

Capítulo 20

AdvPL e Protheus

O AdvPL (*Advanced Protheus Language*) é uma linguagem de programação desenvolvida pela Microsiga e que contém todas as instruções e funções necessárias ao desenvolvimento de um sistema, independente de sua complexidade.

O PROTHEUS, por outro lado, é uma tecnologia que engloba um Servidor de Aplicação e as Interfaces para conexão com o usuário. É o Protheus que executa o código AdvPL e o devido acesso à base de dados.

Faz parte do Protheus uma solução ERP que engloba, além das funcionalidades descritas nos capítulos anteriores, mais de 30 verticais aplicadas a áreas específicas de negócios e o Configurador, que é um programa que permite, de forma fácil, customizar o sistema às necessidades do usuário.

O Ambiente Protheus

O Protheus é constituído de um conjunto de *Softwares* que compõem as camadas de funcionalidades básicas aos serviços de aplicação, interface, banco de dados e repositório, conforme o diagrama da Figura 20.1.

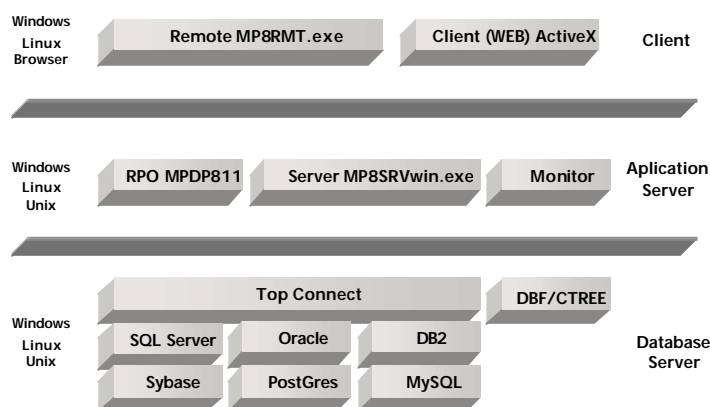


Figura 20.1 Camadas Básicas do Ambiente Protheus.

Para rodar um programa desenvolvido em AdvPl é preciso antes de mais nada escrevê-lo e compilá-lo. Isto é feito no IDE do Protheus (*Integrated Development Environment*, ou Ambiente Integrado de Desenvolvimento). O objetivo do IDE é facilitar a tarefa de escrever programas: através de cores indica se a palavra escrita é uma instrução, uma variável ou um comentário; organiza a biblioteca de programas em Projetos e administra o Repositório de Objetos, aponta erros de sintaxe, permite o *debug* (execução passo a passo do programa, verificando o conteúdo das variáveis) e fornece assistentes (modelos) de programas.

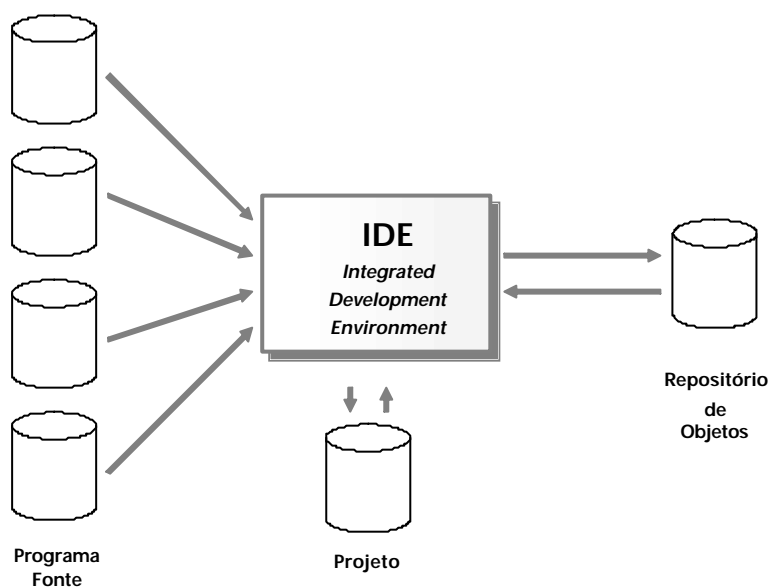


Figura 20.2 Manutenção do repositório de objetos.

Após compilar o programa, o resultado é um objeto. Este objeto é carregado na memória e o Protheus passa a executá-lo. A Figura 20.3 é um diagrama que representa este esquema.

O objeto não é um executável, ou seja, não está convertido para a linguagem nativa do equipamento. Quem faz este trabalho é o Protheus Server em tempo de execução. Por isso o Protheus Server está sempre presente na memória em tempo de execução, permitindo:

- proteger o programa fonte, evitando assim que seja alterado indevidamente, pois somente os objetos são distribuídos com uma

execução mais rápida em função da compilação no IDE.

- uma flexibilização à plataforma de trabalho. Assim, um mesmo programa pode rodar em ambientes Windows, Linux ou mesmo num *Hand Held*, ficando a tarefa de adequação para o Servidor Protheus;
- que o sistema cresça de forma ilimitada pois os objetos ficam fora do executável;
- o uso de macro substituições, ou seja, o uso de rotinas exteriores ao sistema armazenadas em arquivos e que podem ser facilmente ser alteradas pelo usuário, pois o *Server* também interpreta o código fonte em tempo de execução.

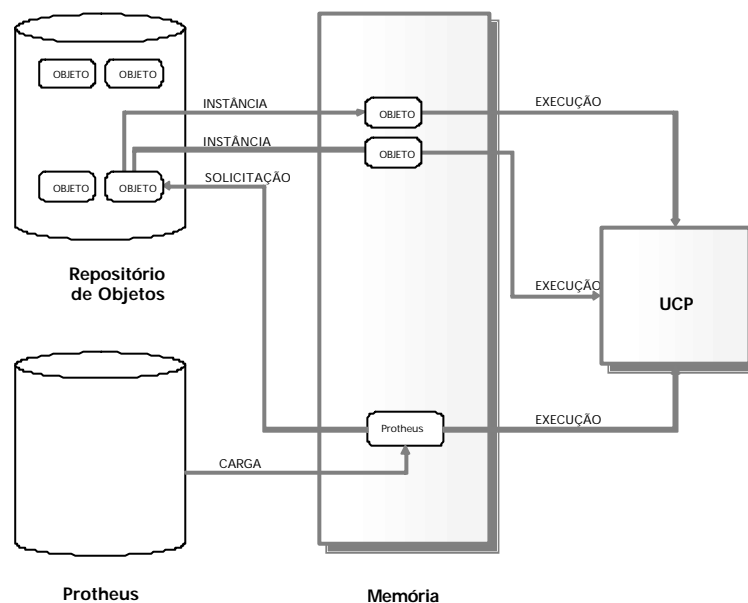


Figura 20.3 Diagrama Esquemático de Objetos Protheus.

O Protheus, por sua vez, é um sistema multi-camada. Entre as diversas camadas, temos a interface de apresentação ao usuário (*Remote*), o tratamento dado para as regras de negócio implementadas (*Server*), o acesso aos objetos do repositório (*Server*), o acesso aos dados disponíveis no Banco de Dados (*Server* ou *Top Connect*) e ao gerenciamento de serviços WEB (*Server*). Neste processo, o Protheus possui, basicamente, quatro aplicativos utilizados com diferentes finalidades:

Protheus Server	Responsável pela comunicação entre o Cliente, o Banco de Dados e o RPO. O nome do executável depende da versão do sistema (MP8SRVWIN.EXE., Microsiga Protheus Server, versão 8 para Windows);
Protheus Remote	Instalado no Server ou na estação. O nome também depende da versão do sistema (MP8RMT.EXE, versão 8);
Top Connect	Responsável pela conversão dos comandos de banco de dados adequando-os ao SQL utilizado.
Monitor	Programa de análise que verifica quem está usando o sistema e possibilita o envio de mensagens ou mesmo derrubar conexões (AP8MONIT.EXE).

Alguns nomes referem-se a um conjunto de programas para facilitar a sua identificação:

RPO	É o arquivo binário do APO (Advanced Protheus Objects), ou seja, os objetos;
Build	Executáveis, DLLs e o RPO completo;
Patch	Atualizações do RPO. São aplicadas por meio do IDE.

A Interface de Apresentação é realizada pelo *Remote* que processa a parte da estação, basicamente tela e teclado. Pode estar gravado no *Server* e ser carregado via rede para a memória da estação. Ou, de preferência, deve ficar armazenado no HD da estação. Pode também ser carregado pelo *Internet Explorer* rodando dentro do próprio *Browser* com o *Protheus Remote ActiveX* e permitindo o acesso ao *Protheus Server* pela Internet, com as mesmas funcionalidades do *Protheus Remote*.

O *Browser* precisa suportar o uso da tecnologia *ActiveX*.

Caso exista algum *Firewall* ou *Proxy* entre o *WEB Server* e o *Browser* que vai acessar o *Protheus Remote ActiveX*, estes deverão ser configurados para permitir o seu *download*.

O Repositório de Objetos é a biblioteca de objetos de todo o ambiente *Protheus*, incluindo tanto os objetos implementados para as funcionalidades básicas do ERP como aqueles gerados pelos usuários.

A Figura 20.4 demonstra a estrutura e a interconexão entre as várias camadas.

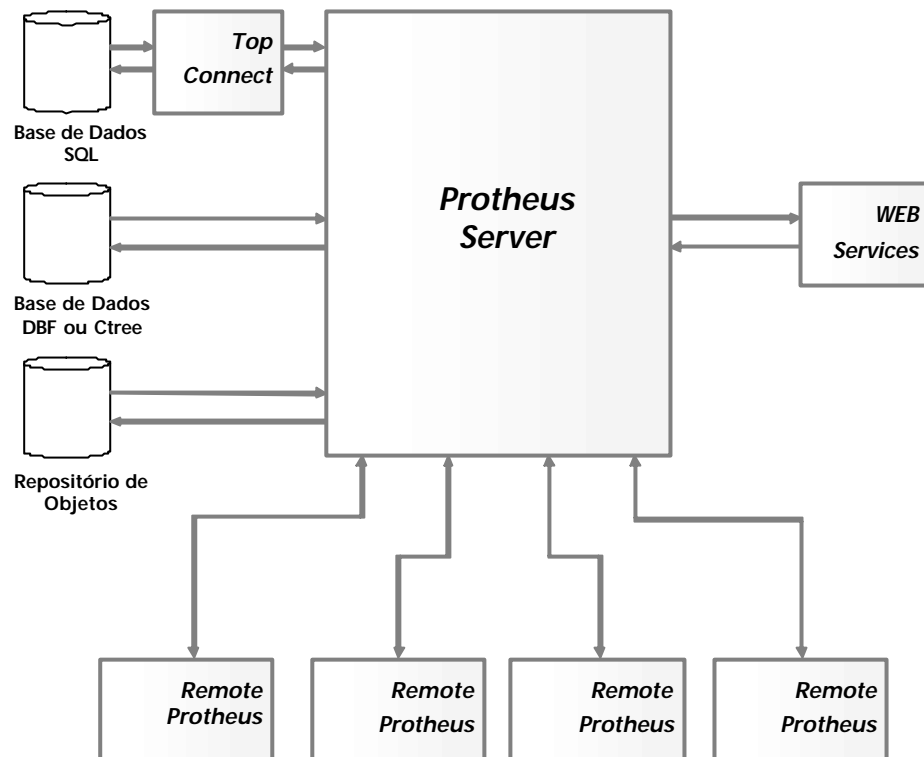


Figura 20.4 Estrutura de Inter Conexão do Protheus.

Ela demonstra também que os dados a serem processados podem estar armazenados ou em bases mais simples como DBF ou Ctree ou em Banco de Dados SQL. No ~~primeiro caso~~ o Server se comunica diretamente com os dados. Em Bancos SQL é a interface *Top Connect* que converte os comandos de entrada e saída adequando-os ao SQL utilizado (SQL Server Microsoft, Oracle, DB2, etc).

Uma vez terminado o processamento do objeto chamado, o mesmo é descartado da memória. Ou seja, o Protheus é um sistema que pode crescer de forma ilimitada pois os objetos, armazenados em um repositório de objetos, praticamente não ocupam espaço no HD (*Hard Disk*) .

Organização e Configuração Inicial do Ambiente Protheus

O Protheus ocupa uma Pasta que tem a seguinte estrutura:

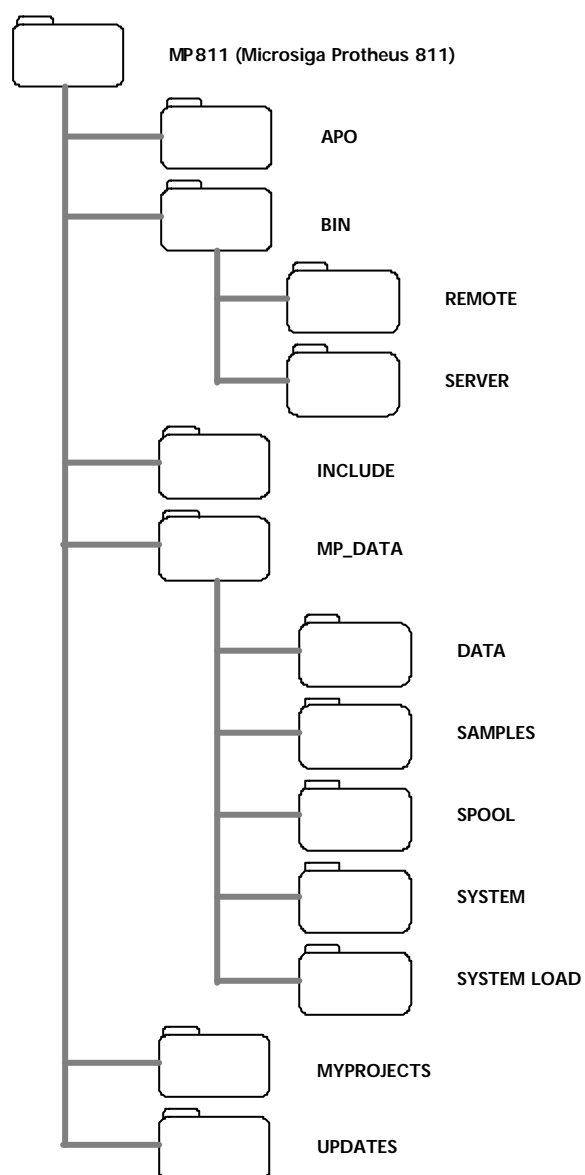


Figura 20.5 Estrutura Básica das Pastas do Protheus.

APO	Contém o arquivo RPO, o repositório de objetos do Protheus.
REMOTE	Reune um conjunto de arquivos executáveis, dll's e arquivos de configuração do sistema para possibilitar o acesso ao Servidor.
SERVER	Reune um conjunto de executáveis, dll's e arquivos de configuração do sistema que compõem o Servidor.
INCLUDE	Contém as bibliotecas necessárias para a compilação de programas Protheus.
DATA	Contém a base de dados DBF ou Ctree.
SAMPLES	Oferece um conjunto de programas exemplo e arquivos ADVPL padrões da Microsiga.
SPOOL	Nesta pasta são gravados os relatórios gerados em disco pelo sistema Protheus.
SYSTEM	Contém os arquivos de menus, os arquivos de configurações e os arquivos de customizações (SXs) do sistema Protheus.
SYSTEMLOAD	Contém o dicionário de dados em formato TXT. É neste arquivo que estão todos os padrões e formatos para a geração dos arquivos de configurações e de customizações (SXs), conforme a localização de país definida pelo usuário na entrada do sistema.
MY PROJECTS	Sugere-se a criação desta pasta para armazenar projetos e fontes das customizações realizadas pelo usuário.
UPDATES	Sugere-se esta pasta para o armazenamento das atualizações a serem aplicadas no sistema Protheus.

Apesar da estrutura ilustrada na Figura 20.5 indicar que as pastas estão subordinadas à pasta MP811 é possível que algumas delas possam estar em máquinas diferentes ou até mesmo em ambientes computacionais diferentes. Para isto é necessário

configurar, ou seja, informar ao Protheus onde está cada uma delas. Este tipo de informação consta nos arquivos de parâmetros de configuração do sistema (MP811SRV.ini e MP811RMT.ini) existentes nas respectivas pastas SERVER e REMOTE. Os parâmetros do MP811SRV.ini são lidos pelo programa MP811SRVWIN.exe logo no início de sua execução. O mesmo ocorre em relação aos parâmetros do MP811RMT.ini pelo programa MP811RMT.exe. A execução destes dois programas é feita por meio de ação do usuário, facilitada pelos atalhos *MP8 Server* e *MP8 Remote*.

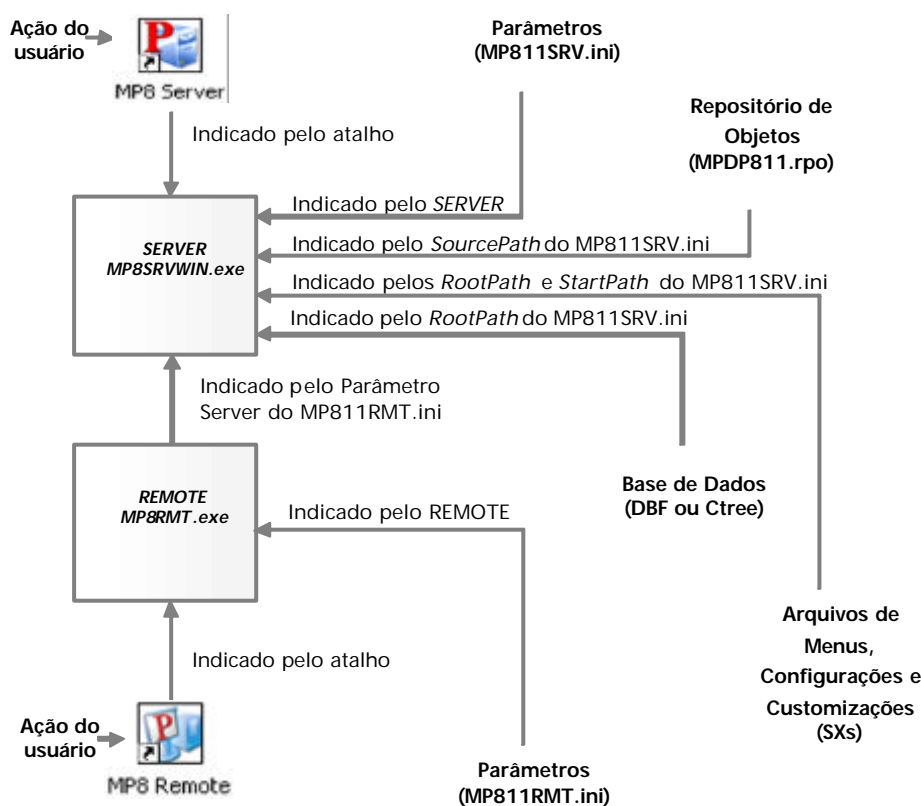


Figura 20.6 Links dos Parâmetros de Configurações.

Para que o Protheus *Server* e o Protheus *Remote* sejam executados, os arquivos MP811SRV.ini e MP811RMT.ini devem estar disponíveis nas respectivas pastas SERVER e REMOTE pois são eles que indicam o endereço das demais pastas conforme a

ilustração da Figura 20.6.

O detalhe de preenchimento das propriedades dos respectivos atalhos Protheus Server e Protheus Remote é demonstrado na Figura 20.7. No atalho do Protheus Server, é necessário que seja informado o parâmetro *-debug* ou *-console*.

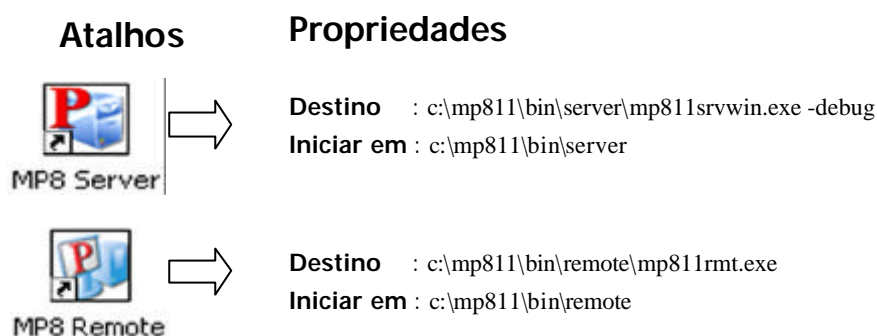


Figura 20.7 Links dos Parâmetros de Configurações.

Os parâmetros que configuram o local do RPO, o Banco de Dados (DBF ou Ctree), os arquivos de menus, configurações e customizações do sistema no arquivo MP811SRV.ini são:

SourcePath	Indica o local de origem dos objetos. É o endereço do Repositório de Objetos (Exemplo: SourcePath=C:\MP811\APO);
RootPath	Aponta para a pasta raiz (inicial), a partir da qual serão localizados os dados (no caso de DBF ou Ctree) bem como o próprio Dicionário de Dados (Exemplo : RootPath=C:\MP811\MP_Data).
StartPath	Indica qual é a pasta dentro da pasta raiz (informada no parâmetro <i>RootPath</i>) que contém os arquivos de menus, os arquivos de configurações e os arquivos de customizações (SXs) do sistema Protheus (Exemplo: StartPath=\SYSTEM\).

Não há necessidade de que os parâmetros estejam em ordem nos arquivos de configuração (.ini) e além dos parâmetros já detalhados existem outros que podem estar indicando a versão do sistema, o tipo de banco de dados, a linguagem do país em que está sendo utilizado e as máscaras de edição e formatação.

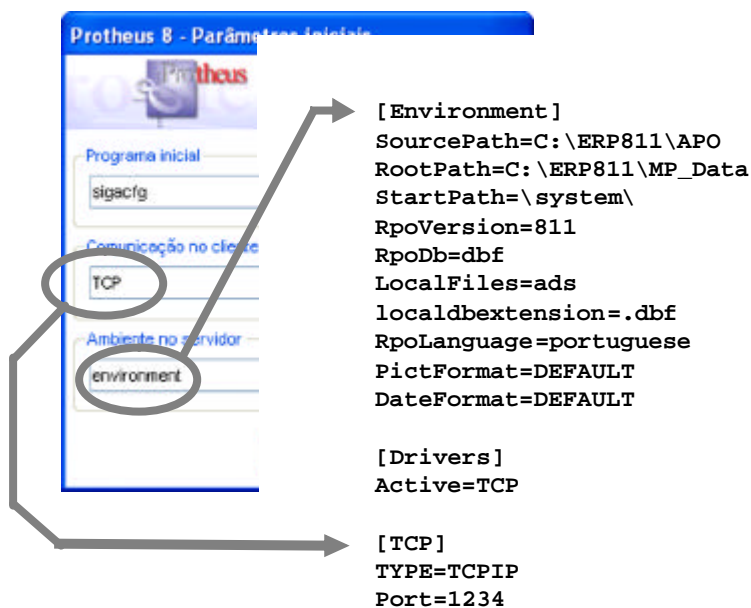


Figura 20.8 Exemplo de um Ambiente em um Arquivo de Parâmetro (MP811SRV.ini).

No exemplo da Figura 20.8, o rótulo [environment] descreve um conjunto de parâmetros que serão inicializados no sistema e os rótulos [Drivers] e [TCP] identificam a comunicação que pode ser estabelecida entre o Protheus *Server* e o Protheus *Remote*.

Outros ambientes podem ser configurados no mesmo arquivo (MP811SRV.ini). É o exemplo da Figura 20.9 que contém outro ambiente (o environmentSQL) para simular como seriam os parâmetros caso estivesse sendo utilizado o SQLServer.

Já o arquivo de parâmetros do Protheus *Remote* (MP8RMT.ini) contém apenas as configurações locais, basicamente as informações necessárias para a inicialização e comunicação com o Protheus *Server*, conforme o exemplo da Figura 20.10.

```

[environmentSQL]
SourcePath=C:\ERP811\APO
RootPath=C:\ERP811\MP_Data
StartPath=\system\
RpoVersion=811
RpoDb=Top
LocalFiles=ads
localdbextension=.dbf
RpoLanguage=portuguese
PictFormat=DEFAULT
DateFormat=DEFAULT

[Topconnect]
Alias=BASE810
ConType=TCPIP
DataBase=MSSQL8
Server=SrvTOP01

[Drivers]
Active=TCP

[TCP]
TYPE=TCPIP
Port=1234

```

Figura 20.9 Exemplo de um Ambiente com Bando de Dados SQL.

```

[Config]
LastMainProg=sigamdi

[Drivers]
Active=TCP

[TCP]
Server=localhost
Port=1234

```

Figura 20.10 Exemplo de Configurações para o Protheus *Remote*.

Neste arquivo os parâmetros podem ser: **LastMainProg** Indica qual é o programa inicial que será apresentado na tela de parâmetros.

Active Indica qual é a forma de comunicação.

Port Indica o número da porta a ser utilizada para a comunicação entre o Protheus *Server* e o Protheus *Remote*. É necessário que a porta utilizada na comunicação seja a mesma em ambos (no MP811SRV.ini e no MP8RMT.ini). Vale ressaltar que a porta 80 é reservada para a Internet e pode causar conflitos caso seja utilizada na comunicação do Protheus.

Server Aponta para o endereço do servidor que pode ser a própria máquina (*localhost*) ou o nome da máquina (Server=Servidor_01) ou mesmo um endereço IP (exemplo Server=172.16.72.41).

Por exemplo, o parâmetro Server=172.16.72.41 no arquivo MP811RMT.ini indica ao Protheus *Remote* o endereço da máquina na qual está funcionando o Protheus *Server*. Esta possibilidade de utilizar o Protheus em duas ou mais máquinas ou mesmo na própria máquina esta ilustrada na Figura 20.11.

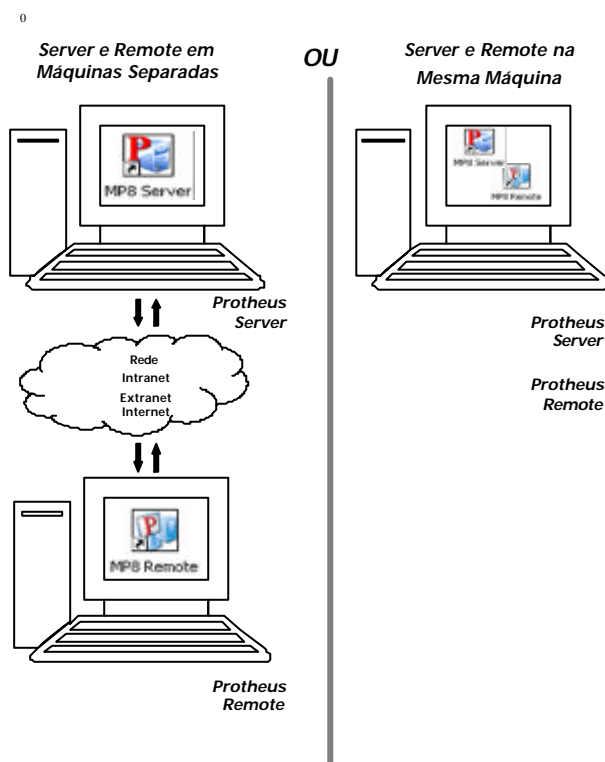


Figura 20.11 Formas de Instalação e Uso do Protheus.

Funcionalidades Abordadas

Antes de falarmos dos comandos e das funções do AdvPL é preciso entender as facilidades que são oferecidas pelo Configurador.

Para tanto usaremos como exemplo um pequeno sistema de Contas Correntes, composto de 2 arquivos:

- 1- Cadastro de Contas;
- 2- Arquivo de Transações

Os seguintes processos serão realizados:

- Criação das 2 tabelas no Dicionário de Dados;
- Atualização do SX2 e demais arquivos SXn;
- Programa de Atualização do Cadastro;
- Três programas para as transações, sendo um com tela cheia (modelo 1), o segundo com um cabeçalho (Modelo 2) e um terceiro mostrando também os dados do cadastro (modelo 3);
- Programa de Consulta;
- Programa de Relatório;

Durante estas etapas serão ensinados os principais comandos do AdvPL, definição de todos os tipos de variáveis e funções utilizadas.

- Desenvolvimento de uma rotina de Workflow que envia um e-mail ao cliente que ficar, após um saque, com saldo negativo e possibilite uma resposta autorizando ou não o saque;
- Uma rotina com Web Services na qual um sistema envia um questionamento para outro informando seu saldo;
- Desenvolvimento de uma página para fazer saques e depósitos através da Web;
- Integração com Excel.

O Configurador

O Configurador é instalado juntamente com os demais programas que acompanham o CD do livro.

Como Iniciar o Configurador

Similar aos outros programas já utilizados, para executá-lo é necessário carregar o Protheus Server e Protheus Remote com apenas duas diferenças: Programa Inicial e a Senha. O programa inicial (sigacfg) deve ser informado na tela de parâmetros iniciais, conforme a tela ilustrada na Figura 20.12.



Figura 20.12 Parâmetros de inicialização do sistema.

Após a confirmação, a validação do acesso é feita conforme tela ilustrada na Figura 20.13.

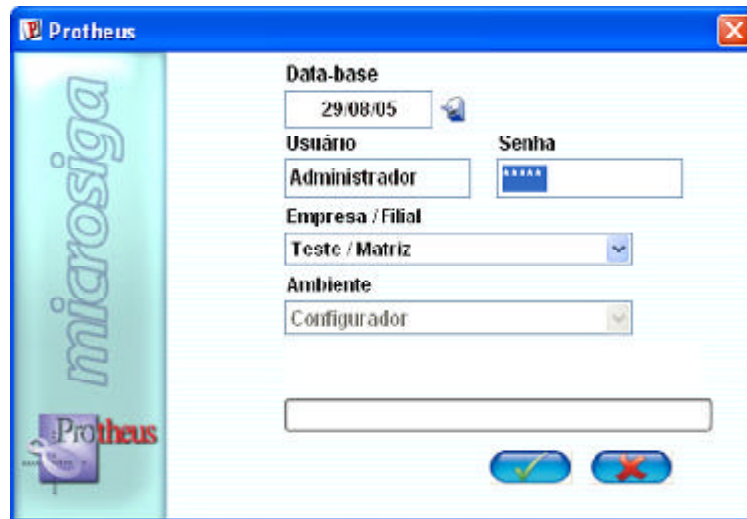


Figura 20.13 Validação do Acesso.

A senha de acesso é admin e, logo após a sua confirmação, será apresentada a tela inicial do configurador, conforme mostra a Figura 20.14.

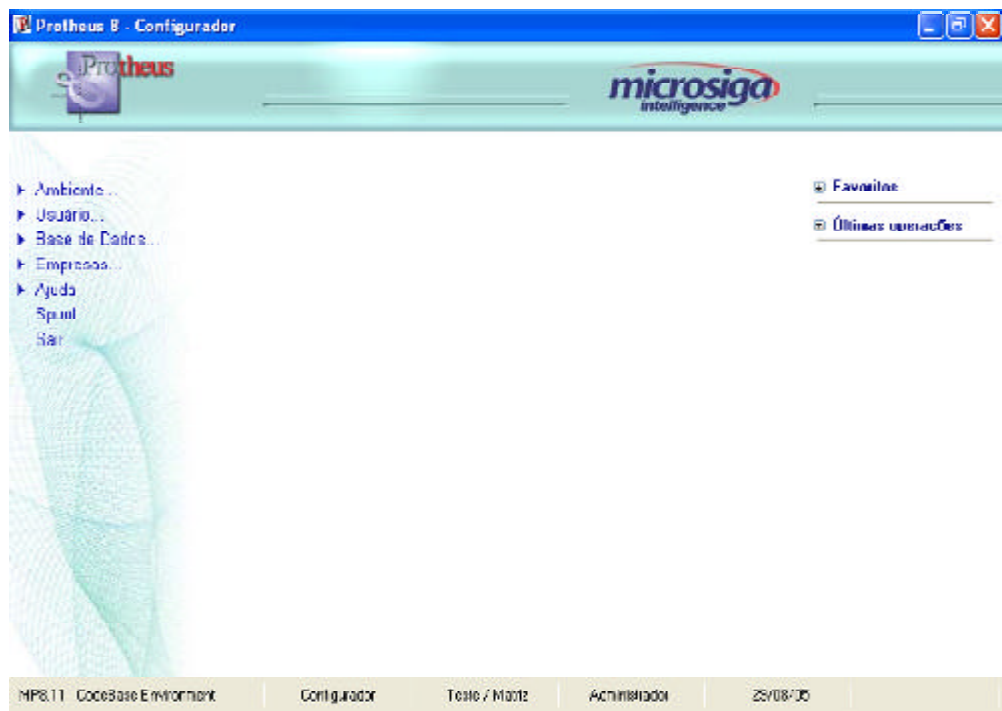


Figura 20.14 Principal interface de acesso às funcionalidades do sistema.

Funcionalidades do Configurador

A customização de um sistema, como o Protheus, para uma determinada empresa consiste em adaptar as suas rotinas para as necessidades do cliente.

A flexibilidade de um sistema, ou seja, sua capacidade de se adaptar (polimorfismo, aquele que assume várias formas) é uma das mais importantes características de uma solução ERP. Alias, Protheus foi o Deus da Transformação na mitologia grega, daí o nome dado ao sistema da Microsiga, que anteriormente se chamava SIGA – Sistema Integrado de Gerencia Automática.

As funcionalidades tratadas pelo configurador é que definem a flexibilidade do Protheus. Flexibilizar sem despadronizar, ou seja, tudo que foi customizado permanece válido, mesmo com o desenvolvimento de novas versões.

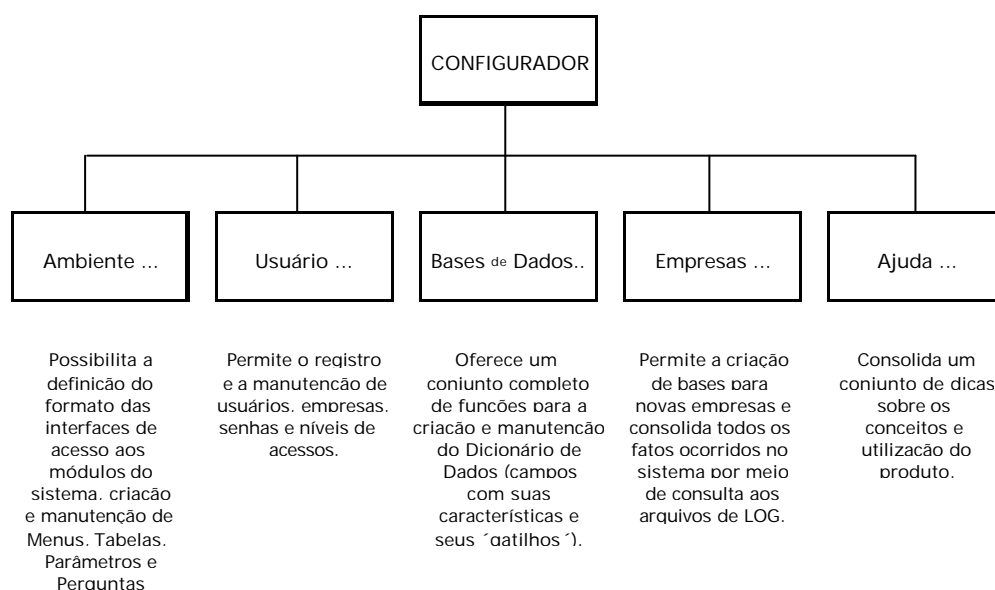


Figura 20.15 Principais funcionalidades do configurador.

O Configurador é o programa básico para o processo de customização do Protheus através da alteração dos arquivos da família SX. Neles, o usuário ou o analista de suporte responsável pela implantação coloca informações que serão utilizadas pelos demais programas do sistema. Estas informações vão de simples parâmetros até complexas expressões e comandos que são interpretadas em tempo de execução. A Figura 20.16 relaciona os objetivos básicos de cada um destes arquivos.

Arquivo	Descrição
SX1	Perguntas e Respostas
SX2	Mapeamento de Arquivos
SX3	Dicionário de Dados
SX4	Agenda do <i>Schedule</i> de Processos
SX5	Tabelas
SX6	Parâmetros
SX7	Gatilhos
SX8	Semáforos
SX9	Relacionamentos entre Arquivos
SXA	Pastas Cadastrais
SXB	Consulta por meio da tecla F3 (Consulta Padrão)
SXC	Lista de Usuários Logados ao Sistema
SXD	Controle do <i>Schedule</i> de Processos
SXE	Seqüência de Documentos (+1)
SXF	Seqüência de Documentos (Próximo)
SXG	Tamanho Padrão para Campos
SXK	Controle de Perguntas (SX1) por Usuários
SXO	Controle de LOG 's por Tabela
SIX	Índices dos Arquivos (1)
SINDEX	Índices dos Arquivos (2)
XNU	Menu de Opções dos Módulos

Figura 20.16 Arquivos da Família SX.

A forma mais simples de customização é através da criação de parâmetros. Um exemplo é o parâmetro ESTNEG (abreviatura de Estoque Negativo). As rotinas que dão baixa do estoque permitem que ele vá a negativo caso seu conteúdo seja S (de sim). E não permitem, caso o conteúdo seja N (de não). O exemplo ilustrado na Figura 20.17 demonstra o tratamento dado dependendo do conteúdo do parâmetro MV_ESTNEG do arquivo SX6.

```
GetMv('MV_ESTNEG') // Traz um parâmetro para a memória

If MV_ESTNEG = 'S' .or. Quant > B2_Saldo
    tratamento normal
Else
    tratamento de erro
EndIf
```

Figura 20.17 Exemplo de Utilização de Parâmetro.

Estes parâmetros (são mais de 4000) são armazenados no arquivo SX6 e sua atualização só pode ser feita no Configurador. O arquivo SX5 é similar ao SX6, onde os parâmetros são mini-tabelas, tais como Cores Válidas, Estados da Federação, Tipos de Produtos, Tipos de Notas Fiscais, etc. É muito usado para facilitar a digitação através de Combo-Box.

De forma análoga temos o arquivo de Perguntas (SX1). A diferença é que neste caso quem atualiza o seu conteúdo é o próprio usuário, em tempo de execução e no momento de sua utilização. Elas são apresentadas ao usuário assim que o programa é carregado. As respostas são gravadas, de modo a serem reapresentadas, como default, no próximo processamento. São exemplos de perguntas típicas:

- imprime de qual cliente a qual cliente (o mesmo para produto, data, vendedor, etc)
- imprime folha inicial?;
- efetua salto de folha por conta ou produto?;
- quantidade de linhas por folha?;
- imprime conta sem movimento? (o mesmo para produto, vendedor, etc);

Até aqui todos os processos de customização são totalmente dependentes do Fonte, ou seja, tudo precisa estar previsto no programa. Este por sua vez explora os parâmetros, agindo de acordo com as suas especificações.

Inserção de Código Fonte

Uma forma mais avançada de customização é o armazenamento de fórmulas ou expressões em determinados campos de arquivos do sistema, fórmulas estas interpretadas em tempo de execução. Assim o próprio usuário pode definir, usando a sintaxe do AdvPL, as rotinas que serão executadas pelo sistema. O cálculo de vencimentos e descontos de uma Folha de Pagamento, a fórmula de um reajuste de preços, a expressão de validação de um campo, a ação de um gatilho, a maneira de como deve ser feito um lançamento automático são exemplos que exploram este recurso.

Este mecanismo de Macro-Substituição é possível pois o Protheus Server é também um interpretador de código fonte. Este processo é inviável em linguagens totalmente compiladas, pois neste caso o código executável já está em linguagem nativa impedindo que comandos sejam traduzidos em tempo de execução.

Exemplo: Reajuste de preços entre o Pedido e o Faturamento.

Imagine que o reajuste de preços possa ser realizado de formas distintas:

- 1) Reajuste Fixo:

SC6->C6_PRCVEN * 1.2;

- 2) Reajuste se preço em dólar:

SC6->C6_PRCVEN * RecMoeda(dDatabase,'2');

- 3) Reajuste pelo dólar:

SC6->C6_PRCVEN*(RecMoeda(dDatabase,'2')/RecMoeda(SC5->C5_EMISSAO,'2'))

No primeiro exemplo, o reajuste será realizado por uma taxa fixa de 20 %, no segundo exemplo pelo respectivo valor em dólar e no terceiro exemplo pela variação do dólar entre a data da emissão do pedido e a data do faturamento.

Estas expressões são armazenadas no arquivo de fórmulas (SM4) de onde são lidas com base no código de reajuste digitado no pedido.

No programa o comando tem esta sintaxe:

Preço := &M4_Formula

O mesmo recurso pode ser utilizado para Validações, Gatilhos e Preenchimento de Campos dos Lançamentos Padronizados, conforme ilustram os exemplos a seguir:

Validações: Dicionário de Dados SX3

Campo Natureza:ExistCpo('SED')

Campo Estado: ExistCpo('SX5','12'+M->A1_EST)

Campo CNPJ: CGC(M->A1_CGC)

Gatilhos: SX7 - Executa a regra gravando no domínio após a digitação do campo D1_VUNIT

Campo: D1_VUNIT

Dominio: D1_TOTAL

Regra: Round(M->D1_VUNIT * M->D1_QUANT,2)

Campos dos Lançamentos Padronizados: Debito no Lançamento Padronizado:

If(SD3->D3_TIPO='MC','33201'+SD3->D3_CC,'11303')

É claro que se podemos escrever uma fórmula ou expressão para ser interpretada em tempo de execução, nada nos impede de escrevermos rotinas mais longas, envolvendo várias linhas de código ou até mesmo um programa completo. Na verdade, o que se escreve são Funções. Neste caso, passa-se pelo processo de compilação, gerando um objeto que é armazenado no Repositório e no campo do arquivo escreve-se o nome da função.

```
U_MinhaRot
Do Case
  Case SB1->B1_TIPO = 'MC'
    Conta := '11302'
  Case SB1->B1_TIPO = 'ME'
    Conta := '11303'
    .....
  EndCase
Return(Conta+SD3->D3_CC)
```

Figura 20.18 Exemplo de Função de Usuário.

Pontos de Entrada

Dentro do processo de Customização falta, no entanto, ainda a possibilidade de se alterar os programas originais do sistema, aqueles que compõem o seu núcleo. Para resolver este problema usa-se os Pontos de Entrada.

Os Pontos de Entrada são pontos pré-determinados onde o usuário pode escrever uma nova Rotina em substituição àquela existente.

O que se faz é substituir uma função escrita pelo programador original por uma outra ou incluir uma nova desenvolvida pelo usuário. É claro que neste caso é preciso todo o cuidado e conhecimento para não prejudicar o conjunto. Os Pontos de Entrada são estabelecidos pelo programador, que documenta todo o processo. A função MT010ALT, ilustrada na Figura 20.19, atualiza o Custo Médio no arquivo SB2 (Saldos Físicos e Financeiros) quando é feita a alteração do custo total de uma mão-de-obra, já que para elas não se calcula o custo médio pelo sistema.

```

////////////////////////////////////
// PONTO DE ENTRADA NA ALTERAÇÃO DO CADASTRO DE PRODUTO           //
////////////////////////////////////

#include "rwmake.ch"

User Function MT010ALT()

    If ALLTRIM(SB1->B1_TIPO) == "MO"

        Dbselectarea("SB2")
        Dbsetorder(1)
        Dbseek(xFilial()+SB1->B1_COD)

        If Reclock("SB2",.F.)
            SB2->B2_CM1 := SB1->B1_CUSTD
        Endif

    Endif

Return Nil

```

Figura 20.19 Exemplo de Ponto de Entrada.

Como Criar Arquivos no Dicionário de Dados

O processo de Customização é completado com a atualização de uma série de outros arquivos (todos da família SX), em especial o SX3 (o Dicionário de Dados).

A idéia do Dicionário de Dados é permitir que o usuário possa incluir ou inibir campos, ou mesmo alterar as propriedades dos campos existentes. Pode ainda criar novos arquivos. Ou seja, os programas ao invés de terem os campos definidos em seu código original, lêem o Dicionário em tempo de execução, montando arrays com as propriedades de cada um. A partir daí sua utilização é normal, através do uso de funções do AdvPl que tornam o trabalho do programador transparente a esta arquitetura.

O objetivo do Dicionário de Dados é permitir que o próprio usuário crie novos arquivos ou altere os campos nos arquivos existentes quanto ao seu uso, sua ordem de apresentação, legenda (nos três idiomas), validação, help, obrigatoriedade de preenchimento, inicialização, etc.

Para exemplificar o uso do Configurador usaremos o sistema de Contas Correntes onde temos um cadastro com Nome, e-mail e Saldo do Cliente e um arquivo de transações que através de seus campos Indicador de Depósito

ou Saque e Valor da Transação, atualiza o saldo da Conta.

A chave de amarração dos dois arquivos é o Nome do Cliente, ou seja, este campo é chave primária no Cadastro da Conta e chave estrangeira no arquivo de Transações. Os campos Número/Item identificam a transação, ou seja, é a chave primária neste arquivo. O Histórico detalha informações sobre a transação.

Os campos e-mail no Cadastro e Arpovação no arquivo de Transações servirão para a rotina de Workflow que enviará mensagens sobre as ocorrências da conta.

O campo Saldo Atual é um campo virtual que mostra na tela de digitação o saldo da conta após cada transação.

A figura 20.20 demonstra os campos e o relacionamento destes dois arquivos

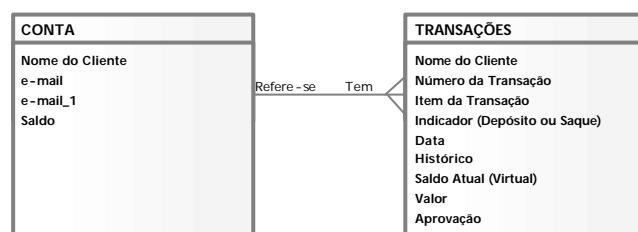


Figura 20.20 Arquivos do Sistema exemplo de Conta Corrente.

Para o cadastramento destes arquivos efetue os procedimentos a seguir:

1. Selecione Base de Dados, Dicionário, Base de Dados. A janela que possibilitará a manutenção do dicionário de dados aparece como mostra a Figura 20.21:

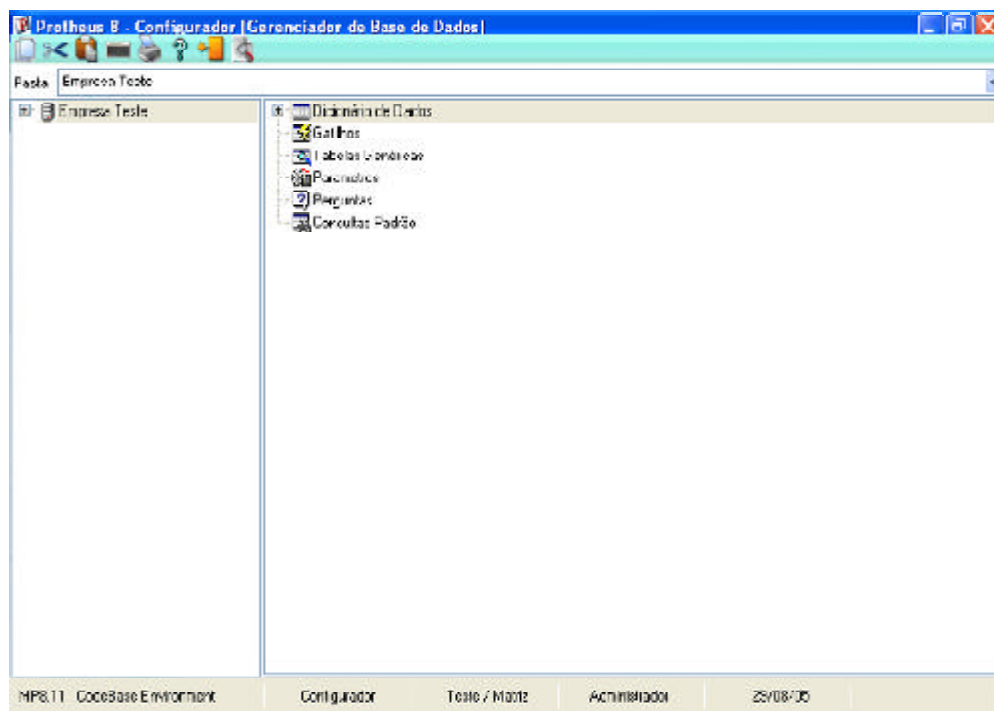


Figura 20.21 Conjunto de pacotes com formatos e informações do dicionário de dados.

Note que a interface disponível nesta janela possibilita uma série de ações por meio dos botões constantes na parte superior da tela, conforme a ilustração da Figura 20.21:

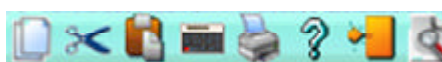




Figura 20.22 Botões de acesso à funções básicas.

No dicionário de dados, todas as informações e parâmetros podem ser realizadas para cada empresa cadastrada no sistema.

2. Selecione a Empresa Teste efetuando um duplo clique na pasta  Empresa Teste e em seguida efetue um clique na pasta  Dicionário de Dados, conforme ilustra a Figura 20.23:

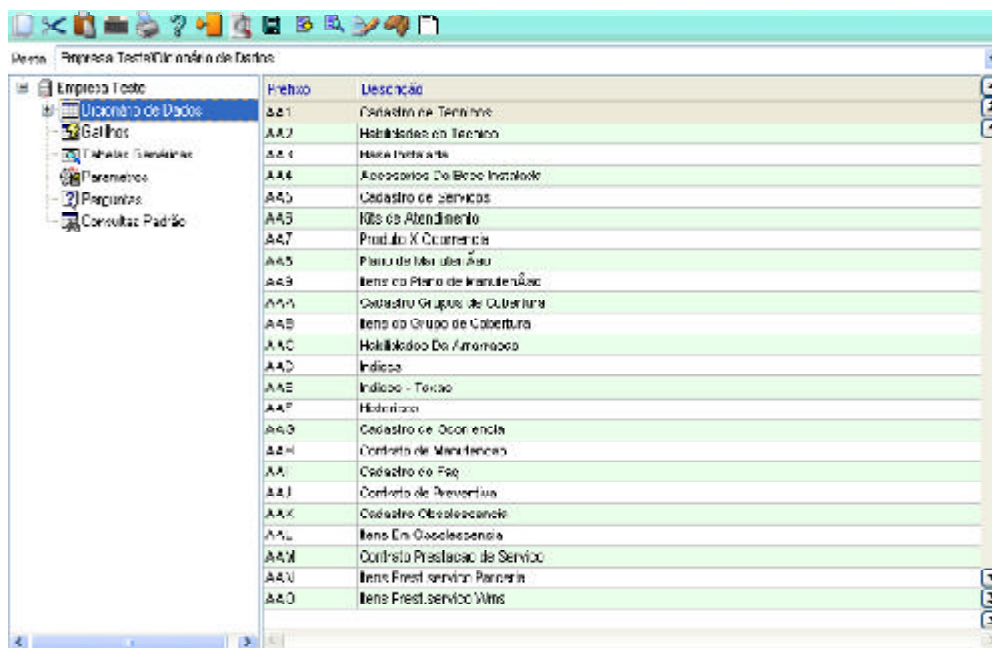


Figura 20.23 Tabelas cadastradas no Dicionário de Dados.

Note que são exibidos os arquivos cadastrados e que um conjunto de funções adicionais foram agregadas na parte superior da janela possibilitando ações de cadastramento e manutenção nos arquivos constantes no Dicionário de Dados.



Figura 20.24 Conjunto estendido de Botões de acesso às funções de Cadastramento e Manutenção dos Arquivos.



3. Pressione o Botão  para adicionar uma nova tabela no Dicionário de Dados e será apresentada uma janela como mostra a Figura 20.25:



Figura 20.25 Janela para cadastrar uma nova Tabela de Dados.

4. Preencha os campos conforme a seguir:




Prefixo	SZ1
Path	\DATA\
Descrição	ARQUIVO DE CONTAS
Desc. Esp.	ARCHIVO DE CUENTAS
Desc. Inglês	ACCOUNT FILE
Modo Acesso	1 Compartilhado

5. Pressione o botão  confirmar a inclusão da tabela SZ1 no dicionário de dados;

Dicas

- O domínio SZ1 até SZZ (considerando todos os número e todas as letras no último byte) é reservado para dados exclusivos do usuário pois este intervalo não será utilizado pelo sistema;
- O nome do arquivo é preenchido automaticamente adicionando 990. Este dado refere-se à empresa 99 (Teste Matriz) a qual está sendo adicionado a tabela;
- O Path refere-se à pasta que conterá efetivamente os dados dos arquivos.

Esta pasta será criada dentro da pasta indicada na configuração do sistema como RootPath;

- *O modo de acesso compartilhado indica que o sistema possibilitará o uso simultâneo do arquivo por duas ou mais filiais. Se for compartilhado o campo Filial fica em branco. Se for exclusivo grava-se o código da Filial ativa e somente ela tem acesso ao registro.*
 - *Após a confirmação, a tabela SZ1 passa a fazer parte do cadastro do Dicionário de Dados. Faltam, porém, os respectivos campos que deverão ser informados a seguir.*
5. Selecione o prefixo SZ1 e em seguida pressione o botão  para que seja exibida a tela de edição da respectiva tabela;
 6. Efetue um duplo clique na pasta  Arquivo de Contas e em seguida efetue um clique na pasta  Campos para efetuar a inclusão dos campos com todas as suas características.

Note que já foi incluído o campo Z1_FILIAL com as características *default*. O campo Ordem indica a seqüência em que os campos são apresentados. Note que, conforme Figura 20.26, fisicamente os campos permanecem gravados na sua seqüência original. Ocorre que o SX3 tem na sua chave de classificação o conteúdo do campo Ordem, campo este que pode ser alterado pelo usuário.

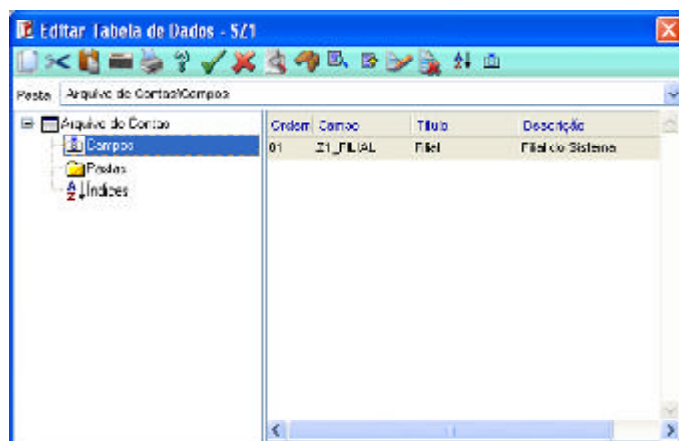


Figura 20.26 Edição da Tabela de Dados.

As propriedades definidas no Dicionário de Dados (SX3) são as seguintes:

Nome do Campo: todos os campos tem como prefixo o próprio nome do arquivo. A identificação dos arquivos é composta de 3 caracteres, conforme os exemplos ilustrados na Figura 20.27.

Identificação	Descrição
SA1	Cadastro de Clientes
SA3	Cadastro de Vendedores
SB1	Cadastro de Produtos
SB2	Saldos dos produtos por almoxarifados
SC1	Solicitações de Compras
SC7	Pedidos de Compras
SD1	Itens das Notas de Entrada
SE1	Títulos a Receber
SF1	Cabecalho das Notas de Entrada
SI1	Plano de Contas
SI2	Lancamentos Contábeis

Figura 20.27 Exemplos de Arquivos.

O primeiro dígito inicialmente era sempre S, de Siga. Posteriormente com o surgimento de novos arquivos (hoje são mais de 1.500) utilizou-se o primeiro caracter para identificar as Verticais.

Assim temos os seguintes exemplos de nomes de campos:

A1_Endereço

B1_Tipo

Nestes casos entende-se que a primeira letra do arquivo é S e a identificação completa dos campos é SA1->A1ENDEREÇO, SB1->B1_TIPO.

Tipo do Campo: Indica se é numérico, caracter, data ou lógico. É claro que a mudança do tipo de campo deve ser feita com muito cuidado, pois se tivermos um campo numérico usado em cálculos e o mesmo for alterado para caracter certamente teremos um erro.

Tamanho do campo: Também aqui é necessário um certo cuidado ao alterá-lo, pois poderemos ter truncamentos em

relatórios e consultas onde não há espaço para conteúdos maiores que o original.

Formato de edição: Define como o campo aparece nas telas e nos relatórios.


Contexto: Pode ser Real ou Virtual. O contexto Virtual cria o campo somente na memória e não no arquivo físico. Isto é necessário porque os programas de cadastramento e consulta genérica apresentam somente um arquivo de cada vez. Assim, se quisermos apresentar um campo de um outro arquivo, ou mesmo o resultado de um cálculo, sem que tal informação ocupe espaço físico no HD, utilizamos o contexto Virtual.

Campos virtuais normalmente são alimentados por Gatilhos.

Exemplos:

O nome do produto na tela de entrada de Nota Fiscal. Este campo pertence ao cadastro de produtos e os campos que estão sendo apresentados pertencem ao arquivo de Itens de Entrada, onde consta fisicamente somente o código do produto.

Propriedade: Indica se um campo pode ou não ser alterado pelo usuário. Exemplo: saldos normalmente não podem, pois quem cuida desta tarefa são os programas.

7. Pressione o botão  e será apresentada uma janela para a inclusão de um novo campo:

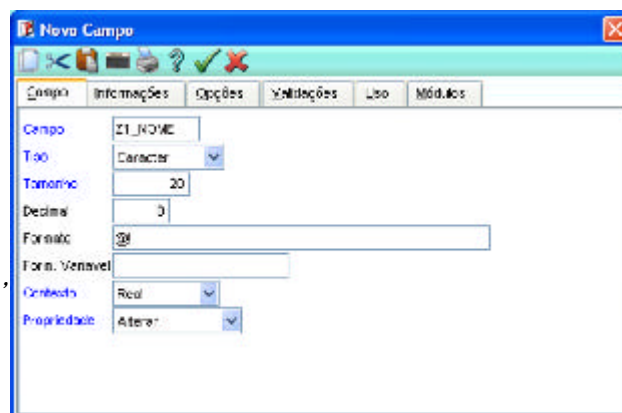


Figura 20.28 Inclusão de Campo na Tabela.

8. Preencha os campos conforme a seguir:

Campo	Z1_NOME
Tipo	1 Caracter
Tamanho	20
Formato	@!
Contexto	Real
Propriedade	Alterar

Dicas

- O campo *Decimal* será solicitado somente para os campos de tipo numérico;
- O formato *!* indica que o caracter será sempre maiúsculo, independente da ação do usuário. O formato *@!* indica que esta característica se estende por todo o campo;
- O contexto *real* indica que o campo existirá efetivamente no Banco de Dados e o contexto *virtual* significa que o campo existirá apenas no dicionário de dados e não fisicamente. O campo *Saldo* no arquivo *SZ2* é um exemplo de um campo virtual pois considerando que os programas mostram 1 arquivo por vez, ao criar-se naquele arquivo um campo virtual *saldo* da conta ele só precisará estar fisicamente no *SZ1*.
- A propriedade *alterar* indica que o campo pode ser alterado;

Nesta janela os dados estão classificados em seis pastas com objetivos de preenchimento bem específicos:

Campo Descreve os dados básicos do campo em si;


Informações Contém as informações a respeito dos títulos;

Opções Contém dados que facilitam o preenchimento

Validações Representam as regras de validação do campo;

Uso Descreve a Forma de Utilização do Campo;

Módulos Relaciona todos os módulos em que o campo será utilizado.

9. Selecione a pasta informações efetuando um clique na respectiva opção disponível  para introduzir os títulos e descrições relacionados ao campo que está sendo editado, conforme ilustra a Figura 20.29:

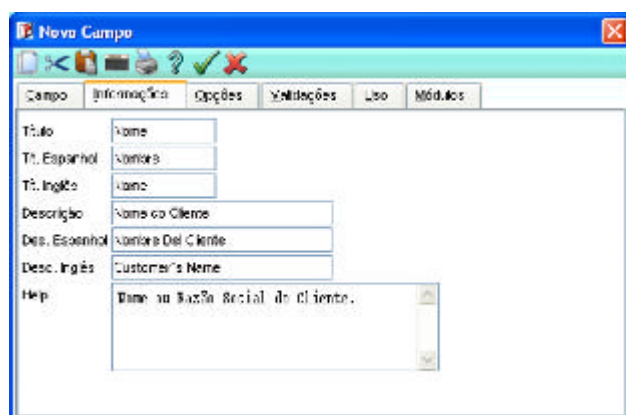


Figura 20.29 Informações do Campo.

Título: É a legenda que aparece nas telas/relatórios. Há inclusive 3 campos para esta finalidade: em português, espanhol e inglês. Esta propriedade pode ser alterada a vontade pois não interfere em nenhum processamento.

Descrição e Help: São propriedades que objetivam documentar o campo.

10. Informe os Títulos e as Descrições do campo conforme a seguir:

Título	Nome
Tít. Espanhol	Nombre
Tít. Inglês	Name
Descrição	Nome do Cliente
Des. Espanhol	Nombre Del Cliente
Desc. Inglês	Customer´s Name
Help	Nome ou Razão Social do Cliente.

11. Selecione a pasta **Validações** e preencha os campos da tela conforme a figura 20.30:

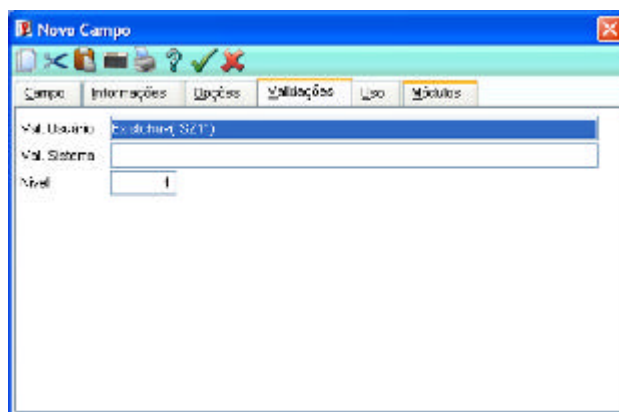


Figura 20.30 Validações do Campo.

Validações: Nesta propriedade escreve-se uma função de validação do campo que está sendo digitado. Existe um conjunto de funções disponíveis no AdvPL apropriadas para este caso.

Exemplos de funções usadas na Validação:


- **Pertence("SP/RJ/MG/etc"):** Esta função poderia ser aplicada ao campo onde será digitada a Unidade da Federação, campo este definido com 2 caracteres. Somente será aceita uma *string* designada no parâmetro. A barra é usada para evitar que, por exemplo, a sigla JM seja aceita;
- **CNPJ(cCNPJ):** testa o dígito de controle do CNPJ;

- **ExistChav(cAlias,cChave,nOrdem,cHelp):** verifica se o campo digitado já existe em cAlias, com base no índice nOrdem, emitindo, caso já exista, a mensagem cHelp. É para evitar chaves repetidas;
- **ExistCpo(cAlias,cChave,nOrdem):** verifica se o campo digitado existe em cAlias. Necessário quando o banco de dados não faz o teste de Integridade Referencial. Em ambos os casos cChave é opcional e se não for informado, o conteúdo será obtido do GET ativo;
- **Vazio0):** verifica se o campo está vazio. Tem o mesmo efeito que indicar que o campo é obrigatório, mas somente funciona se o usuário der um Enter no campo;
- **Positivo0):** verifica se o valor é positivo;
- **Texto0):** não permite a digitação seguida de mais de um espaço em branco, em campo tipo Character.
- **Nível:** Esta propriedade tem a ver com segurança. Cada usuário, através de sua senha, tem um nível. Da mesma forma cada campo também tem o seu nível. O usuário somente consegue enxergar os campos cujos níveis forem menores ou iguais ao seu.

12. Preencha os campos conforme a seguir:

Val. Usuário	Existchave("SZ1")
Nível	1

Dicas

- *Todas as validações informadas serão executadas quando do preenchimento do próprio campo. Uma validação pode ser uma expressão lógica ou uma função de usuário que retorna um valor lógico Verdadeiro ou Falso. O sistema só permitirá o avanço para o próximo campo quando o respectivo preenchimento resultar Verdadeiro seja na expressão ou no retorno da função;*
13. Selecione a pasta  para informar que o campo é obrigatório, como mostra a Figura 20.31:

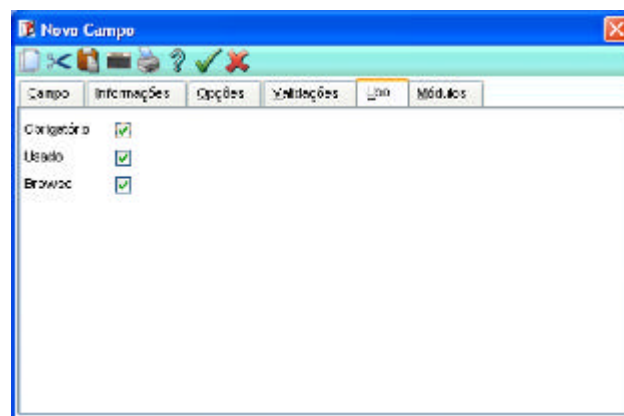



Figura 20.31 Características de Utilização do Campo.

Obrigatório: Esta propriedade indica que a digitação deste campo é obrigatória, ou seja, ele não pode ficar vazio. Aparece, por este motivo, em azul na tela.

A função `Obrigatório(aGets,aTela,tObg)` retorna .F. se algum campo obrigatório estiver vazio.

Usado: Esta propriedade indica se o campo é usado ou deve ser inibido. Note que o sistema não deleta do arquivo os campos que não estão em uso. Apenas faz com que não sejam mostrados nas telas de cadastramento e consultas. Caso se queira ganhar espaço no HD pode-se reduzir o tamanho dos campos não usados. Esta propriedade permite ter telas mais enxutas, de acordo com as necessidades do usuário. Pode-se ainda indicar em quais módulos o campo deve ser usado ou estar inibido.

Browse: Semelhante à propriedade anterior, mostra ou não mostra o campo nos browsers das telas de cadastramento.

14. Efetue a confirmação pressionando o botão  ;

Note que o campo já foi incluído com as suas respectivas características digitadas.

15. Inclua os demais campos da tabela de contas de acordo com os dados constantes na Figura 20.32:

Campo	Tipo	Tamanho	Decimal	Formato	Contexto	Propriedade	Título	Uso
Z1_EMAIL	1 Caracter	40	0		Real	Alterar	e-mail	Obrigatório
Z1_EMAIL_1	1 Caracter	40	0		Real	Alterar	e-mail	Obrigatório
Z1_SALDO	2 Numérico	12	2	@E 999.999.999.99	Real	Visualizar	Saldo	



Figura 20.32 Demais Campos da Tabela Exemplo SZ1.

Dicas

- *O campo e-mail não tem uma máscara de formato o que possibilitará a digitação de qualquer caracter. Coloque o domínio técnico.com.br pois usaremos este para execução dos nossos testes;*
- *O formato @E 999,999,999.99 no campo valor determina o ponto decimal e os separadores das casas de milhar. Este formato refere-se ao tamanho do campo e o número de casas decimais: 11 dígitos (contando os dígitos das casas decimais) e 1 ponto decimal, num total de 12. O E indica Europeu, ou seja, o milhar é separado por pontos e as decimais por vírgula;*
- *A propriedade do campo saldo é Visualizar para que seja ilustrado um exemplo de um campo que não pode sofrer alteração por ação direta do usuário, apenas por meio de programa específico através de uma Transação.*

Como Incluir os Índices no Dicionário de Dados

No ambiente Protheus, um arquivo pode ter vários índices. É por meio dos índices que podem ser acessados os registros dos arquivos tanto através de chaves primárias como de chaves estrangeiras. Os índices servem também para determinar a ordem de apresentação dos registros de um arquivo em consultas e relatórios. Os índices são criados no Configurador, como segue:

1. Selecione a pasta  e em seguida pressione o botão  para informar os campos que estarão compondo as chaves de acesso aos registros do arquivo, conforme ilustra a Figura 20.33:

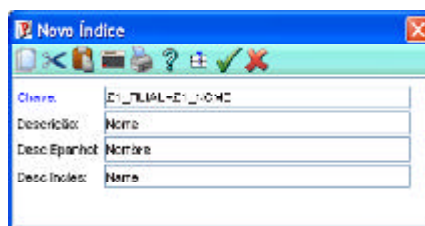



Figura 20.33 Cadastramento de Índices na Tabela.

2. Informe a chave Z1_FILIAL + Z1_NOME;
3. Digite a descrição em português, espanhol e inglês: Nome, Nombre e Name respectivamente.

Dicas

- Para selecionar os campos já cadastrados na tabela, pode ser utilizado o botão . Esta facilidade preenche automaticamente os campos de descrição;
- Caso o índice envolva mais de um campo da tabela poderá ser utilizada a concatenação de campos para a composição da chave, conforme o exemplo a seguir:

Z1_FILIAL + Z1_NOME

- O campo relativo a filial sempre faz parte dos índices para que os registros nas tabelas estejam agrupados por filiais;
- Um arquivo poderá ter vários índices cadastrados no Dicionário de Dados. Em determinado momento, porém, apenas um deles é que oferecerá o acesso ao registro. Esta ordem pode ser alterada em tempo de execução pelos programas da aplicação através do comando `DBSetOrderTo`.

4. Confirme pressionando o botão 

Como Efetuar a Amarração entre os dois Arquivos

Utilize os mesmos procedimentos para a inclusão da tabela SZ2 (Arquivo de Transações) considerando os campos constantes na Figura 20.34. Note que esta tabela também tem o campo Nome do Cliente pois este campo será, no

exemplo, a chave de amarração entre a tabela SZ1 e a tabela SZ2.

Campo	Tipo	Tamanho	Decimal	Formato	Contexto	Propriedade	Título	Uso
Z2_NOME	Caracter	20	0	@l	Real	Alterar	Nome	Obrigatório
Z2_NUMERO	Caracter	4	0	9999	Real	Alterar	Nro. Trans.	Obrigatório
Z2_ITEM	Caracter	2	0		Real	Alterar	Item	Obrigatório
Z2_DATA	Data	8	0		Real	Alterar	Data	Obrigatório
Z2_TIPO	Caracter	1	0	@l	Real	Alterar	Indicador	Obrigatório
Z2_HIST	Caracter	20	0		Real	Alterar	Histórico	Obrigatório
Z2_VALOR	Númerico	12	2	@F 999.999.999.99	Real	Alterar	Valor	Obrigatório
Z2_SLDATU	Númerico	12	2	@F 999.999.999.99	Virtual	Visualizar	Slid. Atual	
Z2_APPROV	Caracter	3			Real	Alterar	Aprovado	

Figura 20.34 Campos da Tabela SZ2.

A chave de amarração é o Nome do Cliente que, para a tabela SZ2 será considerada como chave estrangeira. O Número/Item da transação é a chave primária. Portanto é recomendável que os índices sejam cadastrados na seguinte ordem:

Z2_FILIAL + Z2_NUMERO + Z2_ITEM

Z2_FILIAL + Z2_NOME + Z2_NUMERO + Z2_ITEM

Como Incluir Opções e Validações para os Preenchimentos dos Campos do Arquivo Z2

O exemplo de cadastramento do arquivo SZ2 também contempla opções e validações para alguns campos. Efetue os procedimentos a seguir para incluir as respectivas opções e validações:

1. Efetue um clique na pasta **Validações** para informar, inicialmente as validações do campo Z2_NOME, conforme a Figura 20.35:

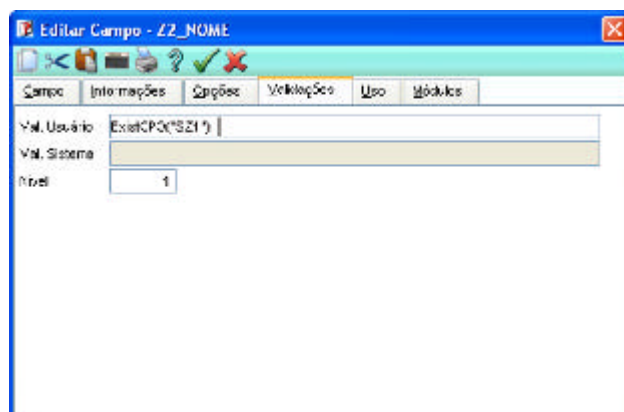


Figura 30.35 Validações de Campo.

2. Preencha os campos conforme a seguir:

Val. Usuário	ExistCPO("SZ1")
Nível	1

Dicas

- Neste caso, a função *ExistCPO("SZ1")* valida o preenchimento deste campo com a existência da respectiva chave no arquivo *SZ1*, pois ele é chave estrangeira no arquivo *SZ2*. Veremos que quando se trabalha com *SQL* esta funcionalidade já é executada pelo próprio banco.
3. No preenchimento do campo *Z2_TIPO*, selecione a pasta **Opções** para descrever as opções disponíveis no preenchimento do campo, conforme a Figura 20.36:

- Lista de opções:** São as opções apresentadas em campos preenchidos através de um Combo-Box.
- Inicializador padrão:** Contém uma constante ou expressão que define o conteúdo *default* do campo apresentado, quando da inclusão de um novo registro.

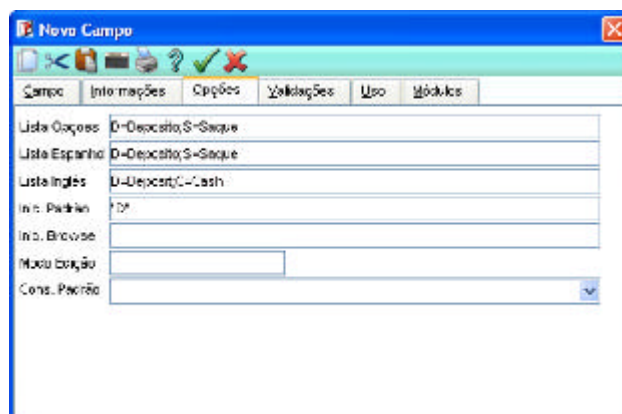


Figura 20.36 Opções de Campo.

2. Preencha os campos conforme a seguir:

Lista Opções	D=Deposito;S=Saque
Lista Espanhol	D=Deposito;S=Saque
Lista Inglês	D=Deposit;S=Cash
Inici. Padrão	"D"

Dicas

- O campo Z2_TIPO pode conter apenas Depósito ou Saque. A lista fica cadastrada no Dicionário de Dados e será apresentada para facilitar o seu preenchimento por meio de um clique na respectiva opção ou digitando-se a letra indicada na lista: D para Depósito ou S para Saque;

Como Criar uma Consulta Padrão

Para ser utilizada, uma consulta padrão tem que estar devidamente cadastrada com todas as suas características. Para tal, efetue os procedimentos a seguir para criar uma consulta padrão:



1. Selecione a opção Consultas Padrão efetuando um clique na pasta  Consultas Padrão e em seguida no botão  para incluir uma nova consulta padrão por meio da tela ilustrada na Figura 20.37.

Figura 20.37 Criação de Consulta Padrão.

2. Digite as Informações da Consulta Padrão, conforme a seguir:

Consulta	SZ1
Descrição	Cliente
Descrição Espanhol	Name
Descrição Ingles	Nome do Cliente

3. Pressione o botão  e escolha a tabela a ser utilizada.

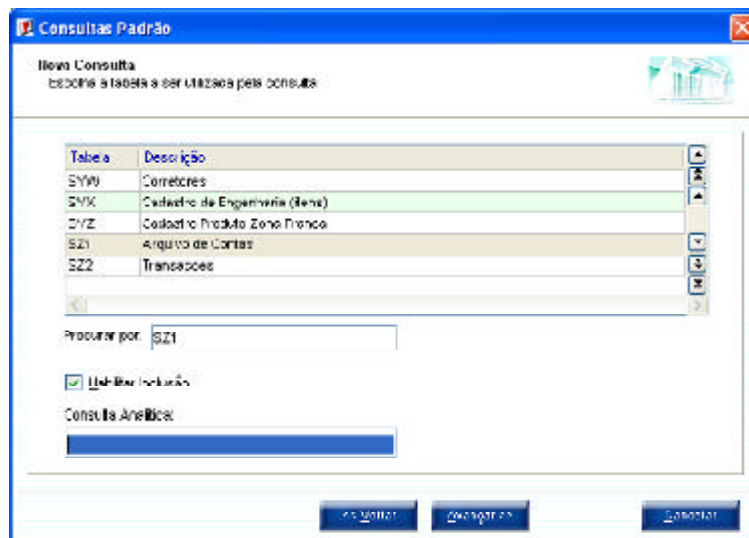



Figura 20.38 Escolha da Tabela a Ser Utilizada pela Consulta.

4. Selecione a tabela SZ1;
5. Selecione a opção Habilitar Inclusão a qual permite novos registros no momento da Consulta Padrão;
6. Pressione mais uma vez o botão  e escolha os índices e as colunas da Consulta Padrão;

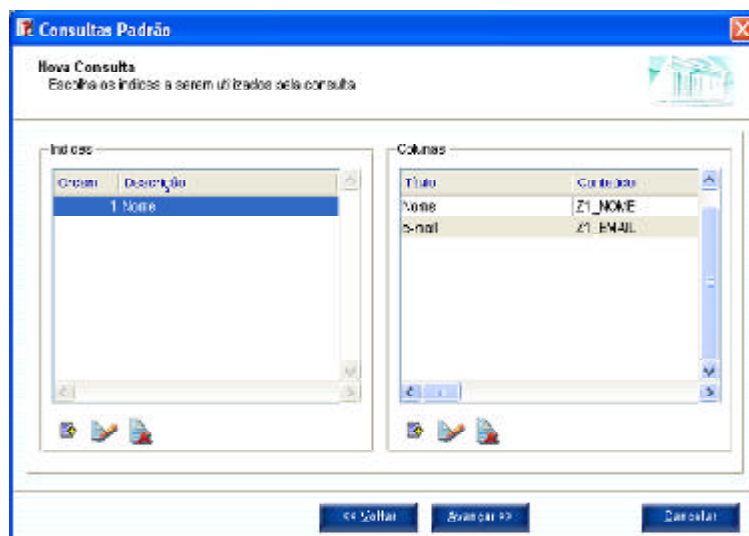




Figura 20.39 Escolha dos Índices e Colunas da Consulta Padrão.

7. Utilize os botões para poder informar as colunas e os índices envolvidos na Consulta Padrão;
8. Pressione mais uma vez o botão , adicione o campo SZ1->Z1_NOME como retorno e conclua a criação da Consulta Padrão pressionando o botão .

Dicas

- O retorno refere-se ao dado a ser preenchido no campo que utiliza a Consulta Padrão;
- Eventualmente os registros apresentados podem passar por um filtro definidos na própria Consulta Padrão. No exemplo este campo pode ficar sem preenchimento.

Como Utilizar uma Consulta Padrão

A função de validação **ExistChav("SZ1")** incluída no Dicionário de Dados verifica se o mesmo registro já existe na Base de Dados. Já uma Consulta Padrão verifica se uma determinada chave digitada em um arquivo existe em outro arquivo. No exemplo do Conta Corrente, na digitação de uma

determinada transação para um cliente deve-se verificar se o mesmo existe. Neste caso, no arquivo SZ2 (Transações) deve ser utilizada a Consulta Padrão, conforme os procedimentos a seguir:

1. Selecione a pasta Dicionário de Dados ;
2. Selecione o arquivo SZ@ (Transações) e pressione o botão ;
3. Efetue um duplo clique no mouse na pasta **Transações**;
4. Selecione a pasta Campos
5. Selecione a respectiva linha do campo Z2_NOME e pressione mais uma vez o botão para editar o registro;
6. Selecione a pasta e note que a Consulta Padrão já está disponível no respectivo campo, como mostra a Figura 20.40.

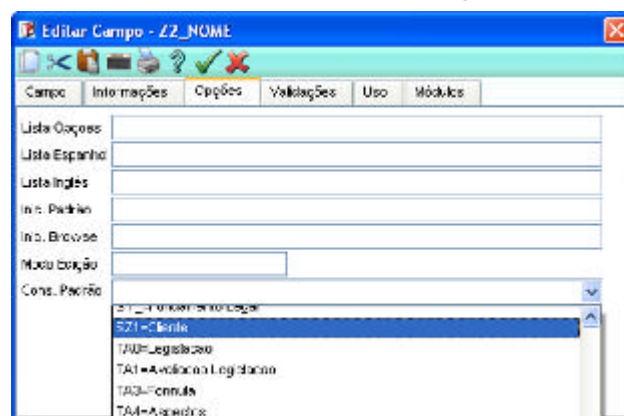


Figura 20.40 Escolha da Consulta Padrão.


7. Confirme a edição do registro e a edição da tabela pressionando o botão duas vezes;

Dicas

- A partir deste momento, toda vez que for solicitada a digitação do campo Cliente no arquivo SZ2 (Transações) é possível, ao comando do usuário, solicitar a apresentação de uma consulta mostrando todos os registros já cadastrados no arquivo SZ1.

Como Confirmar os Dados no Dicionário

Ao término da digitação das tabelas e dos índices, os dados deverão ser salvos no Dicionário de Dados.

1. Pressione o botão  e será apresentada uma janela que exibe as atualizações a serem feitas no dicionário de dados, conforme Figura 20.41:

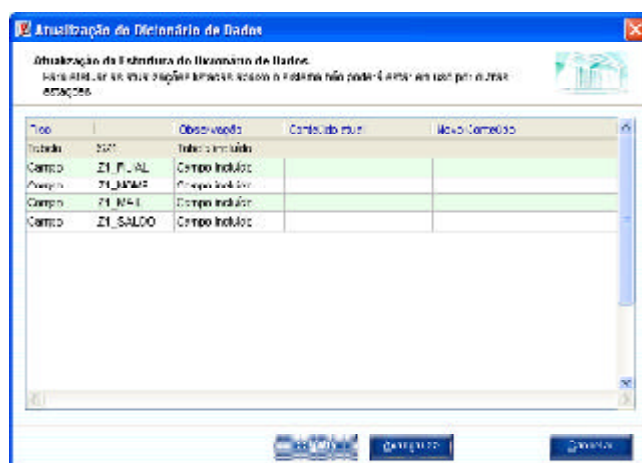


Figura 20.41 Atualização do Dicionário de Dados.


2. Pressione o botão  até que a atualização seja realizada com sucesso;




Figura 20.42 LOG das Atualizações do Dicionário de Dados.

3. Conclua a atualização pressionando o botão **Finalizar** na janela que mostra o LOG de atualizações no Dicionário de Dados;

Como Cadastrar um Gatilho no Dicionário de Dados

Muito embora um gatilho no Dicionário de Dados do Protheus possa executar uma função compilada no Repositório de Objetos a qual retorna o valor a ser preenchido no campo, os procedimentos a seguir mostram um exemplo de gatilho que executa uma regra preenchida no próprio Dicionário de Dados.

1. Selecione a pasta **Gatilhos** e em seguida pressione o botão  para incluir um novo gatilho no Dicionário de Dados conforme tela ilustrada na Figura 20.43.

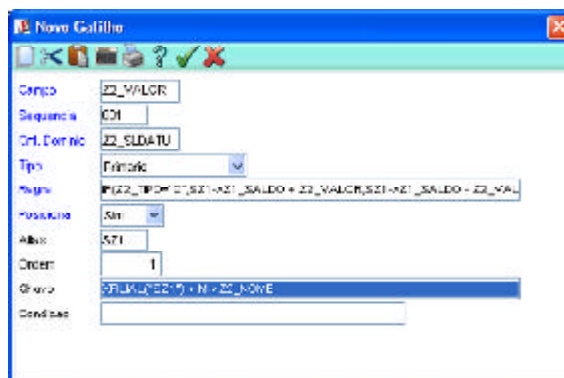




Figura 20.43 Cadastro de Gatilho no Dicionário de Dados.

2. Digite as Informações referentes ao Gatilho, conforme a seguir:

Campo	Z2_VALOR
Sequência	001
Cnt Domínio	Z2_SLDATU
Tipo	Primário
Regra	IF(Z2_TIPO="D",SZ1->Z1_SALDO + Z2_VALOR,SZ1->Z1_SALDO - Z2_VALOR)
Posiciona	Sim
Alias	SZ1
Ordem	1
Chave	XFILIAL("SZ1") + M->Z2_NOME

Dicas

- *Este Gatilho atualiza o campo virtual Z2_SLDATU considerando o saldo (Z1_SALDO) existente no arquivo SZ1 e também o tipo de transação. Depósito soma e Saque subtrai;*
 - *Podem haver vários Gatilhos para o mesmo campo e a ordem de execução é determinada pelo campo Seqüência;*
 - *Os tipos do Gatilho Primário, Estrangeiro e de Posicionamento definem se o Contra Domínio é um campo do mesmo arquivo, de outro arquivo ou se o Gatilho realiza apenas um posicionamento, respectivamente;*
 - *A regra pode ser uma expressão que resulta em um valor a ser preenchido no Contra Domínio;*
 - *O posicionamento igual a Sim indica que será executado um comando de busca do registro de acordo com a Chave indicada, caso contrário não.;*
 - *O Alias, Ordem e Chave descrevem o arquivo envolvido no gatilho, seu índice e uma regra para filtrar os registros.*
3. Confirme o novo gatilho pressionando o botão ;
 4. Pressione o botão  para sair do Dicionário de Dados e voltar ao menu principal do Configurador.

Como Cadastrar as Opções no Menu do Sistema

O cadastro dos arquivos no configurador refere-se apenas ao tipo de dado que será armazenado no Dicionário, descrevendo suas características e comportamentos. Para que os dados sejam efetivamente armazenados no Banco de Dados é necessário fazer a implementação de programas. Embora os programas ainda não tenham sido desenvolvidos é possível cadastrar as opções para a ativação de cada um dos programas em particular. A Figura 20.44 mostra a hierarquia de menu de opções do sistema Conta Corrente.

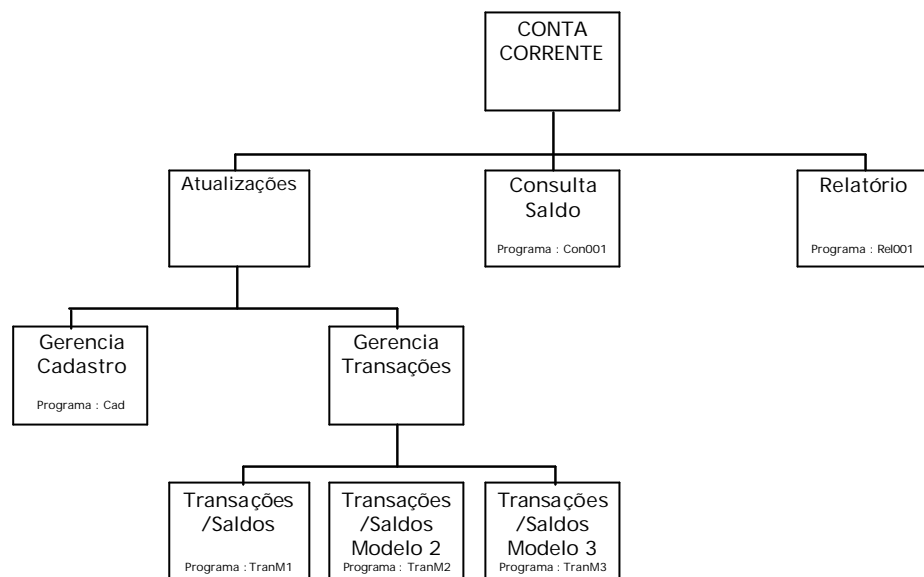


Figura 20.44 Estrutura Hirárquica de Módulos e Programas.

A inclusão do menu do menu é realizada no próprio configurador, conforme os procedimentos descritos a seguir:

1. Seleccione Ambiente, Cadastros, Menus. O sistema apresenta os menus já configurados conforme mostra a figura a Figura 20.45:

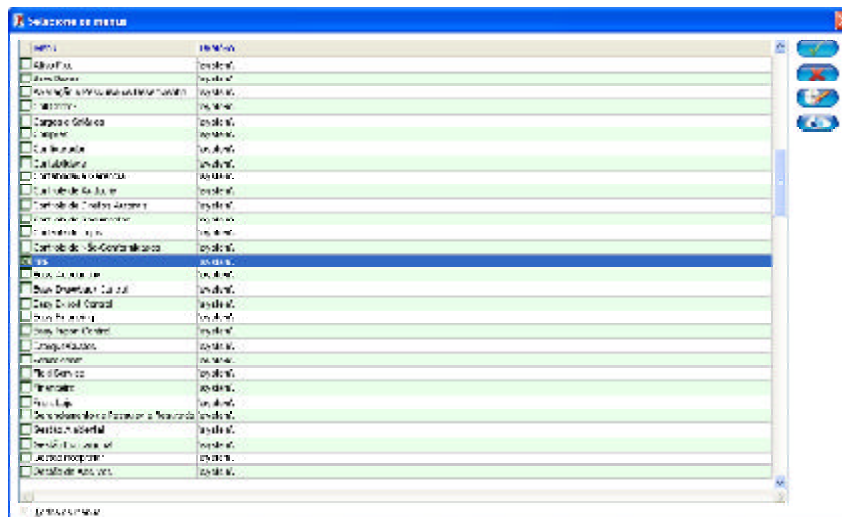



Figura 20.45 Lista de Menus Disponíveis.

2. Selecione o menu ERP e confirme pressionando o botão ;

A Figura 20.46 ilustra a tela de Configuração de Menus. Nesta tela, as opções constantes do lado esquerdo referem-se ao menu selecionado que já estão cadastradas. O novo menu, do lado direito da tela, terá as configurações do menu que será gravado e os botões ao centro possibilitam a manutenção das opções do novo menu. No sistema Protheus, os menus apresentam uma estrutura padrão com quatro grupos básicos: Atualizações, Consultas, Relatórios e Miscelânea.

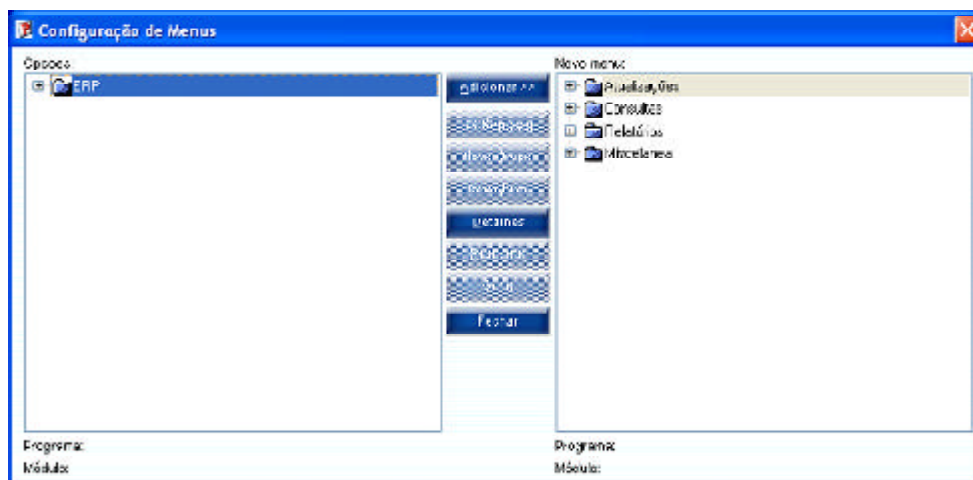


Figura 20.46 Configuração de Menus.




3. Selecione a opção  Atualizações no novo menu e note que os botões que permitem a manutenção do novo menu foram habilitados;
4. Pressione o botão  para incluir uma nova opção e o sistema apresenta uma tela solicitando as respectivas informações, conforme mostra a Figura 20.47:



Figura 20.47 Descrição de Novo Grupo de Opções no Menu.

5. Preencha os campos conforme a seguir:

Português	CONTA CORRENTE
Espanhol	CUENTA CORRENTE
Inglês	CURRENT ACCOUNT

6. Confirme as informações do novo grupo pressionando o botão  e note como o sistema incluiu este novo grupo:

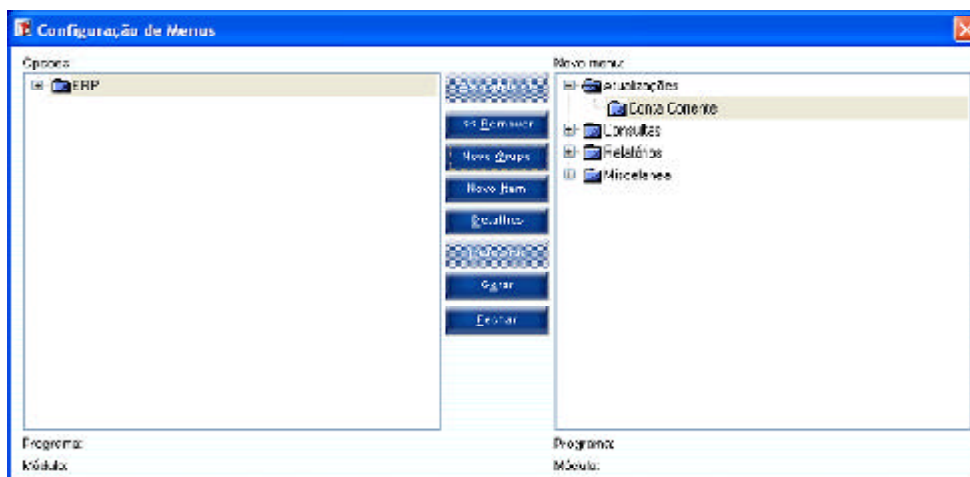




Figura 20.48 Menu com Nova Opção.

7. Selecione a opção  e em seguida pressione o botão  para incluir todas as opções já existentes no menu do ERP para o novo menu;

O menu será configurado de acordo com a Figura 20.44 portanto o grupo de Conta Corrente tem três opções: Atualizações, Consulta Saldo e Relatório, sendo que a opção Gerencia Transações (subordinada à opção Atualizações) é um grupo que contempla o Modelo 1, o Modelo 2 e o Modelo 3. Assim é necessário o cadastramento da opção (item ou grupo), como segue:

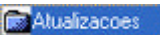

8. Selecione o grupo  e em seguida pressione o botão  para incluir o grupo Atualizações, como ilustrado na Figura 20.49:



Figura 20.49 Descrição do Grupo Atualizações do Conta Corrente.

9. Preencha os campos conforme a seguir:

Português	ATUALIZACOES
Espanhol	ACTUALIZACIONES
Inglês	UPDATES

8. Selecione o grupo  e em seguida pressione o botão  para incluir o item Gerencia Cadastro, como ilustrado na Figura 20.50:

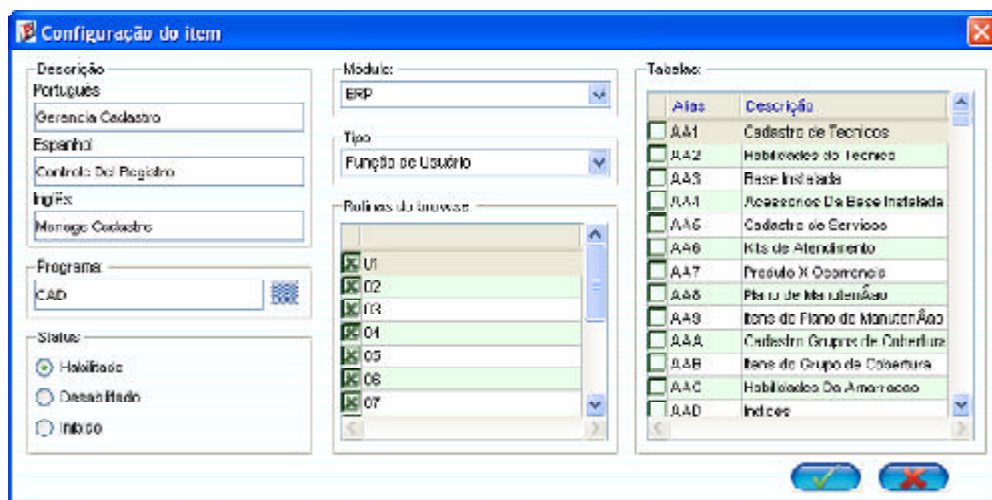


Figura 20.50 Configuração de Item de Menu.

7. Preencha os campos conforme a seguir:

Descrição Português	Gerencia Cadastro
Descrição Espanhol	Controla Del Registro
Descrição Inglês	Manage Cadastre
Status	Habilitado
Módulo	ERP
Tipo de Função	Função de Usuário
Programa	CAD




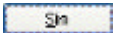
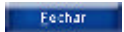


8. Confirme pressionando o botão ;
9. Efetue a inclusão das demais grupos e itens de acordo com a estrutura indicada na Figura 20.51;
10. Ao término da digitação, pressione o botão  e o sistema solicita o nome do arquivo no qual serão armazenadas as informações do respectivo menu, conforme a Figura 20.B1.



Figura 20.51 Geração do Menu de Opções.

11. Informe o nome do arquivo SIGAESP e pressione novamente o botão ;
12. Confirme a substituição do arquivo pressionando o botão ;
13. Pressione o botão ;
14. Pressione novamente o botão ;
15. Por fim cancele a seleção de menus pressionando o botão .

IDE (*Integrated Development Environment*)

O MP8IDE (*Integrated Development Environment*) é um programa que faz parte do Protheus e facilita o trabalho de edição, compilação e depuração de funções escritas em AdvPL.

Uma função para ser compilado precisa estar vinculado a um Projeto. Normalmente funções que fazem parte de um determinado sistema estão em um mesmo projeto. Portanto todas as funções do sistema exemplo de Conta Corrente farão parte do mesmo projeto. A vinculação das funções a um projeto é feita por meio dos arquivos do tipo PRW. Na verdade, um projeto pode ser constituído de um ou mais PRWs que por sua vez podem ter uma ou mais funções, conforme ilustrada a Figura 20.127.

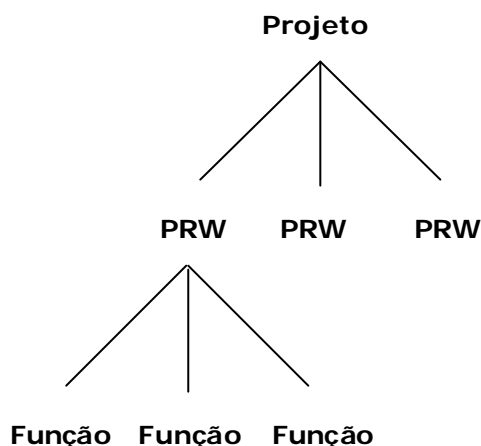


Figura 20.127 Organização das Funções dentro de um Projeto.

Uma vez compilado sem erro o objeto resultante é automaticamente cadastrado no RPO e a partir daí pode ser executado.

Para executá-lo valem as seguintes regras:

- se o programa não manipula arquivos pode-se chamá-lo diretamente do IDE (nome no lado direito da barra de ferramentas)
- se o programa manipula arquivos deve-se colocá-lo no menu do ERP e chamar o Remote. Caso se queira debugá-lo chama-se o SIGAESP a partir do IDE, marcando-se antes os pontos de parada.

Não se pode compilar um programa enquanto o Remote aberto mesmo que tenha terminado com erro.

Por outro lado, para descobrir-se as causas de erros, conta o IDE com um forte sistema de DEBUG. Como foi dito, para acionar o DEBUG é preciso executar o programa a partir do IDE. Para marcar um ponto de parada clica-se do lado direito do número da linha. Para desmarcá-lo clica-se novamente. Para as variáveis e arrays há uma opção para as locais, uma para as privadas, uma para as estáticas e uma para as públicas. Pode-se visualizar campos das tabelas. No watch determina-se quais variáveis devem ser mostradas. Na pilha de chamadas verifica-se a sequência de chamadas das funções. Na pasta de Comandos pode-se, enquanto o programa estiver parado, escrever qualquer comando e ao se dar Enter ele é executado. Pode-se pesquisar palavras e expressões no próprio fonte ou em qualquer fonte armazenado no HD.

Ao parar pode-se ou continuar o programa até o próximo ponto de parada, se houver, ou caminhar de linha em linha.

Por ser um ambiente integrado de desenvolvimento, o IDE proporciona todas estas facilidades por meio de interface única como ilustra a Figura 20.128.

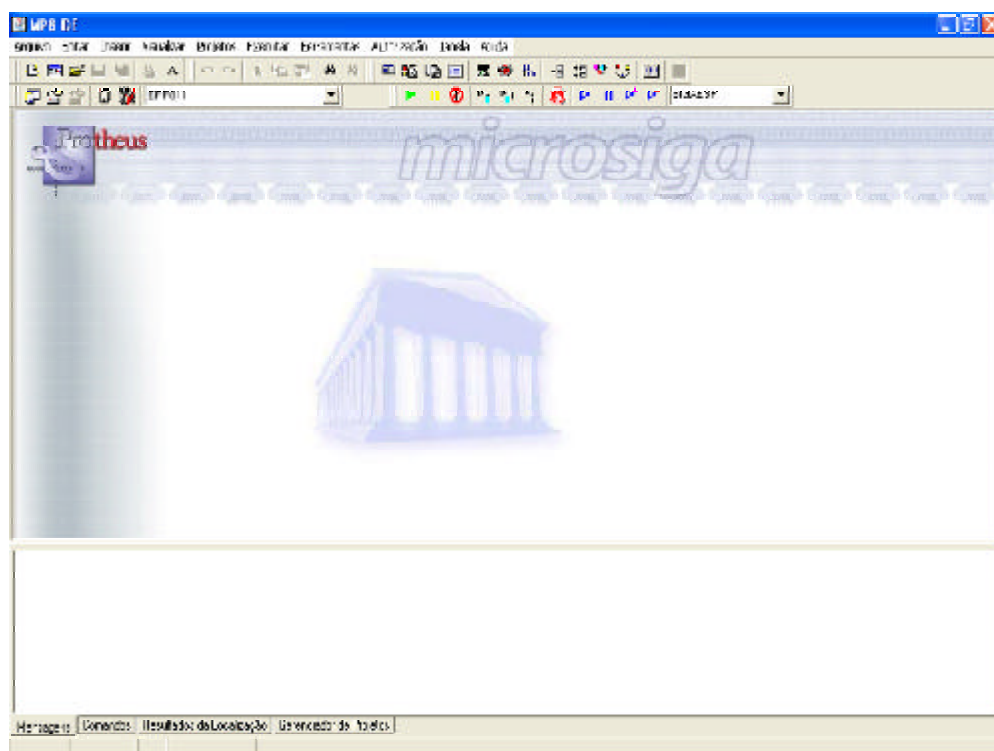


Figura 20.128 Interface Principal do IDE Protheus.

O IDE apresenta, no topo da tela, um conjunto de opções de menu e uma série de botões que facilitam a sua manipulação. Na tela central é apresentado o código das funções em AdvPL. Na parte inferior exibe algumas pastas que facilitam a execução de comandos, exibição de conteúdos de variáveis e mensagens bem como dados sobre o projeto.

A Linguagem AdvPL

Como é sabido uma linguagem de programação é um conjunto de comandos e funções que definem ao computador o passo a passo de uma tarefa.

Basicamente os comandos se dividem em três tipos:

Aritméticos

Somar, subtrair, multiplicar, dividir, etc;

Entrada e Saída

Leitura e gravação em uma mídia eletrônica (disco, memory key, etc), comandos de tela, impressão e teclado e também o comando de atribuição, que move os dados na memória de um campo para outro, através do :=

Comandos Lógicos

São na verdade, aqueles que dão “inteligência” ao computador pois, dependente de uma condição, indicam quais rotinas devem ser executadas. Liderados pelo IF, podem apresentar-se de outras formas: While, For e Do Case.

Da Figura 20.52 até a Figura 20.55 são exemplos que exibem a utilizações dos comandos While, If, For e DoCase, respectivamente.

```
////////////////////////////////////  
// COMANDO WHILE ... END //  
////////////////////////////////////  
  
User Function TstWhile()  
  
    Local i := 1  
  
    While i <= 10  
        MsgAlert(i)  
        i := i + 1  
    End  
  
Return Nil
```

Figura 20.52 Exemplo de Uso do Comando WHILE no AdvPL.

```

/////////////////////////////////////////////////////////////////
// COMANDO IF PARA IDENTIFICAR O MAIOR                                //
/////////////////////////////////////////////////////////////////

User Function TstIf()

    Local nX := 10

    If nX > 5
        MsgAlert("Maior")
    EndIf

Return Nil

/////////////////////////////////////////////////////////////////
// COMANDO IF PARA IDENTIFICAR O MAIOR E O MENOR                    //
/////////////////////////////////////////////////////////////////

User Function TstElse()

    Local nX := 10
    Local cMsg

    If nX < 5
        cMsg := "Maior"
    Else
        cMsg := "Menor"
    EndIf

    MsgAlert(cMsg)

Return Nil

/////////////////////////////////////////////////////////////////
// COMANDO IF COMO UMA ESTRUTURA CASE ...OTHERWISE                //
/////////////////////////////////////////////////////////////////

User Function TstElseIf()

    Local cRegiao := "NE"
    Local nICMS

    If cRegiao == "SE"
        nICMS := 18
    ElseIf cRegiao == "NE"
        nICMS := 7
    Else
        nICMS := 12
    EndIf

    MsgAlert(nICMS)

Return Nil

```

Figura 20.53 Exemplos de Uso do Comando IF no AdvPL.

```

////////////////////////////////////
// COMANDO FOR ... NEXT                                     //
////////////////////////////////////

// Exemplo de 1 a 10.

User Function TstFor1()

    Local i

    For i := 1 To 10
        MsgAlert(i)
    Next

Return Nil

// Exemplo de 2 em 2 com condição de parada.

User Function TstFor2()

    Local i
    Local nIni, nFim
    nIni := 100
    nFim := 120

    For i := nIni To nFim Step 2

        MsgAlert(i)
        If i > 110
            Exit      // Break tambem encerra.
        EndIf

    Next

Return Nil

// Exemplo decrescente (10 para 1).

User Function TstFor3()

    Local i
    Local nIni, nFim
    nIni := 1
    nFim := 10

    For i := nFim To nIni Step -1
        MsgAlert(i)
    Next

Return Nil

```

Figura 20.54 Exemplo de Uso do Comando FOR no AdvPL.

```

// DO Case...EndCase --> avalia a partir do primeiro Case.           //
//                               Ao encontrar o primeiro que satisfaça, //
//                               a condição, executa e vai para o EndCase. //
////////////////////////////////////

User Function TstCase()

    Local nOpc := 2

    Do Case
        Case nOpc == 1
            MsgAlert("Opção 1 selecionada")
        Case nOpc == 2
            MsgAlert("Opção 2 selecionada")
        Case nOpc == 3
            MsgAlert("Opção 3 selecionada")
        Otherwise
            // Otherwise é opcional.
            MsgAlert("Nenhuma opção selecionada")
        EndCase

    Return Nil

```

Figura 20.55 Exemplo de Uso do Comando DO CASE no AdvPL.

Os operadores usados no AdvPL são os seguintes:

Matemáticos

- + Adição ou sinal positivo;
- Subtração ou sinal negativo;
- * Multiplicação;
- / Divisão;
- ** ou ^ Exponenciação;
- % Módulo resto da divisão;

Relacionais

- < Menor que;
- > Maior que;
- == Exatamente igual;

<= Menor igual;
>= Maior igual;
<> ou **#** ou **!=** Diferente;

Lógicos

.Not. ou **!** “Não” Lógico;
.and. “e” Lógico;
.or. “ou” Lógico;

Atribuição

:= Permite atribuir um valor a uma variável;
+= Adiciona antes de atribuir;
-= Subtrai antes de atribuir;
***=** Multiplica antes de atribuir;
/= Divide antes de atribuir;
^= ou ****=** Eleva antes de atribuir;
%= Calcula o Módulo antes de atribuir;

Operadores Incremento/Decremento

++X Soma um ao valor de X e então retorna o valor de X já atualizado;
X++ Retorna o valor de X e então soma um ao valor de X;
-X Subtrai um do valor de X e então retorna o valor de X já atualizado;
X- Retorna o valor de X e então subtrai um do valor de X;

Operadores Strings

x+y Concatenação dos strings x e y;
x-y Concatenação dos strings, mas os espaços em branco à direita do

primeiro operando (x) serão transferidos para o fim do string resultante da concatenação;

x\$y Retorna verdadeiro se o conteúdo da variável x estiver contido no conteúdo da variável y;

Operadores Especiais

&cVariavel Operador Macro

- ||** Indica uma lista de argumento em um bloco de código;
- 0** Função ou argumento em fórmula numérica ou macro operador;
- ||** Referência a um elemento de uma matriz;
- {}** Delimitador do bloco de código ou atribuição de valores literais para um *array* (vetor);
- >** Referência a um alias;
- @** Passagem de parâmetros por referência;
- ;** Possibilita a continuação de comandos na linha seguinte;
- :** Usado como operador “send” quando acessa nova classe de objeto, atribuição a objeto;

Variáveis

São campos na memória manipulados pelo programa. No AdvPl estes campos não são tipados, ou seja, mesmo que um campo seja numérico ele pode receber um texto, mas após esta atribuição não pode ele fazer parte de um cálculo. Esta característica facilita o trabalho de programação.

De qualquer forma ao se definir as variáveis podem elas assumir os seguintes tipos:

Numéricas	para cálculos, podendo ter até 14 inteiros e 8 decimais;
Caractere	para textos;
Lógicas	podem assumir a condição de Verdadeiro (.T.) ou Falso (.F.). São usadas como chaves de decisão.

Data

devido ao tratamento específico dado aos dias/meses/anos há um tipo próprio que permite somar e subtrair um certo número de dias à uma data ou mesmo fazer comparações entre elas.

A notação húngara sugere que iniciemos o nome do campo com uma letra minúscula indicando seu tipo. Ex: `cTexto`, `nValor`, `dVencimento`, `IAchou`.

Para atribuir um valor à variável usa-se :=

Exemplos:

nValor := 123,45;

cTexto := "casa" (as aspas indicam que se trata de um texto);

IAchou := .T. IAchou fica verdadeiro;

dVencimento := CtoD("25/12/05") CtoD converte o texto em data;

Arrays

O *array*, matriz ou vetor também é um tipo de variável. Nele pode-se armazenar uma sequência de dados, identificados pelo nome do *array* e o número do elemento. Cada elemento, por sua vez, pode ser um novo *array*. É como se tivéssemos um arquivo ou uma planilha Excel na memória, com um processamento extremamente rápido. Considerando o tamanho das memórias hoje em dia pode-se criar *arrays* bastante grandes.

O *array* é definido com tamanho fixo ou variável:

Exemplos:

aTabela := Array(30) aTabela tem trinta elementos;

aTabela := {} array de tamanho indefinido ou seja pode receber tantos elementos quantos couberem na memória;

aTabela := {1,2,4} array com três elementos e seus respectivos valores;

aTabela := {1,2,"casa"} os elementos podem ser de tipos diferentes;

aTabela := Array(3,2) o array terá três linhas, cada uma com duas colunas;

aTabela := Array(3,2,2) é um array multidimensional, cada uma das três linha tem duas colunas, que por sua vez tem mais dois elementos cada:

```
aTabela :=  {{{L1,C1,X1} , {L1,C1,X2}} ,;  
             {{L1,C2,X1} , {L1,C2,X2}},;  
             {{L2,C1,X1} , {L2,C1,X2}},;  
             {{L2,C2,X1} , {L2,C2,X2}},;  
             {{L3,C1,X1} , {L3,C1,X2}},;  
             {{L3,C1,X1} , {L3,C2,X2}}}
```

Para evocar um elemento usa-se colchetes:

aTabela[1] é o primeiro elemento de um array simples

aTabela[i] é um elemento identificado pela variável i

aTabela[2,3] é o terceiro elemento do segundo elemento do array

Várias funções permitem adicionar um elemento a um *array*. A mais usada é **AADD**, que inclui um novo elemento no final do *array*. As funções **AFILL**, **AINS**, **ACOPY** também incluem dados em um *array*. A Figura 20.B demonstra alguns exemplos básicos para a criação e preenchimento de *arrays* com o uso destas funções.

```

/////////////////////////////////////////////////////////////////
// EXEMPLOS DE FUNÇÕES PARA CRIAÇÃO DE ARRAYS //
/////////////////////////////////////////////////////////////////

#include "rwmake.ch"

User Function TstArray()

    Local i
    Local aX

    // Inicializa um array vazio --> para os casos em que o numero
    // de elementos é desconhecido.

    aX := {}

    For i := 1 To 5
        AAdd(aX, i*10) // Adiciona elementos ao final do array.
    Next

    // Exclui o terceiro elemento. Os elementos subsequentes sobem
    // uma posição e o ultimo elemento fica com valor NIL.

    ADel(aX, 3)

    // Insere um elemento no segundo elemento. Os elementos subsequentes
    // descem uma posição e o ultimo elemento é descartado.

    AIns(aX, 2)

    // Redimensiona o array. Se o novo numero de elementos for maior
    // que o atual, serão inseridos elementos com valor NIL no final.
    // Caso contrário, os elementos do final serão eliminados.

    ASize(aX, 10)

    // Reinicializa o array. Volta a ser vazio.

    aX := {}

    // Inicializa com 3 elementos vazios.

    aX := Array(3)

    For i := 1 To Len(aX)
        aX[i] := i*2 // Atribue valor aos elementos do array.
    Next

    Return Nil

```

Figura 20.56 Exemplos de Criação de Arrays e a respectiva Inclusão, Alteração e Exclusão de seus Elementos.

Para facilitar o processo de busca nos elementos de um *array*, a função **ASCAN** pode ser utilizada tanto para arrays simples ou mesmo arrays multi-dimensionais, conforme ilustra a Figura 20.57.

```

////////////////////////////////////
// EXEMPLO DE FUNÇÕES DE BUSCA EM ARRAYS                                //
////////////////////////////////////

#include "rwmake.ch"

// Procura de um elemento dentro do array.

User Function TstAScan1()

    Local nItem
    Local aMatriz := {"Joao", "Alberto", "Pedro", "Maria"}

    nItem := AScan(aMatriz, "Pedro")

    MsgAlert(nItem)

Return Nil

// Procura de um elemento dentro de um array multi-dimensional.

User Function TstAScan2()

    Local aMatriz := {{ "Joao",15}, {"ALBERTO",20}, {"Pedro",10}, {"Maria",30}}

    nItem := AScan(aMatriz, {|aX| aX[1] == Upper("Alberto")})

    MsgAlert(nItem)

Return Nil

```

Figura 20.57 Exemplo de Uso da Função **ASCAN**.

A classificação dos elementos de um array na Figura 1.58 é um exemplo que mostra bem como se trabalha com este tipo de dado.

Esta rotina compara sempre um elemento com o próximo. Caso o primeiro seja maior que o segundo, há uma troca entre os dois. A rotina faz tantas passadas quantas forem necessárias e só para quando houver uma passada sem nenhuma troca, ou seja, o *array* está em ordem crescente. Para copiar um array para outro é importante lembrar que uma atribuição simples (exemplo: `aArrayNovo := aArrayAnterior`) cria apenas um ponteiro para os mesmos elementos do *array* anterior.

```

aZ := {"Pedro","Vanderlei","Antonio","Inacio"}

While .t.
  lMudou := .f.
  I := 1
  While I <= len(aZ) - 1 // len (aZ) retorna a quantidade de elementos de aZ.
    If aZ[I] > aZ[I+1]
      nManobra := aZ[I] // Salva o elemento aZ[I].
      aZ[I] := aZ[I+1] // Troca de posição.
      aZ[I+1] := nManobra
      lMudou := .T.
    Endif
    I := I+1
  End
  If .not. lMudou
    Exit // sai do loop indo para o comando após o End.
  Else
    Loop // volta para o while superior.
  Endif
End // também volta ao while superior.
Return Nil

```

Figura 20.58 Exemplo de Classificação em um Array.

Para criar efetivamente um novo array copiando todos os elementos de um array já existente deve-se utilizar a função **ACLONE**, conforme o exemplo ilustrado na Figura 20.59.

```

/////////////////////////////////////////////////////////////////
// EXEMPLO DE CÓPIA DE UM ARRAY PARA OUTRO //
/////////////////////////////////////////////////////////////////

#include "rwmake.ch"

User Function TstClone()

  Local aMatriz := {"Joao", "Alberto", "Pedro", "Maria"}
  Local aCopia

  aCopia := aMatriz

  aCopia[1] := "AAAA"
  aCopia[2] := "BBBB"
  aCopia[3] := "CCCC"
  aCopia[4] := "DDDD"

  aMatriz := {"Joao", "Alberto", "Pedro", "Maria"}

  aCopia := ACLONE(aMatriz)

  aCopia[1] := "AAAA"
  aCopia[2] := "BBBB"
  aCopia[3] := "CCCC"
  aCopia[4] := "DDDD"

Return Nil

```

Figura 20.59 Exemplo de Uso da Função **ACLONE**.

Funções

A maior parte das rotinas que queremos escrever em programas são compostas de um conjunto de comandos. Rotinas estas que se repetem ao longo de todo o desenvolvimento. Uma Função nada mais é do que um conjunto de comandos. Para ser usada basta chamá-la pelo seu nome. Para tornar uma Função mais flexível ao chamá-la pode-se passar parâmetros. Estes parâmetros contém dados e informações que influem no processamento da função. Os parâmetros no AdvPl são posicionais, ou seja, na sua passagem não importa o nome da variável e sim a sua posição dentro da lista. Assim podemos chamar uma função escrevendo:

Calcula(parA,parB,parC)

E a função estar escrita:

User Function Calcula(x,y,z)

... Comandos da Função

...

Neste caso, x assume o valor de parA, y de parB e z de parC.

A Função também tem a faculdade de retornar uma variável, podendo inclusive ser um Array. Para tal encerra-se a Função:

Return(campo)

Assim A := Calcula(parA,parB,parC) atribui à A o conteúdo do retorno da função Calcula.

No Advpl existem milhares de Funções escritas pela equipe de Tecnologia, pelos analistas de suporte e pelos próprios usuários. Existe um ditado que diz que “vale mais um programador que conhece todas as funções disponíveis em uma linguagem do que aquele que reinventa a roda a cada novo programa”. Mas conhecer todas as funções vem com o tempo. No DEM (Documentação Eletrônica Microsiga) mais de 500 estão documentadas. Aqui e no curso não abordaremos mais do que 100. Cabe a cada um estudá-las quando delas necessitar.

No AdvPl até os programas chamados do menu são funções. Num repositório não podem haver funções com o mesmo nome. Para contornar este problema as funções escritas pelo usuário tem o prefixo U_ de User Function.

Escopo das Funções e Variáveis

Um programa, que sempre tem como sufixo de seu nome a sigla .prw, é na verdade um conjunto de funções e de dentro dele são chamadas outras, todas elas presentes no Repositório. Para organizar este ambiente e, de um lado, impedir que uma função destrua as variáveis de outra e, por outro, permitir que haja uma integração entre elas, cada variável tem o seu escopo.

Variáveis Locais São definidas dentro da função e somente são visualizadas e podem ser atualizadas por comandos internos à função.

Variáveis Privadas Podem ser visualizadas e alteradas pela função que a definiu e por todas as outras que ela chamar. Neste caso não há necessidade de passá-la como parâmetro.

Variáveis Públicas Podem ser visualizadas e atualizadas em qualquer função. São normalmente variáveis definidas na chamada do sistema para uso geral. Ex: dDataBase, é a data digitada pelo usuário na chamada do sistema. O mesmo com cFilial.

Variáveis Estáticas Podem ser visualizadas e atualizadas dentro do PRW. Seu uso é raro.

Macro-Substituição

Como já foi visto no Configurador a Macro-Substituição trata o conteúdo do campo como se ele fosse uma linha de código de um programa. Ocorre que este conteúdo pode ser lido de um arquivo externo (SX3, SX7, SI5, SM4, etc), arquivo este que pode ser alterado pelo usuário. Estas expressões são encontradas no dicionário de dados, nos campos de validação e inicialização, nos gatilhos, no arquivo de lançamento padronizado, na folha de pagamento, no arquivo de fórmulas, entre outros. Quando se usa a Macro-substituição em um programa deve ela ser sempre precedida pela leitura da expressão em um arquivo. O símbolo & é utilizado para indicar que se trata de uma macro substituição, conforme ilustra a Figura 20.60.

```

// o arquivo de fórmula é atualizado pelo usuário

DBSelectArea("SM4")      // seleciona o
                        // o arquivo de
                        // fórmula (SM4)
DbSeek(SC5->C5_REAJUSTE) // busca no arquivo selecionado
                        // (SM4) a respectiva fórmula
                        // de acordo com o código do
                        // reajuste informado no pedido
                        // pedido (SC5)
nPreco := &M4_Formula    // calcula o preço

```

Figura 20.60 Exemplo de Macro-Substituição.

Lista de Expressões

O AdvPL aceita que se escreva vários comandos em uma única lista de expressões, abrindo maiores possibilidades no uso de validações e gatilhos onde só temos o espaço de uma linha para escrevê-la. Mas quando se tem um código mais extenso, a solução é escrever-se uma função, que pode ter um tamanho ilimitado e precisa ser compilada.

Bloco de Código

O Bloco de Código permite que se execute uma expressão externa em tempo de execução exatamente da mesma forma que a Macro-substituição, mas com uma vantagem: permite que se passe parâmetros, como se fosse uma função. Para executar um Bloco de Código usa-se a função Eval, que retorna o resultado do processamento do Bloco.

Outra vantagem do uso do Bloco de Código está na existência das funções aEval e DBEval. A aEval executa o Bloco para todos os elementos do array sem necessidade de nenhum comando adicional. A DBEval processa todos os registros de um arquivo. O exemplo a seguir mostra como pode ser feito o reajuste pelo Dólar, multiplicando o preço original pela variação do Dólar:

```
(RecMoeda(dDatabase,'2')/RecMoeda(SC5->C5_EMISSAO,'2')) * SC6->C6_PRCVEN
```

Este exemplo já foi visto quando falamos de Macro-Substituição. A função RecMoeda (recupera moeda) busca o valor do dólar (moeda numero 2) na dDataBase e na Data de emissão (C5_Emissao). Imagine que não existisse a função RecMoeda. O programa teria que ele ir buscar os valores do dólar nestas datas e armazená-los, por exemplo, em nDolarEm e em nDolarHoje.

O Bloco de Código seria:

```
bBloco := {x,y| (y/x) * SC6->C6_PRCVEN}
```

A chamada seria:

Eval(bBloco,nDolarEm,nDolarHoje)

Com o Bloco de Código o programa pode atribuir qualquer nome aos campos `nDolarEm` e `nDolarHoje`.

A exibição do conteúdo de um *array* pode ser facilitado com o uso da função **AEval**, conforme ilustra a Figura 20.61.

```
////////////////////////////////////  
// EXEMPLO DE USO DA FUNÇÃO AEVAL                                     //  
////////////////////////////////////  
  
#include "rwmake.ch"  
  
User Function TstAEval()  
  
    Local aMatriz := {"Joao", "Alberto", "Pedro", "Maria"}  
    Local i  
  
    For i := 1 To Len(aMatriz)  
        MsgAlert(aMatriz[i])  
    Next  
  
    MsgAlert("O mesmo efeito, usando AEval()")  
  
    // A funcao AEval() percorre automaticamente todos os elementos da matriz,  
    // passando cada elemento como parametro para o bloco de codigo.  
  
    AEval(aMatriz, {|x| MsgAlert(x)})  
  
Return Nil
```

Figura 20.61 Exemplo de Uso da Função **AEval**.

A tarefa de ordenação de *arrays* também pode ser facilitada com a combinação das funções **ASort** e **AEval** tanto para *arrays* simples como para *arrays* multi-dimensionais, como ilustra a figura 20.62.

```

////////////////////////////////////
// EXEMPLO DE ORDENAÇÃO COM USO DA FUNÇÃO ASORT                                //
////////////////////////////////////

#include "rwmake.ch"

// Ordenação de arrays.

User Function TstASort1()

    Local aMatriz := {"Joao", "Alberto", "Pedro", "Maria"}

    ASort(aMatriz)

    AEval(aMatriz, {|x| MsgAlert(x)})

Return Nil

// Ordenação de arrays multi-dimensionais.

User Function TstASort2()

    Local aMatriz := {"Joao",15}, {"Alberto",20}, {"Pedro",10}, {"Maria",30}}

    ASort(aMatriz,,{|aX,aY| aX[2] < aY[2]})

Return Nil

```

Figura 20.62 Exemplo de Uso da Função ASort e AEval.

UDC *User Defined Commands*

Os UDCs tem como objetivo principal facilitar a Legibilidade e Manutenibilidade do Fonte. Eles são lidos apenas pelo compilador que gera o objeto de acordo com suas diretivas, identificadas através do símbolo #.

O #DEFINE pode ser usado para permitir que o programador use palavras diferentes daquelas definidas pelo AdvPl e também para definir palavras-chave para o UDC #IFDEF ou #IFNDEF.

As palavras-chave também podem ser definidas no ato da compilação preenchendo /d seguido da palavra-chave no campo Parâmetros. Este campo pode ser informado através da opção Arquivo/Configurações/Editar do IDE.

O #IFDEF permite por exemplo que ao se ter um programa para atender as condições do Brasil e do México, tenha no mesmo fonte tudo que for comum aos dois e dependendo do #Define BRASIL ou #Define México sejam compiladaa as demais partes referentes a cada um dos países.

A Figura 20.63 demonstra o resultado do código após a substituição do compilador de acordo com as UDCs definidas.

```

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// EXEMPLOS DE UDC (USER DEFINED COMMANDS)
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

// Teste de Comandos Definidos pelo Usuário.
// Sem o uso de constantes #define.

User Function TstUDC1()
  Local i
  Local aArray := {{ "Joao", 25, .T., "4567-9876", 2 },,
                    { "Maria", 30, .F., "9517-6541", 0 },,
                    { "Jose", 18, .T., "6348-7537", 3 }}
  For i := 1 To Len(aArray)
    MsgAlert(aArray[i, 1])
    MsgAlert(aArray[i, 4])
    MsgAlert(aArray[i, 2])
    MsgAlert(aArray[i, 3])
    MsgAlert(aArray[i, 5])
  Next
Return Nil

// A mesma rotina, mas com o uso das constantes #define.

#define __NOME      1
#define __IDADE     2
#define __ESTCIVIL  3
#define __FONE      4
#define __NRDEPEND  5

User Function TstUDC2()

  Local i
  Local aArray := {{ "Joao", 25, .T., "4567-9876", 2 },,
                    { "Maria", 30, .F., "9517-6541", 0 },,
                    { "Jose", 18, .T., "6348-7537", 3 }}
  For i := 1 To Len(aArray)
    // O pre-processor substitui o nome da
    // constante pelo seu valor, resultando
    // no seguinte programa-fonte:
    MsgAlert(aArray[i, __NOME  ]) // MsgAlert(aArray[i, 1 ])
    MsgAlert(aArray[i, __FONE  ]) // MsgAlert(aArray[i, 4 ])
    MsgAlert(aArray[i, __IDADE ]) // MsgAlert(aArray[i, 2 ])
    MsgAlert(aArray[i, __ESTCIVIL]) // MsgAlert(aArray[i, 3 ])
    MsgAlert(aArray[i, __NRDEPEND]) // MsgAlert(aArray[i, 5 ])
  Next
Return

// O UDC utilizado para localizações.

#define BRASIL      // Programa-fonte resultante // e sem a constante: //
                    // com a constante BRASIL //
                    // definida: //

User Function TstUDC3() // User Function TstUDC3() // User Function TstUDC3() //
  #IfDef BRASIL // // //
    Local cPais // Local cPais // //
    Local cLingua // Local cLingua // //
    cPais := "Brasil" // cPais := "Brasil" // //
    cLingua := "Portugues" // cLingua := "Portugues" // //
  #Else // // //
    Local cPais // // Local cPais // //
    cPais := "Argentina" // // cPais := "Argentina" // //
  #EndIf // // //
  MsgAlert(cPais+"/"+cLingua) //MsgAlert(cPais+"/"+cLingua) //MsgAlert(cPais+"/"+cLingua) //
Return Nil // Return Nil // Return Nil //

```

Figura 20.63 Exemplos de UDC.

O **#COMMAND** traduz um comando escrito pelo programador de forma simplificada para a forma desejada pelo compilador. Com isso pode-se manter a compatibilidade de fontes escritos em momentos diferentes quando houverem mudanças na sintaxe do AdvPL.

O **#INCLUDE** indica em que arquivo .CH estão os UDCs a serem usados. Estes por sua vez ficam na pasta INCLUDE. Assim, dependendo da situação altera-se o **#INCLUDE** e tem-se um resultado diferente da compilação. Por exemplo, compilar para o ambiente Windows ou Linux.

Semáforos

Alguns campos de numeração do Protheus são fornecidos pelo sistema em ordem ascendente. É o caso, por exemplo, do número do Pedido de Venda e outros que servem como chave primária de arquivos. É preciso ter um controle do fornecimento destes números em especial quando vários usuários estão trabalhando simultaneamente.

O conceito do semáforo é o seguinte: ao ser fornecido um número fica ele reservado até a conclusão da digitação da tela. Se confirmada, o número é indisponibilizado para sempre. Se a tela é abandonada o número volta a ficar disponível mesmo que naquele momento números maiores já tenham sido oferecidos. Com isso mesmo que tenhamos vários usuários digitando, por exemplo, Pedidos de Venda, teremos para cada um números exclusivos e nenhum número não utilizado.

Se porem, futuramente, um Pedido for cancelado, o seu número não é reaproveitado.

São 4 as funções utilizadas neste processo:

GETSXENUM(alias)	obtem o numero sequencial do alias especificado no parâmetro
CONFIRMSXE	confirma o numero
ROLLBACKSXE	descarta o número.
MAYIUSE(numero)	verifica se aquele numero pode ser usado. Por exemplo pode-se querer dar um número diferente a um Pedido que está sendo digitado. Mas este número não pode já ter sido usado.

Ambientes, Arquivos e Índices

É nos Arquivos ou Tabelas (como são chamados os arquivos no SQL) que são armazenados os dados de um sistema. No Protheus estes arquivos podem ter uma estrutura mais simples e econômica, com arquivos do tipo DBF/ADS, do fabricante Extended System ou CTREE do fabricante ou uma estrutura mais robusta e complexa, em bases SQL (SQLSERVER da Microsoft, ORACLE, DB II da IBM, MYSQL, POSTGREE, etc). No caso do SQL o acesso é feito através do TOPCONNECT, que converte os comandos do AdvPL para este ambiente. As vantagens do SQL já foram vistas no Capítulo 6.

Os arquivos servem a todos os ambientes do Protheus, ambientes (antigamente denominados Módulos) estes que na verdade estão divididos por menus (arquivos .XNU), por uma questão meramente organizacional, ou seja, o usuário pode incluir ou excluir a vontade opções de cada menu, pois o que temos são objetos avulsos e não executáveis linkados.

Hoje temos 61 ambientes, descritos de forma genérica no capítulo 3, conforme ilustração da Figura 20.64.

Ambiente	Identificação
SIGAATF	Ativo Fixo
SIGACOM	Compras
SIGACON	Contabilidade
SIGAEST	Estoque e Custos
SIGAFAT	Faturamento
SIGAFIN	Financeiro
SIGAFIS	Livros Fiscais
SIGAPCP	Planejamento e Controle da Produção
SIGAGPE	Gestão de Pessoal
SIGAFAS	Faturamento de Serviços
SIGAVEI	Veículos
SIGALOJA	Controle de Lojas/Automação Comercial
SIGATMK	Call Center
SIGAOFI	Oficinas
SIGAPON	Ponto Eletrônico
SIGAEIC	Easy Import Control
SIGATCF	Terminal
SIGAMNT	Manutenção de Ativos
SIGARSP	Recrutamento e Seleção de Pessoal
SIGAQIE	Inspeção de Entrada – Qualidade
SIGAQMT	Metodologia – Qualidade

SIGAFRT	Front Loja/Automação Comercial
SIGAQDO	Controle de Documentos – Qualidade
SIGAQIP	Inspeção de Processos – Qualidade
SIGATRM	Treinamento
SIGAEIF	Importação Financeiro
SIGATEC	Field Services
SIGAEEC	Easy Export Services
SIGAEFF	Easy Financing
SIGAEEO	Easy Accounting
SIGAAFV	Administração da Força de Vendas
SIGAPLS	Plano de Saúde
SIGACTB	Contabilidade Gerencial
SIGAMDT	Medicina e Segurança no Trabalho
SIGAQNC	Controle de não conformidades – Qualidade
SIGAQAD	Controle de Auditoria – Qualidade
SIGAQCP	Controle Estatístico de Processo – Qualidade
SIGAOMS	Gestão de Distribuição
SIGACSA	Cargos e Salários
SIGAPEC	Autopeças
SIGAWMS	Gestão de Armazenagem
SIGATMS	Gestão de Transportes
SIGAPMS	Gestão de Projetos
SIGACDA	Controle de Direitos Autorais
SIGAACD	Automação de Coleta de Dados
SIGAREP	Replica
SIGAGE	Gestão Educacional
SIGAEDC	Easy Draw Back Control
SIGAHSP	Gestão Hospitalar
SIGAVDOC	Viewer
SIGAAPD	Avaliação e Pesquisa de Desempenho
SIGAGSP	Gestão de Serviços Públicos
SIGACRD	Fidelização e Análise de Crédito
SIGASGA	Gestão Ambiental
SIGAPCO	Planejamento e Controle Orçamentário
SIGAGPR	Gerenciamento de Pesquisa e Resultado
SIGAGAC	Gestão de Acervos
SIGAHEO	Estrutura Organizacional
SIGAHGP	Gestão de Pessoal
SIGAHHG	Ferramentas de Informação
SIGAHPL	Planejamento e Desenvolvimento

Figura 20.64 Ambientes Protheus.

O nome de cada arquivo no Protheus é constituído de 6 dígitos. O primeiro dígito indica a família. De início, tínhamos apenas a família S (de SIGA). Depois praticamente todas as letras do alfabeto, com a entrada dos novos ambientes. Assim temos:

S	Arquivos pertencentes ao sistema básico, também chamado Classic (Compras, Estoque/Custos, PCP, Faturamento, Livros Fiscais, Financeiro, Contabilidade e Ativo Fixo);
A	Gestão de Projetos;
C	Contabilidade Gerencial;
D	Transportadoras e derivados;
E	Comércio Exterior e derivados;
G	Gestão Hospitalar;
J	Gestão Educacional;
N	Serviços Públicos;
Q	Qualidade e derivados;
R	Recursos Humanos e derivados;
T	Plano de Saúde;
W	Workflow;

O segundo e terceiro dígito indicam subdivisões da família. O SZ, por exemplo, são arquivos reservados para os usuários do sistema abrirem seus próprios arquivos.

No Dicionário de Dados os 1682 arquivos do Protheus podem ser analisados com mais detalhes, inclusive o nível de detalhamento de cada campo com seus respectivos *Helps*.

No quarto e quinto dígito fica o código da empresa, lembrando que um arquivo pode ser usado por mais de uma empresa, ou seja, a empresa 06 pode, por exemplo, acessar o arquivo de clientes da empresa 01 (SA1010). Ou seja, o arquivo tem em seu nome o código da empresa que o usa de forma mais determinada.

O 6º dígito é sempre 0 e não é utilizado.

Cada arquivo tem ainda seus índices, definidos no Configurador.

Funções de Leitura e Gravação em Disco

As funções de Entrada (leitura de arquivos) e Saída (gravação em arquivos) definem basicamente:

- qual arquivo está sendo tratado;
- quais campos devem ser lidos ou atualizados;
- como é feito o acesso direto.

Nesta parte faremos um comparativo entre DBF/Ctree e bases SQL.

No DBF/Ctree sempre é lido o registro inteiro, enquanto que no SQL pode-se ler apenas os campos necessários naquele processamento.

O acesso direto é feito através de Índices que são arquivos paralelos ao arquivo de dados e que contém a chave e o endereço do registro, de forma análoga ao índice de um livro. Para cada chave é criado um índice próprio, embora todos eles fiquem armazenados em um único arquivo .CDX. No SQL, os índices ficam no próprio Banco de Dados.

A cada inclusão ou alteração de um registro todos os índices são atualizados. Daí o cuidado em não se criar uma quantidade excessiva de índices, pois o processo de atualização pode perder desempenho. Se for necessária a criação de um novo índice para um determinado relatório ou consulta pode ser feita em tempo de execução.

No DBF/Ctree o endereço do registro é definido pelo seu *RecNumber* (Recno) que é um número ascendente de acordo com a inclusão do registro no arquivo. No entanto, quando se faz um *Pack* para apagar fisicamente os registros deletados, o *RecNumber* é renumerado. Neste caso todos os índices são refeitos automaticamente.

Já no SQL este número é gravado no campo **R_E_C_N_O_** e a marca de deleção no campo **D_E_L_E_T_**.

O **R_E_C_N_O_** jamais é alterado, transformando-se no **ID** (chave primária) do registro. O campo **D_E_L_E_T_** precisa ser testado pelo programa, para, se estiver marcado, o registro ser desprezado.

O AdvPl permite que se escreva os comandos de entrada e saída exatamente iguais para bases DBF/Ctree ou bases SQL, pois nestas últimas o Top Connect se encarrega de traduzi-los para a sintaxe do Banco de Dados em uso.

Porém, para quem tem base SQL, há certas rotinas que tem um desempenho melhor se escritas dentro dos princípios do tratamento em bloco, característica do SQL. E também é preciso entender que certos programadores preferem usar a sintaxe SQL.

A função que seleciona o arquivo a ser tratado é a **DBSelectArea**.

Ela cria o arquivo, se ele ainda inexistir, a partir do SX3, onde foram definidos os seus campos.

Caso exista, mas estiver fechado, ela o abre.

Ou seja os arquivos são abertos sob demanda, quando selecionados. Ao sair da seção (janela principal/thread) o arquivo é fechado.

O **DBSetOrder** indica qual é o índice escolhido. Lembre-se que todos os índices definidos no SX3 são atualizados a qualquer nova inclusão/alteração, mas é apenas o índice selecionado que permite o acesso direto e é usado na leitura seqüencial.

O **DBSeek(chave)** acessa diretamente um registro.

O **DBSkip(n)** avança ou retrocede **n** registros, na seqüência definida pelo índice ativo.

O **DBGoTop** vai para o primeiro registro.

O **DBGoBottom** para o último. É usado quando se quer ler o arquivo em ordem decrescente.

A expressão lógica **EOF0** igual a verdadeiro indica que se chegou ao fim do arquivo.

A função **IndRegua** permite criar um novo Índice, inclusive filtrando registros, o que é muito útil na emissão de relatórios onde nem todos os registros devem ser considerados.

O **DBSetFilter**, por sua vez, filtra registros independente do índice.

Observação	Veja o exemplo do programa TST disponível no CD do Livro, há uma série de exemplos de tratamentos de I/O.
-------------------	---

Estes comandos podem também, opcionalmente, ser escritos em sintaxe SQL sendo traduzidos pelo Top Connect, mesmo porque nem todas as bases SQL são compatíveis entre si.

A principal função é a **TCGenQuery** que trata o comando **SELECT**. Este comando permite que, de forma coloquial, ou seja, em uma linguagem facilmente assimilada

por um leigo em programação, se selecione as linhas (registros) de uma ou mais tabelas (arquivos) relacionadas entre si. O **SELECT** tem ainda a vantagem de trazer para a estação apenas as colunas (campos) solicitados, reduzindo assim o tráfego na rede.

Esta seleção é feita através das cláusulas **Where**, **Like** e **Between**.

A sequência em que as linhas são trazidas é definida pelo **OrderBy**, independente da existência de um índice.

O **SELECT** também permite que se traga valores sintetizados, com as cláusulas **Sum**, **Count**, **Avg**, **Max**, **Min**, **Group By**, **Distinct**, **Having**.

Com o **Joins**, **Inner Join**, **Left Join**, **Right Join**, **Full Join**, **Cross Join**, **Union** e **Sub Selects** pode-se mesclar tabelas.

Uma vez feito o **Select** cria-se uma **Workarea** sobre a qual pode-se navegar com os comandos do AdvPl, como se ela fosse uma tabela, com os comandos **DBSkip**, o **DBGoTop**, **DBGoBottom**.

Veja no programa como isso ocorre na prática.

Até aqui vimos como se recupera ou se lê o conteúdo de um arquivo. Vamos ver, agora, como é feita sua atualização.

A função **RecLock** bloqueia o registro para que a alteração seja feita. Mas se o segundo parâmetro for **.T.**, ela insere um novo registro em branco no arquivo.

A partir daí, em ambos os casos, para atualizá-lo basta atribuir aos campos do arquivo o conteúdo desejado.

Para excluir um registro usa-se **DBDelet()**. Lembrando que o registro só é marcado para deleção. Para excluí-lo fisicamente, deve-se utilizar a função **Pack()**.

Para zerar um arquivo, usa-se a função **DBZap()**, ou seja, é feita uma deleção total seguida de um **Pack**.

De forma análoga, o processo de atualização também pode ser escrito usando a sintaxe SQL.

Isto é feito com a função **TCSQLExec** que trata os comandos **INSERT**, **UPDATE** e **DELET**.

Também aqui o processo é feito em bloco, ou seja, estes comandos tratam, a cada

chamada, um conjunto de registros.

O SQL é na verdade uma linguagem de programação completa. Quando se escreve uma rotina em SQL (StoreProcedure) pode ela ser acionada ou automaticamente a partir de uma atualização no banco, via comandos **INSERT**, **UPDATE** e **DELET** levando então o nome de **TRIGGER** (Gatilhos no Banco de Dados SQL, sem vínculo com os gatilhos do Protheus) ou ser chamada por um programa através da função **TCSPEXEC** (Top Connect, execute a Store Procedure).

Por estarem inseridas no banco, Stored Procedures são muito eficientes em tratamentos tipicamente “batch” (reprocessamento de custos, de atualização de saldos, de cálculo da Folha de Pagamento, etc)

O Protheus já dispõe de um conjunto de rotinas escritas em SQL e que podem substituir aquelas escritas em AdvPL.

Há ainda um conjunto de funções Top Connect, necessárias em tratamentos mais específicos:

TCCanOpen	verifica a existência de tabelas e índices no servidor;
TCConType	especifica o protocolo de comunicação a ser utilizado pelo Top Connect;
TCDelFile	exclui uma tabela do servidor;
TCGetDB	retorna o tipo de Banco de Dados corrente;
TCLink	abre uma conexão com outro servidor. Com o uso de Web Services esta função deixou de ter serventia, pois ele permite a abertura irrestrita e insegura de um Banco de Dados, fato que não ocorre com um Web Service ;
TCQuit	encerra a conexão aberta com o TCLink;
TCSetConn	seleciona a conexão ativa;
TCSetField	converte dados caracter para numéricos, data e lógicos;
TCSPExist	verifica a existencia de um Stored Procedure no servidor;
TCSQLError	retorna o ultimo erro registrado pelo Top Connect durante a execução de uma query;

TCSrvType	retorna o tipo do servidor do Top Connect;
TCUnlink	encerra uma conexão com o Top Connect.

Programas de Atualização

Os programas de atualização de cadastros e digitação de movimentos seguem um padrão que se apóia no Dicionário de Dados.

Basicamente são três os modelos mais utilizados:

Modelo 1	Para cadastramentos em tela cheia. Exemplo: Cadastro de Cliente;
Modelo 2	Cadastramentos ainda envolvendo apenas um arquivo mas com um cabeçalho e opcionalmente um roda-pé e um corpo com quantidade ilimitada de linhas. Ideal para casos onde há dados que se repetem por vários itens, que, por isso, são colocados no cabeçalho. Exemplo: Pedido de Compra;
Modelo 3	Cadastramentos envolvendo dois arquivos, um com dados de cabeçalho e outro, digitado em linhas, com os itens. Exemplo: Pedido de Vendas, Orçamento, etc.

Todos os modelos são genéricos, ou seja, o programa independe do arquivo a ser tratado, bastando praticamente que se informe apenas o seu Alias. O resto é obtido do Dicionário de Dados (SX3).

Modelo 1

Para fazer-se uma tela de cadastramento simples, ou seja, de um único arquivo, com um *browse* inicial seguido da tela de digitação com as opções de Inclusão, Alteração, Visualização, Exclusão e Pesquisa basta escrever-se uma única chamada de função, a **AxCadastro**, conforme o exemplo ilustrado na Figura 20.65.

User Function CAD

```

Local cVldAlt:=".T." // Validação para
                        // permitir a alteracao.
Local cVldExc:=".T." // Validacao para
                        // permitir a exclusao.

Private cAlias := "SZ1"
DBSelectArea("SZ1")
DBSetOrder(1)
AxCadastro(cAlias,"Cadastro de Clientes",cVldAlt,cVldExc)

```

Return Nil

Figura 20.65 Exemplo do Modelo 1.

Este programa seguirá rigorosamente as definições do arquivo SZ1 no Dicionário de Dados, ou seja a atualização do arquivo de Contas. O primeiro parâmetro contém o nome do cadastro, o segundo o título apresentado na janela e os dois últimos permitem um tratamento próprio para validar as alterações e exclusões. Assim, a função de usuário CAD chama a AxCadastro que chama a função mBrowse que por sua vez chama as funções AxPesqui, AxVisual, AxInclui, AxAltera, AxDeleta, conforme demonstra a Figura 20.66.

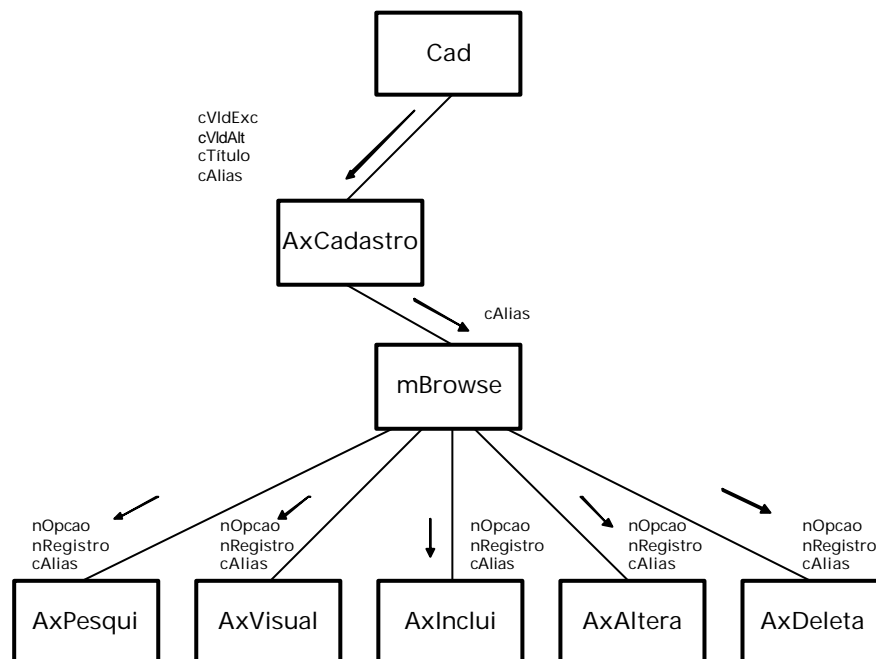


Figura 20.66 Conexão e Passagem de Parâmetros para as Funções Básicas de Cadastro.

Note que o nível de controle sobre a execução das funções de cadastro é mínima pois poucos parâmetros foram passados para a função AxCadastro os quais serão repassados para a função mBrowse.

maginemos agora que se queira fazer um tratamento especial nos processos de inclusão, alteração e exclusão. Isto ocorre, por exemplo, no cadastramento das Transações, pois é preciso, além de gravá-lo, também atualizar o arquivo de contas, sendo que na Alteração é preciso agir em duas, na antiga e na nova, caso a alteração envolva uma mudança de Nome do Cliente.

Para permitir que nestes casos se crie funções próprias existe o *array* aRotina que armazena os seus nomes. Desta forma basta criar as funções próprias e colocar seus nomes no *array*. Por fim, chamar a mBrowse, conforme o exemplo de programa fonte ilustrado na Figura 20.67.

```
#Include "RWMAKE.CH"

User Function TranM1()

    Private cNomAnt, cTipAnt, nValAnt
    Private cAlias      := "SZ2"
    Private aRotina     := {}
    Private lRefresh    := .T.
    Private cCadastro := "Transação de Depósito ou Saque"

    AAdd( aRotina, {"Pesquisar" , "AxPesqui", 0, 1} )
    AAdd( aRotina, {"Visualizar", "AxVisual", 0, 2} )
    AAdd( aRotina, {"Incluir"   , "u_Inclui", 0, 3} )
    AAdd( aRotina, {"Alterar"   , "u_Altera", 0, 4} )
    AAdd( aRotina, {"Excluir"   , "u_Deleta", 0, 5} )

    dbSelectArea(cAlias)
    dbSetOrder(1)

    mBrowse(,,,cAlias)

Return Nil
```

Figura 20.67 Exemplo de Função de Transação.

O mBrowse se incumba de efetuar a chamada das funções atribuídas no *array* aRotina passando os respectivos parâmetros. Desta forma é possível um maior controle sobre as atualizações realizadas por meio das funções definidas, conforme ilustra a Figura 20.68.

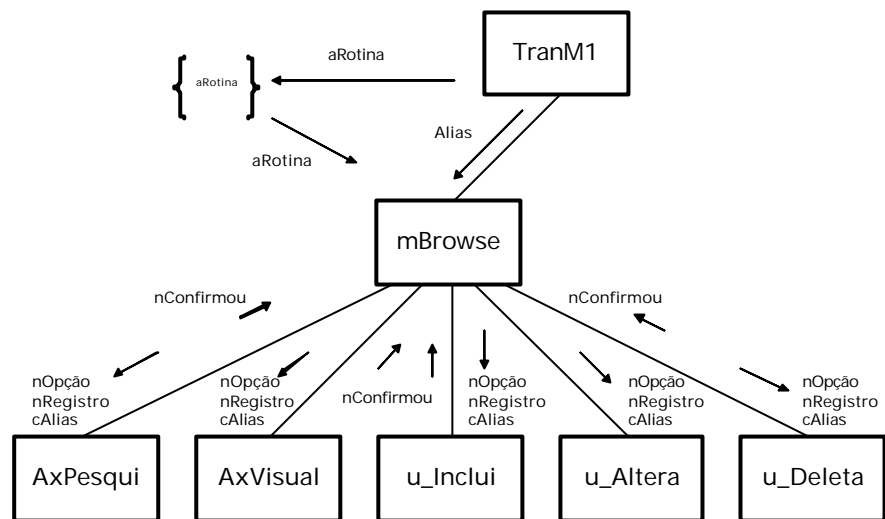


Figura 20.68 A conexão e a Passagem de Parâmetros entre as Funções.

Note que neste caso as funções de inclusão, alteração e deleção serão substituídas por funções de usuário cujos fontes estão exemplificados na Figura 20.69, Figura 20.70 e Figura 20.71, respectivamente.

```
User Function Inclui(cAlias, nRegistro, nOpcao)

    Local nConfirmou
    // A função AxInclui mostra a tela e inclui o registro.
    nConfirmou := AxInclui(cAlias, nRegistro, nOpcao)

    // retorna 1 se confirmou
    If nConfirmou == 1
        Begin Transaction

        // Atualiza o saldo.
        dbSelectArea("SZ1")
        dbSetOrder(1)
        dbSeek(xFilial("SZ1") + SZ2->Z2_Nome)
        RecLock("SZ1", .F.)

        If SZ2->Z2_Tipo = "D" // deposito, logo soma.
            SZ1->Z1_Saldo := SZ1->Z1_Saldo + SZ2->Z2_Valor
        Else // saque, logo subtrai.
            SZ1->Z1_Saldo := SZ1->Z1_Saldo - SZ2->Z2_Valor
        EndIf

        MSUnlock()
        End Transaction
    EndIf
    Return Nil
```

Figura 20.69 Exemplo de Função de Inclusão.

```

User Function Altera( cAlias,nRegistro,nOpcao )

    Local nConfirmou := 0

    cNomAnt := SZ2->Z2_Nome // Salva os dados
    cTipAnt := SZ2->Z2_Tipo // que estavam no
    nValAnt := SZ2->Z2_Valor // arquivo SZ2.

    // Monta a tela, recebe a alteração e retorna 1 se confirmou ou 0 se cancelou.
    nConfirmou := AxAlterar(cAlias,nRegistro,nOpcao)

    If nConfirmou == 1 // Confirmou a alteração.

        Begin Transaction

        // Verifica se houve alteração em algum dos campos de SZ2.
        If (SZ2->Z2_Nome <> cNomAnt .Or.;
            SZ2->Z2_Tipo <> cTipAnt .Or.;
            SZ2->Z2_Valor <> nValAnt)

            // Desatualiza a conta com os dados antigos.
            dbSelectArea("SZ1")
            dbSetOrder(1)
            dbSeek(xFilial("SZ1") + cNomAnt)
            RecLock("SZ1", .F.)
            If cTipAnt <> "D"
                SZ1->Z1_Saldo := SZ1->Z1_Saldo + nValAnt
            Else
                SZ1->Z1_Saldo := SZ1->Z1_Saldo - nValAnt
            EndIf

            MSUnlock()

            // Atualiza o novo movimento.
            dbSelectArea("SZ1")
            dbSetOrder(1)
            dbSeek(xFilial("SZ1") + SZ2->Z2_Nome)
            RecLock("SZ1", .F.)
            If SZ2->Z2_Tipo == "D"
                SZ1->Z1_Saldo := SZ1->Z1_Saldo + SZ2->Z2_Valor
            Else
                SZ1->Z1_Saldo := SZ1->Z1_Saldo - SZ2->Z2_Valor
            EndIf
            MSUnlock()

        EndIf

        End Transaction

    EndIf

Return Nil

```

Figura 20.70 Exemplo de Função de Alteração.

Embora neste exemplo a função Inclui seja uma função de usuário ela continua chamando a função AxInclui. O objetivo, neste caso, é possibilitar a execução de comandos adicionais na função de usuário após a execução da função AxInclui (que inclui a transação em SZ2) e desta forma atualizar o saldo no cadastro da conta. Analogamente, as funções de alteração e de deleção incorporam tratamento similar na atualização do saldo no cadastro da conta (Figuras 20.70 e 20.71).

```
User Function Deleta(cAlias, nRegistro, nOpcao)

    Local nConfirmou

    // Chama a rotina de visualização para mostrar o movimento a ser excluído.
    nConfirmou := AxVisual(cAlias, nRegistro, 2)

    // Mostra os dados e retorna 1 se confirmou.
    If nConfirmou == 1

        Begin Transaction

            // Exclui o movimento.
            dbSelectArea(cAlias)
            RecLock(cAlias, .F.)
            dbDelete()
            MSUnlock()

            // Desatualiza o saldo.
            dbSelectArea("SZ1")
            dbSetOrder(1)
            dbSeek(xFilial() + SZ2->Z2_Nome)
            RecLock("SZ1", .F.)
            If SZ2->Z2_Tipo <> "D"
                SZ1->Z1_Saldo := SZ1->Z1_Saldo + SZ2->Z2_Valor
            Else
                SZ1->Z1_Saldo := SZ1->Z1_Saldo - SZ2->Z2_Valor
            EndIf
            MSUnlock()

        End Transaction
    EndIf
Return NIL
```

Figura 20.71 Exemplo de Função de Exclusão.

O exemplo da exclusão utiliza a função AxVisual para mostrar na tela o registro e solicitar a confirmação. Caso tenha sido confirmado, o programa efetua a eliminação do registro no arquivo de transações e atualiza o respectivo saldo no arquivo de Cadastro de Contas.

Modelo 2 e Modelo 3

Imaginemos agora que se queira fazer uma tela de cadastramento, mas que tenha um Cabeçalho, ou seja, alguns campos se repetem ao longo de todos

os itens, estes colocados em linhas.

Estes dados repetitivos podem ficar ou no mesmo arquivo, como é o caso do Pedido e das Solicitações de Compra, ou em arquivos separados, recomendado quando for grande a quantidade destes campos. Já no Pedido de Venda, onde código do cliente, da transportadora, dos vendedores, a condição de pagamento, data de emissão, código de reajuste, etc são iguais para todos os itens, ficam eles em um arquivo próprio (SC5) e cada um dos itens em outro (SC6). A amarração é feita pelo número do Pedido.

O primeiro caso é o Modelo 2 e o segundo caso é o Modelo 3.

Modelo 2

Considerando que no arquivo de Transações, que tem sete campos (Nome, Número e Item do Documento, Data, Tipo, Histórico e Valor), e que o Nome, o Número e a Data são repetitivos a cada lote, é interessante colocar estes dados no cabeçalho e deixar os campos Item, Histórico, Tipo e Valor para serem digitados em cada linha, conforme ilustra a Figura 20.72.

Cabeçalho (Enchoice)

Nome:	<input type="text"/>	Número:	<input type="text"/>
Data:	<input type="text"/>		

Linhas (GetDados)

Item	Histórico	Tipo	Valor
<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>
<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>
<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>
<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>
<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>

Figura 20.72 Exemplo de Formato dos Dados em Linhas e Cabeçalho no Modelo 2.

O programa TranM2 monta o *array* aRotina e para dar um tratamento próprio às atualizações, através da função Mod2Manut, chama inicialmente a mBrowse. A Mod2Manut prepara as variáveis para a chamada da função Modelo2 que é quem abre a tela de digitação. Isto envolve o *array* aHeader, que contém as informações do Dicionário de Dados, o *array* aCols que contém os dados de cada linha do documento. No caso de inclusão são montadas linhas em branco, respeitando as opções de inicializações.

O cabeçalho é tratado pelo função Enchoice, que por sua vez necessita do array aCabecalho devidamente preenchido com dados de cada campo (neste modelo com campos definidos pelo programador). O corpo do documento é tratado pela função **GetDados**. Ambas as funções são chamadas pela função Modelo2.

Após a chamada da função Modelo2, que retorna .T. ou .F. caso a atualização tenha sido confirmada ou abandonada, são feitos os tratamentos de atualização dos arquivos (por meio das funções MD2Inclu, Md2Alter e Md2Exclu). Neste tratamento é utilizada a função **FieldPut**, que grava o conteúdo no campo com base em um número que indica a sua posição no registro, a **FieldGet** que o traz para a memória e a função **FieldPos**, que obtém este número a partir do nome do campo que está no aHeader. São estas duas funções que permitem que os programas sejam genéricos, independente do arquivo que está sendo tratado.

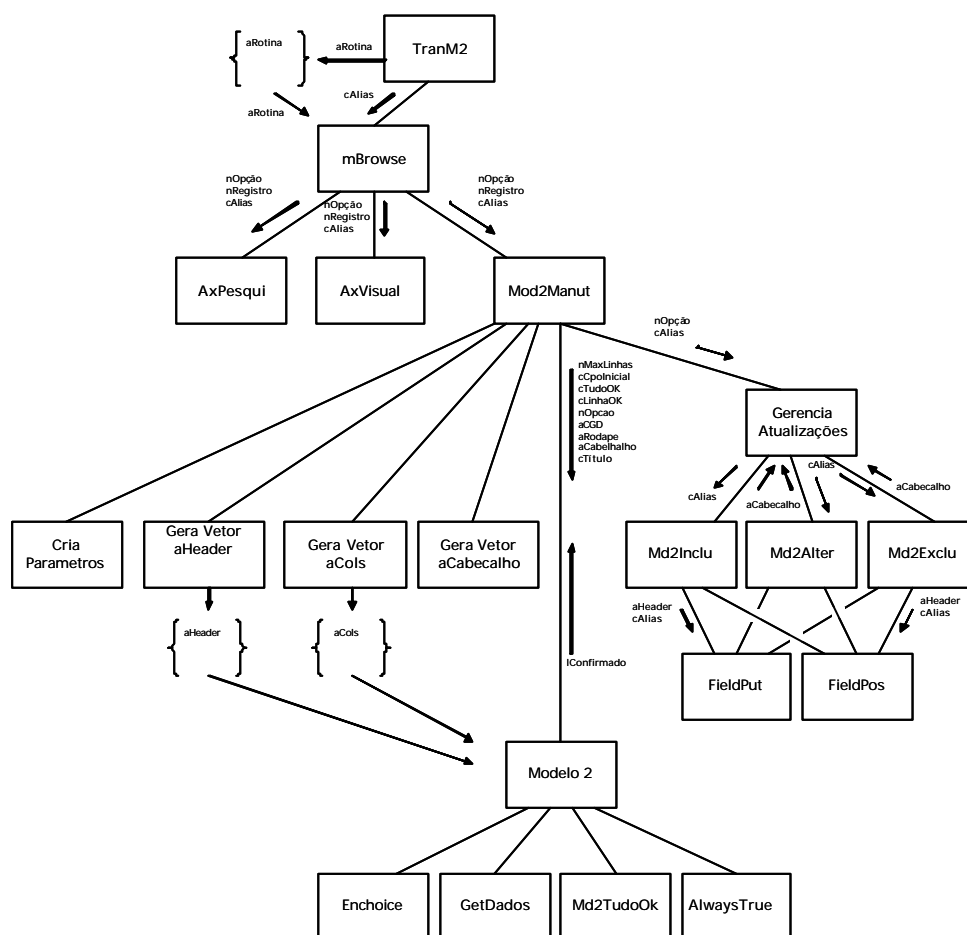


Figura 20.73 Estrutura e Interconexão de Funções para o Modelo 2.

A destacar ainda a função Md2TudoOK que faz um teste final antes de iniciar o processo de atualização e a função AllwaysTrue que valida cada linha digitada, conforme a ilustração do digrama da Figura 20.73.

O exemplo de programa fonte ilustrado na Figura 20.74 mostra como o vetor aRotina redireciona as funções de inclusão, alteração e exclusão para a função Mod2Manut.

```

////////////////////////////////////
// TRANM2                                     //
////////////////////////////////////

#include "rwmake.ch"

User Function TRANM2()

    Private Nomant,Tipant,Valant
    Private cAlias      := "SZ2"
    Private aRotina     := {}
    Private lRefresh    := .T.
    Private cCadastro := "Transação de Saque ou Depósito"

    aAdd ( aRotina,{"Pesquisar","AxPesqui"    ,0,1} )
    aAdd ( aRotina,{"Visualiz.","AxVisual"    ,0,2} )
    aAdd ( aRotina,{"Incluir"  ,"u_Mod2Manut",0,3} )
    aAdd ( aRotina,{"Alterar"  ,"u_Mod2Manut",0,4} )
    aAdd ( aRotina,{"Excluir"  ,"u_Mod2Manut",0,5} )

    dbSelectArea(cAlias)
    dbSetOrder(1)
    dbGoTop()

    mBrowse(,,,cAlias)

Return Nil

```

Figura 20.74 Função TranM2.

A função Mod2Manut efetua a criação dos parâmetros, a geração do vetor aHeader, a geração do vetor aCols, a geração do vetor aCabecalho, a chamada da função Modelo2 e do respectivo gerenciamento da inclusão, alteração e exclusão do cadastro.

Para um melhor entendimento da função Mod2Manut, cada uma das figuras a seguir agrupa os comandos necessários para a realização destas tarefas.

```

////////////////////////////////////
// CRIA PARÂMETROS PARA O MODELO 2
////////////////////////////////////

User Function Mod2Manut(cAlias,nRegistro,nOpcao)

    Local cChave      := ""
    Local nCols       := 0
    Local i           := 0
    Local lConfirmado := .F.

// Parametros da funcao Modelo2().
// Modelo2(cTitulo,aCabecalho,aRodape,aCGD,nOpcao,cLinOK,;
//         cAlloK, , ,cCpoNumInicial,nMaxLinhas)

    Private cTitulo      := "Digitação de Depósitos" // Titulo.
    Private aCabecalho   := {} // Campos do Enchoice.
    Private aRodape      := {} // Campos do Rodape.
    Private aCGD         := {} // Coordenadas do objeto GetDados.
    Private cLinhaOK     := "" // Funcao para validacao de uma linha da GetDados.
    Private cAlloK       := "u_MdaTudOK()" // Funcao para validacao de tudo.
    Private aGetsGD      := {} // posição para edição dos itens (GetDados)
    Private bF4          := {} // Bloco de Codigo para a tecla F4.
    Private cCpoInicial := "+Z2_ITEM" // String com o nome dos
                                     // campos que devem inicializados
                                     // ao pressionar a seta para baixo. +Z2_ITEM"

    Private nMaxLinhas   := 99 // Nr. maximo de linhas na GetDados.
    Private aHeader      := {} // Cabecalho da coluna da GetDados.
    Private aCols        := {} // Colunas da GetDados.
    Private nCount       := 0
    Private bCampo       := {|nField| FieldName(nField)}
    Private dData        := CtoD(" / / ")
    Private cNome        := Space(20)
    Private cTipo        := "D"
    Private aAlt         := {}

// Cria variáveis de memória: para cada campo da tabela,
// cria uma variável de memória com o mesmo nome.

    dbSelectArea(cAlias)
    For i := 1 To FCount() // Criação das Variáveis de Memória.
        M->&(Eval(bCampo, i)) := CriaVar(FieldName(i), .T.)
        // Assim tambem funciona: M->&(FieldName(i)) := CriaVar(FieldName(i), .T.)
    Next

```

Figura 20.75 Função Mod2Manut: Comandos para a Criação dos Parâmetros.

```

////////////////////////////////////
// GERA VETOR aHEADER //
////////////////////////////////////

dbSelectArea("SX3")      // Seleciona o Dicionário de Dados.
dbSetOrder(1)           // Coloca o índice 1 ativo.
dbSeek(cAlias)          // Lê o primeiro registro deste arquivo.

While !Eof() .And. SX3->X3_ARQUIVO == cAlias

    // Verifica se o campo é usado e se o usuário tem nível para usá-lo.
    If X3Uso(X3_USADO) .And. cNivel >= X3_NIVEL

        // monta um array, chamado aHeader, com as
        // propriedades do campo obtidas no SX3

        aAdd(aHeader , { TRIM(X3_TITULO) ,;
                          X3_CAMPO      ,;
                          X3_PICTURE   ,;
                          X3_TAMANHO   ,;
                          X3_DECIMAL   ,;
                          X3_VALID     ,;
                          X3_USADO     ,;
                          X3_TIPO      ,;
                          X3_ARQUIVO    ,;
                          X3_CONTEXT    })

    Endif
    dbSkip()             // Evolve no Arquivo.

End

```

Figura 20.76 Função Mod2Manut: Comandos para a Geração do Vetor aHeader, rotina que carrega as propriedades de cada campo buscando-as no próprio Dicionário de Dados (SX3).


```

////////////////////////////////////
// GERA VETOR aCOLs
////////////////////////////////////

// Seleciona o Arquivo.
dbSelectArea(cAlias)
dbSetOrder(2)

// A opção selecionada não é INCLUIR. É ALTERAR, EXCLUIR OU VISUALIZAR.
If nOpcao <> 3
  cNumero := (cAlias)->Z2_Numero
  cNome    := (cAlias)->Z2_Nome
  dData    := (cAlias)->Z2_Data
  cTipo    := (cAlias)->Z2_Tipo
  dbSeek(xFilial(cAlias) + cNumero)
  While !EOF() .And. (cAlias)->(Z2_Filial+Z2_Numero) == xFilial(cAlias)+cNumero
    // Cria uma linha vazia em aCols.
    AAdd(aCols, Array(Len(aHeader)+1))
    nCols++
    // Preenche a linha que foi criada com os dados contidos na tabela.
    For i := 1 To Len(aHeader)
      // Campo não é virtual.

      If aHeader[i][10] <> "V"
        // Carrega o conteúdo do campo.
        aCols[nCols][i] := FieldGet(FieldPos(aHeader[i][2]))
      Else
        // A função CriaVar() lê as definições do campo no Dic.Dados e
        // e carrega a variável de acordo com o Inicializador-Padrão,
        // que, se não foi definido, assume conteúdo vazio.
        aCols[nCols][i] := CriaVar(aHeader[i][2], .T.)
      EndIf
    Next
    // Cria a ultima coluna para o controle da GetDados: deletado ou nao.
    aCols[nCols][Len(aHeader)+1] := .F.
    // Atribui o numero do registro neste vetor para o controle na gravacao.
    AAdd(aAlt, Recno())
    dbSelectArea(cAlias)
    dbSkip()
  End
End

// A opção selecionada é INCLUIR.
Else
  // Atribui à variável o inicializador padrão do campo.
  cNome    := space(20)
  cTipo    := "D"
  dData    := dDataBase
  cNumero  := GetSXENum()
  // Cria uma linha em branco e preenche de acordo com o
  // Inicializador-Padrao do Dic.Dados.
  AAdd(aCols, Array(Len(aHeader)+1))
  For i := 1 To Len(aHeader)
    aCols[1][i] := CriaVar(aHeader[i][2])
  Next
  // Cria a ultima coluna para o controle da GetDados: deletado ou nao.
  aCols[1][Len(aHeader)+1] := .F.
  // Atribui 01 para a primeira linha da GetDados.
  aCols[1][AScan(aHeader, {|x| Trim(x[2])=="Z2_ITEM"})] := "01"
EndIf

```

Figura 20.77 Função Mod2Manut: Comandos para a Geração do Vetor aCols, com os dados do cabeçalho de cada linha.

```

/////////////////////////////////////////////////////////////////
// GERA VETOR aCABECALHO //
/////////////////////////////////////////////////////////////////

// aCabecalho[n][1] = Nome da variavel. Ex.: "cNome"
// aCabecalho[n][2] = Array com as coordenadas do Get [x,y], em Pixel.
// aCabecalho[n][3] = Titulo do campo
// aCabecalho[n][4] = Picture
// aCabecalho[n][5] = Validacao
// aCabecalho[n][6] = F3
// aCabecalho[n][7] = Se o campo é editavel, .T., senao .F.

AAdd(aCabecalho,{ "cNome" ,{15, 10},"Nome" , "@" , existcpo("SZ1",M->Z2_NOME,1),"SZ1", (nOpc==3)})
AAdd(aCabecalho,{ "cTipo" ,{15,200},"Tipo" , "@" , , , .F.})
AAdd(aCabecalho,{ "dData" ,{35, 10},"Data" , "99/99/99" , , , (nOpc==3)})
AAdd(aCabecalho,{ "cNumero",{35,200},"Número","9.999" , , , (nOpc==3)})

// Coordenadas do objeto GetDados.
aCGD := {55,5,128,315}

// Validacao na mudanca de linha quando clicar no botao OK.
cLinOK := "ExecBlock('AllwaysTrue',.F.,.F.)"
dData := dDataBase
cTitulo := "Transações Modelo 2"

```

Figura 20.78 Função Mod2Manut: Comandos para a Geração do Vetor aCabecalho.

```

/////////////////////////////////////////////////////////////////
// CHAMA A FUNÇÃO MODELO 2 E GERENCIA FUNÇÕES DE CADASTRO //
/////////////////////////////////////////////////////////////////

// Chama a Função Modelo 2 e devolve confirmação na variável lConfirmado.
lConfirmado := Modelo2(cTitulo,aCabecalho,aRodape,aCGD,nOpcao,cLinha,;
                      cAllok, , ,cCpoInicial,nMaxLinhas)

// Confirmou (.T.) Nao confirmou (.F.).
If lConfirmado

    If nOpcao == 3 // Inclusão.

        If MsgYesNo("Confirma a gravacao dos dados?", cTitulo)
            // Processa a Iclusão.
            Md2Inclu(cAlias)
        EndIf

    ElseIf nOpcao == 4 // Alteração.

        If MsgYesNo("Confirma a alteracao dos dados?", cTitulo)
            // Processa a Alteração.
            Md2Alter(cAlias)
        EndIf

    ElseIf nOpcao == 5 // Exclusão.

        If MsgYesNo("Confirma a exclusao dos dados?", cTitulo)
            // Processa a Exclusão.
            Md2Exclu(cAlias)
        EndIf

    EndIf

Else

    // RollBackSX8()

EndIf

Return NIL

```

Figura20.79 Função Mod2Manut: Chamada da Função Modelo2 e Gerenciamento das Funções de Cadastramento.

Os exemplos das funções que realizam a Inclusão, a Alteração e a Exclusão estão ilustrados na Figura 20.80, na Figura 20.81 e na Figura 20.82, respectivamente. Note que estas funções estão declaradas como *Static Function*. Isto é necessário porque estas funções foram declaradas no mesmo arquivo de programa (PRW) em que foi declarada a função Mod2Manut, porém devem ser visíveis para ela.

```

////////////////////////////////////
// EFETUA A INCLUSÃO DA TRANSAÇÃO E ATUALIZA O SALDO NO CADASTRO      //
////////////////////////////////////

Static Function Md2Inclu(cAlias)

Local i := 0
Local y := 0

DbSelectArea(cAlias)
DbSetOrder(1)

For i := 1 To Len(aCols)

    // A linha nao esta deletada, logo, deve ser gravada.

    If !aCols[i][Len(aHeader)+1]

        RecLock(cAlias, .T.)

        For y := 1 To Len(aHeader)
            FieldPut(FieldPos(Trim(aHeader[y][2])), aCols[i][y])
        Next

        (cAlias)->Z2_Filial := xFilial(cAlias)
        (cAlias)->Z2_Numero := cNumero
        (cAlias)->Z2_Nome   := cNome
        (cAlias)->Z2_Data   := dData
        (cAlias)->Z2_Tipo   := cTipo
        MSUnlock()

        // Atualiza saldo

        DbSelectArea('SZ1')
        DBSetOrder(1)
        dbseek(xFilial("SZ1")+SZ2->Z2_Nome)
        RecLock('SZ1',.F.)

        If SZ2->Z2_Tipo = 'D'
            SZ1->Z1_Saldo := SZ1->Z1_Saldo + SZ2->Z2_Valor
        Else
            SZ1->Z1_Saldo := SZ1->Z1_Saldo - SZ2->Z2_Valor
        Endif
        MSUnlock()
        DbSelectArea('SZ2')

    EndIf

Next

//ConfirmSX8()

Return Nil

```

Figura 20.80 Exemplo de Função para Gerenciar a Opção Inclusão.

```

////////////////////////////////////
// EFETUA A ALTERAÇÃO DA TRANSAÇÃO E ATUALIZA O SALDO NO CADASTRO //
////////////////////////////////////
Static Function Md2Alter(cAlias)
  Local cNomAnt
  Local cTipAnt
  Local nValAnt
  Local i := 0
  Local y := 0
  DbSelectArea(cAlias)
  DbSetOrder(1)

  For i := 1 To Len(aCols)
    If i <= Len(aAlt)
      // aAlt contem os Recno() dos registros originais.
      // O usuario pode ter incluido mais registros na GetDados (aCols).
      DbGoTo(aAlt[i]) // Posiciona no registro.
      RecLock(cAlias, .T.)
      If aCols[i][Len(aHeader)+1] // A linha esta deletada.
        DbDelete() // Deleta o registro correspondente.
        DbSelectArea("SZ1") // Desatualiza.
        DbSeek(xFilial()+SZ2->Z2_Nome)
        RecLock("SZ1",.F.)
        If SZ2->Z2_Tipo = 'D'
          SZ1->Z1_Saldo := SZ1->Z1_Saldo - SZ2->Z2_Valor
        Else
          SZ1->Z1_Saldo := SZ1->Z1_Saldo + SZ2->Z2_Valor
        Endif
        MsUnLock()
        DbSelectArea(cAlias)
      Else
        // Salva os Dados que estavam no arquivo SZ2.
        cNomAnt := SZ2->Z2_Nome
        cTipAnt := SZ2->Z2_Tipo
        nValAnt := SZ2->Z2_Valor
        // Regrava os dados.
        For y := 1 To Len(aHeader)
          FieldPut(FieldPos(Trim(aHeader[y][2])), aCols[i][y])
        Next
        MSUnLock()
        // Acerta Saldo no Z1.
        // Desatualiza.
        DbSelectArea("SZ1")
        DbSeek(xFilial("SZ1")+cNomAnt)
        RecLock("SZ1",.F.)
        If cTipAnt = 'D'
          SZ1->Z1_Saldo := SZ1->Z1_Saldo - nValAnt
        Else
          SZ1->Z1_Saldo := SZ1->Z1_Saldo + nValAnt
        Endif
        MsUnLock()
        // Atualiza.
        DbSelectArea("SZ1")
        DbSeek(xFilial("SZ1")+SZ2->Z2_Nome)
        RecLock("SZ1",.F.)
        If SZ2->Z2_Tipo = 'D'
          SZ1->Z1_Saldo := SZ1->Z1_Saldo + SZ2->Z2_Valor
        Else
          SZ1->Z1_Saldo := SZ1->Z1_Saldo - SZ2->Z2_Valor
        Endif
        MsUnLock()
      Endif
    Endif
  Endif

```

```

Else
    // Foram incluídas mais linhas na GetDados (aCols), logo, precisam ser incluídas.

    If !aCols[i][Len(aHeader)+1]

        RecLock(cAlias, .T.)

        For y := 1 To Len(aHeader)
            FieldPut(FieldPos(Trim(aHeader[y][2])), aCols[i][y])
        Next

        (cAlias)->Z2_Filial := xFilial(cAlias)
        (cAlias)->Z2_Nome := cNome
        (cAlias)->Z2_Data := dData
        (cAlias)->Z2_Tipo := cTipo
        (cAlias)->Z2_Numero := cNumero

        MSUnlock()

        // Atualiza saldo

        DbSelectArea('SZ1')
        DbSetOrder(1)
        Dbseek(xFilial("SZ1")+SZ2->Z2_Nome)
        RecLock('SZ1',.F.)
        If SZ2->Z2_Tipo = 'D'
            SZ1->Z1_Saldo := SZ1->Z1_Saldo + SZ2->Z2_Valor
        Else
            SZ1->Z1_Saldo := SZ1->Z1_Saldo - SZ2->Z2_Valor
        Endif

        MsUnLock()

        DbSelectArea('SZ2')

    EndIf

EndIf

Next

Return Nil

```

Figura 20.81 Exemplo de Função para Gerenciar a Opção de Alteração.

```

////////////////////////////////////
// EFETUA A EXCLUSÃO DA TRANSAÇÃO E ATUALIZA O SALDO NO CADASTRO //
////////////////////////////////////

Static Function Md2Exclu(cAlias)

    DbSelectArea(cAlias)
    DbSetOrder(1)
    DbSeek(xFilial(cAlias) + cNome + cNumero)

    While !EOF()                                .And.;

        // não precisa testar o nome pois número é chave primária.

        (cAlias)->Z2_Filial == xFilial(cAlias) .And.;
        (cAlias)->Z2_Numero == cNumero

        // efetua incremento na régua que demonstra o andamento do processamento.

        IncProc()

        // RecLock(cAlias, .F.) - A função Modelo 2 efetua o bloqueio do registro.

        dbDelete()
        MSUnlock()

        // desatualiza, ou seja, deposito subtrai, saque soma.

        DbSelectArea("sz1")
        DbSeek(xFilial()+SZ2->Z2_Nome)
        RecLock("SZ1",.F.)

        If SZ2->Z2_Tipo = 'D'
            SZ1->Z1_Saldo := SZ1->Z1_Saldo - SZ2->Z2_Valor
        Else
            SZ1->Z1_Saldo := SZ1->Z1_Saldo + SZ2->Z2_Valor
        Endif

        MsUnLock()
        DbSelectArea("sz2")
        dbSkip()

    End

Return Nil

```

Figura 20.82 Exemplo de Função para Gerenciar a Opção de Exclusão.

A Figura 20.83 ilustra mostra o código fonte da função que efetua a validação das linhas da GetDados ao ser realizada a confirmação. O nome da função a ser chamada foi definida no parâmetro cAllok. Este parâmetro foi criado na função Mod2Manut juntamente com a criação dos demais parâmetros necessários para o Modelo 2.

```

////////////////////////////////////
// VALIDA TODAS AS LINHAS DA GETDADOS AO CONFIRMAR A GRAVAÇÃO //
////////////////////////////////////

User Function Md2TudOK()

    Local lRetorno := .T.
    Local i       := 0
    Local nDel    := 0

    For i := 1 To Len(aCols)

        If aCols[i][Len(aHeader)+1]
            nDel++
        EndIf

    Next

    If nDel == Len(aCols)
        MsgInfo("Para excluir todos os itens, utilize a opção EXCLUIR", cTitulo)
        lRetorno := .F.
    EndIf

Return lRetorno

```

Figura 20.83 Função de Validação das Linhas da GetDados na Confirmação.

Já a Figura 20.84 mostra a função que será chamada quando for pressionada a tecla seta para cima ou seta para baixo na GetDados. Note que a chamada desta função será feita pela própria GetDados e o nome da função a ser chamada foi passado para o Modelo2 por meio do parâmetro cLinhaOK. Neste momento é sugerido o desenvolvimento da uma validação nesta função.

```

////////////////////////////////////
// VALIDA A LINHA ATUAL DA GETDADOS AO TECLAR SETA PARA BAIXO OU PARA CIMA //
// PARA A MUDANÇA DE LINHA. //
////////////////////////////////////

User Function AllwaysTrue()

// *** EXERCICIO *** Desenvolva uma rotina para não aceitar valor acima de
//                    1.000.000,00.

Return .T.

```

Figura 20.84 Função de Validação da GetDados na Movimentação das Linhas.

Modelo 3

O Modelo 3 é uma evolução do Modelo 2 já que os dados do cabeçalho fazem parte de um arquivo “pai”, e o corpo é formado por um arquivo “filho”. Um exemplo do uso do Modelo 3 é o cabeçalho e itens do Pedido de Venda. No caso do nosso exemplo usaremos o Z1 como pai e o Z2 como filho, ou seja, no cabeçalho teremos os dados cadastrais e no corpo os movimentos. Estes arquivos são declarados como cAlias1 e cAlias2, conforme mostra a Figura 20.85.

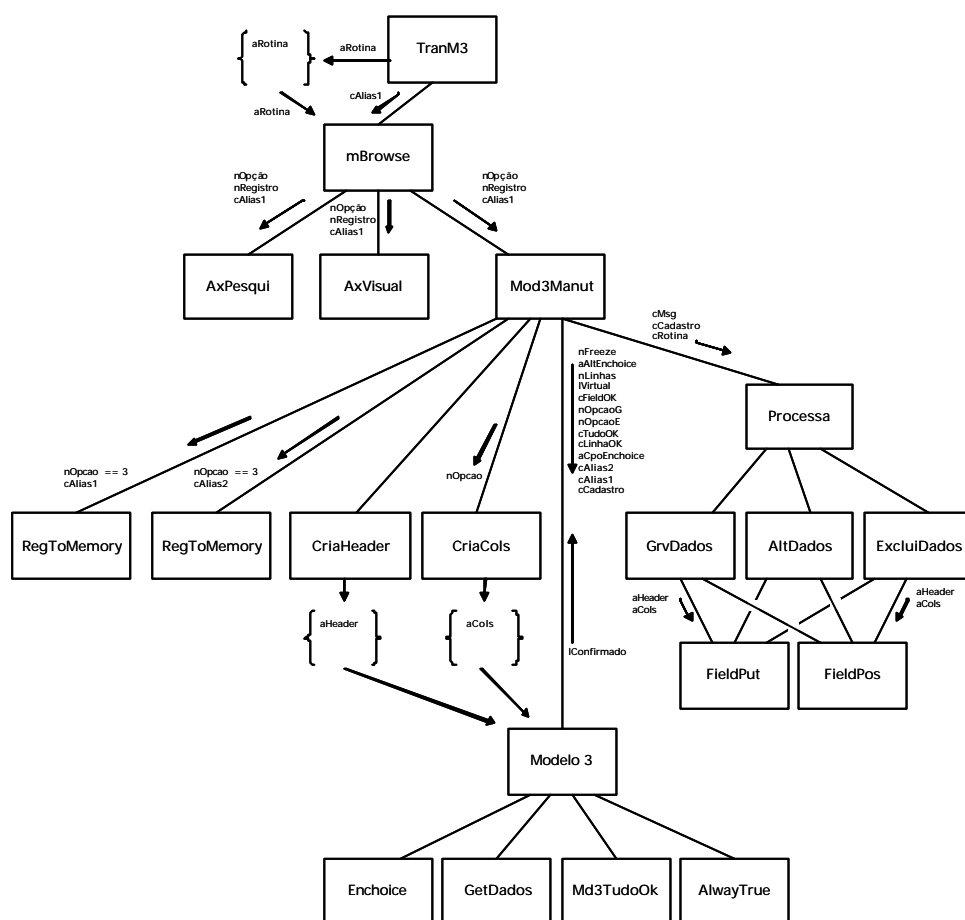


Figura 20.85 Estrutura de Interconexão de Funções para o Modelo 3.

A função **Mod3Manut** gera as variáveis de memória por meio da função **RegToMemory** a qual recebe como parâmetros o **cAlias** correspondente e a expressão **nOpcao == 3** verdadeiro (.T.), caso a opção seja exclusão, ou falso (.F.) caso a opção seja inclusão ou alteração. Este recurso dispensa a utilização do comando **IF** ou **CASE** para a verificar qual foi a opção escolhida pelo usuário.

Embora para a **mBrowse** seja passado o **cAlias1** (relativo ao **SZ1**), tanto o **cAlias1** como o **cAlias2** são declarados como variáveis do tipo *Private*, conforme mostra a Figura 20.86. Isto porque justamente estes arquivos serão utilizados nas chamadas de outras funções.

```

////////////////////////////////////
// TRANM3                                     //
////////////////////////////////////

#include "PROTHEUS.CH"

User Function TranM3()

    Private aRotina := {}
    Private cCadastro := "Transações Modelo 3"
    Private cAlias1 := "SZ1"                // Alias da Enchoice.
    Private cAlias2 := "SZ2"                // Alias da GetDados.

    AAdd(aRotina, {"Pesquisar" , "AxPesqui" , 0, 1})
    AAdd(aRotina, {"Visualizar", "u_Mod3Manut", 0, 2})
    AAdd(aRotina, {"Incluir" , "u_Mod3Manut", 0, 3})
    AAdd(aRotina, {"Alterar" , "u_Mod3Manut", 0, 4})
    AAdd(aRotina, {"Excluir" , "u_Mod3Manut", 0, 5})

    dbSelectArea(cAlias1)
    dbSetOrder(1)
    dbGoTop()
    mBrowse(,,,cAlias1)

Return Nil

```

Figura 20.86 Função TranM2.

Além de chamar as funções para a geração das variáveis de memória e os vetores necessários para o Modelo 3, a função **Mod3Manut** gerencia o tipo de transação (Inclusão, Alteração ou Exclusão) chamando a função **Processa**, conforme ilustração da Figura 20.87.

```

////////////////////////////////////
// MOD3MANUT                                                                //
////////////////////////////////////

User Function Mod3Manut(cAlias, nRecno, nOpcao)

    Local i          := 0
    Local cLinOK      := "AllwaysTrue"
    Local cTudoOK     := "u_Md3TudOK"
    Local nOpcaoE     := nOpcao
    Local nOpcaoG     := nOpcao
    Local cFieldOK    := "AllwaysTrue"
    Local lVirtual    := .T.
    Local nLinhas     := 99
    Local nFreeze     := 0
    Local lRet        := .T.

    Private aCols      := {}
    Private aHeader    := {}
    Private aCpoEnchoice := {}
    Private aAltEnchoice := {}
    Private aAlt       := {}

    // Cria variáveis de memória dos campos da tabela Pai.
    RegToMemory(cAlias1, (nOpcao==3))
    // Cria variáveis de memória dos campos da tabela Filho.
    RegToMemory(cAlias2, (nOpcao==3))
    // Chama a função que cria o vetor aHeader.
    CriaHeader()
    // Chama a função que cria o vetor aCols.
    CriaCols()
    // Chama a função Modelo 3.
    lConfirmado := Modelo3(cCadastro, cAlias1, cAlias2, aCpoEnchoice, cLinOK, ;
                          cTudoOK, nOpcaoE, nOpcaoG, cFieldOK, lVirtual, nLinhas, ;
                          aAltEnchoice, nFreeze)
    // Confirmou (.T.) Não confirmou (.F.).
    If lConfirmado
        If nOpc == 3                                // Inclusão.
            If MsgYesNo("Confirma a gravação dos dados?", cCadastro)
                Processa({|GrvDados()}, cCadastro, "Gravando os dados, aguarde...")
            EndIf

        ElseIf nOpc == 4                            // Alteração.

            If MsgYesNo("Confirma a alteração dos dados?", cCadastro)
                Processa({|AltDados()}, cCadastro, "Alterando os dados, aguarde...")
            EndIf

        ElseIf nOpc == 5                            // Exclusão.
            If MsgYesNo("Confirma a exclusão dos dados?", cCadastro)
                Processa({|ExcluiDados()}, cCadastro, "Excluindo os dados, aguarde...")
            EndIf
        EndIf
    Else
        RollBackSX8()
    EndIf

Return Nil

```

Figura 20.87 Função Mod3Manut.

Embora a criação dos *arrays* *aHeader* e *aCols* é semelhante ao **Modelo 2**, o exemplo de **Modelo 3** apresentado aqui utiliza as funções *CriaHeader* e *CriaCols* para tal tarefa. Neste ponto, a diferença básica entre os exemplos de **Modelo 2** e **Modelo 3** é que a função *CriaHeader* gera também o *array* contendo os campos de Cabeçalho necessários para a função **Enchoice**.

Os exemplos das funções que realizam a Inclusão, a Alteração e a Exclusão estão ilustrados na Figura 20.88, na Figura 20.89 e na Figura 10.90, respectivamente. Nesta funções, para realizar a atribuição dos valores aos campos do banco, são utilizados os *arrays* *aHeader* e *aCols* e as funções *FieldPut* e *FieldPos*.

```

////////////////////////////////////
// GRVDADOS                                                                    //
////////////////////////////////////

Static Function GrvDados()

    Local bCampo := {|nField| Field(nField)}
    Local i      := 0
    Local y      := 0
    Local nItem  := 0

    ProcRegua(Len(aCols) + FCount())

    // Grava o registro da tabela Pai, obtendo o valor de cada campo
    // a partir da var. de memória correspondente.
    dbSelectArea(cAlias1)
    RecLock(cAlias1, .T.)

    For i := 1 To FCount()
        IncProc()

        If "FILIAL" $ FieldName(i)
            FieldPut(i, xFilial(cAlias1))
        Else
            FieldPut(i, M->&(Eval(bCampo,i)))
        EndIf
    Next
    MSUnlock()

    // Grava os registros da tabela Filho.
    dbSelectArea(cAlias2)
    dbSetOrder(1)

    For i := 1 To Len(aCols)

        IncProc()

        If !aCols[i][Len(aHeader)+1]      // A linha não esta deletada
                                           // logo, pode gravar.
            RecLock(cAlias2, .T.)
            For y := 1 To Len(aHeader)
                FieldPut(FieldPos(Trim(aHeader[y][2])), aCols[i][y])
            Next
            MSUnlock()
        EndIf
    Next

    Return Nil

```

Figura 20.88 Exemplo de Função de Inclusão no Modelo 3.

```

////////////////////////////////////
// ALTDADOS //
////////////////////////////////////

Static Function AltDados()
    Local i      := 0
    Local y      := 0
    Local nItem  := 0
    ProcRegua(Len(aCols) + FCount())
    dbSelectArea(cAlias1)
    RecLock(cAlias1, .F.)
    For i := 1 To FCount()
        IncProc()
        If "FILIAL" $ FieldName(i)
            FieldPut(i, xFilial(cAlias1))
        Else
            FieldPut(i, M->%(fieldname(i)))
        EndIf
    Next
    MSUnlock()
    dbSelectArea(cAlias2)
    dbSetOrder(1)
    nItem := Len(aAlt) + 1
    For i := 1 To Len(aCols)
        If i <= Len(aAlt)
            dbGoTo(aAlt[i])
            RecLock(cAlias2, .F.)
            If aCols[i][Len(aHeader)+1]
                dbDelete()
            Else
                For y := 1 To Len(aHeader)
                    FieldPut(FieldPos(Trim(aHeader[y][2])), aCols[i][y])
                Next
            EndIf
            MSUnlock()
        Else
            If !aCols[i][Len(aHeader)+1]
                RecLock(cAlias2, .T.)
                For y := 1 To Len(aHeader)
                    FieldPut(FieldPos(Trim(aHeader[y][2])), aCols[i][y])
                Next
                (cAlias2)->Z2_Filial := xFilial(cAlias2)
                (cAlias2)->Z2_Numero := (cAlias1)->Z2_Numero
                (cAlias2)->Z2_Item  := StrZero(nItem, 2, 0)
                MSUnlock()
                nItem++
            EndIf
        EndIf
    Next
Return Nil

```

Figura 20.89 Exemplo de Função de Alteração no Modelo 3.

```

////////////////////////////////////
// EXCLUIDADOS //
////////////////////////////////////

Static Function ExcluiDados()

    ProcRegua(Len(aCols)+1) // +1 é por causa da exclusao do arq. de cabeçalho.
    dbSelectArea(cAlias2)
    dbSetOrder(1)
    dbSeek(xFilial(cAlias2) + (cAlias1)->Z1_Nome)

    While !EOF() .And.(cAlias2)->Z3_Filial == xFilial(cAlias2) .And.;
        (cAlias2)->Z2_Nome == (cAlias1)->Z1_Nome

        IncProc()
        RecLock(cAlias2, .F.)
        dbDelete()
        MSUnlock()
        dbSkip()

    End

    dbSelectArea(cAlias1)
    dbSetOrder(1)
    IndProc()
    RecLock(cAlias1, .F.)
    dbDelete()
    MSUnlock()

Return Nil

```

Figura 20.90 Exemplo de Função de Exclusão no Modelo 3.

A função Md3TudoOK é semelhante à função Md2TudoOK utilizada no Modelo2. Neste ponto é sugerido o desenvolvimento de uma validação nesta função para não aceitar transações repetidas dentro da GetDados.

Programas de Relatórios

Programas de consultas e relatórios normalmente podem ser feitos com geradores de código. Outra solução interessante é trazer os dados para o Excel e de lá obter os relatórios e consultas desejados. No IDE está disponível um assistente que gera um fonte básico, facilmente adaptável a um relatório específico.

O SIGARPM é um gerador de relatórios onde se informa quais arquivos e dentro destes quais campos devem ser listados. Há sempre um arquivo principal,

que define a seqüência dos registros (índices existentes ou novos) a serem listados e os demais são a ele relacionados através das chaves estrangeiras. Um campo pode ser uma expressão, desde uma simples fórmula até funções mais complexas. O relatório pode ter quebras com salto de folha, sub-totais e totais. Aceita perguntas para o processo de Filtros e segue todas as características de um relatório normal (impressão em disco, salvar a configuração para uso futuro, ser incluído no menu, etc.).

Caso se queira um tratamento especial no processo de impressão, deverá ser escrito uma função de usuário. Isto pode ser necessário para efetuar impressões que tem fórmulas que envolvem campos de diversos arquivos. Para ilustrar um exemplo bastante simples deste tipo de função, a Figura 20.91 especifica os parâmetros necessários.


```

/////////////////////////////////////////////////////////////////
// FUNÇÃO QUE CONFIGURA OS PARÂMETROS PARA A IMPRESSÃO DO EXTRATO DE
// CONTAS-CORRENTES DOS CLIENTES
/////////////////////////////////////////////////////////////////

User Function Rel001()

    Local cAlias      := ""           // Alias do arquivo a ser impresso.
    Local cNomeArq    := FunName()    // Nome do arquivo, para impressão em disco.
    Local wnRel        := ""           // Retorno da função SetPrint().
    // Descrição do relatório.
    Local cDesc1      := "Imprime o extrato das contas-correntes de todos os Clientes."
    Local cDesc2      := ""
    Local cDesc3      := ""
    Private cTitulo    := "Extrato de Contas-Correntes" // Título do relatório.
    Private cCabec1    := " Data      Tipo Mov          Valor          Saldo"
    Private cCabec2    := ""
    Private cNomeProg  := FunName()    // Nome do programa no cabeçalho do relatório.
    Private cTamanho   := "P"          // Tamanho do relatório.
    Private nTipo      := ""           // Relatório normal ou comprimido.
    Private m_Pag      := 1            // Número da página.
    // O array aReturn usado na SetPrint(). Define o formato, tipo de impressão e o nome do arquivo
    // aReturn[1] = Reservado para formulário.
    // aReturn[2] = Reservado para número de vias.
    // aReturn[3] = Destinatário.
    // aReturn[4] = Formato: 1-Retrato, 2-Paisagem.
    // aReturn[5] = Tipo mídia: 1-Disco, 2-Via spool, 3-Direto na porta, 4-Email.
    // aReturn[6] = "NomeArq"-Disco, "LPT1"-Via spool, "LPT1"-Direto na porta, "-Cancelado.
    // aReturn[7] = Expressão do filtro.
    // aReturn[8] = Ordem a ser selecionada.
    Private aReturn := {"Zebrado", 1, "Administracao", 1, 1, "CANCELADO", "", 1}
    // Função SetPrint(): prepara os parâmetros para a impressão do relatório. Parâmetros:
    // cAlias --> Arquivo a ser listado. Se passado, permite a seleção dos campos
    // a imprimir (Dicionário de Dados) e definição de um filtro.
    // cNomeArq --> Nome do arquivo a ser gerado no caso impressão em disco.
    // cPerg --> Caso seja passado, permite apresentar o conjunto de perguntas.
    // cTitulo --> Título do relatório.
    // cDesc1 --> Descrição do relatório.
    // cDesc2 --> Descrição do relatório.
    // cDesc3 --> Descrição do relatório.
    // lDic --> .T.Possibilita seleção dos campos (Dicionário) a imprimir .F. Não Possibilita.
    // aOrd --> Array com as ordens de impressão.
    // lCompres --> .T. comprimido. .F. normal.
    // cSize --> Tamanho do relatório: "P"-80 col., "M"-132 col., "G"-220 col.
    // Atribui o nome do arquivo a ser gerado em wnRel, caso a impressão seja em disco.
    wnRel := SetPrint(cAlias, cNomeArq, , @cTitulo, cDesc1, cDesc2, cDesc3, .F., .F., cTamanho)
    // Define se o relatório é normal ou comprimido.
    nTipo := IIf(aReturn[4] == 1, 15, 18)
    // Se na função SetPrint() foi clicado o botão Cancelar, o sexto elemento
    // do vetor aReturn conterá "CANCELADO".
    If aReturn[6] <> "CANCELADO" // Não cancelou o relatório.
        SetDefault(aReturn, cAlias) // Prepara a impressora ou o arquivo para o relatório.
        // Estabelece os campos a serem impressos e o filtro, caso tenham sido definidos.
        RptStatus({| | Imprime()}) // Executa a rotina de impressão do relatório.
        OurSpool(wnRel) // Envia o relatório para o spool ou exibe na tela,
        // dependendo da opção selecionada.
    Endif
    MS_Flush() // Libera a memória.
    Return Nil

```

Figura 20.91 Exemplo de Função para Atribuição de Parâmetros de impressão.

Neste exemplo, a impressão propriamente dita será realizada pela função *Imprime*, cujo fonte está ilustrado na Figura 20.92.

```

////////////////////////////////////
// ROTINA DE RELATÓRIO QUE IMPRIME O EXTRATO DE CONTAS-CORRENTES DOS CLIENTES //
////////////////////////////////////
Static Function Imprime()
  Local cNomeAnt
  Local nValor
  Local nSaldo
  cNomeAnt := ""
  nSaldo := 0
  DbSelectArea("SZ2") // Seleciona o arquivo de Movimentações.
  DbSetOrder(2)       // Seleciona a chave "Filial + Nome + Nro.Trans. + Item".
  DbGoTop()           // Vai para o primeiro registro, conforme a ordem selecionada.
  While !Eof()        // Executa os comandos enquanto não for Fim de Arquivo.
    If PRow() > 60      // Se o "cursor" da impressora ultrapassou 60 linhas,
      Eject            // salta para a pagina seguinte e
      SetPrc(0, 0)     // zera o "cursor" da impressora.
    EndIf
    // Se o "cursor" da impressora estiver no inicio de uma nova pagina
    If PRow() == 0
      // imprime o cabeçalho.
      Cabec(cTitulo, cCabec1, cCabec2, cNomeProg, cTamanho, nTipo)
    EndIf
    // Para cada Cliente, deve haver uma separação e o seu nome deve aparecer antes
    // da listagem de suas movimentações e também o seu saldo deve ser inicializado.
    // A variável cNomeAnt guarda o nome do Cliente que está sendo impresso.
    // A cada registro lido, vê se é diferente do conteúdo do campo Z2_Nome.
    // Se for diferente, significa que mudou o Cliente. Neste caso, deve-se:
    // - imprimir uma linha de separação com o Cliente anterior;
    // - imprimir o nome do novo Cliente;
    // - guardar o nome do novo Cliente na variável cNomeAnt;
    // - inicializar a variável nSaldo.
    If SZ2->Z2_Nome <> cNomeAnt // Mudou o Cliente.
      // Imprime uma linha para separar o Cliente anterior.
      @PRow()+1,000 PSay __PrtThinLine()
      @PRow()+1,000 PSay "Cliente: " + SZ2->Z2_Nome
      @PRow()+1,000 PSay " "
      cNomeAnt := SZ2->Z2_Nome // Guarda o nome do novo Cliente.
      nSaldo := 0 // Inicializa o saldo.
    EndIf
    // Imprime o Número da Transação, a Data, o Tipo de Movimentação, o
    // histórico, o Valor (positivo ou negativo), o Saldo e a aprovação.
    @PRow()+1,000 PSay SZ2->Z2_NroTra + SZ2->Z2_Item
    @PRow() ,PCol()+2 PSay SZ2->Z2_Data
    @PRow() ,PCol()+2 PSay IIf(SZ2->Z2_Tipo=="D", "Deposito", "Saque ")
    @PRow() ,PCol()+2 PSay SZ2->Z2_Hist
    nValor := SZ2->Z2_Valo * IIf(SZ2->Z2_Tipo=="D", 1, -1)
    nSaldo += nValor
    @PRow() ,PCol()+2 PSay nValor Picture "@E 999,999,999.99"
    @PRow() ,PCol()+2 PSay nSaldo Picture "@E 999,999,999.99"
    @PRow() ,PCol()+2 PSay SZ2->Z2_Aprov
    // Vai para o próximo registro, também pela ordem selecionada.
    dbSkip()
  End
  // Imprime uma linha final, após o ultimo Cliente listado.
  @PRow()+1,000 PSay __PrtThinLine()
  Return Nil

```

Figura 20.92 Exemplo de Função de Impressão.

A Integração por Meio de Arquivos

Embora os relatórios possam ser gerados em arquivos de disco e transportados para impressão em outros locais não proporcionam a utilização de seus dados para a atualização de outros sistemas. Neste caso os dados devem ser digitados novamente a partir da listagem. Para evitar este problema, os dados podem ser gravados em arquivos dentro de um formato específico para serem lidos por outros sistemas. Dentre os diversos formatos existentes vamos destacar o TXT e o XML.

Geração de Arquivos Tipo Texto

Arquivos do tipo texto (também conhecidos como padrão TXT) são arquivos com registros de tamanho variável. A indicação do final de cada registro é representada por dois bytes, 0D 0A em hexadecimal ou 13 10 em decimal ou ainda CR LF para padrão ASCII. Apesar do tamanho dos registros poder ser variável, a maioria dos sistemas geram este tipo de arquivo com registros de tamanho fixo. De acordo com um *Lay-Out* específico que indicam quais são os dados gravados.

Para ilustrar, no exemplo do Conta Corrente, será gerado um arquivo TXT que contém os dois arquivos: o SZ1 e o SZ2. Para gravar os dados dos dois arquivos, SZ1 e SZ2, dentro do mesmo TXT é utilizado o ID com 3 bytes que identifica qual o *Lay-Out* a ser utilizado, conforme ilustra a Figura 20.B.

ID (3)	Nome (20)	Saldo (11)	Space (9)	CR (1)	LF (1)
SZ1				CHR(13)	CHR(10)

ID (3)	Data (8)	Tipo (1)	Histórico (20)	Valor (11)	CR (1)	LF (1)
SZ2					CHR(13)	CHR(10)

Figura 20.B Exemplo de *Lay-Out* para Arquivos TXT.

No exemplo da Figura 20.B, para que tenham o mesmo tamanho, o *Lay-Out* do SZ1 tem um espaço de 9 bytes. Desta forma tanto o SZ1 como o SZ2 tem o mesmo tamanho: 45 bytes.

O programa ilustrado na figura 20.A mostra como é gerado o arquivo CLIENTES.TXT que contém os dados de SZ1 e SZ2.

```

/////////////////////////////////////////////////////////////////
// Geração de Arquivo TXT                                     //
/////////////////////////////////////////////////////////////////

User Function OkGeraTXT()

    Local cArq
    Local nHdl
    Local cLinha

    // Se for especificado o drive no caminho do arquivo, sera criado no
    // Client, caso contrario sera criado no Server, no diretorio RooPath.
    cArq := "\TXT\CLIENTES.TXT"
    nHdl := FCreate(cArq)

    If nHdl == -1
        MsgAlert("O arquivo " + cArq + " nao pode ser criado!", "Atencao!")
        Return
    Endif

    dbSelectArea("SZ1")
    dbSetOrder(1)
    dbGoTop()

    While !SZ1->(EOF())

        // Cliente.
        cLinha := "SZ1" + SZ1->Z1_Nome + Str(Int(SZ1->Z1_Saldo*100), 11) + ;
            Space(9) + Chr(13) + Chr(10)

        If FWrite(nHdl, cLinha, Len(cLinha)) <> Len(cLinha)
            If !MsgAlert("Ocorreu um erro na gravacao do arquivo.", "Atencao!")
                Exit
            Endif
        Endif

        dbSelectArea("SZ2")
        dbSetOrder(2)          // Filial + Nome + Numero + Item.
        dbSeek(xFilial("SZ2") + SZ1->Z1_Nome)

        While SZ2->Z2_Nome == SZ1->Z1_Nome
            // Transacoes.
            cLinha := "SZ2" + DtoC(SZ2->Z2_Data) + SZ2->Z2_Tipo + SZ2->Z2_Hist + ;
                Str(Int(SZ2->Z2_Valor*100), 11) + Chr(13) + Chr(10)

            If FWrite(nHdl, cLinha, Len(cLinha)) <> Len(cLinha)
                If !MsgAlert("Ocorreu um erro na gravacao do arquivo.", "Atencao!")
                    Exit
                Endif
            Endif

            dbSelectArea("SZ2")
            SZ2->(dbSkip())
        End

        dbSelectArea("SZ1")
        SZ1->(dbSkip())

    End

    FClose(nHdl)
    MsgInfo("Arquivo TXT gerado!")
Return Nil

```

Figura 20.A Programa Exemplo de Geração de Arquivo Padrão TXT.

Note que para a geração do arquivo TXT foram utilizadas, basicamente, as funções **FCreate**, **FWrite** e **FClose** que, respectivamente, gera o arquivo, adiciona dados e fecha o arquivo. No exemplo, o formato é estabelecido pela concatenação dos dados na variável `cLinha` a qual é utilizada na gravação dos dados.

Geração e Leitura de Arquivos XML

Os arquivos XML são mais flexíveis na formatação dos dados, pois a demarcação ocorre para cada campo, ou seja o comprimento dos campos podem ser variáveis. Outra característica dos arquivos XML é a identificação do dado no próprio arquivo. Dispensa a utilização de *Lay-Outs* para descrever os dados e possibilita a redução de manutenção dos programas em função de alteração de Lay-Out.

O fonte ilustrado na Figura 20.C demonstra como pode ser gerado um arquivo XML no AdvPL. O exemplo de um programa que lê um arquivo XML e apresenta os dados na tela é exibido na Figura 20.D.

```

/////////////////////////////////////////////////////////////////
// Geração de Arquivo XML                                     //
/////////////////////////////////////////////////////////////////

#include "PROTHEUS.CH"
#include "XMLXFUN.CH"

User Function GeraXML()

Local cEstrutura
Local nXMLStatus
Local oXML
Local cXML
Local nCli
Local nTran

// Cria a estrutura do XML.
cEstrutura := "<?xml version='1.0'?>"
cEstrutura += "<Clientes>"
cEstrutura += "    <Cliente>"
cEstrutura += "        <Nome></Nome>"
cEstrutura += "        <EMail></EMail>"
cEstrutura += "        <Saldo></Saldo>"
cEstrutura += "        <Transacao>"
cEstrutura += "            <Numero></Numero>"
cEstrutura += "            <Item></Item>"
cEstrutura += "            <Data></Data>"
cEstrutura += "            <Tipo></Tipo>"
cEstrutura += "            <Hist></Hist>"
cEstrutura += "            <Valor></Valor>"
cEstrutura += "        </Transacao>"
cEstrutura += "    </Cliente>"
cEstrutura += "</Clientes>"

```

```

// Cria o objeto XML, definindo o Cliente e a Transacao como arrays.
CREATE oXML XMLSTRING cEstrutura SETASARRAY _Clientes:_Cliente, _Clientes:_Cliente:_Transacao
nXMLStatus := XML_Error()

If nXMLStatus == X_ERROR_SUCCESS
// Se nao houver nenhum erro na criação do objeto, calcula o total de livros
// e percorre os elementos do array para criar os nodes dos livros no XML.
dbSelectArea("SZ1")
dbSetOrder(1)
dbGoTop()
nCli := 1

While !SZ1->(Eof())

    If nCli > 1
        // Apenas acrescenta nodes novos caso já tenha realizado a primeira volta
        // do loop, que irá atribuir os valores do primeiro cliente ao node já
        // existente no objeto Xml.
        ADDNODE oXML:_Clientes:_Cliente NODE "_Cliente" ON oXML
    EndIf

    // Atribui os dados do cliente ao objeto Xml.
    oXML:_Clientes:_Cliente[nCli]:_Nome:TEXT := SZ1->Z1_Nome
    oXML:_Clientes:_Cliente[nCli]:_Email:TEXT := SZ1->Z1_Email
    oXML:_Clientes:_Cliente[nCli]:_Saldo:TEXT := SZ1->Z1_Saldo
    dbSelectArea("SZ2")
    dbSetOrder(2) // Filial + Nome + Numero + Item.
    dbSeek(xFilial("SZ2") + SZ1->Z1_Nome)
    nTran := 1

    While SZ2->Z2_Nome == SZ1->Z1_Nome

        If nTran > 1
            ADDNODE oXML:_Clientes:_Cliente[nCli]:_Transacao NODE "_Transacao" ON oXML
        EndIf

        oXML:_Clientes:_Cliente[nCli]:_Transacao[nTran]:_Numero:TEXT := SZ2->Z2_Numero
        oXML:_Clientes:_Cliente[nCli]:_Transacao[nTran]:_Item:TEXT := SZ2->Z2_Item
        oXML:_Clientes:_Cliente[nCli]:_Transacao[nTran]:_Data:TEXT := SZ2->Z2_Data
        oXML:_Clientes:_Cliente[nCli]:_Transacao[nTran]:_Tipo:TEXT := SZ2->Z2_Tipo
        oXML:_Clientes:_Cliente[nCli]:_Transacao[nTran]:_Hist:TEXT := SZ2->Z2_Hist
        oXML:_Clientes:_Cliente[nCli]:_Transacao[nTran]:_Valor:TEXT := SZ2->Z2_Valor
        dbSelectArea("SZ2")
        SZ2->(dbSkip())
        nTran++

    End
    dbSelectArea("SZ1")
    SZ1->(dbSkip())
    nCli++
End

// Gera o XML para um arquivo.
SAVE oXML XMLFILE "TXT\CLIENTES.XML"
MsgInfo("XML gerado: " + LTrim(Str(nCli-1)) + " clientes")

Else

    MsgStop("Erro (" + Str(nXMLStatus,3) + ") na criação do XML.")

EndIf

Return Nil

```

Figura 20.C Programa Exemplo de Geração de Arquivo XML.

Integração com Excel

A integração com qualquer outro produto, como por exemplo o Excel, implica em poder efetuar uma conexão na qual poderão ser passados dados de alguma forma. Isto significa que os dois produtos, *Advanced Protheus* e o Excel, deverão estar sendo executados simultaneamente para a realização de tal tarefa. Para estabelecer esta conexão e chamar o Excel, a linguagem AdvPL conta com a função **APEXCEL()**. Ela prepara todo o ambiente para possibilitar a troca dos dados. Normalmente é ativada por uma opção constante no próprio menu.

Uma vez ativado o Excel por meio da **APEXCEL()**, qualquer função de usuário AdvPL pode ser chamada em qualquer célula. A única restrição é que a função chamada deverá devolver um *Array* simples ou bidimensional. Os dados contidos no *Array* são distribuídos nas células de forma posicional, ou seja, a partir da célula corrente.

Por exemplo, no Excel, uma forma para chamar qualquer função AdvPL é utilizar o comando

=MSGetArray(A1,Siga("U_PLANMOV"))

preenchido na própria célula, como por exemplo A1 e os do *Array* retornados pela função serão distribuídos nas colunas e linhas.

Outra forma é escrever uma macro, conforme ilustração da Figura 20.E, e associá-la a um botão.

```
Sub Atualiza()  
    Range("A1").FormulaR1C1 = "=MSGetArray(RC,Siga("U_PLANMOV"))"  
End Sub
```

Figura 20.E Exemplo de Macro para Preencher as Células a partir de Chamada de Funções AdvPL.

Neste caso o resultado da execução da macro é exatamente o mesmo da digitação do comando na própria célula. No entanto, pode ser necessário armazenar os dados em uma variável a ser trabalhada pela macro, como mostra o exemplo da Figura 20.F.


```

Sub Atualiza()

    aMovimentos = MSExecArray(siga("U_PLANMOV"))

End Sub

```

Figura 20.F Exemplo de Macro para Atribuir o Retorno de uma Função AdvPL à uma Variável VBA.

Para ilustrar o exemplo do Conta Corrente, a Figura 20.G e a Figura 20.H demonstram as funções que geram os dados do cadastro e da movimentação nos respectivos Arrays que preenchem as planilhas abertas na integração.

```

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Retorna Cadastro                                                                    //
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

User Function PlanCAD()

    Local aCad := {}

    dbSelectArea("SZ1")
    dbSetOrder(1)
    dbGoTop()

    While !SZ1->(Eof())

        AAdd(aCad, {SZ1->Z1_Nome, SZ1->Z1_Email, SZ1->Z1_Saldo})
        SZ1->(dbSkip())

    End

Return aCad

```

Figura 20.G Exemplo de Função de Usuário AdvPL para Retornar o Cadastro de Clientes.

```

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Retorna Movimentação                                                                //
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

User Function PlanMOV()

    Local aMovim := {}

    dbSelectArea("SZ2")
    dbSetOrder(2)                // Filial + Nome + Numero + Item
    dbGoTop()

    While !SZ2->(Eof())
        AAdd(aMovim, {SZ2->Z2_Nome, SZ2->Z2_Numero, SZ2->Z2_Item, SZ2->Z2_Data,;
                      SZ2->Z2_Tipo, SZ2->Z2_Hist, SZ2->Z2_Valor, SZ2->Z2_Aprov})
        SZ2->(dbSkip())
    End

Return aMovim

```

Figura 20.H Exemplo de Função de Usuário AdvPL para Retornar a Movimentação de Transações.

Integração Através da Internet

O uso de recursos de WorkFlow, E-Mail, Web Services e Jobs facilitam a integração entre os sistemas independente da plataforma tecnológica. Embora possam ser considerados como conceitos de integração de sistemas, aqui serão tratados como serviços que disponibilizam recursos adicionais para facilitar o trabalho de programação.

WorkFlow

O WorkFlow pode ser bastante facilitado com o uso do e-mail para a aprovação dos processos. Por exemplo, uma conta corrente com saldo negativo necessita da aprovação de um gerente. Este processo, sob o ponto de vista do sistema, pode ser chamado toda vez que ocorrer uma mudança no saldo da conta tanto por inclusão, alteração ou exclusão, conforme ilustra a Figura 20.93.

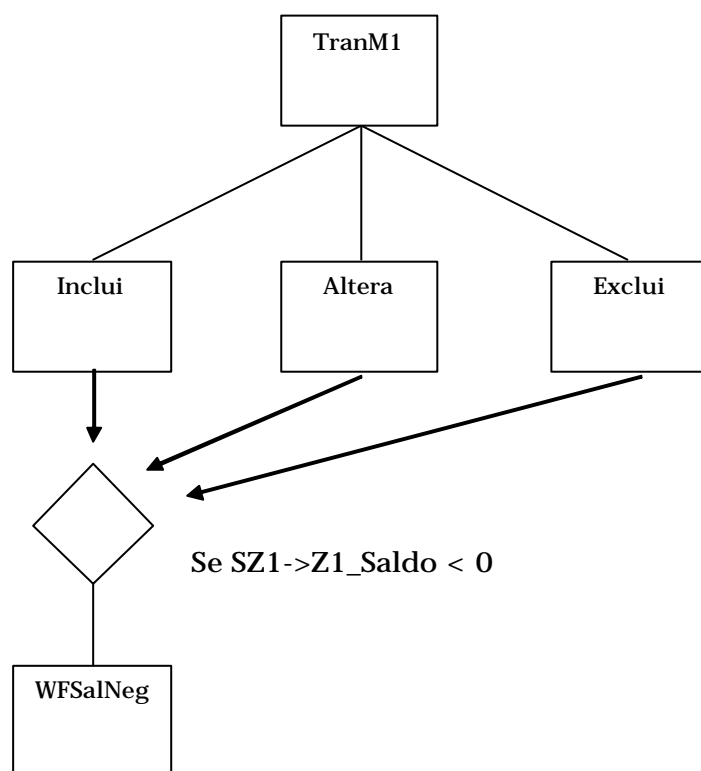


Figura 20.93 Chamada da Função WFSalNeg (Workflow de Aprovação de transação).

O exemplo da Figura 20.94 mostra o trecho a ser modificado no código fonte da função de Inclusão para chamar a função WFSalNeg.

```
User Function Inclui(cAlias, nRegistro, nOpcao)

    Local nConfirmou
    // A função AxInclui mostra a tela e inclui o registro.
    nConfirmou := AxInclui(cAlias, nRegistro, nOpcao)

    // retorna 1 se confirmou
    If nConfirmou == 1
        Begin Transaction

        // Atualiza o saldo.
        dbSelectArea("SZ1")
        dbSetOrder(1)
        dbSeek(xFilial("SZ1") + SZ2->Z2_Nome)
        RecLock("SZ1", .F.)

        If SZ2->Z2_Tipo = "D" // deposito, logo soma.
            SZ1->Z1_Saldo := SZ1->Z1_Saldo + SZ2->Z2_Valor
        Else // saque, logo subtrai.
            SZ1->Z1_Saldo := SZ1->Z1_Saldo - SZ2->Z2_Valor
        EndIf

        // Se o saldo ficou negativo, envia um WF para o aprovador e
        // a resposta será gravada no campo Z2_APROV

        If SZ1->Z1_SALDO < 0

            WFSalNeg(Z1_NOME,Z1_EMAIL,Z2_NUMERO,Z2_ITEM,Z2_DATA,Z2_HIST,Z2_VALOR,Z1_SALDO)

        Endif

        MSUnlock()

        End Transaction

    EndIf

    Return Nil
```

Figura 20.94 Chamada da Função WFSalNeg na função Inclui do Modelo 1.

Embora a chamada da função WFSalNeg seja uma conexão normal com a passagem dos respectivos parâmetros, a realização do processo de *WorkFlow* envolve dois momentos distintos: um para a geração da solicitação de aprovação da transação e outro para a gravação da resposta do aprovador, conforme os diagramas representados nas Figuras 20.95 e 20.96, respectivamente.



Figura 20.95 Caso de Uso para a Geração de Solicitação de Aprovação.

A Solicitação de Aprovação é gerada e enviada para um aprovador e pode se estender para um Reenvio, caso não seja respondida em um determinado intervalo de tempo pelo primeiro aprovador. A Figura 20.96 mostra a interação entre o usuário, os aprovadores, a atualização no banco de dados e os objetos instanciados ou mesmo criados na própria interação.

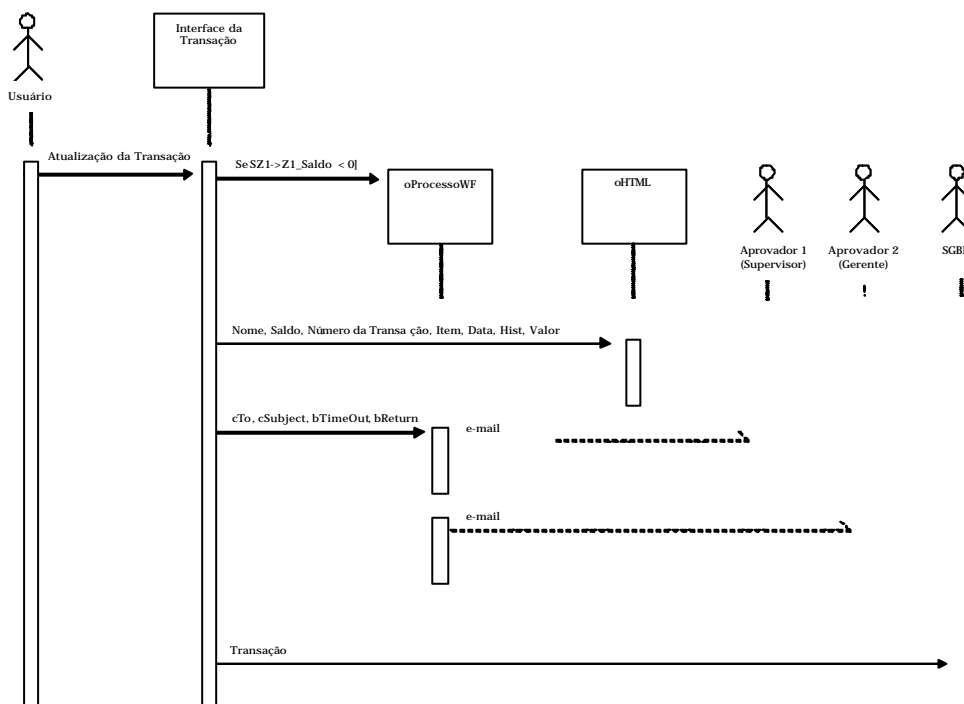


Figura 20.96 Interação da Geração da Solicitação de Aprovação e seu

Respectivo Envio.

Criação dos Arquivos HTML utilizados pelo *WorkFlow*

Para possibilitar o envio da solicitação de aprovação, a pasta `..\MP_DATA\WORKFLOW` precisa conter o arquivo HTML que compõe o corpo do e-mail enviado ao aprovador quando o saldo tornar-se negativo. Este arquivo contém, além do próprio texto da mensagem, algumas variáveis, como, por exemplo, `!NOME!` ou `!SALDO!`, destacadas entre os sinais de exclamação (!) para diferenciá-las dos demais componentes do HTML. A rotina de *WorkFlow* substituirá estas variáveis pelos dados correspondentes, de forma que a mensagem resultante conterá o nome e o saldo do cliente.

Para ilustrar o exemplo do Conta Corrente, a Figura 20.97 mostra o fonte do arquivo `WFSALNEG.htm`.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<title>Aprovação de Transação</title>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
</head>

<body>
<p><font size="2" face="Verdana, Arial, Helvetica, sans-serif">Prezado Sr(a).
<strong>!CLIENTE!</strong> !</font></p>
<p><font size="2" face="Verdana, Arial, Helvetica, sans-serif">Ap&ocirc;se o
lan&ccedil;amento abaixo, o seu saldo ficou negativo em <strong>R$ !SALDO!</strong>
.</font></p>
<font size="2" face="Verdana, Arial, Helvetica, sans-serif">Nr.
Transa&ccedil;&atilde;o: <strong>!NROTRA!</strong></font>
<br><font size="2" face="Verdana, Arial, Helvetica, sans-serif">Item:
<strong>!ITEM!</strong></font>
<br><font size="2" face="Verdana, Arial, Helvetica, sans-serif">Data:
<strong>!DATA!</strong></font>
<br><font size="2" face="Verdana, Arial, Helvetica, sans-serif">Hist&ocirc;rico:
<strong>!HIST!</strong></font>
<br><font size="2" face="Verdana, Arial, Helvetica, sans-serif">Valor:
<strong>!VALOR!</strong></font>
<form name="form1" method="post" action="mailto:%WFMailTo%">
  <p><font size="2" face="Verdana, Arial, Helvetica, sans-serif"><input type="radio"
name="%APROVA%" value="Sim">Aprova este lan&ccedil;amento</font></p>
  <p><font size="2" face="Verdana, Arial, Helvetica, sans-serif"><input type="radio"
name="%APROVA%" value="Nao">N&atilde;o aprova</font></p>
  <p><font size="2" face="Verdana, Arial, Helvetica, sans-serif"><input type="submit"
name="Submit" value="Enviar"></font> </p>
</form>
</body>
</html>
```

Figura 20.97 Fonte HTML para *WorkFlow*.

Embora este arquivo possa ter qualquer nome, sugere-se o `WFSalNeg.htm` em função das devidas amarrações que serão feitas no exercício para a realização do *WorkFlow*. Este arquivo pode ser editado por meio de qualquer ferramenta que gera fontes HTML como o Front Page da Microsoft e o Dreamweaver da Macromedia. No exemplo

do exercício, o resultado que será apresentado no e-mail está ilustrado na Figura 20.98 com as respectivas substituições para as variáveis destacadas entre os sinais de exclamação (!).

Prezado Sr(a). **!NOME!**

Após o lançamento abaixo, o seu saldo ficou negativo em **R\$!SALDO!** .

Nr. Transação: **!NUMERO!**

Item: **!ITEM!**

Data: **!DATA!**

Histórico: **!HIST!**

Valor: **!VALOR!**



Aprova este lançamento



Não aprova

Enviar

Figura 20.98 HTML para *Workflow*.

Classes de Objetos para a Criação e envio do e-mail

Tanto o arquivo HTML utilizado no *Workflow* como os respectivos dados na base são tratados por funções da linguagem AdvPL e no ambiente Protheus. Neste caso são também utilizadas algumas classes de objetos adequadas à solução deste tipo de problema.

A Figura 20.99 é um exemplo de código fonte em AdvPL para criar os objetos, atribuir os respectivos atributos, efetuar a troca de mensagens e a respectiva inicialização do envio do e-mail de acordo com a sequência descrita na Figura 20.96. No exemplo, envia-se a mensagem quando o saldo for negativo e grava-se, no retorno, a resposta do aprovador. Para simplificar o processo, o saldo é atualizado independente da aprovação.

```

////////////////////////////////////
// WFSALNEG - Workflow para Saldo Negativo                                     //
////////////////////////////////////

User Function WFSalNeg(cNome, cEmail, cNroTra, cItem, dData, cHist, nValor, nSaldo)

    Local oProcessoWF, oHtml

    // Inicializa o Objeto oProcessoWF a partir da classe TWFPProcess (WorkFlow),
    // que gera o e-mail.
    oProcessoWF := TWFPProcess():New( "INICIO", "Aprovação do Lançamento" )

    // Cria uma nova tarefa para o processo e indica o endereço do HTML que vai
    // no corpo do e-mail. O número 100200 é um número de controle de execução
    // a ser gravado e administrado pelas rotinas de WorkFlow. Eventualmente,
    // poderá ser criado um campo específico para gerenciar este número.
    oProcessoWF.NewTask( "100200", "\workflow\WFSalNeg.htm" )

    // Atribui dados ao html.
    oProcessoWF.oHtml.ValByName("CLIENTE", cNome )
    oProcessoWF.oHtml.ValByName("SALDO" , nSaldo )
    oProcessoWF.oHtml.ValByName("NROTRA" , cNroTra)
    oProcessoWF.oHtml.ValByName("ITEM" , cItem )
    oProcessoWF.oHtml.ValByName("DATA" , dData )
    oProcessoWF.oHtml.ValByName("HIST" , cHist )
    oProcessoWF.oHtml.ValByName("VALOR" , nValor )

    // Define o destinatário do WorkFlow.
    oProcessoWF.cTo := cEmail

    // Assunto da mensagem.
    oProcessoWF.cSubject := "Aprovação do Lançamento"

    // Nome da rotina e tempos limite de espera das respostas, em dias,
    // horas e minutos.
    oProcessoWF.bTimeout := {{ "U_MovTmOut", 0, 0, 10 }}

    // Função a ser executada quando a resposta chegar.
    oProcessoWF.bReturn := "U_WFMovRet"

    // Envia a mensagem e avisa o usuário.
    oProcessoWF.Start()
    MsgAlert("SALDO NEGATIVO: Enviado WorkFlow para aprovação do lançamento.")

Return Nil

```

Figura 20.99 Exemplo de Programa Fonte de *Workflow* para Aprovação de Transações com Saldo Negativo.

A ilustração da Figura 20.99 mostra também como é acionado outro aprovador

com base na propriedade **oP:bTimeOut**. Serve para descrever qual é a rotina e o intervalo de tempo que deve ser executada. No exemplo, aponta para a função de usuário MovTmOut, cujo código está ilustrado na Figura 20.100.

```

////////////////////////////////////
// MOVTMOUT - REENVIO DA MENSAGEM                                     //
////////////////////////////////////

User Function MovTmOut(oProcessoWF)

// Faz um reenvio da mensagem... Neste instante, é possível mudar o
// endereço do destinatário.
cNome := oProcessoWF:oHtml:RetByName("CLIENTE") // Recupera o nome no oHTML.
dbSelectArea("SZ1")                             // Seleciona o arquivo Z1.
If dbSeek(xFilial () + cNome)                    // Localiza o registro.
    oProcessoWF:cTO := Z1_EMAIL_1                // Atribui novo endereço de e-mail

// Concatena o texto já existente no Subject com a Identificação do Processo
// do reenvio
oProcessoWF:cSubject+="(Timeout processo: "+oProcessoWF:ProcessID()+") REENVIO: 1"

// Envia novamente a mensagem.
oProcessoWF:Start()

Endif

Return Nil

```

Figura 20.100 Exemplo de Programa Fonte depara Reenvio de Workflow para Aprovação de Transações.

No exemplo da Figura 20.100 é usado o campo Z1_EMAIL_1 do arquivo de Contas. A chave para o acesso ao registro foi resgatada no objeto oHtml pois a função MovTmOut é executada automaticamente após um período de tempo e esta variável não consta mais na memória.

Note que o e-mail do envio e o e-mail do reenvio para outro aprovador são os que estão cadastrados no arquivo de Contas apenas para simplificar a ilustração do exemplo. O ideal é que haja um arquivo especificamente de aprovadores com controles de envio de e-mails.

O aprovador responde no próprio e-mail e, após o seu envio, o Servidor Protheus disponibiliza os atributos dos objetos que tem o seu conteúdo. A Figura 20.101 mostra a seqüência de mensagens envolvidas neste processo até a atualização dos respectivos dados no Banco de Dados.

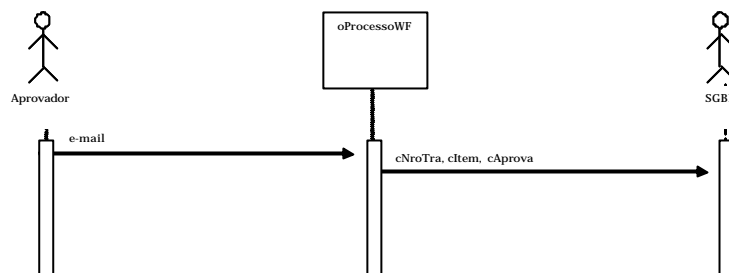


Figura 20.101 Interação da Resposta do Workflow.

No exemplo, quem realiza esta tarefa é a função de usuário WFMovRet conforme atribuição feita em `oProcesso:bReturn := u_WFMovRet`. É ela que faz a ativação do objeto `oP` assim que o e-mail for respondido pelo aprovador. A Figura 20.102 mostra o tratamento do retorno.

```

////////////////////////////////////
// WFMovRET - Workflow para Saldo Negativo
////////////////////////////////////

User Function WFMovRet(oProcesso)

    Local oHtml
    Local cNroTra
    Local cItem
    Local cAprova

    oHtml := oProcesso:oHtml
    cNroTra := oHtml:RetByName("NROTRA") // Obtém o Nr.do Lançamento.
    cItem := oHtml:RetByName("ITEM") // Obtém o Nr.do Item.
    cAprova := oHtml:RetByName("APROVA") // Obtém a resposta do Aprovador:
                                           // ("SIM" ou "NÃO").
    dbSelectArea("SZ2") // Seleciona o arquivo de Movimentos.

    If dbSeek(xFilial() + cNroTra + cItem) // Procura o lançamento.
        RecLock("SZ2") // Bloqueia o registro.
        SZ2->Z2_Aprov := cAprova // Grava a resposta do Cliente.
        MSUnlock() // Desbloqueia o registro.
    Endif

    oProcesso:Finish() // Finaliza o processo.

Return Nil
  
```

Figura 20.102 Tratamento do Retorno do Workflow.

Configurações do Protheus Server para Suportar *WorkFlow*, *WEB Services* e *AdvPL ASP*

Alguns procedimentos devem ser adotados para preparar o Protheus Server a desempenhar as tarefas necessárias para oferecer serviços de *WorkFlow* e de *WEB Services* bem como possibilitar a compilação de comandos *AdvPL ASP* juntamente com a aplicação que está sendo desenvolvida.

Configurações no Protheus Server para *WorkFlow*

Para configurar o Protheus Server para o *WorkFlow* é necessário, inicialmente, que sejam incluídas algumas chaves no arquivo *MP8SRV.ini*, conforme mostra a Figura 20.103.

```
[ONSTART]
jobs=Agenda

[Agenda]
main=WFONSTART
environment=ENVIRONMENT
```

Figura 20.103 Chaves para *WorkFlow* no *MP8SRV.ini*.

A chave *[ONSTART]* indica para o Servidor Protheus quais *jobs* devem ser ativados. No exemplo da Figura 20.103, *WFONSTART* é uma palavra reservada que inicializa todos os requisitos necessários para o processo de *WorkFlow*. O *job* *Agenda* realiza tal tarefa especificando o respectivo ambiente.

Configuração do *ArgoMail* para o *WorkFlow*

O recurso de correio eletrônico é utilizado para o envio de mensagens para o aprovador estabelecendo assim a interação do *WorkFlow*. No caso específico do exercício do Conta Corrente, estão envolvidas duas contas de e-mail:

1. do gerente, para o recebimento do aviso de saldo negativo e aprovação do lançamento, e;
2. do próprio *WorkFlow*, que deve ser uma conta exclusiva para este fim.

Numa situação real, a criação destas contas deve ser solicitada ao provedor de acesso à Internet ou ao administrador do servidor de e-mail, caso a empresa possua um. Para efeito do exercício de Conta Corrente, será utilizado o **ArgoMail**, que é um servidor de e-mail bastante simples e roda localmente. A sua configuração e criação das contas de e-mail são feitas executando-se o programa *MailServer.exe*, da pasta *..\ArgoMail* disponibilizada na instalação

do CD que acompanha o Livro.

No ArgoMail, este processo é realizado em duas etapas: Configuração do Servidor de e-mail e a Criação das Contas de e-mail, conforme a seguir.

Configuração do servidor de e-mail: menu Tools / Options

- Pasta **General**, campo **DNS Server**: localhost;
- Pasta **Local Domains**, adicionar o domínio tecnico.com.br;
- Pasta **Ports**:
 - Campo **SMTP**: 255
 - Campo **POP3**: 110
 - Campo **Finger**: 79

Criação das contas: menu Tools / Users

- São duas as contas de usuários : uma para o Aprovador e outra para o *WorkFlow* (utilizada pelo Protheus). O aprovador com senha aprovador e o usuário *workFlow* com senha *workFlow*. Os campos a serem informados são exclusivamente **User Name**, **Password** e **Confirm Password**;
- No exemplo é utilizado o endereço de e-mail do cadastro de conta para o aprovador.

Cadastramento das contas de e-mail no Outlook para o *WorkFlow*

No caso do Outlook poderá ser cadastrada apenas a conta aprovador. Esta conta deverá ser definida como padrão a qual será utilizada para simular a resposta do aprovador.

Cadastramento das Contas de e-mail no Protheus (SIGACFG) para o *WorkFlow*

A conta de e-mail criada com o nome *WorkFlow* é utilizada apenas pelo Protheus e portanto deve ser cadastrada no próprio Protheus. Este procedimento é realizado no configurador (SIGACFG) que consiste nos seguintes procedimentos:

1. Selecionar a opção **Ambiente / WorkFlow / Contas de Email** / do menu;

2. Pressionar o botão **Incluir** e preencher os campos das respectivas pastas:

- Pasta **Caixa de Correio**, campos:
 - Correio: WORKFLOW
 - Tempo Espera: 60
 - Nome: workflow
 - Endereço: workflow@tecnico.com.br
- Pasta **Receber mensagens**, campos:
 - Nome: 127.0.0.1
 - Porta: 110
 - Conta: workflow
 - Senha: workflow
- Pasta **Enviar mensagens**, campos:
 - Nome: 127.0.0.1
 - Porta: 255
 - Usuário: *não preencher*
 - Senha: *não preencher*
- Pasta **Conexão**: selecionar a opção **LAN**

Jobs

Um *Job* é um conjunto de tarefas (ou programas) que podem ser executadas a qualquer momento. Eventualmente, a execução de um *Job* pode ser realizada automaticamente pelo sistema, em períodos e frequências pré estabelecidos pela opção *Schedule* (Agendamento) disponível no Configurador. A principal característica do agendamento de *Jobs* é o fato de possibilitar a sua execução de forma autônoma, sem depender de qualquer interferência manual, e nem possuir qualquer atividade que interrompa a sua execução, como, por exemplo, a exibição de mensagens na tela. Para mostrar qualquer informação na execução, um programa executado por um *Job* deve fazer uso da função **CONOUT0**. Neste caso a mensagem é exibida no console do Seridor Protheus, tela exibida quando de sua ativação. O sistema de *WorkFlow*, por exemplo, depende de um job que, periodicamente, verifica se existe algum e-mail a ser enviado ou se chegou alguma resposta que deva ser processada.

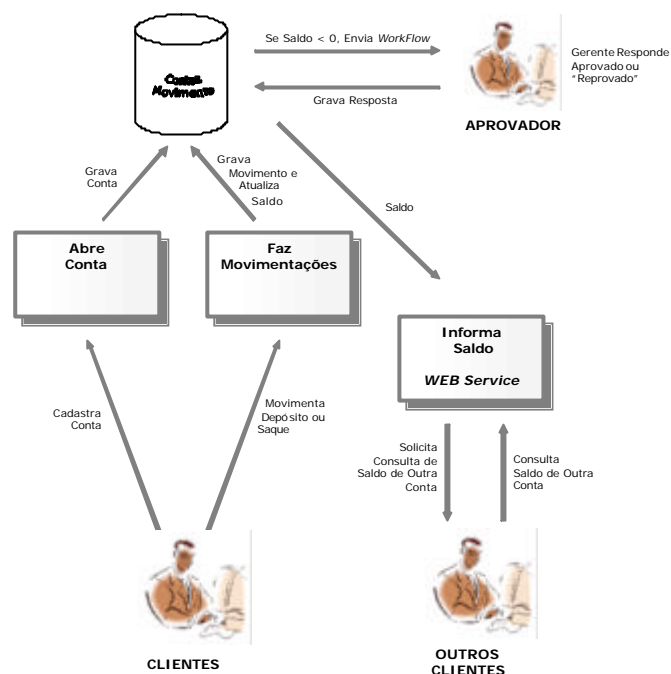


Figura 20.104 Uso de WEB Services.


Cadastramento do Job para o Workflow

O *Workflow* funciona em conjunto com um *Job* que, periodicamente, executa uma rotina a qual verifica se existe algum e-mail a ser enviado ou se recebeu alguma resposta que deva ser processada.

Para cada *Job*, devem ser cadastradas as seguintes informações:

- Código, Nome e Descrição;
- Frequência (diária, semanal ou mensal);
- Data/hora inicial e final, isto é, intervalo de tempo em que estará ativo;
- Intervalo (em horas e minutos) em que será ativado;
- Ação (nome da função a ser executada, passando a Empresa e a Filial como parâmetros);
- Ambiente (ambiente definido no .INI do Server).

As informações a respeito do *Job* também são cadastradas no configurador (SIGACFG) por meio dos seguintes procedimentos:

1. Selecionar a opção **Ambiente / Schedule / Schedule** do menu;
2. Selecionar as pastas **Schedule, Administrador, Filial01, Processos Especiais, Diário**;
3. Incluir, pressionando o botão  o *Job* que verifica a chegada da resposta do cliente, conforme ilustra a Figura 20.105.

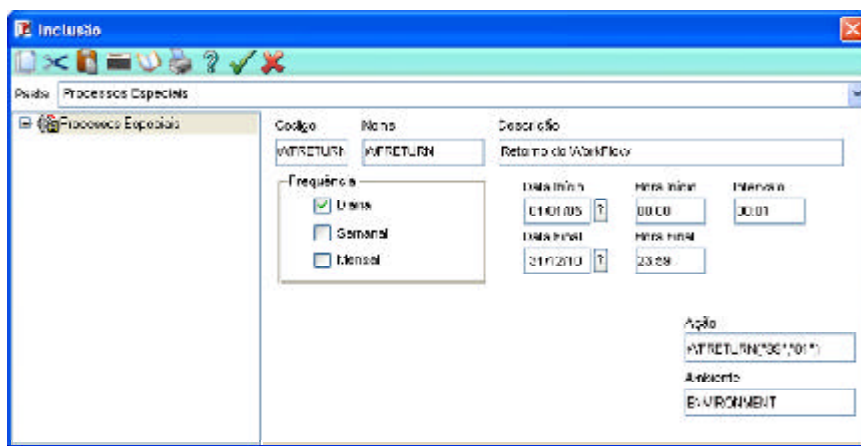


Figura 20.105 Job para Verificar as Respostas de *WorkFlow* Enviadas.

Dicas

- Concluídos estes passos, movimente a conta e gere um saldo negativo para simular o *WorkFlow* e veja como todo o processo de envio do e-mail, resposta e gravação é realizado.

WEB Services

É cada vez maior a integração e o intercâmbio de sistemas entre as empresas por meio do uso da Internet. Com o uso de *WEB Services* extrapola-se os limites da empresa, integrando-a com os Clientes, Fornecedores, Bancos, enfim o Mercado. Enquanto o *WorkFlow* integra sistemas com pessoas, *WEB Services* permitem a comunicação eletrônica entre sistemas, independente da plataforma que cada um utiliza. É o que denominamos de ECOSISTEMAS.

Tradicionalmente, sem o uso de *WEB Services*, esta comunicação é resolvida por meio de troca de arquivos entre os sistemas, normalmente padrão TXT.

Neste ponto o XML é uma excelente solução. É uma forma mais flexível de formatação de dados, pois pode-se criar um número ilimitado de campos já que apenas aqueles necessários é que são enviados, reduzindo o tráfego na rede. O tamanho do campo também é flexível, pois cada um é demarcado por 2 *tags*, ou seja, seu comprimento é variável, por exemplo:

```
<nome>Alberto Freitas</nome>
<data>20/11/2005</data>
<histórico>compra de tapete</histórico>
<valor>2.000,00</valor>
```

Qualquer sistema pode fazer uso ou disponibilizar *WEB Services*, inclusive tarifando estes serviços.

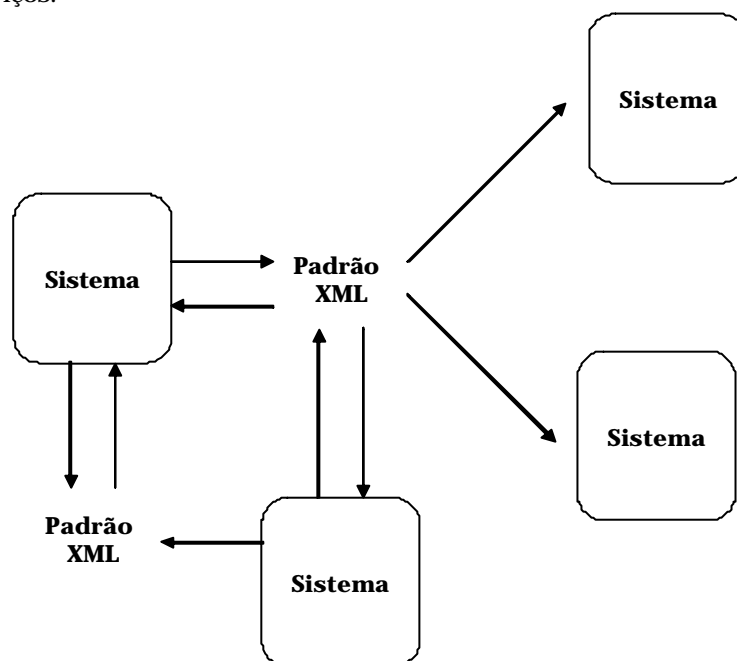


Figura 20.106 Integração com o Uso de Padrões XML.

Na verdade, *WEB Services* são rotinas embutidas no sistema e sua finalidade é disponibilizar funcionalidades ao mundo externo. O sistema passa então a oferecer algum tipo de serviço que pode ser “consumido” por outros sistemas, que farão uso deste serviço por meio de chamadas ao *WEB Service*.

Para simular este processo, o exemplo do Conta Corrente disponibiliza um *WEB Service* que fornece o saldo de uma determinada conta. São dois os programas

(prw) envolvidos: um para disponibilizar o WEB Service que fornece o saldo (WSForneceSaldo.prw) e outro para fazer uso deste WEB Service e acessar o saldo de uma determinada conta (ConsultaSaldo.prw), conforme ilustrado na Figura 20.107.

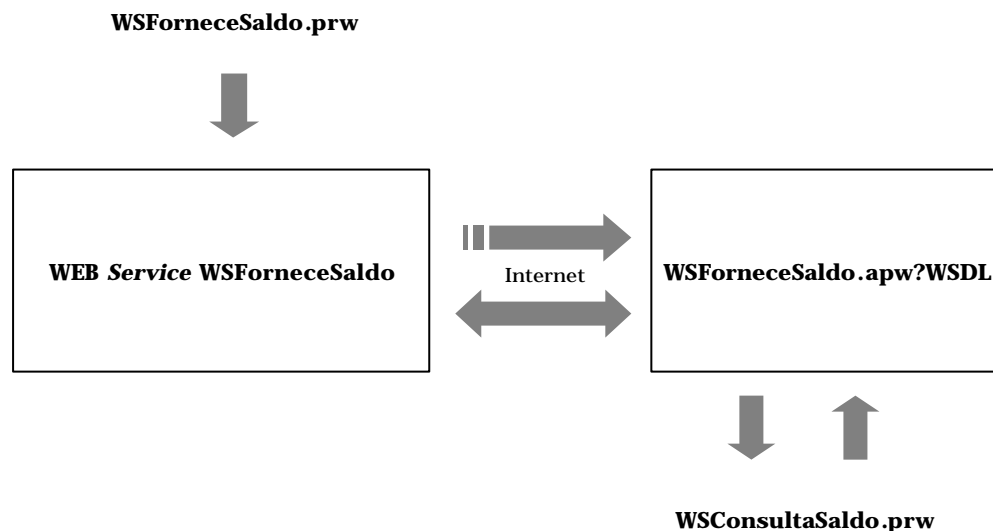


Figura 20.107 Programas de WEB Services.

No exemplo, O WEB Service WSForneceSaldo é acessado em dois momentos. Inicialmente é extraído o arquivo WSForneceSaldo.apw?WSDL o qual é compilado juntamente com programa BuscaSaldo.prw. Este processo é necessário para que o programa conheça as características do WEB Service a ser utilizado. Na próxima etapa, o programa BuscaSaldo.prw utiliza-se do WSForneceSaldo.apw?WSDL para buscar os dados disponibilizados pelo WEB Service.

O programa WSForneceSaldo.prw é compilado no Servidor Protheus que disponibilizará o WEB Service. Este programa implementa o método identificado como WSForneceSaldo conforme mostra a Figura 20.108. Na verdade, uma vez disponibilizado o respectivo WEB Service, este método poderá ser acessado por qualquer outro sistema via Internet.


```

#INCLUDE "PROTHEUS.CH"
#INCLUDE "APWEBSRV.CH"

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Informa o saldo por meio do WEB Service WSFORNECESALDO.
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

// Definição do WEB Service WSFORNECESALDO

WsService WSForneceSaldo Description "Informa o saldo do cliente."

    WsData Nome As String
    WsData Saldo As Float

    WsMethod BuscaSaldo Description "Busca o saldo."

EndWsService

// Definição do Método BuscaSaldo.
// Receber Nome e envia Saldo por meio do WEB Service WSForneceSaldo.

WsMethod WSForneceSaldo WsReceive Nome WsSend Saldo WsService WSForneceSaldo

    ::Saldo := 0

    dbSelectArea("SZ1")
    dbSetOrder(1)

    If dbSeek(xFilial() + Nome)
        ::Saldo := SZ1->Z1_Saldo
    EndIf

Return .T.

```

Figura 20.108 Programa WSForneceSaldo.prw: Exemplo para WEB Services.

O meio de comunicação utilizado é a Internet, sob o protocolo HTTP. Para que o WEB Service possa então ser encontrado, seu endereço deve ser divulgado pelo sistema que disponibiliza tal serviço. De outro lado, o sistema que faz uso deste serviço precisa ter um meio de conhecer a estrutura do WEB Service em questão. Para tanto, existe um padrão de documentação denominado WSDL – *Web Service Definition Language*, que fornece detalhes de cada WEB Service, tais como a localização (endereço URL), a denominação e o conteúdo dos campos a serem enviados e devolvidos (padrão XML).

O programa que irá invocar, ou seja acessar o WEB Service, fará uso do WSDL como uma espécie de ponte para alcançá-lo e também para passar e receber os dados.

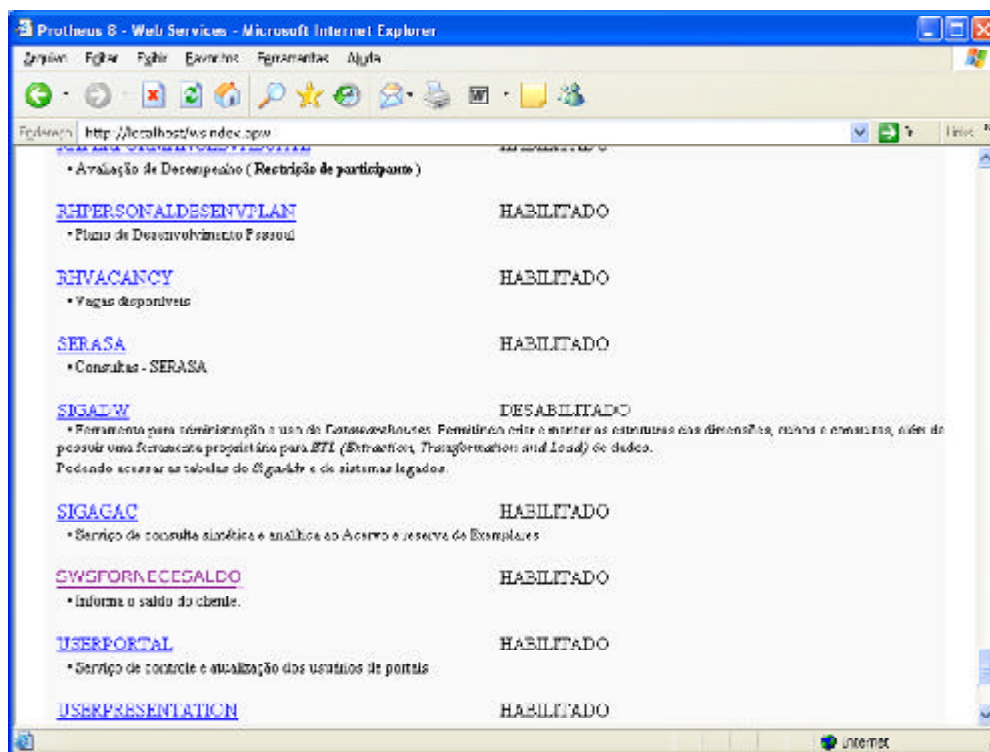


Figura 20.109 WEB Services Disponibilizados pelo Servidor Protheus.

Por exemplo, as informações sobre os WEB Services disponibilizados pelo Servidor Protheus podem ser acessadas via *browser* no arquivo WSINDEX.apw (no *localhost* caso o servidor seja a mesma máquina utilizada pela estação), conforme mostra a Figura 20.109.

Maiores detalhes obtidos com um clique no respectivo WEB Service mostram o endereço e o nome do arquivo que contém o WSDL. No exemplo do contas correntes este arquivo foi gerado com o nome TECSALDO.apw?WSDL, conforme mostra a Figura 20.110.

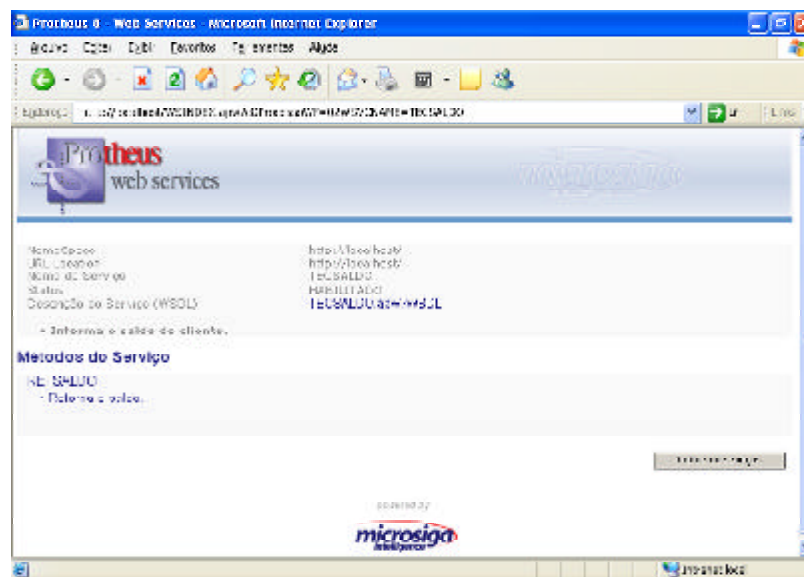


Figura 20.110 Informações Disponibilizadas sobre os WEB Services.

Efetuando mais um clique na respectiva opção `TECSALDO.apw?WSDL` é exibido o fonte do respectivo arquivo, conforme ilustra a Figura 20.111.

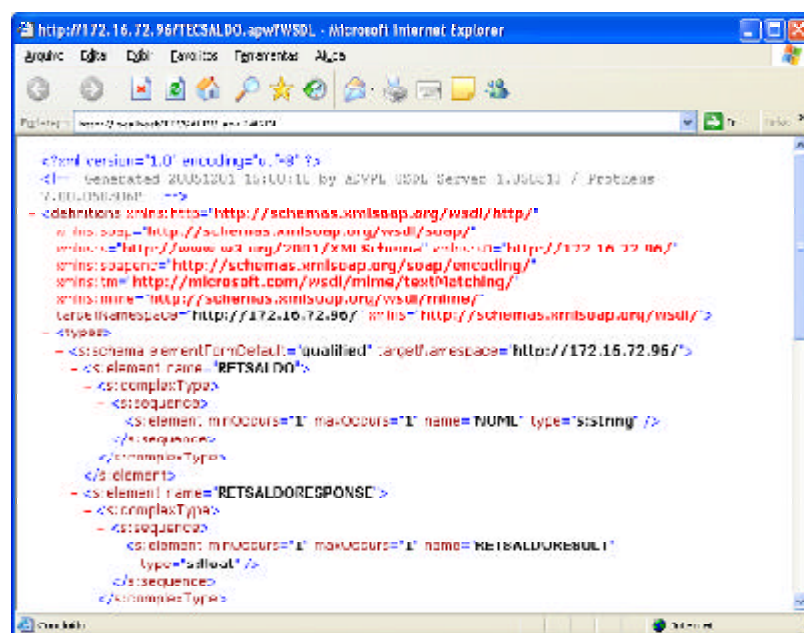


Figura 20.111 Exemplo de Fonte WSDL.

Embora seja exibido o fonte quando selecionado o `TECSALDO.apw?WSDL`, será necessário apenas copiar o endereço `http://localhost/TECSALDO.apw?WSDL`. Este deve ser informado como endereço URL de WEB Service na opção “Gerar Cliente WebServices”, no menu “Ferramentas” do IDE. O objetivo desta opção é buscar as características do WEB Service disponibilizadas no Servidor Protheus que no caso do exemplo do Contas Correntes é o próprio *localhost*. Se for um serviço oferecido por exemplo pelo SERASA, esta disponibilidade estaria no servidor do próprio SERASA. Neste caso, poderia ser acessado pela Internet informando o respectivo endereço.

Fornecendo o endereço URL do WEB Service nesta opção, o Servidor Protheus localiza-o, extrai os seus detalhes e gera o código-fonte do WSDL. Este código fonte deve ser compilado juntamente com a rotina que invoca o respectivo WEB Service. Obviamente, é necessário haver uma conexão à Internet tanto para a geração do WSDL como para invocar o próprio WEB Service.

No exemplo de WEB Services do Conta Corrente, a empresa fornecedora disponibiliza em seu sistema um WEB Service cuja função é ler e fornecer o saldo de uma determinada conta na base de dados. Na empresa solicitante, a cada solicitação de um saldo, o sistema invoca o WEB Service, passando-lhe o nome do respectivo cliente. O WEB Service captura esses dados ao receber a chamada e dispara o processo de busca e devolução do saldo, completando a automatização de todo o processo, desde a origem até o destino, conforme ilustração da Figura 20.111.

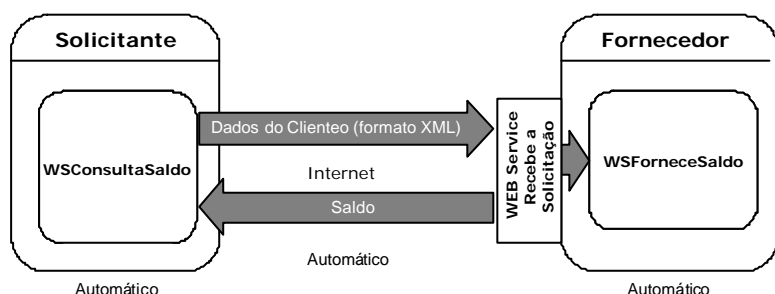


Figura 20.112 Uso de WEB Services.

No exemplo do Conta Corrente, o programa que efetua esta consulta ao saldo em outra Conta de outro Servidor é o `WSConsultaSaldo.prw`. É ele que formata a tela de consulta e chama a função `Saldo` para invocar o WEB Service.

No exemplo ilustrado na Figura 20.113, este programa solicita o endereço IP do Servidor para que, didaticamente, em um laboratório, seja fornecido apenas um WSDL porém acessado em qualquer outra máquina.

```

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Consulta Saldo de outras contas
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

#include "FILEIO.CH"

// Função que solicita do usuário o Nome e o IP do Servidor e mostra o saldo.

User Function WSConsultaSaldo()

    Local oDlg
    Local oBtnOk
    Local oBtnCancel
    Local cNome
    Local oSaldo, nSaldo

    Private cServer := Space(15)

    cNome := Space(10)
    nSaldo := 0

    Define MSDialog oDlg Title "Consulta de Saldo" From 0,0 To 220,220 Pixel

    // Solicita a digitação do Nome.
    @010,10 Say "Nome:" Pixel Of oDlg
    @008,50 Get cNome Size 50,10 Picture "@" Pixel Of oDlg

    // Solicita a digitação do IP do Servidor.
    @030,10 Say "IP do Servidor:" Pixel Of oDlg
    @028,50 Get cServer Size 50,10 Picture "@9" Pixel Of oDlg

    // Cria o objeto oSaldo com a variável nSaldo do tipo GET.
    // O parâmetro When .F. faz com que este objeto não aceite digitação,
    // apenas apresenta o campo.
    @050,10 Say "Saldo:" Pixel Of oDlg
    @048,50 Get oSaldo Var nSaldo Picture "@E 999,999,999.99" ;
        Size 50,10 Pixel When .F. Of oDlg

    // Cria dois botões (OK e CANCELAR) deslocados para cima e para a esquerda.
    // a partir do tamanho da juanela e 42 pixels.
    @oDlg:nHeight/2-37,oDlg:nClientWidth/2-77 Button oBtnOk ;
        Prompt "&Ok" ;
        Size 30,13 Pixel ;
        Action Saldo(cNome, oSaldo, @nSaldo) ;
        Of oDlg
    @oDlg:nHeight/2-37,oDlg:nClientWidth/2-42 Button oBtnCancel ;
        Prompt "&Cancelar" ;
        Size 30,13 Pixel ;
        Action oDlg:End() Cancel ;
        Of oDlg

    // Ativa a janela centralizando-a
    Activate MSDialog oDlg Centered

Return Nil

```

Figura 20.113 Programa WSConsultaSaldo: Exemplo para WEB Services.

Note que é possível fazer uso do método `WSForneceSaldo` porque ele foi trazido com todas as características no arquivo WSDL e compilado juntamente com os programas e funções que as utilizam. A função `Saldo`, ilustrada na Figura 20.114, é que faz uso deste método.

```

/////////////////////////////////////////////////////////////////
// Consulta Saldo de outras contas                                     //
/////////////////////////////////////////////////////////////////

#include "FILEIO.CH"

// Função que invoca o WEB Service WSForneceSaldo.

Static Function Saldo(cNome, oSaldo, nSaldo)

    // Cria o objeto a partir do WEB Service WSForneceSaldo.
    Local oObj := WSForneceSaldo():New()

    Local nSaldo := 0

    // Executa o método BuscaSaldo do WEB Service WSForneceSaldo.
    If oObj:WSForneceSaldo(cNome)

        // Atribui o resultado para nSaldo.
        nSaldo := oObj:nBuscaSaldoRESULT

    Else

        // WEB Service não disponível, erro.
        MsgStop("WSForneceSaldo: Impossível pegar o Serviço!")

    EndIf

    oSaldo:SetText(nSaldo) // Joga o conteúdo para o objeto oSaldo.
    oSaldo:Refresh()       // Exibe na tela o saldo por meio do objeto oSaldo.

Return .T.

```

Figura 20.114 Programa Exemplo para Invocar um WEB Services.

Configurações no Protheus Server para WEB Services

Para a configuração do Protheus Server para WEB Services execute o Assistente de Configuração do Protheus (programa `MP8WIZARD.EXE`, da pasta `..\BIN\REMOTE`).

Configuração do módulo de Web Services:

- Selecionar “Módulos Web” e clicar no botão “Novo Módulo”;
 - . Módulo Web: WS-Protheus 8 Web Services
 - . Nome da Instância: WS
 - . Diretório raiz das imagens: \web\WS
 - . Selecione o Environment: ENVIRONMENT
- Clicar no botão “Avançar”;
 - . Host: localhost
 - . Selecione a Empresa/Filial: selecionar 9901 – Teste/Matriz e clicar no botão “Relacionar”
- Clicar no botão “Avançar”;
 - . Mínimo usuários: 1
 - . Máximo usuários: 10
- Clicar no botão “Finalizar” e confirmar a operação.

Os web services são executados via protocolo HTTP-Internet, portanto, o Protheus deverá ser configurado como Servidor Internet.

Configuração do Servidor Internet:

- Selecionar “Servidor Internet (HTTP/FTP)”;
- Selecionar “HTTP” e clicar no botão “Editar Configuração”;
 - . Protocolo Habilitado: Ativar
 - . Ambiente: ENVIRONMENT
 - . Processo de resposta: JOB_WS_9901
 - . Clicar no botão “Finalizar” e confirmar a operação.

AdvPL ASP

Uma outra forma de apresentar dados é através de um *Site*, também chamado de Portal, *Home Page* ou Página. Vale dizer ainda que muitas tarefas que poderiam ser feitas via *Workflow* ou *WEB Services* são feitas, talvez pela disseminação da Internet, via Portal.

De qualquer forma é notório que a cada dia os Portais passam a ser muito mais uma ferramenta de comunicação de um sistema com seus usuários do que uma simples coleção de textos e imagens de divulgação de uma empresa. Em suma, interação mais dinâmica com os usuários conforme mostra a Figura 20.115.

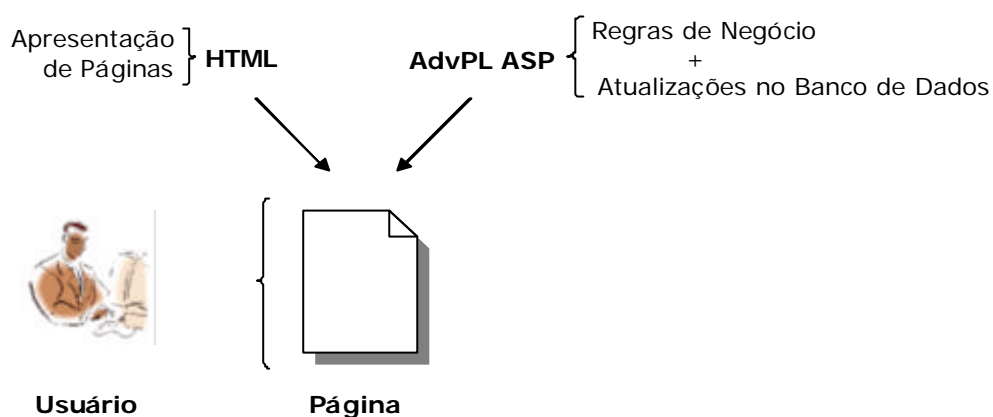


Figura 20.115 Páginas HTML com AdvPL ASP.

Para tal, no entanto, o HTML como linguagem de desenvolvimento não é suficiente. É preciso mesclá-lo com um código mais potente e flexível. No caso do AdvPl, os comandos podem ser “inseridos” no meio do HTML usando-se como tags de início e fim o caracter %, conforme o fonte da Figura 20.116.


```

<html>
<head>
<title>AdvPL/ASP</title>
<body>

<p>Data: <%=HttpSession->dData%></p>
<p>Hora: <%=HttpSession->cHora%></p>
<p></p>

<!--
  Exercício 1:
    Faça um loop para mostrar os dias da semana.
  Exercício 2:
    No dia da semana de dData, mostre '<===== Hoje'.
-->

<!--<p> <%=HttpSession->i%> </p>-->

</body>
</html>

```

Figura 20.116 HTML com AdvPL ASP.

A Figura 20.116 mostra como são incluídos comandos especiais em um fonte HTML. No caso do exemplo ilustrado na figura, **HttpSession** é uma palavra reservada do que cria uma sessão para a atribuição de variáveis. Portanto HttpSession->dData e HttpSession->cHora são variáveis de data e hora que foram atribuídas no AdvPL.

O objetivo destes comandos especiais, chamados de AdvPL ASP é exatamente o de tornar as páginas mais dinâmicas, acessando e mostrando dados e respondendo questões colocadas pelos usuários, ou seja, pode-se incluir em uma Página uma rotina completa de busca, formatação ou mesmo atualização em um Banco de Dados. Neste contexto, destacam-se os comandos para receber o dado da página (httpget e httppost) e o que coloca na tela um dado novo (httpsession)

No Protheus a montagem da página é feita a partir de um fonte, que após compilado recebe o sufixo .apw, ou seja, um link de uma página qualquer ou a chamada no Browser evoca um .apw. Este programa é que chama o .aph (advanced protheus html), como mostra o exemplo:

```

WEB EXTENDED INIT cHtml

cHtml += ExecInPage("ASP1")

WEB EXTENDED END

```

Por outro lado o .aph é o script escrito em html em conjunto com comandos AdvPL tagueados pelo <% comandos %>

Caso o leitor desconheça HTML, leia o anexo xx que contem uma rápida descrição desta linguagem.

Assim em nossos exemplos, o Asp1 (Figura 20.117) é uma simples mostra de uma página, sem nenhum tratamento em AdvPL, mas mesmo assim há necessidade de se escrever o Asp1.prw (Figura 20.118), que é quem chama o Asp1.aph.

```
<html>
<head>
<title>AdvPL/ASP</title>
<body>
Exemplo de pagina em AdvPL/ASP.
</body>
</html>
```

Figura 20.117 Exemplo de Fonte HTML com AdvPL ASP.

```
////////////////////////////////////
// Página simples. (http://localhost/pp/u ASP1.apw) //
////////////////////////////////////

#include "APWEBEX.CH"

User Function ASP1()

    Local cHtml := ""

    WEB EXTENDED INIT cHtml

    cHtml += ExecInPage("ASP1")

    WEB EXTENDED END

Return cHtml
```

Figura 20.118 Exemplo de Fonte AdvPL ASP.

Já no segundo exemplo (Asp2 ilustrado na Figura 20.119 e 20.120), há a captura de um dado e o respectivo botão de **Submit** que leva o controle para o Asp2.apw. Este por sua vez mostra o dado na console do Server e retorna o controle para o .aph. Note que neste caso o .aph ainda não se utiliza de comandos AdvPL, apenas recursos

da própria linguagem HTML (comando Input). Veja como o texto foi capturado no .apw através do comando **httppost**.

```
<html>
<head>
<title>AdvPL/ASP</title>
<body>

Exemplo AdvPL/ASP - metodo POST

<form name="form1" action="u_ASP2.apw" method="post">
Campo 1: <input type="text" name="Camp01"><br>
<input type="submit" value="Enviar">
</form>

</body>
</html>
```

Figura 20.119 Exemplo de Fonte HTML com AdvPL ASP com Captura de Dados.

```
////////////////////////////////////
// Metodo POST.
////////////////////////////////////

#include "APWEBEX.CH"

User Function ASP2()

    Local cHtml := ""

    WEB EXTENDED INIT cHtml

    If !Empty(HttpPost->Camp01)
        ConOut(HttpPost->Camp01)
    Endif

    cHtml += ExecInPage("ASP2")

    WEB EXTENDED END

Return cHtml
```

Figura 20.120 Exemplo de Fonte AdvPL ASP.

O terceiro exemplo se diferencia pela forma de como são passados os dados da página para o programa. Neste caso os dados são passados, via link (**href**), juntamente com o endereço (que no caso do Protheus é o nome do programa .apw com o caminho completo) e são dados definidos pelo programa. A sintaxe deste envio começa com o caráter ? seguido dos nomes do campo e conteúdo. No apw, por sua vez os dados são capturados com o comando **httpget**.

```
<html>
<head>
<title>AdvPL/ASP</title>
<body>

<p>Exemplo AdvPL/ASP - metodo GET</p>
<p><a href="u_ASP3.apw?Campo1=Teste_do_metodo_GET">Teste metodo GET</a></p>

</body>
</html>
```

Figura 20.121 Exemplo de Fonte HTML com AdvPL ASP usando href e httpget.

```
////////////////////////////////////
// Metodo GET.
////////////////////////////////////

#include "APWEBEX.CH"

User Function ASP3()

    Local cHtml := ""

    WEB EXTENDED INIT cHtml

    If !Empty(HttpGet->Campo1)
        ConOut(HttpGet->Campo1)
    Endif

    cHtml += ExecInPage("ASP3")

    WEB EXTENDED END

Return cHtml
```

Figura 20.122 Exemplo de Fonte AdvPL ASP com href e httpger.

No quarto exemplo é usado o comando **httpSession** que cria no apw uma variável similar a uma variável pública ou seja ela pode ser explorada no apw ou outro apw.

Inicialmente são criadas 2 variaveis com a data e hora atual e em seguida elas são mostradas na página.

O quinto exemplo mostra a atualização da Conta Corrente via web. Inicialmente o Asp5.apw chama o Asp5.aph. Este por sua vez monta a página e pede o nome, o tipo, a data, o histórico e o valor da transação de forma análoga ao TranM2. Após a digitação dos dados, um botão do tipo **Submit** pode ser utilizado para evocar o *Action do Form* que chama justamente a função Asp5Grava, que está no asp5.apw e que faz o acesso ao SZ1, atualiza o saldo e grava a transação no SZ2, conforme a ilustração dos respectivos fontes na Figura 20.125 e na Figura 20.126.

```
<html>
<head>
<title>AdvPL/ASP</title>
<body>

Transação de Depósito ou Saque

<form name="form1" action="u_ASP5Grava.apw" method="post">
  <p>Nome: <input name="Nome" type="text" id="Nome"></p>
  <p>Data (dd/mm/aa): <input name="Data" type="text" id="Data"></p>
  <p>Tipo:</p>
  <p><input type="radio" name="Tipo" value="D"> Depósito</p>
  <p><input type="radio" name="Tipo" value="S"> Saque</p>
  <p>Histórico: <input name="Hist" type="text" id="Hist"></p>
  <p>Valor: <input name="Valor" type="text" id="Valor"></p>
  <p><br><input type="submit" value="Enviar"></p>
</form>

</body>
</html>
```

Figura 20.125 Exemplo de Fonte Html com *Actcion Form*.

```

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Pagina para entrada de dados: Transação de Depósito/Saque.
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

#include "APWEBEX.CH"
#include "TbiConn.ch"

User Function ASP5()

    Local cHtml := ""

    WEB EXTENDED INIT cHtml

    cHtml += ExecInPage("ASP5")

    WEB EXTENDED END

Return cHtml

// Função para a gravação dos dados
User Function ASP5Grava()

    Local cHtml := ""

    WEB EXTENDED INIT cHtml

    Prepare Environment Empresa "99" Filial "01" Modulo "ESP" Tables "SZ1", "SZ2"

    dbSelectArea("SZ2")
    RecLock("SZ2", .T.)
    SZ2->Z2_Filial := xFilial("SZ2")
    SZ2->Z2_Nome := HttpPost->Nome
    SZ2->Z2_NroTra := GetSXENum("SZ2", "Z2_NROTRA")
    SZ2->Z2_Item := "01"
    SZ2->Z2_Data := CtoD(HttpPost->Data)
    SZ2->Z2_Tipo := HttpPost->Tipo
    SZ2->Z2_Hist := HttpPost->Hist
    SZ2->Z2_Valor := Val(HttpPost->Valor)
    MSUnlock()

    // Atualiza a sequência de numeração.
    ConfirmsX8()

    // Atualiza o saldo.
    dbSelectArea("SZ1")
    dbSetOrder(1)
    dbSeek(xFilial("SZ1") + SZ2->Z2_Nome)
    RecLock("SZ1", .F.)

    If SZ2->Z2_Tipo = "D"
        SZ1->Z1_Saldo := SZ1->Z1_Saldo + SZ2->Z2_Valor
    Else
        SZ1->Z1_Saldo := SZ1->Z1_Saldo - SZ2->Z2_Valor
    EndIf

    MSUnlock()
    cHtml := "Transação gravada com sucesso!"

    WEB EXTENDED END

Return cHtml

```

Figura 20.126 Exemplo de Fonte AdvPL ASP para a Gravação de Dados.

A partir destes exemplos fica fácil perceber que qualquer processo de digitação e consulta em um sistema para ser considerado web tem que ser feito em cima do Html pois é esta a linguagem entendida pelos Browsers. Note que o processamento da página é feito no Servidor Internet do Protheus, ou seja, páginas escritas desta forma precisam estar hospedadas em servidores Protheus.