
        private string str;
        private boolean boole;
        public Classe_1(){
             Numeric = 10; string = "Olá mundo"; boole = true;
       }
}
```

Para o exemplo em JAVA trocamos os nomes daxs variáveis “string” e “boolean” para “str” e “boole” por causa da proibição do uso de palavras reservadas como nome de variáveis. Observe que como dito anteriormente os valores preenchidos pelo usuário dentro das variáveis do painel frontal já são considerados os valores default a serem colocados em cada novo objeto criado a partir daquela classe. Isso elimina a necessidade de se elaborar um construtor simples para a classe.

Uma vez criada a classe e definidos os seus atributos privados o usuário deve então criar os métodos da classe. Para isso basta que ele crie a partir do projeto uma “vi” dentro da classe. Todas as “vis” localizadas abaixo da classe na árvore de arquivos serão consideradas métodos dessa classe. A figura abaixo apresenta a janela de projeto com um método chamado “Exibir mensagem”.

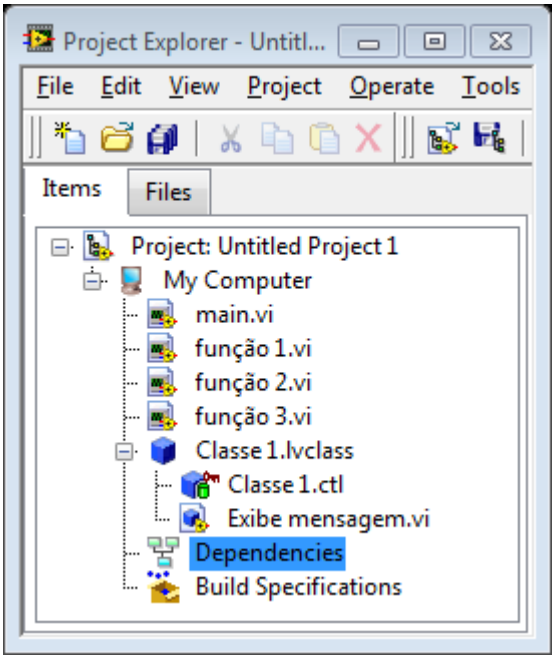

Figura 26 – Método

Para o exemplo vamos usar uma função para calcular uma mensagem em função do valor numérico armazenado no atributo *Numeric*. Observe na figura abaixo o código dentro do método *Exibe mensagem*:

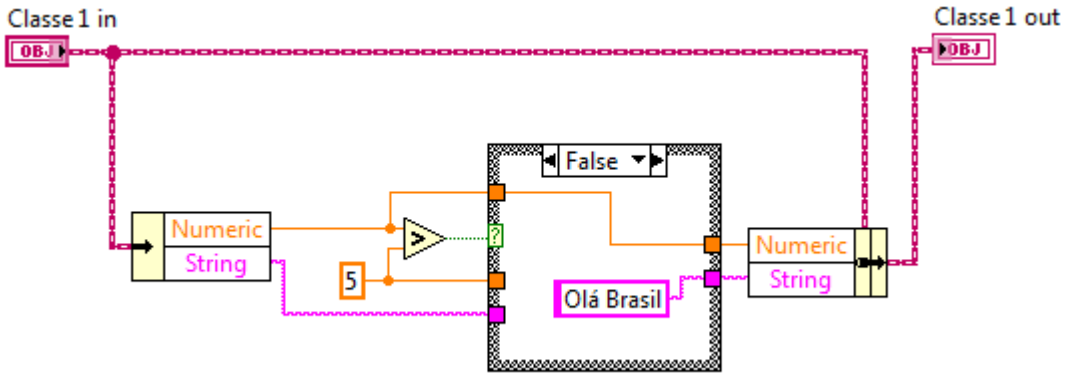

Figura 27 - Código de "Exibir mensagem" caso "false"

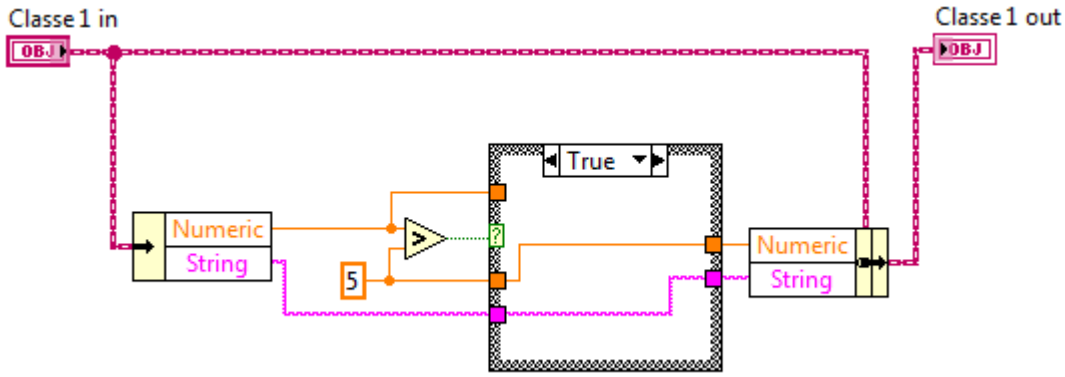

Figura 28 - Método "Exibir mensagem" case "true"

O código desse método analisa o valor dentro do atributo *Numeric* do objeto passado como argumento. Se o valor desse atributo for maior que 5 o método não altera a mensagem e caso o atributo seja menor que 5 o método altera a mensagem no atributo *String* para “Olá Brasil”. Nesse caso o objeto corrente é representado pelo fio roxo no formato de corrente. Para se chamar um método sobre um objeto basta conectar o fio que representa aquele objeto no terminal de entrada do método. Esse procedimento será mostrado em um exemplo a diante. O código JAVA equivalente ao método “Exibir mensagem” seria o seguinte:

```
...
Public void exibe_mensagem(){
      if (this.numeric > 5) {
```

```
            this.numeric = 5;
      else
            this.str = "Olá Brasil";
      }
}
```

Agora que o método *Exibir mensagem* está definido basta que a mensagem seja mostrado ao usuário. Como todos os atributos das classes em LabVIEW são sempre privados todos eles precisam de um método de acesso. A figura abaixo mostra a árvore de projetos com um método de acesso para o atributo *String*:

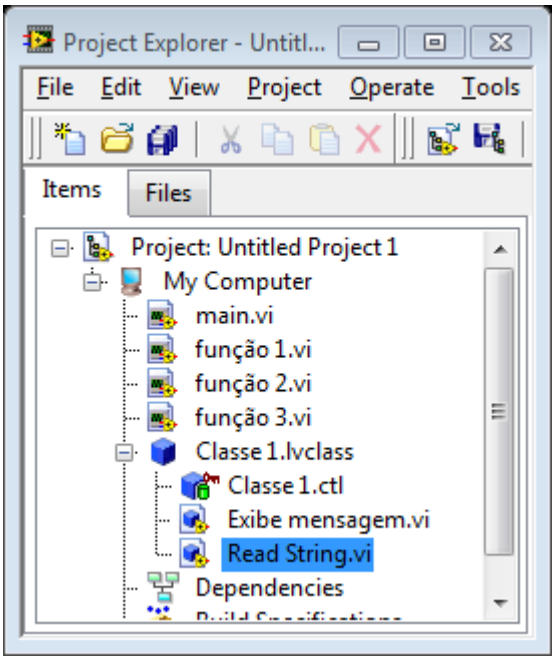

Figura 29 - Método "Read string"

Observe o código para o método *read string*:

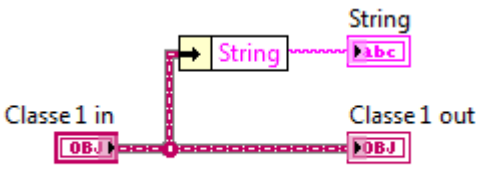

Figura 30 - Método "Read String"

Percebemos que a única tarefa desempenhada por esse método é retornar o valor do parâmetro *String* do objeto tipo *Classe 1*. O fio roxo representa o objeto. Todos os seus atributos estão

contidos dentro desse fio, que oferece uma espécie de blindagem contra acessos externos. O único lugar onde seus dados se tornam transparentes é dentro dos métodos da classe a qual ele pertence. Se a função *unbundle by name* (representada pelo bloco retangular com uma seta para a direita) fosse usada fora de um método da classe o compilador retornaria um erro.

Para fechar o exemplo vamos demonstrar uma aplicação completa da classe que acabamos de criar. Para se montar o código a partir da árvore de projetos basta que o programador abra o bloco diagrama da função que ele deseja editar e arrastar os elementos da árvore para o diagrama:

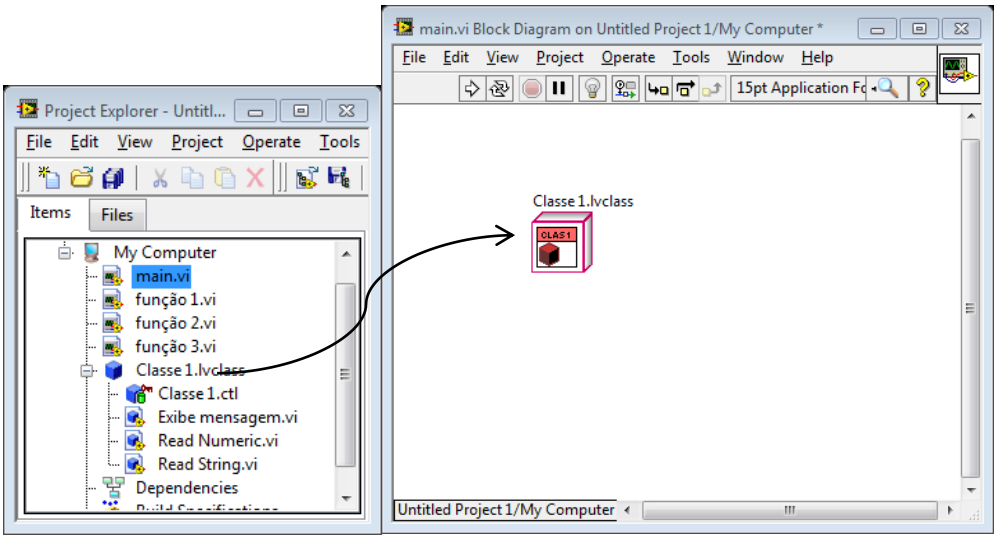

Figura 31 - Drag and drop

O código abaixo mostra a criação de dois objetos do tipo *Classe 1*. Recapitulando o que foi explicado anteriormente cada vez que um fio em LabVIEW é bifurcado uma nova cópia do objeto é criada, ou seja, temos dois objetos. Em se tratando de objetos a linguagem trabalha somente com passagem de parâmetros "por valor". Em seguida o método exibir mensagem é chamado para esse objeto e então o resultado é mostrado na tela para o usuário. Para auxiliar nesse exemplo criamos um método de escrita para a variável *Numeric* de forma análoga à criação do método de acesso à variável *string*.

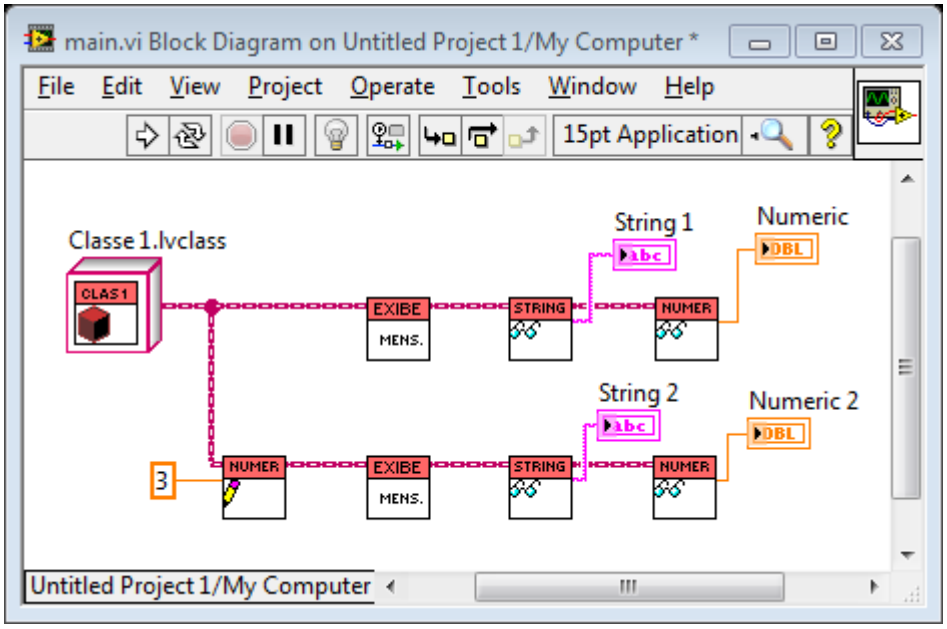

Figura 32 - Código orientado a objeto

A figura mostra no canto esquerdo superior a declaração do tipo de classe. Observe que no fio que se origina desse bloco existe uma bifurcação. Isso faz com que o dado seja conduzido “em duas direções diferentes”, criando então dois objetos do tipo “Classe 1”. Mas como nomear os objetos? Em linguagens tradicionais de programação o *label* do objeto serve para que aquele objeto em particular seja identificado, tanto pelo compilador quanto para o programador, ao longo do código. Como no LabVIEW o objeto é identificado pelo fio não faz sentido falar em *labels*. Para modificar um objeto específico basta identificar o fio pelo qual o objeto de interesse “passa” e conecta-lo a um método pertencente à sua classe. Após a bifurcação temos então dois objetos. O primeiro deles é ligado diretamente ao método “Exibir mensagem” enquanto que o segundo tem seu atributo *Numeric* alterado para “3” através do método de acesso *Write numeric*, criado de forma análoga aos métodos de acesso. Em seguida os valores dos parâmetros são exibidos para o usuário através dos métodos *Read string* e *Read numeric*. O resultado da execução desse pequeno exemplo no painel frontal é o seguinte:

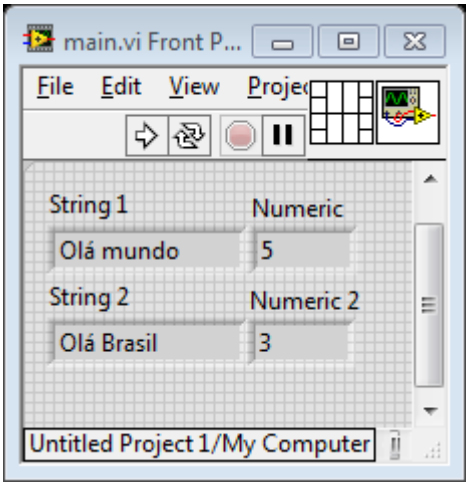

Figura 33 – Resultado

O código do programa acima escrito em JAVA ficaria da seguinte forma:

```
class Classe_1{
       private int numeric;
       private string str;
       private boolean boole;
 public Classe_1(){
            Numeric = 10; string = "Olá mundo"; Boolean = true;
}
public void exibe_mensagem(){
      if (this.numeric > 5) {
            this.numeric = 5;
      else
            this.str = "Olá Brasil";
      }
public int read_numeric() {
      return this.numeric;
}
public String read_string() {
      return this.str;
}
public void write_numeric (int num) {
            this.numeric = num;
      }
Class Principal{
      Public static void main(){
            Classe_1 objeto1 = new Classe_1();
```

```
            Classe_1 objeto2 = new Classe_1();
            objeto2.write_numeric(3);
            objeto1.exibe_mensagen();
            objeto2.exibe_mensagen();
            System.out.println(Objeto1.read_numeric() +
                        Objeto2.read_numeric() + objeto1.read_string() +
                        objeto2.read_string());
}
```

Para o exemplo usamos todos os termos explícitos da linguagem como *this* e a palavra *private* para que o exemplo fique mais claro. O código mostrado acima em JAVA geraria as mesmas saídas obtidos através do LabVIEW. Percebemos, no entanto, que mesmo sabendo que o código JAVA poderia ser mais enxuto do que o apresentado no exemplo, o diagrama LabVIEW é de compreensão e depuração muito mais simples e direta. Esse é um dos principais objetivos dessa linguagem, ser mais acessível e simples de se usar.

7.2. Estendendo uma classe

Estender uma classe é o mesmo que criar uma classe filha. Vamos explicar como se estender uma classe em LabVIEW continuando o exemplo anterior. Para isso vamos criar uma nova classe, usando os mesmos procedimentos já demonstrados. A figura abaixo mostra a tela de projeto com a nova *"Classe 2"* ainda sem nenhum método definido. Para fazer com que a *"Classe 2"* herde da *"Classe 1"* basta acessar o menu, clicando com o botão direito sobre a *"Classe 2"* no projeto:

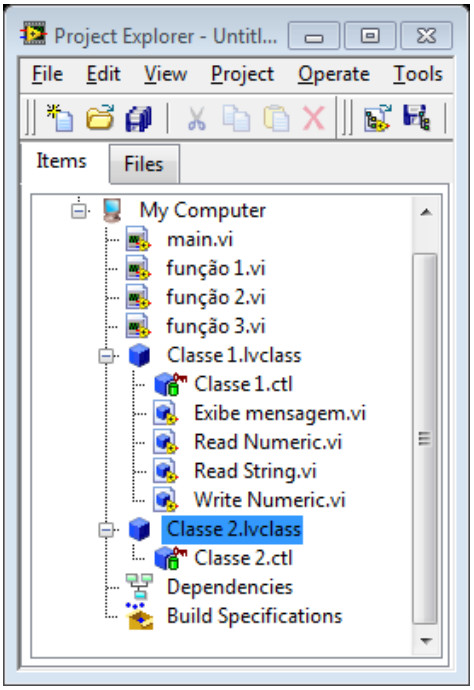

Figura 34 - Criação da Classe 2

Ao abrir o menu de herança a seguinte janela é mostrada:

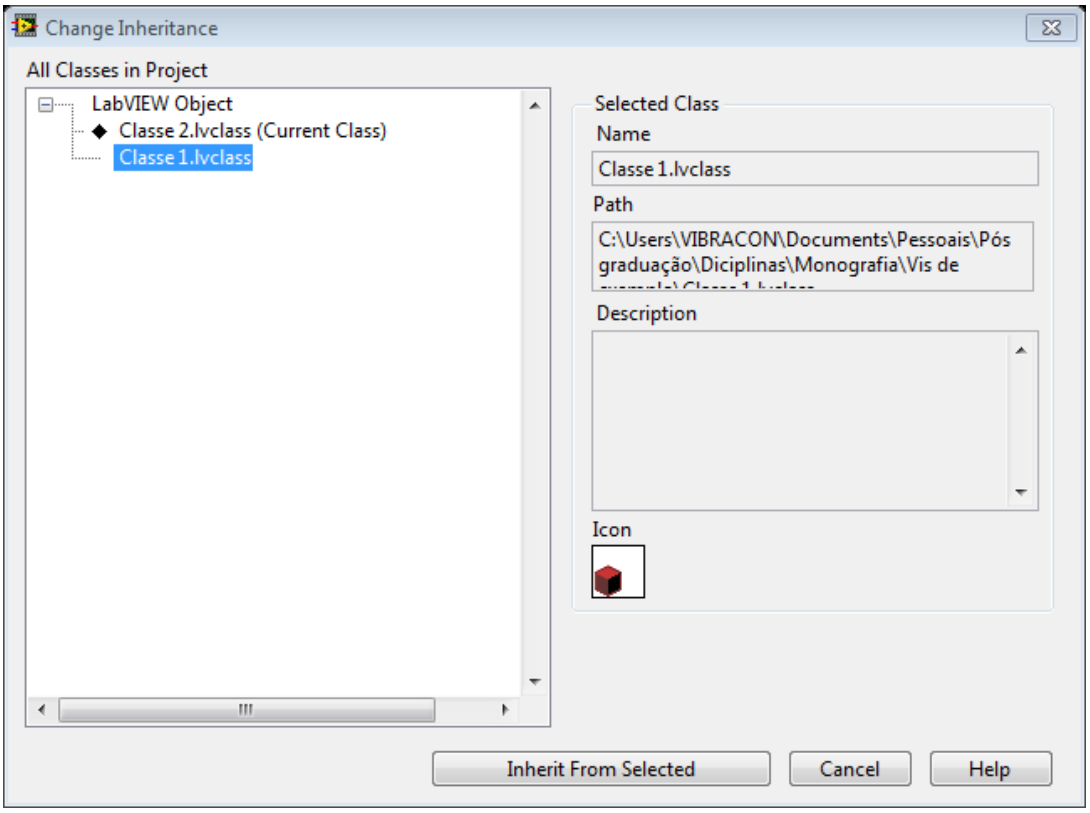

Figura 35 - Menu de herança

A janela mostra todas as classes no projeto corrente. A classe que será afetada pela seleção fica marcada com um losango preto. Isso indica que ela não pode ser selecionada, pois uma classe não pode estender ela mesma. Nesse caso, para fazer com que a *"Classe 2"* herde da *"Classe 1"* basta selecionar a *"Classe 1"* na lista e clicar em *"inherit from selected"* (herdar de selecionado). A partir desse ponto objetos do tipo *"Classe 2"* poderão ser passados como parâmetros para métodos pertencentes a classe *"Classe 1"*. Para auxiliar o programador, o LabVIEW gera automaticamente uma árvore hereditária para o acompanhamento da estrutura do programa. Essa árvore pode ser visualizada a partir de uma opção na janela de projeto:

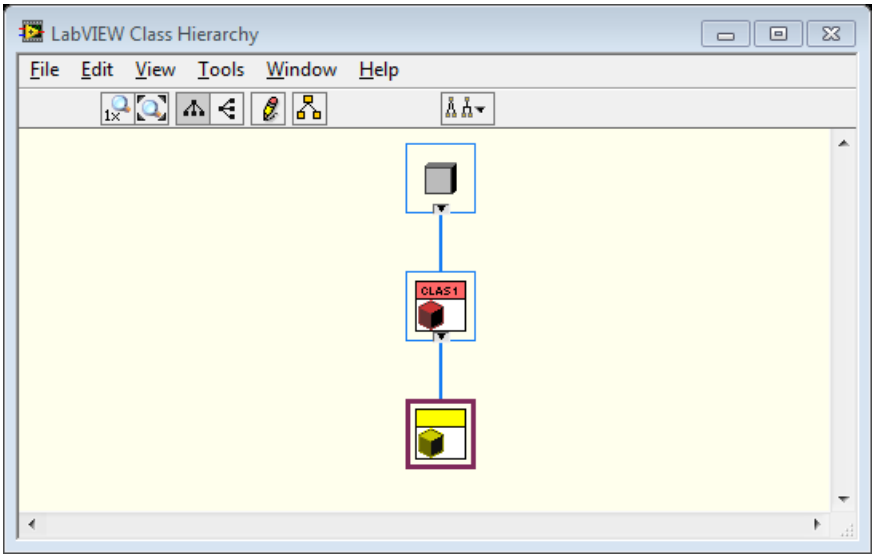

Figura 36 - Hierarquia de classes

Veja que a "*Classe 1"* herda de *"LabVIEW Object"* e que *"Classe 2"* herda de *"Classe 1"*. Vamos demonstrar agora como os objetos das classes interagem com os métodos.

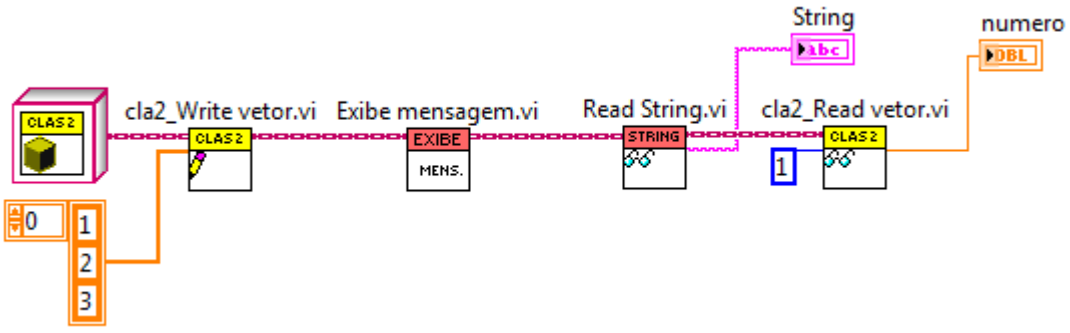

Figura 37 - Trecho de código OO

A figura mostra a declaração de um objeto do tipo *"Classe 2"*. Essa classe possui como atributo privado apenas um vetor de *"double"*. O primeiro método na linha de chamada é um método de acesso ao vetor privado da *"Classe 2"*. Esse método recebe como parâmetro um vetor e o coloca no atributo *"vetor"*. Como o objeto declarado é do tipo *"Classe 2"* esse método é executado sem problemas. Em seguida o objeto é entregue para um método pertencente a *"Classe 1"*. Como *"Classe 2"* herda de *"Classe 1"* esse objeto também pode ser trabalhado por métodos de "Classe 1". No momento da criação do objeto tipo "Classe 2" o construtor implícito de *"Classe 1"* foi executado para colocar os valores default nos atributos relativos a *"Classe 1"* que estão no objeto declarado. O último bloco do código é justamente o método de acesso ao parâmetro privado de *"Classe 2"*. Ele recebe um valor de índice e retorna o elemento correspondente a esse índice. Após a execução do código o painel frontal se apresenta da seguinte forma:

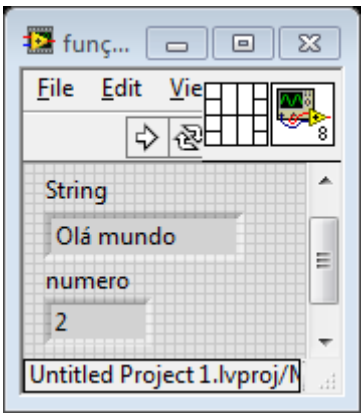

Figura 38 - Painel frontal

Como dito anteriormente, os valores default dos atributos de *"Classe 1"* são (10, Olá mundo, true). Vemos aqui que, mesmo criando um objeto do tipo *"Classe 2"*, os valores dos atributos de *"Classe 1"* foram inseridos corretamente no objeto.

Agora vamos demonstrar o que ocorre quando tentamos passar objetos de uma classe que não possui relação com a *"Classe 1"* isso foi criada uma terceira classe que não se relaciona de nenhuma forma com *"Classe 1"* nem *"Classe 2"*. Observe a figura abaixo:

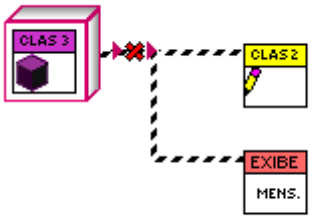

Figura 39 - Fios quebrados

No caso acima os fios ficam pontilhados, indicando que não houve uma conexão entre as classes. Quando isso ocorre o compilador não permite que o programa execute.

7.3. Upcast e Downcast

Assim como em outras linguagens o LabVIEW permite que objetos sofram *upcast* e *downcast.* O *upcast* é uma operação realizada sobre um objeto que faz com que ele seja tratado como um objeto de uma classe da qual ele é herdeiro. Em outras palavras essa função tipifica o objeto para uma classe superior na sua hierarquia (Labview, 2011). O *downcast* tem o mesmo efeito da função *upcast*, porém ela tipifica o objeto como sendo um objeto de uma classe abaixo na sua hierarquia. No exemplo anterior temos a *"Classe 2"* herdando de *"Classe 1"*. A figura abaixo mostra como podemos fazer o *upcast* de um objeto *"Classe 2"* para um objeto *"Classe 1".*

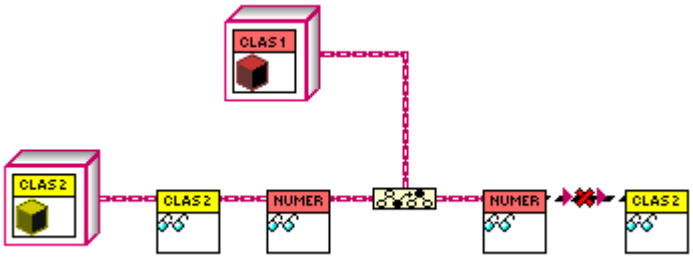

Figura 40 – Upcast

No início do código foi declarado um objeto do tipo "Classe 2". Percebemos que tanto métodos de "Classe 2" quanto métodos de "Classe 1" são aceitos pelo objeto. Em seguida o objeto sofre um *upcast.* A função que realiza essa operação é o pequeno bloco retangular localizado no centro do código. Ele recebe um objeto no seu terminal de entrada, localizado

no seu lado direito. Esse objeto é o que será alterado. Na entrada localizada na parte superior a função recebe o tipo de objeto para o qual o objeto corrente deve ser convertido, que nesse caso trata-se de um objeto tipo *"Classe 1".* O objeto convertido é obtido no terminal de saída, localizado na parte esquerda do bloco. Percebemos que após passar pelo *upcast* o objeto é aceito por métodos de *"Classe 1"* porém não é mais aceito por métodos de *"Classe 2",* provando que o objeto agora se comporta como um objeto tipo *"Classe 1"* puramente. O funcionamento da função *downcast* é análogo ao de *upcast.* Quando tentamos tipificar um objeto *"Classe 2"* para um objeto tipo *"Classe 3"* observe o que acontece:

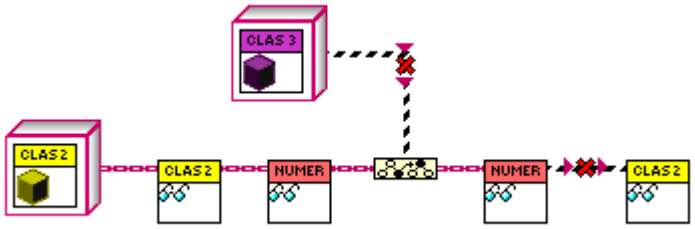

Figura 41 - Upcast não realizado

O compilador detectou que *"Classe 3"* não possui parentesco com *"Classe 2"*. Dessa forma o fio do terminal de entrada da função que realiza o *upcast* permanece pontilhado, indicando que aquela entrada não é aceita. Todo esse processo ocorre durante a programação. Não é necessário que o usuário execute o programa para que erros dessa natureza sejam detectados. Observe a figura abaixo que mostra uma operação de *downcast.*

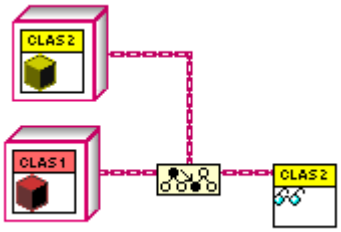

Figura 42 – Downcast

Percebemos que após a realização do *downcast* o objeto passa a ser aceito por um método *"Classe 2".*

8. PADRÕES DE PROJETO

Trouxemos toda a discussão sobre orientação a objeto implementado em LabVIEW até aqui para responder a pergunta: Porque implementar uma lógica de programação mais elaborada em uma linguagem que se desenvolveu inteiramente sem a necessidade do paradigma da orientação a objeto? A resposta é simples: A orientação a objeto permite que sejam implantados em LabVIEW padrões de projeto de eficiência constatada, aumentando o grau de robustez do software gerados por essa linguagem (Kerry, 2010). Nesse tópico iremos abordar dois dos padrões de projeto OO mais conhecidos entre os desenvolvedores e como eles são implantados em LabVIEW.

8.1. Sigleton

O propósito do padrão *singleton* é garantir que uma classe tenha sempre uma única instancia de objeto na memória e que todas as chamadas de métodos se refiram sempre a aquela instância da classe (Moehring, 2010). Quando se cria uma classe algumas vezes é uma grande vantagem que o programa sempre se refira a uma mesma instância global. Um bom exemplo disso é se estivermos trabalhando com uma classe que representa um banco de dados. Seria desastroso se por acidente o programa instanciasse outro objeto dessa classe, criando um segundo banco de dados. Criar uma instância global de uma classe significa que todas as funções do programa podem acessar os mesmos dados contidos nesse objeto, sem o risco de que essas funções por algum motivo alterem esses dados de forma incorreta.

Para exemplificar a construção de uma classe *singleton* em LabVIEW devemos primeiro explicar o conceito de *shift registers*. Esse recurso do LabVIEW é uma variável que só pode ser instanciada na borda de algum frame de laço de repetição e que serve para passar valores de uma interação do loop para a interação seguinte. Observe o pequeno exemplo abaixo:

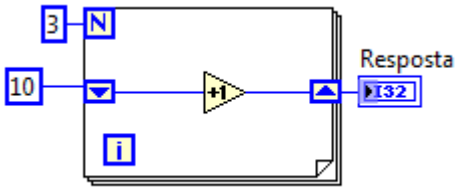

Figura 43 - Shift register

O código JAVA equivalente a esse diagrama seria:

```
int a = 10;
      for  (int i = 1; i<=3; i++){
      a = a + 1;
      }
System.out.println("Resposta = " + a);
```

O *shift register* é representado pelas setas na borda do loop. A cada interação o valor presente no bloco da parte esquerda (seta para baixo) do loop é atualizado pelo valor conectado ao bloco da parte direita (seta para cima) da interação anterior. Funciona como um túnel que leva a informação de uma interação para a outra. O valor "10" ligado ao bloco vigora somente durante a primeira iteração. Nas iterações seguintes, na caixa da esquerda, vigora o valor passado para a caixa do *shift register* na borda direita da iteração anterior.

Observe o código abaixo. Ele apresenta um exemplo de uma classe *singleton* em JAVA.

```
public class Singleton {

      private static Singleton instance;
      private  Singleton() {//codigo de inicialização}

      public static getInstance() {
               if (instance==null)
                        instance = new Singleton()
      return instance;


Singleton obj = Singleton.getInstance();
obj.metodo1();
```

Percebemos a partir do exemplo que o construtor dessa classe é privado, ou seja, somente métodos da classe *singleton* podem dar origens a objetos. O único método dessa classe é justamente o método que retorna uma instancia desse objeto. Esse método se chama "get instance". Ele se encarrega de verificar se o atributo "instance" aponta para "null" e se isso ocorre ele cria um novo objeto, caso contrário retorna o objeto existente. Isso garante que o acesso é feito sempre ao mesmo objeto e que não teremos jamais dois objetos dessa classe.

O conceito aplicado no LabVIEW funciona de forma muito semelhante. Observe na figura abaixo o código do método análogo ao método *get_instance* do exemplo apresentado em JAVA.

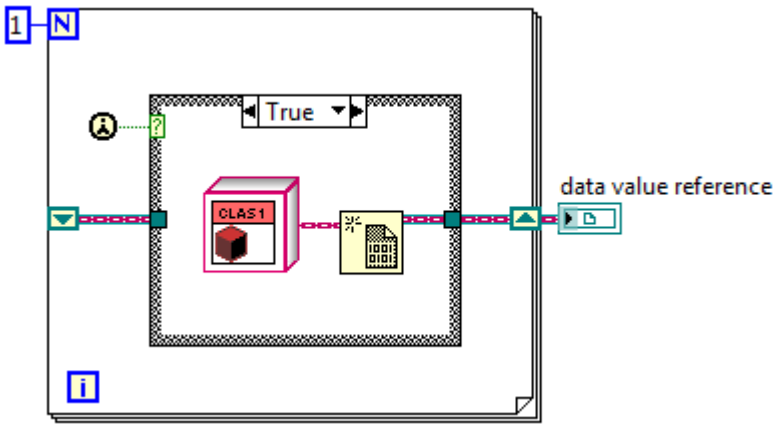

Figura 44 - Método get_instance

Esse método possui uma função chamada de *first call* representada por um circulo amarelo ligado ao terminal "?" da estrutura "case" que retorna "true" somente na primeira vez que esse método é executado dentro de uma aplicação. Sendo assim, na primeira vez que o método é chamado ele cria um objeto tipo "Classe 1", cria uma referência (ou ponteiro) para esse objeto através da função *new data value reference* e coloca seu valor *shift register* do loop *for*. Como esse loop está configurado para iterar apenas uma vez esse código executa apenas uma vez quando o método é chamado. Na próxima vez que esse método for invocado o valor da ultima interação do loop, ou seja, da última vez na qual o método foi invocado, estará disponível na saída do lado esquerdo da estrutura de repetição. Enquanto isso a função *first call* irá retornar *false*, uma vez que já não é mais a primeira vez que a função é executada. A função *new data value reference* recebe um objeto como parâmetro e retorna a referência para aquele objeto. Dessa forma conseguimos burlar a prerrogativa do LabVIEW de sempre fazer cópias "por valor" dos objetos. A figura abaixo mostra o frame do case *false.*

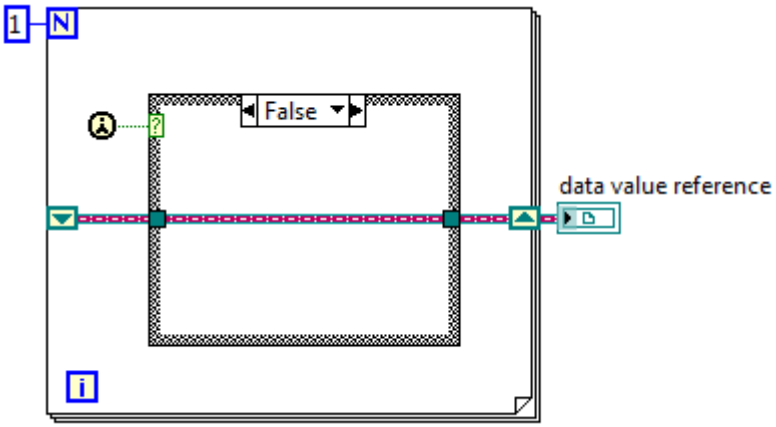

Figura 45 - Case false

O case *false* simplesmente "recircula" a referência do objeto já criado dentro do método, além de liberar a referência no terminal *Data value reference*. Sendo assim, todas as vezes que esse método for chamado ele irá sempre retornar a referencia para o mesmo objeto.

Diferentemente dos atributos, os métodos podem ser declarados públicos ou privados, sendo assim esse método sempre é declarado privado, para poder ser acessado somente por outros métodos pertencentes à classe.

Observe como ficaria a estrutura do método *Exibe mensagem* já mostrado durante a parte introdutória dessa monografia se a classe "*Classe 1*" fosse uma classe *singleton:*

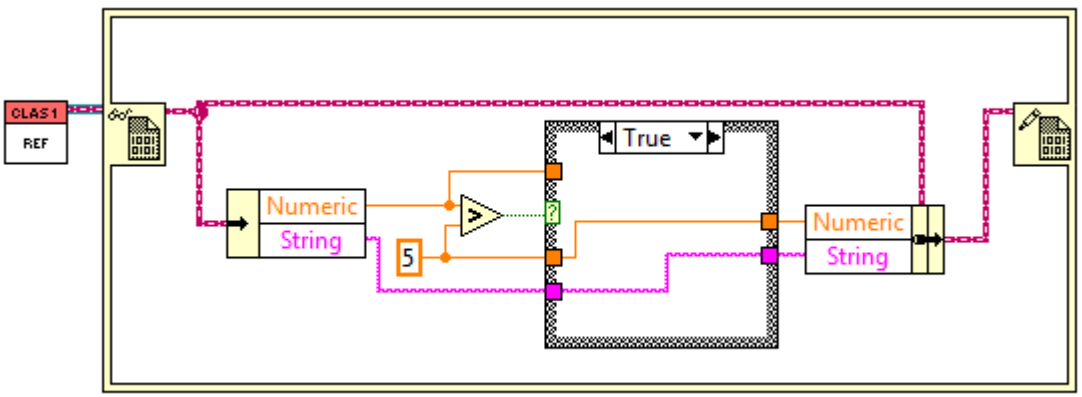

Figura 46 - Reestruturação de "exibe mensagem" para classe singleton

O código se encontra agora dentro de uma estrutura chamada *in place elemet strucrure*. Essa estrutura se encarrega de transformar a referência retornada pelo método *"get reference"* em um objeto que pode ser trabalhado pelas funções. Em seguida o objeto modificado é escrito no mesmo local da memória. Isso nada mais é do que uma

implementação prática de um ponteiro. Porém como a linguagem não preconiza esse paradigma como padrão a sua utilização deve ser explicitada através da utilização desses elementos.

8.2. Factory method

O padrão Factory, que no português significa fábrica, permite várias abordagens em sua utilização. Esse padrão define uma interface para a criação de objetos, sendo que a decisão de qual objeto deve ser instanciado fica a cargo das subclasses herdeiras da interface principal. É um padrão de projeto muito usado em aplicações que necessitam de um alto grau de expansibilidade. Uma das utilidades desse padrão de projeto é a criação de um framework universal que trabalhe com plug-ins, implantados a partir de classes filhas de uma superclasse que rege o framework. Em outras palavras podemos dizer que o padrão *factory* cria uma estrutura, na qual serão "penduradas" diversas classes filhas. Essa estrutura provém condições para que essas classes filhas sejam executadas. Quando surge a necessidade de expansão das funcionalidades do software novas classes filhas podem ser escritas estendendo a superclasse do framework. Dessa forma as alterações ficam isoladas dentro dessas classes e não interferem na execução das outras partes do programa. Essa padrão é usado quando não se sabe exatamente quais tipos de objetos serão criados futuramente (Moehring, 2010), dando alto suporte para a expansão de código.

Em linguagens como JAVA a implantação do padrão factory é feita a partir do uso de classes abstratas. A classe principal, que prepresenta o "plug" do framework, define uma série de métodos que serão sobrescritos pelas classes filhas. Dessa forma todo o código pode ser escrito definindo a ordem das operações a serem realizadas sem, no entanto, conhecer essas operações. No LabVIEW esse processo ocorre de forma semelhante. Define-se uma superclasse que consiste a estrutura do framework.

O exemplo abaixo é bem simples, mas demonstra de forma clara o conceito de *factory pathern* em LabVIEW. Observe o código principal:

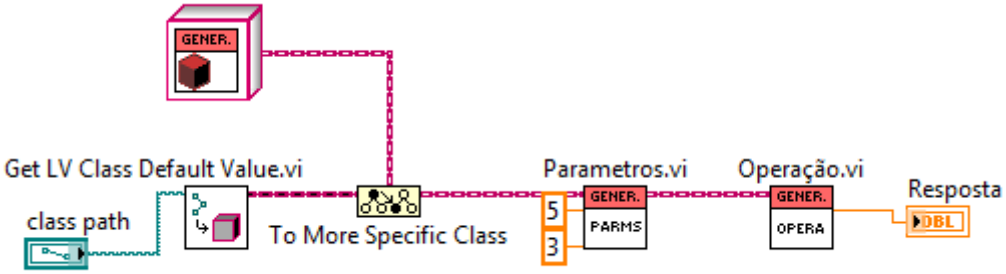

Figura 47 - Factory pathern

A função *"Get LV Class Default Value.vi"* recebe um caminho em disco de um arquivo tipo *lvclass* e retorna o formato do objeto padrão dessa classe. No entanto, esse objeto é criado como LabVIEW Object, o ancestral supremo de todas as classes LabVIEW. Para que esse objeto possa ser trabalhado dentro do framework ele deve sofrer um *casting* para a classe padrão da aplicação. Para o exemplo acima foi criada a classe "Generica". Essa classe é o esqueleto da aplicação, ou seja, todas as classes carregadas para dentro do framework através de "*Get LV Class Default Value.vi*" devem estender a classe "*Generica*". Ela possui dois métodos "abstratos" chamados de "Parametros" e "Operacao" Esses métodos abstratos definem o nome e a interface da função e sempre serão sobrescritos pelos métodos de mesmo nome das classes herdeiras que serão carregadas dinamicamente ao longo da execução do programa. A classe "Genérica" também não possui atributos. Para demonstrar o funcionamento desse exemplo foram criadas duas classes "plug-ins", a classe "*soma*" e a classe "*subtração*". Observe a estrutura hierárquica da aplicação:

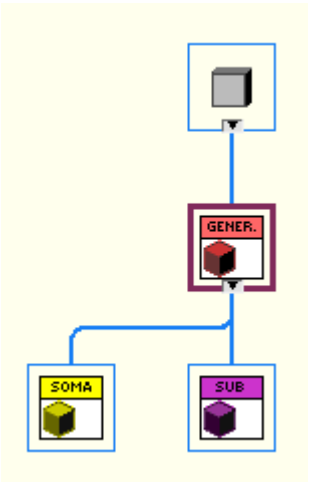

Figura 48 - Estrutura da aplicação

As classes “*Soma*” e “*Subtração*” definem os métodos abstratos “*parâmetros*” e “*operação*” declarados na classe “*Genérica*”. Dessa forma, quando carregados os métodos definidos nas classes herdeiras sobrescrevem os métodos abstratos da classe principal do framework e determinam o comportamento da aplicação. Observe o código do método “*parametros*” definido pela classe “Soma”:

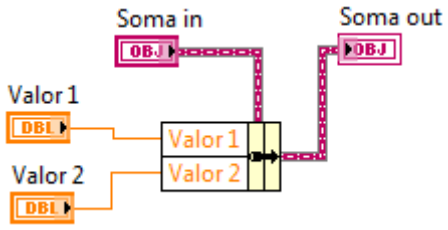

Figura 49 - Método “parâmetros” da classe "Soma”

A classe “Soma” possui dois objetos do tipo *double*. Esse método simplesmente atribui valores aos dois atributos do objeto.

Observe agora o código do método “*operação*” definido pela classe *“Soma”:*

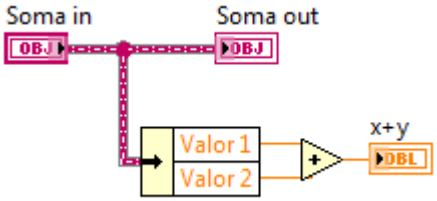

Figura 50 – Método “operação” da classe “Soma”

A classe *“Soma”* possui dois atributos tipo *double* chamados *“Valor 1”* e “*Valor 2*”. O método acima faz a leitura desses dois parâmetros do objeto e retorna o valor da soma entre eles através do indicador “x+y”.

A estrutura da classe *“Subtração”* é muito semelhante ao da classe *“Soma”*, exceto que nessa classe os atributos são chamados de *“Numero 1”* e *“Numero 2”.* Observe no entanto, que a interface dos métodos análogos são iguais, ou seja, possuem sempre dois argumentos e um retorno.

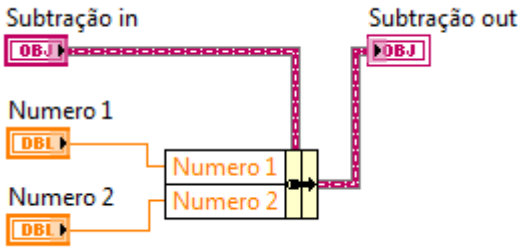

Figura 51 - Método "parametros" de "subtração"

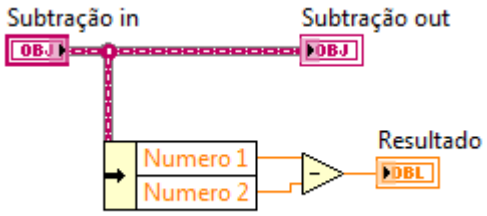

Figura 52 - Método "operação" de "subtração"

Dependendo da classe que for carregada para dentro do código observamos um comportamento diferente da aplicação. Observando agora o código da função principal podemos prever o resultado de sua execução. Se a classe carregada para o framework for *"Soma"* a versão do método *"parâmetros"* executada será a da figura 49. Caso a classe *"Subtração"* seja carregada a versão do método *"parâmetros"* executada será a da figura 51. O mesmo ocorre para o método *"operação"*. A função principal, mostrada na figura 47, carrega uma dessas duas classes para dentro do código. Em seguida realiza um *casting* do objeto tipo *"LabVIEW Object"* para um objeto tipo *"Generica"*. Então o método *"parâmetros"* é invocado, preenchendo-se os parâmetros do objeto com os valores "5" e "3" e em seguida o método *"operação"* é executado, exibindo a resposta no indicador "resposta". A figura abaixo mostra o painel frontal da função principal. O painel frontal possui apenas duas interfaces. A primeira é um seletor, onde o usuário define o caminho em arquivo da classe a ser carregada pelo framework e o segundo é um indicador que mostra a resposta da operação realizada pelo método *"operação"*.

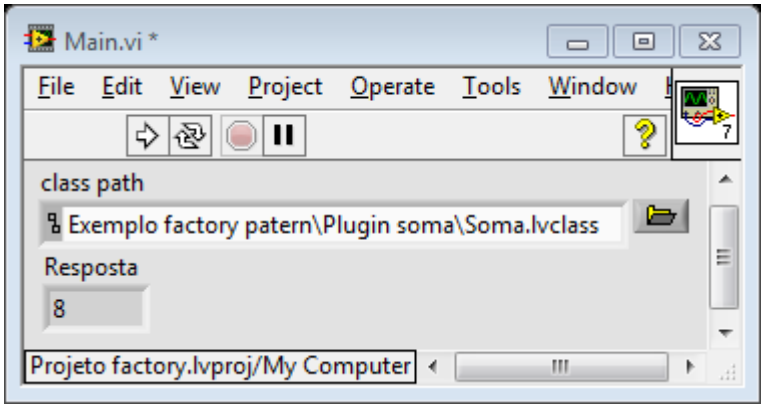

Figura 53 - Carregando a classe "Soma"

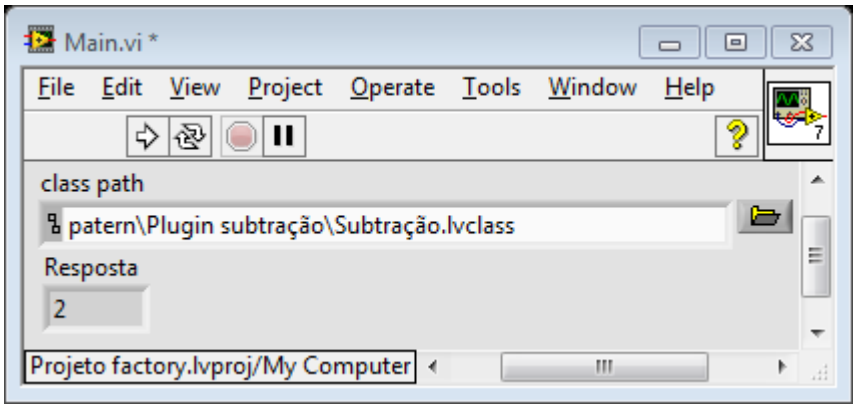

Figura 54 - Carregando a classe "Subtração"

Observe que quando a classe “Soma” é carregada a resposta é “8”, que é o resultado de “5 + 3” e quando a classe “Subtração” é carregada a resposta é “3” que é o resultado de “5 - 3”.

9. CONCLUSÃO

A linguagem de programação LabIVIEW se mostra um ambiente de programação rico e diferenciado. Desde seu surgimento vem quebrando paradigmas e preconceitos entre os engenheiros de software e se provando uma linguagem prática e eficaz acima de tudo. Seu paradigma de programação por *dataflow* construído a partir de um diagrama de blocos pode em um primeiro momento parecer ingênuo e improdutivo, mas quando estudamos a fundo as capacidades da linguagem percebemos que tal fato não ocorre. Certa vez disse Conway (2003) *"I was convinced that this stupid "picture" language I was being told to use was absolute rubbish".* Pouco tempo depois de fazer essa afirmação Conway (2003) escreveu em eu livro: *"In our experience, for a large proportion of projects that come to us, LabView is the best language to program in. The arguments against this statement are reasonably weak."* A partir de exemplos como esse percebemos que ao longo de seus quase 30 anos de existência a programação em blocos tem se mostrado eficaz. A cada ano que passa novas funcionalidades são introduzidas na ferramenta, expandindo ainda mais a sua capacidade e interconectividade com outros ambientes de programação.

A Orientação a Objeto foi somente mais um grande passo dado pela National Instruments no sentido de melhorar ainda mais a ferramenta. Apesar das "simplificações" realizadas no conceito OO sua implementação em LabVIEW se mostra aplicável e suficientemente elaborada para resolver as questões a que se propõe. Através do estudo de linguagens de programação totalmente diferentes na sua forma de "pensar" as coisas, o engenheiro se software expande a sua forma de raciocínio. Não que ele tenha que aplicar o LabVIEW em seus projetos, mas os conceitos por ele aplicados quebram vários paradigmas e certamente enriquecem o raciocínio de um programador.

As aplicações orientadas a objeto escritas em LabVIEW têm ganhado o mercado a cada dia. No Brasil existem várias empresas especializadas no desenvolvimento de programas baseados nesse ambiente. Como explicado na introdução desse trabalho, a National Instruments não se compete a somente desenvolver a linguagem de programação LabVIEW, mas também desenvolve vários tipos de hardware com arquitetura projetada voltada para atingir altíssimos desempenhos quando integrados com programas compilados em LabVIEW. Esses equipamentos são utilizados em uma ampla gama de aplicações industriais. O maior acelerador de partículas existente atualmente, o CERN, é controlado inteiramente por um software desenvolvido nessa linguagem utilizando hardware da National Instrumens (Losito & Masi, 2008).

Vimos que a orientação a objeto pode ser tratada da uma forma mais direta, e que a forma visual de programar pode ajudar bastante no processo de elaboração de um programa. Simplificar o conceito OO, de forma a manter somente aquilo que se mostra mais confiável e útil para o domínio de problema que se deseja tratar, é uma ótima forma de garantir simplicidade e robustez para as aplicações. O paradigma do *dataflow* permite que várias seções de código sejam executadas paralelamente, imprimindo um ganho de desempenho na aplicação e facilitando a elaboração e programas que devem executar *threads* paralelas.

A orientação a objeto sem dúvida simplifica e torna o desenvolvimento de programas complexos uma tarefa mais robusta em LabVIEW. Sabemos que ainda existe muito a ser feito para dar aos desenvolvedores controle pleno e absoluto sobre a forma como a linguagem atua, mas os passos até agora dados pela linguagem merecem o reconhecimento da comunidade de desenvolvimento de softwares e principalmente entre engenheiros e cientistas.

10. BIBLIOGRAFIA

CAMARGO, Daniel; Labview o início de tudo, 2011. Disponível em <http://www.artigonal.com/programacao-artigos/labview-o-inicio-de-tudo-4161467.html> Acesso em 14/01/2012

SUCCESSFUL DEVELPMENT PRACTICES COURSE MANUAL: National instruments training program: apostila cap. 2, p.17, Austin [s.n], 2005

MANUAL DE TREINAMENTO DO LABVIEW BASICO I: apostila. Programa de treinamento National instruments: cap. 3, p. 2, São Paulo [s.n], 2006

NATIONAL INSTRUMENTS. LabVIEW Object-Oriented Programming: The Decisions Behind The Design. Austin [s.n], 2009. Disponível em: <http://zone.ni.com/devzone/cda/tut/p/id/3574 >. Acesso em 16/01/2012.

LABVIEW 2011 for Windows 7 32-btis, versão 2011: Arquivo de ajuda. [S.1] National Instruments, 2011.

KERRY, Elijah. LabVIEW Object – Oriented Design Patterns. In: LABVIEW TRAINNING PROGRAM, [s.n], AustiN. slide 23 – 58.

MOEHRING, Nate. Analysis of "Moving Common OO Design Patterns from Other Languages into LabVIEW". In: Advanced LabVIEW User Group Meeting. [s.n]., 2010, Gilbert, AZ.
Disponível em <https://decibel.ni.com/content/docs/DOC-2875> acesso em: 29/01/2012

CONWAY, Jhon. *A software engineering approach to LabVIEW:* Create more robust , more flexible LabVIEW applications. 1. Ed. Saddle River: Prentice Hall Professional, 2003. 221p.

LOSITO, R., & MASI, A. *CERN Uses NI LabVIEW Software and PXI Hardware to Control World's Largest Particle Accelerator.* 2008. Acesso em 29 de 01 de 2012, disponível em www.ni.com/dvzone: http://sine.ni.com/cs/app/doc/p/id/cs-10795