embarcadero Developer Camp

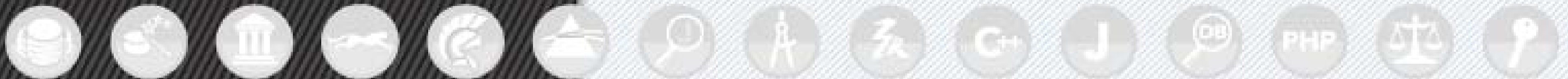
【B1】テクニカルセッション

「業務で使える！ C++実践ワンポイントテクニック」

アナハイムエンジニアリング株式会社
代表取締役
服部励起

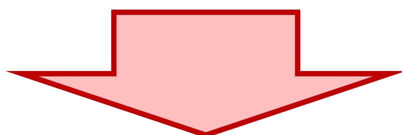
embarcadero
Developer Camp

エンジニアとして「都市伝説！？」



近頃！？の見たり聞いたり

- 「周りの技術レベルが低い」
- 「改修よりスクラップアンドビルドが効率的」
- 「オブジェクト指向だから開発工数が削減できる」
- 「アジャイル開発なので各種標準化は不要」
- 「ソースコードレビューは不要（出来ない）」
- 「ドキュメントよりもソースコードが重要」
- 「インデントが無いソースコードの方が読み易い」
- 「コピペしただけなので、バグは作成元に責任があり私では無い」



プロフェッショナルの仕事では無い

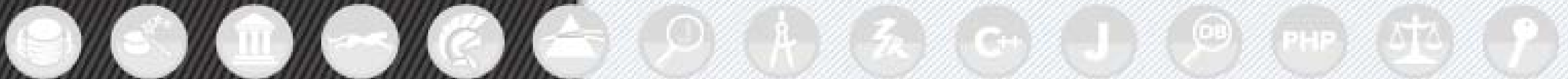
embarcadero Developer Camp

それなら私がやるしか･･･　　(;´∀`)

- 技術指向より業務指向
  - ハイテクニック依存は属人的になりやすい
  - 組織全体としての効率化とそのテクニック中心
- プロフェッショナルとは何か？
  - ソフトウェア品質特性
  - 特に保守について
- C++言語特性を再確認
  - 業務的視点から見たメリットとデメリット

embarcadero Developer Camp

集団開発技術として「業務指向開発！？」



技術指向と業務指向

- 技術指向とは
  - 理解できる人のみ理解できれば良い
  - 華麗なテクニックや最新ライブラリ等を駆使
  - 自分が頑張って勉強したので、他人も同様の努力を要求
  - コメントは極力少ない傾向
- 業務指向とは
  - 組織としての最大の効果を常に考える
  - 技術レベルは、8割が理解できるレベルで統一
  - 他人がソースを読むことを常に念頭に
  - 仕様変更を念頭にクラス設計を

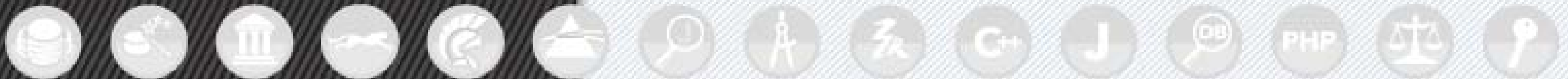
業務指向の効果とは

- 技術指向の罠は、属人化
  - 担当した仕事の保守は、概ね元担当者へ
    （会社側は新たな仕事を担当させられない事に）
  - 属人的ソースコードの行方はスクラップアンドビルド
    （スクラップアンドビルドの非効率工数は誰が支払う？）
  - 但し、ソフトウェア寿命がワンオフなら話は別

- 業務指向では他人に任せられる点
  - 面倒な保守は新人等に担当させたい
  - 転職時にも面倒な引き継ぎ期間の短縮
  - 特に複雑な部分をブラックボックス化する等の柔軟性も

embarcadero Developer Camp

プロフェッショナルとして「期待される特性」



ソフトウェア品質特性(ISO9126)

- 機能性：要求通り実装されている等
- 信頼性：バグが少ない等
- 使用性：使い易い等のUX
- 効率性：レスポンス等
- 保守性：保守する場合の対応等
- 移植性：環境変更時の対応等
- セキュリティ：機能性の一部

非機能要件と呼ばれる内容！

Note. 移植性の補足

- 移植性の構成要素
  - 環境適応性
    - インストール先変更しても問題が発生しない
  - 設置性
    - インストールのしやすさ等
  - 共存性
    - 複数のソフトウェア同士で、共存することによる問題点回避
  - 置換性
    - ソフトウェアの依存度や他ライブラリバージョン依存の回避

ソフトウェアアーキテクチャでの対応

Note. 保守の重要性

- ソフトウェア目標寿命まで運用できること
- 保守工数がソフトウェアライフサイクルの大半を占める
  - 氷山に例えられることも
    - 氷山は海上は全体の10%
      海面下は90%
    - 初期開発工数は10%で、
      保守工数は90%
  - 保守工数を削減し利益へ貢献
- 経営環境が迅速に変化しているため、ソフトウェア保守に時間を割くことが難しくなっている

参照: http://ja.wikipedia.org/wiki/%E3%83%95%E3%82%A1%E3%82%A4%E3%83%AB:Iceberg.jpg
Created by Uwe Kils (iceberg) and User:Wiska Bodo (sky).

Note. 保守の重要性

- オブジェクト指向技術
  - 保守工数の削減に利用を主眼に置き、開発工数削減を狙うと失敗する危険性が高い
    - クラス設計や構成等は十二分に検討
- ソースコードに対する技術不足
  - 他人のソースコードを読み、理解し、推測できる技術が特に重要
  - 優れたオープンソースを読む訓練は役に立つ

保守性の構成要素に対応するテクニック

- 解析性
  - コーディング規約全般、命名規約、コメント規約
  - 標準化運用ルール

  ※リバースエンジニアリングツールを過信してはダメ

- 変更容易性
  - コーディング規約全般、クラス設計標準
  - 各種ドキュメント

- 試験性
  - ソフトウェア変更標準
  - ユニットテスト

C++の言語特性

マルチパラダイムプログラミング言語

- 構造化プログラミング
  - 階層的に抽象化されたプログラム記述形式
- 命令型（手続き型）プログラミング
  - プロシージャ、ルーチン、サブルーチン、メソッド（グローバル関数・変数）
- オブジェクト指向プログラミング
  - オブジェクトの集まりとしてプログラムを構成（C++標準）
- ジェネリックプログラミング
  - データ形式に依存しないプログラミング方式（C++テンプレート機能）

マルチパラダイムプログラミング言語

- マクロ
  - テンプレートはマクロの発展系
  - C++では問題点も多いと指摘されることも
- 演算子のオーバーロード
  - 演算子を別の処理にコンパイル時に置き換える
  - iostreamでは普通に使われている「<<」等

embarcadero Developer Camp

C++の歴史

規格制定日	C++ 標準規格	非公式名称
1998年9月1日	ISO/IEC 14882:1998	C++98
2003年10月16日	ISO/IEC 14882:2003	C++03
2007年11月15日	ISO/IEC TR 19768:2007	C++TR1
2011年9月1日	ISO/IEC 14882:2011	C++11

- 言語仕様が変遷しており、今後も変更される
- 関数型プログラミングも今後取り込む予定
- 恐らく前方互換性を持たせるため、旧規格で作成しても問題は無い
- 保守業務を考えると最新技術仕様を取り込む場合には注意が必要

embarcadero Developer Camp

実践ワンポイント「標準化編」

1.クラス定義

- 1つの物理ソースファイルに、複数のクラス定義を実装するのを是か非か
  - 関連性が維持できるクラス定義は統合するべき。
    但し、担当者が分散するケースが散見されるなら敢えて1ファイルを1クラスにするのも一考

```
class HogeBase {
protected:
        virtual int Get() = 0;
        virtual void Set(int st) = 0;
public:
        HogeBase();
        ~HogeBase();
};

class HogeData : private HogeBase {
public:
        int Get();
        void Set(int st);
};
```

仮想クラスを多用する場合には1クラスの方が見通しが良い

2.メソッド行数制限と行内桁数制限

- メンバ関数行数制限
  - 最大でも200～250行
  - それ以上は分割する。
    但し、初期化処理は例外とする

- 1行内の桁数制限
  - ディスプレイサイズで決定すれば良いのかなと・・・
  - 個人的には160～180桁

※昔は80桁以内か以上で論争がありましたねぇ。

3.カプセル化方針

- 内部変数のpublic公開について
  - 異論はあるが、内部変数をpublic公開は禁止
  - getter/setterメンバ関数(アクセッサ)を利用するべき
  - 全てのクラス内定義は、privateで作成すると便利
  - クラスデフォルトはprivateだが全て記述するべき

```
class HogeBase {
private:
        int st;
protected:
        virtual int Get() = 0;
        virtual void Set(int st) = 0;
public:
        HogeBase();
        ~HogeBase();
};
```

DBデータを扱うクラスも、継承してメソッドをpublicで公開すると、仕様変更時の柔軟性が上がる。

4.継承方針

- 多重継承の禁止
  - 混乱と仕様変更時の柔軟性に影響
- 基底クラスを使い、継承を利用して公開する仕様も検討してはどうでしょう？

```
//基底クラス定義
class Calc_Base {
public:
        Calc_Base();
        ~Calc_Base();
};
// 計算クラスを継承して加算クラスを作成
class Calc_Add : private Calc_Base {
public:
        int Opereation(int data1, int data2);
};
// 計算クラスを継承して減算クラスを作成
class Calc_Sub : private Calc_Base {
public:
        int Opereation(int data1, int data2);
};
```

継承階層が深くなり過ぎないように注意する必要はあります。

5.コンストラクタとデストラクタ

- 使用しない場合でも定義だけはコーディング
  - 修正が必要となった場合に楽
- 継承時の実行順番は重要！
  - ①スーパークラスのコンストラクタ
    ②サブクラスのコンストラクタ
    ③サブクラスのデストラクタ
    ④スーパークラスのデストラクタ

6.パラメタ設計方針

- 大きく分類して2種類
  - getter/setterメンバ関数は欲しい結果をreturnで受け取る
  - returnで受け取るのは、メンバ関数が動作したbool情報とし、結果値は別メンバ関数経由で返却

- getter/setter系メンバ関数
  - 結果情報を直接returnで返却する方式

```
class CalcAdd : private CalcBase {
public:
        int Opereation(int data1, int data2);
};

int CalcAdd::Opereation(int data1, int data2) {
        return data1 + data2;
}
```

6.パラメタ設計方針

- returnでbool値(int値も可)で動作状態判別系
  - メンバ関数が正常に動作したのかをreturnで通知し、その時メンバ関数が行なったデータは別メンバ関数(getter)で行う

```
class hoge {
private:
    DB_EmpData data;
public:
    hoge();
    ~hoge();
    bool getEmployee(int empNumber);   //社員情報の取得メンバ関数
    DB_EmpData getEmpData();           //取得した社員データ提供メンバ関数
};
```

7.構造体･共用体

- C++ではクラスで代用できるため、コーディング標準的には積極的には使用しない方向で
- C言語開発経験者が多いプロジェクト等ではC++でも利用されている可能性も

```
struct hoge {
protected:
        int a;
public:
    hoge();
    int getA() {return a;}
};
struct hoge2 : public hoge {
private:
    int b;
public:
    hoge2();
    int getB() {return b;}
};
```

8.演算子のオーバーロード

- 一般的には極力控えた方が、保守性に貢献するものと想定します。
  - 演算子のオーバーロードを自由自在に操れる人は少数
  - 属人的になりやすいため敬遠することが賢明

- 演算子としか見えないため、オーバーロードされていると気が付きにくい。
  - きちんと管理されていれば、利用することは吝かではない

9.マクロの扱い

- マクロとは
  - コンパイラがソースコードをコンパイルする直前に、プリプロセッサで実行される命令群
  - スコープがプロジェクト全体
  - コンパイラに近い言語と言え、習熟している人は少ない
- 基本的にマクロ使用は最低限度に抑え、保守性を向上させる方針を推奨
  - 関数形式マクロを使うならテンプレート化を検討

```
オブジェクト形式マクロ文法
#define [マクロ名] [置換文字列]

関数形式マクロ文法
#define [マクロ名](引数リスト) [置換文字列]
```

10.テンプレート

- テンプレートのメリット
  - マクロより型安全
- テンプレートのデメリット
  - 移植性が低い
  - エラーメッセージが分かり辛い
  - 実行ファイルが大きくなる危険性もある
- テンプレートの使用は、テンプレート関数までとし複雑になりやすいテンプレートクラスは避ける
- 特に実行時デバッグでは分かり辛いため、デバッグ効率性かも極単機能で分り易いものだけに限定

10.テンプレート

例）

```
template <class X> int operation(X *obj, int data1, int data2) {
    return obj->Operation(data1, data2);
}

void __fastcall TMyCalc::bt_equClick(TObject *Sender)
{
    int ans;
    switch(calc_kind) {
        case C_add: {
            CalcOpe_Add obj;
            ans = operation(obj, data1.Get(), data2.Get());
            break;
        }
        case C_sub: {
            CalcOpe_Sub obj;
            ans = operation(obj, data1.Get(), data2.Get());
            break;
        }
    }
    Panel1->Caption = IntToStr(ans);
```

11.define, const, typedef

- defineはマクロのため、コンパイル時に置き換え
- constは修飾子のため、全てに使用可
- typedefは変数の型を独自に定義

- マジックナンバーはdefineかconstで定義しておく
  - constは型を指定できるため、こちらを推奨
  - スコープはdefineは全体。constはソース定義範囲。

```
const int MAX_ARRAY_SIZE = 3000;

#define MAX_ARRAY_SIZE 3000

typedef int myInt16;            //myInt16という型を定義
#define myInt16 int             //typedefと同じ効果を持つ
```

12.constメンバ関数

- メンバ関数に付与することで、その関数内でメンバ変数の値を変更しない
  - getterメンバ関数等で使うと非常に便利

```
class constClass
{
        int i_test;
        const int MAX_SIZE;
        mutable int m_test;    //mutableを使うとconst中でも変更可だが・・・
public:
        constClass() : MAX_SIZE(10) {}          //コンストラクタで初期化
        int getMaxSize() const {                //constメンバ関数
                m_test = 3;
//              i_test = 10;                    コンパイルエラー
//              MAX_SIZE =  20;                 コンパイルエラー
                return(MAX_SIZE);
        }
};
```

12.各種命名規約

- 組織等で分り易い記述形式にし、変数からメソッド、クラス名、例外エラークラス等ざっくりとでも決定
- ハンガリアン記法
  - データ属性を小文字の接頭語として使用
- キャメル記法
  - 単語の区切りを大文字を利用する

```
// ハンガリアン記法
int iCount;
char cName[10];

// キャメル記法
std::string lastName;  //ローワーキャメルケース
std::string FirstName; //アッパーキャメルケース
```

embarcadero Developer Camp

13.コメント規約

- ファイルコメント
  - .cppや.hファイルの先頭に、作成者やファイルの概要や名前、修正履歴等を記入
    (ドキュメントの不備等に対応させる目的)
- クラス・メソッドコメント
  - クラス定義部と各メソッド実装部に、簡単でも目的
  - 実装でトラブルがあった場合、トラブル内容等を簡易に記述する
- 実装コメント
  - 例外処理や、規約違反している場合の詳細説明、実装に苦労した内容等を詳細に記載

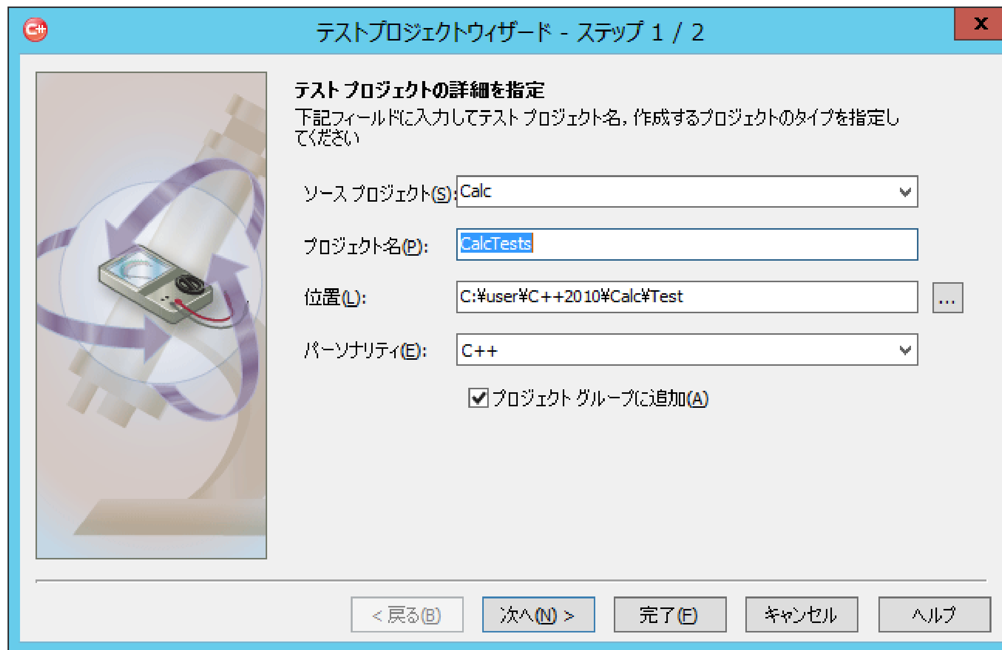
embarcadero Developer Camp

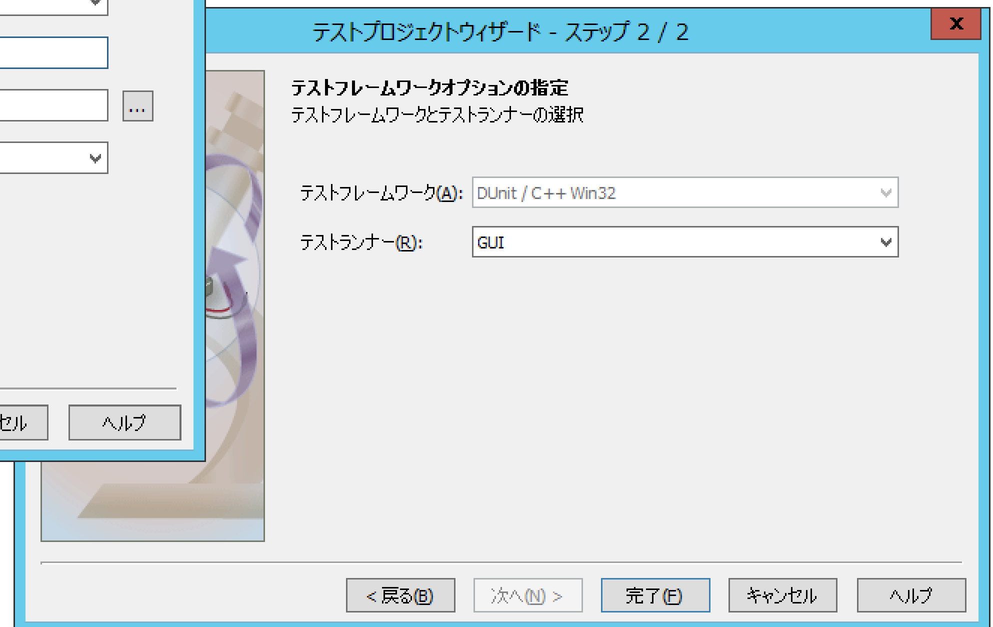
embarcadero
Developer Camp

実践ワンポイント「ツール編」

1.ユニットテストの活用

- テストプロジェクトの追加





1.ユニットテストの活用

- テストケースの追加

新規作成

C++Builder プロジェクト
ActiveX
C++Builder ファイル
DataSnap Server
IntraWeb
Web サービス
WebBroker
WebSnap
XML
多層サポート
Profiling
Web ドキュメント

検索

テスト プロジェクト
テストケース

テストケースウィザード - ステップ 1 / 2

テストするメソッドの選択

テストするメソッドを選択してください。メソッドがチェックされな
生成されます。

ソース ファイル(S):

C:¥user¥C++2010¥Calc¥CalcMain.h

指定可能なクラスとメソッド(A)::

TMyCalc
void bt_nClick(TObject *Sender)
void bt_clrClick(TObject *Sender)
void FormShow(TObject *Sender)
void bt_opeClick(TObject *Sender)
void bt_equClick(TObject *Sender)

< 戻る(B) 次へ(N) > 完了(F)

テストケースウィザード - ステップ 2 / 2

テストケースの詳細を指定

下記フィールドに入力して，テストを追加するプロジェクト，ファイルの名前，使用する
テストフレームワーク，およびテストケースに使うベースクラスを指定してください。

テスト プロジェクト(P): CalcTests

ファイルの名前(A): TestCalcMain.cpp

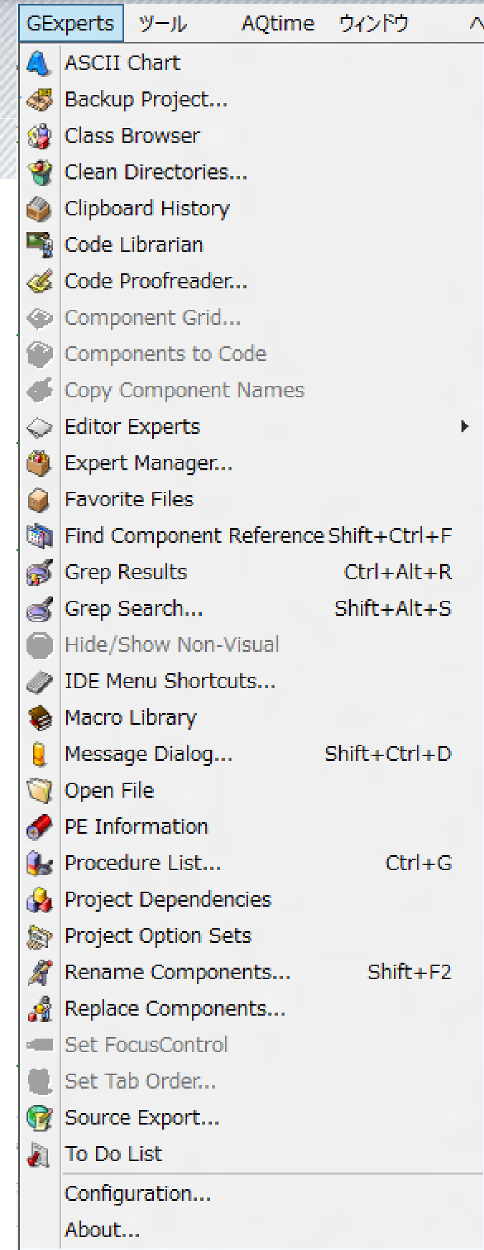
テストフレームワーク(M): DUnit / C++ Win32

基本とするクラス(C): TTestCase

< 戻る(B) 次へ(N) > 完了(F) キャンセル ヘルプ

2.GExpertsの活用

- GExperts
  - 機能が多すぎで分かりません。(^^;;
  - Editor Experts
    - エディタの補完を行ってくれます
  - Grep Results/Grep Search
    - プロジェクト全体のGrepも出来ます
  - Procedure List
    - 今開いているファイル内の関数一覧



http://www.gexperts.org/

embarcadero Developer Camp

embarcadero®
Developer Camp

Q & A