【セッションNo. 4】

Delphi/400技術セッション

開発者が知りたい実践プログラミングテクニック！

～明日から使えるテクニック集～

株式会社ミガロ.
システム事業部　システム2課
辻野　健

【アジェンダ】

Delphi/400技術セッション

1. **ユーザビリティ**の向上
   1－1. 同時に複数画面を起動する方法
   1－2. グリッド操作時の自動フォーカス制御方法

2. 開発効率、保守性の向上
   2－1. 画面共通処理のフレームによる部品化

1. ユーザビリティの向上

■ ユーザーがシステムに求めるもの

- 安定性（信頼性）
  - エラーやシステム障害が発生しない
  - セキュリティがしっかりしている
- **使い勝手**
  - 本来の業務がはかどる
    - 作業をできるだけ簡単に
    - 入力したデータの変更、修正を簡単に
    - 資料の作成を簡単に
- レスポンス
  - 処理速度、操作に対する反応が早い
- 汎用性、拡張性に優れている
  - 柔軟性があり新機能の追加や運用でカバーできる設計になっている

■ 使い勝手 ＝ ユーザビリティ

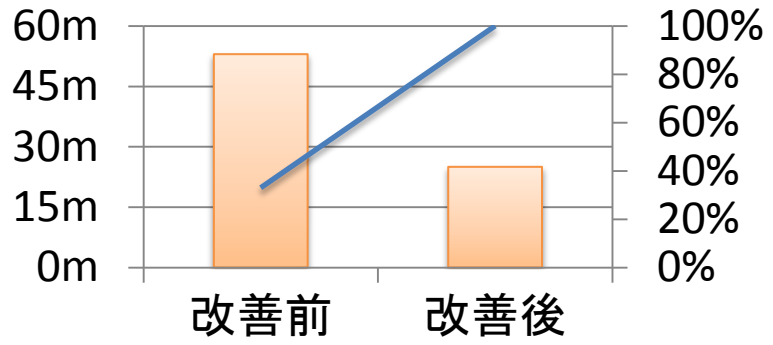
- ユーザビリティの向上による効果
  - システム使用前
    - システムがあまりに複雑すぎるせいで、一人前のオペレータになるまでに集中的なトレーニングが必要

      →ユーザビリティを重視したデザインによって、研修期間を日単位、場合によっては週単位で短縮できる
  - システム使用時
    - 求める情報を取得するのに時間がかかる
    - 誤った操作によるミスが発生

      →オペレーターの作業効率向上によりコストが削減できる

■ 使い勝手 ＝ ユーザビリティ

- ユーザビリティの向上がもたらす実例
  - 申請システムの例



タスク達成時間
（目的達成までに要した時間）
53分 ⇨ 25分

タスク達成率
（60分以内に目的を達成したユーザの割合）
33% ⇨ 100%

ユーザビリティの向上により、作業効率を向上させることができる

  - その他の実例

業種	対策	効果
不動産会社	マンション販売促進サイト	閲覧者のモデルルーム予約率が3倍に
新聞社	ニュースサイト改善	一人当りのページ閲覧数が2倍に増加
携帯電話	請求書デザイン改善	コールセンターへの問合せ件数が半減

UNISYS TECHNOLOGY REVIEW　第 110 号，NOV. 2011
「Web 戦略を成功に導くためのユーザインタフェース設計プロセス」参照

■ 使い勝手 ＝ ユーザビリティ

- そこで、今回はユーザビリティの向上を行う方法として

同時に複数画面を起動する方法

グリッド操作時の自動フォーカス制御方法

１－１．同時に複数画面を起動する方法

こんな要望ないでしょうか？

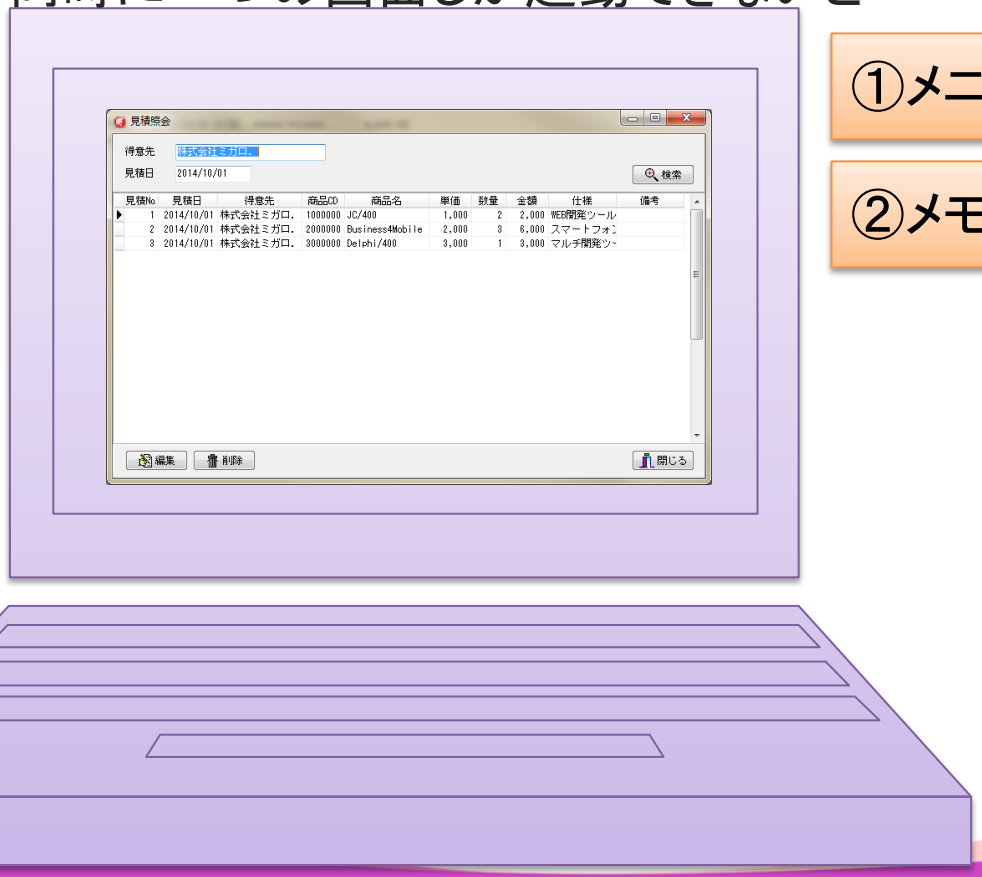
見積照会で参照した過去の情報を基に、新しい見積を作成したい

- 同時に一つの画面しか起動できないと・・・

こんな要望ないでしょうか？

見積照会で参照した過去の情報を基に、新しい見積を作成したい

- 同時に一つの画面しか起動できないと・・・



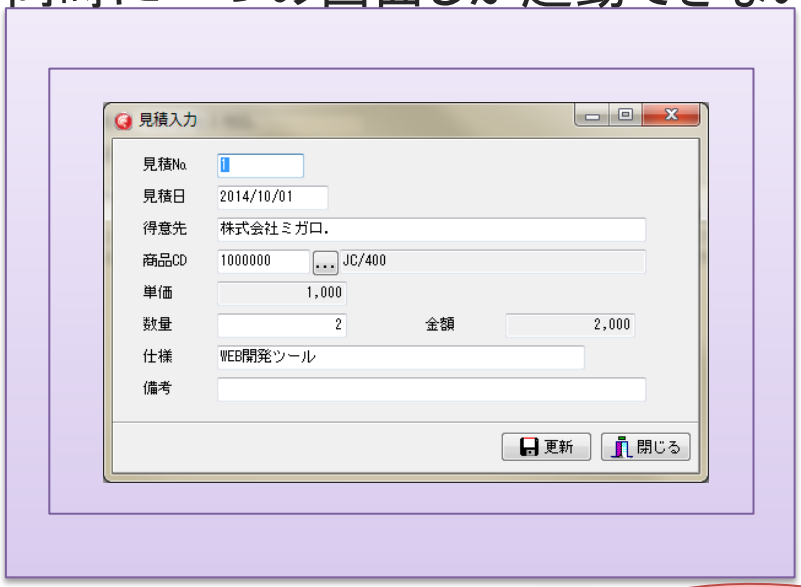
①メニューより見積照会起動

②メモを取るなどし、画面終了

こんな要望ないでしょうか？

見積照会で参照した過去の情報を基に、新しい見積を作成したい

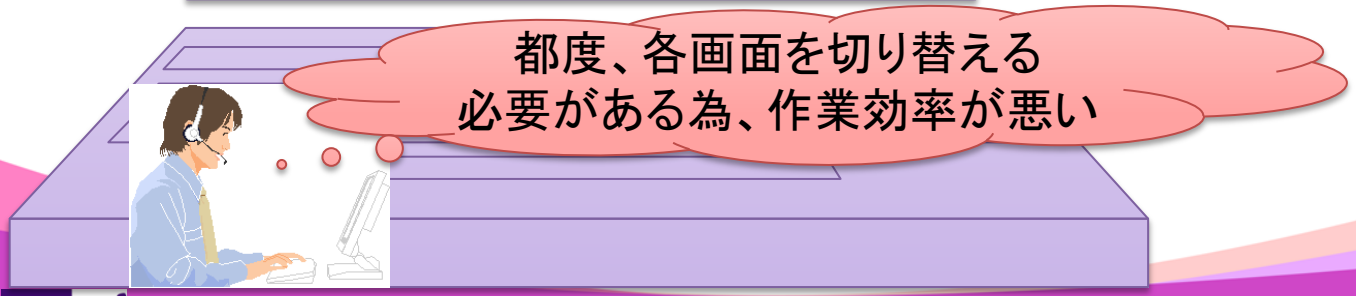
- 同時に一つの画面しか起動できないと・・・



①メニューより見積照会起動

②メモを取るなどし、画面終了

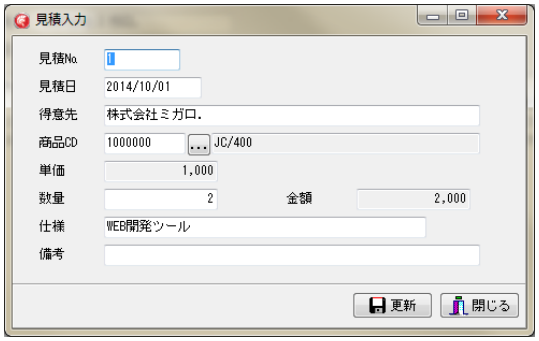
③メニューより見積入力起動



■ こんな要望ないでしょうか？

見積照会で参照した過去の情報を基に、新しい見積を作成したい

➢ 一つの画面に様々な機能を盛り込むと・・・



見積照会、受注照会、発注照会などの機能を盛り込む
→過去のデータを見ながら見積の登録が可能

◆しかし・・・

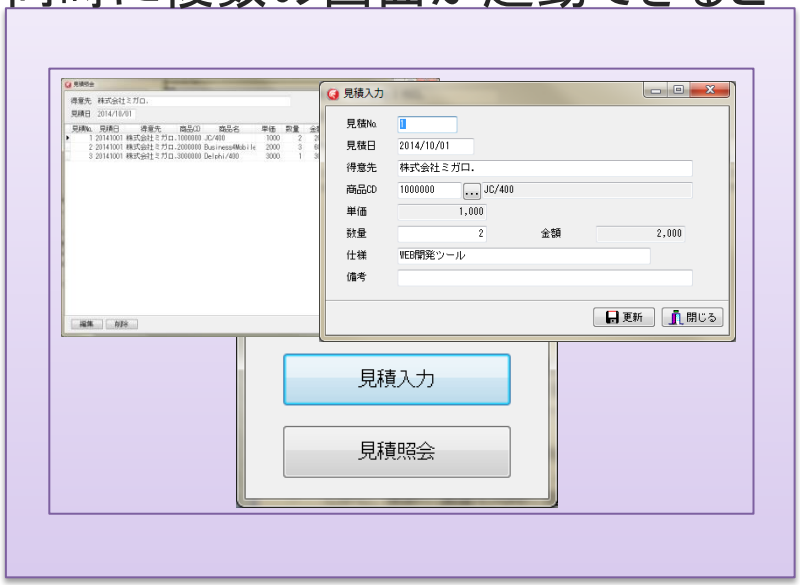
各画面を作りこむ必要があるため、
開発コストが増大してしまう

システムが複雑になり使いづらい

こんな要望ないでしょうか？

見積照会で参照した過去の情報を基に、新しい見積を作成したい

- 同時に複数の画面が起動できると‥



①メニューより見積照会起動

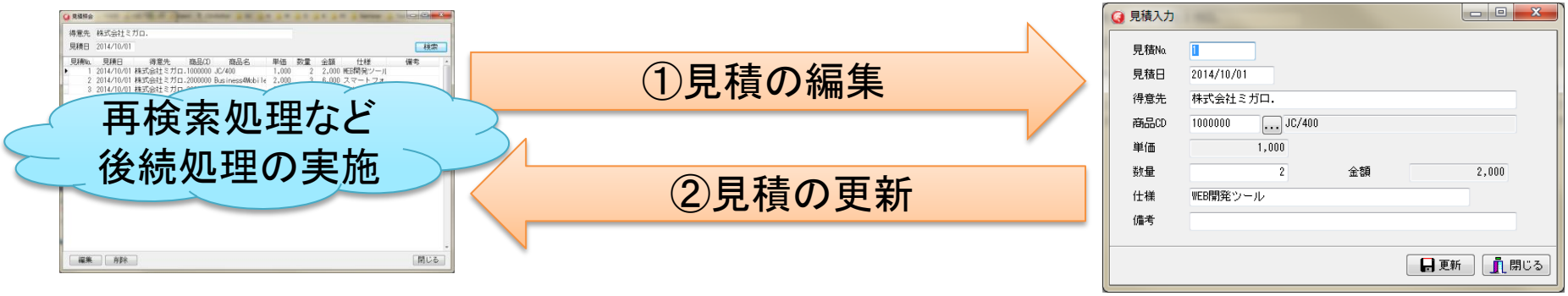
②メニューより見積入力起動

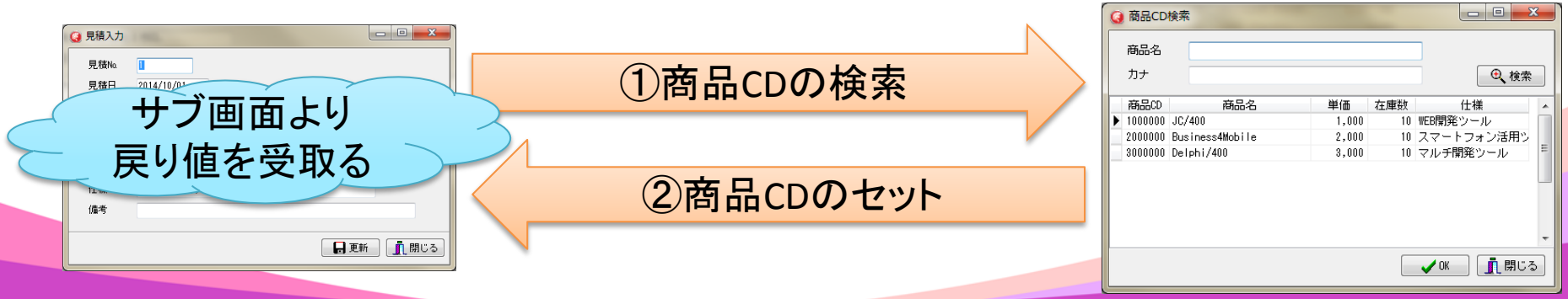
過去の見積を検索しながら新規見積を登録できる為、作業効率が良い

複数画面を同時に起動する方法をご紹介

■ 画面の起動方法

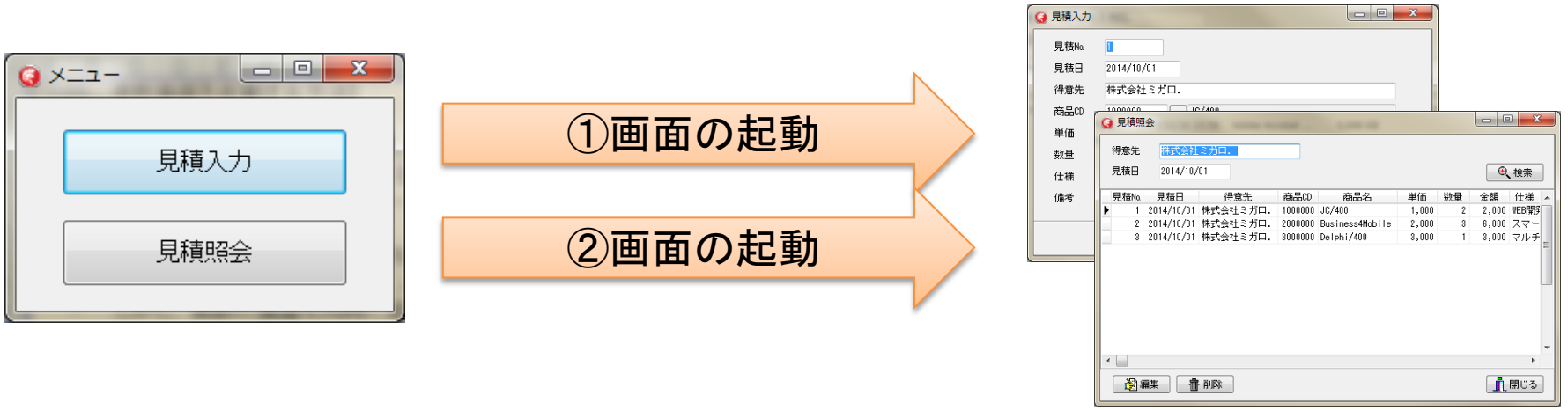
- Delphiにて画面を起動させる方法
  1. ShowModal
     - ➢ 遷移元画面とのつながりを持ったサブ画面として表示する方法
     - ➢ 同一Exe内の他の画面は一切操作不可





■ 画面の起動方法

- Delphiにて画面を起動させる方法
  2. Show
     - ➢ 他の画面とつながりを持たない独立した画面として表示する方法
     - ➢ 起動画面以外も操作可能



Showを使用することで複数画面の
同時起動が可能になる

Show使用時の注意点

①ShowModalとShowではメモリの解放方法が異なる

- Show
  - 遷移先画面のFormのOnCloseイベントにメモリ解放処理を記述する

サブ画面のOnCloseイベント

```
procedure TfrmSubForm.FormClose(Sender: TObject; var Action: TCloseAction);
begin

  // 画面終了後、メモリ解放
  Action := caFree;
end;
```

サブ画面終了時に、メモリを解放する処理を記述する

Show使用時の注意点

①ShowModalとShowではメモリの解放方法が異なる

- ShowModal
  - 遷移元画面に遷移先画面のメモリ解放処理を記述する

メイン画面「サブ画面起動用」ボタンのOnClickイベント

```
procedure TfrmMainForm.btnShowSubFormClick(Sender: TObject);
begin
  // サブ画面を生成
  frmSubForm := TfrmSubForm.Create(Self);
  try
    // サブ画面を起動
    frmSubForm.ShowModal;

    // サブ画面より戻り値を受け取る
    Edit1.Text := frmSubForm.RValue;
  finally
    // サブ画面のメモリを解放
    FreeAndNil(frmSubForm);
  end;
end;
```

メイン画面の処理はサブ画面が終了するまで停止状態となる

サブ画面終了後、サブ画面のメモリを解放する

Show使用時の注意点

②同一画面を複数起動する場合、画面生成時に各画面のグローバル変数ではなく、ローカル変数を使用する

- 各画面のグローバル変数とは
  - 各画面毎に自動的に設定されているクラスの変数

フォーム生成時のグローバル変数

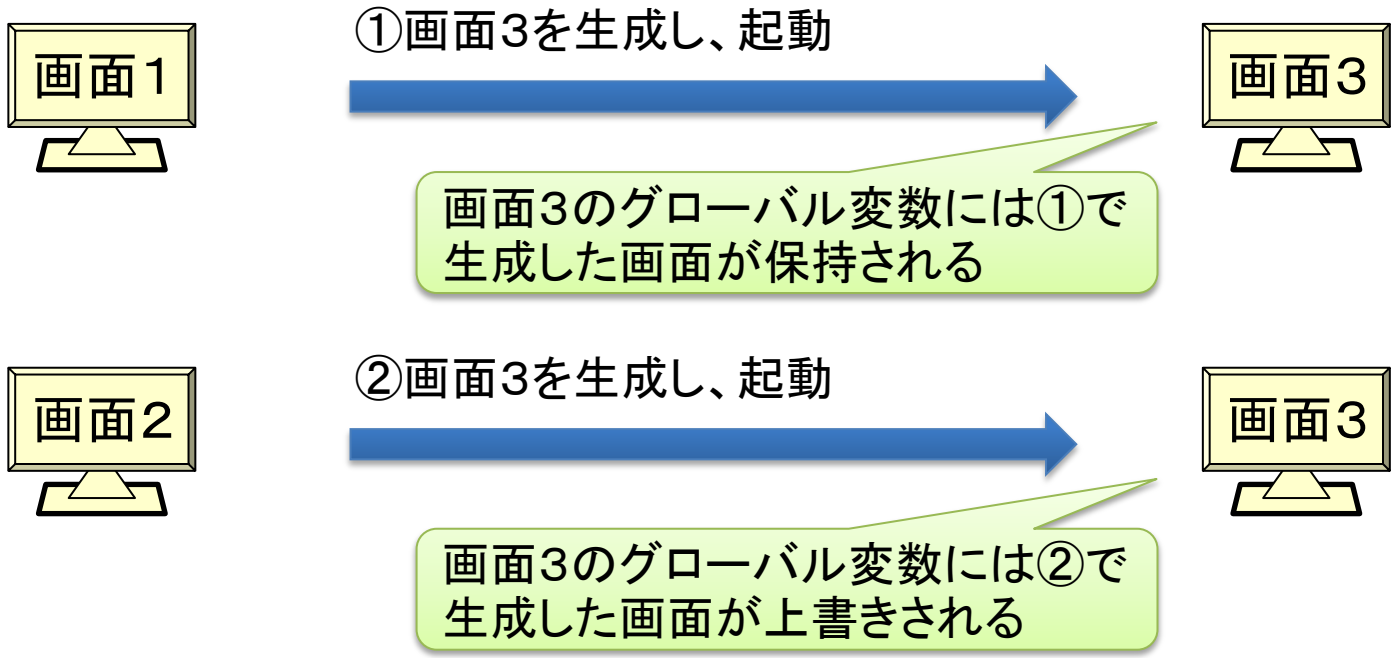
```
type
  TForm1 = class(TForm)
  private
    { Private 宣言 }
  public
    { Public 宣言 }
  end;

var
  Form1: TForm1;

implementation
```

画面毎のグローバル変数

Show使用時の注意点

②同一画面を複数起動する場合、画面生成時に各画面のグローバル変数ではなく、ローカル変数を使用する

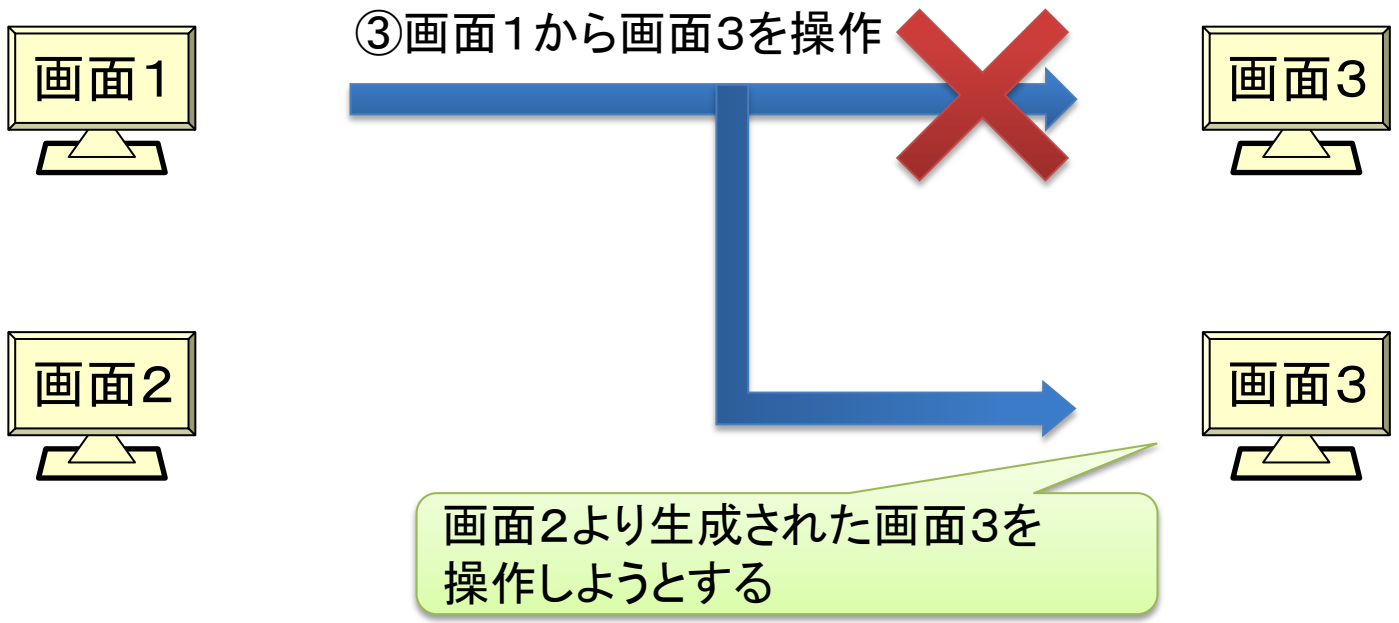
- 各画面のグローバル変数を使用した場合



Show使用時の注意点

②同一画面を複数起動する場合、画面生成時に各画面のグローバル変数ではなく、ローカル変数を使用する

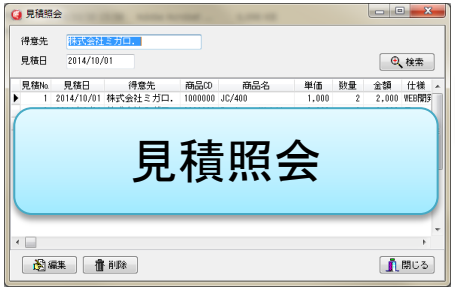
- 各画面のグローバル変数を使用した場合



■ 更にこんな要望ないでしょうか？

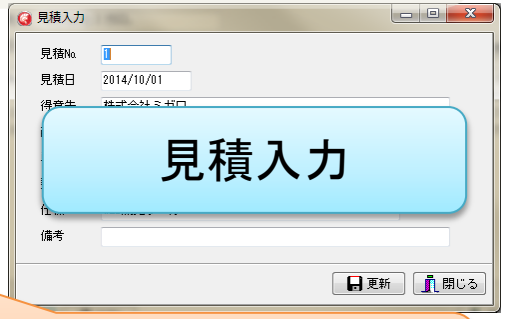
見積照会より見積の編集を行いながら、過去の
受注情報を確認したい

編集対象を検索



見積照会

見積を編集



見積入力

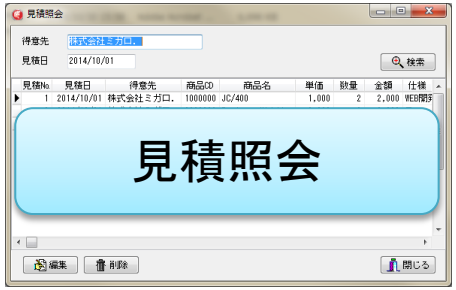
過去の受注情報
が知りたい

後続処理を実施するため
一般的にShowModalにて起動

■ 更にこんな要望ないでしょうか？

見積照会より見積の編集を行いながら、過去の
受注情報を確認したい

編集対象を検索



一旦見積入力
終了

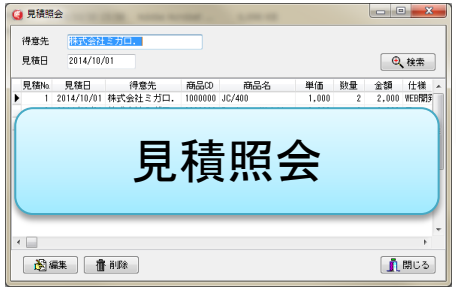
過去の受注情報
が知りたい

受注情報を検索

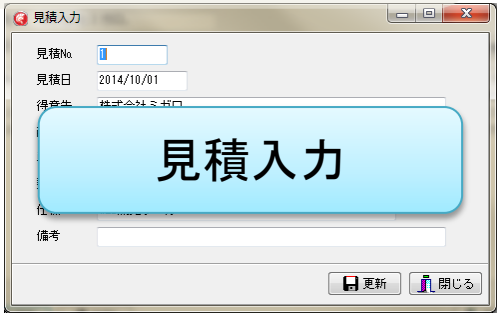
■ 更にこんな要望ないでしょうか？

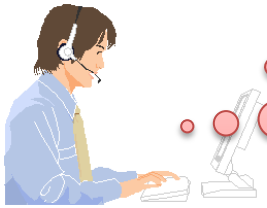
見積照会より見積の編集を行いながら、過去の
受注情報を確認したい

編集対象を検索



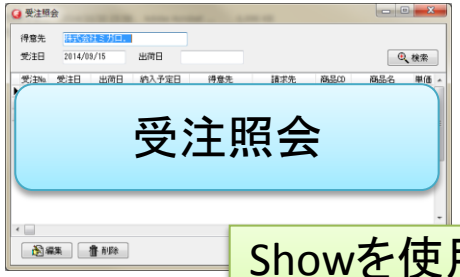
再度編集





見積入力を終了することなく
過去の受注の検索ができれば
もっと使い勝手が良くなるのに・・・

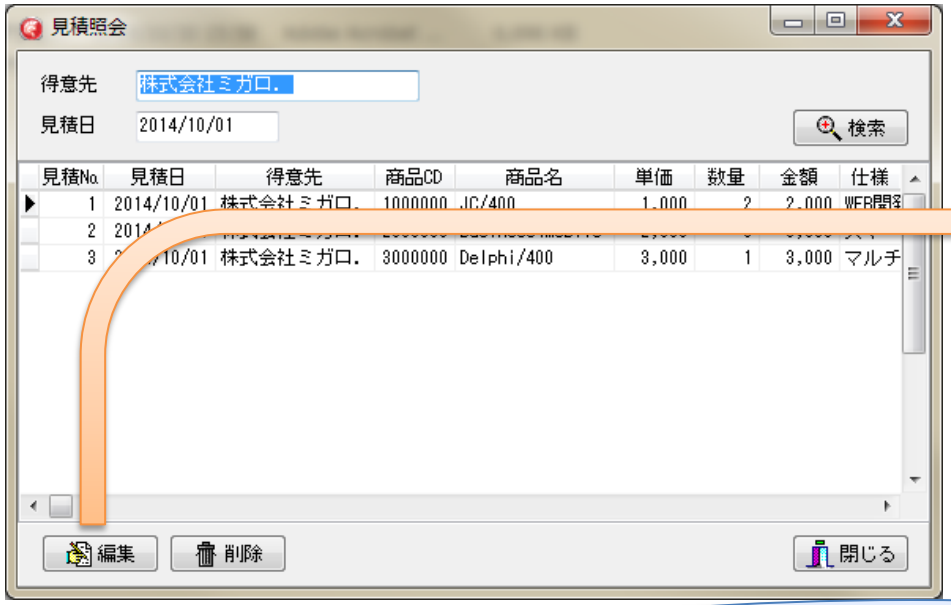
受注情報を検索

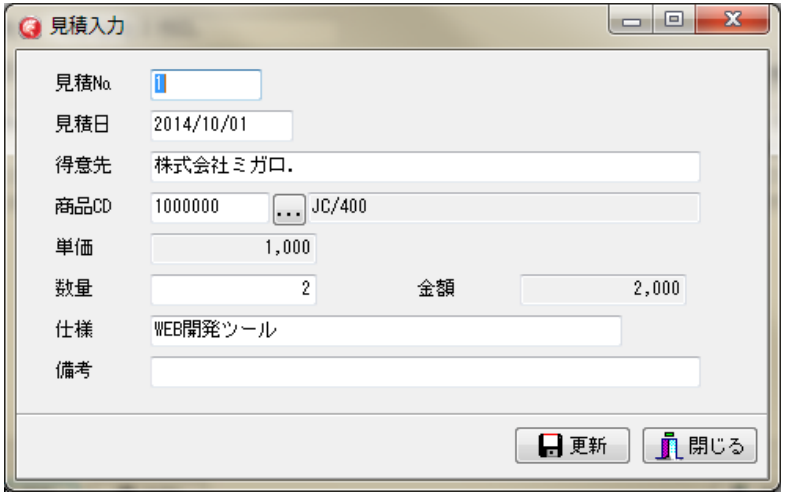


Showを使用したワンランク上のテクニックをご紹介

■ Show使用による擬似ShowModalの実装

- 実装概要
  - ① 見積照会の編集ボタン押下時に、見積入力をShowで起動する
  - ② 見積照会にて後続処理実施の為、見積入力の終了を監視する





ShowModalのように、照会画面の
後続処理のタイミングを取得しつつ、
他の画面も操作可能となる

■ Show使用による擬似ShowModalの実装

①見積入力の起動

- ShowModalではなく、Showを使用する

見積照会「編集」ボタンのOnClickイベント

```
  private
    FMGR010From: TfrmMGR010;  // 見積入力画面
-------------------------------------------------------------
procedure TfrmMGR030.btnEditClick(Sender: TObject);
begin

  // 見積入力を生成
  FMGR010From := TfrmMGR010.Create(Self);

  // 見積入力を表示
  FMGR010From.Show;

  // 0.5秒間隔で見積入力が表示されているかを確認
  tmrWatch.Enabled := True;

  // 画面を使用不可にする
  Enabled := False;
end;
```

Showを使用するため、グローバル変数ではなく、ローカル変数を使用する

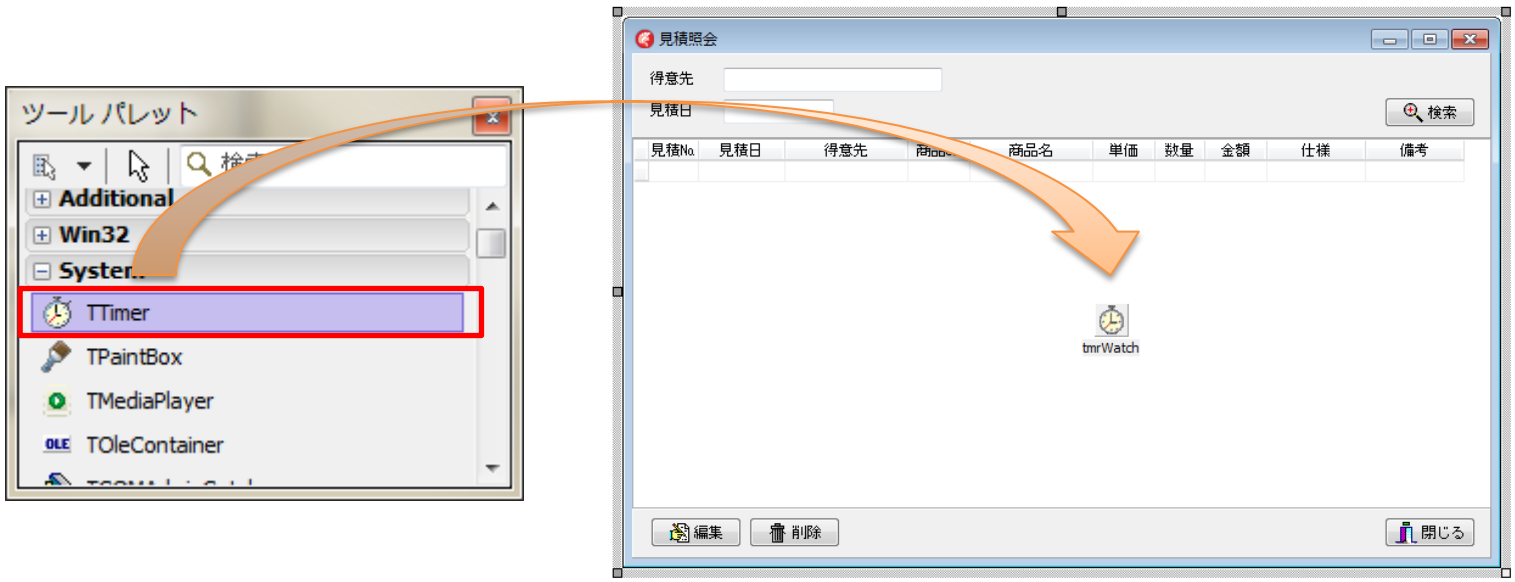
見積入力終了後の後続処理を実施するため、見積入力がまだ表示されているかを監視する

見積入力表示中は、見積照会のみを操作不可に設定する

■ Show使用による擬似ShowModalの実装

②見積照会より起動した見積入力の終了の監視

- TTimerコンポーネントの使用



画面へTTimerコンポーネントを配置する
Nameプロパティを「tmrWatch」、
Intervalプロパティを「500」に設定する
（0.5秒間隔で終了確認）

■ Show使用による擬似ShowModalの実装

②見積照会より起動した見積入力の終了の監視

- TTimerコンポーネントの使用

見積入力終了監視タイマーのTimerイベント

```
procedure TfrmMGR030.tmrWatchTimer(Sender: TObject);
begin

  // 見積入力が終了している場合
  if (not Assigned(FMGR010From)) or
     (Assigned(FMGR010From) and (not FMGR010From.Showing)) then
  begin
    // 見積入力終了の監視を停止
    tmrWatch.Enabled := False;

    // 再検索処理を実施
    btnSearch.Click;

    // 画面を使用可能にする
    Enabled := True;
  end;
end;
```

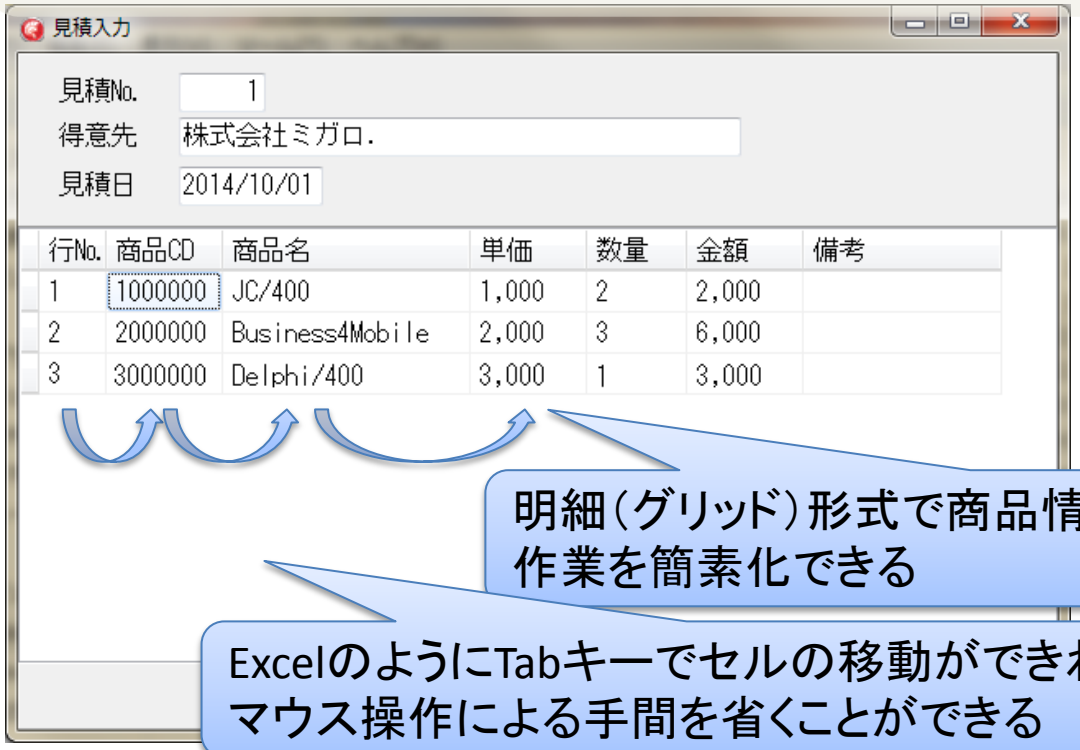
見積入力が解放されている または
見積入力が表示されていない場合、
後続処理を実施する

見積入力が表示されている間、見積照会は
操作不可にする為、見積入力終了後、
操作可能に戻す

１－２. グリッド操作時の自動フォーカス制御方法

こんな要望ないでしょうか？

一つの得意先に対して、複数の商品の見積を行いたい



明細（グリッド）形式で商品情報を入力することで
作業を簡素化できる

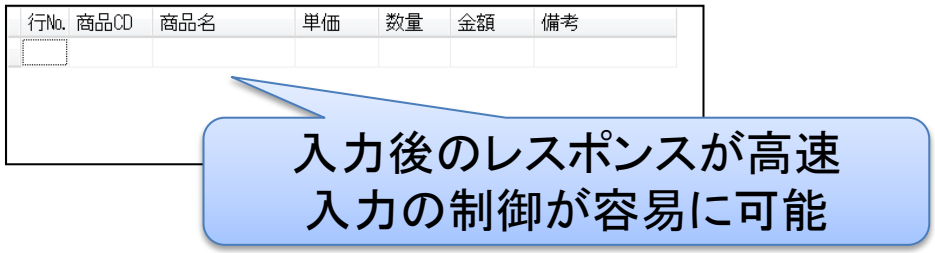
ExcelのようにTabキーでセルの移動ができれば
マウス操作による手間を省くことができる

Tabキーによる自動フォーカス制御方法をご紹介

■ 明細への入力

- Delphiにて、グリッドを実装するコンポーネント
  1. TDBGrid
     - ➢ 直接データベースまたはデータセットと接続して使用するコンポーネント

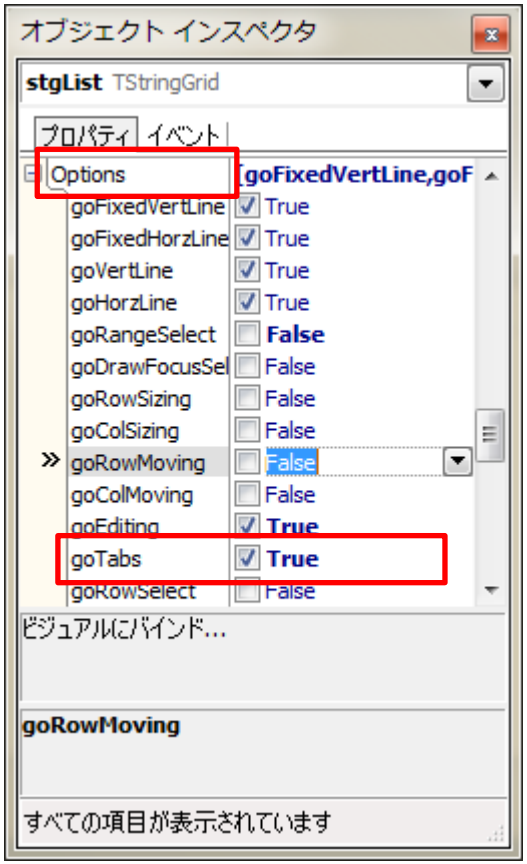
  2. TStringGrid
     - ➢ テキストデータを表形式で表示するコンポーネント

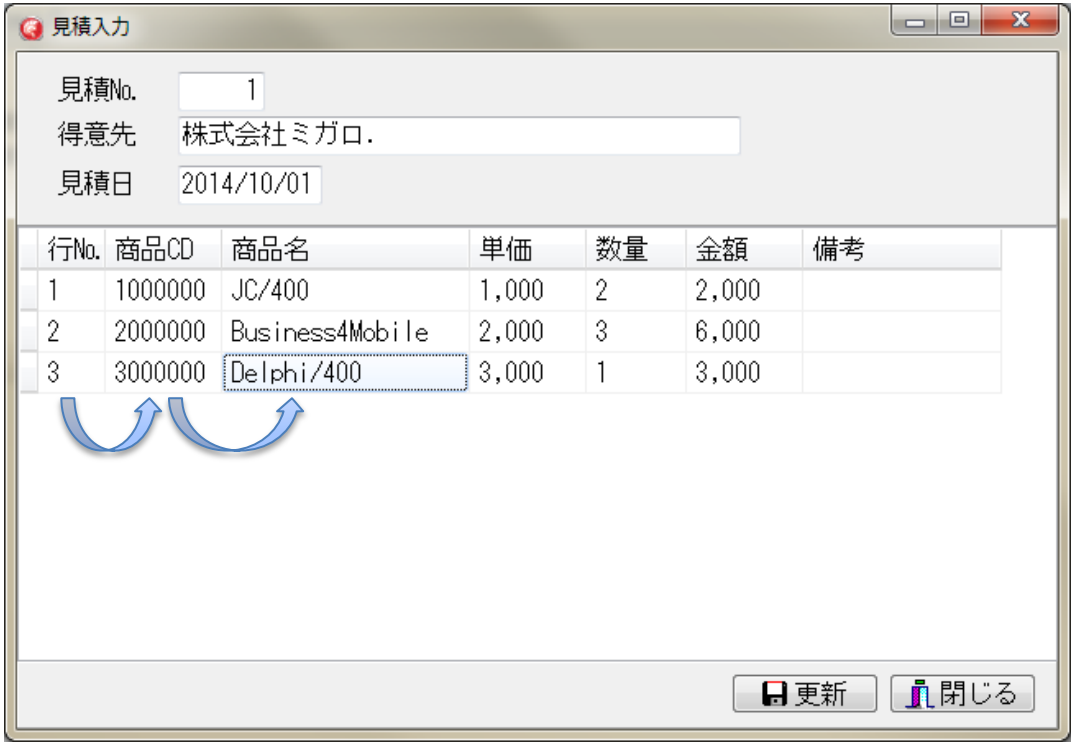


TStringGridを用いてカーソル移動テクニックをご紹介

TStringGridでのTabキーによるカーソル移動

実は・・・　Optionsプロパティの設定のみで実装可能

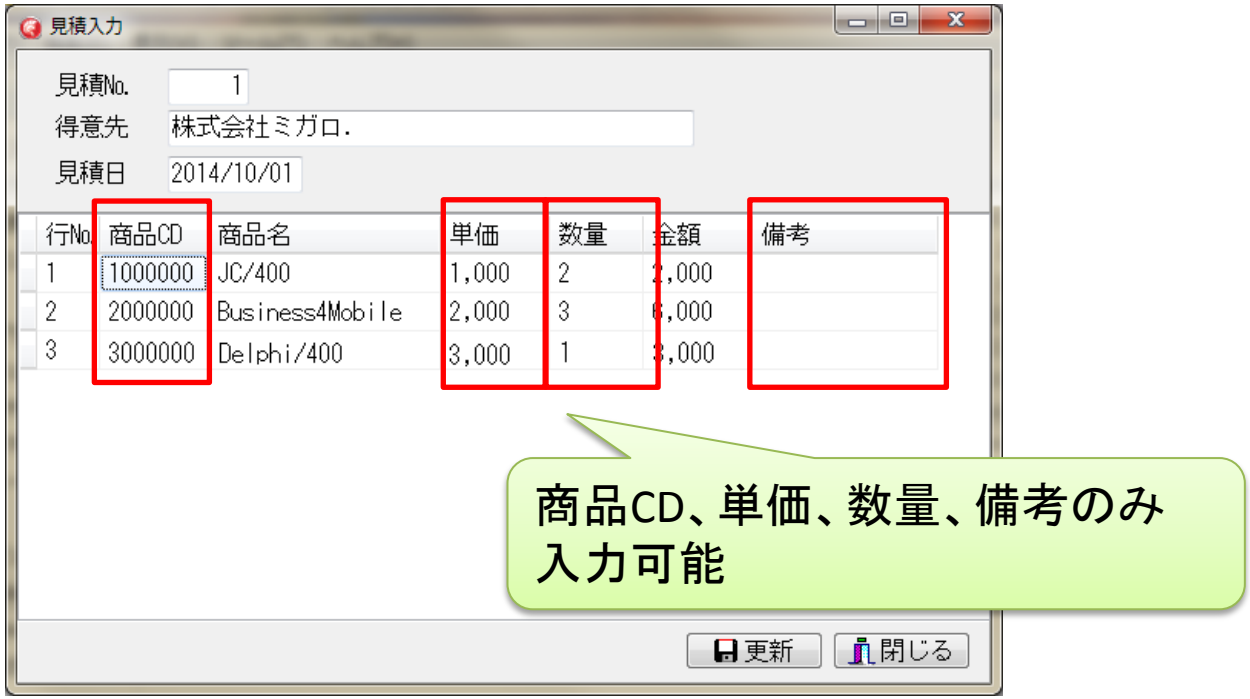




しかし・・・

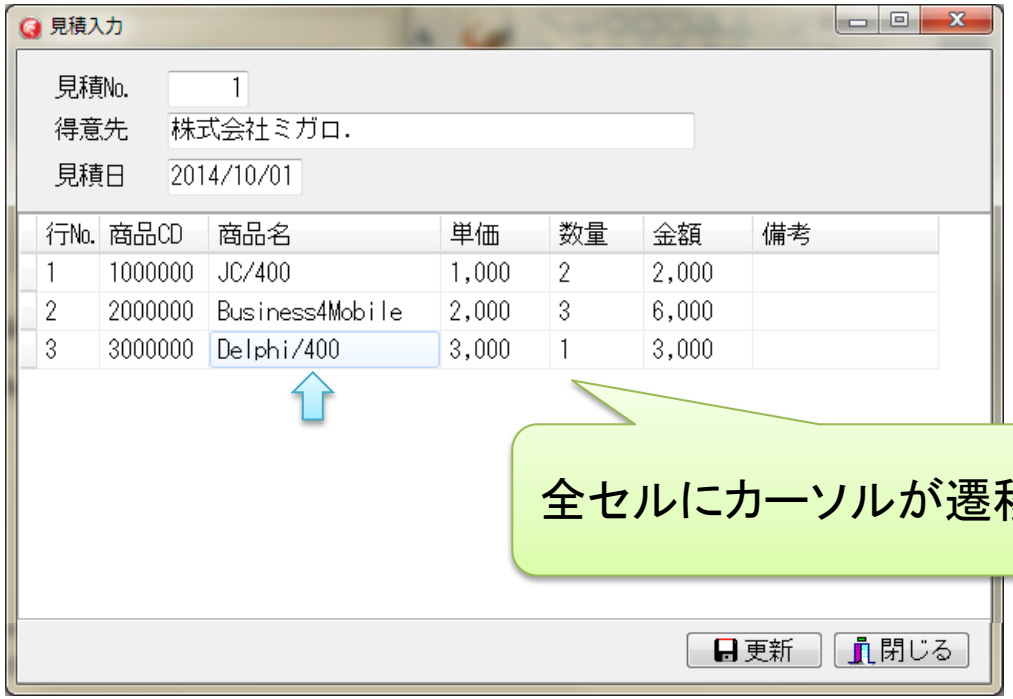
TStringGridでのTabキーによるカーソル移動

Optionsプロパティの設定のみの場合



TStringGridでのTabキーによるカーソル移動

Optionsプロパティの設定のみの場合

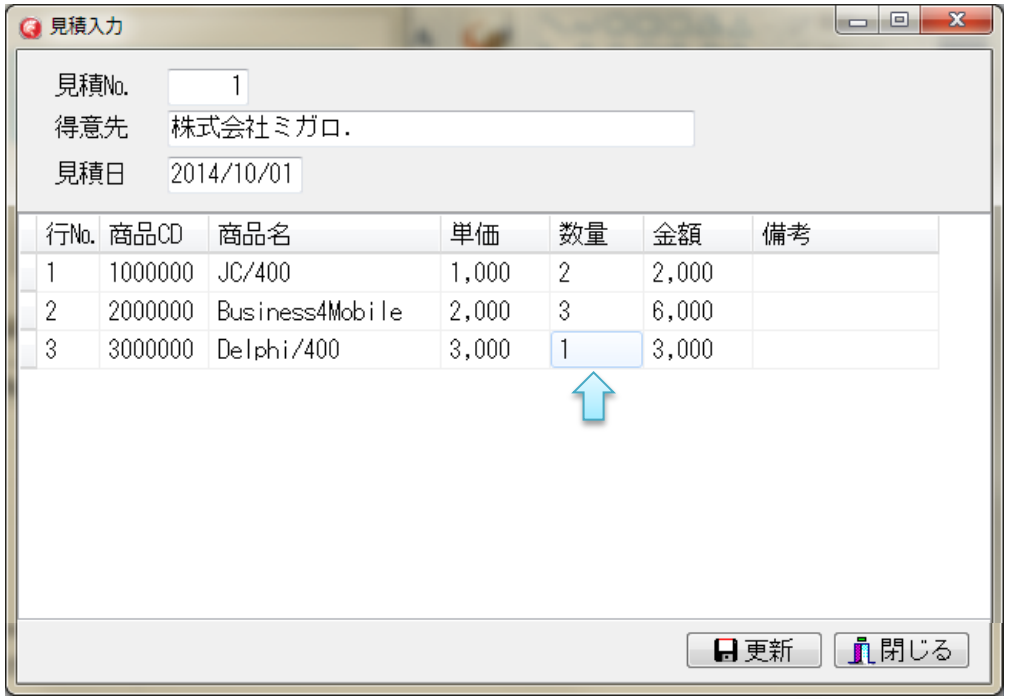


全セルにカーソルが遷移する

入力可能なセルのみカーソルが
遷移すれば簡単に入力できるのに・・・

TStringGridでのTabキーによるカーソル移動

入力可能セルのみカーソルが移動すれば・・・



簡単に入力できる為
作業が捗る

入力可能セルのみカーソル移動させる方法をご紹介

■ TStringGridでのTabキーによるカーソル移動

- 実装概要
  - ① 指定のセルが、入力可能かどうかを判定する
  - ② 次の入力可能セルを見つけ、そのセルにカーソルを移動する

■ TStringGridでのTabキーによるカーソル移動

①入力可能セルの判定

- 入力可能セルを判定するための関数

CanEdit関数

```
  private
    function CanEdit(ACol: Integer): Boolean;  // 変更可能判断
```

```
const
  // 列位置情報
  cCol_GYNO = 1;    // 行No.
  cCol_SHCD = 2;    // 商品CD
  cCol_SHNM = 3;    // 商品名
  cCol_TANK = 4;    // 単価
  cCol_SURY = 5;    // 数量
  cCol_KING = 6;    // 金額
  cCol_BIKO = 7;    // 備考
```

どの列に、何の情報が表示されているかを定数にて管理する

```
function TfrmMGR010.CanEdit(ACol: Integer): Boolean;
begin

  // 商品CD、単価、数量、備考列のみ変更可能
  Result := ACol in [cCol_SHCD, cCol_TANK, cCol_SURY, cCol_BIKO];
end;
```

入力可能なセルの場合にTrueを返却する関数を作成する

TStringGridでのTabキーによるカーソル移動

②カーソルの移動

- CanEdit関数を使用した入力可能セルへのカーソル移動

明細のOnKeyDownイベント＜1＞

```
procedure TfrmMGR010.stgListKeyDown(Sender: TObject; var Key: Word; Shift: TShiftState);
var
  iCol, iRow : Integer;
begin
  // Tabキーが押下された場合
  if Key in [VK_TAB] then
  begin
    Key := 0;    // 標準処理が実行されないようキーを初期化

    // Shiftキーが押下されていない場合
    if not (ssShift in Shift) then
    begin
      // 次のセルを決定
      iRow := stgList.Row;
      iCol := stgList.Col + 1;
      while iRow < stgList.RowCount do
      begin
        while iCol < stgList.ColCount do
        begin
```

次のカーソル位置を割り出すための変数を定義する

Tabキーが押下された場合のみ処理を実施する

Tabキー押下時の標準の処理が実行されると、カーソルの位置設定後、もう一つ次のセルへカーソルが移動してしまう

Shiftキーが押下されていない場合、一つ次の入力可能セルへカーソルを移動させる

TStringGridでのTabキーによるカーソル移動

②カーソルの移動

- CanEdit関数を使用した入力可能セルへのカーソル移動

明細のOnKeyDownイベント＜2＞

```
      // 編集可能セルが見つかった場合
      if CanEdit(iCol) then
      begin
        // セルを指定し、処理終了
        stgList.Col := iCol;
        stgList.Row := iRow;
        Exit;
      end;

      // 次の列をチェック
      Inc(iCol);
    end;

    // 次の行をチェック
    Inc(iRow);
    iCol := 1;
  end;
```

各セル順番に入力可否をチェックし、
入力可能セルが見つかったタイミングで
カーソルをセットする

■ TStringGridでのTabキーによるカーソル移動

②カーソルの移動

- CanEdit関数を使用した入力可能セルへのカーソル移動

明細のOnKeyDownイベント＜3＞

```
      // 編集可能セルが見つからなかった場合、行追加
      stgList.RowCount := stgList.RowCount + 1;
      stgList.Row      := stgList.RowCount - 1;  // 最終行にカーソル
      stgList.Col      := cCol_SHCD;             // 商品CDにカーソル

      stgList.Cells[cCol_GYNO, stgList.RowCount - 1] := IntToStr(stgList.RowCount - 1);  // 行No.
    end

    // Shiftキーが押下されている場合
    else
    begin

      （中略）

    end;
  end;
end;
```

一つ次の入力可能セルが存在しなかった場合、最終行に1行追加し、その行の商品CDへカーソルをセットする

Shiftキーが押下されている場合、一つ前の入力可能セルへカーソルを移動させる

TStringGridでのTabキーによるカーソル移動

③【補足】 セルの入力制御

- CanEdit関数を使用したセルの入力制御

明細のOnSelectCellイベント

```
procedure TfrmMGR010.stgListSelectCell(Sender: TObject; ACol, ARow: Integer;
  var CanSelect: Boolean);
begin

  // 入力可否制御
  if CanEdit(ACol) then    // 入力可能セルの場合
  begin
    stgList.Options := stgList.Options + [goEditing];
  end
  else                     // 入力不可セルの場合
  begin
    stgList.Options := stgList.Options - [goEditing];
  end;
end;
```

入力可能セルにカーソルがセットされている場合、Optionsプロパティに goEditingを設定する

入力不可セルにカーソルがセットされている場合、Optionsプロパティから goEditingを解除する

2. 開発効率・保守性の向上

■ 開発効率・保守性の高いプログラム

- 開発効率
  - 開発を行う際に、どれほど早く、または少ない労力で、作業を進められるかといった度合い
- 保守性
  - 改定を行うために必要な労力に関係するもの
  - 保守性は、4つの品質副特性を持つ
    - ① 解析性
      - ➢ある変更を行うための修正箇所の割り出しの容易さ
    - ② 変更性
      - ➢ある変更を行うための修正の容易さ
    - ③ 安定性
      - ➢ある変更を行う際のデグレードの発生確率
    - ④ 試験性
      - ➢ある変更を行う際のテストの容易さ

■ 処理の共通化 ⇨ 開発効率・保守性の向上

- 処理の共通化がもたらす効果
  - 開発効率
    - ➢ 各画面に同じ処理を複数回記述する事になり、開発に時間がかかる

      →処理を共通化することで、その処理が使用される画面にかかる開発工数を大幅に削減できる
  - 保守性
    - ➢ 仕様の変更が発生した場合、各画面への修正が必要となり、作業時間、デグレードが発生するリスクが高い

      →共通化された処理のみの修正で全画面へ修正が反映される為メンテナンスに費やす労力を最小限に抑えることができる

■ 処理の共通化 ⇨ 開発効率・保守性の向上

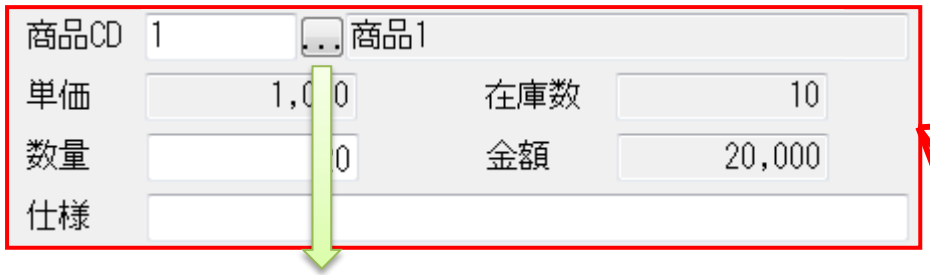
- そこで、今回は処理の共通化を行う方法として

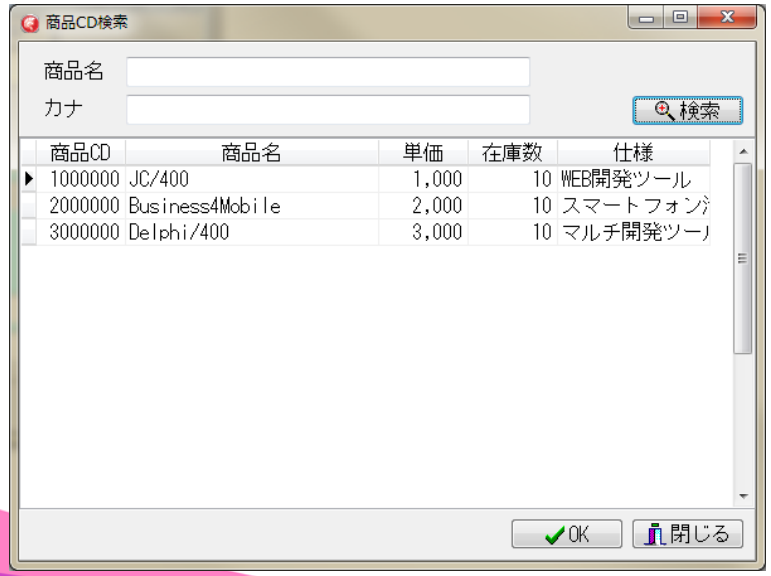
画面共通処理のフレームによる部品化

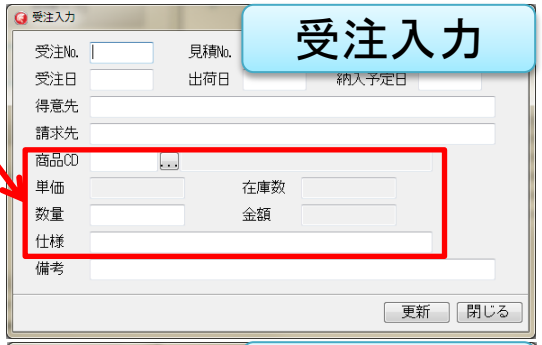
２－１．画面共通処理のフレームによる部品化

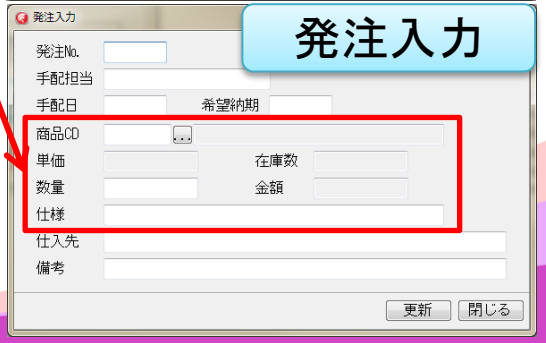
こんな時ないでしょうか？

複数のコンポーネントで1セットとなる項目を
複数の画面へ配置したい









こんな時ないでしょうか？

複数のコンポーネントで1セットとなる項目を
複数の画面へ配置したい

- 各画面へ処理を記述した場合

何度も同じ項目の配置や処理の記述
を行う為、開発効率が悪い

- 処理を共通化した場合

同じ処理を記述する必要が無いため
開発時間の短縮ができる

■ こんな時ないでしょうか？

複数のコンポーネントで1セットとなる項目を
複数の画面へ配置したい

項目の桁数が変更になり、各項目の修正が必要になった・・・

- 各画面へ処理が記述されていると・・・

修正箇所の割り出しに時間がかかる
修正自体にも時間がかかる

⇒ 保守性が低いシステムになってしまう

- 処理が共通化されていると・・・

修正箇所が1箇所の為、
短時間で修正することができる

⇒ 保守性が高いシステムになる

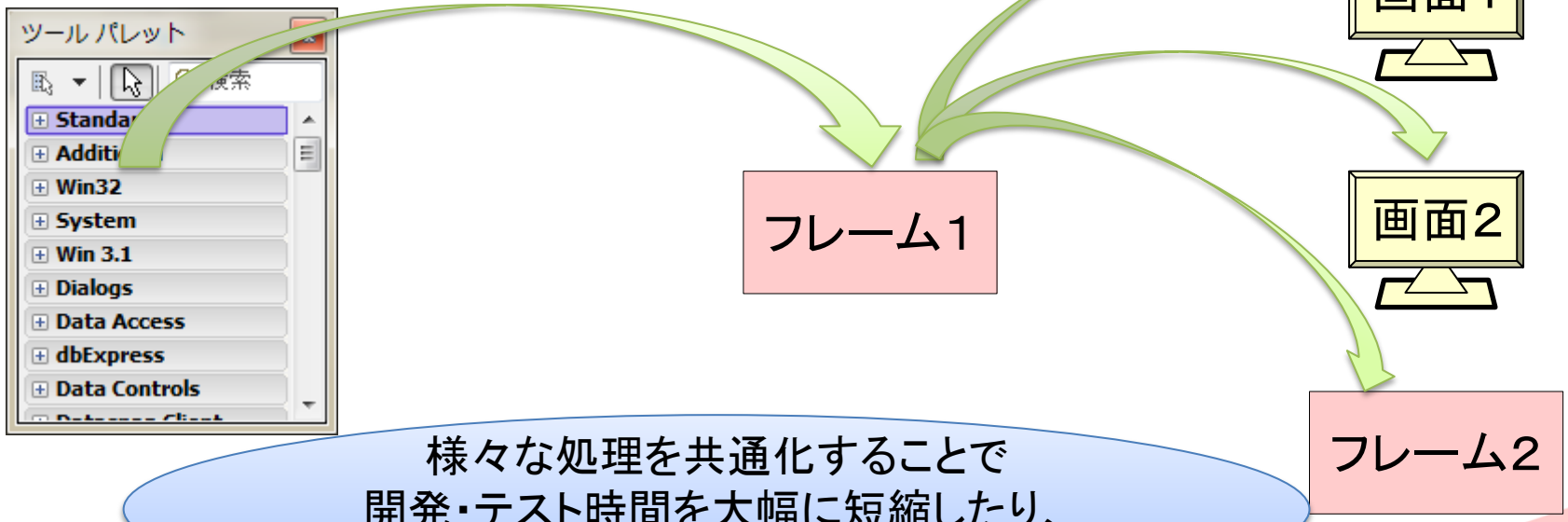
■ 処理の共通化

- 処理を共通化する方法
  1. 共通関数
     - ユニットなどを使用し、処理のみを共通化する方法
     - 複数画面で使用される処理を共通化する場合に最適
     - ex) 明細データのCSV出力、Iniファイルの参照、etc…
  2. カスタムコンポーネント
     - コンポーネントに、処理を記述することで共通化する方法
     - コンポーネント独自の処理を共通化する場合に最適
     - ex) Enterキーの押下によるフォーカス移動、
       エラー項目の背景色反転、etc…
  3. フレーム
     - 複数コンポーネントの処理や制御などを共通化する方法

フレームを利用した処理の共通化をご紹介

■画面共通処理のフレームによる部品化

- フレームとは？
  - 複数のコンポーネントを一つにまとめる為のコンポーネント
  - グループ単位でフォームや他のフレームへ配置することができる



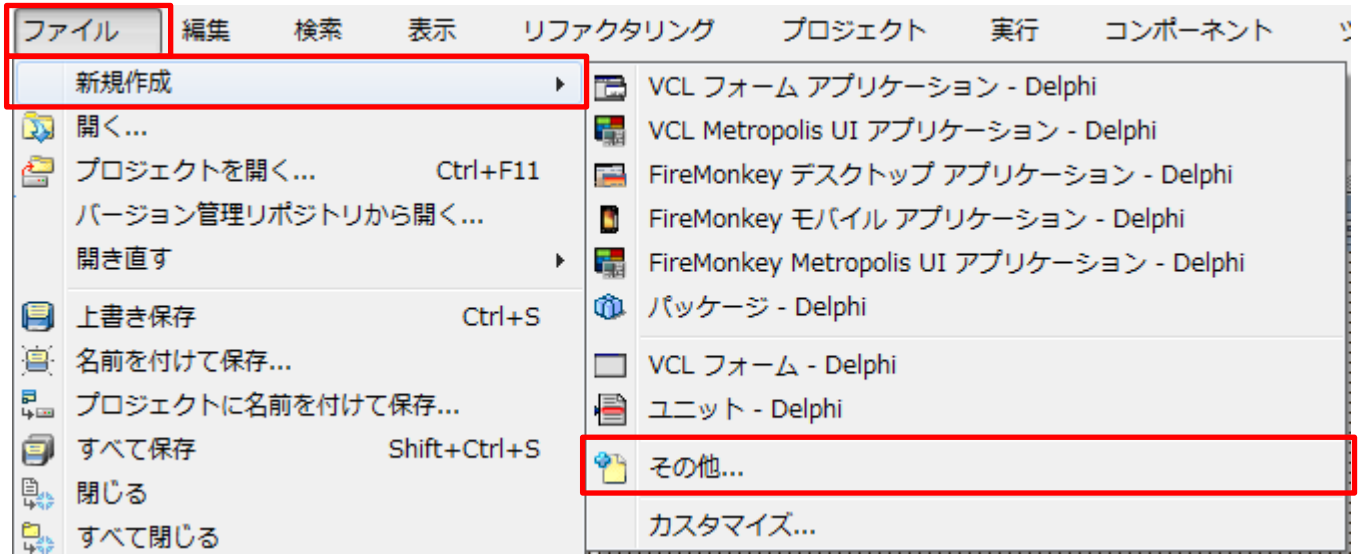
様々な処理を共通化することで
開発・テスト時間を大幅に短縮したり、
保守性を大幅に向上させることができる

フレームの使用方法をご紹介

画面共通処理のフレームによる部品化

①フレームの生成

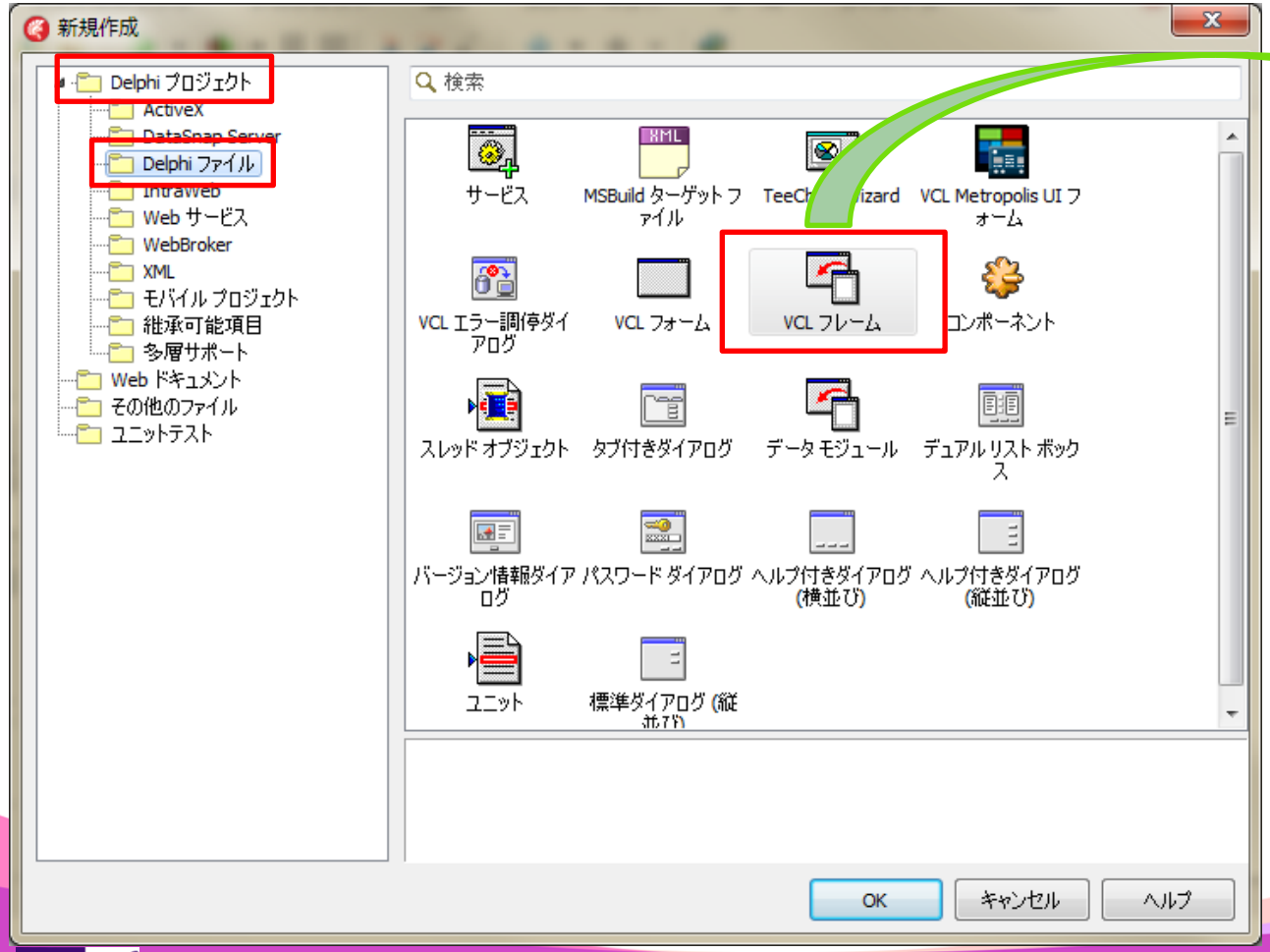
- ツールバーより[ファイル｜新規作成｜その他...]を選択

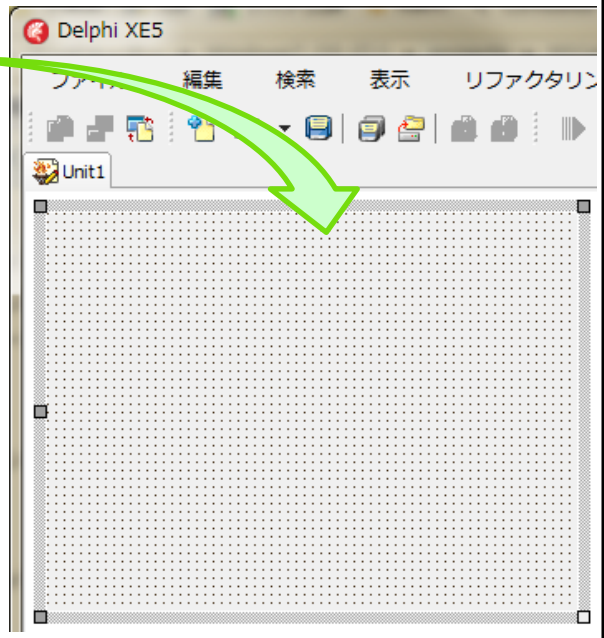


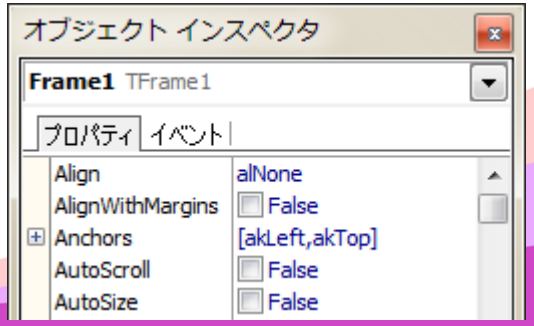
■ 画面共通処理のフレームによる部品化

①フレームの生成

- ダイアログより[Delphiプロジェクト | Delphiファイル]のVCLフレームを選択



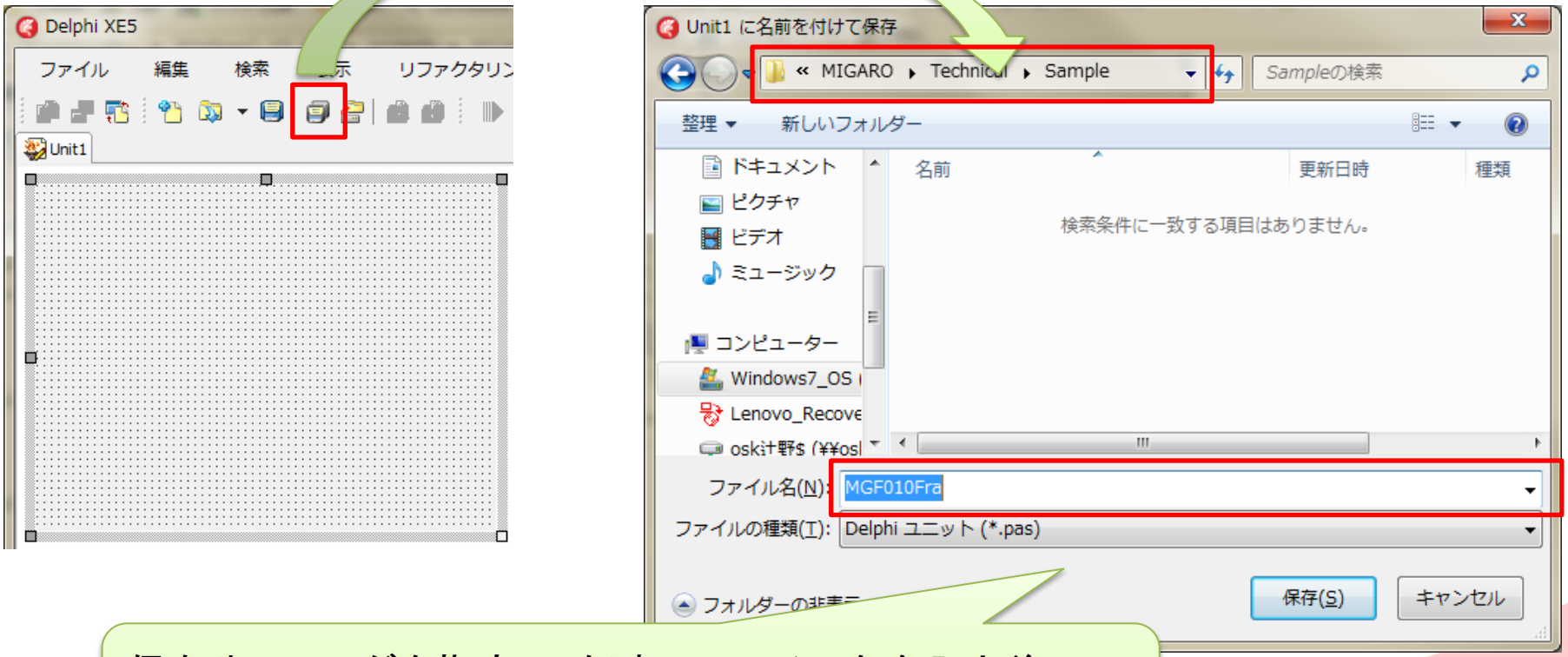




画面共通処理のフレームによる部品化

①フレームの生成

- フレームの保存

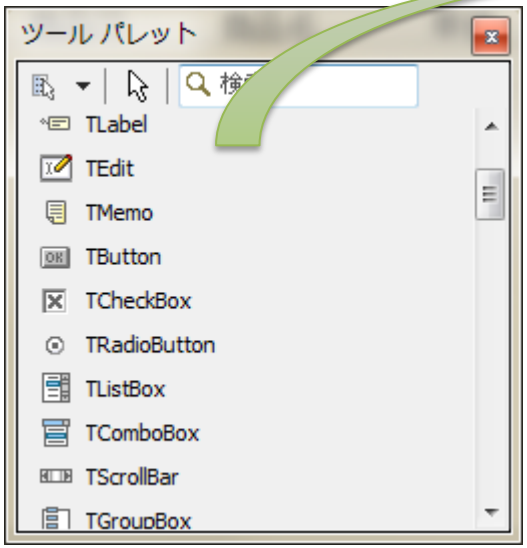


保存先フォルダを指定し、任意のファイル名を入力後、
保存ボタンを押下する

画面共通処理のフレームによる部品化

②フレームへの処理の記述

- フレームへコンポーネントの配置や処理の記述



フォームの場合と同様の方法でコンポーネントを
配置、プロパティやイベントへの処理の記述を行う

画面共通処理のフレームによる部品化

②フレームへの処理の記述

フレーム生成時の処理を記述したい場合

```
  public
    constructor Create(AOwner: TComponent); override;
-------------------------------------------------------
constructor TfraMGF010.Create(AOwner: TComponent);
begin
  inherited;

  // フレーム生成時の処理を記述
end;
```

フレーム解放時の処理を記述したい場合

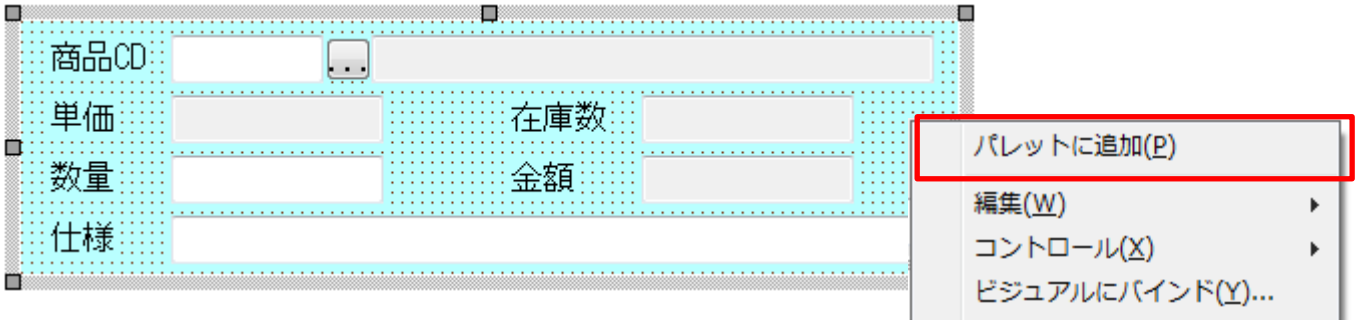
```
  public
    destructor Destroy; override;
-----------------------------------
destructor TfraMGF010.Destroy;
begin
  // フレーム解放時の処理を記述

  inherited;
end;
```

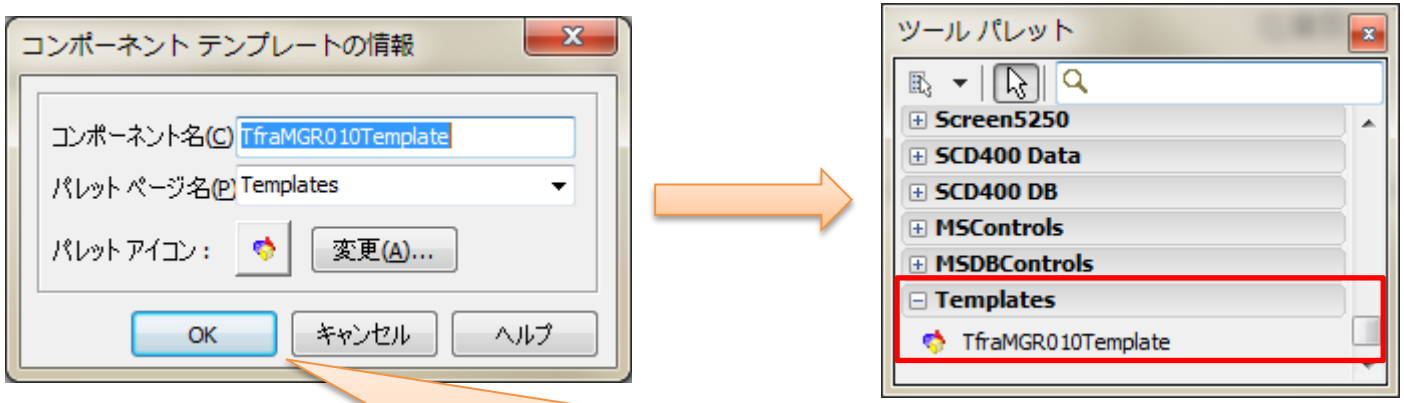
inheritedにて、フレーム自体のメモリが解放される為、個別の処理はinheritedの前に記述する

画面共通処理のフレームによる部品化

③フレームの利用

- ツールパレットへの追加



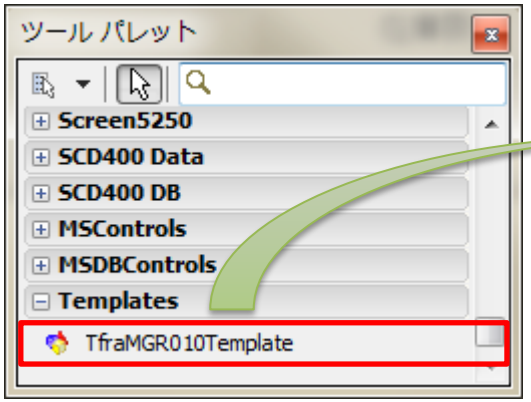


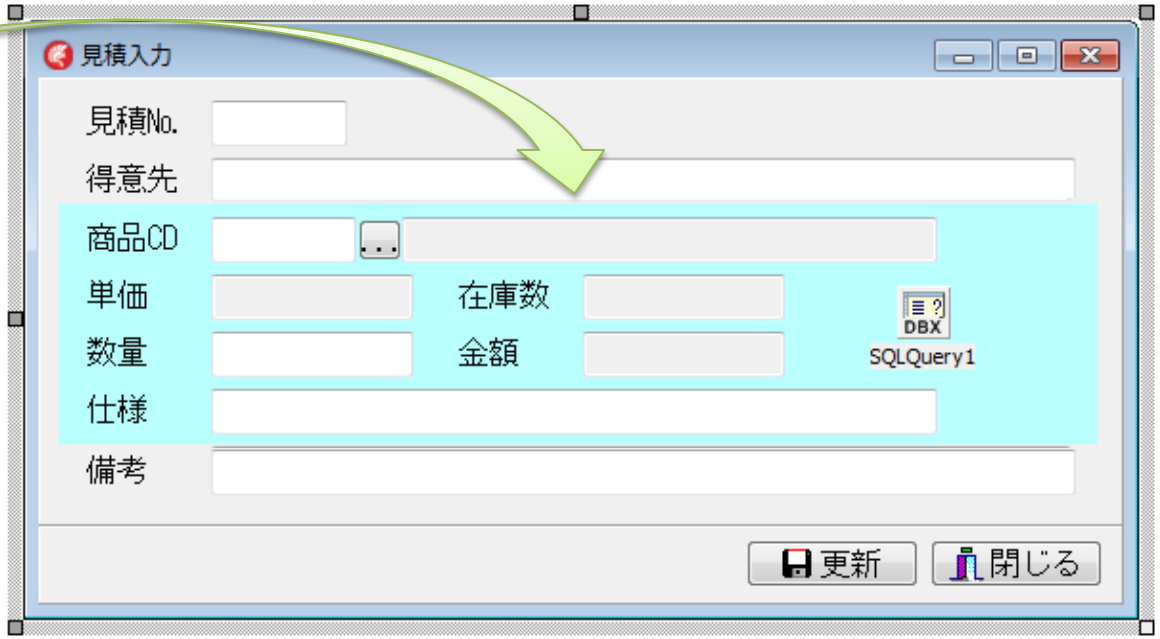
コンポーネント名　：　任意のコンポーネント名
パレットページ名　：　追加先のページ名
パレットアイコン　：　任意のアイコン

画面共通処理のフレームによる部品化

③フレームの利用

- ツールパレットより画面への配置

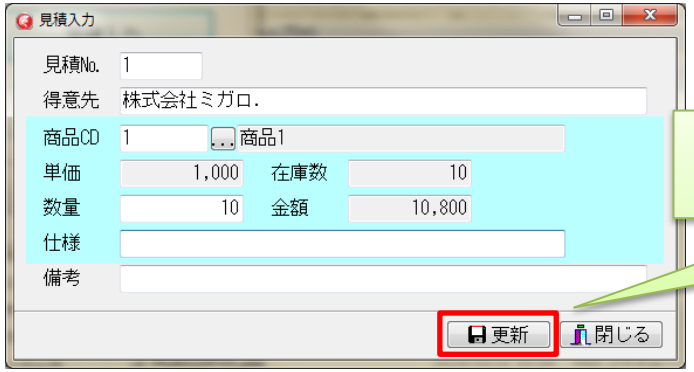




画面共通処理のフレームによる部品化

③フレームの利用

- フレームの項目より値を取得する方法



フレームの項目値をファイルに更新

更新ボタンのOnClickイベント

```
procedure TfrmMGR010.btnUpdateClick(Sender: TObject);
begin
  // フレームの項目よりファイルを更新
  SQLQuery1.ParamByName('SHCD').AsInteger := StrToIntDef(TfraMGF0101.edtSHCD.Text, 0);  // 商品CD
  SQLQuery1.ParamByName('SURY').AsInteger := StrToIntDef(TfraMGF0101.edtSURY.Text, 0);  // 数量
  SQLQuery1.ParamByName('SIYO').AsString  := TfraMGF0101.edtSIYO.Text;                  // 仕様

  // ファイル更新
  SQLQuery1.ExecSQL;
end;
```

配置直後のフレームはNameプロパティに「クラス名　＋　連番」が設定されている

■ まとめ

1. ユーザビリティの向上

1－1. 同時に複数画面を起動する方法

- 複数の作業を同時に行うことができる
  - ➢ 作業効率の向上

1－2. グリッド操作時の自動フォーカス制御方法

- 入力可能セルのみへのカーソル移動
  - ➢ スピーディな入力操作を実現
- TDBGridでも同様の操作感を実装可能

2. 開発効率、保守性の向上

2－1. 画面共通処理のフレームによる部品化

- 一連の処理を共通化
  - ➢ 開発時間の短縮、保守性の向上
- VCLのみでなくFireMonkeyでも同様の手順で実装可能

ご静聴ありがとうございました