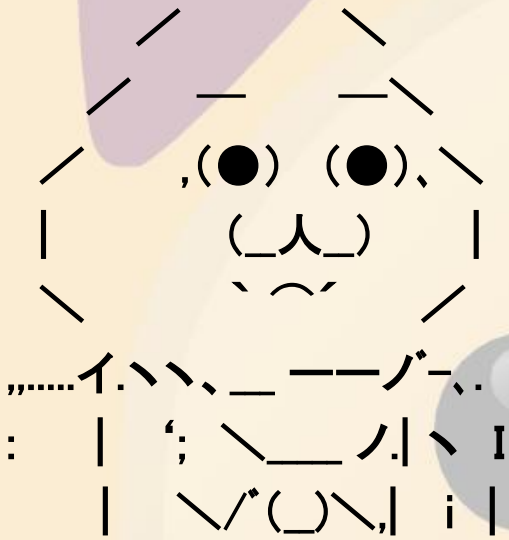
匠の伝承w

マルチな時代の設計と開発

パート3

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。_|￣|○

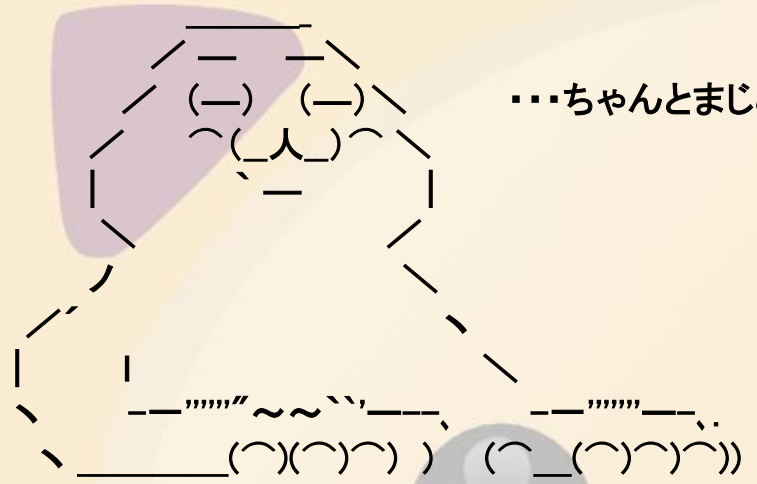
株式会社シーソフト代表取締役です。

現役のエンジニアです。プログラム書いてます。

メールソフト Becky! 用の BkReplyer という作品が微妙に有名らしす。
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



・・・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。

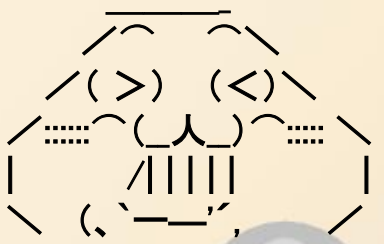
オブジェクト指向は『モノ』をとらえる。

『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

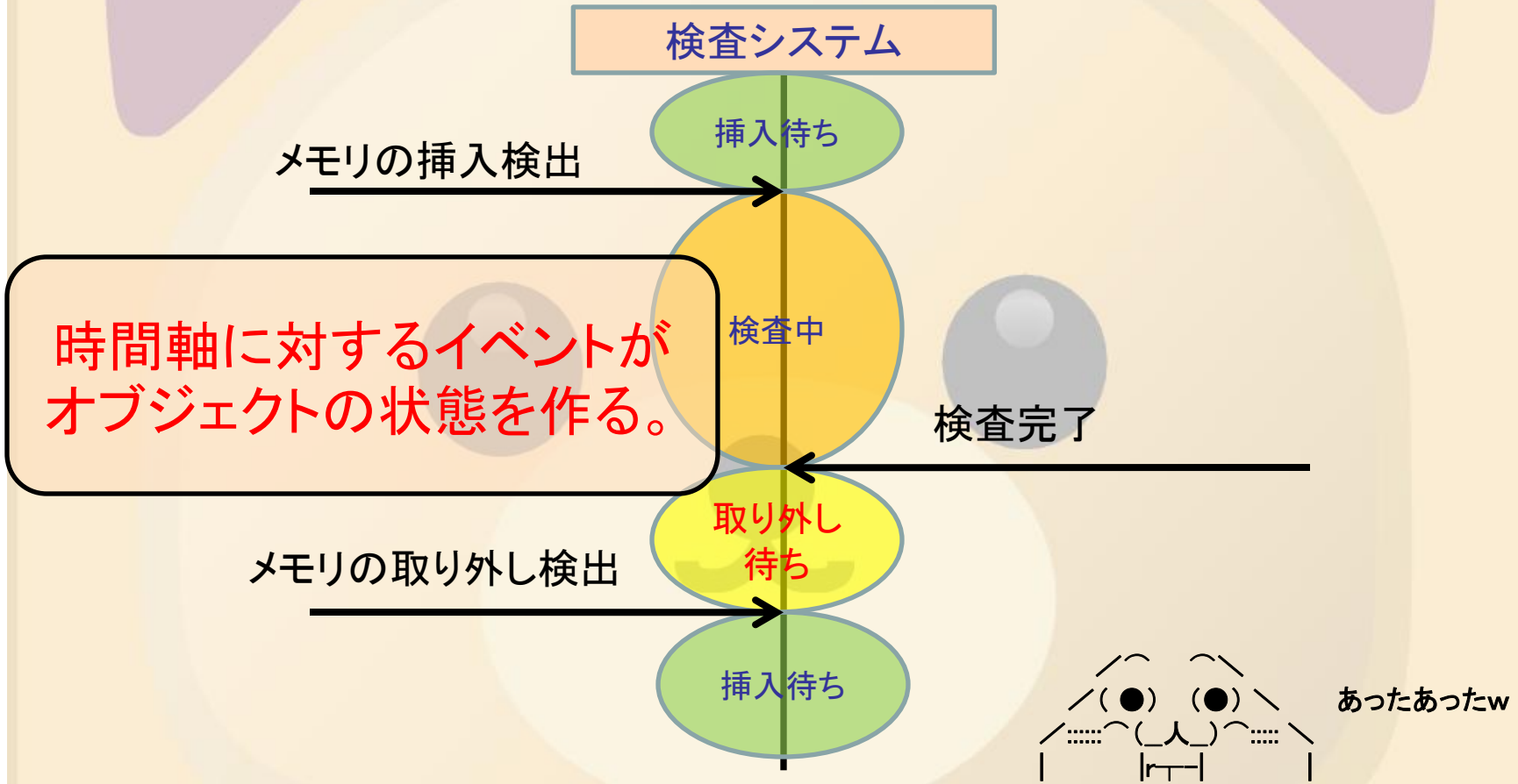


非同期の事象はループを分断して考える。

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。

イベントトレースと状態の関係



あったあったw

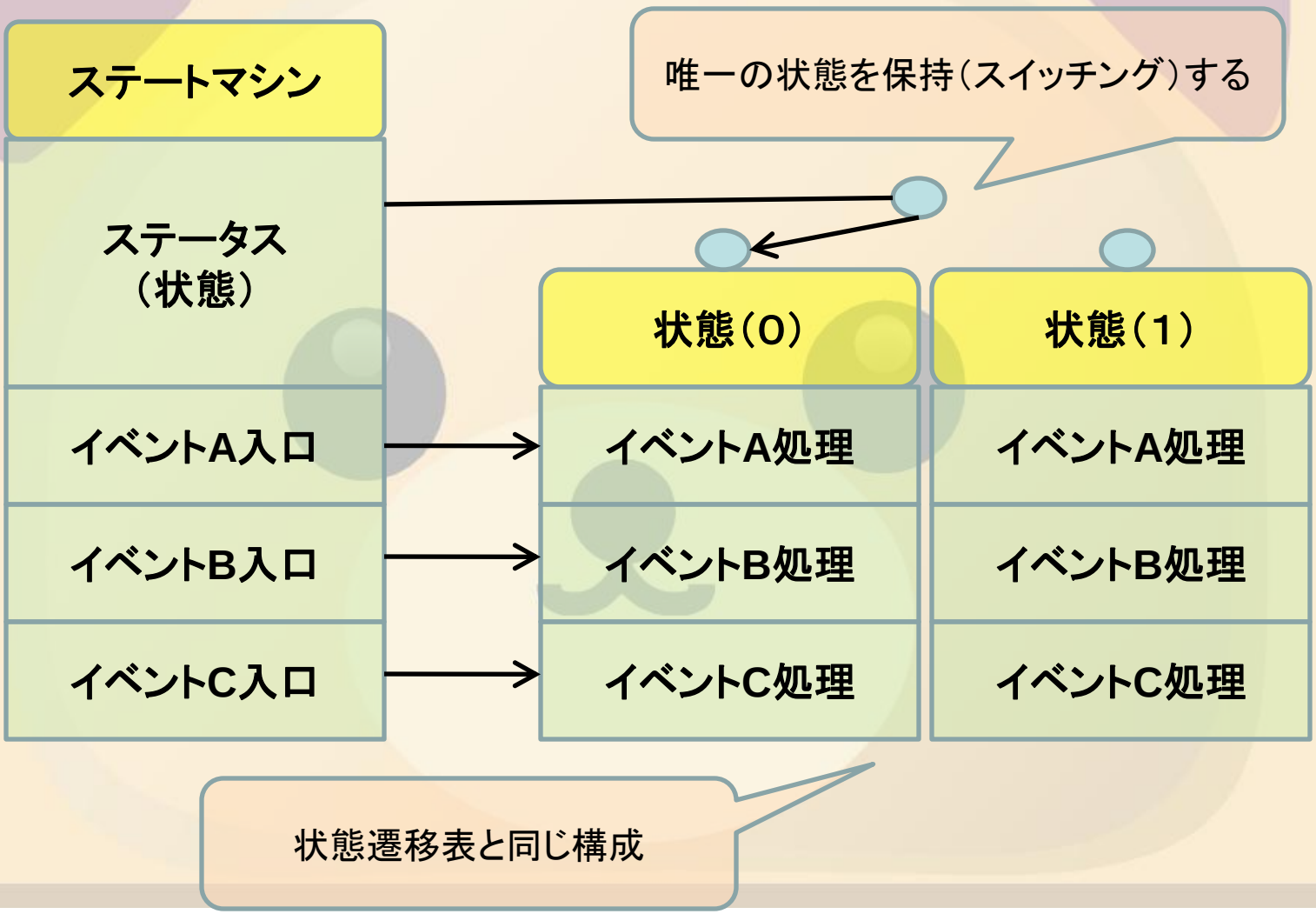
状態遷移表

状態 / イベント	メモリ挿入待ち (0)	検査中 (1)	取り外し待ち (2)
メモリ挿入検出	検査を開始 状態を検査中に →(1)	エラー表示 検査中断 →(2)	ログ記録 検査を開始 →(1)
検査完了通知	ログ記録 →(0)	検査結果を表示 →(2)	ログ記録 →(2)
メモリ取り外し検出	ログを記録 →(0)	エラー表示 検査中断 →(0)	次の検査準備 →(0)

マトリクスを必ず埋める

状態別にイベント発生時の処理を書く

ステートパターンの構成要素



本日のテーマ

時間軸に対するイベントによって状態を変化させるオブジェクトには、通知の受け口が必要。

オブジェクトに対してイベントをどのように通知すればよいのか？

クラスの考え方

社員コード
氏名
生年月日

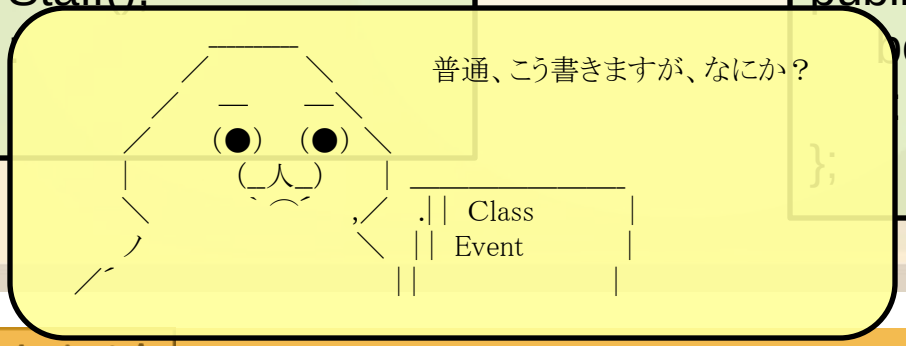
```
class Staff
{
    long  Id;
    string Name;
    DateTime Birthday;
public:
    Staff();

};
```

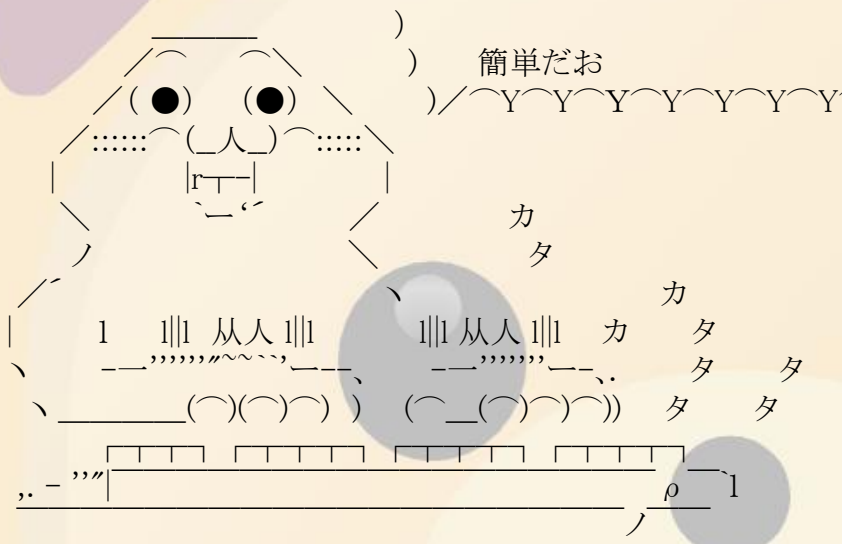
ファイル名
ファイルハンドル
バッファ
バッファの大きさ

```
class File
{
    string Name;
    HANDLE Handle;
    void *Buffer;
    size_t BufferSize;
public:
    bool Open( string name );

};
```



クラスの要素は？

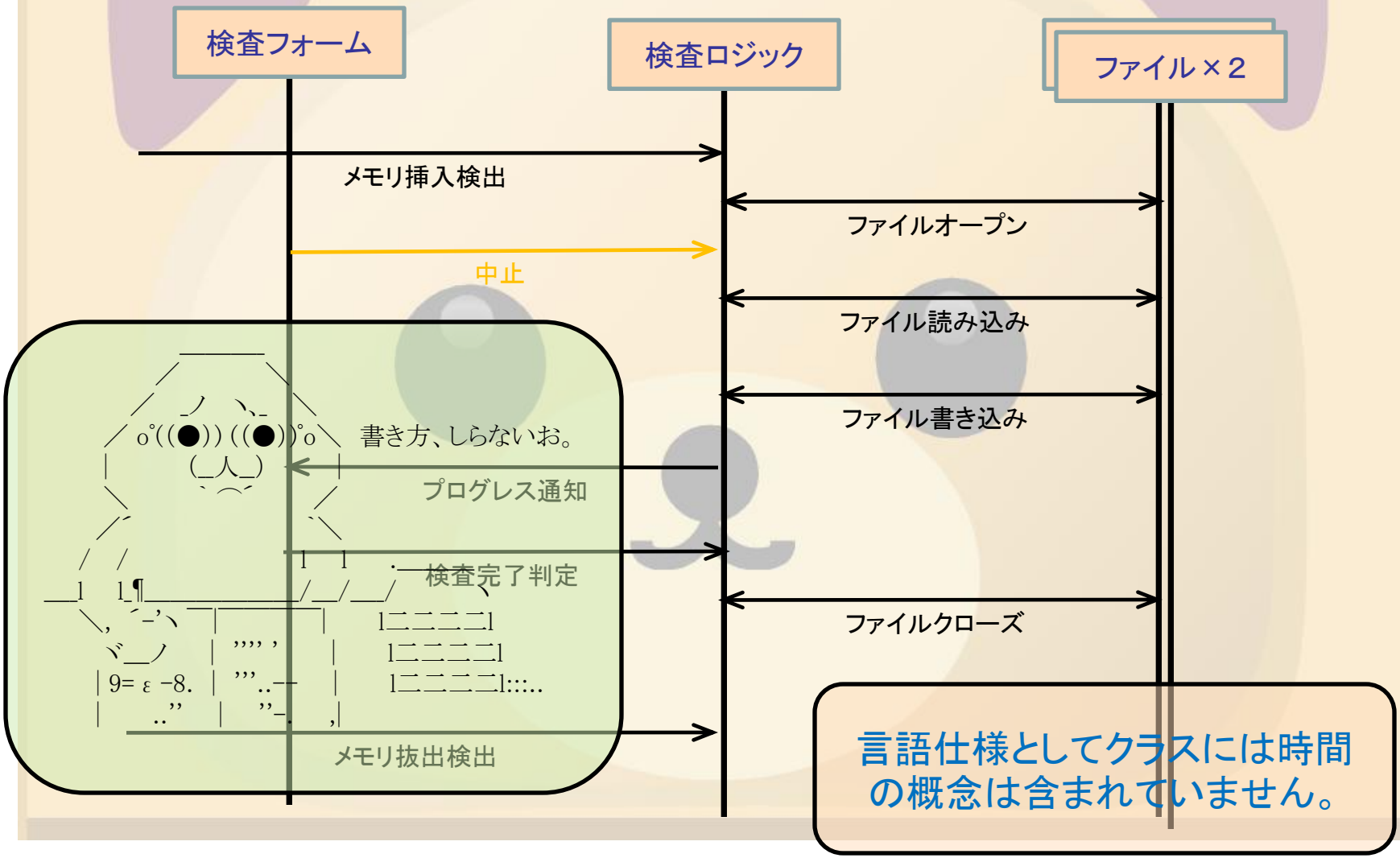


メンバ（データ）

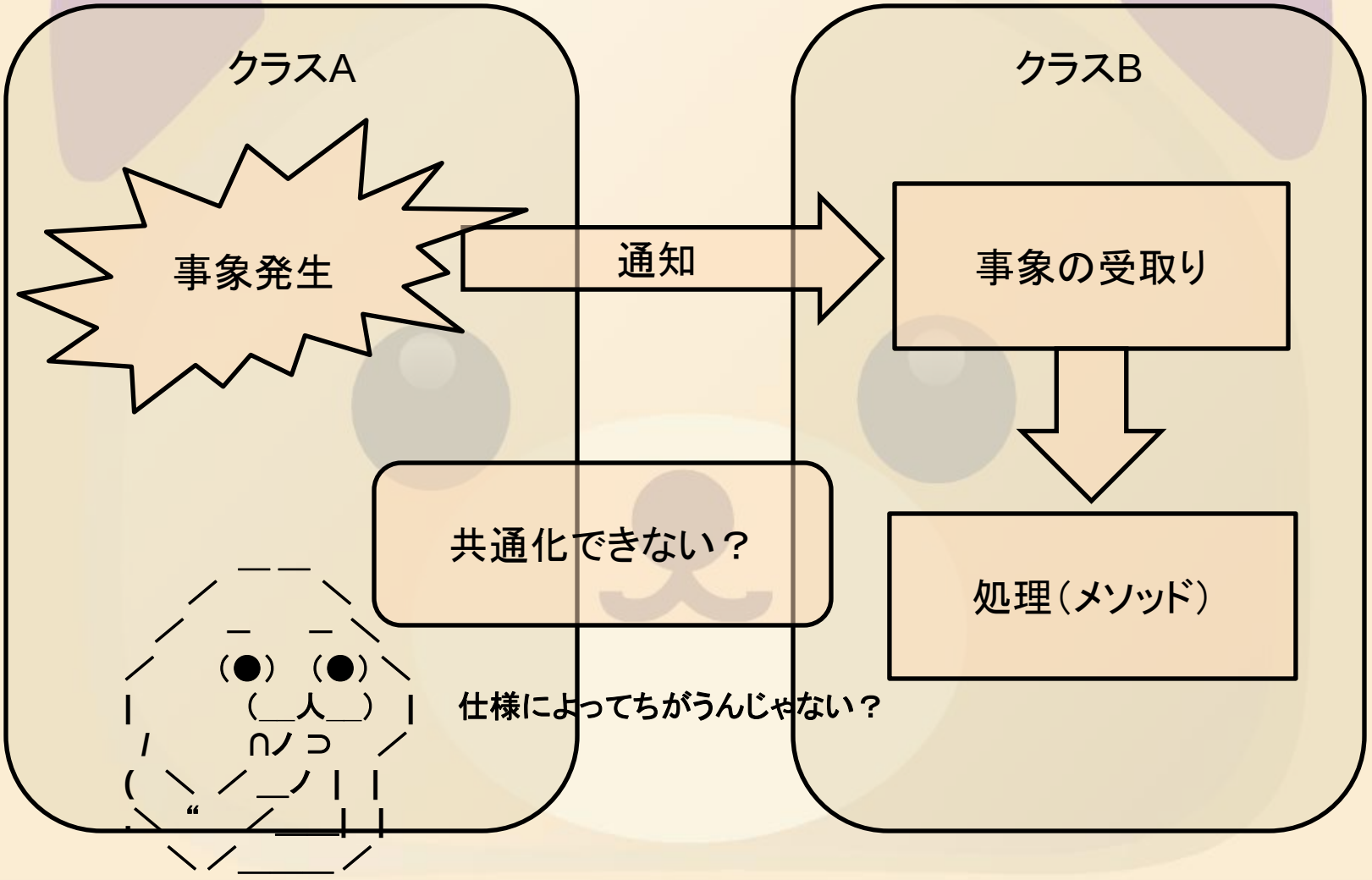
メソッド（処理）

時間軸！？

クラスに「時間」の概念を追加する



イベントの発行と受け口の考察



事象(イベント)って？

関数呼び出し
ウィンドウメッセージ
シグナル
割り込み
ミューテックス
キュー
セマフォ
：
：

通知する内容

通知する方法

通知すべき相手

受け取る方法

設計仕様にあわせて
個別に実装される

要素の考察

通知側のクラス
（通知者）

受取側のクラス
（観察者）

通知情報

通知方法

受取方法

通知先

オブザーバ(観察者)パターン

通知側のクラス

受取側のクラス
（観察者）

Observable
（通知情報）

Notify()
（通知方法）

OnNotify()
（受取方法）

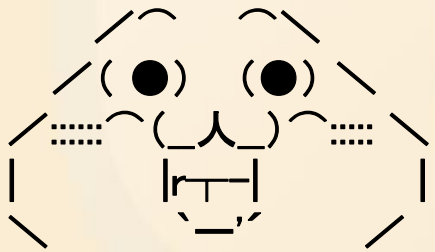
Observers
（通知先）

通知情報の基底クラス

```
class Observable
{
public:
    Observable(){ }
    virtual ~Observable(){}
};
```

Observable.h

クラス間の通信情報は、仕様ごとに異なる。
基底クラスでは、実体を持たないクラスとして定義。



からっぽ！？

観察者の基底クラス

```
#include "Observable.h"

class Observer
{
public:
      Observer(){}
    virtual ~Observer(){}
      // 受け取ったときの振る舞いを処理する仮想メソッド
    virtual void OnNotify( Observable * data ) = 0;
};
```

Observer.h

通知情報の型

受取方法

情報配信者の基底クラス

```
#include <vector>
#include "Observable.h“
class Observer;
class ObserverSubject
{
protected:
    std::vector<Observer *> Observers;
public:
    ObserverSubject();
    virtual ~ObserverSubject();

    void Add( Observer *obvserver );
    void Remove( Observer *observer);

    virtual void Notify( Observable * data );
};
```

ObserverSubject.h

通知情報の型

観察者のリスト

観察者の追加と削除

通知

```
#include <algorithm>
#include "observer.h"
#include "ObserverSubject.h“
ObserverSubject::ObserverSubject(){}
ObserverSubject::~ObserverSubject(){}
void ObserverSubject::Add(Observer *observer ){
    Observers.push_back(observer);
}
void ObserverSubject::Remove(Observer *observer){
    std::vector<Observer *>::iterator it;
    it = std::find( Observers.begin(), Observers.end(), observer );
    if( it != Observers.end() ){   Observers.erase( it ); }
}
void ObserverSubject::Notify( Observable * data ){
    std::vector<Observer *>::iterator it;
    for( it = Observers.begin(); it != Observers.end(); ++it ){
        (*it)->OnNotify( data );
    }
}
```

ObserverSubject.cpp

観察者の追加

観察者の削除
（リストから observer を見つけて削除）

通知処理

ObserverにOnNotify()があることを知っている

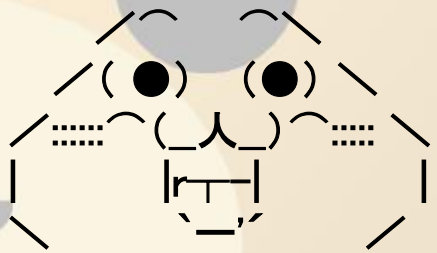
プログレスバーを動かしてみよう

LinearPosition.h

```
#include "Observable.h"

class SizeInformation : public Observable{
protected:
    int TotalSize;                  // 全体の大きさ
public:
    SizeInformation( int totalSize ){    TotalSize = totalSize;   }
    int GetTotalSize(){ return TotalSize; }
};
class LinearPosition: public Observable{
protected:
    int Position;                   // 現在位置
public:
    LinearPosition( int position ){      Position = position;    }
    int GetPosition(){ return Position; }
};
```

大きさの通知



だんだん具体的になってきたお

現在位置の通知

プロパティ使いたい(^◇^;

観察者(受取側)

LinearPositionObserver.h

```
#include "Observer.h“
class CProgressCtrl;
class LinearPositionObserver : public Observer{
private:
      CProgressCtrl *Progress;

public:
    LinearPositionObserver( CProgressCtrl *progress ){
        Progress = progress;
    }

protected:
    virtual void OnNotify( Observable* data );
};
```

変更するプログレスバー

コンストラクタでプログレスバーを受け取り

受取側の処理

LinearPositionObserver.cpp

```
#include "LinearPosition.h"
#include "LinearPositionObserver.h"

void LinearPositionObserver::OnNotify( Observable* data ){
	SizeInformation *sizeInformation = dynamic_cast<SizeInformation *>( data );
	if( sizeInformation != NULL ){
		Progress->SetRange( 0, sizeInformation->GetTotalSize(); );
		return;
	}
	LinearPosition *linearPosition = dynamic_cast<LinearPosition *>( data );
	if( linearPosition != NULL ){
		Progress->SetPos(linearPosition->GetPosition() );
		Progress->UpdateData( FALSE );
	}
}
```

プログレスバーの範囲を設定する

プログレスバーのポジションを変更し描画

メッセージ配信側

```
#include “ObserverSubject.h”
class LinearPositionObserverSubject : public ObserverSubject
{
public:
    LinearPositionObserverSubject();
    virtual ~LinearPositionObserverSubject();
};
```

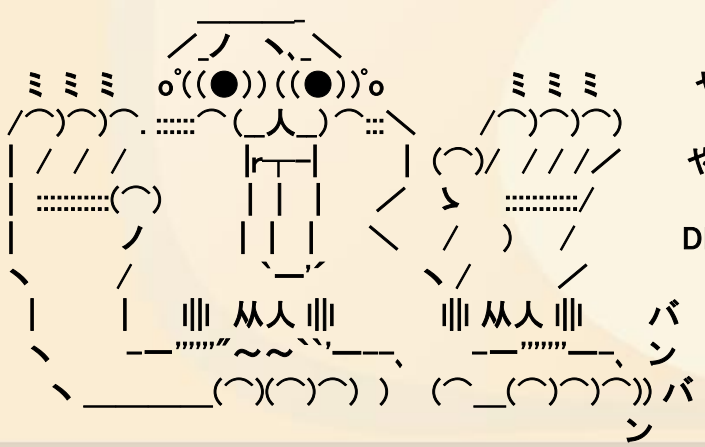
観察者の追加と削除、情報の配信
方法は基底クラスで実装済み

つまり、このクラスはとりあえずいらないってこと！

LinearPositionObserverSubject.h

DEMO　1

実際に動かしてみよう。



DEMO1の大まかな構造

通知側のクラス
(CopyFileLogic)

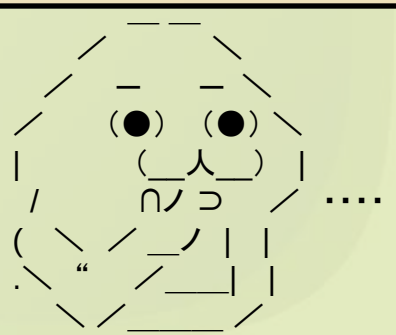
受取側のクラス
（demo1Dlg）

LinearPosition
（通知情報）

Execute()
の内部でNotify()

LinearPositionObserver::
OnNotify()でプログレスバーを変化

Observers
（通知先）



Windowsメッセージでの通知

WindowObservable.h

```
#include <windows.h>
#include "Observable.h"

class WindowObserver;
class WindowObservable : public Observable{
      friend class WindowObserver;
protected:
    UINT                Message;
    WPARAM              WParam;
    LPARAM              LParam;
public:
    WindowObservable( UINT message, WPARAM wParam, LPARAM lParam ) {
        Message         = message;
        WParam          = wParam;
        LParam          = lParam;
    }
};
```

ずるw。
プロパティ使いたくなる今日この頃（笑）

本格的だお

Windowsメッセージ観察者クラス

WindowObserver.h

```
#include <windows.h>
#include "Observer.h“
#include "WindowObservable.h“
class WindowObserver : public Observer{
protected:
    HWND Handle;
public:
    WindowObserver( HWND handle ){  Handle = handle;   }
protected:
    virtual void OnNotify( Observable * data ) {
        WindowObservable *msg = dynamic_cast<WindowObservable *>( data );
        if( msg != NULL ){
            ::SendMessage( Handle, msg->Message, msg->WParam, msg->LParam );
        }
    }
};
```

観察者のウィンドウハンドル

friend で直接アクセスw

実際にはここでの受け取りではないわけでw
Windows が独自の受け口を用意するので、そこを利用します。

通知情報を変更しましょう

LinearPosition.h

```
#include <windef.h>
#include "WindowObservable.h“
static const UINT MSG_PROGRESSTOTALSIZE   = ( WM_APP + 10 );
static const UINT MSG_PROGRESSPOSITION    = ( WM_APP + 11 );
class SizeInformation : public WindowObservable{
public:
    SizeInformation( int totalSize )
        : WindowObservable( MSG_PROGRESSTOTALSIZE, 0, totalSize ){ }
};
class LinearPosition: public WindowObservable{
public:
    LinearPosition( int position )
        : WindowObservable( MSG_PROGRESSPOSITION, 0, position ){}
};
```

メッセージの定義

サイズの通知

位置の通知

コードの変更部分

demo2Dlg.cpp

```
BEGIN_MESSAGE_MAP(Cdemo2Dlg, CDialog)
    :
    ON_MESSAGE( MSG_PROGRESSTOTALSIZE,      OnProgressTotalSize )
    ON_MESSAGE( MSG_PROGRESSPOSITION,       OnProgressPosition )
END_MESSAGE_MAP()
LRESULT Cdemo2Dlg::OnProgressTotalSize( WPARAM wParam, LPARAM lParam )
{
    Progress.SetRange( 0, (int)lParam );
    UpdateWindow();
    return 0;
}
LRESULT Cdemo2Dlg::OnProgressPosition( WPARAM wParam, LPARAM lParam )
{
    Progress.SetPos( (int)lParam );
    UpdateWindow();
    return 0;
}
```

メッセージマップを配置

サイズ変更

ポジション変更

DEMO 2

Windowsメッセージを使った事象通知

DEMOが2つもあるお

時間大丈夫かお？

DEMO2の大まかな構造

通知側のクラス
(CopyFileLogic)

受取側のクラス
（demo2Dlg）

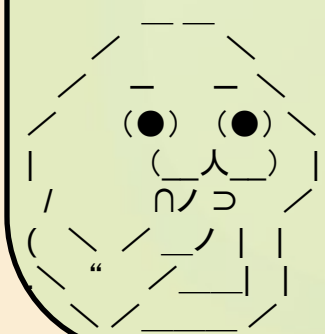
LinearPosition
（メッセージと値の定義）

Execute()
の内部でNotify()

OnNotify()ではPostMessage()だけ

Observers
（通知先）

OnMsgXXXXで自分自身のコントロール
（プログレスバー）を変化



独立性が高くなったお

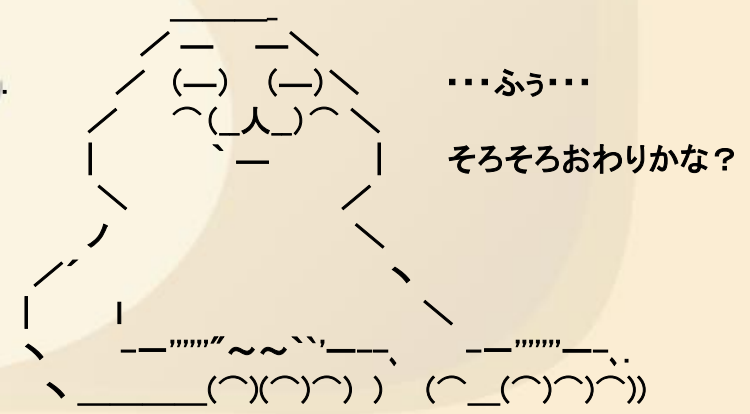
まとめ

オブザーバパターン：オブジェクト間の事象通知

Observable：発信側と受取側の共通情報を抽象化

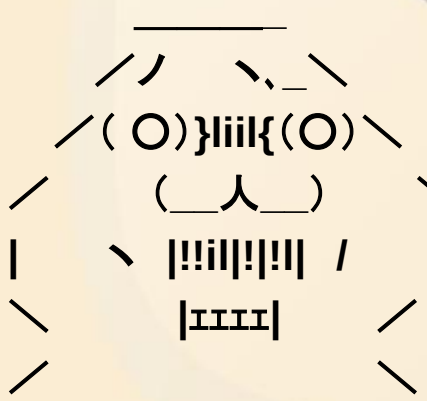
Observer：事象の受け取り方を抽象化

ObserverSubject：複数の観察者に通知



ステートパターンはどうなった！？

オブザーバの説明は、ステートパターンの説明からはじまっているのにww



あ！？。

そういえば。

ファイルコピーの状態遷移表

状態 / イベント	アイドル (0)	オープン中 (1)	書き込み待ち (2)
オープン依頼	ファイルを開く成功→(1) 失敗→(0)	ログ記録 →(1)	ログ出力 ファイルを閉じ、削除する →(0)
クローズ依頼	ログ記録 →(0)	ファイルを閉じる →(0)	ログ出力 ファイルを閉じ、削除する →(0)
読み込み依頼	ログを記録 →(0)	読み込み処理 継続→書き込み依頼(2) 終了→クローズ依頼(1)	ログ出力 ファイルを閉じ、削除する →(0)
書き込み依頼	ログを記録 →(0)	ログ出力 →(0)	書き込み処理 読み込み依頼→(1)
中止依頼	→(0)	ファイルを閉じ、削除する →(0)	ファイルを閉じ、削除する →(0)

ファイルコピーのステートマシン化

CopyFileState.h

```
class CopyFileLogic;
class CopyFileStateMachine;
class CopyFileState{
protected:
    CopyFileLogic               *Logic;
    CopyFileStateMachine        *StateMachine;
public:
    CopyFileState( CopyFileLogic *logic, CopyFileStateMachine * stateMachine ) {
        Logic                   = logic;
        StateMachine            = stateMachine;
    }
    virtual ~CopyFileState(){}
    virtual void OnOpen() = 0;
    virtual void OnClose() = 0;
    virtual void OnReadNext() = 0;
    virtual void OnWriteNext() = 0;
    virtual void OnCancel() = 0;
};
```

それぞれの状態別クラスを派生

CopyFileIdleState.h

```
#include "CopyFileState.h";

class CopyFileIdleState : public CopyFileState
{
public:
    virtual void OnOpen();
    virtual void OnClose();
    virtual void OnReadNext();
    virtual void OnWriteNext();
    virtual void OnCancel();
};
```

ステートマシンを実装

CopyFileIdleStateMachine.h

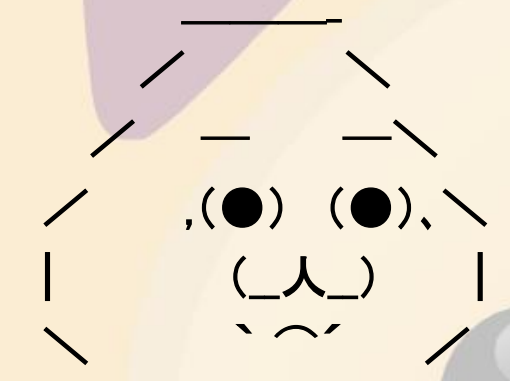
```
#include "CopyFileIdleState.h";
#include "CopyFileOpenState.h";
#include "CopyFileWriteState.h";
class CopyFileIdleStateMachine
{
protected:
    CopyFileIdleState *IdleState;
    CopyFileOpenState *OpenState;
    CopyFileWriteState *WriteState;
    CopyFileState *Status;
public:
    virtual void OnOpen();
    virtual void OnClose();
    virtual void OnReadNext();
    virtual void OnWriteNext();
    virtual void OnCancel();
};
```

DEMO3
説明もややこしいので、さっそく・・・

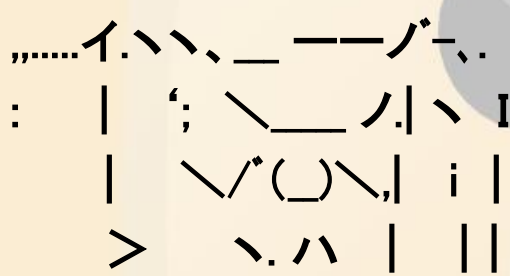
ほんとに時間大丈夫かお？

発表時間どころか
いそがしくて、
サンプル間に合いません
でした。

ごめんなさい。



まあ、ふいんき（へんかんできないお）は、十分に伝わったことにしておきましょう。

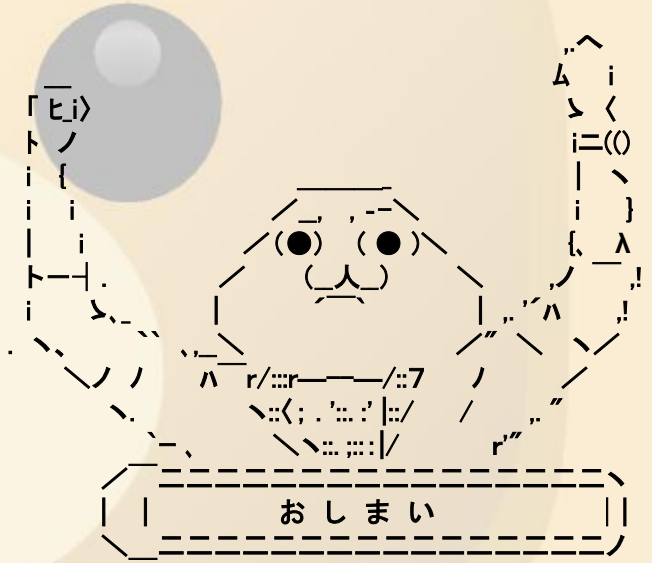


次回は、依存性をなくそうという試みをまじえて本シリーズは次のステップに進む予定です。
より進化した斬新なアプローチへと展開か！？

ご静聴ありがとうございました。

m(_._)m

あんまり期待しないでね



Special thanks for Yaruo charactors