ユニットテスト導入の技術的課題とその解決例

益　弘和

Ubisoft Osaka 株式会社

アジェンダ

セッションの最終目的:
コマンドラインでユニットテストを実行できる環境を構築する！

対象としている受講者:
- ユニットテストは知っている
- まだユニットテストを導入していない

備考:
- できるだけ直接の経済的コストをかけないようにする

アジェンダ

本セッションで扱わないこと

- ユニットテスト自体の説明
- 「技術」以外の課題
- 特定の製品について

ユニットテスト環境における問題

一体どうやってユニットテストを実行するのか？

C++ だし･･･

ゲーム開発機だし･･･

プラットフォームSDK（非標準）はあるし･･･

Java のように簡単ではない！

ユニットテスト環境における問題

ユニットテスト環境の要求定義

- C++のコードをテストできる
- 開発機上で実行できる
- コマンドラインから実行できる
- テスト結果をコマンド起動側で認識できる
- 記述したソースに関するテストをすぐに実行できる
- ソースの依存関係に基づいたビルドとテストが実行できる

ユニットテスト環境における問題

ユニットテスト環境の要求定義

開発言語はC++とする

・・・もしかするとスクリプト言語も？
（とりあえず今回は無視）

ユニットテスト環境における問題

ユニットテスト環境の要求定義

SDKやハードに依存するコードのテストは結合テスト？

SDKに対するスタンス：
独自に定義された型なども多々あるので・・・
× SDKのモックを作る
○ 標準ライブラリのように扱う
（SDKの受け入れテストまでできれば素晴らしい・・・！）

ゲームのコードと同様に開発機上で実行できると楽

ユニットテスト環境における問題

ユニットテスト環境の要求定義

IDEは使わない！

↓

代わりに自前で実装する

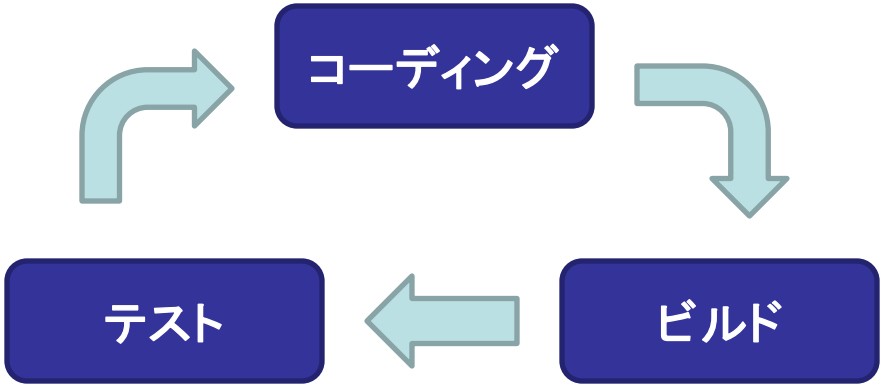
（製品を購入しないためでもあるが、
結局は高くつく可能性はある）

ユニットテスト環境における問題

ユニットテスト環境の要求定義

ビルドシステムの要求

- ユニットテストをビルドして実行できること
- テスト駆動開発（TDD）をサポートすること



このコーディングサイクルを素早く回せることが重要

ユニットテスト環境における問題

テスト実行環境構築の前提条件

・コマンドラインからプログラムの実行ができること
（これができないと話にならない）
・プログラムからコマンドの出力に文字列を出力できること
（不可能であればデバッガのコンソールへの出力で妥協）

もしそのようなコマンドが用意されていない場合は・・・
・自作する
・そういった機能を持った製品を導入する

ここではそのような文字列出力関数とコマンドがあるものとする

ユニットテスト環境の構築

テストフレームワークの選択

C++用テストフレームワーク

- CppUnit
- QTestLib
- GoogleTest
- UnitTest++
- Boost::unittest
- Cutter
- C++test（製品）

など・・・

ユニットテスト導入の技術的課題とその解決例

「Boost::unittest　君に決めた！」

Boost::unittest

ぽっきょ～ん

ユニットテスト環境の構築

テストフレームワークの選択

Boost::unittest を選んだ積極的理由

- テストインターフェイスが豊富
- テストフィクスチャの記述がスマート
- テストコードに C++ のあらゆる機能が使用可能
- 実行環境を限定しない
- 実行時に指定可能なパラメータが豊富
- 様々なコンパイラに対応している
- 工夫次第でテストに役立つ様々な機能を追加できそう
- 現在でも保守されている

ユニットテスト環境の構築

ストリームライブラリの実装

テストフレームワークが標準出力ストリームを使用している

↓

しかし、ゲーム開発環境では標準出力が機能していない

↓

有効な標準出力ストリームを実装する
（streambufを実装する）

＊プラットフォームSDKにC言語のprintf()系のような
＊出力関数があるものとする

ユニットテスト導入の技術的課題とその解決例

streambuf のカスタマイズバージョンの実装例

```
#include <streambuf>
template < typename charT, typename traits = std::char_traits<charT> >
class ConsoleOutputStreamBuf : public std::basic_streambuf<charT,traits>
{
public:
    typedef typename std::basic_streambuf<charT,traits>::char_type char_type;
    typedef typename std::basic_streambuf<charT,traits>::int_type int_type;
private:
    virtual int_type overflow( int_type c = traits::eof() )
    {
        if ( this->pbase() )
        {
            *this->pptr() = char_type();
            ::_PutString( this->pbase() );    // 処理系の出力関数
        }
        setp( buf_, buf_ + buf_size_ - 1 );
        if ( traits::eq_int_type( c, traits::eof() ) )  return traits::not_eof( c );
        ::_PutChar( c );                  // 処理系の出力関数
        return c;
    }
    virtual std::basic_streambuf<charT,traits>* setbuf( char_type* s, std::streamsize n )
    { /* ... */ }
    virtual int sync()
    {
        this->overflow( traits::eof() );
        return 0;
    }
    /* ... */
private:
    char_type* buf_;              // バッファ
    std::streamsize buf_size_;    // バッファサイズ
};
```

ユニットテスト導入の技術的課題とその解決例

カスタムストリームの有効化の実装例

```
#include “ConsoleOutputStreamBuf”

// ストリームの文字型
typedef char StreamCharType;

// カスタムストリーム用バッファ
 char custom_cout_buf[BUF_SIZE];

// 処理系定義のメイン関数
int main()
{
    // 出力ストリームバッファ型
    typedef std::basic_streambuf<StreamCharType> OutStreamBuffer;

    // カスタムストリームバッファを構築
    ConsoleOutputStreamBuf<StreamCharType> custom_stream_cout;
    custom_stream_cout.pubsetbuf( custom_cout_buf, BUF_SIZE );

    // デフォルトストリームバッファと入れ替え
    OutStreamBuffer* const cout_buf = std::cout.rdbuf( &custom_stream_cout );


    /* ここでメイン処理を実行 */


    // ストリームバッファを元に戻す
    std::cout.rdbuf( cout_buf );

    return 0;
}
```

ユニットテスト導入の技術的課題とその解決例

＊標準入出力ストリームが使えると・・・＊

・ストリームを使う一般のプログラムを
ゲーム開発機上で動かすことができる

・クラスオブジェクトのダンプ処理の記述が楽
（出力先の仮想化もおまけで付いてくる）

・printf()系の書式指定ミスとは無縁である

```
char name[];
// 名前を出力する
std::printf( “%d¥n”, name );
```

あ、間違えた……とかいうミス

ユニットテスト環境の構築

Boost::unittest ライブラリのビルド

unittest ライブラリは事前のビルドが必要

プラットフォームコンパイラ向けに・・・
- jam ファイル（ビルドスクリプト）の追加、変更
- unittest ライブラリのソースファイルの変更
- 標準ライブラリの代替実装

上記のものが揃ったら以下のコマンドでビルドできる（「???」はツールセット名）

```
bjam.exe --toolset=??? --with-test --stagedir=.
link=static threading=single,multi release stage
```

ユニットテスト環境の構築

Boost::unittest ライブラリのビルド

経験上変更の可能性が高いところをうっすらと・・・

Boost ビルドスクリプト（V1.40.0）
・まずツールセット（プラットフォームコンパイラ）名を決める（以下、「???」と表記）

・tools/build/v2/tools/???.jam
プラットフォームコンパイラ設定が既存のツールセット用の設定とは大きく異なる場合は
同ディレクトリの最も似ているコンパイラの設定ファイルをコピーして
ツールセット名の jam ファイルとして配置し、必要な部分だけ変更すると楽かも・・・

・tools/build/v2/tools/common.jam
（上記のように新規にツールセット用 jam ファイルを用意した場合）
toolset-tag ルールの switch にプラットフォームコンパイラを追加
case ??? : tag += ??? ;

ユニットテスト環境の構築

Boost::unittest ライブラリのビルド

経験上変更の可能性が高いところをうっすらと・・・

Boost 実装（V1.40.0）
- ・メイン関数置換（カスタムストリーム設定、引数の受け渡し、実行結果をコマンド側に返す）
  test/impl/unit_test_main.ipp
- ・ <ctime>を使用しているところ
  （<ctime>が標準仕様でない場合、プラットフォームSDKによる代替処理を施す）
  timer.hpp
  test/impl/framework.ipp
- ・環境変数を扱う部分
  （環境変数はおそらく実行環境ではアクセスできないので無効にする）
  test/impl/unit_test_parameters.ipp, test/utils/runtime/config.hpp,
  test/utils/runtime/env/environment.hpp, test/utils/runtime/env/variable.hpp,
  test/utils/runtime/file/config_file_iterator.cpp

ユニットテスト環境の構築

標準ライブラリの代替実装

プラットフォームコンパイラの標準ライブラリの実装が
不完全である場合は代替実装を施してそちらを使用する

・<ctime>
Boost::unittestにおいてはそれほど重要ではないので
代替実装使を用意するよりも使用しないように修正するとよいかも

・<stdexcept>
std::invalid_argumentなどが
もしかすると正常に動作しないかもしれないので要チェック
（標準例外を一切使用するつもりがなければ必要ない）

ユニットテスト環境の構築

ビルドシステムの実装

ユニットテストにおけるビルドシステムの詳細要求定義：

- ユニットテスト用実行ファイルをビルドできる
- ユニットテスト用実行ファイルを開発環境で実行できる
- 編集したソースに対応したユニットテストが実行できる

今開発で使っているビルドシステムに
これらを実現するルールを追加実装するのが近道

ユニットテスト環境の構築

（makeによる）ビルドシステムのテスト登録例

テスト対象ソースファイル：src/Hoge.h, src/Hoge.cpp
テストコードファイル　　：src/HogeTest.cpp

「ABSGM」におけるスクリプト例（makefile）：

```
# ユニットテストルールインスタンスを生成
$(call _new,UnitTestRule,unittest_rule)

# ユニットテストインスタンスを生成
$(call _new,UnitTest,unittest_Hoge)
# HogeTest.cpp をテストコードとする
unittest_Hoge.source_rpath := HogeTest.cpp

# Hoge テストを登録
$(call unittest_rule.add-test,unittest_Hoge)
```

弊社独自のビルドシステム「ABSGM」は
makefile用ライブラリで構成され、
オブジェクト指向風の記述が可能

この例ではユニットテストルールを生成し、
そのルールにテストコードを追加している
（テストコードに対応するテスト対象ソースは
テストコードファイル名から推論される）

ユニットテスト環境の構築

（makeによる）ビルドシステムのインターフェイス例

テスト対象ソースファイル：src/Hoge.h, src/Hoge.cpp
テストコードファイル　　：src/HogeTest.cpp

```
make unittest SRC=src/Hoge.h
```

```
make unittest SRC=src/Hoge.cpp
```

```
make unittest SRC=src/HogeTest.cpp
```

このようなコマンドをIDEやエディタから起動できるようにしておく

上記のどのコマンドを実行しても（テスト対象ソースとテストコードのどれを指定しても）同じテストが走るようにしておくのがミソ

ユニットテスト環境の構築

（makeによる）ビルドシステムのインターフェイス例

プロジェクト内のすべてのユニットテストを実行

```
make unittest
```

これは CI サーバーなどで実行する場合などに有効

テスト実行も通常のビルドと同様にソースファイル間の依存関係に
基づいて動作するので、変更の影響があるテストのみを
実行させることも可能

＊せっかくなので CI にはユニットテストも含めるとよい

ユニットテストコードのサンプル

```
#define BOOST_MODULE_NAME HogeTest
#include <boost/test/unit_test.hpp>

#include "Hoge.h"

BOOST_AUTO_TEST_CASE( GetHashTest )
{
    // 期待するハッシュ値
    static const unsigned char expected_hash[] = {
        32, 217, 38, 125, 45
    };

    Hoge hoge( 34 );
    unsigned char result[16];

    // ここでは Hoge::GetHash() はバッファに 5 バイトのハッシュ値を取得するという仕様
    BOOST_CHECK_EQUAL( 5, hoge.GetHashLength() );
    hoge.GetHash( result );
    BOOST_CHECK_EQUAL_COLLECTIONS( result, result + sizeof(expected_hash),
                                   expected_hash, expected_hash + sizeof(expected_hash) );
}
```

ユニットテストのための知識と設計

ユニットテストを有効に活用するための設計

- 適切な粒度で分割する
- 継承よりもオブジェクトコンポジションを
- 「凝集度」を高く、「結合度」を低く
- 継続的リファクタリングをしないと破綻するかも

ユニットテストのための知識と設計

NVI（Non Virtual Interface）イディオム

```
// class std::basic_streambuf
{
public:
    // インターフェイス関数
    std::basic_streambuf<charT,traits>*
    pubsetbuf( char_type* s, std::streamsize n )
    { return setbuf( s, n ); }
protected:
    // この関数をサブクラスでカスタマイズする
    virtual std::basic_streambuf<charT,traits>*
    setbuf( char_type* s, std::streamsize n );
}

class ConsoleOutputStreamBuf
    : public std::basic_streambuf<...>
{
    virtual std::basic_streambuf<charT,traits>*
    setbuf( char_type* s, std::streamsize n )
    {
        buf_ = s; buf_size_ = n;
        this->setp( buf_, buf_ + buf_size_ - 1 );
        return this;
    }
}
```

インターフェイス関数を virtual にしないというイディオム

std::streambuf などにも使われている

pubsetbuf() が インターフェイス関数で
サブクラスは setbuf() を実装する

（ついでにカスタムストリームバッファの
setbuf()の実装例）

ユニットテストのための知識と設計

NVI（Non Virtual Interface）イディオム

```
// インターフェイスクラス
class IHoge
{
public:
    // インターフェイス関数
    int Func()
    {
        // ここで事前条件をチェックする

        // 実装関数を呼ぶ
        const int result = FuncImpl();

        // ここで事後条件をチェックする

        return result;
    }

private:
    // サブクラスではこの関数を実装する
    virtual int FuncImpl() = 0;
};
```

インターフェイス関数の中で
事前条件と事後条件をテストする

↓

インターフェイスクラスに対するテストコードを
用意すれば、すべてのサブクラスで満たすべき
条件をテストできる

「契約による設計（DbC）」の実践

ユニットテストのための知識と設計

C++の例外の取り扱い

```
class A
{ /* ... */ };
class B
{ /* ... */ };

// テスト関数
void TestAandB( A* pa, B* pb )
{
    /* ... */
    delete pa;
    delete pb;
}

// テストコード
try
{
    // 例外安全ではないコード
    TestAandB( new A, new B );
}
catch (...)
{ /* ... */ }
```

TestAandB()の呼び出し部分は例外安全ではない

class A または class B のコンストラクタで
例外が投げられると pa か pb が指すはずの
オブジェクトのどちらかは解体されない可能性が
あるのでこのコードはメモリリークを
発生させることがある

さらなる工夫

メモリ関連のエラー検出

メモリアロケータがメモリの不整合を検出できる

↓

ユニットテストの終了時に検出すればよい

メモリリークの紙一重原理：

まだ解放していない ←→ メモリリーク

ユニットテストはプログラムに「終了」を与えるので
メモリリークをとらえる絶好の機会

さらなる工夫

処理系を越える

プラットフォームSDKなどに依存しないコード

↓

どの処理系でも動作する

↓

短時間で実行できる処理系でテストすればよい

ビルドスクリプトやビルドコマンドパラメータによって
テストを実行する処理系を指定できるようにしておけば便利

＊automake の configure などとは目指しているものが異なる

さらなる工夫

「契約による設計（DbC）」の実現

```
// 事前条件違反例外
class PreconditionErrorException : public std::logic_error
{
public:
    PreconditionErrorException(
      const char* condition, const char* file, int line );
    /* ... */
};

// 事前条件チェック関数
// PRECONDITION_ASSERT( condition )
#ifdef BUILD_TEST_   // ユニットテスト時のみ定義される

  #define PRECONDITION_ASSERT( condition )  ¥
  do {  ¥
    if ( !(condition) ) {  ¥
        throw PreconditionErrorException(  ¥
          #condition, __FILE__, __LINE__ ); }  ¥
  } while ( false )

#else
  #define PRECONDITION_ASSERT( condition )  ¥
    assert( condition )
#endif
```

事前条件チェック用コードの
最も単純なバージョン

事前条件違反例外クラスと
事前条件チェック用マクロを定義

事前条件違反を検出したときは
例外を投げるか停止する

さらなる工夫

「契約による設計(DbC)」の実現

```
class Hoge
{
public:
    // テストする関数 (NVI)
    void Func( int value ) {
        // 事前条件チェック
        PRECONDITION_ASSERT( 2 <= value && value < 10 );
        FuncImpl( value );
    }
private:
    virtual void FuncImpl( int value ) {
        /* 処理 */
    }
}

// テストコード
BOOST_AUTO_TEST_CASE( PreconditionTest )
{
    Hoge hoge;

    // 事前条件違反例外が投げられれば正常
    BOOST_CHECK_THROW( hoge.Func( 1 ),  PreconditionErrorException );
    BOOST_CHECK_THROW( hoge.Func( 10 ), PreconditionErrorException );
}
```

事前条件チェック用コードを
使用して DbC を実現

**このテストコードは
「事前条件(契約)違反検出」の
正確性をテストしている**

**ユニットテストで Hoge クラスが
どんな契約違反も検出することを
確認しておくと、通常実行時に
Hogeクラスの使用側の
契約違反を確実に検出できる**

今後の展望

- 技術以外の領域の問題解決
- テストの効果を定量化したい
- 導入した経験談を聞きたい
- ゲーム業界でも体系化されたテスト技術を盛り上げたい

参考

CEDEC
- 『バグを限りなくゼロにする方法』（2009）
  株式会社セガ　伊藤　周さん

JaSST
- JaSST Kansai ’10
  - 『ユーザ視点からの提言 ～ 一般製造業の品質の考え方 ～』
  - 『テストの読み・書き・そろばん』

書籍
- 『オブジェクト指向入門 第2版 方法論・実践』（2008）
  Bertrand Meyer・翔泳社

質疑応答

お手柔らかに…

ご清聴ありがとうございました！

ユニットテスト導入の技術的課題とその解決例

2010年9月1日

Ubisoft Osaka 株式会社
益　弘和
mail : hirokazu.eki@ubisoft.com