.NETで始めるオブジェクト指向プログラミング入門
～俺流オブジェクト指向をあなたに～

もり　ひろゆき（ひろえむ）
わんくま同盟
http://blogs.wankuma.com/hirom/

Agenda

オブジェクト指向とは？

オブジェクト指向プログラミングことはじめ。

ちょっと概念説明。

間に合えば･･･実際に作ります！

私にとってのオブジェクト指向

- オブジェクト指向とは

整理術！

己を知る

明日の自分へのメッセージ

OgnacさんのBlog[**Ognacの雑感**]1月7日のエントリより

「職人は、明日楽して手を抜く為に,今日は如何なる努力も惜しまない。」

オブジェクト指向の現状

人(資料)によって説明のしかたが変わってしまう。

- 同じもののはずなのに・・・
- 具体的なものではなく、従来の開発方法の問題点に対する解決方法。

技術的側面と概念的側面

- とりあえず技術から入れば・・・

言語？開発？アーキテクチャ？

- OOと名のつくものが多くて混乱　ー　実はすべて支援技術
- 大切なのはどうやって開発していけばいいか。

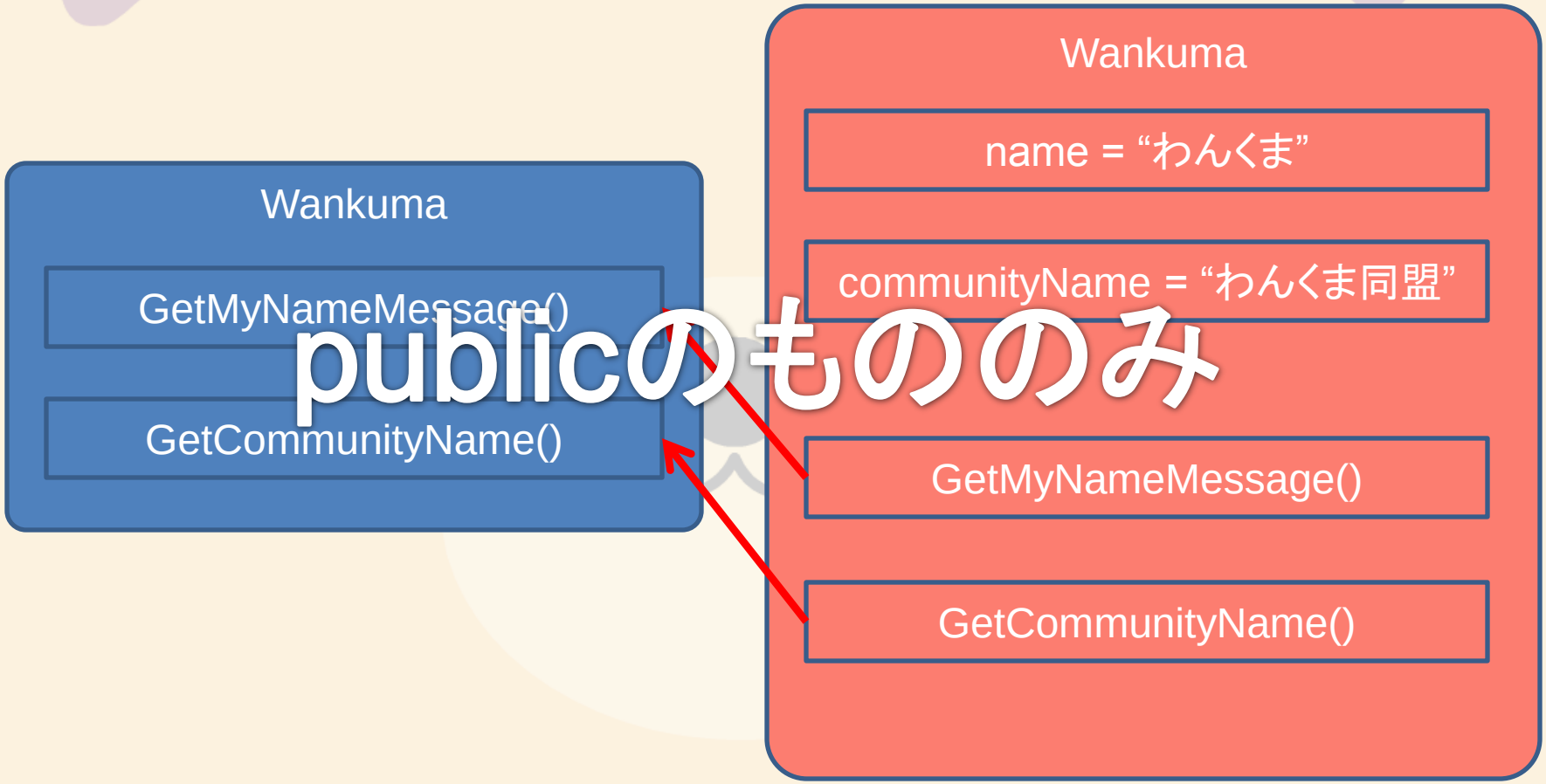
それではオブジェクト指向言語で見てみましょう

それではちょっとおさらいです。

クラス定義メカニズム

Wankuma kuma; [C#]　　Dim kuma As Wankuma[VB]

Kuma = new Wankuma(); [C#] kuma = New Wankuma()[VB]



クラスの特徴

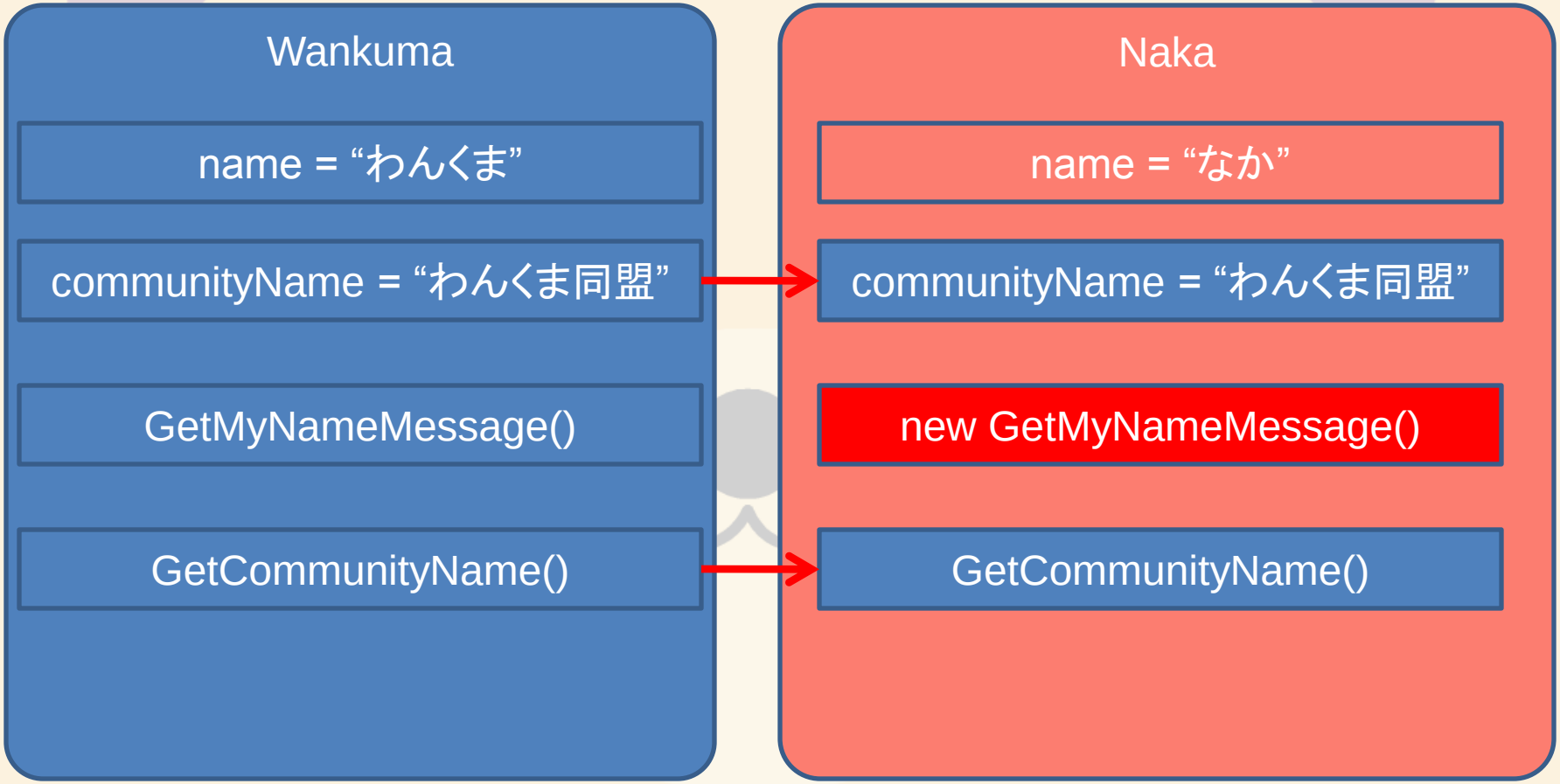
- データ(属性)と処理(操作)をひとまとめにした塊
- オブジェクトの型
- クラスのみでは実行できない
- インスタンス（オブジェクト）を作成する必要がある。
- ・・・・つまり
- 実体へのアクセスを遅らせることもできる。
- 実体そのものを後から代入することができる。

続けて中さん

中さんクラスを追加してみましょう

Demo

newで再定義ができる!

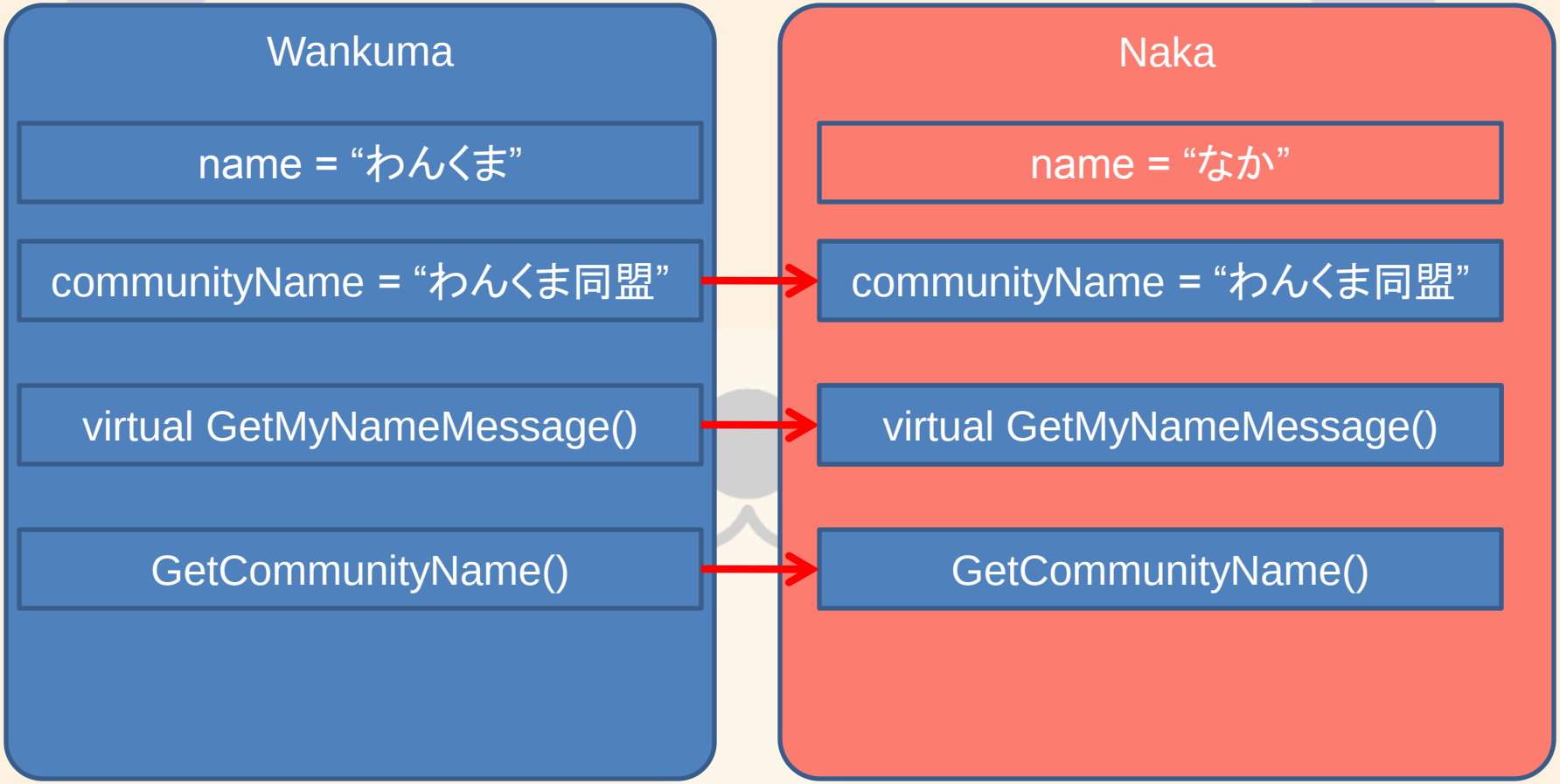


ただ問題が・・・・

実はWakumaクラスとしても定義できたりする

Demo

仮に・・・本物は・・・

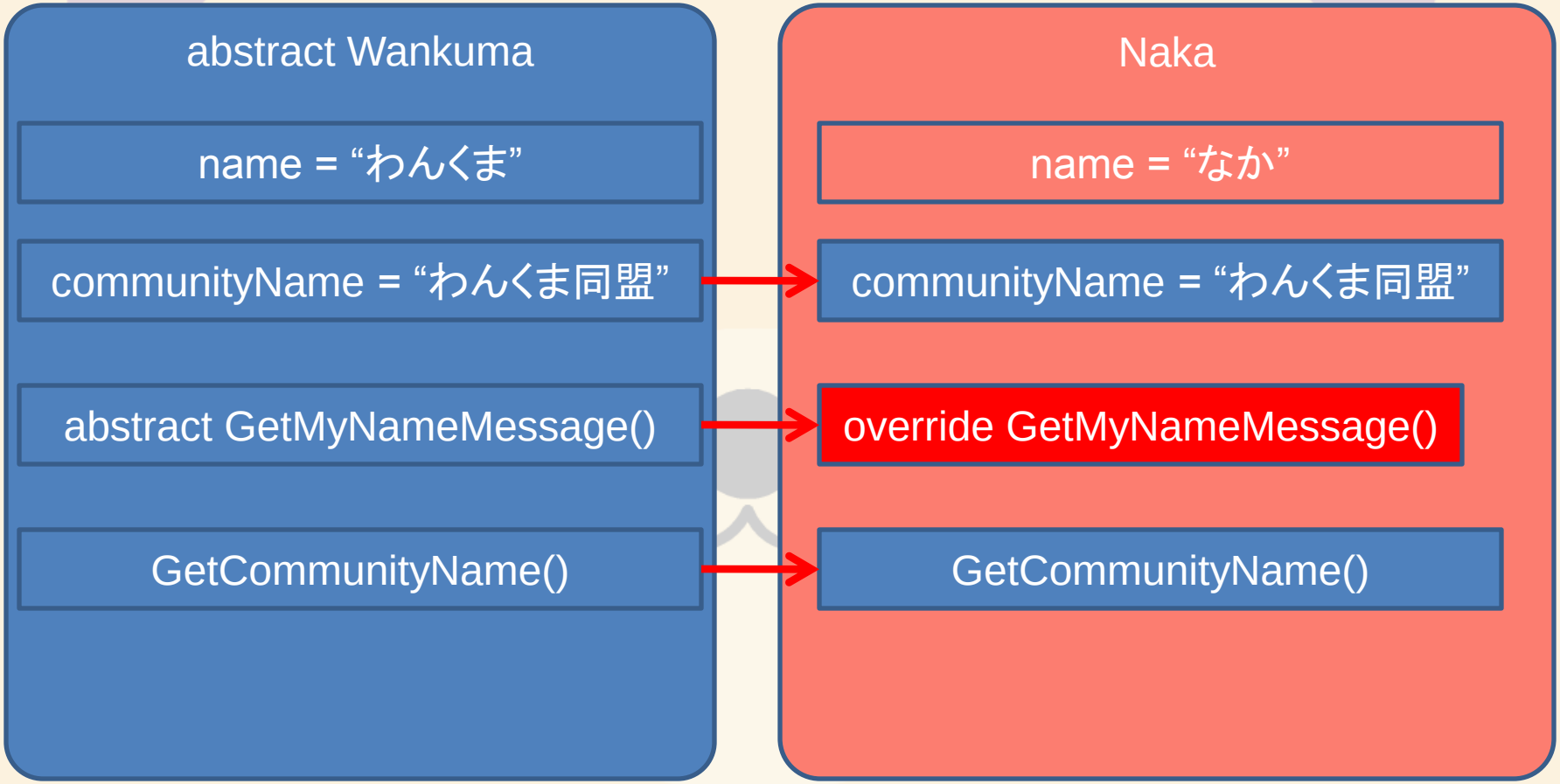

Wankuma
name = “わんくま”
communityName = “わんくま同盟”
virtual GetMyNameMessage()
GetCommunityName()
Naka
name = “なか”
communityName = “わんくま同盟”
virtual GetMyNameMessage()
GetCommunityName()

共通で使えるところだけ・・・

みんな同じことをするところだけまとめちゃえ

Demo

抽象クラス・具象クラス

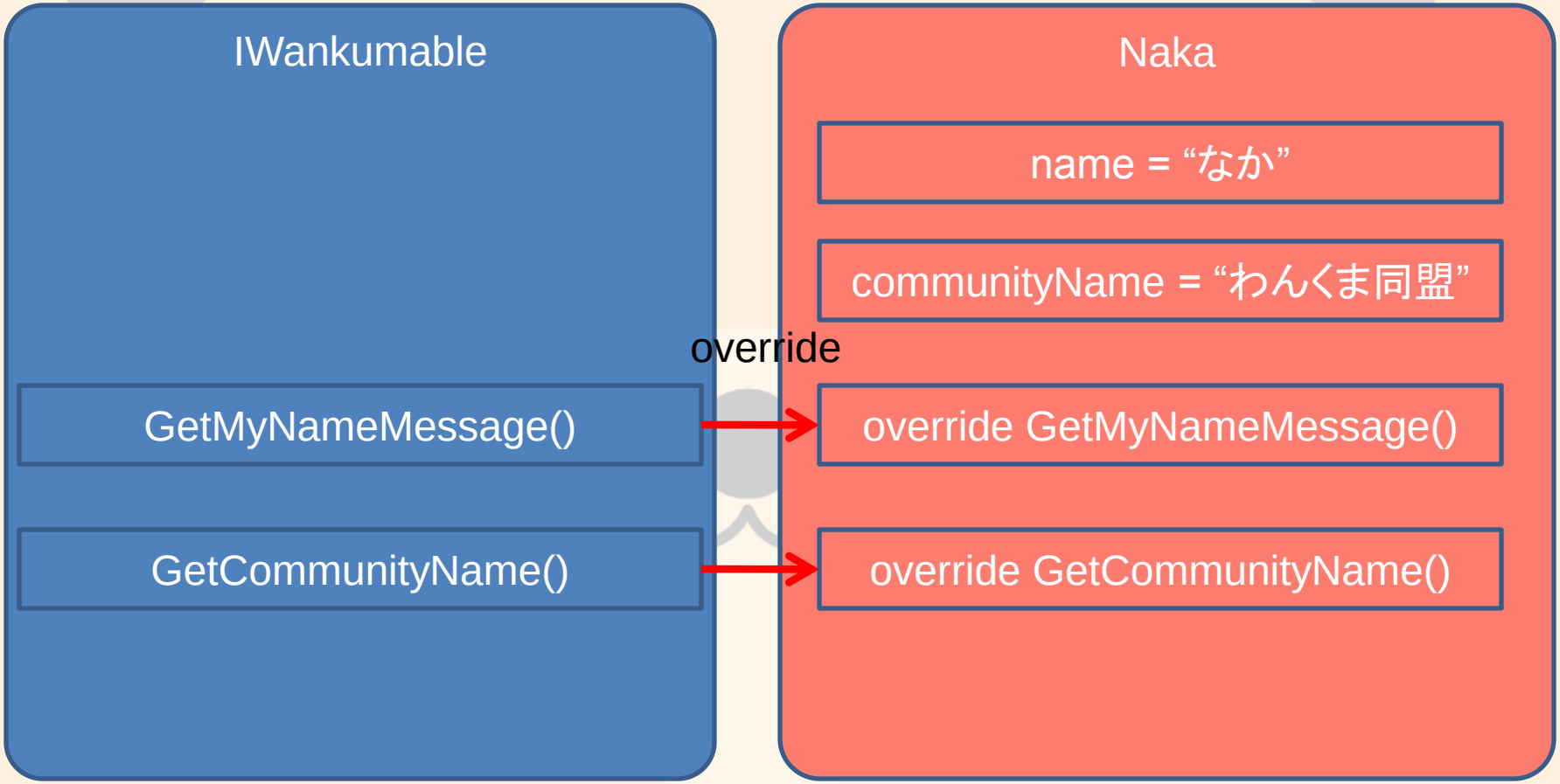

abstract Wankuma
name = “わんくま”
communityName = “わんくま同盟”
abstract GetMyNameMessage()
GetCommunityName()
Naka
name = “なか”
communityName = “わんくま同盟”
override GetMyNameMessage()
GetCommunityName()

んじゃ、必要なものを列挙しちゃえ

とりあえず、列挙しておく

Demo

インターフェース


IWankumable
GetMyNameMessage()
GetCommunityName()
override
Naka
name = “なか”
communityName = “わんくま同盟”
override GetMyNameMessage()
override GetCommunityName()

ソフトウェア開発者の業務とは

ユーザーに対してのBusiness(業務)/Problem(問題)に対して
提供すべきSolution(解決策)を提示しなければならない

Solutionの品質が悪いとそれはSolutionとならない

高い品質のSolutionつまりソフトウェアを提供しなければならない

じゃ、何故にこんな仕組みが必要なの？

品質

外的品質要因とは

- スピード、使いやすさなどユーザーがその有無を認識できるもの

内的品質要因とは

- モジュール性、読みやすさなどコンピューターの専門家しかわからないようなもの

つまり

- 外的品質要因を満たす鍵は内的品質要因にある

外的品質要因とは

正確さ	頑丈さ	拡張性	再利用性
互換性	効率性	可搬性	使いやすさ
機能性	適時性	実証性	統合性
修復性	経済性		

正確さと頑丈さ

正確さ：仕様によって定義された動作をおこなう能力

- 第一に優先されるべき事項
  これができなければ他の要件はまったく無意味
- 口に言うほど容易ではない

正確さを証明するにはシステム要求を正確に記述していなければならない
→前提条件依存

頑丈さ：異常な条件に対して適切に対応する能力

- 異常な条件に対して適切に対応する能力
  「正確さ」を補完する能力
- 正確さと違い、頑丈さは仕様以外のところで起きる→本質的にあいまい

「正確さ」と「頑丈さ」をあわせて「信頼性」という

拡張性と再利用性

拡張性：仕様変更にたいする適応しやすさ

〇ソースコードにアクセスできるからといってプログラムを変更できるかというと・・・・

〇小さいコードでは問題にならないが大きくなると問題になる

〇拡張性が必要なのは目的が実はMoving Targetだから

→伝統的なソフトウェア開発方法は変化を考慮されていない

〇拡張性を高めるには

1）設計の単純さ

単純なアーキテクチャは複雑なアーキテクチャより変更しやすい

2）非集中化

モジュールの自治性が高まると、単純な変更による影響が1つないし少ないモジュールでとどまる可能性が高い。

拡張性と再利用性（2）

再利用性：多様なアプリケーション構築に利用し易さ

〇再利用が要求されるのは同じようなパターンのコードを作成するから

〇共通性を利用すれば前に遭遇した同じ問題の解決方法を考え直さなく済む

〇再利用性が高くなると書くべきコードが減る

→同じコストで信頼性など他の要因に時間を割くことができる

拡張性と再利用性(3)

これらをふたつを併せて

モジュール性
(Modulability)

と呼ぶ。

モジュール性を向上させるには

５つの基準

５つの規則

５つの原則

5つの基準

分解しやすさ
組み合わせやすさ
わかりやすさ
連続性
保護性

分解しやすさ

単純な構造で組み合わさる
独立性を保った少数の複雑でない問題に
分割することを助ける。

組み合わせやすさ

もともと開発されたものとは異なる環境において互いを自由に組み合わせることができる新しいシステムの制作を助ける。

わかりやすさ

ほかのモジュールの知識を必要としない

（最悪でもほんのわずかのモジュールを知るだけでいい）

読み手が理解できる

ソフトウェアの生成を助ける。

連続性

問題の使用に小さな変更が生じたとき
1つまたは少数モジュールの変更しか
引き起こされない

（解析数学における連続関数の概念の類推から命名）

保護性

モジュール内で発生した異常な条件の
影響をモジュール内に閉じこめることができる
もしくは周辺の少数モジュールにしか
影響が広がらない

5つの規則

直接的な写像

少ないインターフェース

小さいインターフェース（弱い結びつき）

明示的なインターフェース

情報隠蔽

直接的な写像

システム構築過程で構築された
モジュール構造をそれに先立つ
問題領域のモデル化の過程で考案された
任意モジュールとその互換性を
維持していなければならない

少ないインターフェース

すべてのモジュールが
出来る限り少ないモジュールとの
通信で済むようにする

小さいインターフェース（弱い結びつき）

2つのモジュールが通信する場合、
そこで更新される情報は
できるだけ少なくするべきである

明示的なインターフェース

AとBの2つのモジュールが通信するときは
必ず、そのことがAまたはBあるいは両方の
テキストから明らかにわからなければならない

情報隠蔽

どのようなモジュールであれ、
モジュールを設計する人は
そのモジュールの属性の中から
いくつかの属性をそのモジュールに関する
正式な情報として選択し、顧客モジュールの
作成者がその情報を利用できるように
しなければならない。

5つの原則

言語としてのモジュール単位

自己文書化の原則

統一形式アクセスの原則

開放／閉鎖の原則

単一責任選択の原則

言語としてのモジュール単位の原則

モジュールは使用される言語の構文単位に
対応していなければならない

自己文書化の原則

モジュールを設計する人は
モジュールについてすべての情報を
そのモジュールの一部として作るよう
あらゆる努力をすべきである

統一形式アクセスの原則

あるモジュールによって提供されるサービスは
すべて統一された表記によって
利用できなければならない。
その標記はサービスが記憶領域によって
実装されるか計算によって
実装されるかにかかわらず
一定でなければならない。

開放／閉鎖の原則

モジュールは開いていると
同時に閉じているべきである

拡張が受け入れられるとき→開放されている
モジュールが他のモジュールから使用できるとき→閉じている

単一責任選択の原則

ソフトウェアシステムが選択肢を
提供しなければならないとき、
そのシステムの中の1つのモジュールだけが
その選択肢をすべて把握すべきである。

それじゃ非OOからOOへ

こんな風に使ってみるのはいかが？

Demo

例：簡単なテキストエディタ

2つのモード切替：参照のみモード・編集モード

なんか雰囲気わかりました？

処理がまとまっていく様子

- インライン化
- メソッド化
- クラス化

追加・変更がしやすくなっていく様子

- 抽象クラスの作り方
- インターフェースの作り方
- 具象クラスの作り方

オブジェクト指向言語であるメリット

- このように、安全に変更しやすくなる
- 具象クラスへ必要項目をきちんと教えることができる。

お疲れ様でした。

ご静聴ありがとうございました。