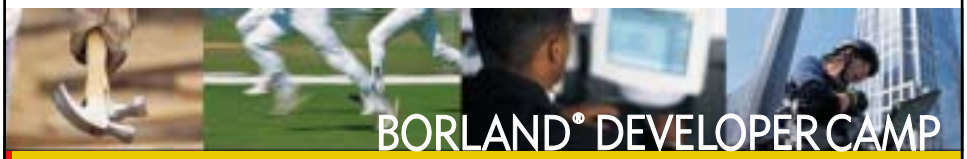
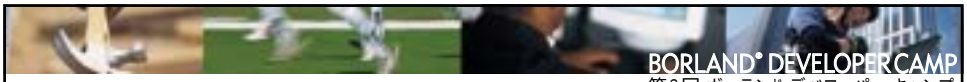


「パターン化されたロジックのコンポーネント化」

株式会社シーソフト
代表取締役社長
内山康広

Borland



UI部品だけでなく、ロジックも再利用するアーキテクチャの採用で、ソフトウエアをよりスピーディに開発することができるはずです。
C++Builderを用いて開発した SEAXERコンポーネントライブラリを例に、残業のない開発スタイルに挑戦した開発事例を技術的側面から紹介します。

Borland

講演者プロフィール

- １９８５年よりソフトウエア開発業務に従事
- CAD開発会社、システムハウスを経て１９９２年独立
- 金融、制御、音響、通信、画像処理などの開発実績
- BASIC、アセンブラ（CISC/RISC）、C、C++、Perl、Java...
- ニフティ他、各種フォーラムを「ゆーち」のハンドル名で攪乱
- 技術本はある程度読んでいるが、物覚えが悪い
- 肩書きは社長だが、職人。
- ４６歳。老体に無知。

技術動向の流れについて

- スパゲティプログラム
- サブルーチン化
- 構造化プログラミング
- オブジェクト指向設計･開発
- デザインパターン
- PM、設計プロセス、要求仕様プロセス

開発意識から

経験と反省から必然的に進化

設計意識へ

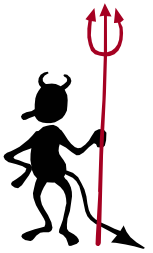
でも．．．

なくならない

デスマーチプロジェクト

(T^T)

なぜ？



現場の声 (周囲のことに関して)

- 仕様があいまい、固まっていない。
- 数社まじりで混乱。
- ユーザがわがまま。
- 短納期。
- デキル人材が少ない。
- 下っ端が未熟。
- 教育システムがなっていない。
- 優れたソースを読め！？(UNIXカーネル)
- 腐ったソース流用。(秘伝のタレ！)
- コメントがない、ドキュメントがそろってない。
- あの言語はイヤだ、嫌いだ、やったことがない。

他のセイにしすぎ・・・

政治的背景

技術的背景

現場の声（自分のことに関して）

- プロジェクト開始から中盤まで
  →自分がやった方が早い

メンバーに分散させることが重要！

- 中盤から終了まで
  →もう一度最初からやれたら、次はもっとうまく書くことがデキル

プロフェッショナルなら完成されてる？

この繰り返しばっかりでは？
せめて、他人のせいにばかりはしないようにしましょうね。

BORLAND® DEVELOPER CAMP
第3回 ボーランド デベロッパー キャンプ

残業でどんなことしてますか？

- 障害対応
- 仕様変更対応

ほとんど、このふたつに集約されませんか？

深みにはまる要因は？

要因その1：シーケンスの不統一

- 表現の統一
  「中止」ボタンと「キャンセル」ボタン

修正はわりとカンタン！

- エラー処理
  true/false
  SetLastError();
  例外 throw()
  BOOL な複数のバリエーション
  abort だね

上位と下位のリンク大変。

- ログ出力
  聞いてねぇよ（とか）

ロジックの変更を要する
「当然」も比較的多くある

意思の疎通

BORLAND® DEVELOPER CAMP
第3回 ボーランド デベロッパー キャンプ

要因その2：過度の期待

- 工数の見込み「なんとかなるだろう」（ボリューム読めてない？）
- 徹夜と休日出勤で「なんとかなるだろう」（体育会系ですか？）
- これだけ頭数揃えれば「なんとかなるだろう」（スタッフは優秀？）

スタッフの方々は同じように育っていますか？

技術習得に個人差がありませんか？

UML、PM

OOPS、
DesignPattern

C/C++
ポインタ？

要因その3：多量のコピーソース

- もちろん、ソース流用。
- 顧客別の対応→さらに展開。
- 顧客別、シリーズ別に開発環境がコピーされている。
- 一カ所のバグを『水平展開』。
- 確認する方法が少ない。

なんですかそれ？

なおってないじゃん！

DRY：Don't Repeat Yourself
残業時間＝（コピーコード）^{2} に比例

要因その4：コピペ推奨スタンダード

- Windows3.1：プログラミングスタイルで推奨してました・・・
- 誰かが作って、動いたサブルーチンをこぞって利用する。

データベースのテーブルに登録するにはどうしたらいい？

こんなかんじぃ

(例1) データベースレコードの挿入

```
1  bool TXxxx::Insert( TYyyy *_Data ){
2     bool rc = false;
3     rc = StartTransaction();                          } トランザクション開始
4
5     StoredProc1->StoredProcName = "ストアドプロシージャの名前";
6     StoredProc1->Prepare();
7
8     StoredProc1->ParamByName("@パラメータ名1")->AsString   = _Data->メンバ名1;   } SQLパラメータに
9     StoredProc1->ParamByName("@パラメータ名2")->AsInteger = _Data->メンバ名2;   } 値を渡す
10          :
11    rc = ExecProcedure();                             } Exec() を実行させる
12
13    rc = ( rc == true ? ) Commit() : RollBack();      } トランザクション終了
14    return rc;
15 }
```

Database, テーブル, SQL文／ストアドプロシージャ,パラメータの異なる同じ手順のコードだらけ！

BORLAND® DEVELOPER CAMP
第3回 ボーランド デベロッパー キャンプ

(例2) モーダルダイアログのOKボタン処理

```
1  TFormXX::OnOkButton?Click( Tobject *Sender )
2  {
3     try{
4        int    Year    = EditYear->Text.ToIntDef( 0 );
5        int    Month   = EditMonth->Text.ToIntDef( 0 );
6        int    Day     = EditDay->Text.ToIntDef( 0 );
7        if(CheckDate( Year, Month, Day )== true ){
8           UpdateData( Year, Month, Day );
9        }else{
10          throw( Exception( “BAD” ) );
11       }
12    }catch(…){
13       ;
14    }
15 }
```

コントロールから値取得

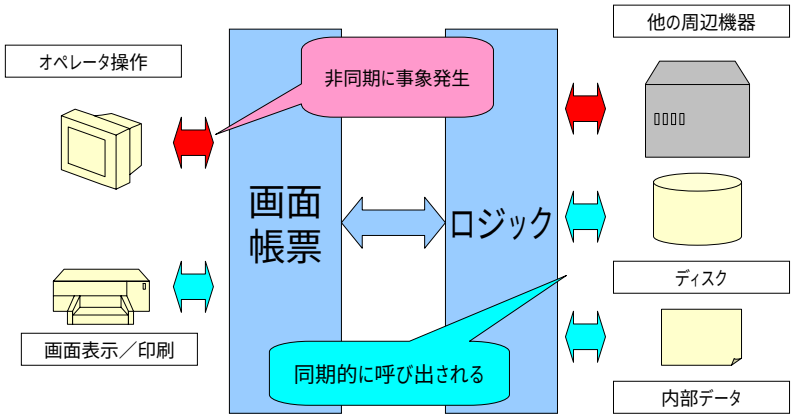
うまくいけば登録

失敗すればエラー処理

値をルールに基づいてチェック

画面のハンドラってのは、似たようなコードだらけではないか！？

一般的なソフトウエア構造



一般的なソフトウエア・アクション

ヘルプ	戻る
中止	次へ
終了	先頭
確定	末尾
印刷	ジャンプ
検索	追加
前ページ	挿入
次ページ	削除
前項	コピー
次項	ペースト
更新	切り取り

それぞれやることって毎回同じようなことではないでしょうか？

とりあえず、コピーをなくせないかな？

- 手続き指向
  ↓
- 構造化手法
  ↓
- オブジェクト指向
  ↓

ロジックの部品化！？

どうやって？

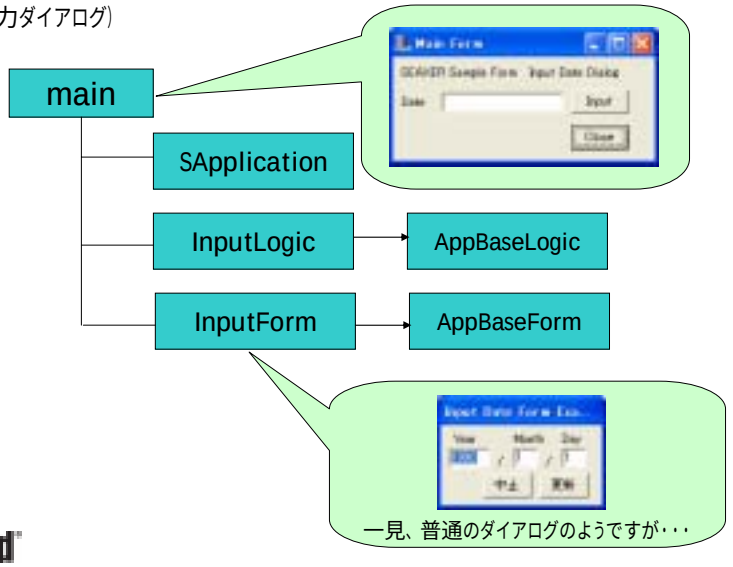
ヒント？

- 第4世代言語（これ知りませんけど・・・）
- ミドルウエア（そういったものなのでしょうか？）
- ドキュメントビュー
- オブジェクト指向
- デザインパターン
- アスペクト指向

別に斬新な発明じゃありません。
これらを組み合わせただけです。
(´◇`;

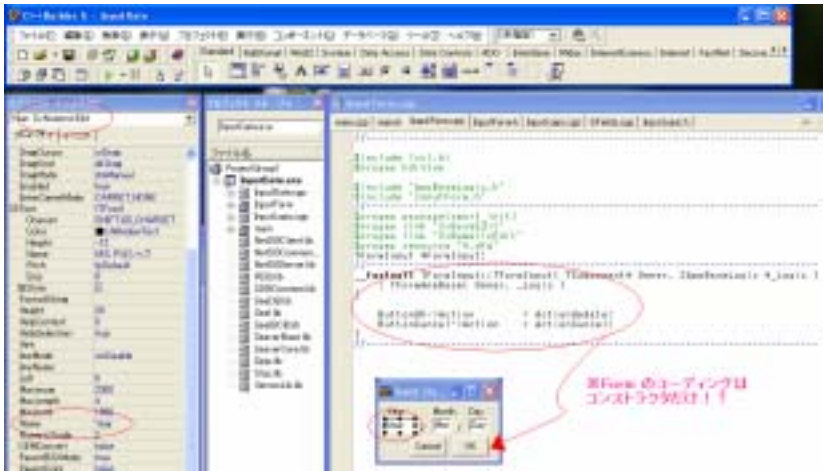
ＳＥＡＸＥＲの実装例

（日付入力ダイアログ）



BORLAND® DEVELOPER CAMP
第3回 ボーランド デベロッパー キャンプ

InputForm ユニットの実装



InputLogic ユニットの実装

InputLogic.h

```
 1 #include <classes.hpp>
 2 #include <Forms.hpp>
 3
 4 #include "AppBaseLogic.h"
 5
 6 //---------------------------------------------------
 7 class SInputLogic : public SAppBaseLogic
 8 {
 9   static const AnsiString Year;
10   static const AnsiString Month;
11   static const AnsiString Day;
12
13    AnsiString      FDateString;
14
15 public:
16    __fastcall SInputLogic( SApplication
      *_Application );
17    virtual __fastcall ~SInputLogic();
18
19    static int GetLastDay( int _Year, int _Month );
20
21 protected:
22    virtual bool __fastcall AtCreate();
23    virtual bool __fastcall AtUpdate( TObject
      *_Sender );
24    virtual bool __fastcall Check();
25
26 public:
27    __property AnsiString   DateString =
      { read=FDateString };
28 };
```


BORLAND® DEVELOPER CAMP
第3回 ボーランド デベロッパー キャンプ

InputLogic.cpp

```
#include <vcl.h>
#pragma hdrstop

#include <SysUtils.hpp>
#include "InputLogic.h"
#include "SApplication.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)

// フォームで使用する連動フィールド名を設定
const AnsiString SInputLogic::Year  = "Year";
const AnsiString SInputLogic::Month = "Month";
const AnsiString SInputLogic::Day   = "Day";

__fastcall SInputLogic::SInputLogic( SApplication *_Application )
    : SAppBaseLogic( _Application )
{
    // フィールド定義ＸＭＬファイル名を設定
    AnsiString xmlFilePath = _Application->IniFile-
        >ReadString( "SApplication", "XMLFilePath", "" );
    ParseXmlUIField( ExcludeTrailingBackslash( xmlFilePath ) +
        "¥¥InputDate.xml" );
}

__fastcall SInputLogic::~SInputLogic()
{
}

// 初期設定
bool __fastcall SInputLogic::AtCreate()
{
    unsigned short year, month, day; // 現在の日付を取得
    TDate   date = TDateTime::CurrentDate();
    date.DecodeDate( &year, &month, &day );

    // 画面にデフォルトとして表示する
    Fields[ Year ]     = year;
    Fields[ Month ]    = month;
    Fields[ Day ]      = day;

    return true;
}
// 更新時の処理
bool __fastcall SInputLogic::AtUpdate( TObject *_Sender )
{
    unsigned short year, month, day;
    year    = Fields[ Year ];
    month   = Fields[ Month ];
    day     = Fields[ Day ];

    // 取得した値をプロパティに格納
    TDate   date( year, month, day );

    FDateString = date.FormatString( "yyyy/mm/dd" );
    return true;
}
```

```
// 値を検証する
bool __fastcall SInputLogic::Check()
{
    unsigned short year, month, day;

    // 一般的な年月日チェック
    year    = Fields[ Year ];
    month   = Fields[ Month ];
    day     = Fields[ Day ];

    int lastDay = GetLastDay( year, month );
    if( 0 < lastDay )
    {
        if( ( 0 < day ) && ( day <= lastDay ) )
        {
            return true;
        }
    }

    return false;
}
```

```
// 最終日の取得
int SInputLogic::GetLastDay( int _Year, int _Month )
{
    int lastDay = -1;

    // 有効な日付かチェック
    switch( _Month )
    {
      case 1: case 3: case 5: case 7:
      case 8: case 10: case 12:
        lastDay = 31;
        break;
      case 4: case 6: case 9: case 11:
        lastDay = 30;
        break;
      case 2:
        if( ( ( _Year % 4 == 0 ) && ( _Year % 100 != 0 ) ) || ( _Year % 400 == 0 ) )
        {
            lastDay = 29;
        }
        else
        {
            lastDay = 28;
        }
        break;
      default:
        break;
    }

    return lastDay;
}
```


実装方法の違い

【従来の設計方法】
- フォームにコンポーネントを貼り付けて設計(Tform 派生クラス)
- 初期表示を FormShow() イベントなどに記述
- OKボタンのイベントルーチン OnButtonOkClick() を実装し、
- コンポーネントの値を取得
- 取得値の検証（チェック）
- データの格納

【SEAXERを利用した設計方法】
- フォームにコンポーネントを貼り付けて設計（SAppBaseForm 派生クラス、独自コンポーネント）
- Logic クラスをFormとは別に生成（SAppBaseLogic 派生クラス）
- 初期表示関数 AtCreate() を実装
- 更新時の処理 を AtUpdate() に実装
- フィールドデータの検証（チェック）を Check() に実装

※フィールド上のデータは、Fields[] プロパティで取得できる。

大して変わらないか、むしろややこしそうな気がしますか？

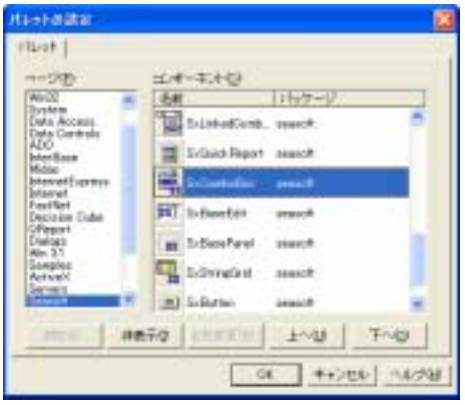
Logicを抽象化して実際の機能と分離することで、ソフトウエアの再利用性が確実に高まります！

アプリケーションの手続き（ロジック）要件は、その都度、固有のものです。
抽象化ロジックの動きを継承した派生ロジッククラスでその動きを定義すれば、要件（ロジック）ごとの狭い範囲での開発およびデバッグに集中して開発ができる（他のさまざまなシステム要件に惑わされにくくなる）ため、ソフトウエアの品質が高まります。
また、将来的なロジックの仕様変更の際にも、特定のクラスに限定した改修になることが多くなります。（広範囲の変更も少ない箇所で改修できるようになりやすい）

Formコードを書かずになぜ値の取得ができるか

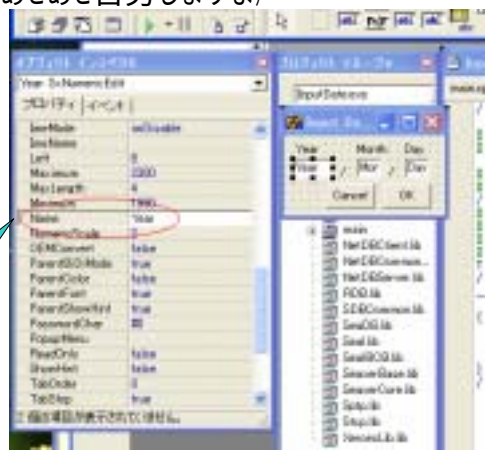
1．コンポーネントの提供

SEAXERを利用するフォームでLogicと連動させるためには、専用のコンポーネントを使って設計する



2. 設計時のコンポーネントにわかりやすい名前を付ける

(Edit1でもかまいませんけど、あとあと苦労しますよ)



「年」にはYear
「月」にはMonth
「日」にはDay
をネーミング

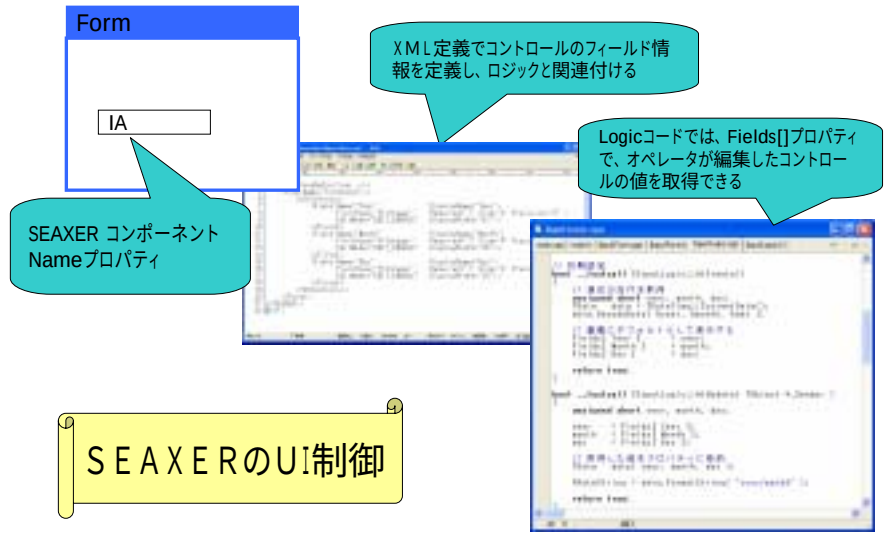
3. フィールドの属性をフォーム別にXML定義する

InputDate.xml
ファイルの内容

```
<?xml version="1.0" encoding="Shift_JIS" ?>

<SEAXER>
  <!-- FormDefinition -->
  <Form Name="FormInput">
    <UIControls>
      <Field Name="Year"          DisplayName="年"
          FieldType="ftInteger"   Required="1" Size="4" Precision="0"
          EditMode="EDIT_ENABLE"  DisplayWidth="41">
      </Field>
      <Field Name="Month"         DisplayName="月"
          FieldType="ftInteger"   Required="1" Size="4" Precision="0"
          EditMode="EDIT_ENABLE"  DisplayWidth="25">
      </Field>
      <Field Name="Day"           DisplayName="日"
          FieldType="ftInteger"   Required="1" Size="4" Precision="0"
          EditMode="EDIT_ENABLE"  DisplayWidth="25">
      </Field>
    </UIControls>
  </Form>
</SEAXER>
```

この定義内容が Form のコントロールと
Logic上のフィールド変数を関連づける！



実装した連動アクション

内部処理連動

```
virtual bool __fastcall AtCreate();
virtual bool __fastcall AtDestroy();
virtual bool __fastcall AtCreateCache( TObject *_Sender );
virtual bool __fastcall AtReadCache( TObject *_Sender );
virtual bool __fastcall AtDeleteCache( TObject *_Sender );
virtual bool __fastcall AtCreateReportCache( TObject *_Sender );
virtual bool __fastcall AtReadReportCache( TObject *_Sender );
virtual bool __fastcall AtDeleteReportCache( TObject *_Sender );
virtual bool __fastcall AtDisplayFields( TObject *_Sender );
virtual bool __fastcall AtInitializeGrid( TObject *_Sender );
virtual bool __fastcall AtDisplayGrid( TObject *_Sender );
virtual bool __fastcall AtTerminateGrid( TObject *_Sender );

virtual bool __fastcall AtBeforePrint( TObject *_Sender );
virtual bool __fastcall AtAfterPrint( TObject *_Sender );
virtual bool __fastcall AtNeedData( TObject *_Sender, int _Line );
virtual bool __fastcall AtReportHeader( TObject *_Sender );
virtual bool __fastcall AtReportTitle( TObject *_Sender );
virtual bool __fastcall AtReportColumnHeader( TObject *_Sender );
virtual bool __fastcall AtReportDetail( TObject *_Sender, int
    _Line );
virtual bool __fastcall AtReportSummary( TObject *_Sender );
virtual bool __fastcall AtReportFotter( TObject *_Sender );
virtual bool __fastcall PageBreak( int _PageNu
```

印刷処理連動

```
virtual bool __fastcall AtFormCreate( TObject *_Sender );
virtual bool __fastcall AtFormShow( TObject *_Sender );
virtual bool __fastcall AtFormDestroy( TObject *_Sender );
virtual bool __fastcall AtCloseQuery( TObject *_Sender,
                bool& _CanClose );
virtual bool __fastcall AtHelp( TObject *_Sender );
virtual bool __fastcall AtPrint( TObject *_Sender );
virtual bool __fastcall AtSearch( TObject *_Sender );
virtual bool __fastcall AtCancel( TObject *_Sender );
virtual bool __fastcall AtUpdate( TObject *_Sender );
virtual bool __fastcall AtExit( TObject *_Sender );
virtual bool __fastcall AtPrevPage( TObject *_Sender );
virtual bool __fastcall AtNextPage( TObject *_Sender );
virtual bool __fastcall AtAction( TObject *_Sender );

virtual bool __fastcall AtPrevGroup( TObject *_Sender );
virtual bool __fastcall AtNextGroup( TObject *_Sender );
virtual bool __fastcall AtPrev( TObject *_Sender );
virtual bool __fastcall AtNext( TObject *_Sender );
virtual bool __fastcall AtTop( TObject *_Sender );
virtual bool __fastcall AtBottom( TObject *_Sender );
virtual bool __fastcall AtJump( TObject *_Sender );
virtual bool __fastcall AtInsert( TObject *_Sender );
virtual bool __fastcall AtAdd( TObject *_Sender );
virtual bool __fastcall AtDelete( TObject *_Sender );
virtual bool __fastcall AtCopy( TObject *_Sender );
virtual bool __fastcall AtPaste( TObject *_Sender );
virtual bool __fastcall AtCut( TObject *_Sender );
```

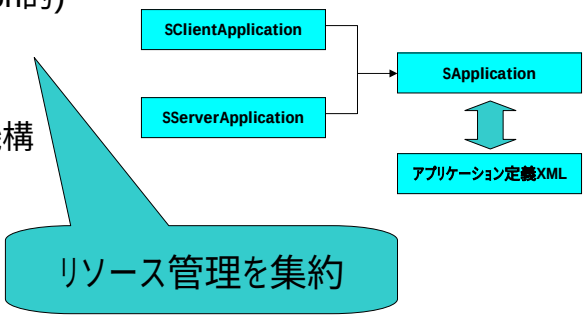
操作処理連動

連動コントロール

- 帳票フォーム
  QuickReport の QRLabel::Name プロパティ
- データベーステーブル
  SQL文、ストアドプロシージャのパラメータとロジック内部変数の連動
- グリッド(TStringGrid)表示
  仮想キャッシュと入出力による、グループ処理・ページ切替処理など

アプリケーションクラス機能

- クライアント／サーババージョン
- 各種データベースエンジン吸収(InterBase,Oracle,SQLServer,)
- ネットワーク通信(TCP/UDP)
- シリアル通信(Option的)
- ログ管理
- 情報暗号化
- 分散データベース機構

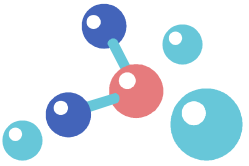


部品化で楽をしようという過去の試み

- サブルーチンライブラリの蓄積と再利用
- クラスライブラリの蓄積と再利用
- バイナリ互換性を持った再利用可能なコンポーネント（COM、ActiveX）

ガチガチに固めた部品化は、融通が利かなくなってしまい、再利用性を低くする

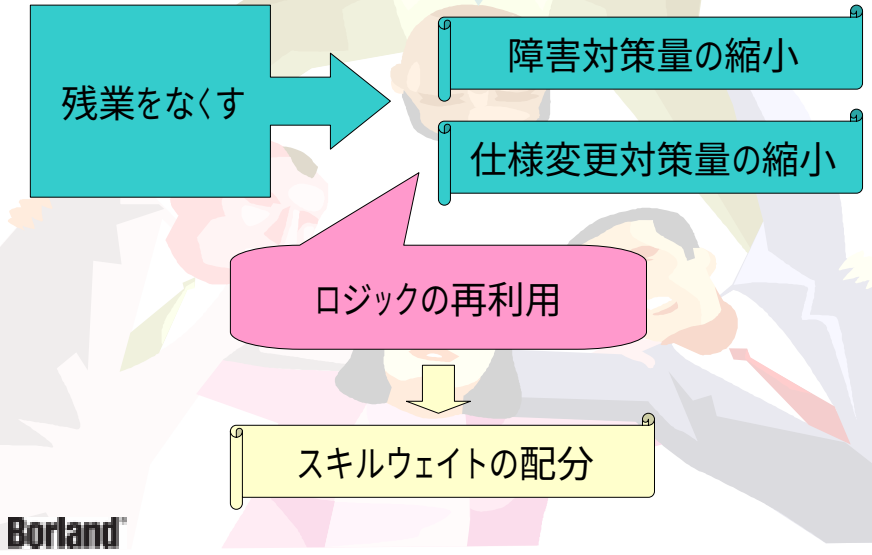
再利用可能な部品化に必要な条件とは？



- データを拡張できること
  →派生クラスで拡張
- ロジックを拡張できること
  →派生クラスでメソッドを追加する
  →メソッドをオーバーライドする→別の部品になってしまう

コアなメソッドの多くは、単純な入出力であることが多い。
→仮想関数による拡張をサポート

SEAXER の実装コンセプト



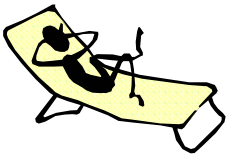
Borland

BORLAND® DEVELOPER CAMP
第3回 ボーランド デベロッパー キャンプ

- スタッフの精神的（スキルを超えた）負担を軽減する
- スタッフの体力的負担（残業）を軽減する

↓

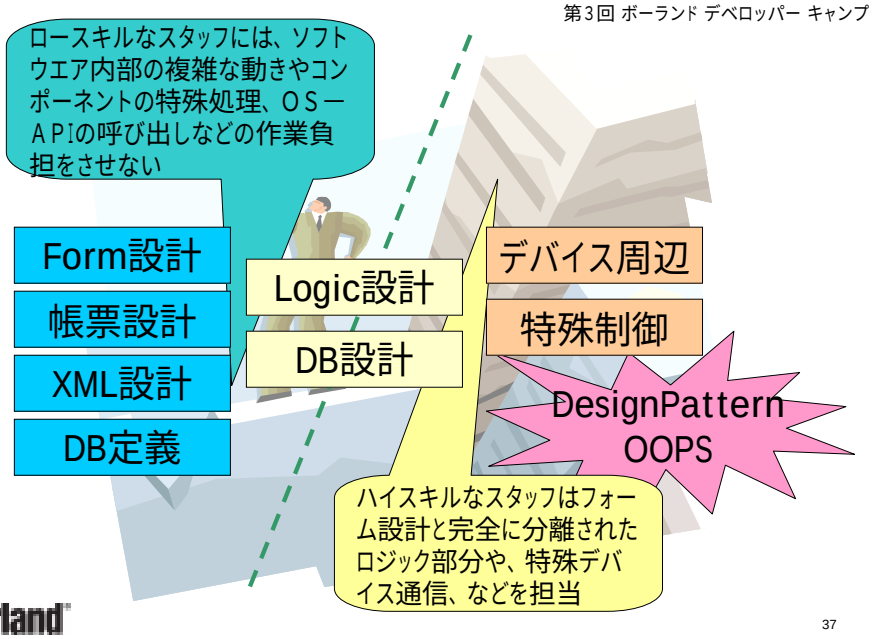
- スキルにあわせたソフトウエア開発階層の分離



・画面の動きに連動するコードをLogicへ切り離すことで、画面設計作業と内部処理を分離
・分離した両者の共通認識をXML定義内容において共有させた
・表記の変更、レイアウトの変更などの固定的で変更の多い部分をXMLから設定できるように
・SQL文を代表とするデータベース処理の多くをＸＭＬ定義に移動
※特に WHERE 節の編集だけで済んでしまう仕様変更には効果的
・QuickReport による印字サポート
・グリッド表示に代表されるテーブル構造データとそのグルーピング処理までの範囲をカバー

Borland

本文書の一部または全部の転載を禁止します。本文書の著作権は、著作者に帰属します。



参考1：基底クラスの内部（Update処理）

Logic クラスの定義（抜粋）

```
virtual bool __fastcall Check();

virtual bool __fastcall CheckSearch();
virtual bool __fastcall CheckPrint();
virtual bool __fastcall CheckUpdate();
virtual bool __fastcall CheckDo();
virtual bool __fastcall CheckDelete();

virtual bool __fastcall AtCancel
   ( TObject *_Sender );
virtual bool __fastcall AtUpdate
   ( TObject *_Sender );
```

BaseFormクラス：：ActionUpdate（）の内部（抜粋）

```
{
        PreUpdate();

        FLogic->GetFieldForm( this );

        FLogic->CheckUpdate();

        bool rc = FLogic->AtUpdate( this );

        FLogic->SetFieldForm( this );

        PostUpdate( rc );
}
```

UI入力値を転送

値を検証

値の確定手続き

確定値の整形表示

所定の動作をするようにできているが、クラスを公開し、仮想関数としてカスタマイズすることでアプリケーションに固有の処理を実装できる
（固有の特殊要件なら、従来通りそのまま書けばOK）

参考2：データベース処理コードの切り離し

```
<Database>
   <SQL Name="SELECT"
      Text="SELECT
                Code
               ,LedgerTitle
               ,LengthWidth
               ,Field01
               ,Field02
               ,Field03
               ,Used
               ,InsertDate
               ,UpdateDate
            FROM SOMS.BarcodeManageMST
            WHERE Code  = :Code
            ORDER BY Code" >
   </SQL>
   <SQL Name="SELECT_MAX_CODE"
      Text="SELECT MAX( Code )          AS MaxCode
           FROM  SOMS.BarcodeManageMST " >
   </SQL>
```

Logic 内部で、TParams やメモリブロックに連動

```
<SQL Name="INSERT"
   Text="INSERT INTO SOMS.BarcodeManageMST(
            Code
           ,LedgerTitle
           ,LengthWidth
           ,Field01
           ,Field02
           ,Field03
           ,Used
           ,InsertDate
           ,UpdateDate
           ) VALUES (
            :Code
           ,:LedgerTitle
           ,:LengthWidth
           ,:Field01
           ,:Field02
           ,:Field03
           ,:Used
           ,GETDATE()
           ,GETDATE()
           )" >
   </SQL>
</Database>
```

Logic 内部の Field データと連動

参考3：帳票

- QuickReport を使って設計

「伝票番号」印刷用に設計された QRLabel コントロールの Name プロパティに、CheckNumber の名前を付けておく。

- XMLファイルに定義された帳票情報 (抜粋)

```
<!-- CustomerLedger ReportDefinition -->
  <Report Name="CustomerLedgerReport">

      <Field Name="ReportTitle"          FieldType="ftString"   Required="0" Size="13" Format="" ></Field>
      <Field Name="Page"                 FieldType="ftWord"     Required="0" Size="2"  Format="" ></Field>
      <Field Name="ReportPages"          FieldType="ftWord"     Required="0" Size="2"  Format="" ></Field>
      <Field Name="SelectCustomer"       FieldType="ftString"   Required="0" Size="61" Format="" ></Field>

      <Field Name="CheckDate"            FieldType="ftDate"     Required="0" Size="8"  Format="" ></Field>
      <Field Name="CheckNumber"          FieldType="ftFloat"    Required="0" Size="8"  Format="" ></Field>
      <Field Name="BranchNumber          FieldType="ftInteger"  Required="0" Size="4"  Format="" ></Field>
      <Field Name="DealType"             ieldType="ftString"   Required="0" Size="11" Format="" ></Field>
      <Field Name="CommodityName         ieldType="ftString"   Required="0" Size="41" Format="" ></Field>
      <Field Name="Amount"               e="ftInteger"  Required="0" Size="4"  Format="%.0n" ></Field>

      <Field Name="TotalSalesMoney"                 tCurrency" Required="0" Size="8"  Format="%.0n" ></Field>
      <Field Name="TotalConsumptionTax              Currency" Required="0" Size="8"  Format="%.0n" ></Field>
                                  :
                                  :
```

Logic は、画面やデータベースと連動する同一名の値を
内部で保持している。
印刷処理は、保持データを使って自動的に処理される

- Reportのコード

```
#pragma hdrstop

#include "CustomerLedgerReport.h"
#include "CustomerLedgerLogic.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
//#pragma link "AppCoreReport"
#pragma resource "*.dfm"
TReportCustomerLedger *ReportCustomerLedger;
//---------------------------------------------------------------------------
// コンストラクタ
//---------------------------------------------------------------------------
__fastcall TReportCustomerLedger::TReportCustomerLedger(TComponent* Owner, SAppCoreLogic *_Logic/*=NULL*/)
    : TQReportAppCore(Owner,_Logic)
{
}
//---------------------------------------------------------------------------
// 頁毎の印刷前処理
//---------------------------------------------------------------------------
void __fastcall TReportCustomerLedger::QuickRepStartPage(
      TCustomQuickRep *Sender)
{
    SCustomerLedgerLogic *logic = dynamic_cast<SCustomerLedgerLogic *>(FLogic);
    if( logic != NULL )
    {
                    // 頁位置をセット
                        logic->ReportFields[ "Page" ] = PageNumber;
    }
}
//---------------------------------------------------------------------------
```

- Logicのコード

```
//---------------------------------------------------------------------------
bool __fastcall SCustomerLedgerLogic::AtBeforePrint( TObject *_Sender )
{
    bool rc = true;

                        // 帳票の日付をセット
    ReportFields[ PrintDate   ] = DateToStr( Now() );

    // 帳票の頁数をセット
    PrintLineCount    = FCustomerLedgerTbl->Count;        // 印刷行数
    ReportFields[ ReportPages ] = ( PrintLineCount + ( FReportLines - 1 ) )
        / FReportLines;

    // タイトル
    ReportFields[ ReportTitle ] = "得意先元帳";

    // 抽出条件をセット
    TDate  startDate            = Fields[ StartDate ];          // 開始日
    TDate  endDate              = Fields[ EndDate   ];          // 終了日
    ReportFields[ StartDate ]   = startDate.FormatString( "yyyy/mm/dd");
    ReportFields[ EndDate   ]   = endDate.FormatString( "yyyy/mm/dd");
    Word year, month, day;
    DecodeDate( IncMonth( startDate, -1 ), year, month, day );

    return rc;
}
//---------------------------------------------------------------------------
bool __fastcall SCustomerLedgerLogic::AtNeedData( TObject *_Sender, int
{
    return ( _Line < PrintLineCount  ) ?true : false;
}
```

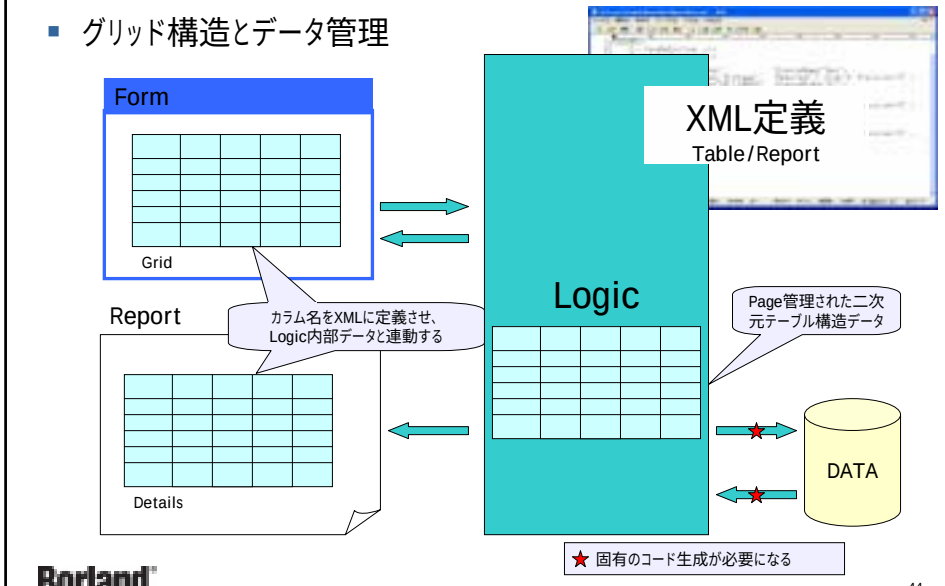
```
//---------------------------------------------------------------------------
bool __fastcall SCustomerLedgerLogic::AtPrintDetail( int _Line )
{
// 受注日, 取引区分, 商品名, 数量, 単価, 売上金額, 消費税額, 入金金額, 調整金額,
// 差引金額 は、ReportFields[ x ] = TableFields[ _Line ][ x ] にフィールド自動転送

    // 受注番号の特殊処理
    if( (AnsiString)TableFields[ _Line ][ CheckNumber ] != "" )
    {
        // 受注番号
        ReportFields[ CheckNumber ] = (AnsiString)TableFields[ _Line ][ CheckNumber ]
            + "-" +  (AnsiString)TableFields[ _Line ][ BranchNumber ];
    }
    else
    {
        // 受注番号
        ReportFields[ CheckNumber ] = "";
    }

    return true;
}
//---------------------------------------------------------------------------
bool __fastcall SCustomerLedgerLogic::PageBreak( int _PageNumber, int _Line )
{
    if( ( _Line < PrintLineCount ) && ( _PageNumber * FReportLines <= _Line ) )
    {
        return true; // 改頁
    }
    return false;
}
```

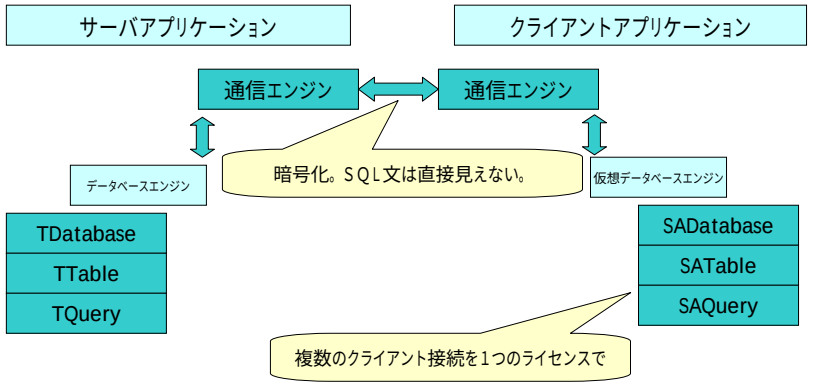
QRLabel にセットする処理の記述が不要

- グリッド構造とデータ管理



その他

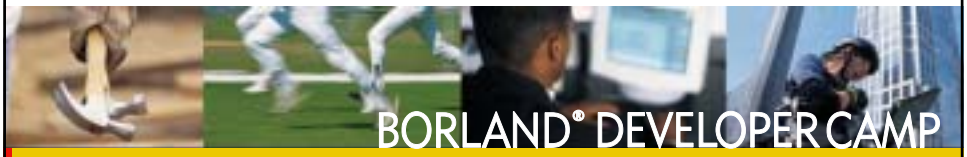
- 簡易分散データベースエンジン



- 汎用コンポーネントライブラリ
  SEAXER なしでも使用可能な各種コンポーネント(Edit,ComboBox,Button,Grid,Fkey,...)
- ダイナミックログビューワ連動
  稼働中のアプリケーションのログを別に起動したビューワにダイナミック接続
- ＯＥＭライセンスマネージャ
  作成アプリケーションのユーザ管理
- 簡易マスターメンテナンスプログラム
  テーブル一覧表示→個別レコード設定のアプリケーションを自動化
- パケットテンプレート
  アプリケーション固有の構造体 struct を定義するだけで通信パケットデータの構築を自動化
- 各種共通クラス
  アプリケーション重複起動、ファイルシステム関連、ダイアログ関連、オブザーバ関連など各種

今後の展望

- このアプローチは斬新か？
  実は、古くから提唱されている形式にすぎないのかどうか。
- 名前を付けるべきではないだろうか？
  「見える化」の先生方が言っておられますので・・・
- SEAXERは、受け入れられるのではないか？
  ものすごくコストかかってます。(T^T)
  市場に出せないかどうか、模索中です・・・



株式会社シーソフト：http://www.seasoft.co.jp
本稿に対するお問い合わせ：uchiyama@seasoft.co.jp
お気軽にお問い合わせください。

Thank you
ご静聴ありがとうございました