Canonicus: Um modelo para portabilidade de programas interativos

Marcelo Soares Pimenta, CEC-UFSC
Carlos Alberto Heuser, II-UFRGS
UFRGS/II, Caixa Postal 1501, 90001 Porto Alegre RS
cec1msp@brufsc.bitnet
heuser@sbu.ufrgs.anrs.br

Resumo

Um programa interativo é formado por *componente computacional* e *componente de diálogo*. O componente de diálogo pode ser construído usando alguma Ferramenta para Desenvolvimento de Interfaces com o Usuário (FIU) para definição e manipulação de interfaces. Devido às idiossincrasias de cada FIU, o programa é desenvolvido direcionado para uma FIU específica, necessitando uma série de reformulações em caso de mudança de FIU. Este artigo propõe um enfoque para propiciar portabilidade de programas interativos: o modelo Canonicus. Canonicus é desenvolvido segundo o paradigma de orientação a objetos e pretende prover um suporte mais adequado aos projetistas do programa interativo, mais notadamente o programador. Suas características principais e alguns detalhes de sua implementação sobre FIUs comerciais são descritos.

1 Introdução

Um programa interativo é formado por *componente computacional* (doravante denominado aplicação) e *componente de diálogo* (que implementa a interface com o usuário, abreviada IU). IUs são mais difíceis de construir à medida em que são mais fáceis de usar. Para aliviar esta dificuldade, existem FIUs (*Ferramentas para Desenvolvimento de IUs*), que são conjuntos de rotinas ou programas que auxiliam a definição e manipulação de IUs /Myers 89/.

Um programa interativo é desenvolvido, em geral, direcionado para o uso de uma FIU específica. Uma mudança de ambiente implica em uma série de reformulações dos conceitos de interação e reescrita do código relativo à IU. Devido às idiossincrasias das FIUs, a aplicação precisa se adaptar a cada FIU particular. O programador da aplicação cria diferentes versões da mesma aplicação: uma para cada FIU. Visando diminuir este esforço, tem-se estudado

diferentes possibilidades de prover portabilidade dos programas interativos em relação ao componente de diálogo.

Dois enfoques genéricos para esta portabilidade têm sido utilizados. O primeiro enfoque é o uso de FIUS transportáveis, que são FIUs projetadas para serem portáveis em relação a diferentes ambientes. Em verdade, uma FIU transportável é suportada em cada ambiente, isto é, cada ambiente possui uma implementação da FIU. A portabilidade da aplicação é obtida porque seu código usa rotinas da FIU transportável e evita chamadas diretas aos sistemas operacionais subjacentes. Um exemplo típico de uso deste enfoque é o sistema TAE (Transportable Applications Environment) /Szczur 88/. Uma característica é que a aplicação tem aparência e comportamento ("look and feel") similares em qualquer ambiente, pois há apenas uma FIU.

O segundo enfoque é o estabelecimento de uma camada intermediária entre a aplicação e cada FIU. Em verdade, esta camada intermediária provê uma interface comum com a aplicação para as diferentes FIUs. As chamadas da aplicação à camada intermediária são mapeadas para chamadas à FIU nativa. Neste enfoque, o "look and feel" do programa é o "look and feel" da FIU nativa, característica adequada aos usuários habituados aos princípios de um determinado ambiente. Um exemplo típico de uso deste enfoque é o sistema comercial XVT (eXtended Virtual Toolkit) /Valdes 89/, uma biblioteca de funções implementada inicialmente sobre a Mac Toolbox e o MS-Windows.

Neste artigo são descritas as características principais de um modelo baseado no segundo enfoque para suportar aplicações em diferentes ambientes: o modelo Canonicus – modelo CANÔNico de Interface Com o USuário. O Canonicus diferencia- se do XVT por usar o paradigma de orientação a objetos /Girardi 90/ para definição e uso da camada intermediária e para a implementação do seu mapeamento para as FIUs nativas.

Na seção 2, é destacada a relação entre portabilidade e diálogo interno. Na seção 3, apresenta-se o modelo Canonicus. Os objetos de interface, a base do Canonicus, são descritos na seção 4. A notação do Canonicus e um exemplo de sua utilização são apresentados respectivamente nas seções 5 e 6. A seção 7 discute vantagens e desvantagens do Canonicus. Finalmente, na seção 8 são tecidas algumas observações conclusivas.

2 Portabilidade = Padronização do Diálogo Interno

Os dois enfoques para portabilidade de componentes de diálogo vistos são baseados na uniformização da comunicação entre aplicação e componente de diálogo, de forma que a aplicação use o mesmo protocolo de comunicação para chamar rotinas de diálogo em qualquer ambiente. Para o programador da aplicação não faz diferenca se este protocolo está associado ao primeiro ou ao segundo enfoque e sim se o protocolo é fácil de usar e permite isolamento dos detalhes de IU irrelevantes à sua tarefa.

Este protocolo de comunicação é denominado diálogo interno, em contraposição ao diálogo externo, nome dado à interação entre o usuário final e o componente de diálogo de um programa interativo, conforme figura 1.

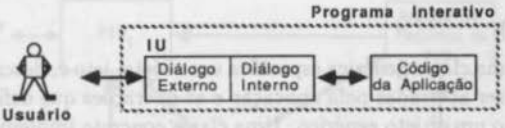

Figura 1: Diálogo Interno e diálogo externo

O diálogo interno lida com os aspectos relativos ao controle de execução das rotinas (da IU e da aplicação) e às trocas de informação entre o componente de diálogo e a aplicação, enquanto o diálogo externo lida com os aspectos relativos a apresentação (o "look") e estilos de interação (o "feel") da IU.

3 Canonicus = Portabilidade + Facilidade de Uso

Em geral, na maioria das FIUs existentes (como por exemplo Mac Toolbox, MS-Windows e X-Windows), os protocolos de diálogo interno são difíceis de usar pelo programador, pois contêm literalmente centenas de rotinas /Myers 90/. Estas rotinas devem ser adequadamente usadas para manipular todos os eventos de entrada (expressos em geral em baixo nível como "botão direito do mouse pressionado" e "cursor na posição [20,30]") e para desenhar e manipular os elementos da IU (menus, janelas, botões, etc.) . Este excesso de detalhes a uma so' vez acaba confundindo frequentemente os papéis do projetista de interface e do programador da aplicação. Mesmo com o uso de editores e arquivos de recursos, estas FIUs ainda deixam muita informação sobre apresentação e forma da IU dentro do código da aplicação, fazendo com que o programador não esteja isolado dos detalhes da IU. Isto termina acarretando prejuízos a almejada independência de diáldogo.

O modelo Canonicus é um modelo de representação de IUs (ver /Hartson 89/ para definição de modelos de representação de IUs) orientado a objetos. Dessa forma, pretende-se uma descrição do diálogo interno de maneira mais adequada aos usuários projetistas (o projetista de interface e o programador da aplicação), diminuindo a dificuldade de definição/manipulação de IUs.

A premissa para a orientação a objetos é que uma IU é um conjunto de objetos que tanto podem ser usados diretamente, quanto especializados para um ambiente de interação específico (uma FIU como a Mac Toolbox ou o MS-Windows, por exemplo) ou para uma determinada aplicação /Barth 86/.

A Mac Toolbox, por exemplo, pode ser considerada orientada a objetos na visão do usuário (diálogo externo), enquanto seu diálogo interno é feito segundo o paradigma convencional de rotinas e funções. O Canonicus tem seu diálogo interno orientado a objetos - independente do paradigma ou metáfora empregados para o diálogo externo. Então, toda comunicação entre IU e aplicação é estabelecida via mensagens direcionadas para os (e/ou a partir dos) objetos pertencentes a IU.

O Canonicus explora a idéia de separar a definição abstrata do diálogo interno das suas implementações através da noção de *classes genéricas* (ou abstratas) e *classes concretas* (ou

de implementação).

Basicamente, uma classe genérica especifica um objeto, isto é, descreve as propriedades do objeto que podem ser acessadas pela aplicação e as operações que definem o comportamento do objeto, definindo um objeto genérico. Uma classe concreta implementa os objetos de uma classe genérica provendo estruturas de dados e rotinas específicas para o seu funcionamento.

De fato, as classes genéricas formam a camada intermediária abstrata acima da camada concreta das FIUs, formada pelas classes concretas. Esta camada intermediária também é denominada camada canônica /Frainer 90/.

Em suma, uma classe genérica possui a descrição dos objetos genéricos e as rotinas que operam sobre eles. As subclasses concretas possuem a definição dos atributos de apresentação dos objetos e a implementação das rotinas descritas nas classes genéricas. As instâncias estão associadas às subclasses concretas. A figura 2 descreve esta hierarquia.

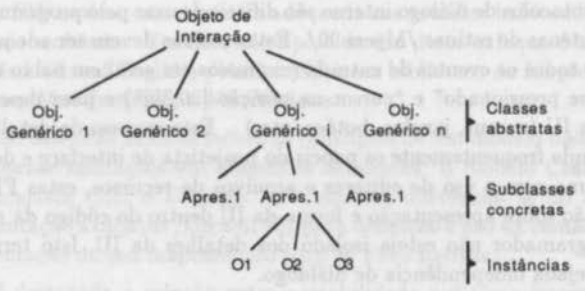

Figura 2: Hierarquia de classes genéricas e concretas

Esta hierarquia de classes (genéricas e concretas) permite diferentes visões destes níveis de abstração, com cada usuário enxergando/manipulando o nível adequado a suas tarefas, conforme figura 3.

O projetista de interfaces trata problemas diferentes em momentos (e níveis) diferenciados:

1. Primeiramente, especifica os objetos de IU a serem usados e as operações necessárias para manipulá-los, definindo e/ou usando classes da camada abstrata;

2. Depois, define a implementação destes objetos incluindo sua apresentação (cores, formas, fontes, etc.), podendo utilizar os editores de recursos disponíveis na FIU específica, e provendo as estruturas de dados e rotinas específicas para a execução das operações, definindo e/ou usando classes da camada concreta.

Ao programador, por sua vez, resta saber *o quê* fazer com relação à IU (o controle dos momentos de interação, por exemplo), não precisando saber *como* fazê-lo. Pode, assim, usar os objetos de IU de modo mais abstrato (acessando a camada abstrata) sem se preocupar em

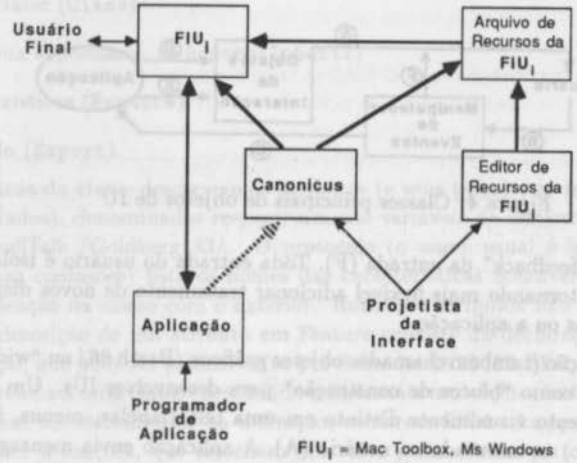

Figura 3: A FIU Canônica e seus usuários

programar todos os passos para exibi-los/manipulá-los, que estão encapsulados na camada concreta.

O usuário final usa a IU da FIU específica com o auxílio do Canonicus.

Uma classe concreta é então onde são descritas os atributos e rotinas de uma FIU específica, que implementa a definição abstrata dos objetos definidos na classe genérica.

O Canonicus foi desenvolvido com estes princípios por três motivos principais:

- a preocupação com a uniformização da comunicação IU-aplicação (diálogo interno) como base para a portabilidade do componente de diálogo;
- a flexibilidade decorrente do uso possível de vários diálogos externos, desde que a tradução do Canonicus para as FIUs que o suportam seja provida;
- facilidade de experimentação, pois é mais direto construir tradutores do que FIUs.

4 Objetos de IU

Há duas classes principais de objetos de IU: manipuladores de eventos e objetos de interação. Na figura 4 estão representadas estas classes de objetos e suas comunicações (rotuladas por letras) com o usuário e a aplicação.

Referenciando a figura 4, um manipulador de eventos reconhece as ações do usuário (B) coordenando a ação dos dispositivos de entrada ("mouse", teclado, etc.), gerando mensagens

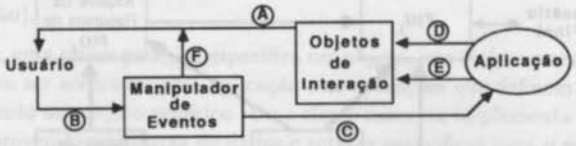

Figura 4: Classes principais de objetos de IU

de eventos (C) e o "feedback" da entrada (F). Toda entrada do usuário é isolada no manipulador de eventos, tornando mais flexível adicionar tratamento de novos dispositivos sem alterar outros objetos ou a aplicação.

Objetos de interação (também chamados objetos gráficos /Barth 86/ ou "widgets" /Scheifler 86/) são usados como "blocos de construção" para desenvolver IUs. Um objeto de interação é todo elemento visualmente distinto em uma tela (janelas, menus, ícones, textos fixos e editáveis, etc.) apresentado ao usuário (A). A aplicação envia mensagens aos objetos de interação para chamar rotinas de interação (D) e passar parâmetros e resultados de computações (E).

Objetos de interação podem ser combinados formando um objeto composto ou painel, que pode ser manipulado como uma unidade. Painéis são ligados a seus componentes através de atributos contendo uma referência ao componente, que pode por usa vez ser também um painel. Um painel é definido como uma instância de uma classe painel, que é a agregação (composição) das várias classes componentes.

Um objeto genérico de IU pode ser então:

- um objeto primitivo, que pode ser definido sem referenciar outros objetos genéricos ou
- um painel, composto através da combinação de outros objetos genéricos.

O Canonicus possui então 2 hierarquias ortogonais de objetos:

- hierarquia taxonômica, com generalização/especialização/classificação envolvendo classes genéricas, suas subclasses concretas e instâncias dos objetos;
- hierarquia composicional, com agregação/decomposição envolvendo objetos compostos (painéis) e objetos componentes.

5 A notação do Canonicus

A notação do Canonicus é baseada na linguagem orientada a objetos Eiffel /Meyer 88/. Descrever IUs usando o Canonicus é descrever classes (genéricas e concretas) de objetos de IU. Consequentemente, para operar sobre a IU definida deve-se instanciar objetos destas classes e manipulá-los de acordo com o seu protocolo de mensagens.

A definição de uma classe de objetos de IU consiste de:

- o nome da classe (`Class`)
- o nome de sua superclasse, se houver (`Inherit`)
- suas características (`Feature`)
- seu protocolo (`Export`).

As características da classe descrevem os atributos (e seus tipos) e as rotinas (com seus parâmetros associados), denominados respectivamente variáveis de instância e métodos na nomenclatura SmallTalk /Goldberg 83/. O protocolo (o nome usual é interface mas foi trocado para evitar confusões) lista os nomes das características acessíveis fora da classe, ou seja, a comunicação da classe com o exterior. Rotinas e atributos não são distinguidos no protocolo. A descrição de um atributo em Feature consiste da declaração de um nome associado a um tipo, que pode ser primitivo (os pré-definidos `INTEGER`, `REAL`, `BOOLEAN`, `CHAR`, etc.) ou uma referência a uma classe (denotando referência a um objeto, instância da classe). Os atributos podem ser inicializados na definição, conforme Eiffel. As operações (rotinas) podem corresponder a funções, que retornam um valor (como `Criar` na classe `Janela` do exemplo mostrado na seção 6), ou a procedimentos ("procedures"), em caso contrário (como `Remover` na classe `Janela` na seção 6).

A definição das classes genéricas tem as seguintes peculiaridades:

- o rótulo `Generic` antecede o léxico `Class` e
- todas as rotinas são deferidas, i.é, não apresentam descrição de implementação.

Rotinas deferidas são rotinas especificadas na classe (genérica) mas realmente implementadas nas suas subclasses (concretas), podendo ser encaradas como a descrição genérica de um grupo de implementações específicas. Por isto, as classes genéricas são também denominadas classes deferidas e não podem ter instâncias, pois não há rotinas da classe para manipulá-las.

Na figura 2, um objeto (O1) é uma instância de uma classe concreta (apres1), a qual é uma subclasse que contém uma das aparências/implementações possíveis de uma classe genérica (obj. genérico i).

A definição das classes concretas, por sua vez, é feita como segue:

- o rótulo `Concret` antecede o léxico `Class`;
- os atributos podem possuir tipos relacionados à FIU específica (p.ex., `wRecord` em `Jan-Mac` na seção 6), adicionalmente aos tipos primitivos e os referenciais às classes;
- todas as rotinas possuem implementação associada.

Como uma classe concreta é subclasse de uma classe genérica, todos os atributos desta são herdados pela concreta, podendo também ser redefinidos ("overriden"). Os atributos adicionais presentes nas classes (p.ex., `Retangulo`, `Tipo`, `Porta` em `Jan-Mac` na seção 6) são tipicamente necessários para a apresentação dos objetos e/ou à execução das rotinas.

A definição da implementação das rotinas deferidas das classes genéricas é feita através do mapeamento de uma rotina para um conjunto (possivelmente unitário) de rotinas de uma FIU, conjunto este que corresponde à implementação das tarefas da rotina. Isto é ilustrado na figura 5.

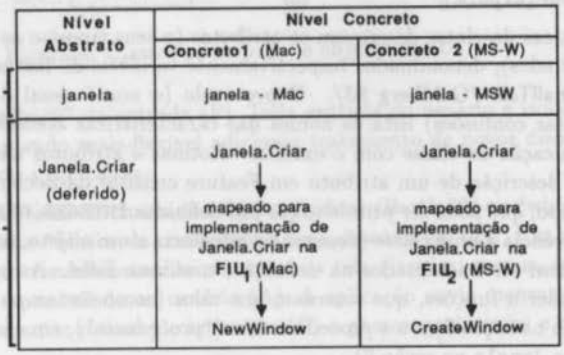

Figura 5: Implementação de rotinas deferidas: um exemplo

As rotinas deferidas são a chave para a portabilidade. O programador *deve* usar as rotinas deferidas, especificadas no nível abstrato. Assim, os detalhes de manipulação/exibição dos objetos, encapsulados no nível concreto, são escondidos do programador, que não precisa tomar conhecimento de qual implementação está sendo executada/utilizada pelo projetista de interface.

Esta definição tem por proposta:

- encapsular no nível concreto as características de apresentação dos objetos de IU;
- servir para a descrição da implementação das rotinas deferidas das classes genéricas;
- servir de base para o processo de tradução do Canonicus para uma FIU específica, descrevendo o mapeamento entre eles.

Uma descrição preliminar de enfoques para a tradução do Canonicus está contida em /Frainer 90/.

6 Exemplo

Este exemplo contém as definições de uma classe genérica `Janela` e duas subclasses concretas `Jan-Mac` e `Jan-MSW`, implementando a classe `Janela` sobre respectivamente a Mac Toolbox /Apple 85/e o MS-Windows /Jamsa 87/. Por questão de espaço, estas descrições não estão completas mas são suficientes para a compreensão da notação e de sua utilização. As descrições completas com todos atributos e rotinas estão apresentadas em /Pimenta 91/. Uma breve descrição do uso das classes é feita após as definições.

6.1 Definições de Classes

`Janela` é uma classe genérica, subclasse de `Objeto-de-interação`, herdando portanto os atributos e rotinas definidos para esta classe. O protocolo (`Export`) de `Janela` é o conjunto de rotinas e atributos acessíveis externamente e inclui, no caso, coincidentemente, todos atributos e rotinas definidas na classe. As características (`Feature`) englobam os atributos `Titulo`, `Classe`, `Icone-assoc`, `Ativa`, `Prox-Jan` e as rotinas `Criar`, `Remover` e `Exibir`. Os atributos `Titulo`, `Classe`, `Icone-assoc`, `Ativa` e `Prox-Jan` dizem respeito respectivamente ao título da janela, à classe da janela (documento, diálogo, etc.), se ela possui ou não ícone associado a ela (se sim, o fechamento da janela faz aparecer o ícone dela na tela), se a janela está ou não ativa e uma referência à próxima janela. A rotina `Criar` é uma função que cria uma instância de janela, retornando o identificador da janela criada enquanto as rotinas `Exibir` e `Remover` são procedimentos que servem respectivamente para exibir a janela e remover a janela. Todas as rotinas da classe genérica `Janela` são deferidas, não possuindo descrição de implementação.

```
Generic Class Janela
Inherit Objeto-de-Interação
Export

  Titulo, Classe, Icone-Assoc, Ativa,
  Prox-Jan, Criar, Remover, Exibir

Feature
  Titulo     : Texto;
  Classe     : INTEGER;
  Icone-Assoc: BOOLEAN;
  Ativa      : BOOLEAN;
  Prox-Jan   : Janela;

  Criar   (): INTEGER;
  Remover ();
  Exibir  ();
```

`Jan-Mac` é uma subclasse concreta da classe genérica `Janela`, da qual herda seus atributos e rotinas. `Jan-Mac` é a classe concreta que visa implementar `Janela` na Mac Toolbox. Seu protocolo permite visualizar as características acessíveis externamente (`Export`) e as que são encapsuladas. Entre as características encapsuladas temos `Tipo`, `Retangulo`, `wRecord` e `Porta`, que são atributos necessários para a implementação/apresentação de uma janela na FIU Mac Toolbox e que não precisam realmente ser conhecidos pelo programador.

```
Concret Class Jan-Mac
Inherit Janela

Export
```

```
  Tipo, Retangulo, Icone-Assoc, Criar,
  Remover, Exibir
Feature

  Tipo : INTEGER;          -- (RDocProc, dBoxProc, etc...)
  Retangulo : Coord;
  wRecord: Ptr;
  Porta : GrafPort is GetWMgrPort (Porta);
  Icone-Assoc: BOOLEAN is FALSE;

  Criar () is
  do
     Current.Id := NewWindows (@wRecord, Retangulo,
                               Titulo, visivel, tipo, Prox-Jan,
                               TRUE, 0);
     SetPort (Porta);
  end

  Remover () is
  do
     DisposeWindow (Current.Id);
  end

  Exibir () is
  do
     ShowWindow (Current.Id);
  end
```

Tipo identifica a forma básica da janela Mac (RDocProc, p. ex., é uma janela com bordas simples, cantos arredondados, barra de títulos escura e uma "close box" no extremo esquerdo da barra de títulos). Retangulo é a definição das coordenadas da janela. wRecord é um ponteiro para uma área de informações sobre a janela. Porta é a parte gráfica associada à janela (obtida através do procedimento GetWMgrPort). Como não existe ícone associado à janela no Mac, Icone-assoc é inicializado com valor FALSE.

A implementação da rotina Criar do objeto genérico Janela na Mac Toolbox é a invocação das rotinas NewWindow e SetPort. Os parâmetros para a execução da NewWindows são definidos na classe Jan-Mac (retângulo, p. ex.) ou herdados da classe Janela (Título, p. ex.) ou de suas superclasses (Visível, p. ex., é herdado de Objeto-de-Interação, superclasse de Janela). O valor retornado pela NewWindows é usado como identificador do objeto janela criado. A palavra reservada Id referencia o identificador do objeto corrente no momento da ativação da rotina. Referências não qualificadas de atributos nas operações significam "atributos do objeto corrente". A variável especial Current pode ser usada, se

desejado, para denotar explicitamente este objeto. Um atributo `Retângulo` não qualificado usado no corpo de uma operação é equivalente a `Current.retângulo`.

`Jan-MSW` é uma subclasse concreta da classe genérica `Janela`, visando implementar janelas no MS-Windows. De modo similar a `Jan-Mac`, `Jan-MSW` possui características encapsuladas que são necessárias à apresentação/implementação de janelas no MS-Windows, como p. ex., `Coordx`, `Coordy`, e `lpParam`. Os parâmetros para a execução das rotinas também podem ser definidos na classe ou herdados das superclasses.

```
Concret Class Jan-MSW
Inherit Janela

Export

  Coordx, Coordy, Largura, Altura,
  Tipo, Icone-Assoc, Criar, Remover, Exibir

Feature
  Coordx,    -- coord x do pt sup esq do retangulo
  Coordy,    -- coord y do pt sup esq do retangulo
  Largura,   -- largura do retangulo
  Altura,    -- altura do retangulo
  Tipo : INTEGER;
  Jan-Pai : Jan-MSW;     -- janela pai
  Menu-Jan : Menu-MSW;  -- barra de menu da janela
  Modulo: Ptr is GetModuleHandle(APPLICATION-NAME);
  Icone-Assoc: BOOLEAN is IsIconic(Id);
  lpParam : FARPROC;

  Criar () is
    do
      Id := CreateWindow (Classe, Titulo, Tipo, Coordx,
            Coordy, Largura, Altura, Jan-pai, Menu-jan,
            Modulo, lpParam);
    end

  Remover () is
    do
      DestroyWindow (Id);
    end
  Exibir () is
    do
```

```
        If Icone-Assoc = TRUE
        then OpenIcon (Id)
        elsif ShowWindow(Id, SHOW-ICONWINDOW);
    end
end
```

É interessante discutir duas características para compreender algumas diferenças entre Jan-Mac e Jan-MSW: o atributo Icone-Assoc e a rotina Exibir.

Icone-assoc, que em Jan-Mac era inicializado com valor FALSE, é agora implementado como a invocação de uma função do MS-Windows que retorna TRUE ou FALSE caso uma janela respectivamente tenha ou não ícone associado.

Exibir em Jan-MSW, ao contrário da opção única de implementação de Jan-Mac, pode ser implementado de duas maneiras devido ao fato de permitir ícones associados a janelas. Se existe o ícone, a execução de OpenIcon exibe a janela representada pelo ícone; senão, a rotina ShowWindow é que deve ser invocada para exibir a janela.

6.2 Uso de Classes

O uso das classes segue tipicamente dois passos:

1. a criação de instâncias das classes e a inicialização de seus atributos e
2. a manipulação das instâncias através das rotinas para prover seu comportamento.

A criação de instâncias (através da operação Criar) tem os seguintes efeitos:

- inicializa todos os atributos dos objetos como os valores da inicialização (pré- definidos ou explicitos) correspondentes;
- executa os procedimentos para a rotina Criar, conforme definidos nas classes.

Objetos de IU são criados como resultado da execução da rotina Criar sobre o nome da classe a que pertencem. Por exemplo, uma janela de nome j1 seria criada da seguinte maneira:

```
j1 := Janela.criar();
```

teria por efeito a criação do objeto j1, instância de uma subclasse concreta da classe Janela. A escolha da classe concreta é feita no momento da tradução.

Como no Canonicus todas as operações (exceto Criar, explicada acima) são relativas aos objetos, para remover ou exibir o objeto janela j1, basta respectivamente invocar:
`j1.remover();` e
`j1.exibir();`
no decorrer do programa.

A seguir, é apresentado um trecho de um programa exemplo escrito em Pascal, extraído de /Apple 85/. Este trecho usa a Mac Toolbox (junto com um arquivo de recursos também descrito em /Apple 85/) para as seguintes funções de IU:

- estabelecer limites para movimentação da janela (linhas 1 e 2);
- criar janela (linha 4);
- estabelecer porta gráfica da janela (linha 5);
- preparar retângulo para edição de texto na janela (linhas 6 e 7) e
- preparar recepção de texto (linha 8).

```
/* Trecho de código usando a Mac Toolbox */
1       WITH screenBits.bounds DO
2          SetRect(dragRect,4,24,right-4,bottom-4);
3       doneflag := FALSE;
4       myWindow:=GetNewWindow(windowID,@wRecord,POINTER(1));
5       SetPort(myWindow);
6       txRect := thePort^.portRect;
7       InsetRect(txRect,4,0);
8       textH := TENew(txRect,txRect);
```

O trecho a seguir possui as mesmas funções do trecho anterior e é também escrito em Pascal. A diferença é a utilização do Canonicus, ao invés da Mac Toolbox. Limites, retângulos e portas gráficas são encapsuladas em classes concretas (definidas pelo projetista de interface), o que possibilita o uso de funções de IU num nível mais alto pelo programador, que fica escondido destes detalhes.

```
/* Trecho de código usando o Canonicus*/
1    doneflag := FALSE;
2    myWindow := Janela.Criar();
3    textH    := Texto.Criar();
```

7 Vantagens e desvantagens do uso do Canonicus

Desde que classes genéricas podem ser acessadas somente através das operações associadas a elas, os usuários das classes genéricas não tem acesso às especificidades das classes concretas. Isto prove um completo encapsulamento das peculiaridades das FIUs subjacentes e torna o uso de classeas genéricas adequado para o programador.

A possibilidade de ter múltiplas classes concretas implementando uma mesma classe genérica é um ponto importante para prover portabilidade. Portanto, dada uma classe genérica, podem haver várias classes concretas, uma para cada FIU para a qual o mapeamento pode ser feito.

Outra vantagem do Canonicus é a reusabilidade. O enfoque orientado a objetos adotado possibilita definir novas classes/componentes de IU a partir das classes primitivas existentes, que provêm um conjunto de objetos que podem ser usados como "blocos de construção", já devidamente testados e depurados.

A principal desvantagem do Canonicus é que o controle da interação fica a cargo do programador, uma vez que as mensagens aos objetos de interação são ativadas sob controle da aplicação. Este tipo de controle é denominado *controle interno* /Hartson 89/ e é típico em FIUs conhecidas por "toolboxes", como a Mac Toolbox e o MS- Windows. O Canonicus manteve este tipo de controle mas sua alteração está sendo investigada.

8 Conclusões

Neste trabalho foram apresentadas as características principais do modelo Canonicus, que visa propiciar a portabilidade de programas interativos desenvolvidos sobre algumas FIUs comerciais (por enquanto, a Mac Toolbox e o MS-Windows) e diminuir a dificuldade de definição/manipulação de IUs.

Canonicus é uma camada orientada a objetos padronizada entre as FIUs e a aplicação e é implementada através de mecanismos de tradução, com seus objetos e operações mapeados para elementos e operações das FIUs subjacentes.

Canonicus pode servir como suporte a uma metodologia de desenvolvimento de IUs, com a separação dos papéis dos usuários projetistas (projetista de interfaces e do programador de aplicações) e com suas hierarquias de classes propiciando projeto de IUs com níveis diferenciados de detalhes. O uso de uma mesma notação para especificação (classes genéricas) e implementação (classes concretas) de IUs evita os erros decorrentes da tradução de um formalismo de projeto para um de implementação e beneficia as tarefas dos usuários projetistas, mais notadamente o programador. Este pode usar a IU sem se preocupar em conhecer ou fornecer os detalhes para sua execução.

A continuidade desta pesquisa desenvolve-se em três direções:

- a implementação de tradutores do Canonicus;
- a investigação do X-Windows para propiciar o uso do Canonicus também para aplicações no ambiente UNIX;
- a mudança do tipo de controle de interação do Canonicus.

Bibliografia

/Apple 85/ Apple Computer "Inside Macintosh V. I,II,III", Addison- Wesley, 1985.

/Barth 86/ BARTH, P. "An Object-Oriented Approach to Graphical Interfaces" ACM Transactions on Graphics, V.5 n. 2 Apr 86.

/Frainer 90/ FRAINER, A.S.; PIMENTA, M.S.; PRICE, R.T. "Como Obter Portabilidade de Programas Interativos" Anais X Congresso SBC, Vitória, Julho 90.

/Girardi 90/ GIRARDI, M.R.; PRICE, R.T.: "O paradigma de desenvolvimento por objetos" Revista de Informática Teórica e Aplicada vol. 1, no. 2, 69–98, 1990.

/Goldberg 83/ GOLDBERG, A. "Smalltalk 80: The Language and its Implementation". Addison-Wesley, 1983.

/Hartson 89/ HARTSON, H.R.; HIX, D. "Human-Computer Interface Development: Concepts and Systems for its Management" ACM Computing Surveys V.21 n. 1 Mar 89.

/Jamsa 87/ JAMSA, K. "Windows Programming Secrets", Osborne McGraw- Hill, Berkeley, USA, 1987.

/Meyer 88/ MEYER, B. "Object Oriented Software Construction", Prentice-Hall, Series in Computer Science, 1988.

/Myers 89/ MYERS, B. "User Interface Tools: Introduction and Survey" IEEE Software, Jan 89.

/Myers 90/ MYERS, B. et alli "Garnet: Comprehensive Support for Graphical Highly Interactive User Interfaces". IEEE Computer Nov 90.

/Pimenta 91/ PIMENTA, M.S. "Um Modelo Canônico de Ferramenta para Desenvolvimento de Interface com o Usuário", Dissertação de Mestrado, CPGCC-UFRGS, Jan 1991.

/Scheifler 86/ SCHEIFLER, R.W.; GETTYS, J. "The X Windows System" ACM Transactions on Graphics V.5 N.2 Apr 86.

/Szczur 88/ SZCZUR, M. "Transportable Applications Environment - An Integrated Design-to-Production UIMS", Ninth Annual Conference and Exposition to Computer Graphics Applications, Proc., Mar 88.

/Valdes 89/ VALDES, R. "A Virtual Toolkit for Windows and the Mac", BYTE Mar 89.