Desenvolvimento

Boas Práticas

- Rotinas devem ter cabeçalho no formato Protheus.doc, exemplo:

```
/*/ ------------------------------------------------------------------
/*/{Protheus.doc} AreaTrap()
Rotina para calcular a área do trapézio.
@param nBase        Medida do lado ou da base
@param nAltura      Medida da altura
@param nBaseMenor   Medida da base menor
@return nArea       Área calculada
@author  Luciano Santos
@since   20/01/2020
@version P12
/*/ ------------------------------------------------------------------
```

  Mais informações: https://tdn.totvs.com/display/tec/ProtheusDOC

- Comentário nos códigos-fontes. Além do cabeçalho das rotinas, adições de comentários de regra de negócio são sempre bem-vindas. Não poluir o código-fonte com comentários óbvios como descrever um comando da linguagem.

- Não deixar trechos de código comentado, principalmente em novas codificações. Nós utilizamos controle de versão, não é necessário deixar linhas ou até blocos inteiros de códigos comentados.

- Sempre fazer identação do código, em rotinas, em laços de repetição, blocos de condição e comandos de querys. Isso facilita a leitura, entendimento e manutenção do código-fonte. O código deve estar visível na tela, sem precisar usar a barra horizontal para visualizar o código. Quando necessário deve ser feita a quebra de linha ‘**;**’

- Não fazer funções extensas. É comum ver funções Advpl com mais de 500 linhas de código. Certamente essas funções poderiam ser divididas em sub-rotinas menores para realizar operações secundárias, facilitando a manutenção do código-fonte. Por exemplo: separar as validações e a carga de dados da montagem visual de uma tela, ou em uma rotina de processamento dividir o processo em rotinas menores.

- Proteger área nas rotinas que manipulam base de dados com GetArea() e RestArea().
- Sempre fazer ponteiramento em rotinas de acesso a banco de dados. Exemplo:

  `SE2->(MsUnlock()), SA2->(DbSkip())` etc.

- Não fazer chamada de API em laço. Fazer chamada de um parâmetro usando GetMV() ou SuperGetMV() no laço quando o parâmetro é compartilhado ou não há mudança de filial no laço da função representa custo desnecessário para a rotina. Usar funções como TamSx3() dentro de um laço tendo como parâmetro sempre o mesmo campo também deve ser evitado. Em resumo, todo código que não está condicionado a mudanças de variáveis deve ficar fora do laço.
- Utilizar macro execução em último caso. Macro execuções sempre tem um custo maior para o processamento que a execução de um código explicito, principalmente quando é possível contorná-las. Nesse exemplo é perfeitamente possível trocar a macro exceção da gravação de um campo por FieldPut:

  Com macro execução: `(Tabela)->&(cCampo) := xCoteudo`

  Trocando por FieldPut: `(Tabela)->(FieldPut(FieldPos(cCampo), xCoteudo))`

- Uso correto de escopo de Funções. Somente utilize user function se a rotina tiver chamada externa, por exemplo, chamada de menu, job de um serviço ou um schedule. Para as outras sub-rotinas, utilizar Static function.
- Colocar um User Function *Dummy* em fontes somente com classes. Fontes somente com classes não conseguem ser visualizadas no inspetor de objetos. É necessário adicionar uma função Dummy como:

```
/*/ ------------------------------------------------------------------
/*/{Protheus.doc} __Produt()
Função Dummy para a classe Produto.

@return  Nil

@author  Luciano Santos
@since   20/01/2020
@version P12
/*/ ------------------------------------------------------------------

User Function __Produt()
Return Nil
```

- Uso correto de escopo de variáveis Globais, Estáticas e Privada. Não será permitido o uso de variáveis globais e na codificação por ter um escopo muito abrangente envolvendo risco para o sistema. Também não será permitido o uso de variáveis estática, apesar de terem um escopo mais restrito, é usado abusivamente por ser cômodo não precisar passar a variável por parâmetro. O emprego de variáveis do tipo *Private* terá que ser justificado, pois o seu uso pode ser facilmente contornado com variáveis locais passadas por referência. GMUD's com códigos-fonte utilizando variáveis globais, estáticas e demasiada ocorrência de variáveis do tipo *Private* serão devolvidas. Ao utilizar corretamente o escopo das variáveis, torna-se desnecessário o uso de *under-line* no nome da variável.

- Usar notação húngara. Notação húngara é fundamental para linguagens não tipadas como o ADVPL, ajudam na compreensão em uma futura manutenção do código-fonte.

- Não utilizar nomenclatura de variáveis com mais de 10 caracteres. As variáveis no ADVPL têm o nome limitado a 10 caracteres válidos. Se em uma rotina houver uma variável com o nome **nvalorsaldoini** e outra **nvalorsaldofin**, na compilação o sistema só considera os 10 primeiros caracteres. Desse modo, as duas variáveis teriam o mesmo conteúdo, pois para o código seria a mesma variável com nome de **nvalorsald**.

- Usar nomes inteligíveis e capital para nome de variáveis. Às vezes é difícil expressar a ideia do conteúdo da variável com apenas 9 dígitos, sendo 1 utilizado pela notação húngara, mas podemos usar abreviações, remover vogais e utilizar capital para facilitar a leitura. Usando o exemplo do item anterior:

  Sem uso de capital: **nvalorsaldofin**

  Com uso de capital e removendo vogais: **nVlrSldFin**

- Não utilizar nomenclatura de user functions com mais de 8 caracteres. Por limitação da linguagem, o mesmo problema das variáveis em ADVPL ocorre nas user fuctions, sendo que dois caracteres são designados pela TOTVS para especificar a função de usuário '**U_**'. Se tivermos as funções UserFuncIni, UserFuncMed e UserFuncFim, na compilação o sistema só considera os 8 primeiros caracteres. Desse modo, a função chamada seria a última a ser encontrada no APO e todas teriam o nome de **UserFunc**.

- Não utilizar nomenclatura de static functions com mais de 10 caracteres. Apesar do escopo desse tipo de função ser restrita ao arquivo do código-fonte, o mesmo problema

ainda pode ocorrer. Utilizar o mesmo parâmetro para construção de classes e métodos em Advpl.

- Usar nomes inteligíveis e capital para nome de funções e classes. Assim como nas variáveis, podemos usar abreviações, remover vogais e utilizar capital para facilitar a leitura em funções, classes e métodos.

- Não utilizar Return no meio da rotina. É boa pratica utilizar controles lógicos de execução da rotina no ligar de interromper no meio da execução, principalmente se o código é muito longo. Exemplo:

Não utilizar `Return`:

```
If !lCondicao
   Return
EndIf
```

Utilizar o controle lógico:

```
If !lCondicao
   lRet := .F.
EndIf

If lRet
   (Codificação)...
EndIf
```

- Não será permitido acesso direto a qualquer tabela do dicionário de dados. Exemplo: SX3->X3_CAMPO. Consulte o TDN, existem várias funções disponibilizadas pela TOTVS para evitar o acesso direto ao dicionário de dados. GMUD's com esse tipo de codificação serão devolvidas.

- Validações de um registro devem estar sempre no campo X3_VLDUSER. As validações feitas no campo de validação do produto padrão (X3_VALID) podem ser perdidas ao fazer atualizações do sistema.

- Campos customizados devem ter por padrão o X depois o *under-line*, exemplo: E1_**X**DOCBON.

Configurações customizadas devem ter os campos referentes a proprietário marcados com '**U**', exemplo: X3_PROPRI, X6_PROPRI, X2_PROPRI etc.

Cuidados na solicitação de inclusão de índices. Não serão aceitos índices sem o campo de filial, índices repetidos (índice igual ao início de um índice maior que já tem na base). Serão avalizados índices em tabelas com grande volume de dados ou que já tenham uma grande quantidade de índices criados. Pode ser estudada a possibilidade de criar o índice apenas no banco de dados para melhorar performance.

- Uso racional de campos do tipo memo. Justificar a criação de campos do tipo memo, principalmente em tabelas com grande movimento. Observar a real necessidade de trazê-los em Browser, pois campos desse tipo costumam deixar a abertura de telas muito lenta.

- Utilizar a forma correta para trazer a informação de um campo do tipo memo. É comum ver algumas *querys* com uma função específica do banco de dados para trazer a informação de campos. Essas funções geralmente convertem um campo do binário para caractere, entretanto, além de deixar a extração mais lenta, existe uma limitação que dependendo do banco de dados pode chegar a 4 mil caracteres. Utilize os recursos da linguagem para posicionar no registro e trazer a informação.

- Não utilizar funções obsoletas. Algumas funções de manipulação de formulário como as rotinas de interface AXCadastro(), AXAltera(), AXdetelta(), ou de montagem de browser usando a rotina NewGetDados() foram substituídas pelo MVC. A rotina CriaTrab() foi substituída FWTemporaryTable(). A função DbSetFilter() foi descontinuada por ser extremamente lenta. Verifique se a função foi descontinuada antes de utilizar na codificação.

Importante: Para identificar no banco de dados tabelas criadas com FWTemporaryTable(), usar como nome da tabela "TEMP"+ GetNextAlias()

- Garantir posicionamento de registro. Posicionar em um registro e seguir com o processo sem validar o posicionamento não é uma boa prática. O registro pode não ter sido localizado, e o processo seguir com as alterações em um registro errado. Usar condicional para validar, exemplo:

Proteção para Seek():

```
If SB1->(DbSeek(xFilial('SB1')+cCodigo))
     cDescr := SB1->B1_DESC
EndIf
```

Proteção para DbGoto():

```
SB1->(DbgoTo(nRecno))
If SB1->(!EOF())
     cDescr := SB1->B1_DESC
EndIf
```

- Não usar TCSqlExec(), para *Insert, Merge, Update* e *Delete.* Não será permitido apagar fisicamente registros do banco. Também não será permitido inserir registros por comando SQL, o que pode afetar o controle de numeração da tabela. Utilize os recursos da linguagem para alterar os dados. Se não for possível acessar a informação por ADVPL, por meio de Seek(), é possível que exista algum problema com construção da rotina, modelagem da tabela ou falta de índice. Se não for possível fazer a correção, utilize o filtro do update para

fazer um select trazendo o *recno* do registro a ser alterado, em seguida utilize o comando DbGoto(recno) para posicionar e alterar o registro.

- Não será permitido resolver tudo na única query. Deve-se simplificar a query e utilizar a linguagem para trazer o restante da informação. Exemplo: em algumas codificações, é realizado um Join na tabela de Cliente somente para trazer o nome. Assim, o resultado dessa query traz o código e o nome, sendo que, de uma forma otimizada, poderia ter trazido somente o código e utilizar um Posicione() no laço para trazer a informação do nome do Cliente.

- Usar os recursos da linguagem no lugar de subquerys. Pedir para o banco de dados resolver tudo nem sempre é uma boa ideia. Dependendo da quantidade de dados movimentada e da solicitação de uso das tabelas envolvidas, o melhor é fracionar a consulta. Muitas vezes é possível decompor as subquerys em laços advpl, e dentro desses laços acessar o dado por índice ou, em último caso, montando querys mais simples. Isso diminui a quantidade de dados movimentados por consulta.

- Não fazer `Select *` nas querys. Todo resultado de query retorna uma matriz de linhas e colunas. Informe somente os campos a serem utilizados na query para evitar trazer informações desnecessárias que custam tempo e processamento para o sistema.

- Não fazer conversões e operações matemáticas dentro da query. Ao fazer qualquer tipo de alteração no retorno de um campo em uma query, para o banco de dados nós criamos um novo campo não indexado. Dessa maneira, ao fazer operações matemáticas com campos numéricos, manipulação de string como `Substring`, `Trim` ou concatenação com `||` e `Case When` para condicionar um retorno etc. deixa a query mais lenta. Deve-se fazer esse tratamento dentro do laço com os recursos da linguagem ADVPL, tendo um custo muito menor do que trazer o dado tratado pela query.

- Não usar concatenação `'||'` na condição `where` de uma query. Ainda que os dois campos da concatenação sejam indexados, a soma deles para o banco de dados representa um novo campo não indexado, e ao adicionar a operação de soma a consulta torna-se muito mais lenta.

- Filtros de querys com campos não indexados, partes de conteúdo de campo ou uso de `Like`. Fazer uma `select` com um campo não indexado degrada o desempenho da query a média que aumenta o volume de dados na tabela. Se a regra permitir, utilize outros campos

para fazer um pré-filtro. Antes de usar um `Like` em uma query verifique se é possível utilizar o uso de outro recurso ou realizar esse filtro usando os recursos da linguagem. Verifique também se é necessário usar o operador `'%'` dos dois lados do parâmetro, já o Protheus é posicional. A utilização desses recursos será questionada.

`' '` é diferente de `' '` que é diferente de `''`. Bancos de dados como Oracle são mais rigorosos com o armazenamento e validação da informação semelhante ao padrão ISAM. O sistema Protheus quando grava uma informação vazia em um campo, na verdade, grava o tamanho do campo em espaços. Quando for fazer uma comparação em branco de qualquer campo, deve-se fazer o teste usando o comando `Criavar()` do campo, exemplo:

```
cQuery += " AND SE1.E1_BAIXA = '" + CriaVar("E1_BAIXA", .F.) + "' "
```

Importante: O segundo parâmetro da função `Criavar()` precisa ser false (`.F.`) para a rotina não retornar à informação do registro posicionado.

- Não usar `D_E_L_E_T_ <> '*'`. Usar a comparação diferente `'<>'` sempre tem um custo maior para o banco de dados porque prejudica a busca binária do banco de dados. No caso das querys Advpl é perfeitamente possível usar `D_E_L_E_T_ = ' '`. Em outro exemplo, ainda é possível substituir `<> ' '` por `> ' '` já que o caractere espaço tem o menor valor entre os caracteres digitáveis da tabela ASCII.

Importante: Não usar `D_E_L_E_T_ = ''`, (sem espaço entre as aspas simples) para o Oracle `''` significa `Null` e a comparação não trará os registros válidos.

- Observar a ordem dos índices ao montar as querys. Algumas vezes não temos todas as informações pertinentes a um determinado índice. Caso não precise filtrar a informação adicione na query o campo usando artifício de `> ' '` ou `BETWEEN ' ' AND 'ZZZ'` conforme a necessidade. Exemplo, temos na Tabela SE1 o índice:

  ```
  E1_FILIAL+E1_EMISSAO+E1_NATUREZ+E1_CLIENTE+E1_LOJA+E1_TIPO
  ```

  Ao filtrar os títulos de todas as filiais somente com o campo data, não estaremos utilizando nenhum índice o que tornaria a busca extremamente lenta considerando o volume dados da empresa. Agora se adicionarmos o campo filial podemos utilizar esse índice para filtrar os títulos a receber de todas as filiais:

  ```
  cQuery := "SELECT E1_FILIAL, E1_PREFIXO, E1_NUM, E1_PARCELA, E1_TIPO"
  cQuery +=  " FROM " + RetSqlName('SE1') + " SE1 "
  ```

```
cQuery += " WHERE SE1.E1_FILIAL > '04        ' "
cQuery +=   " AND SE1.E1_EMISSAO BETWEEN '20200201' AND '20200229' "
cQuery +=   " AND SE1.D_E_L_E_T_ = ' ' "
```

Importante: O índice somente é utilizado pelo banco de dados se os campos estiverem na mesma ordem do índice. Dessa forma se o campo `E1_EMISSAO` estivesse antes do campo `E1_FILIAL` o banco não estaria resolvendo a query pelo índice 11.

- Cuidado com a quantidade de informação filtrada na Query. É de responsabilidade do desenvolvedor implementar rotinas que racionalizem os recursos de hardware dos sistemas da empresa. No exemplo acima foi filtrado todo o mês de fevereiro na query o que representa cerca de 1,5 milhões de registros. Se o processamento for feito por dia, teremos uma média de 50mil títulos. Desse modo podemos desenvolver uma rotina que faça a divisão do volume de dados por dia de processamento:

```
For nI := 1 to Len(aDias)
   cQuery := "SELECT E1_FILIAL, E1_PREFIXO, E1_NUM, E1_PARCELA, E1_TIPO"
   cQuery +=  " FROM " + RetSqlName('SE1') + " SE1 "
   cQuery += " WHERE SE1.E1_FILIAL > '04        ' "
   cQuery +=   " AND SE1.E1_EMISSAO = " + aDias[nI] + " "
   cQuery +=   " AND SE1.D_E_L_E_T_ = ' ' "

   DbUseArea(.T.,"TOPCONN", TcGenQry( ,,cQuery), cQryRes, .T., .F.)

   Codificação)...

Next nI
```

- Velocidade e tempo de processamento. Muitos desenvolvedores acreditam que a única forma de ganhar desempenho na aplicação é passar por cima da arquitetura do sistema Protheus, fazendo *merges* e *updates* usando direto a linguagem do banco. Essa prática pode prover uma agilidade inicial no processamento, entretanto, não é confiável por não ser transacionada (colocar 'Begin Transaction' e 'End Trasaction' não resolve o problema porque esse comando só tem efeito em operações realizada através do TOP), além de não ser elástica, uma vez que com o crescimento do volume de dados o problema retorna. E quanto maior for a quantidade de registros afetados, maior será a possibilidade de *Dead Locks* nas tabelas do sistema.

Bons desenvolvedores utilizam o processamento em múltiplos threads que permite dividir e executar simultaneamente o processamento. Caso o volume de dados aumente, isso pode ser resolvido aumentando a quantidade de threads.

Ainda utilizando o exemplo da query de títulos, poderíamos usar a função StarJob() para criar uma Thread e rodar a rotina simultaneamente para cada dia do mês de fevereiro. Esse paralelismo diminuiria consideravelmente o tempo de execução da rotina.

- Não utilizar o comando obsoleto `TcQuery()`, para padronizar a codificação, utilizar preferivelmente strings SQL com o comando `dbUseArea()`, exemplo:

```
DbUseArea(.T.,"TOPCONN", TcGenQry( ,,cQuery), cQryRes, .T., .F.)
```

Pode ser usado o comando Embedded Sql, apesar de não ser tão prática para fazer depuração. Caso precise depurar uma query desse tipo, utilize o comando `Getlastquery()[2]` no console do depurador para obter a query.

Mais informações: https://tdn.totvs.com/display/framework/Embedded+SQL

- Sempre usar a função `GetNextAlias()` para criar alias temporários de querys. Não será permitido **"*chumbar*"** um nome de alias por correr o risco de repetir o nome da área no sistema. Isso evita aqueles famosos e desnecessários testes para verificar se a área está aberta.

- Sempre fechar a área de uma query ao sair da função. É comum ver códigos com abertura de área em uma rotina para usar em outra. Abra a área somente na rotina que será utilizada, isso deixa o código mais organizado e evita problemas de estouro com área aberta. Feche com o comando `DbCloseArea()` assim que não for mais preciso mantê-la como no término de um laço.

Importante: Se o arquivo do código-fonte for novo, todo o código será avaliado conforme as regras e o desenvolvimento. Em arquivos existentes será avaliado as rotinas alteradas. Se o novo código-fonte ou trecho alterado não estiver conforme as regras de desenvolvimento, a GMUD será devolvida para ser ajustada.