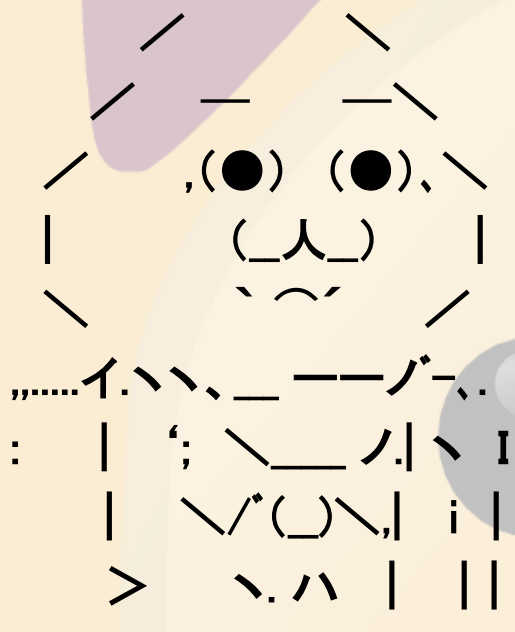
匠の伝承w

マルチな時代の設計と開発

駆け足で PART(3, 4),5,6

スピーカー自己紹介



ゆーちです。

ハンドル名です。

本名は、内山康広といいます。

48歳です。

おっさんです。_|￣|○

株式会社シーソフト代表取締役です。

現役のエンジニアです。プログラム書いてます。

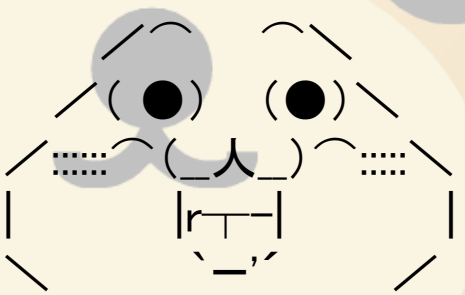
Becky!用 BkReplyer 2 よろしくお願いしますよ。ほんと。

にこにこカレンダーシートを販売しています。

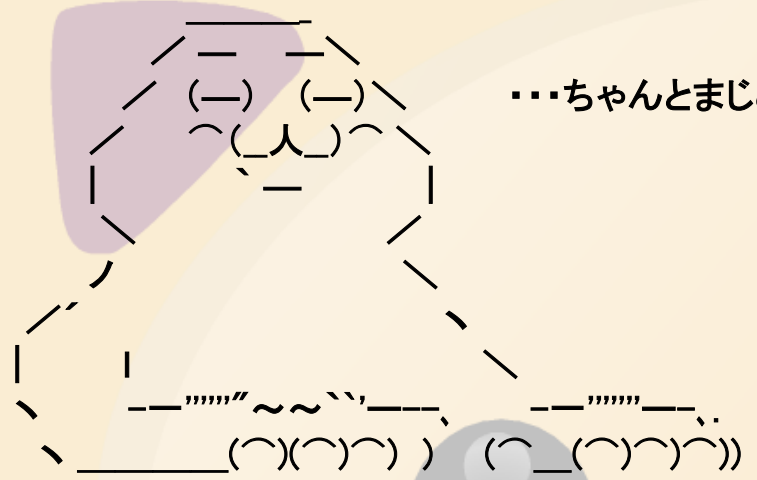
2ちゃんねらーではありません。

Special thanks for 2ch.

前回までのおさらい



そんなのがあったのか？。



・・・ちゃんとまじめな話をしたんだお。

PART 1

開発者はプロセス指向にとらえがち。

オブジェクト指向は『モノ』をとらえる。

『モノ』に対する時間軸のイベントを列挙。

時間軸へのイベントが『状態』を作る。

開発は『状態』別に分けて考える。

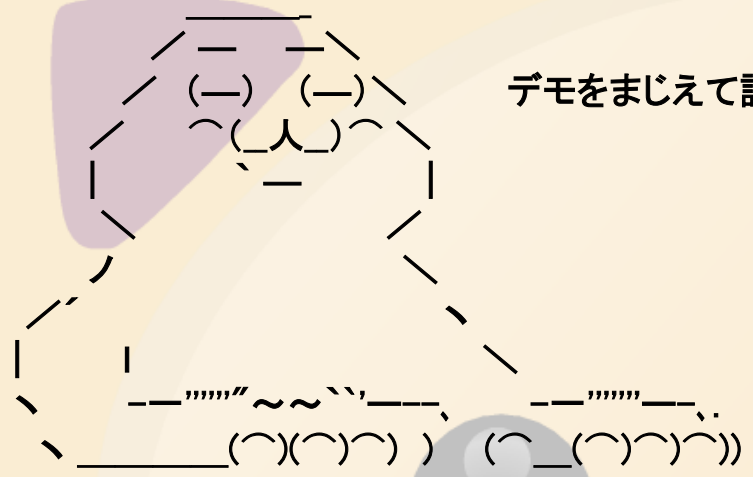
PART 2

処理の依存性を切り離す

非同期の事象はループを分断して考える。

イベントトレース図→状態遷移表。

ステートパターンの実装。



デモをまじえて説明したんだお。

PART　3

クラスに「時間軸」を考える。

オブザーバパターンによるイベント通知。

クラスの依存性を少なくし独立性を高める。

PART4？なんだっけ・・・DIかな？

では、PART5，6行きましょう！ｗ

いきなりですが。言語は何を使ってますか？

Ada
Pascal
PHP
APL
Scala
NC
Shakespeare
Whitespace
C++
C
Smalltalk
Java
C++
C++/CLI
UML
Objective-C
PostScript
D言語
LISP
PL/I
アセンブラ
なでしこ
ActionScript
F#
Tcl
SQL
Ruby
Visual Basic
Mind
Delphi
Prolog
Grass
FORTRAN
COBOL
Modula-2
LOGO
Scheme
Perl
ひまわり
JavaScript
＃＄％◎▽▲×・・・
Eiffel
ALGOL
Python
R言語

プログラミング言語って覚えるの大変!?

制御文

if, for, while, switch, goto, return, (), {} ...

演算子

+, -, /, *, %, mod, and, or, xor, sizeof ...

データ型と変数

int, float, string, struct/class, variant, ...

ややこしいのはライブラリや統合環境

変数の種類

- グローバル変数
- 静的変数
- 動的変数
- ローカル変数
- 関数やメソッドの引数

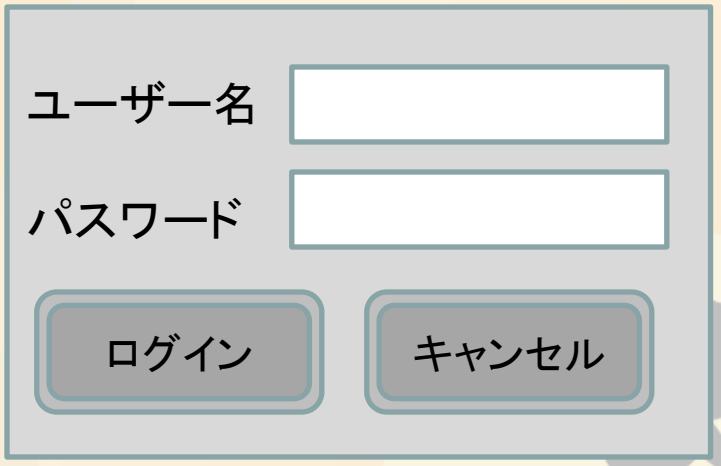
- メンバ変数

PART4で各変数について
説明したんだお。

でも、すっとばすw

実際にどのように使われているかな？

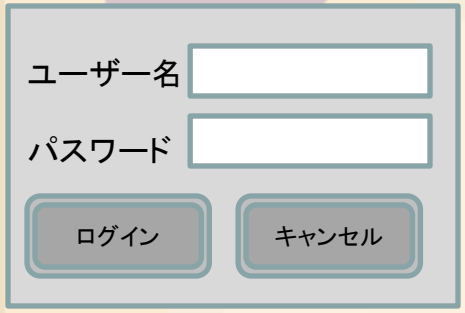
たとえば、こんな画面があったとしましょう。



ユーザー名とパスワードを入力
ログインボタンが押されたら、
入力情報が正しい文字の組み合わせになっているか検査
ユーザーが存在し、パスワードが正しいかを検証

どのように実装しますか？

ビューとロジックは、分けるよね？



ビュー

```
Form::Form()
{
}

Form::OnLoginClick()
{
}
```

ロジック

```
Logic::Logic()
{
}

Logic::Check (...)
{
}

Logic::Login(...)
{
}
```

データアクセス

```
DAC::DAC()
{
}

DAC::Connect(...)
{
}

DAC::QueryUser(...)
{
}
```

Formのコードイメージ

```
Form::OnLoginClick()
{
    string UserName = Text1->Text;
    string Password  = Text2->Text;

    bool ret = Logic->Check( UserName, Password );
    if( ret == true )
    {
        ret = Logic->Login( UserName, Password );
    }
    if( ret == false )
    {
            :
            :
```

編集テキストを内部で取得
ロジックに問い合わせ

Logicのコードイメージ

```
bool Logic::Check( string UserName, string Password )
{
    // UserName の妥当性検証
    // Password の妥当性検証
    return 真偽;
}

bool Logic::Login( string UserName, string Password )
{
    bool exist = DAL->QueryUser( UserName, Password );
    return exist;
}
```

DALのコードイメージ

```
DAL::QueryUser( string UserName, string Password )
{
    string SQL="SELECT COUNT * from UserTable "
                    "where (UserName=¥'%s¥')"
                    "and (Password=¥'%s¥')";
    try
    {
        SQL.FormatString( UserName, Password );

        DataBase->Query( SQL );
        if( DataSet->Count >= 1 )
            return true;
        }
    }catch( ... ){
    }
   return false;
}
```

【余談】保守性を下げる好き勝手な変数の命名

```
bool Logic::Login( string sUser, string sPsw )
{
    bool exist = DAL->QueryUser( sUser, sPsw );
    return exist;
}
```

アナタ以外の人が見ることを忘れずに

```
node *search(node *lhs, node *rhs);
```

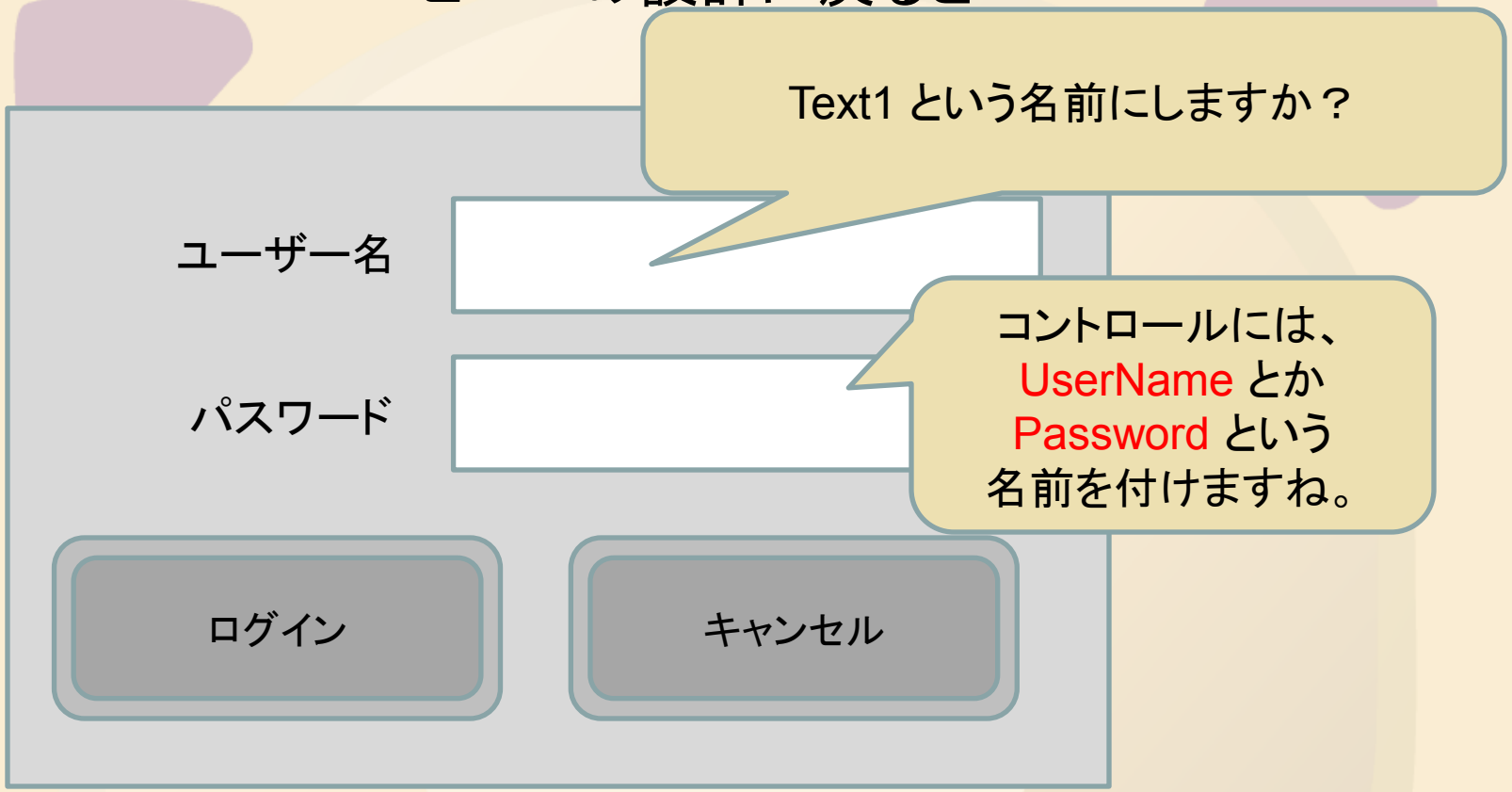
node *search(node *left, node *right);

統一された命名規則であることが重要！

ついでに、UserTableのイメージ

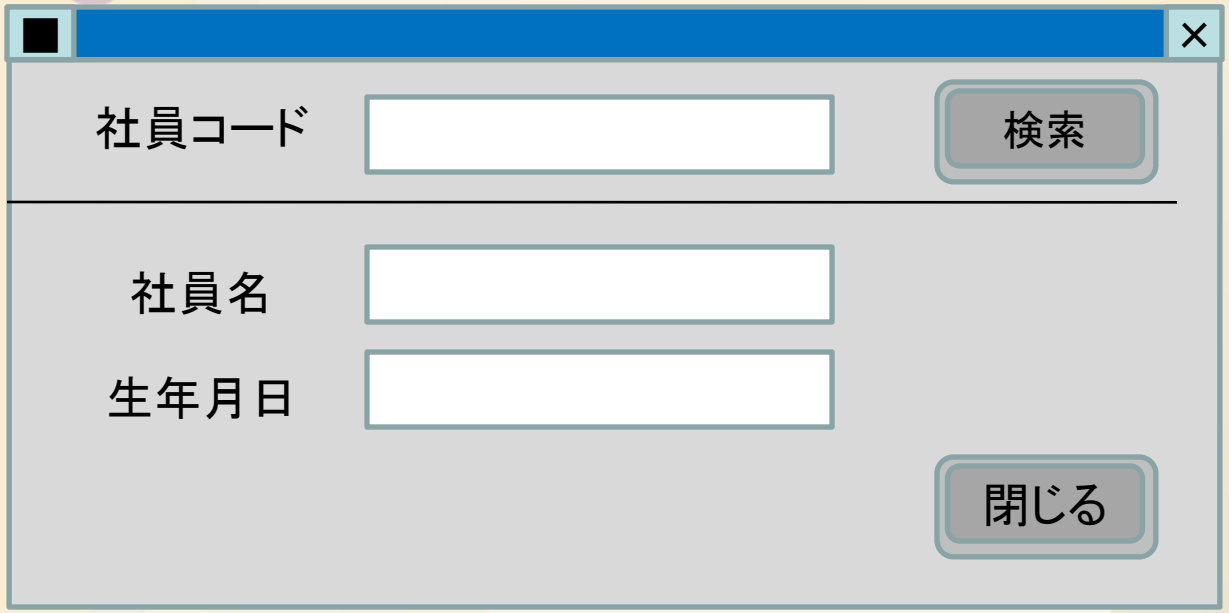
フィールド	型	サイズ	NULL許容
UserName	CHAR	40	×
Password	CHAR	20	×
:	:	:	:

ビューの設計に戻ると・・・

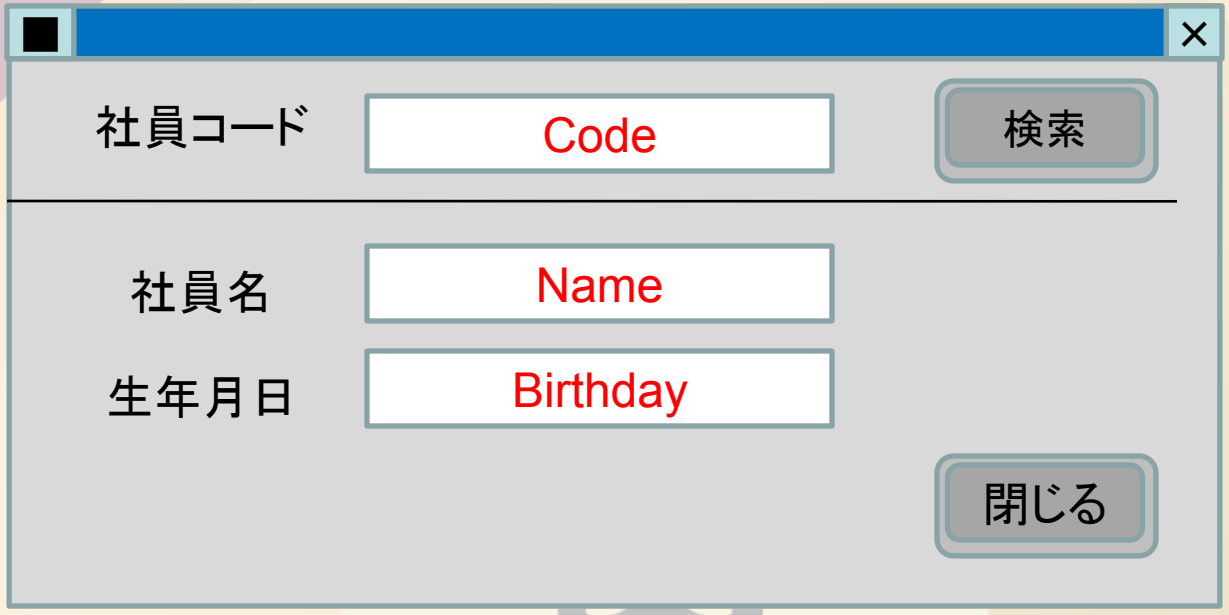

Text1 という名前にしますか？
ユーザー名
パスワード
コントロールには、
UserName とか
Password という
名前を付けますね。
ログイン
キャンセル

別の例を考えてみましょう。

こんな画面があったら？



画面の設計イメージ



Formのコードイメージ

```
Form::OnSearchClick()
{
    string Code = Code->Text;
    string Name, Birthday;

    bool ret = Logic->Check( Code );
    if( ret == true )
    {
        ret = Logic->GetPerson( Code, &Name, &Birthday );
    }
    Name->Text = Name;
    Birthday->Text = Birthday;
          :
```

Logicのコードイメージ

```
bool Logic::Check( string Code )
{
    // Codeの妥当性検証
    return 真偽;
}

bool Logic::GetPerson( string Code, string *Name, string *Birthday )
{
    bool exist = DAL->QueryPerson(Code, Name, Birthday );
    return exist;
}
```

DALのコードイメージ

```
DAL::QueryPerson( string Code, string *Name, string *Birthday )
{
    string SQL="SELECT * from PersonTable "
                    "where (Code=¥'%s¥')";
    try
    {
        SQL.FormatString( Code );

        DataBase->Query( SQL );
        if( DataSet->Count >= 1 )
            *Name     = Dataset->GetField("Name");
            *Birthday = Dataset->GetField("Birthday");
             return true;
        }
    }catch( ... ){
    }
   return false;
}
```

ついでに、PersonTableのイメージ

フィールド	型	サイズ	NULL許容
Code	INTEGER	8	×
Name	CHAR	40	×
Birthday	DateTime	8	×

ついでに、帳票設計があるとしたら？

社員名簿

コード	氏名	誕生日
Code	Name	Birthday

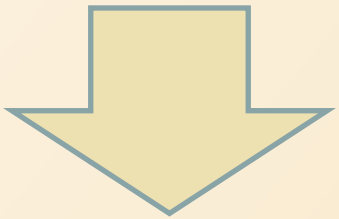
このように・・・

変数の名前って、あちこちで出てきますね。

毎回毎回、同じ名前を書かなきゃいけないよね。

あたりまえ？

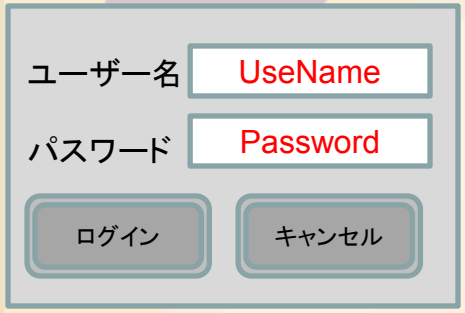
世界中のプログラマが、似たような画面やコードや帳票を毎日せっせと書いています。



めんどくさくないですか？

なんとかならないんでしょうか？

定義と宣言を見てみましょう。



ビュー

```
Form::Form()
{
}

Form::OnLoginClick()
{
}
```

コントロールとの入出力

ロジック

```
Logic::Logic()
{
}

Logic::Check (...)
{
}

Logic::Login(...)
{
}
```

Form／DALとの橋渡し

データアクセス

```
DAC::DAC()
{
}

DAC::Connect(...)
{
}

DAC::QueryUser(...)
{
}
```

Logicとのやりとり

フィールド	型	サイズ	NULL許容
UserName	CHAR	40	×
Password	CHAR	20	×
:	:	:	:

アプリケーションプログラム変数の宣言要因

- 画面のコントロール名
- データベースフィールド名
- 帳票ラベル／カラム名

そこで・・・

Form／Logic／DAL に設計情報から
自動的に変数を作り出してしまおう

という試み。

XMLファイルに定義情報を用意する

```
<Form Name="Form1">
  <UserInterface>
    <Field Name="UserName", DisplayName="ユーザー名", Type="String", ... >
    <Field Name="Password", DisplayName="パスワード", Type="String", ... >
  </UserInterface>
</Form>
```

```
<Database Name="AppDB">
  <Table Name="UserTable">
    <Field Name="UserName", DisplayName="ユーザー名", Type="String", ... >
    <Field Name="Password", DisplayName="パスワード", Type="String", ... >
  </Table>
</Database>
```

基本クラスでXMLを取り込み自動的に内部変数を用意する Form/Logic/DAL クラスを作る。

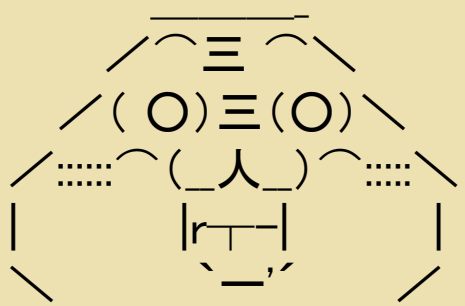
ちょっとまって。ホントに便利になる？

```
Form::OnSearchClick()
{
    string Code = Code->Text;
    string Name, Birthday;

    bool ret = Logic->Check( Code );
    if( ret == true )
    {
        ret = Logic->GetPerson( Code, &Name, &Birthday );
    }
    Name->Text = Name;
    Birthday->Text = Birthday;
          :
```

基底クラスの Fields[] 変数を利用してみると・・・

```
Form::OnLoginClick()
{
    bool ret=Logic->Check(Fields[“UserName”], Fields[“Password”]);
    if( ret == true )
    {
        ret = Logic->Login(Fields[“UserName”], Fields[“Password”]);
    }
    if( ret == false )
    {
          :
          :
```



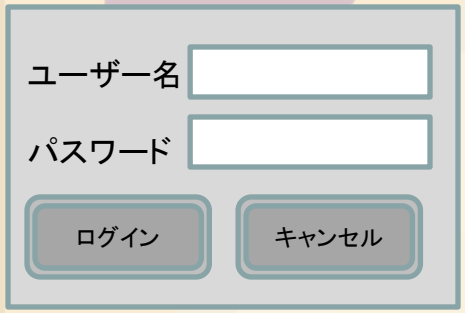
よけいにめんどくさくなったお

PART6

アプリケーション・パターン

似通った処理をコンポーネントにしてしまおうぜっ！みたいな(笑)

ビューとロジックの内部処理



ビュー

Form::OnLoginClick()
{
}

コントロールから値を取り出し、ロジックでCheck後にLogin

Form::OnCancelClick()
{
}

ロジックでCancel

ロジック

Logic::Check (...)
{
}

文字妥当性
登録されてるの？

Logic::Login(...)
{
}

ログインの記録とか

Logic::Cancel (...)
{
}

終了処理

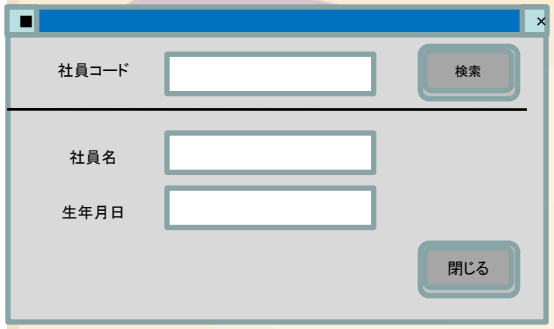
データアクセス

DAC::QueryUser(...)
{
}

登録されてるの？

DAC::Connect(...)
{
}

なにかする



ビュー

Form::OnSearchClick()
{
}

コントロールから値を取り出し、ロジックでCheck後にSearch

Form::OnChangeField()
{
}

ロジックから値の変更通知を受け取ったら、コントロールに設定

ロジック

Logic::Check (...)
{
}

文字妥当性
登録されてるの？

Logic::Search(...)
{
}

社員コードでデータ検索

データセットからフォーム用変数の通知

データアクセス

DAC::QueryUser(...)
{
}

登録されてるの？

DAC::Search(...)
{
}

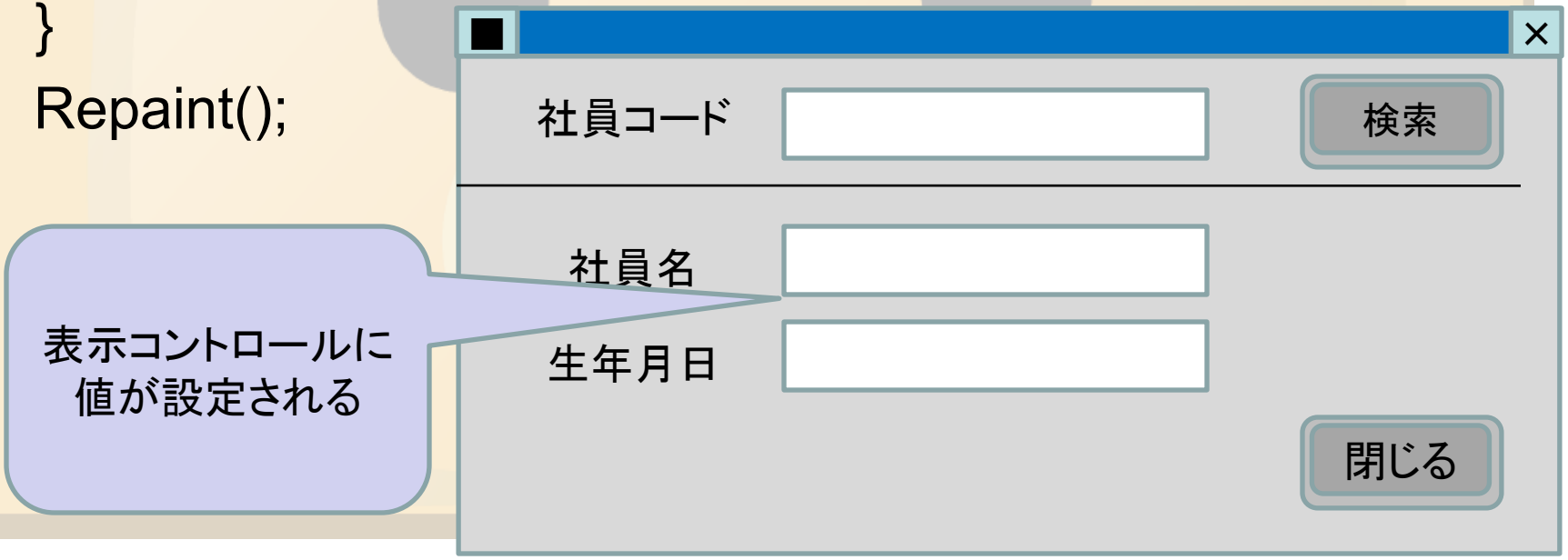
社員検索SQL発行

これら2つを1つにできないでしょうか？

```
BaseForm::OnActionExecute()
{
    UpdateFields();//おまじない
    //入力コントロールを列挙
    foreach( Control *control = InputControls  ){
      // ロジックのプロパティに入力値を転送
       Logic->Fields[control->Name] = Fields[ control->Name];
    }
     // ロジックに値チェックをお願いする
     if(   Logic->Check() ==   true ){
          Logic->Execute();
```

注意：コードはあくまでもふいんきwです。

```
BaseForm::OnPropertyNotifyChanged()
{
    //表示コントロールを列挙
    foreach( Control *control = DisplayControls  ){
      // ロジックのプロパティに入力値を転送
       Fields[control->Name] = Logic->Fields[control->Name];
    }
    Repaint();
}
```



Logicはどうなる？

```
bool BaseLogic::Check()
{
    return true;
}
bool BaseLogic::Execute()
{
    return true;
}
```

検証すべきフィールドや条件は毎回異なるので、基底クラス(BaseLogic)では汎用的な実装はできませんね。

何を実行するのか、というのも要件によって異なりますので、基底クラスで実装できにくいですね・・・

（意味なさそうだお）

派生クラスで個別対応？

```
class LoginLogic : public BaseLogic …

bool LoginLogic::Check()
{
    return DAL->QueryUser(Fields[ “UserName” ],
                          Fields[ “Password” ]);
}
```

仮にロジックを個別実装することになったとしても、
この方法が実現できれば、少なくともフォームの
コードについては基底クラスが処理してくれるので、
画面設計だけやっておけばロジックの必要箇所の
実装だけですむことになります。

標準のアクションリスト

一般的な業務アプリケーションの場合、オペレータのアクションの入り口（メニューやボタン）には、次のようなアクションが用意されます。

「キャンセル」

「OK（実行）」「更新」「登録」「確定」

「削除」「検索」

「前ページ」「次ページ」

「コピー」「貼り付け」などなど

定型ロジックを基底クラスにうめこんじゃおう

Formの[OK]処理

入力フィールドを検証（Check）

＞チェックNGの場合、メッセージ表示

＞間違った箇所にカーソル移動

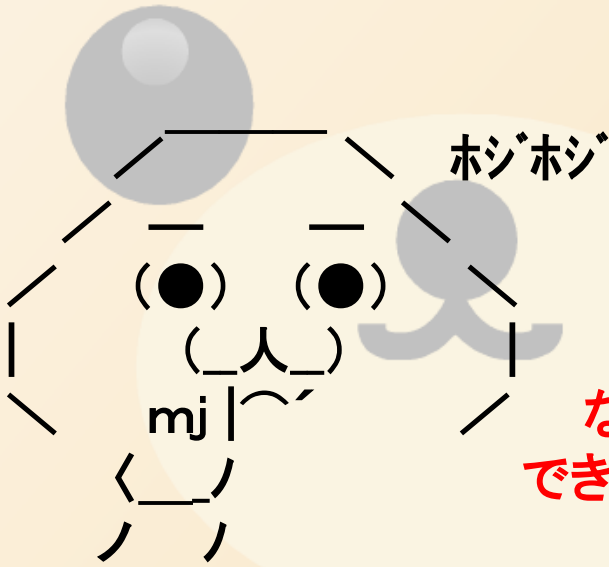
チェックOKなら、実行処理(Execute)

＞必要な保存処理を実施する

Formの[キャンセル]処理

フォームを閉じる。

アプリケーションフォームならプログラムを終了する



え一？？？
このくらいまでならわかるけど
なかなかそう簡単に共通化は
できないんじゃなぁい？？？

グリッドを持つ画面


月間一覧表
月
Month
表示
商品コード
商品名
顧客コード
数量
単価
金額
ProductId
Product
CutomerId
Count
UnitPrice
Amount
赤い文字はXMLで定義された名前
前ページ
次ページ
閉じる

[表示]ボタン処理

基底データアクセスロジックの一部・・・

```
string SQL = “SELECT “
foreach( field = OutputFields ){
   SQL += field->Name;
   if( field != End of List ){
      SQL += “,“;
   }
}
SQL += “from “ +DatabaseTable;
SQL += “WHERE “;
foreach( field = InputFields ){
    SQL += field->Name + “= ¥””
           + field->Value + “¥””;
    if( field != End of List ){
        SQL += “&&”;
    }
}
```

SELECT
ProductId,Product,CutomerId,Count,UnitPrice,Amount

from UserDatabase.**UserTable**

where
Month = “5”;

この辺もXMLで定義

ページ行数＋1を選択とかもあり。

[次ページ]ボタン処理（あくまでもイメージw）

```
BaseLogic::NextPage()
{
   DAL->NextPage( CurrentPage );

   foreach( field = Grid[ “グリッドの名前” ].Fields )
   {
      field[ フィールド名 ] = DAL->Dataset[ フィールド名 ];
   }

   Observers->Notify( GRID_CHANGE, グリッド名);
}
```

SQL発行

データセットからローカルに値を取得

グリッドを表示しているフォームに変更通知

データベースがあるとは限らない！

そのとおり。

DBとは異なる格納なら、DALだけかえてしまえばいいんじゃない？

（設定情報をiniファイルに書いたりレジストリにしたり）

データベース、ini ファイル、レジストリの入出力があれば、多くのアプリケーションでは満足なのでは？

通信を利用するアプリケーションに適用できない！

そんなことはないっす。

一般的な通信手段として

・TCP/IPとか

・COMポートとか

は、標準的なやりとりで吸収できます。

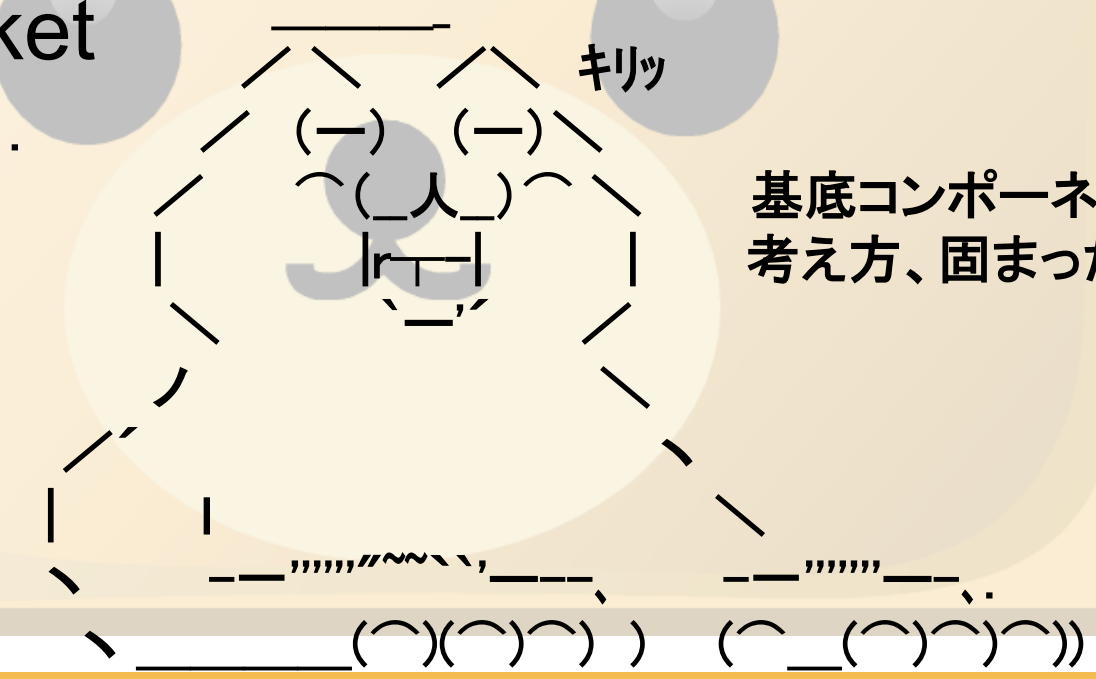
サーバアプリとクライアントアプリ

- サーバーアプリケーション

  ServerSocket

- クライアントアプリケーション

  ClientSocket



たとえば、どうなる？

典型的な例では、「マスターメンテナンス画面」

テーブル設計
メンテナンス画面
それぞれのXML定義

ほんとかお？

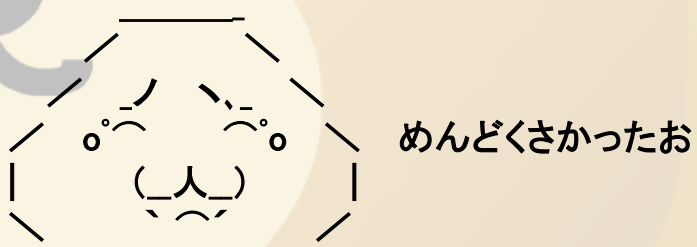
よろしくおながいしますお！

だけで、コードを書かずにできあがってしまう！

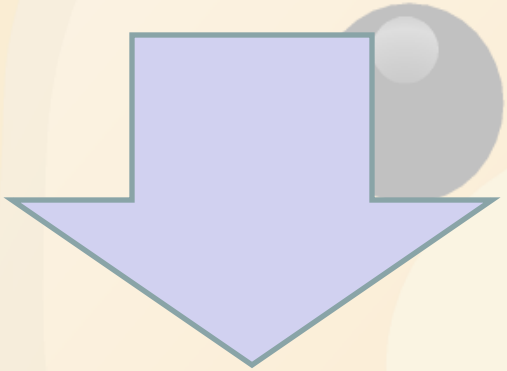
たとえば、どうなる？

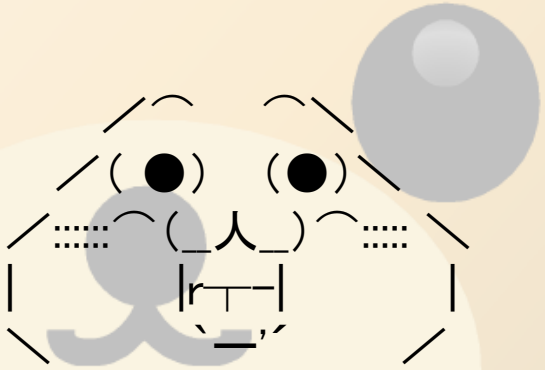
アプリケーションの設定情報

画面レイアウト、XML定義だけでコードを書かなくて、できちゃう。



プログラムの開発時間を大幅に短縮できる！



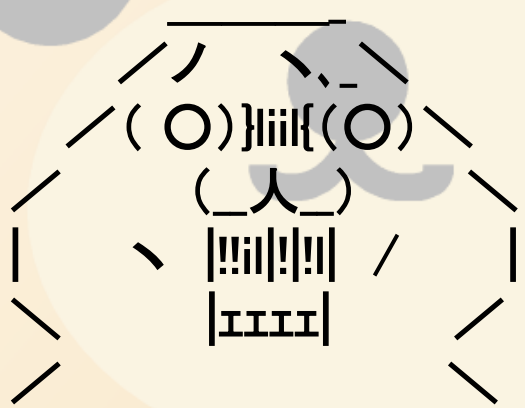


ほんとかお？

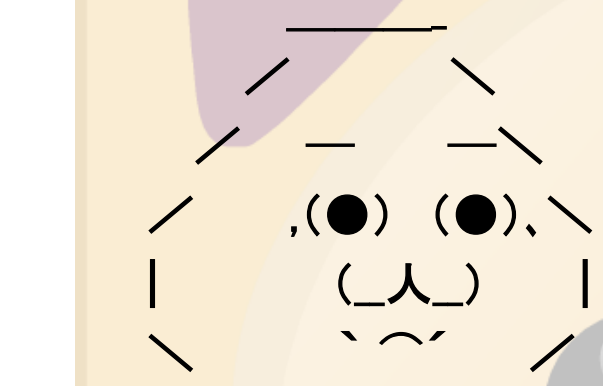
定時で帰っていいのかお？

デスマーチからの解放！

現在、誠意開発中（笑）



まだできてないのかお！



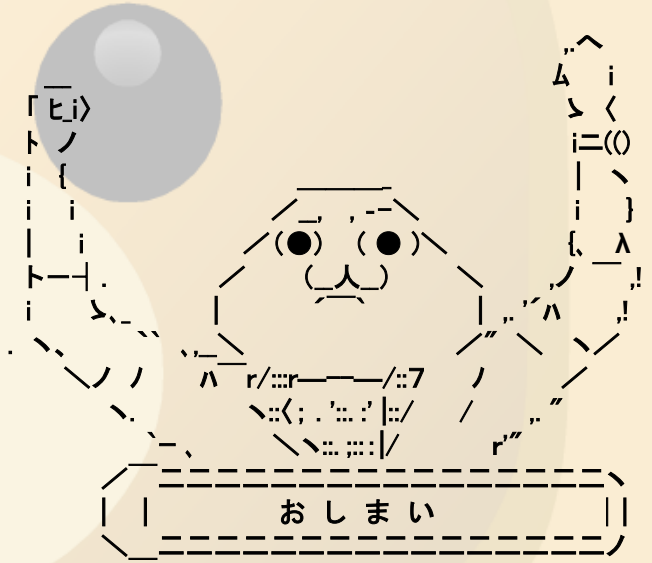
残念ながら時間がきてしまったようです。

次回は、ここで紹介したフレームワークを
実際にデモを交えてご紹介しましょう。

いつになるのか、皆目見当もつかないわけだw

ご静聴ありがとうございました。

m(_._)m



Special thanks for Yaruo charactors