ソフトウェア工学実習
Software Engineering Practice
(第06回)

SEP06-001 MVC（Observer / Observable）

慶應義塾大学・理工学部・管理工学科
飯島 正
iijima@ae.keio.ac.jp

MVC (Model-View-Controller) の分離

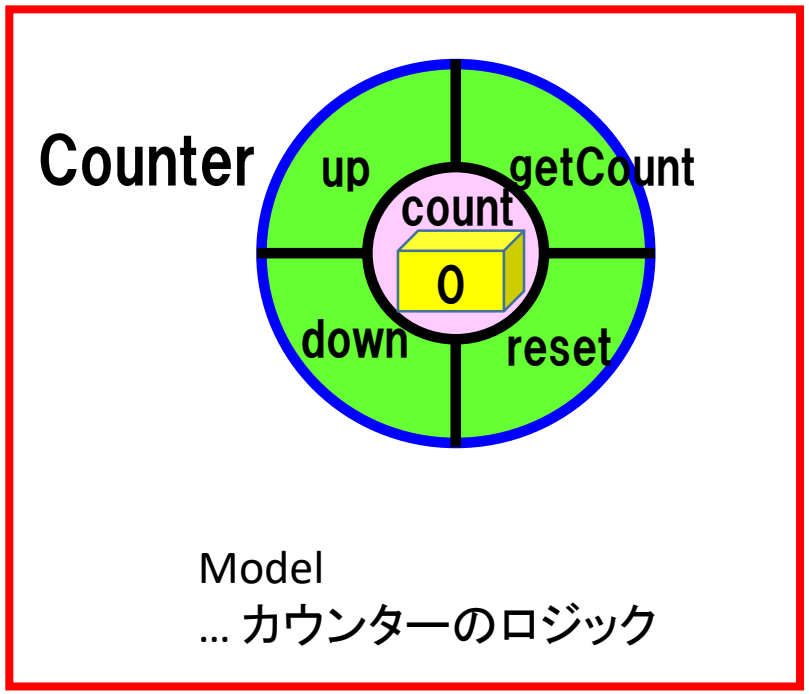
今日の話題: モデル-ビュー-コントローラ (Counterの分離)

- Model … カウンターのロジック
- View … カウンターの見た目 (表示)
- Controller …カウンターの操作

本質的なロジック

ユーザインタフェース

まず，
MVCのそれぞれで
パッケージを
分離します．



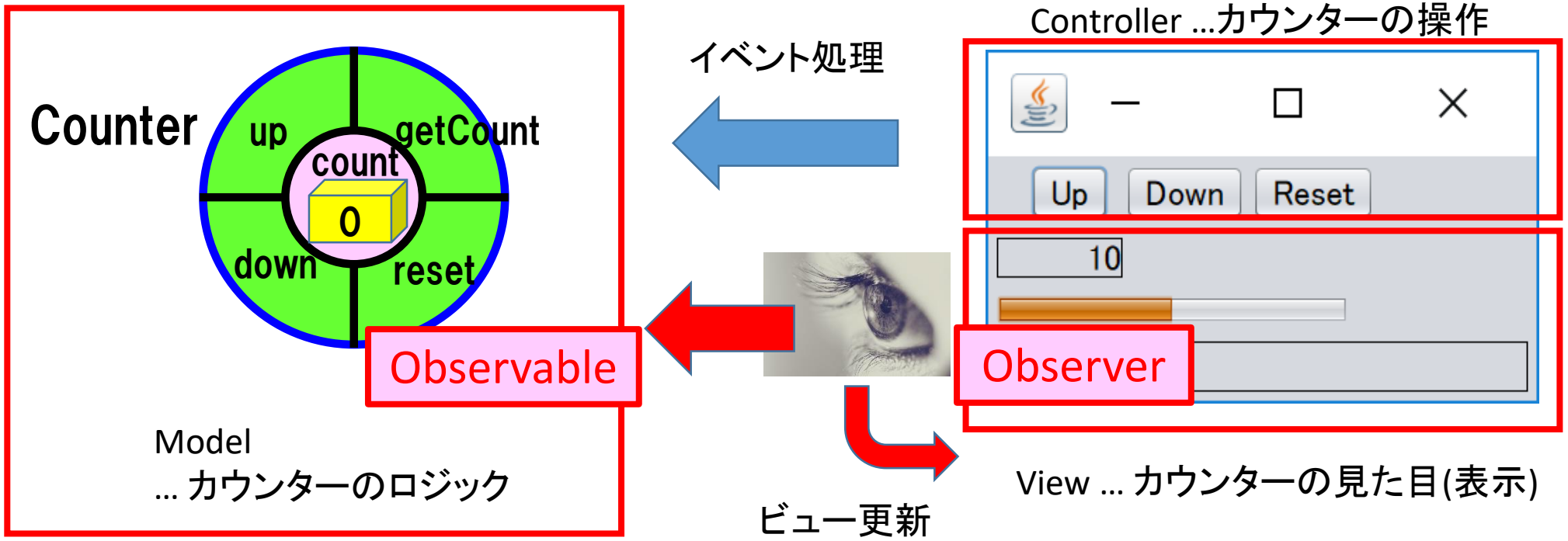
今日の話題: Observer/Observableパターンの導入

- Model … カウンターのロジック
- View … カウンターの見た目(表示)
- Controller …カウンターの操作

本質的なロジック

ユーザインタフェース

まず，
MVCのそれぞれで
パッケージを
分離します．

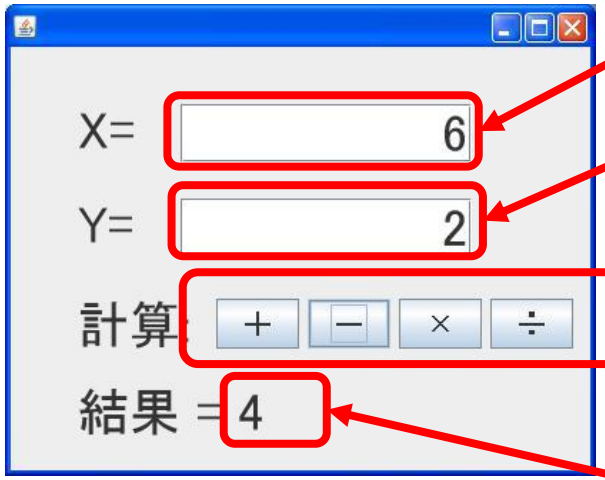


別の例題: 簡単な電卓（5月に入ったら，やりましょう）

- 簡単な電卓を作ってみます（5月に入ってから自力で，作っていただき，拡張していきます）
  - テキストフィールド，ボタン，ラベルを使います．
- まずは，簡単に　GUIの中に計算機能が埋め込んで作ってしまいましょう．
- 問題点→　計算が複雑になったら，ゴチャゴチャになってしまう．
- 解決→　MVC (Model-View-Controller) の考え方を導入します．

別の例題: 簡単な電卓（GUIビルダですぐ作れますが…）

- NetBeansのGUIビルダで簡単に作れます.
  - 素早く試作品（プロトタイプ）を作ってしまいましょう
  - ラピッドプロトライピングといいます.



テキストフィールド(JTextField): xTextField

テキストフィールド(JTextField): yTextField

ボタン(JButton):
addButton, subButton,
mulButton, divButton

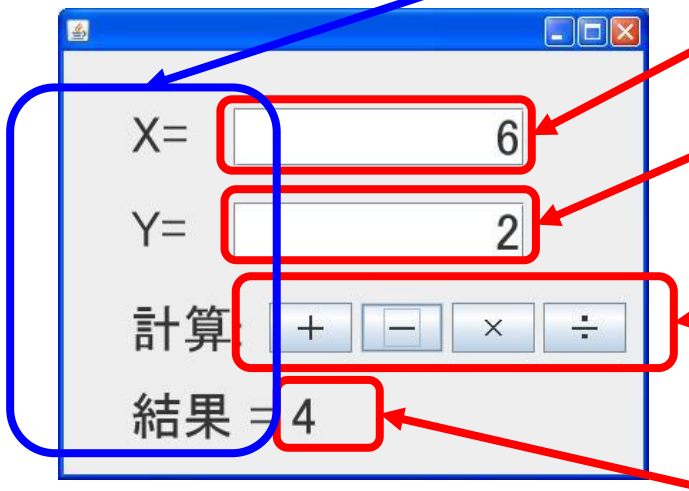
ラベル(JLabel):
resultLabel

ラピッド
プロトタイピング

別の例題: 簡単な電卓（GUIビルダですぐ作れますが…）

- NetBeansのGUIビルダで簡単に作れます.
  - 素早く試作品（プロトタイプ）を作ってしまいましょう
  - ラピッドプロトライピングといいます.

こちら側の見出しラベル(JLabel)の変数名は特に考えなくてもいいでしょう



テキストフィールド(JTextField): xTextField

テキストフィールド(JTextField): yTextField

ボタン(JButton):
addButton, subButton,
mulButton, divButton

ラベル(JLabel):
resultLabel

別の例題: 簡単な電卓（設計→MVCの切り分け）

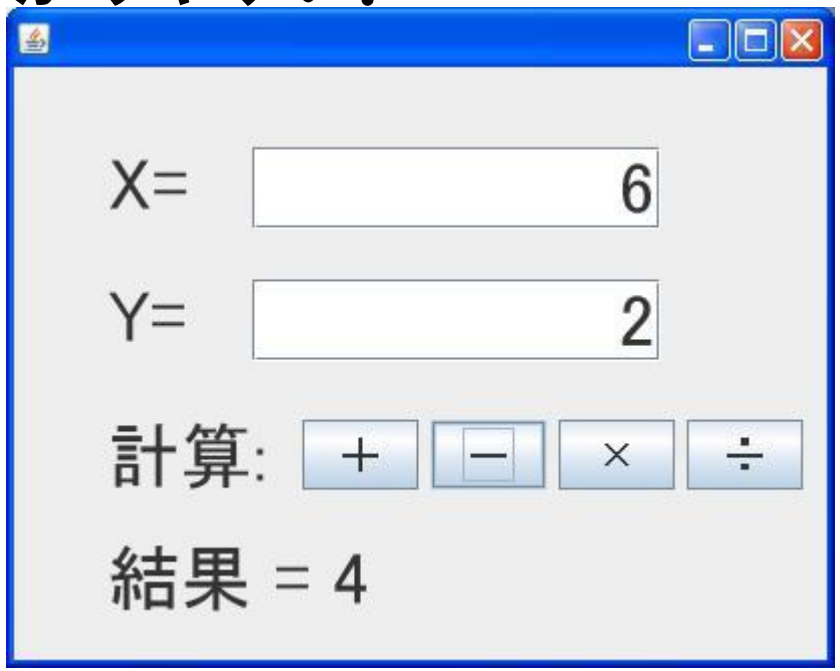
- モデルと，GUIの分離は，わかりやすい．

数値x
数値y
四則演算
例外処理
計算結果

Model

計算機能
＝モノとしての
電卓の概念

クラスSimpleCalc



View-Controller

見た目
＝結果表示

制御
＝ボタン

クラスCalcFrame

では，ちゃんと
設計して
みましょう．

MVCの分離を
意識します

モデルとGUIの
分離は分かりやす
いですね

別の例題: 簡単な電卓（設計→MVCの切り分け）

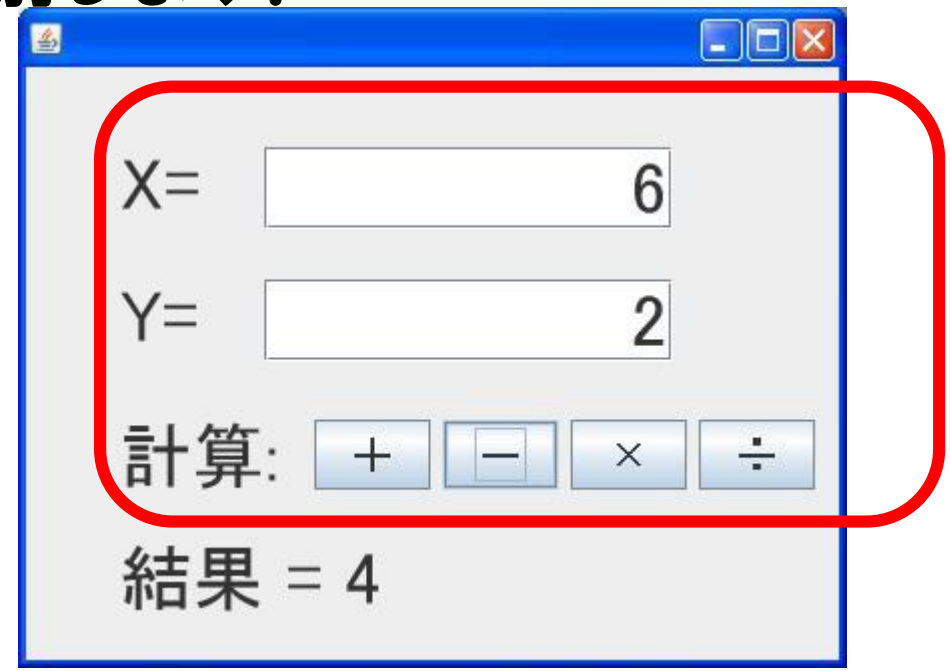
- ビューとコントローラを識別します．

数値x
数値y
四則演算
例外処理
計算結果

Model

数値x 数値y

演算の種類



View-Controller

ビューとコントローラを識別します

アプリケーションの動きを考えましょう

入力はコントローラの仕事です

計算機能
＝モノとしての電卓の概念
クラスSimpleCalc

見た目
＝結果表示

制御
＝ボタン

クラスCalcFrame

※もう少し正確には，入力だけではなく，モデルを操作する（コントロールする）ロジックがコントローラの本質なのですが…

別の例題: 簡単な電卓（設計→MVCの切り分け）

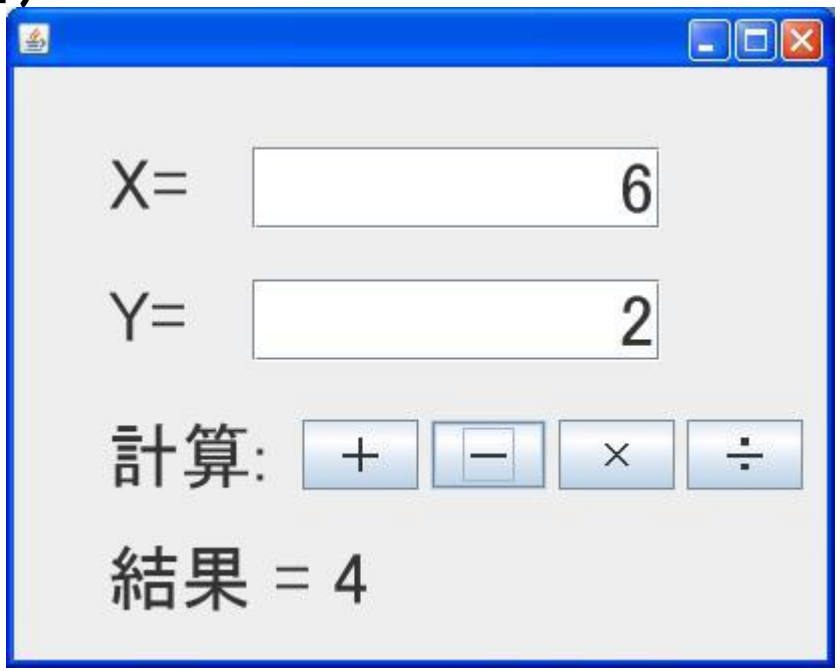
- MVC (Model-View-Controller)

数値x
数値y
四則演算
例外処理
計算結果

Model

計算機能
＝モノとしての
電卓の概念

クラスSimpleCalc



View-Controller

見た目
＝結果表示

制御
＝ボタン

クラスCalcFrame

別の例題: 簡単な電卓（設計→MVCの切り分け）

- MVC (Model-View-Controller)

数値x
数値y
四則演算
例外処理
計算結果

Model

計算結果

View-Controller

計算機能
= モノとしての
電卓の概念

クラスSimpleCalc

見た目
＝結果表示

制御
＝ボタン

クラスCalcFrame

結果表示は
ビューの
仕事です

別の例題: 簡単な電卓（設計→MVCの切り分け）

- MVC (Model-View-Controller)

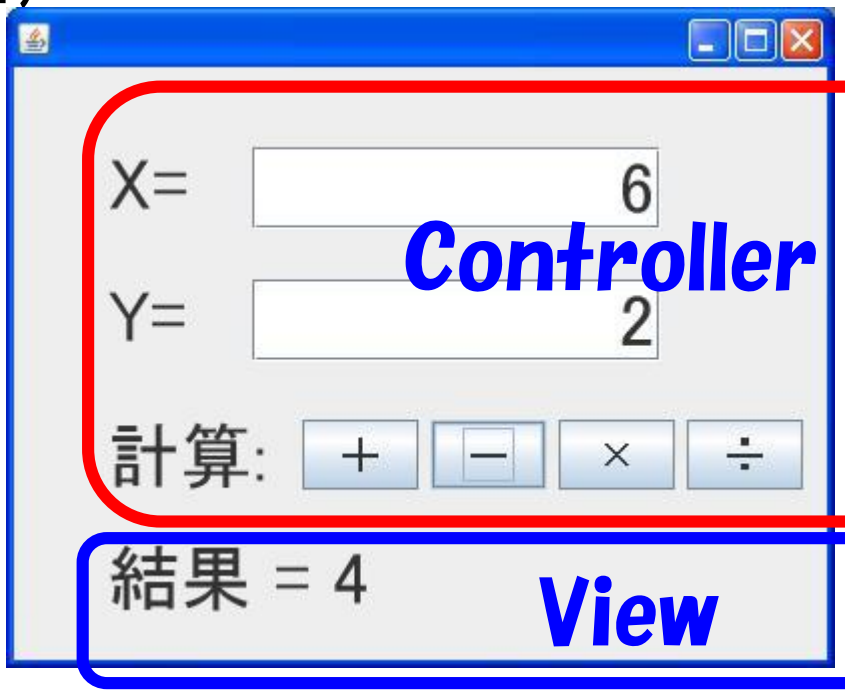
数値x
数値y
四則演算
例外処理
計算結果

Model

数値x 数値y

演算の種類

計算結果



View-Controller

※もう少し正確には，
これは，コントローラのUI
部分であって，コントローラ
のロジックではありませんが，
今のところ，ボタンの
イベントハンドラやそれに
付随するメソッドに
ロジックが定義されています．

これで，
コントローラ
と
ビューの
区別がつきま
した

計算機能
= モノとしての
電卓の概念
クラスSimpleCalc

見た目
＝結果表示

制御
＝ボタン

クラスCalcFrame

別の例題: 簡単な電卓（設計→MVCの切り分け）

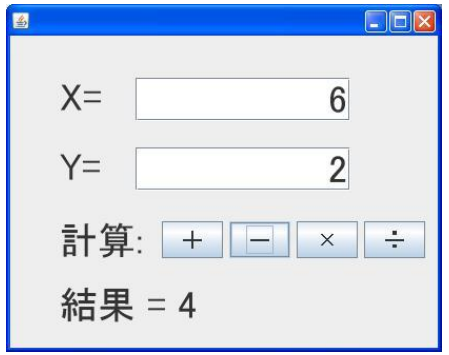
- クラス構成の概要設計

クラスSimpleCalcFrame

数値x　数値y

X= 6
Y= 2

GUI



数値x
数値y
四則演算
例外処理
計算結果

演算の種類

クラスSimpleCalcController

Controller

+ − × ÷

Model

View　クラスSimpleCalcView

計算結果

結果 = 4

計算機能
= モノとしての
電卓の概念

クラスSimpleCalc

それぞれ
クラス名を決めて，
概要設計としま
しょう

別の例題: 簡単な電卓（クラス内部の設計）

- パッケージ構成

simplecalcパッケージ

Mainクラス

SimpleCalcFrameクラス

modelパッケージ

SimpleCalcクラス

数値x
数値y
四則演算
例外処理
計算結果

viewパッケージ

SimpleCalcViewクラス

結果 = 4

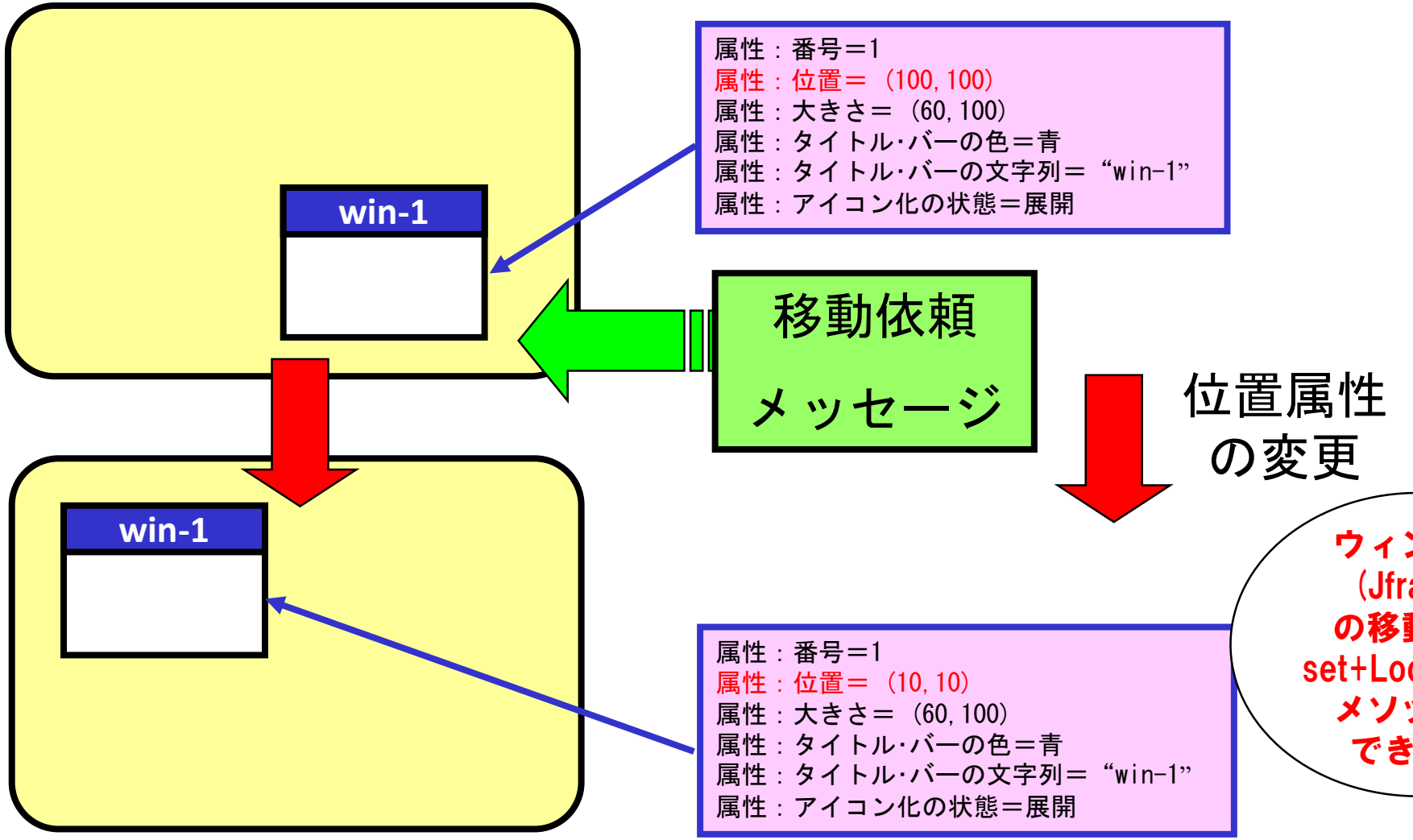
controllerパッケージ

SimpleCalcControllerクラス

パッケージに
配分します

MVCアーキテクチャ

例:ウィンドウへのメッセージ



iijima@ae.keio.ac.jp

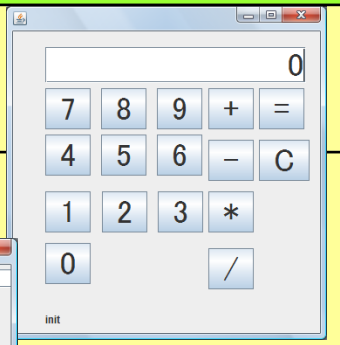
MVCアーキテクチャ

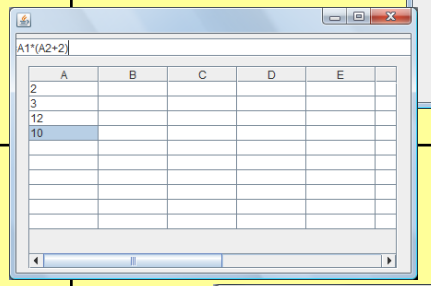
- モデル（M; Model）
  - モデル化対象の振る舞いを記述する
  - しかし，位置属性の属性値を変更しても，
    それだけでは画面表示は変わらない．
- ビュー（V; View）
  - モデルの内容を画面表示に反映させるコード
  - モデルとビューを分離することが，
    わかりやすくて永く使えるソフトウェアを作ることに重要
  - 両者の連携は依存関係である．
  - 画面全体を書き直すのは無駄なこともあり，
    いかに必要最小限の更新に留めるかが改良のポイント
- コントローラ（C: Controler）
  - モデルを外部から操作する要素（GUI）
  - パラメータを設定したり，外部から入力したりする

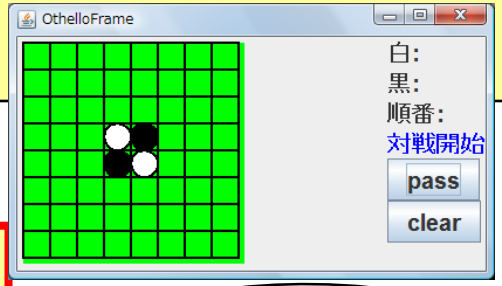
iijima@ae.keio.ac.jp

MVCの適用

	モデル	ビュー	コントローラ
電卓	内部動作 メモリ	表示パネル	ボタン
表計算	内部動作 数式の表	計算結果の 表示された表， グラフ	GUI
オセロ ゲーム のボード	内部状態 ルール	画面表示 （盤面/得点）	







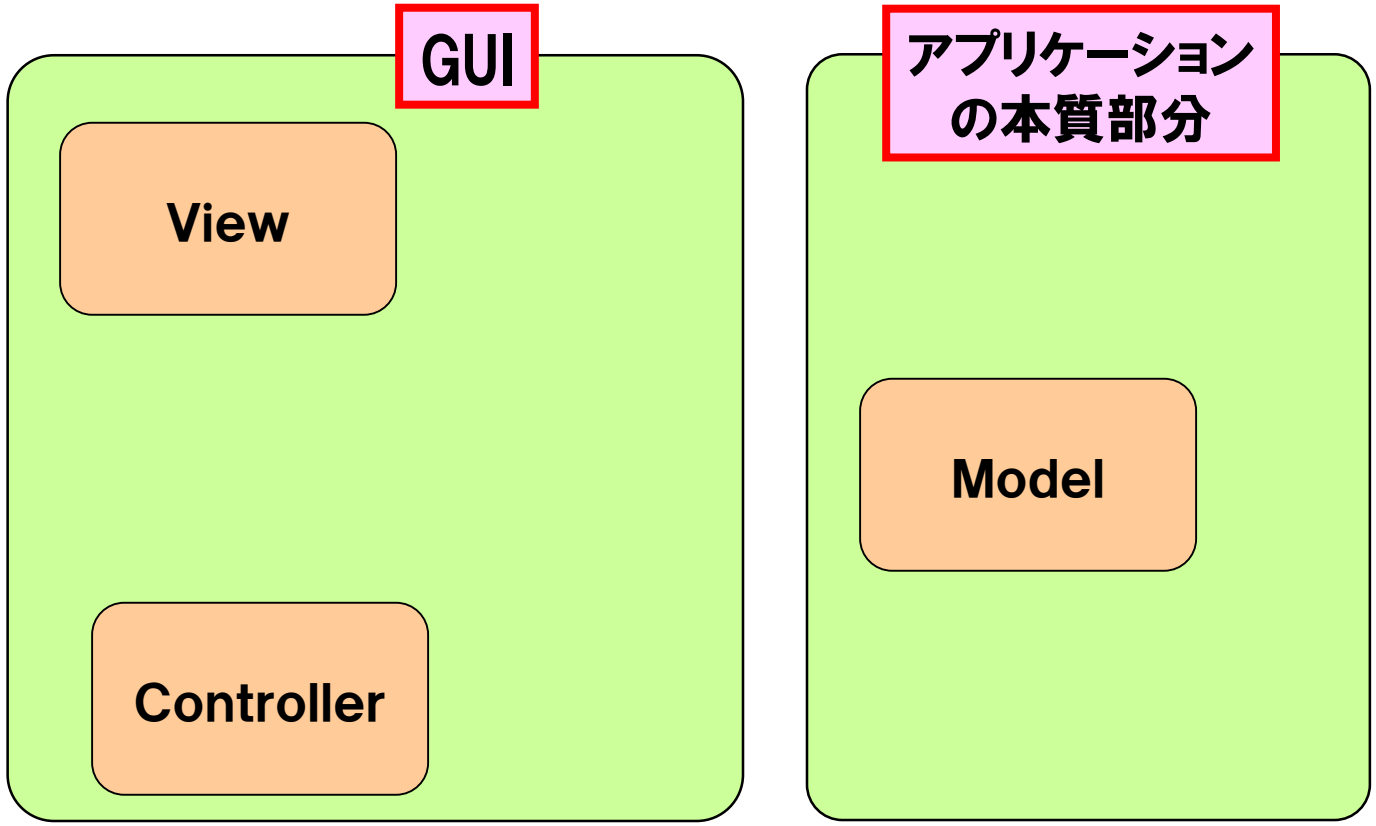
MVCアーキテクチャの考え方は，最近では，WWWをプレゼンテーション層に位置付けたエンタープライズ・システムのアーキテクチャ（Web三層モデルと呼ばれる）に転用されていて，むしろそちらが主流として使われているが，オリジナル（Smalltalk-80）はGUIアプリケーションのためのフレームワークであり，ここでは，それを取り扱う．

具体的な
アプリケーション
で
再確認しましょう

iijima@ae.keio.ac.jp

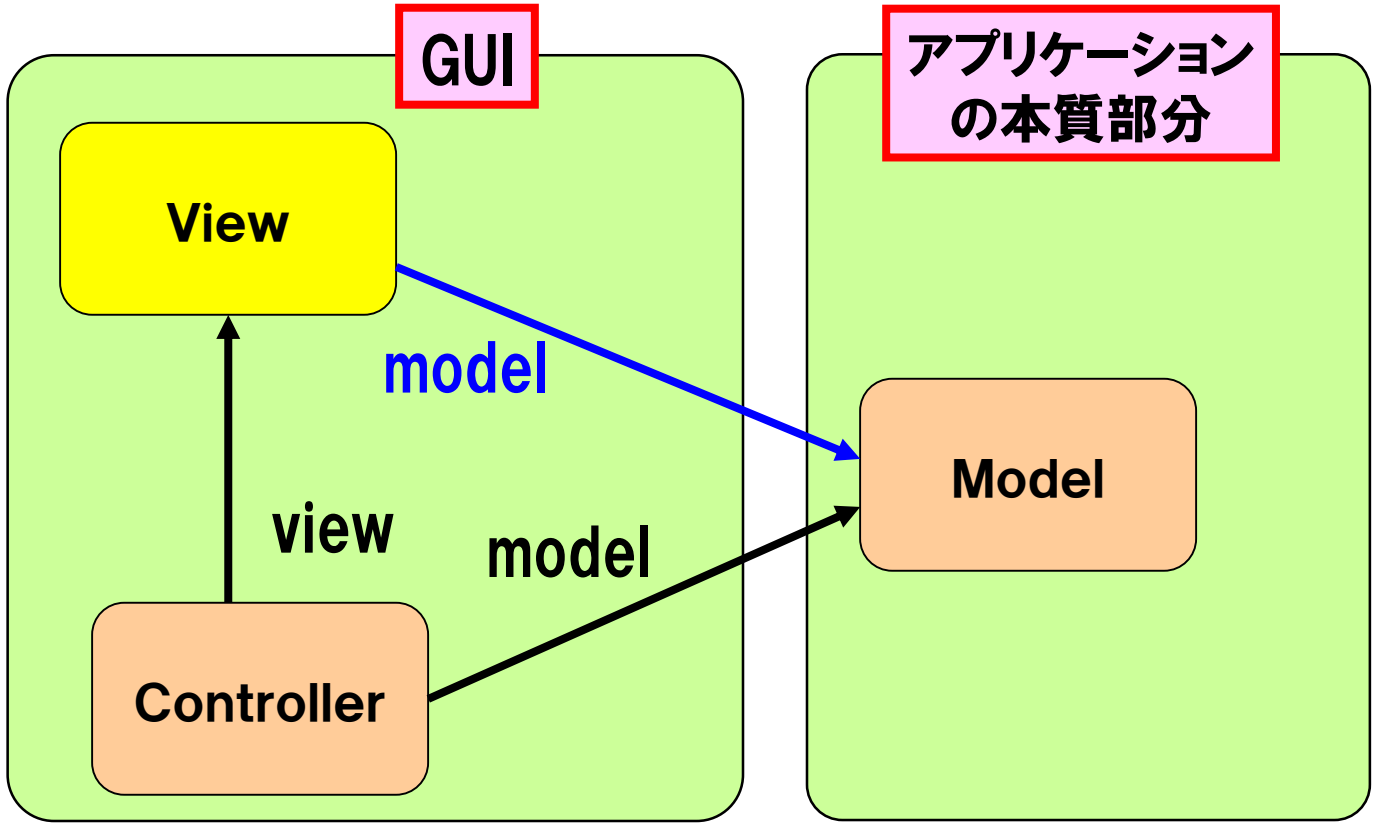
MVCの構造(1/3)

アプリケーションの本質部分	→	モデル
GUI	→	ビューとコントローラ



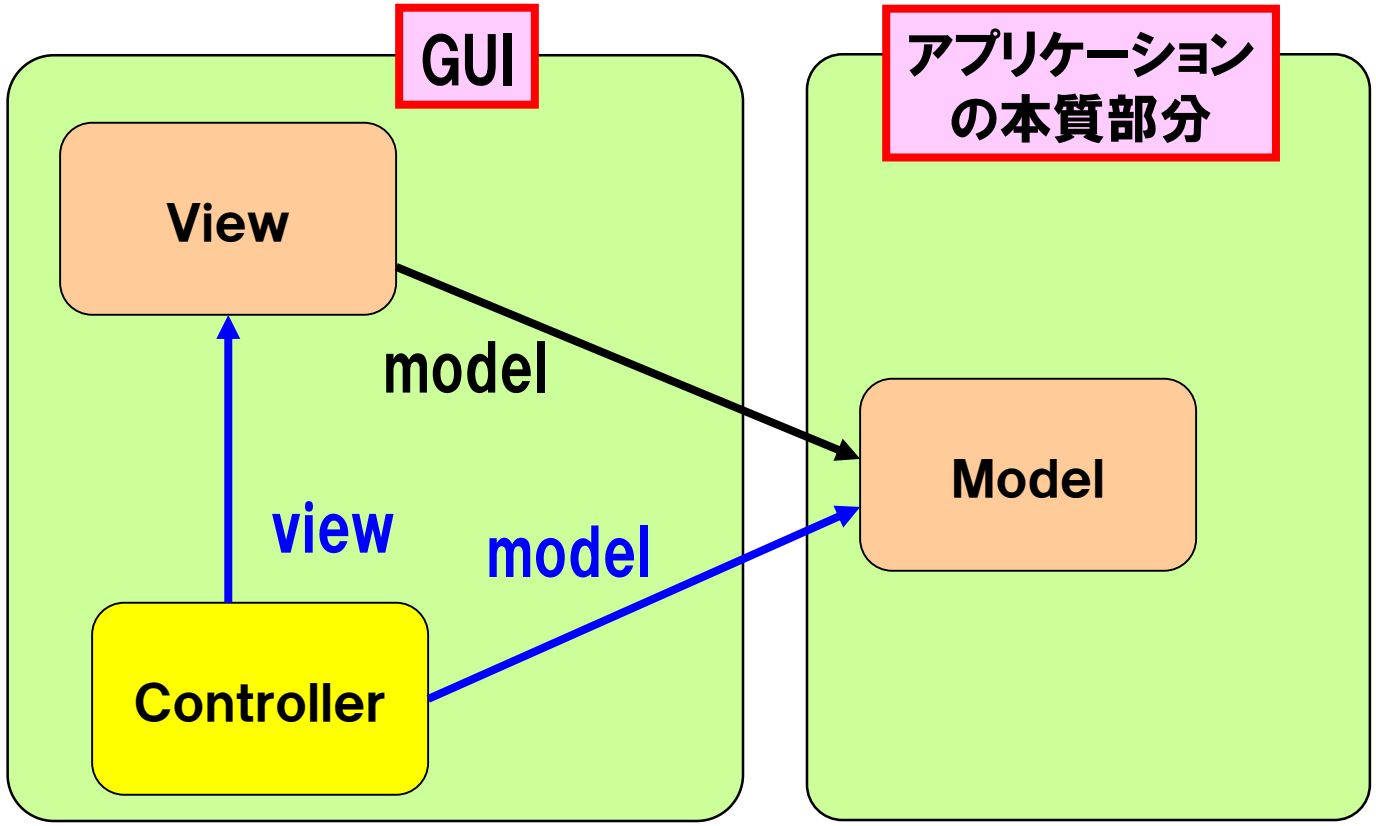
MVCの構造(2/3)

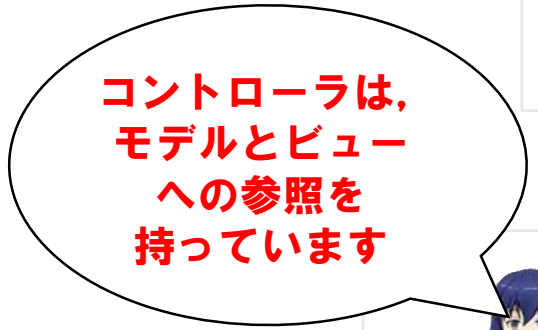
- モデル…モデルはGUI(ビューとコントローラ)への参照を持たない(GUIから独立している)
- ビュー…ビューはモデルへの参照を属性modelとして持っている
- コントローラ…コントローラはモデルとビューのそれぞれへの参照を持っている



MVCの構造(3/3)

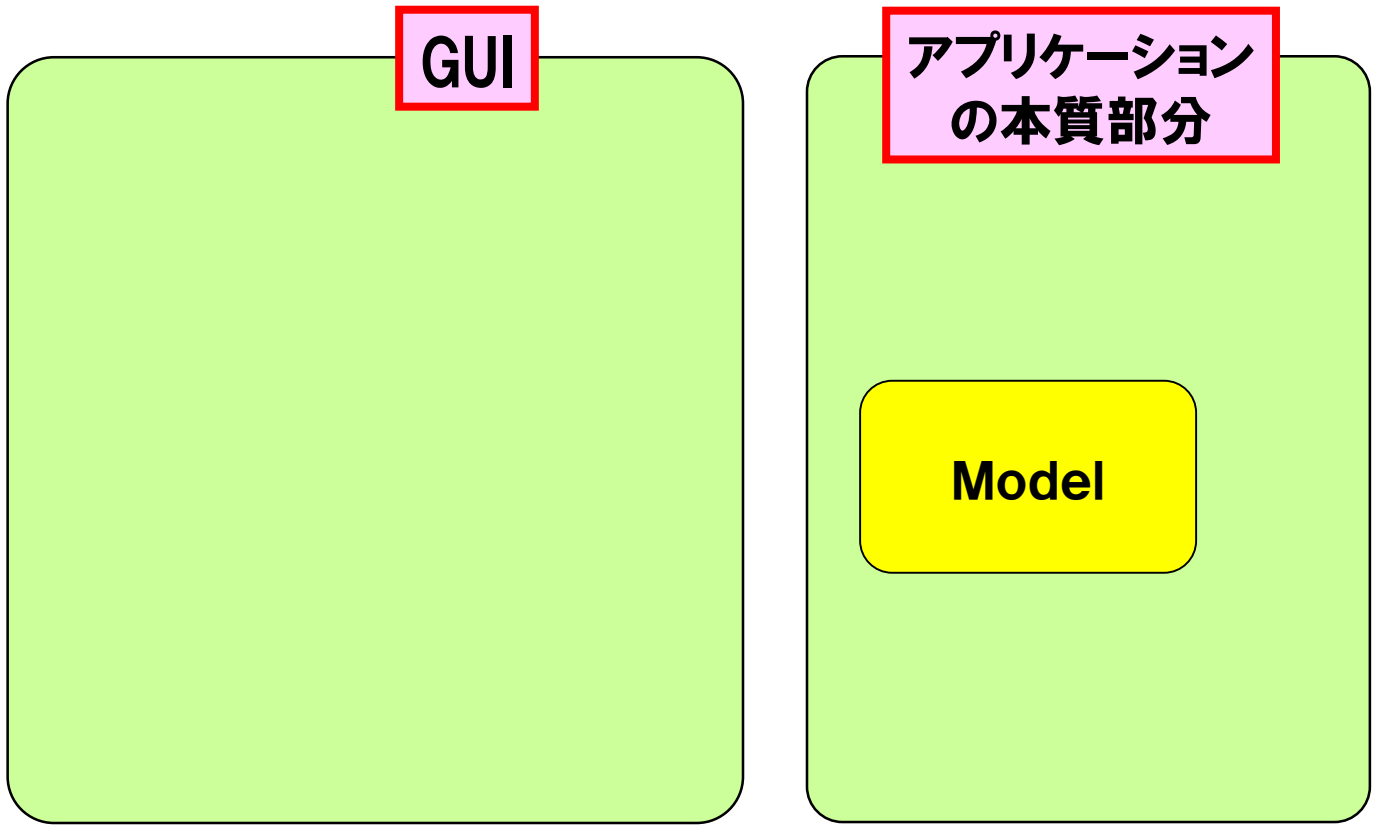
- モデル…モデルはGUI(ビューとコントローラ)への参照を持たない(GUIから独立している)
- ビュー…ビューはモデルへの参照を属性modelとして持っている
- コントローラ…コントローラはモデルとビューのそれぞれへの参照を持っている





MVCの各オブジェクトの構築の順序 (1/3)

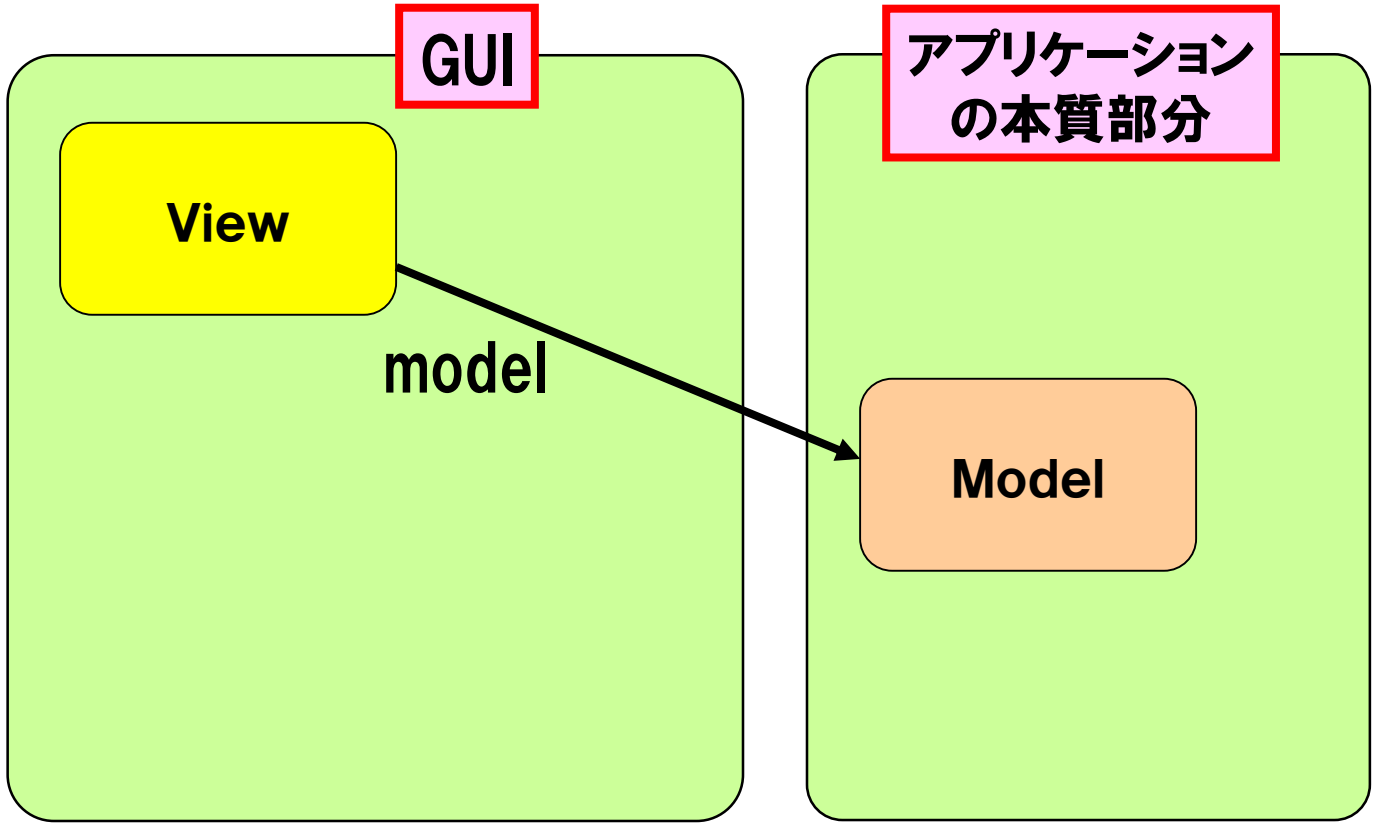
- MVCの各オブジェクトの構築の順序
  - 決してこのジュ所でないといけないというわけではないが，
  - まずモデルを作り，次にビューを作り，最後にコントローラを作るのが，よさそうです．



iijima@ae.keio.ac.jp

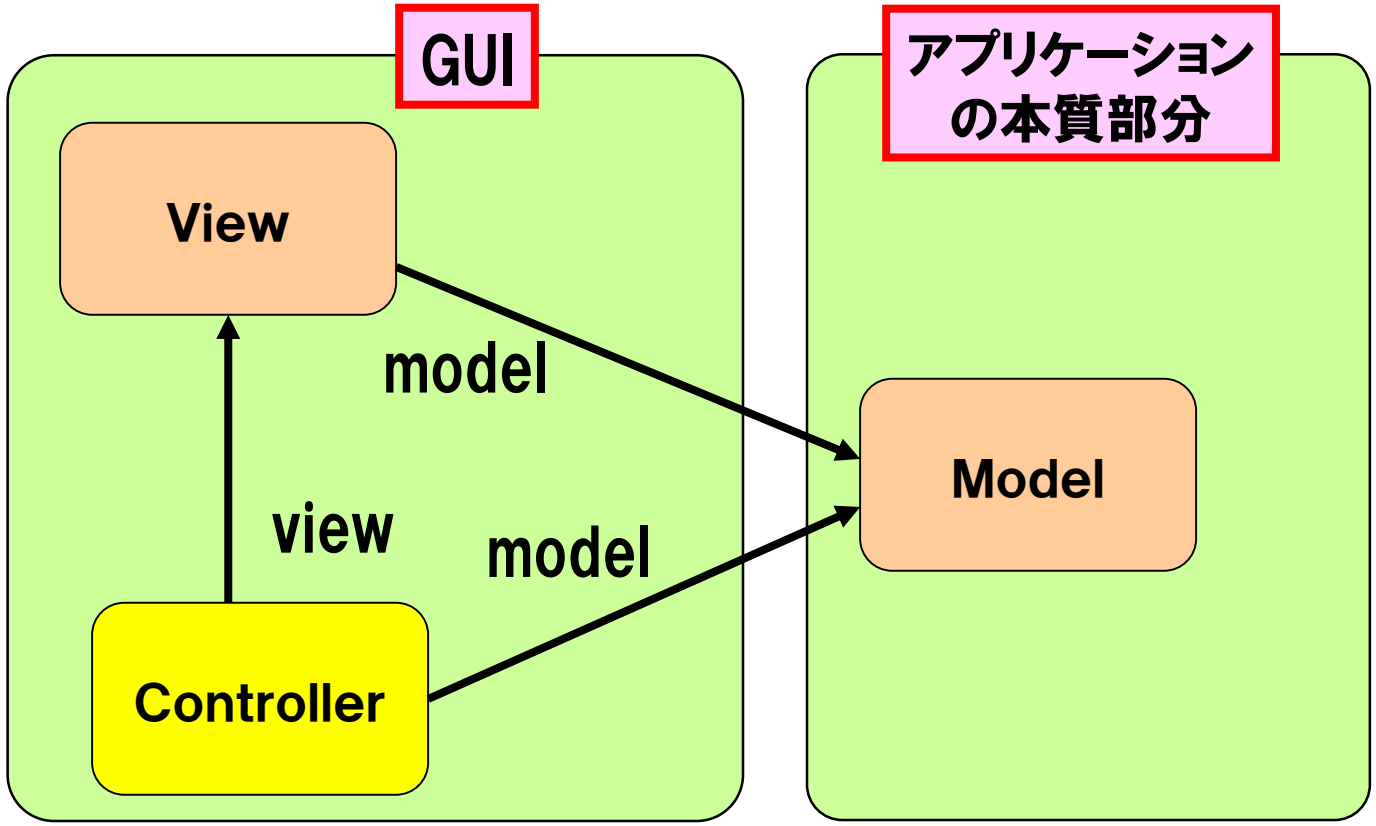
MVCの各オブジェクトの構築の順序 (2/3)

- MVCの各オブジェクトの構築の順序
  - 決してこのジュ所でないといけないというわけではないが，
  - まずモデルを作り，次にビューを作り，最後にコントローラを作るのが，よさそうです．



MVCの各オブジェクトの構築の順序 (3/3)

- MVCの各オブジェクトの構築の順序
  - 決してこのジュ所でないといけないというわけではないが，
  - まずモデルを作り，次にビューを作り，**最後にコントローラを作る**のが，よさそうです．



Observer/Observableパターン による MVCの実装

Javaでは，MVCを実装するのにObserver / Observableパターンを使うのが一般的です

Smalltalkの依存性メカニズム

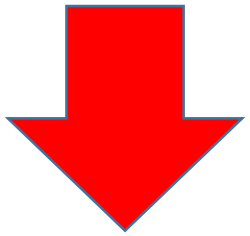
- 主オブジェクトとこれに依存するオブジェクト
- 主オブジェクトに変化
  - changed:　メッセージ
  - このメッセージを送るのも受けるのも主オブジェクト自身のことが多い
- 依存しているオブジェクトの更新
  - update:　メッセージ
- MVCでは
  - Modelが主オブジェクト、Viewが依存

ソフトウェア・アーキテクチャとしてのMVC

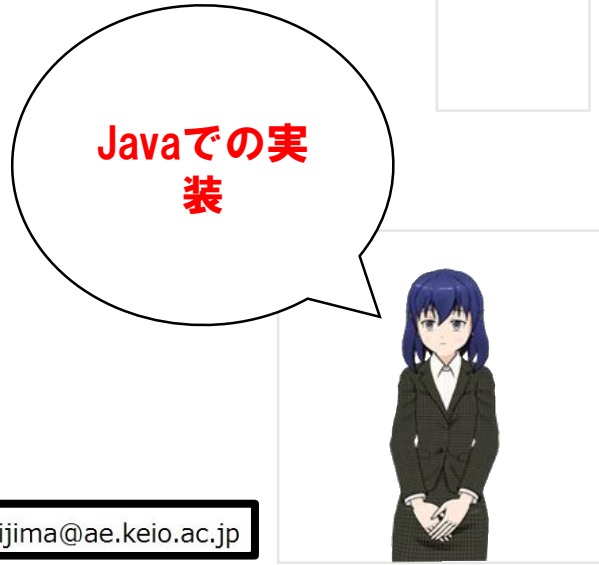
- SmalltalkのMVC（Model, View, Controller）モデル
  - GUIを含む対話型アプリケーション・フレームワーク（依存性メカニズム）
    - モデル:　　ユーザインタフェース以外のアプリケーションのロジック
    - ビュー:　　モデルからの情報を表示
    - コントローラ:　ユーザからの入力をモデルとビューに

ソフトウェア・アーキテクチャとしてのMVC

- SmalltalkのMVC（Model, View, Controller）モデル
  - GUIを含む対話型アプリケーション・フレームワーク（依存性メカニズム）
    - モデル:　　ユーザインタフェース以外のアプリケーションのロジック
    - ビュー:　　モデルからの情報を表示
    - コントローラ:　ユーザからの入力をモデルとビューに

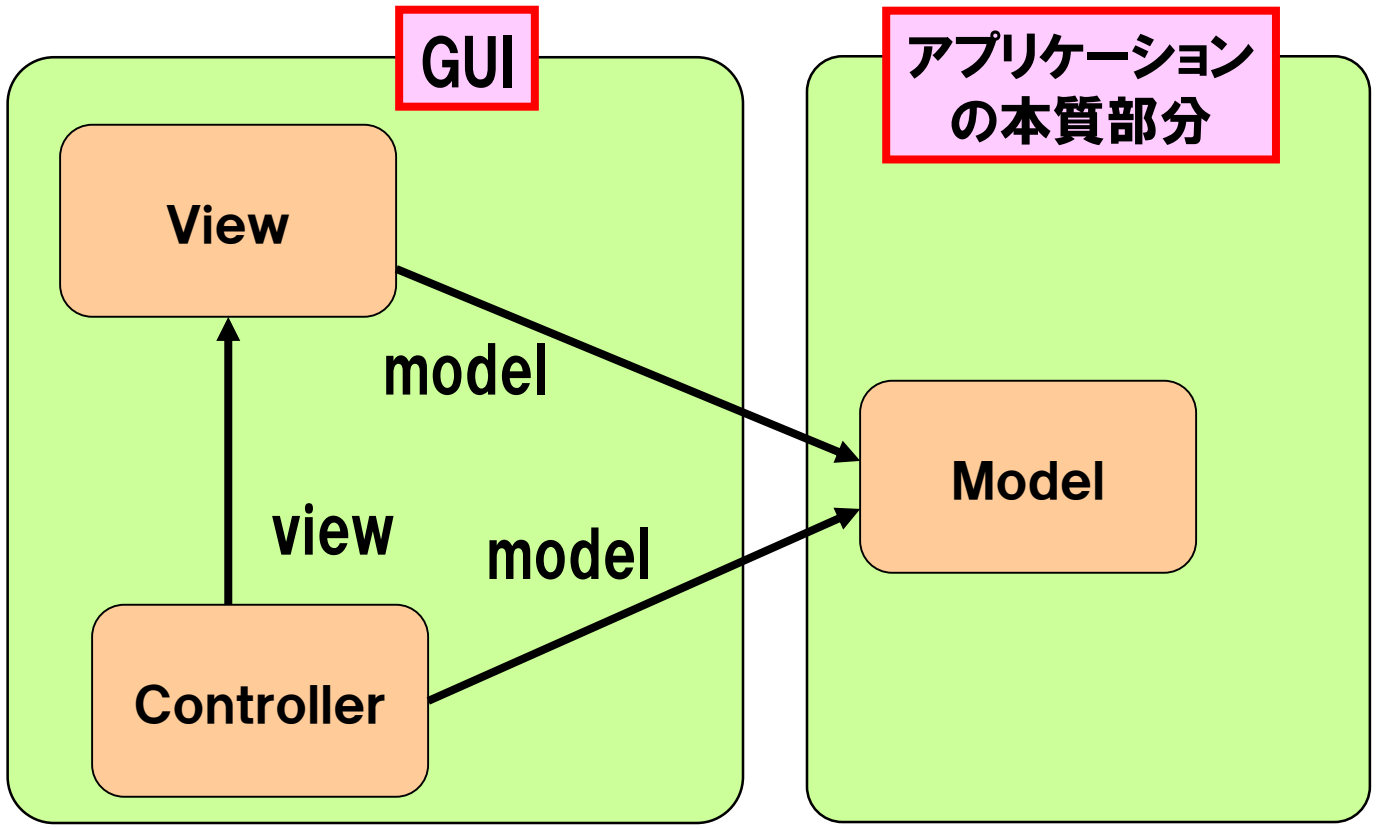


- JavaのMVC（Model, View, Controller）モデル
  - Smalltalkに組み込まれていたMVCモデルを
    Javaで実装するには,
    Observer/Observableパターンを使うのが一般的です



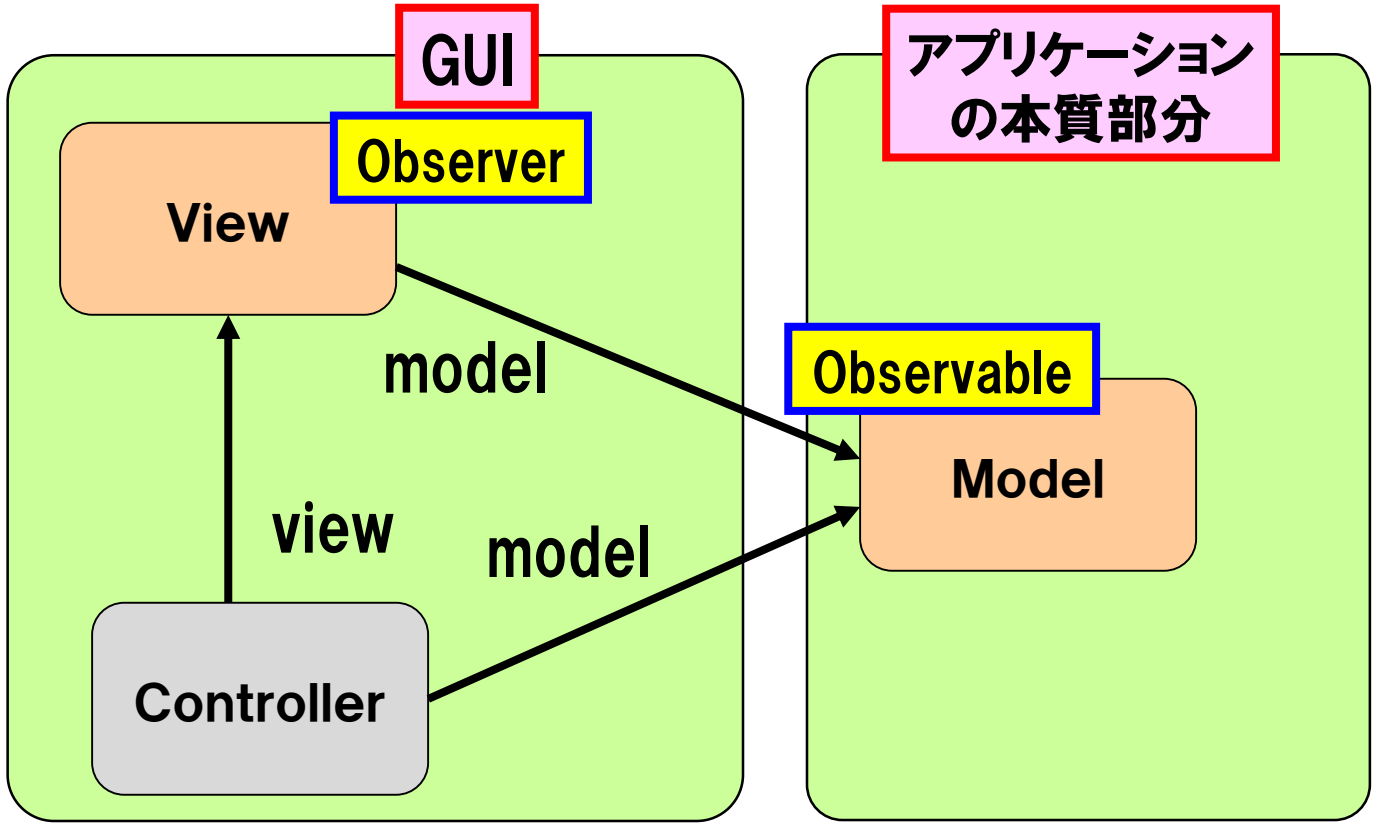
Observer/Observableパターンの動き(0/5)

- Observer/Observableパターンの動き:



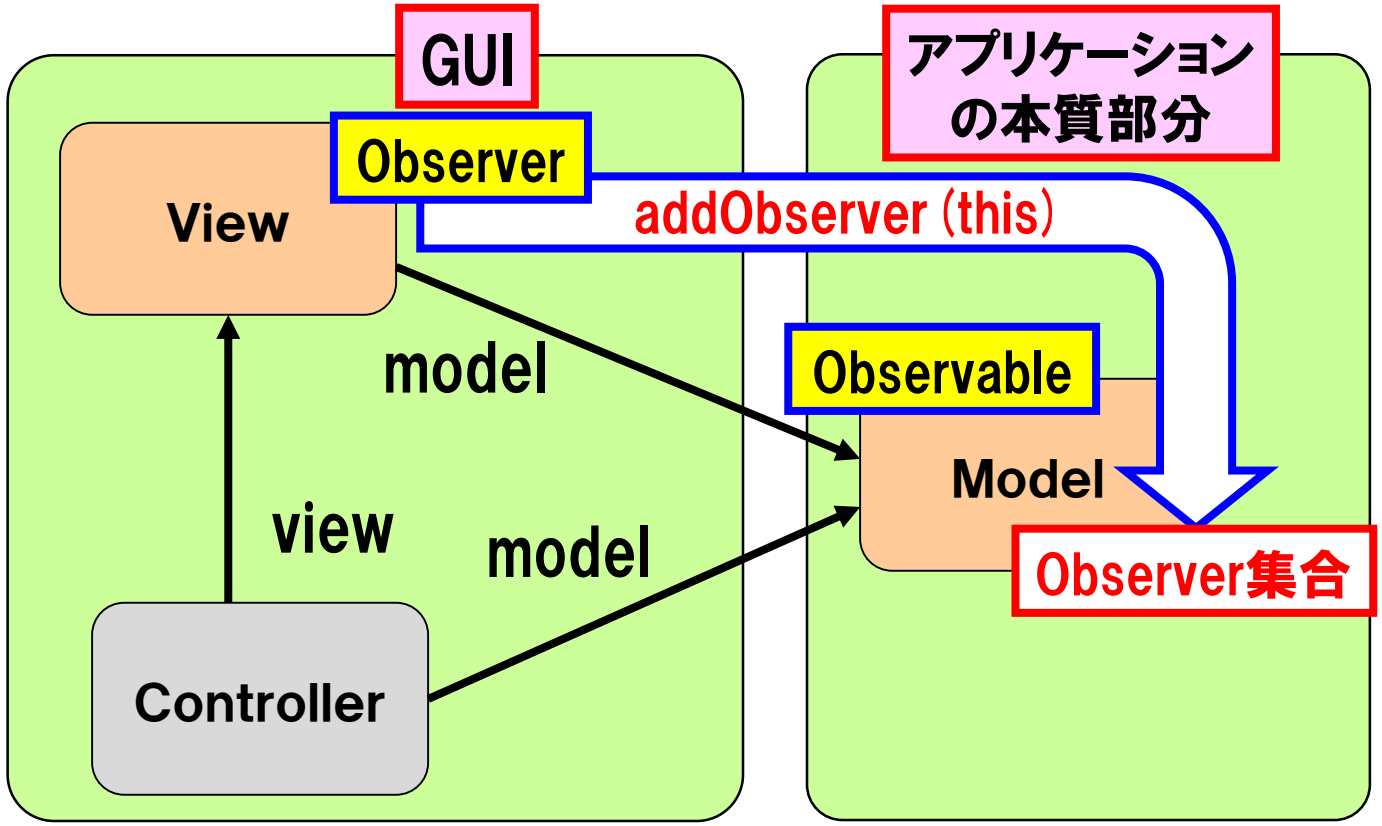
Observer/Observableパターンの動き(0/5)

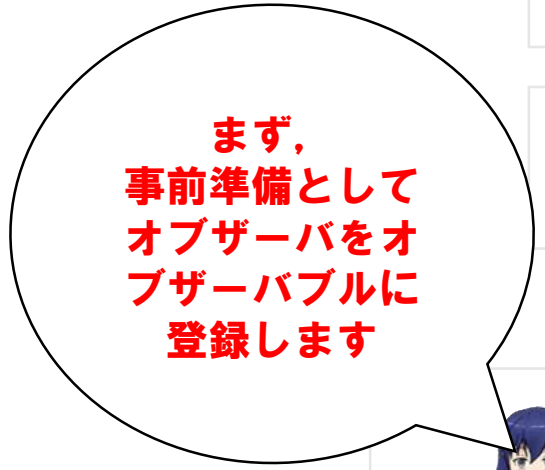
- Observer/Observableパターンの動き:



Observer/Observableパターンの動き（1/5）

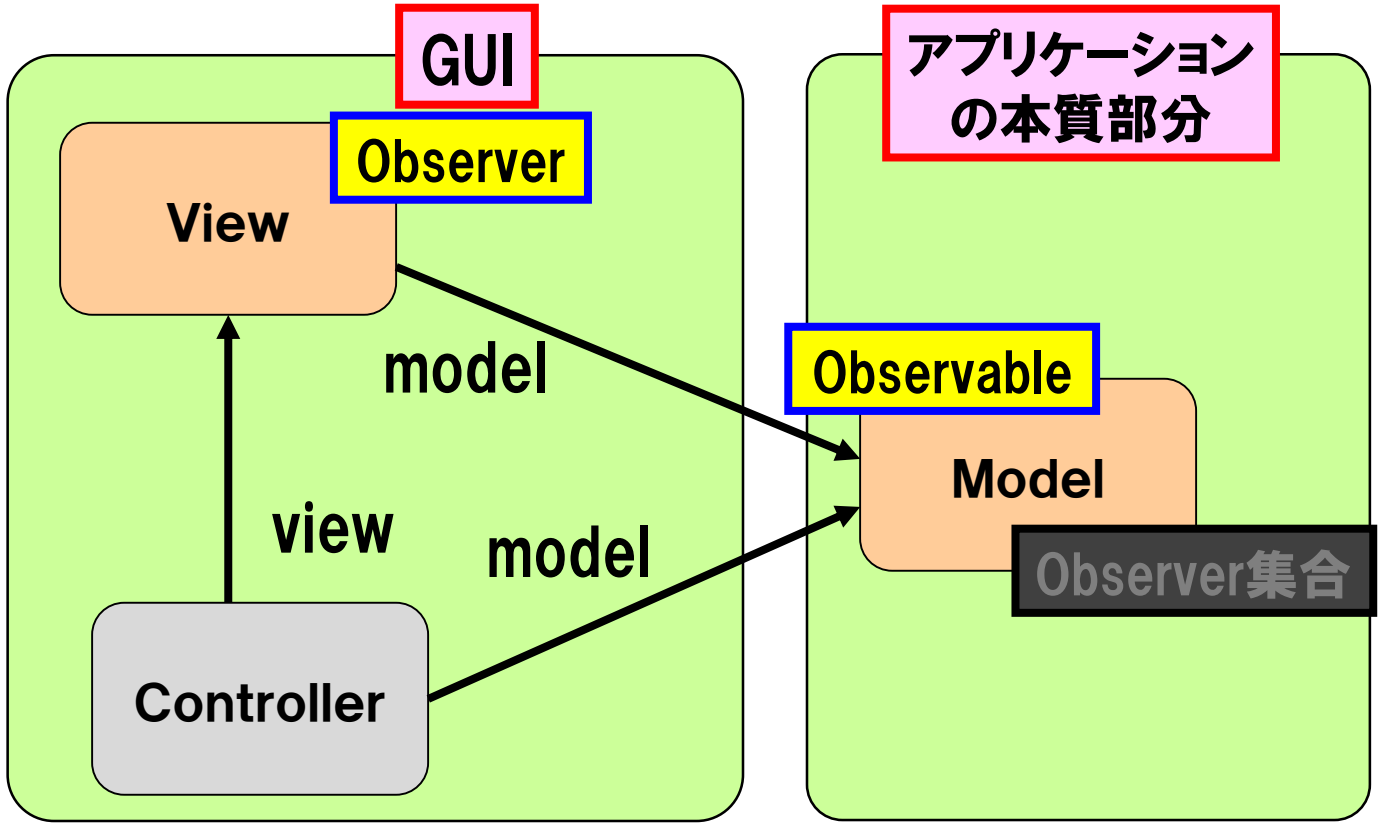
- Observer/Observableパターンの動き:　【事前準備】
  - ビュー (Observer) は，事前にモデル (Observable) に登録される
  - しかし，モデルの側は，その登録データを意識的参照しない点が重要





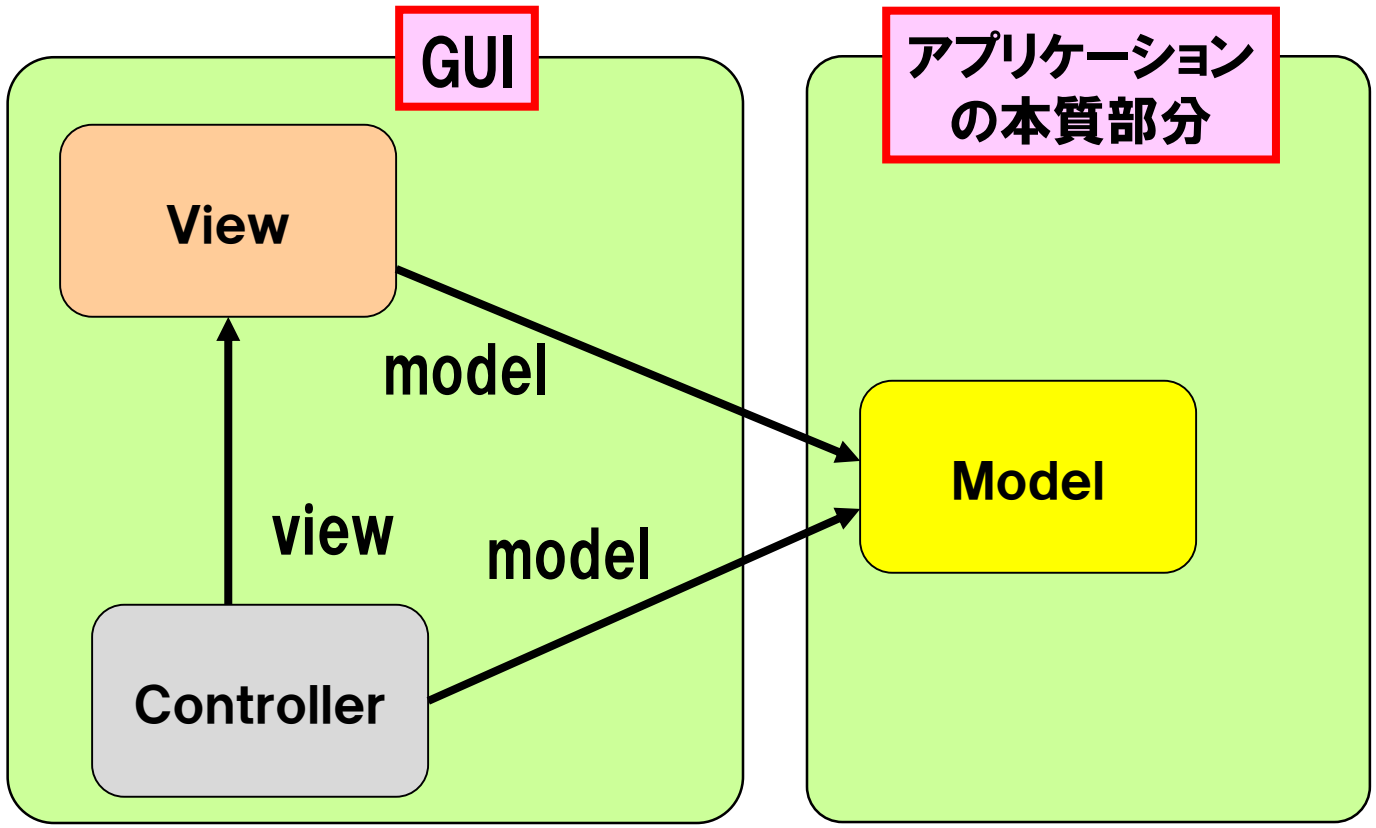
Observer/Observableパターンの動き（1/5）

- Observer/Observableパターンの動き:　【事前準備】
  - ビュー (Observer) は，事前にモデル (Observable) に登録される
  - しかし，**モデルの側は，その登録データを意識的に参照しない**点が重要



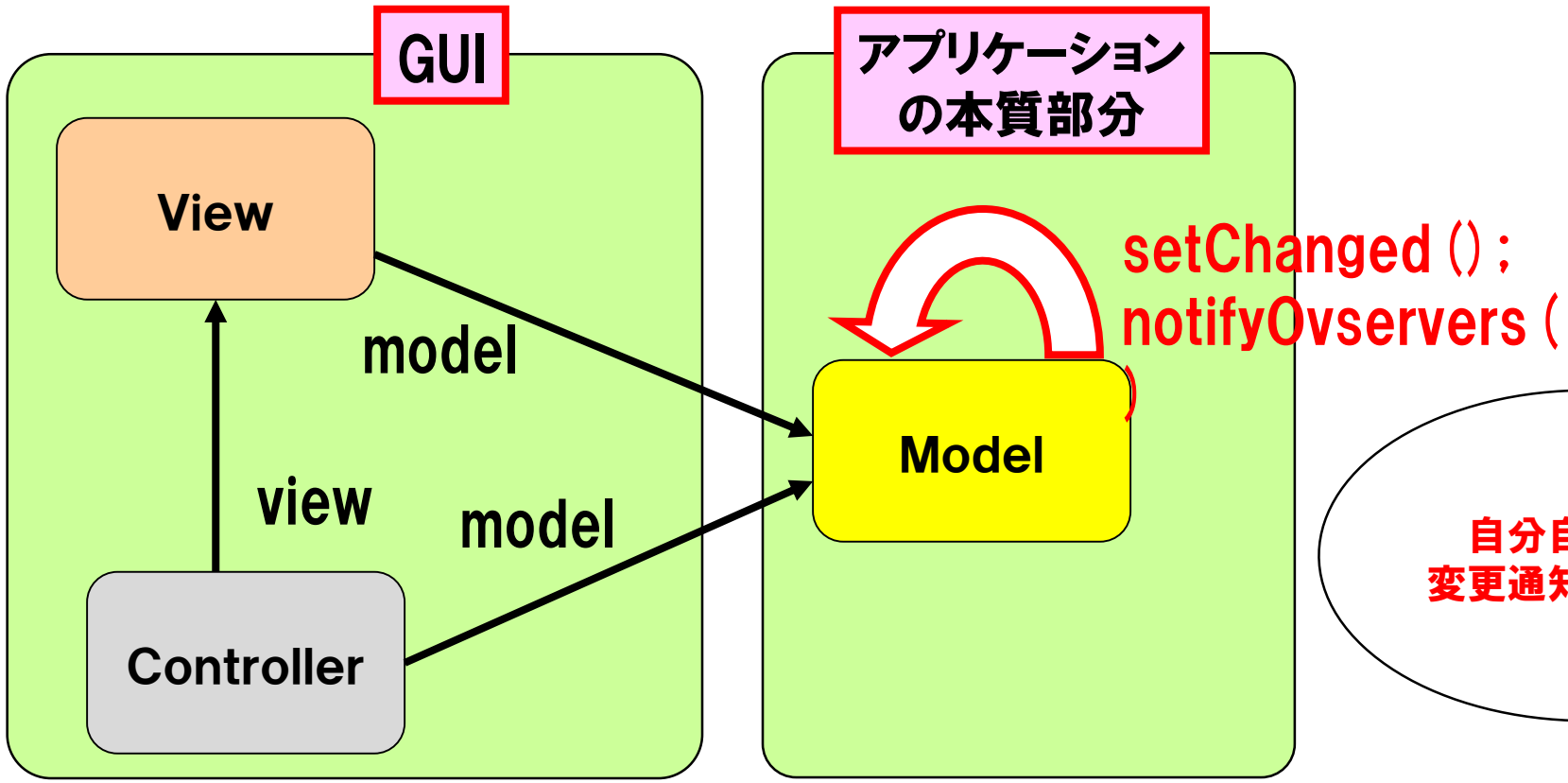
Observer/Observableパターンの動き (2/5)

- Observer/Observableパターンの動き:
  【モデルが自分自身の内部状態を変更すると...】



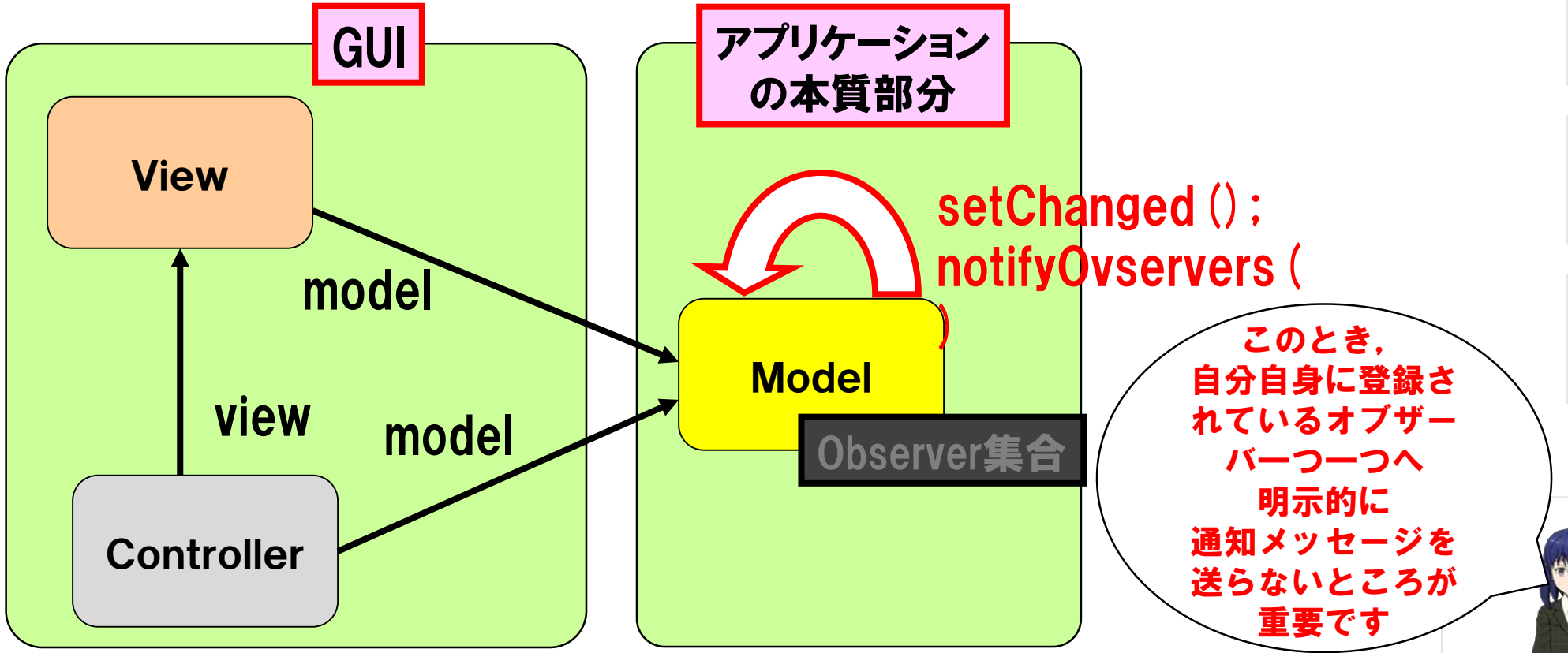
Observer/Observableパターンの動き(3/5)

- Observer/Observableパターンの動き:【モデルによる自分自身への変更通知】
  - setChanged();notifyOvserbers();を呼び出す
  - 自分に登録されているオブザーバー一つ一つへ，明示的に通知メッセージを送らないところが重要



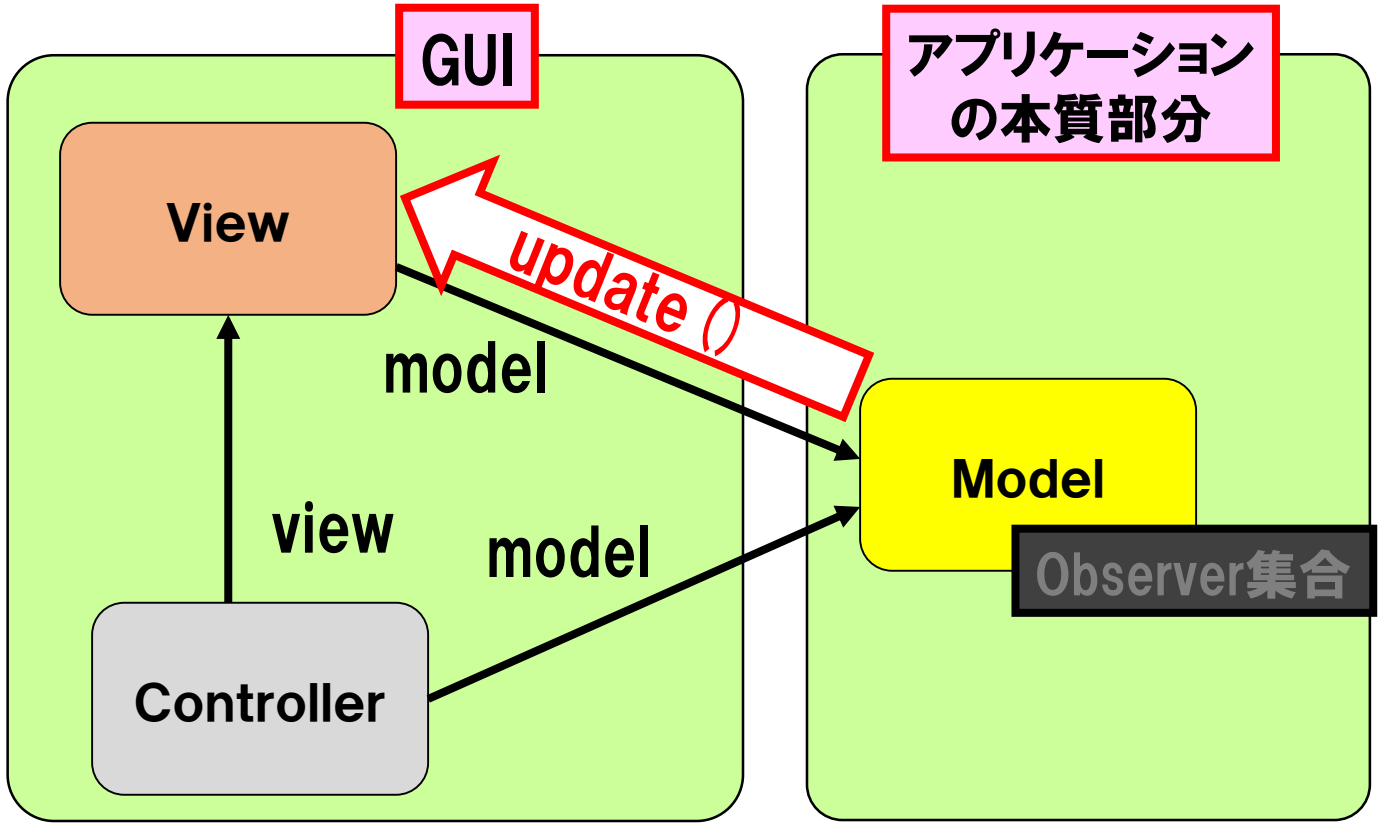
Observer/Observableパターンの動き(3/5)

- Observer/Observableパターンの動き:【モデルによる自分自身への変更通知】
  - setChanged();notifyOvserbers();を呼び出す
  - 自分に登録されているオブザーバー一つ一つへ，明示的に通知メッセージを送らないところが重要



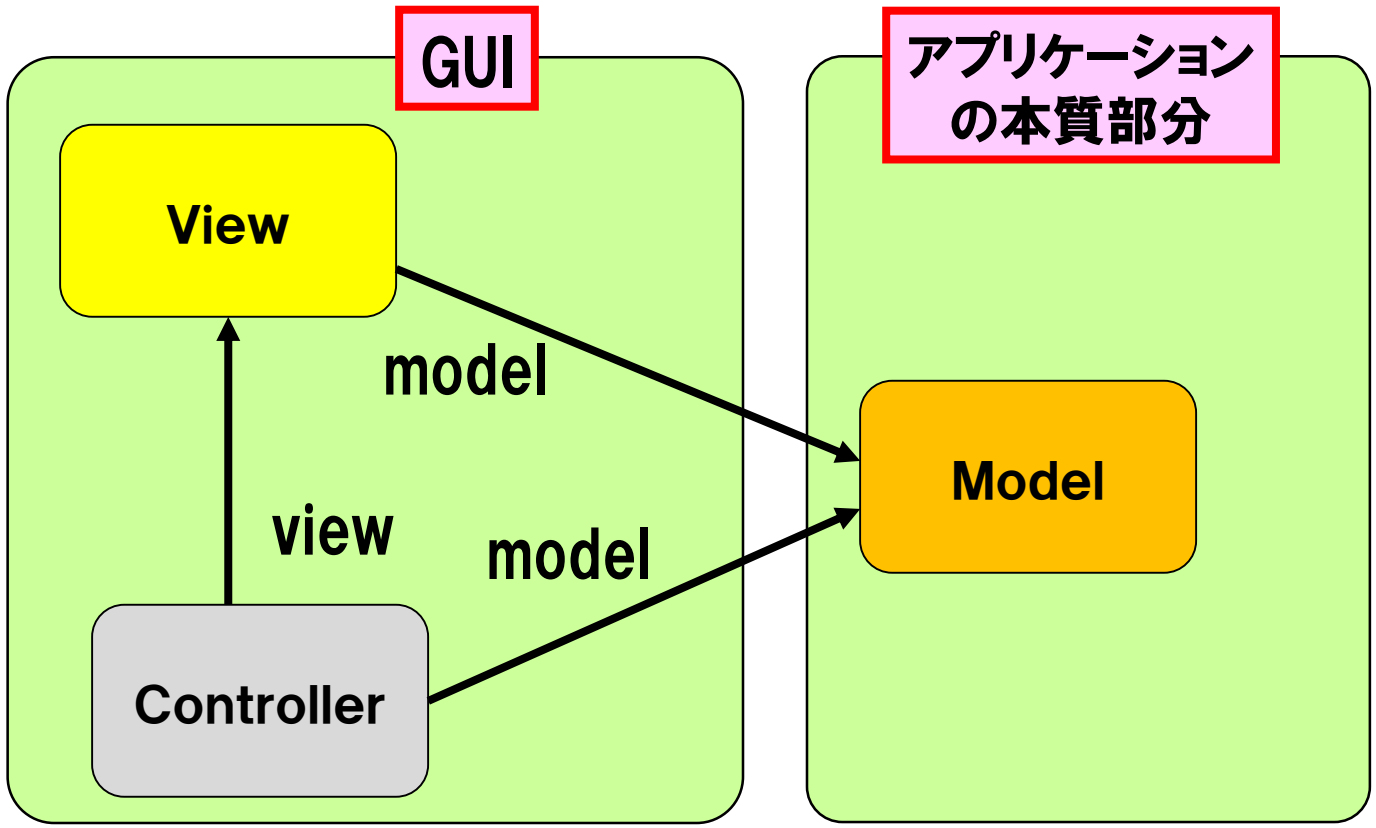
Observer/Observableパターンの動き (4/5)

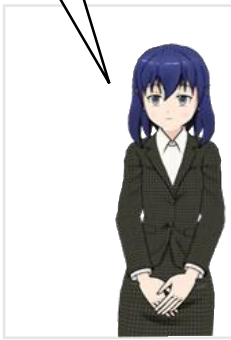
- Observer/Observableパターンの動き:
  【暗黙的に，個々のビューへ通知が行われる】
  - 暗黙的に，登録されているすべてのビューへ，update () メッセージが送られる



Observer/Observableパターンの動き (5/5)

- Observer/Observableパターンの動き: 【ビューが更新される】
  - update () メッセージに反応して，update () メソッドでビューを更新する
  - モデルの内部状態がどのように変化したかは，（パラメータと）モデルのゲッターで取得する

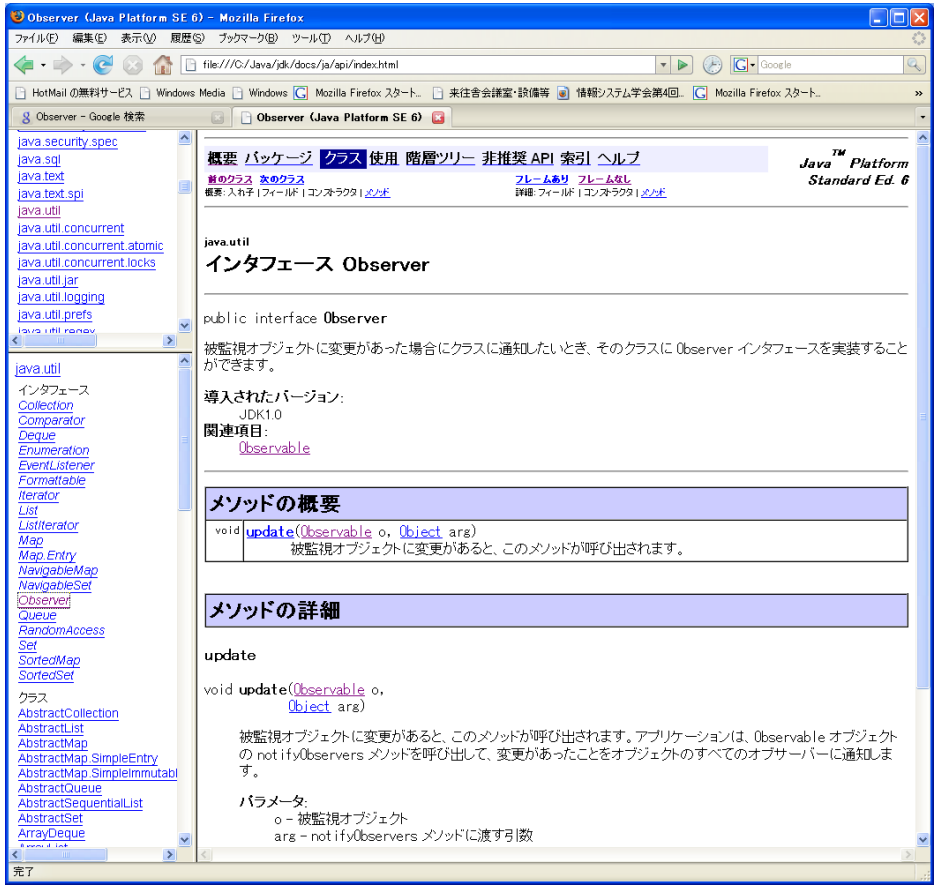




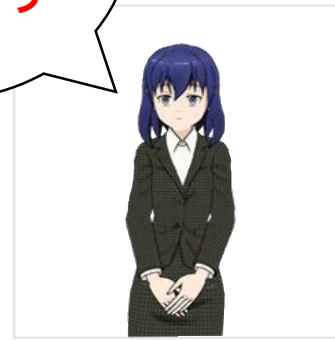
ObserverとObservableのAPI

Observerインタフェースの API

- java.utilパッケージ
  Observerインタフェース



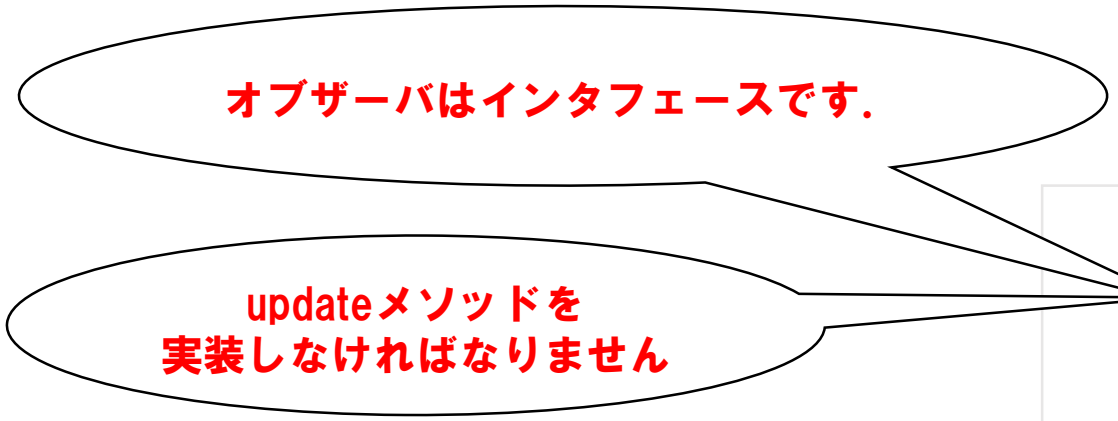
APIドキュメントを見てみましょう



Observerインタフェースの API

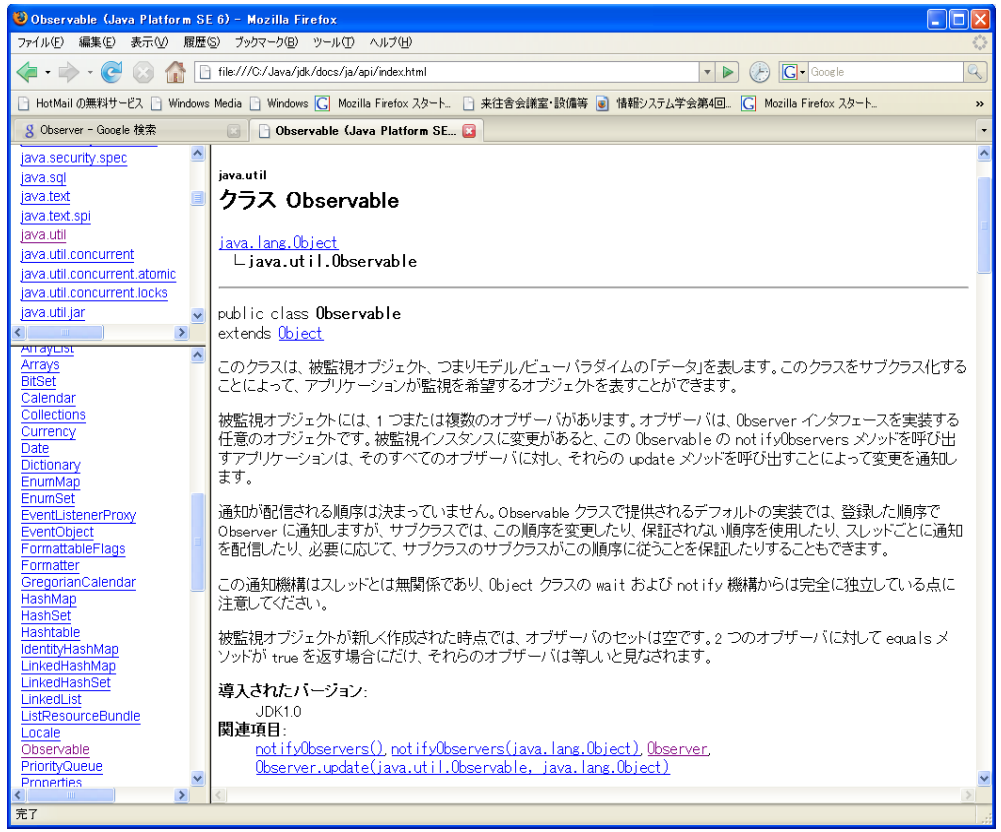
- java.utilパッケージ
  Observerインタフェース

	型	名前	引数	説明
メソッド	void	update		被監視オブジェクトに変更があると， このメソッドが呼び出される
			Observable o	被監視オブジェクト
			Object arg	notifyObserversメソッドに渡す引数

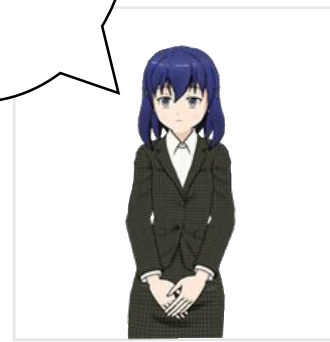


ObservableクラスのAPI

- java.utilパッケージ
  Observableクラス



APIドキュメントを見てみましょう



ObservableクラスのAPI

- java.utilパッケージ

  Observableクラス

	型	名前	引数	説明
コンストラクタ	Observable			被監視オブジェクトを生成する．オブザーバは未登録．
メソッド	void	update		被監視オブジェクトに変更があると，呼び出される
			Observable o	被監視オブジェクト
			Object arg	notifyObserversメソッドに渡す引数

updateメソッドには，
どの被監視オブジェクトで
変更が起こったかを引数で
識別できる．
また，変更内容を識別するための
引数を送ることもできる．

iijima@ae.keio.ac.jp

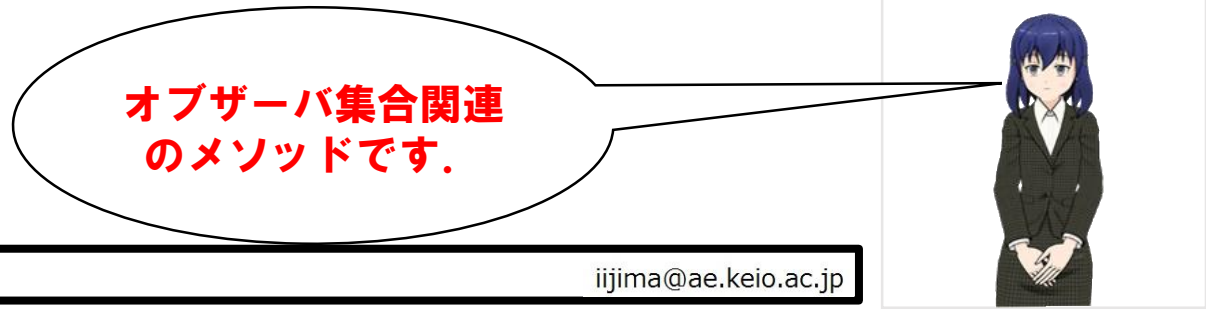
ObservableクラスのAPI

- java.utilパッケージ
  Observableクラス

オブザーバ集合関連のメソッド

	型	名前	引数	説明
メソッド	void	addObserver		監視オブジェクトをオブザーバ集合に追加登録する
			Observer o	追加登録する監視オブジェクト(オブザーバ)

	型	名前	引数	説明
メソッド	int	countObservers		オブザーバ集合に登録されている監視オブジェクトを数える
	void	deleteObservers		監視オブジェクトを全てオブザーバ集合から削除する
	void	deleteObserver		監視オブジェクトをオブザーバ集合から削除する
			Observer o	削除する監視オブジェクト(オブザーバ)



iijima@ae.keio.ac.jp

ObservableクラスのAPI

- java.utilパッケージ
  Observableクラス

変更マーク関連のメソッド

	型	名前	引数	説明
メソッド	protected void	setChanged		オブジェクトを変更されたものとしてマーキングする. マーキングしただけでは，通知されない.

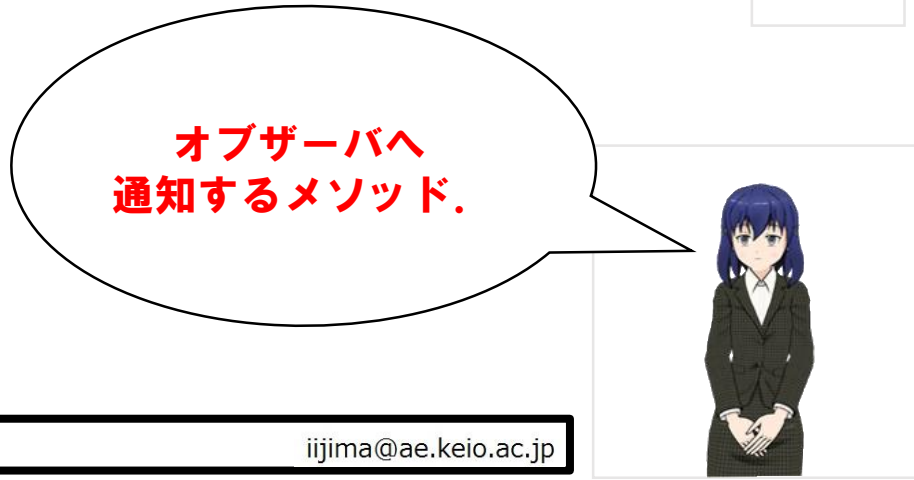
	型	名前	引数	説明
メソッド	protected void	clearChanged		オブジェクトがもはや変更された状態ではないこと，すなわち，最新の変更がすべてオブザーバに通知されたことを示す
	boolean	hasChanged		オブジェクトが変更されたかどうかを判定する

ObservableクラスのAPI

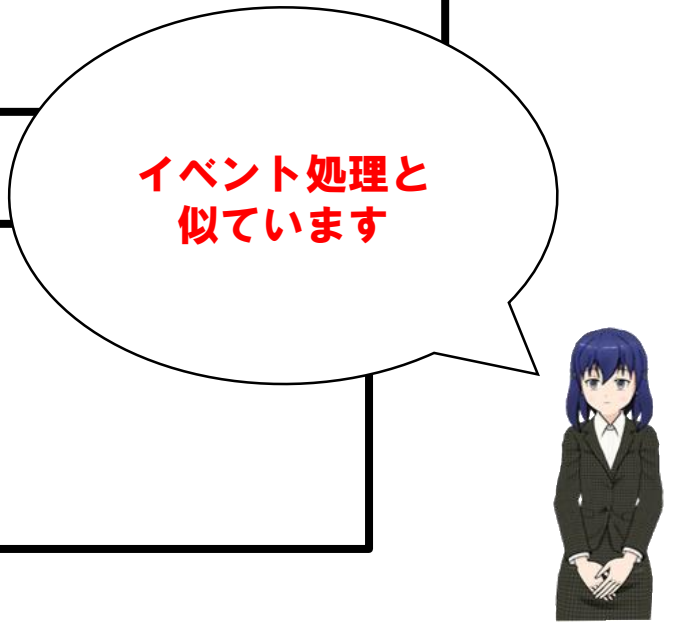
- java.utilパッケージ
  Observableクラス

オブザーバへ通知するメソッド

	型	名前	引数	説明
メソッド	void	notifyObservers		オブジェクトが、hasChanged メソッドに示されるように変更されていた場合、そのすべてのオブザーバにそのことを通知し、次に clearChanged メソッドを呼び出して、このオブジェクトがもはや変更された状態でないことを示します
	void	notifyObservers		
			Object arg	監視オブジェクト(オブザーバ)へ渡す引数



イベント処理との類似

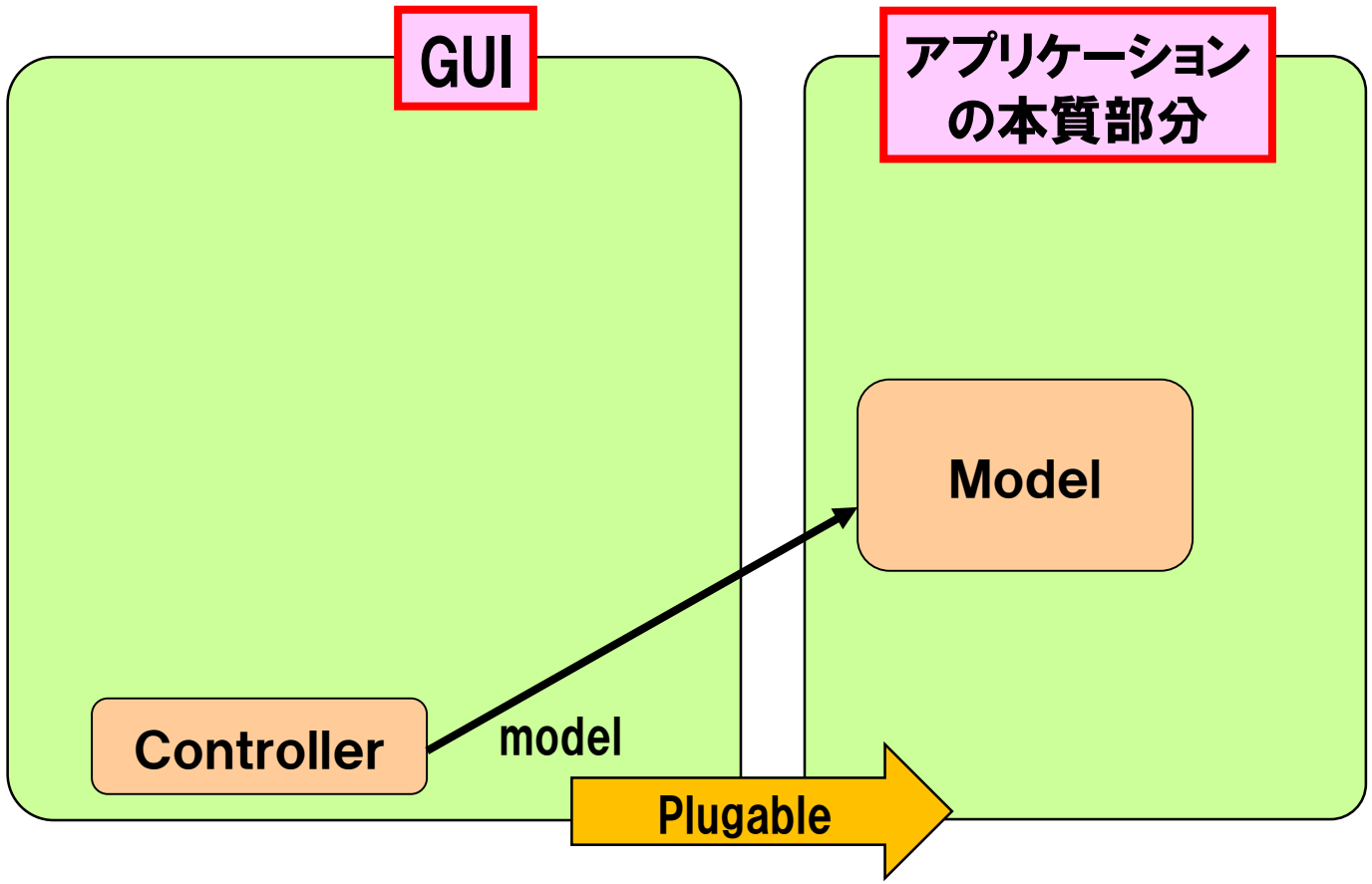


多重ビュー

モデルには
複数のビューを
関連付けることが
できます

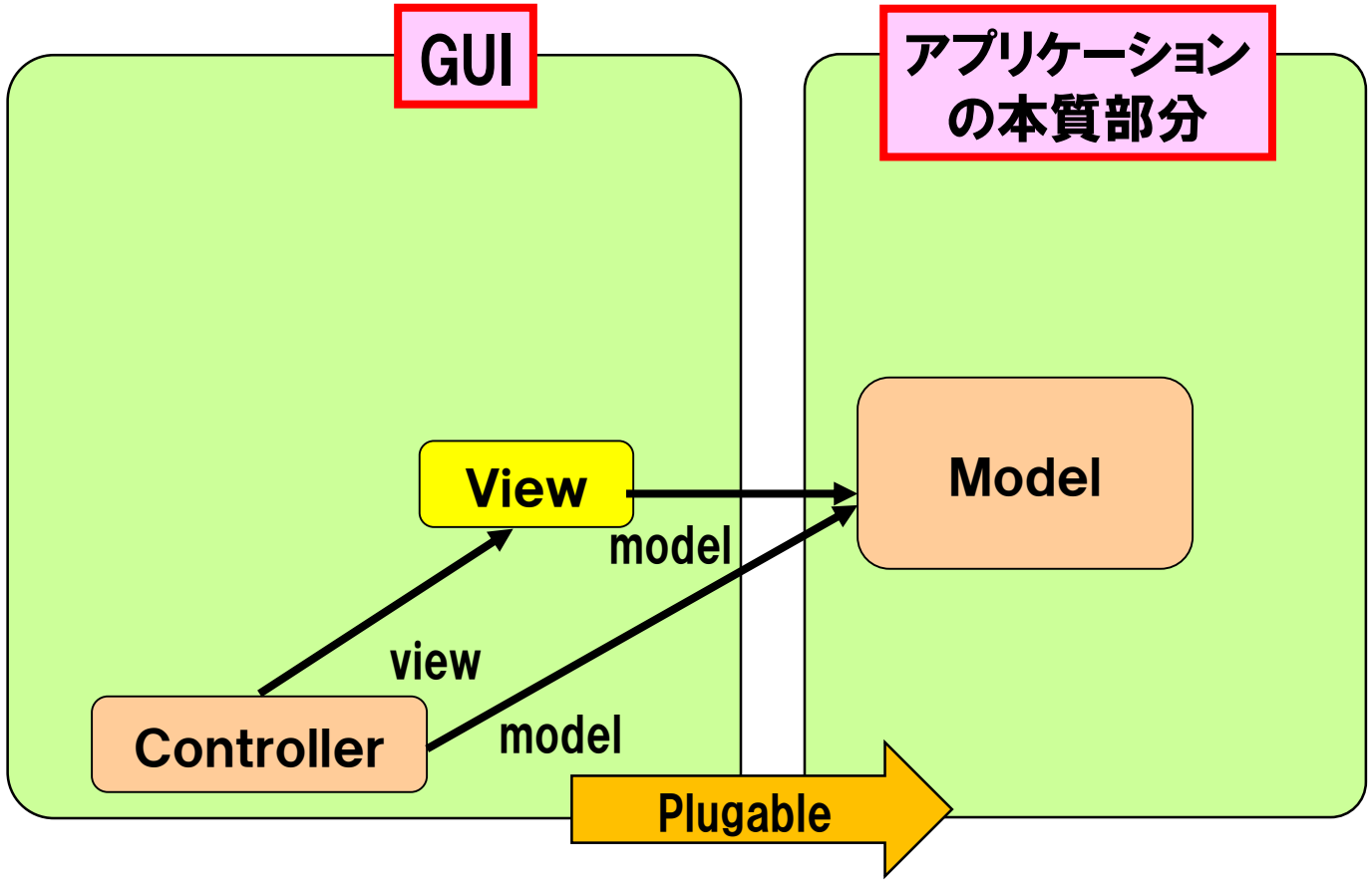
プラガブルなUI（1/4）

- 多重ビュー:
  - 一つのモデルに複数のビューとコントローラを後から追加することもできる



プラガブルなUI（2/4）

- 多重ビュー:
  - 一つのモデルに複数のビューとコントローラを後から追加することもできる

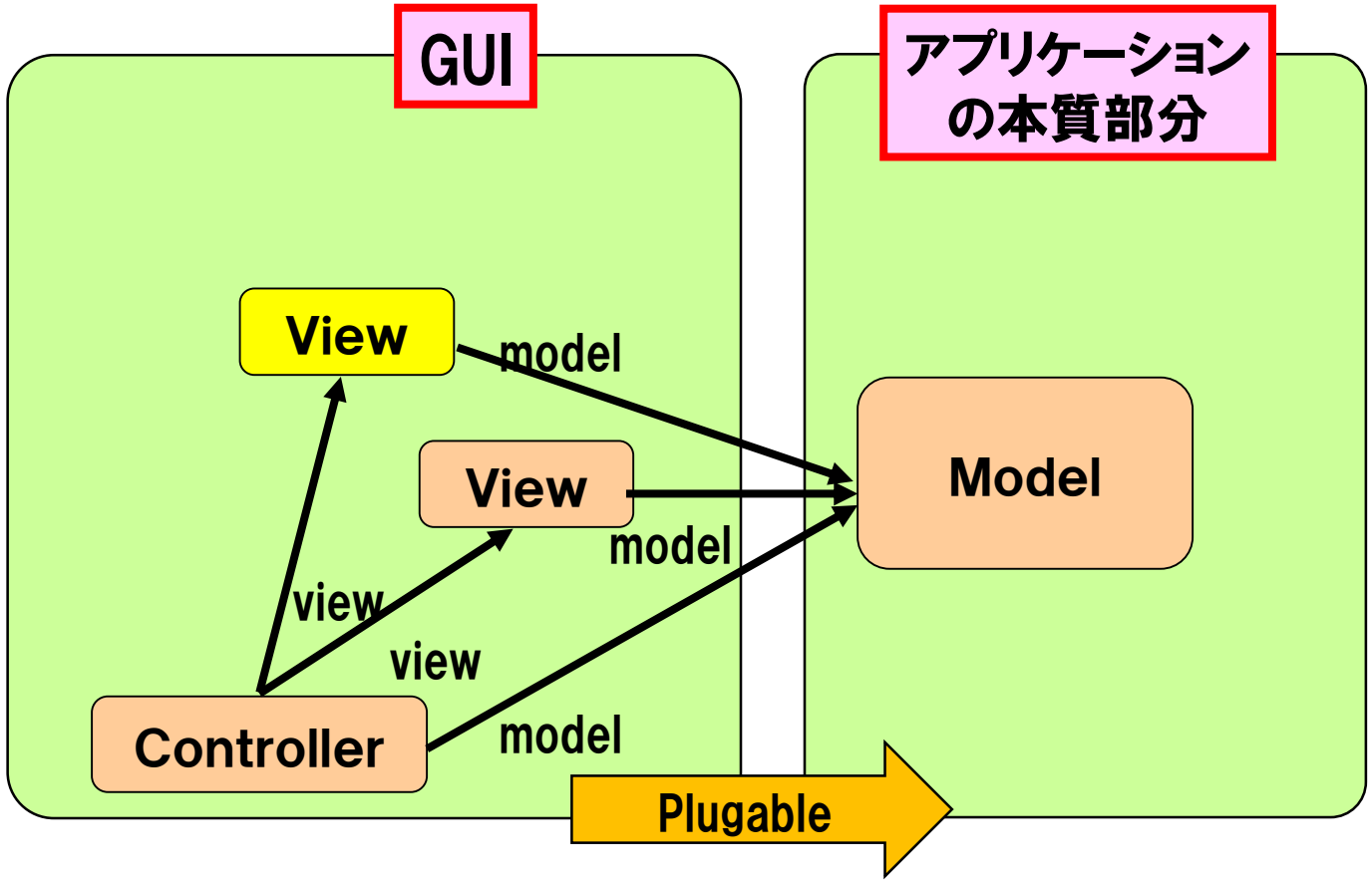


後から
ビューを
追加できます

iijima@ae.keio.ac.jp

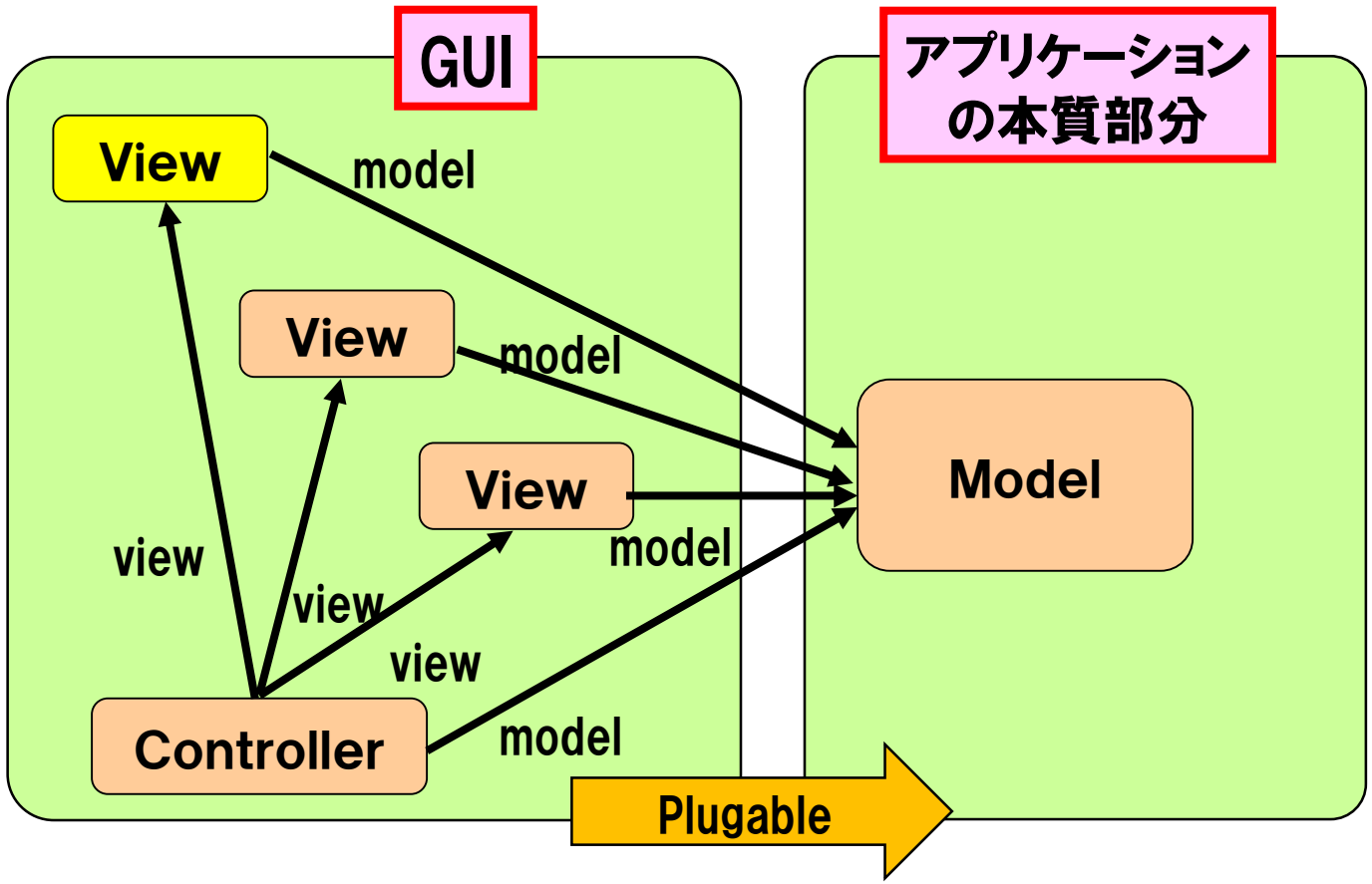
プラガブルなUI（3/4）

- 多重ビュー:
  - 一つのモデルに複数のビューとコントローラを後から追加することもできる



プラガブルなUI（4/4）

- 多重ビュー:
  - 一つのモデルに複数のビューとコントローラを後から追加することもできる



いくつ追加しても
モデル自体の
コードは
変更する必要が
ありません

iijima@ae.keio.ac.jp

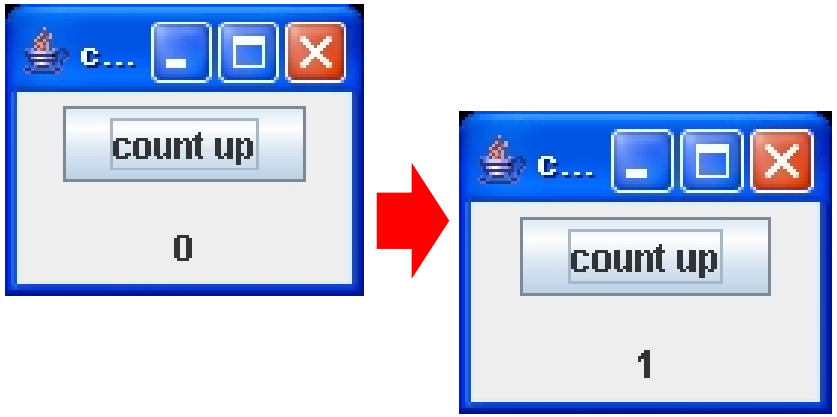
簡単な例題: カウンタ

MVCの例題

- マウスのクリック回数を画面に表示するプログラム
  - Counter
  - CounterView
  - CounterController

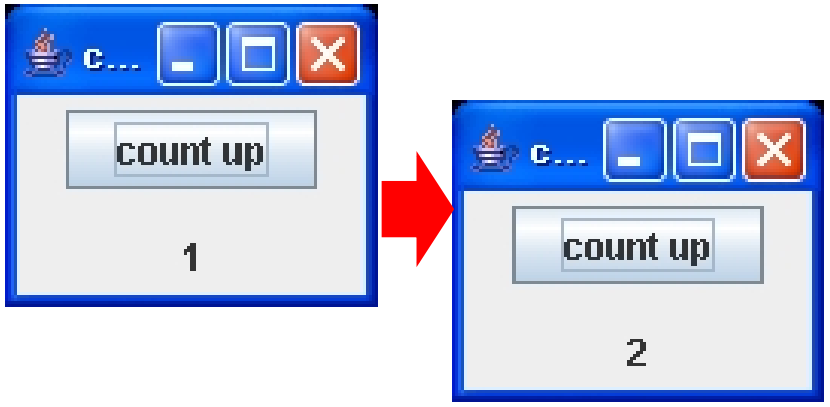
MVCの例題

- マウスのクリック回数を画面に表示するプログラム
  - Counter
  - CounterView
  - CounterController



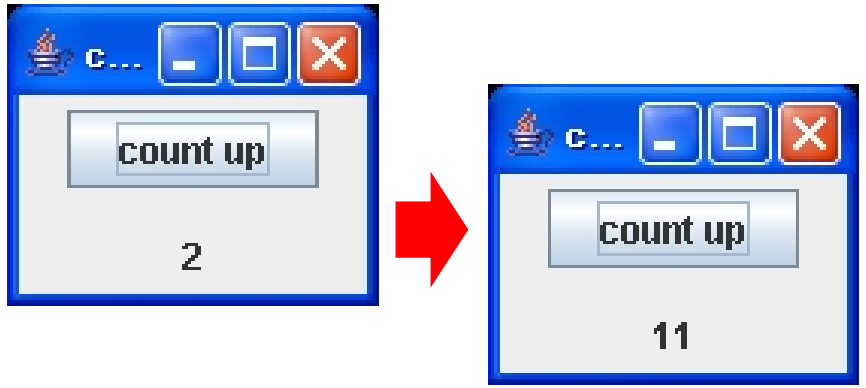
MVCの例題

- マウスのクリック回数を画面に表示するプログラム
  - Counter
  - CounterView
  - CounterController



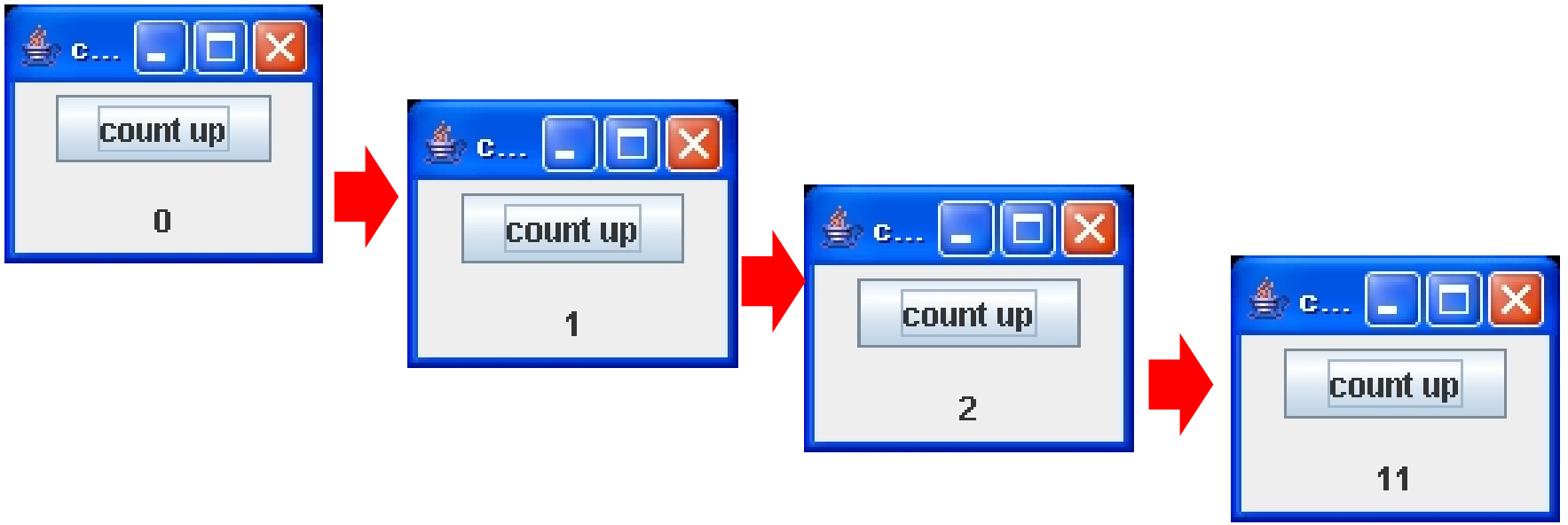
MVCの例題

- マウスのクリック回数を画面に表示するプログラム
  - Counter
  - CounterView
  - CounterController



MVCの例題

- マウスのクリック回数を画面に表示するプログラム
  - Counter
  - CounterView
  - CounterController



オブザーバであるビューがそれを検出して，自身を更新するモデルです

例題の動き（1/7）

CounterView
のインスタンス

CounterModel
のインスタンス
count

① ボタンの
クリック

CounterController
のインスタンス

ボタンのクリックによりイベントが発生し，イベントリスナーに暗黙的な通知が行われます

例題の動き（2/7）

CounterView
のインスタンス

CounterModel
のインスタンス
count

CounterController
のインスタンス

② countUp()

イベントリスナの持つイベントハンドラが反応します．

モデルへ，状態を更新するメッセージが送られます

iijima@ae.keio.ac.jp

例題の動き（3/7）

CounterView
のインスタンス

CounterController
のインスタンス

③ setChanged ();
notifyObservers ()

CounterModel
のインスタンス
count

モデルは自身の内部状態を更新したら，更新があったことをマークします

一連の更新が終わったら，オブザーバへの通知を行います

但し，このとき，具体的なビューを意識しません

例題の動き（4/77）

CounterView
のインスタンス

④ update ()

CounterModel
のインスタンス
count

登録されているビューへ更新指示が通知されます

CounterController
のインスタンス

iijima@ae.keio.ac.jp

例題の動き（5/7）

CounterView
のインスタンス

⑤ getCount ()

CounterModel
のインスタンス
count

CounterController
のインスタンス

ビューの側
から
モデルの内
部状態を
調べに行き
ます

iijima@ae.keio.ac.jp

例題の動き（6/7）

⑥ 表示更新

CounterView
のインスタンス

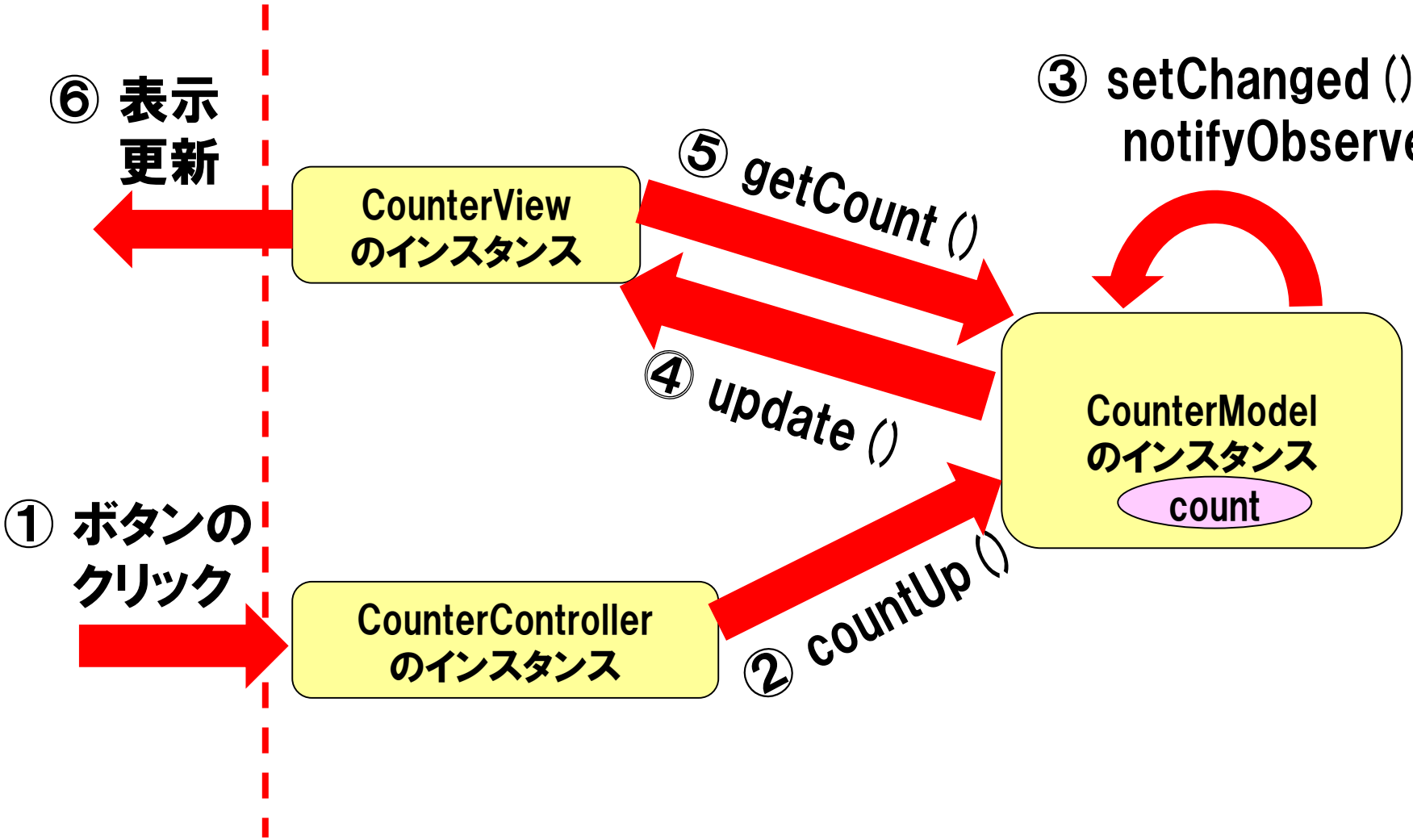
CounterModel
のインスタンス
count

CounterController
のインスタンス

ビューを
更新します

iijima@ae.keio.ac.jp

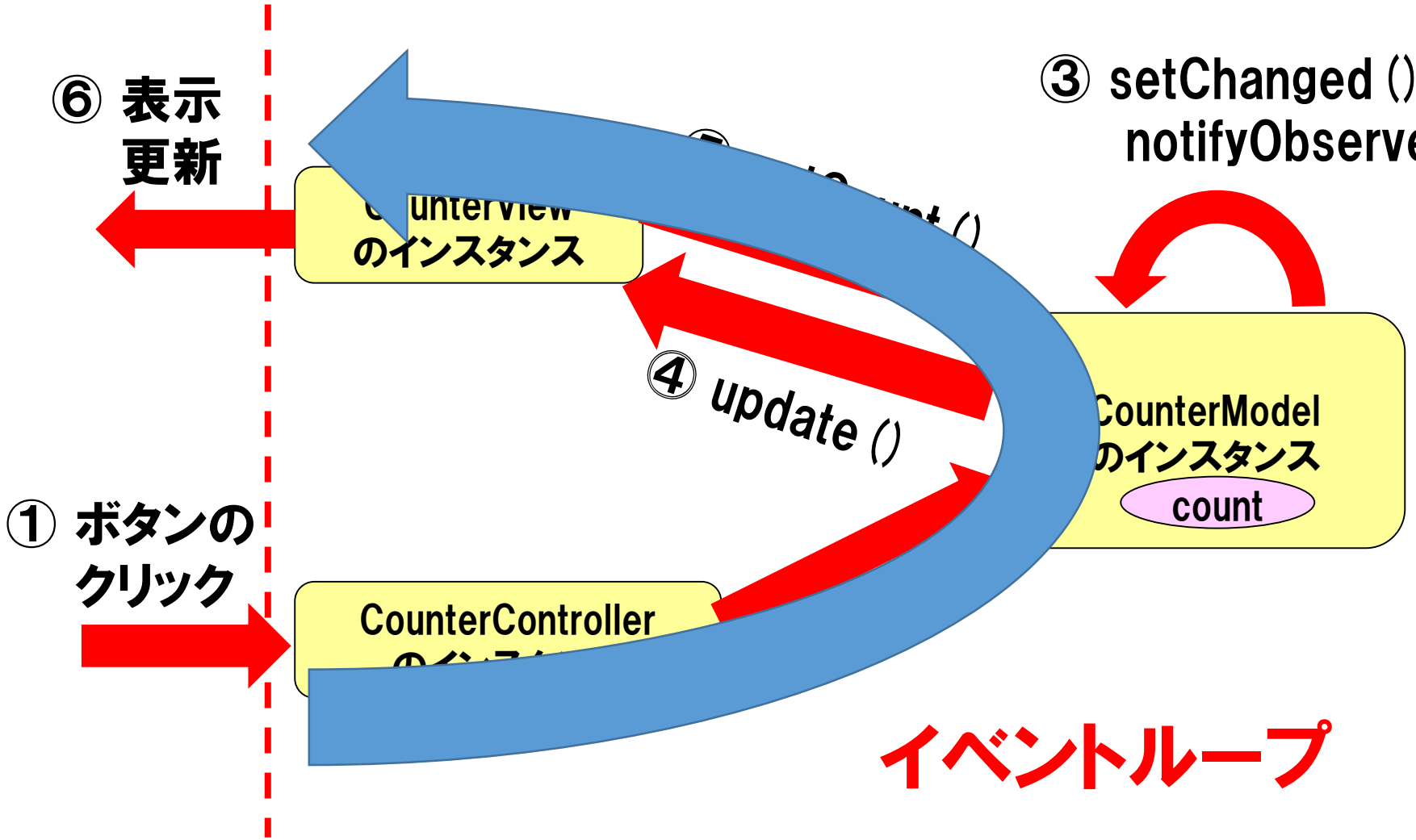
例題の動き（7/7）



一連の流れを示すとこうなります

iijima@ae.keio.ac.jp

例題の動き（7/7）

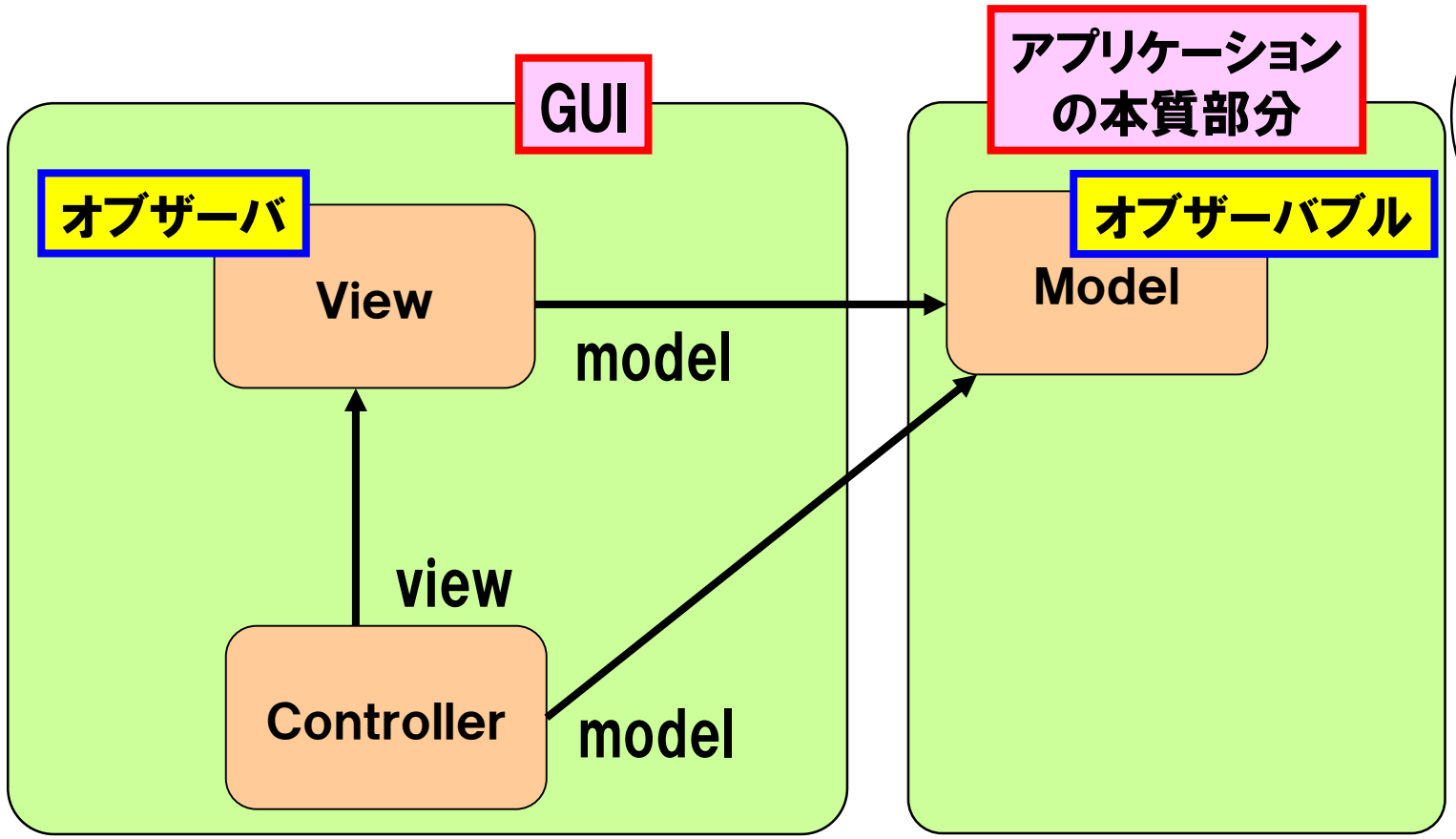


イベントループ

イベント
ループ

Observer/Observableパターンとイベント処理（01/11）

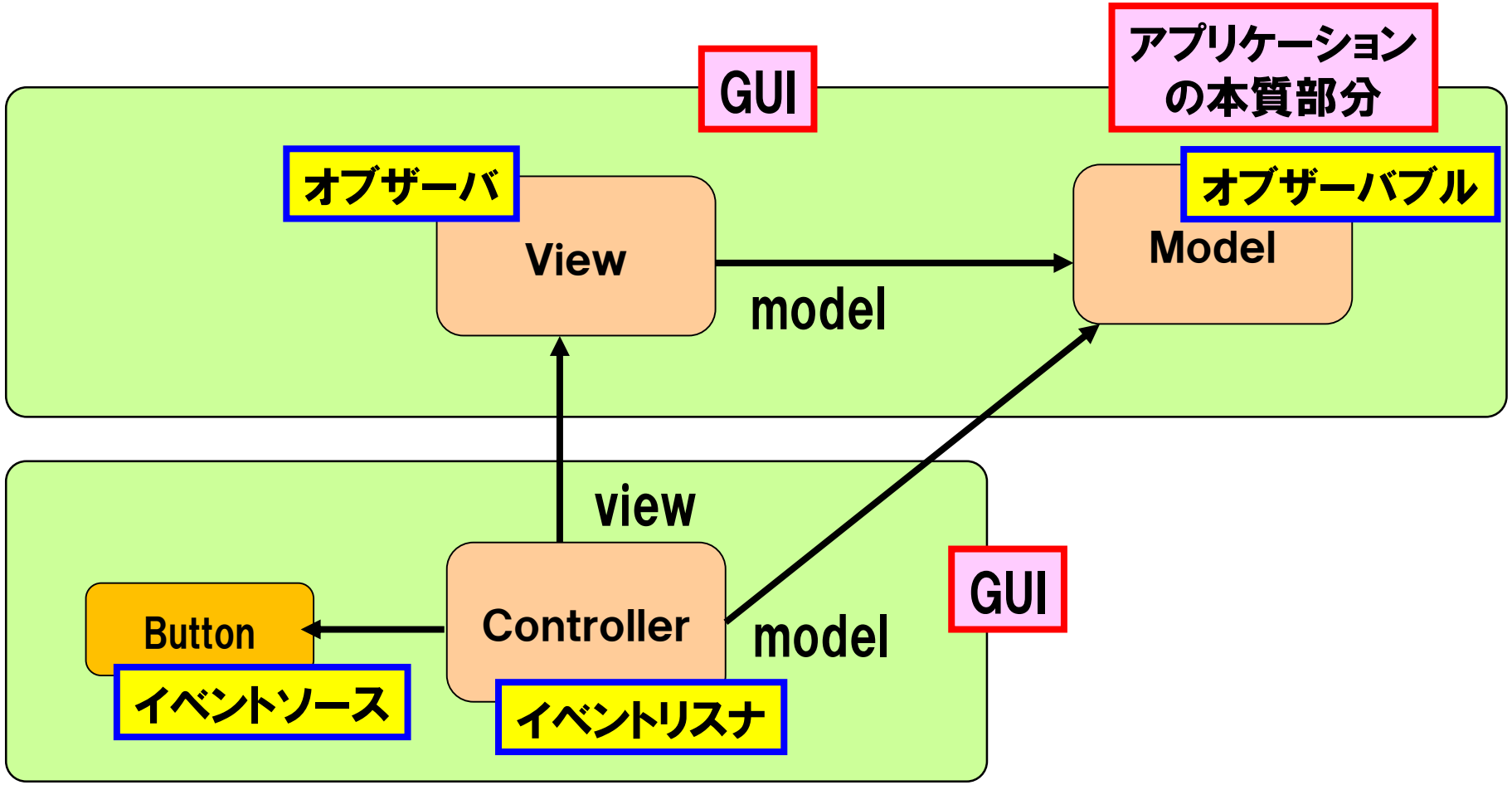
- Observer/Observableパターンとイベント処理：



モデルとビューの間はObserver / Observable パターンで，非同期的に疎結合されています

Observer/Observableパターンとイベント処理（02/11）

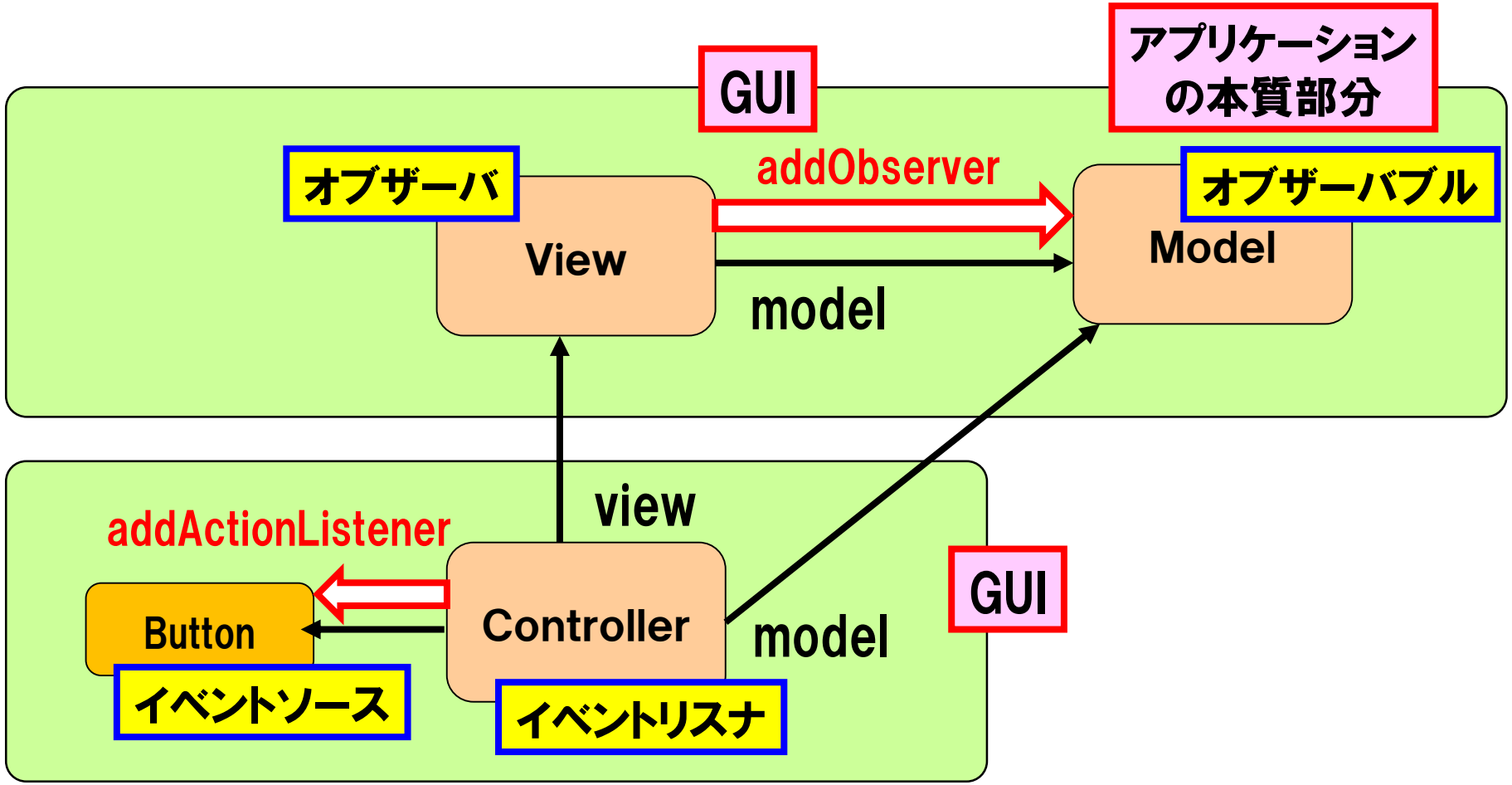
- Observer/Observableパターンとイベント処理：



コントローラと
GUI部品は，
イベントハンドラで
疎結合されており，
その結果，
モデルへの操作が
イベントをトリガと
して，
非同期的に
呼び出されます

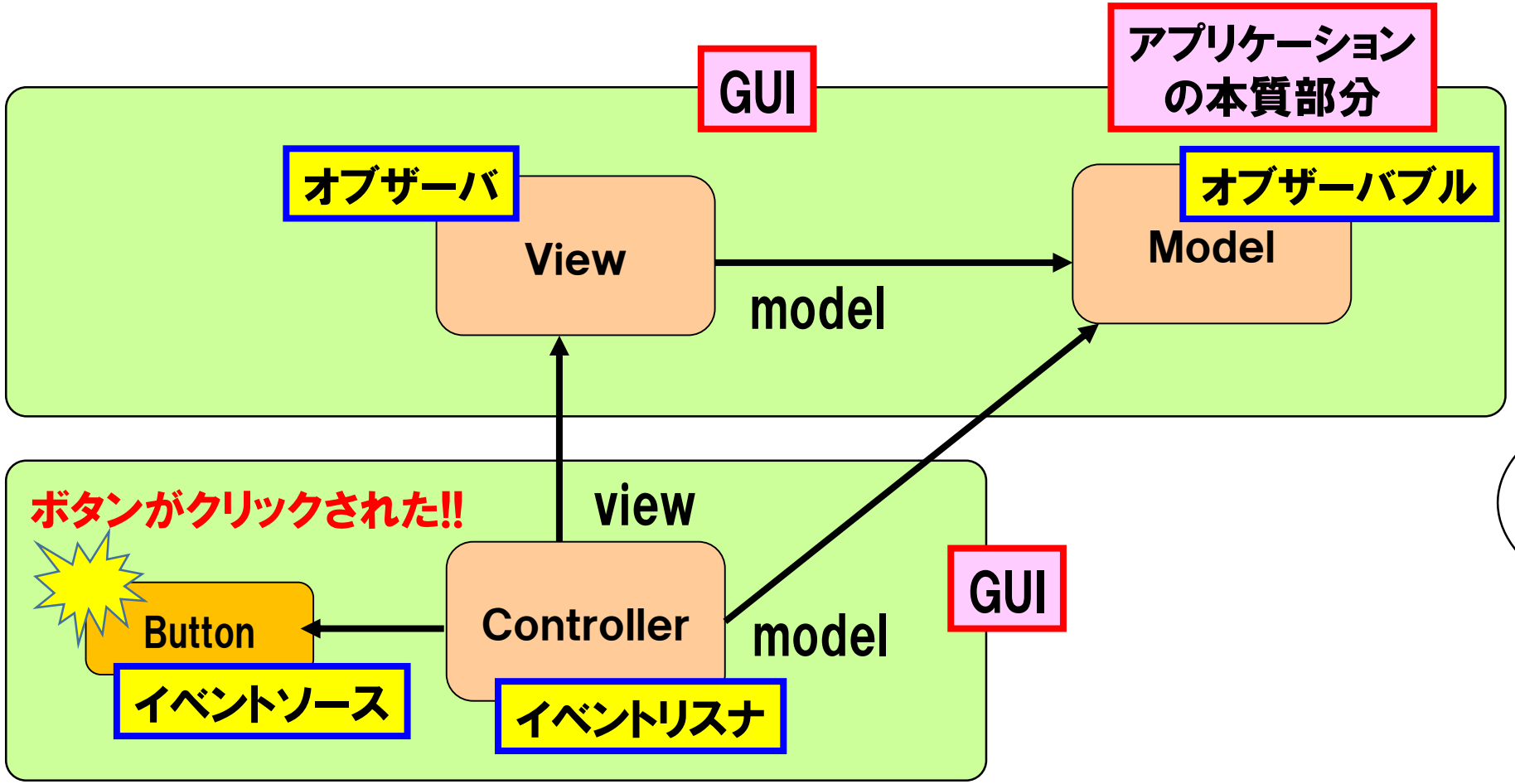
Observer/Observableパターンとイベント処理（03/11）

- Observer/Observableパターンとイベント処理：　【事前登録】



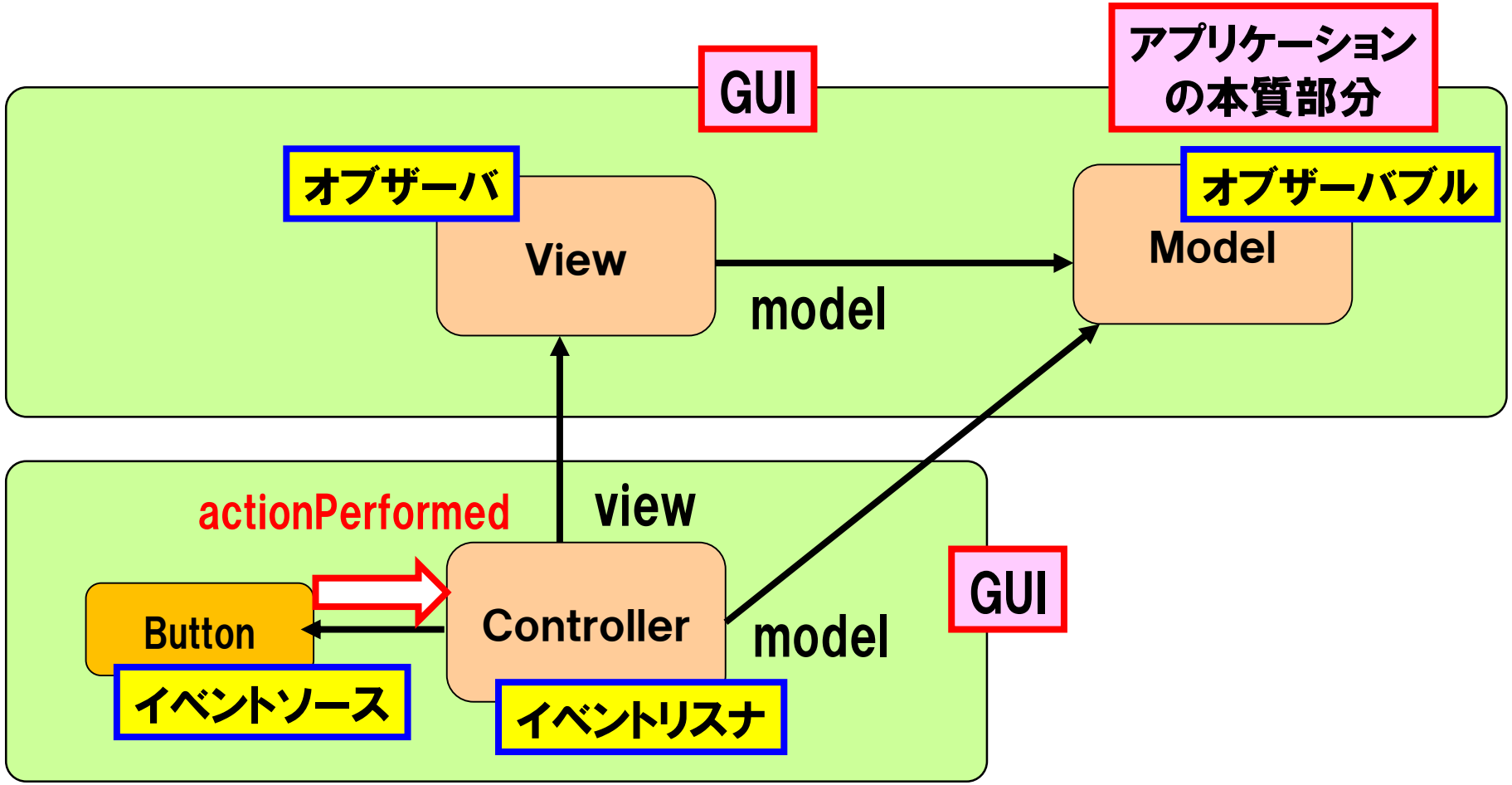
Observer/Observableパターンとイベント処理（04/11）

- Observer/Observableパターンとイベント処理：　【イベントの発生】



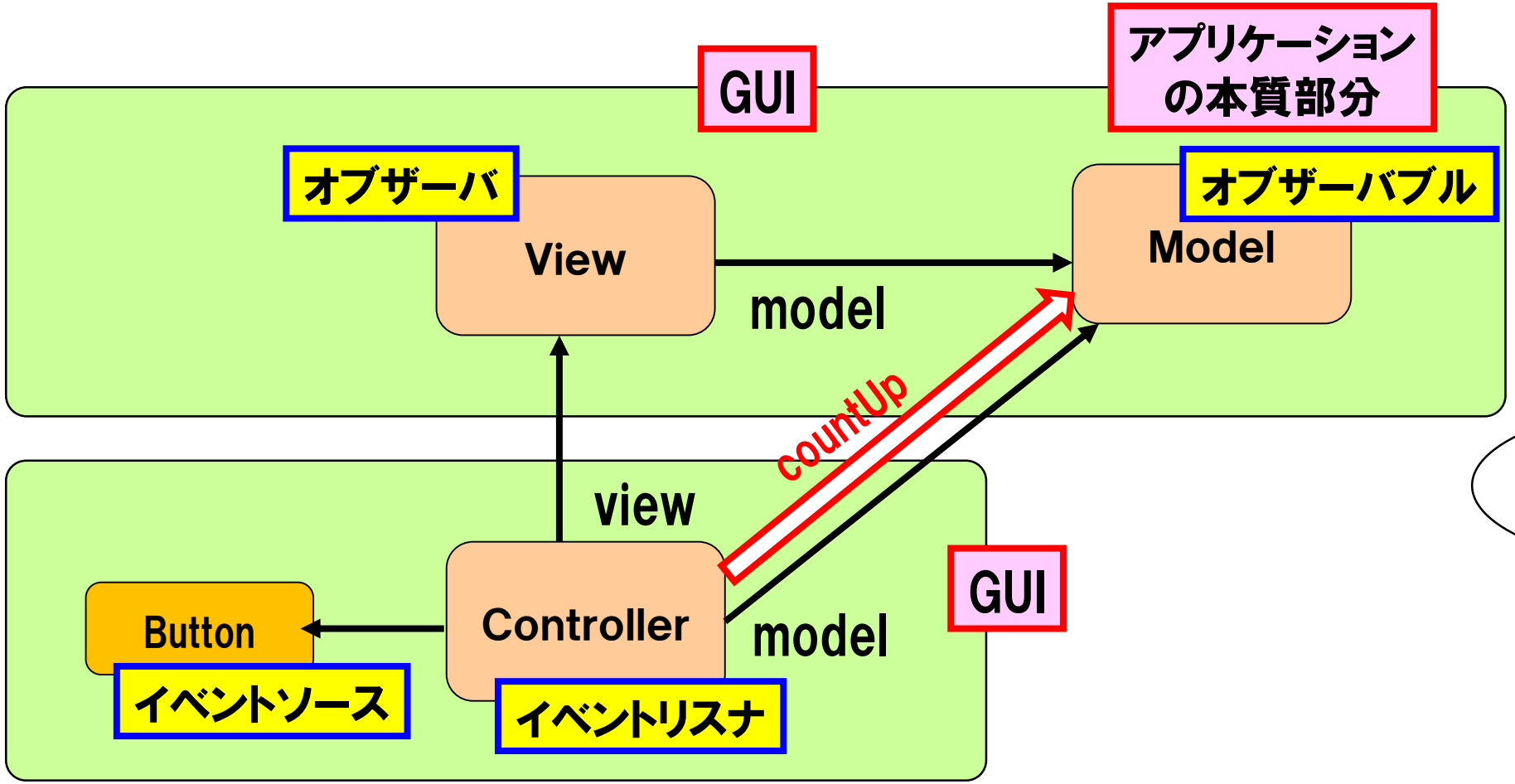
Observer/Observableパターンとイベント処理（05/11）

- Observer/Observableパターンとイベント処理：　【イベントハンドラの起動】



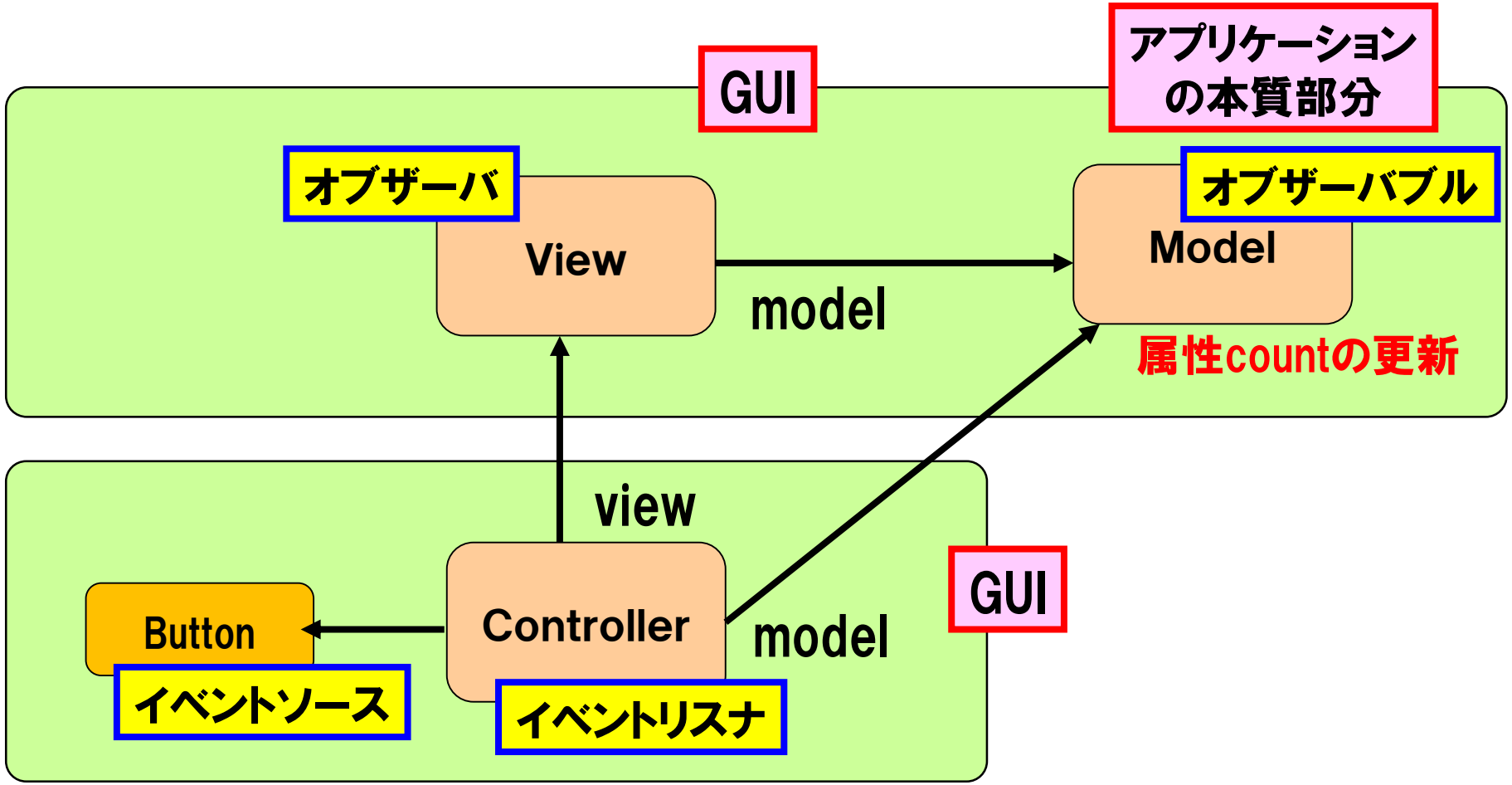
Observer/Observableパターンとイベント処理（06/11）

- Observer/Observableパターンとイベント処理：　【モデルが操作される】



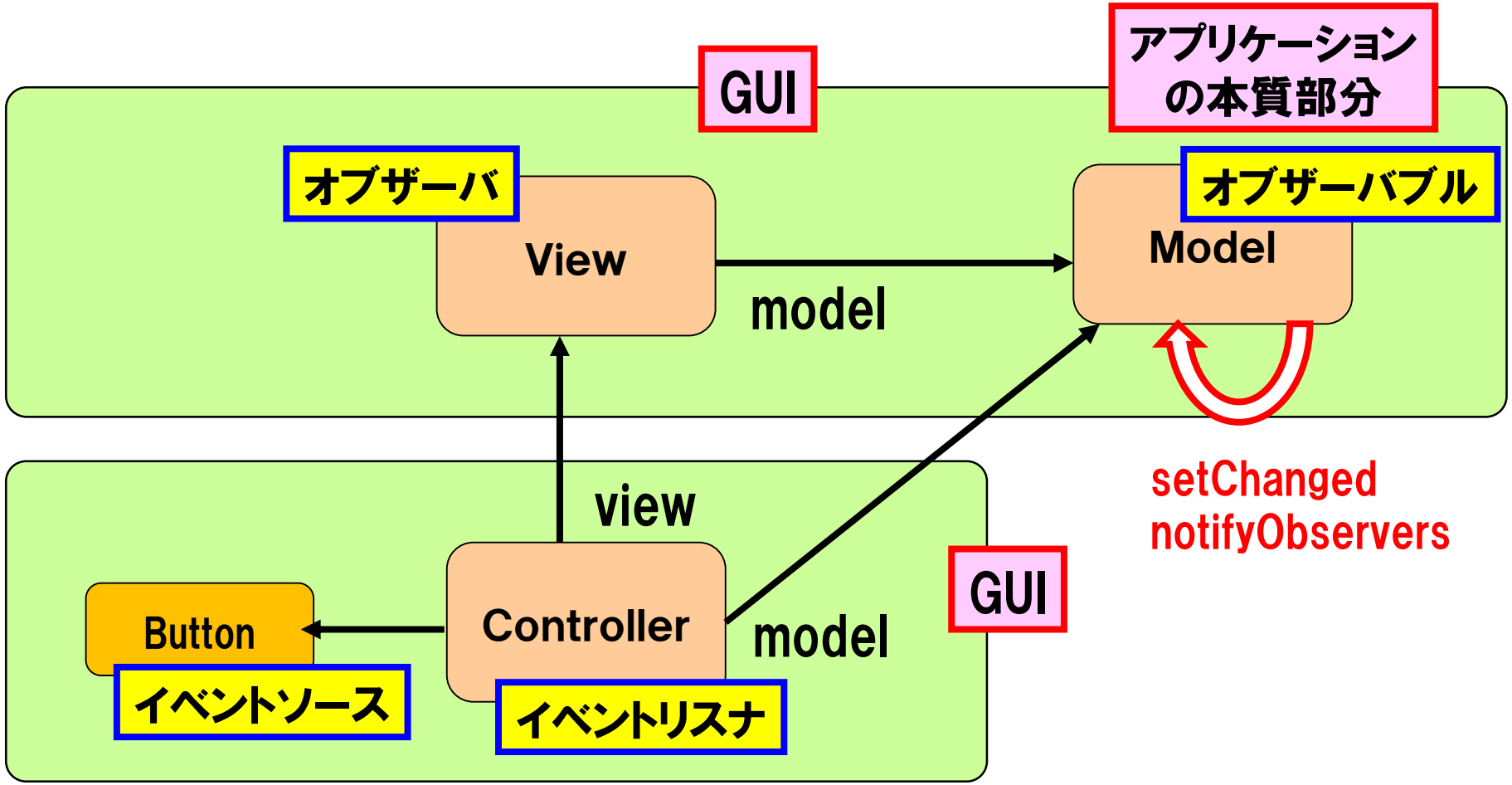
Observer/Observableパターンとイベント処理（07/11）

- Observer/Observableパターンとイベント処理：　【モデルの内部状態の更新】



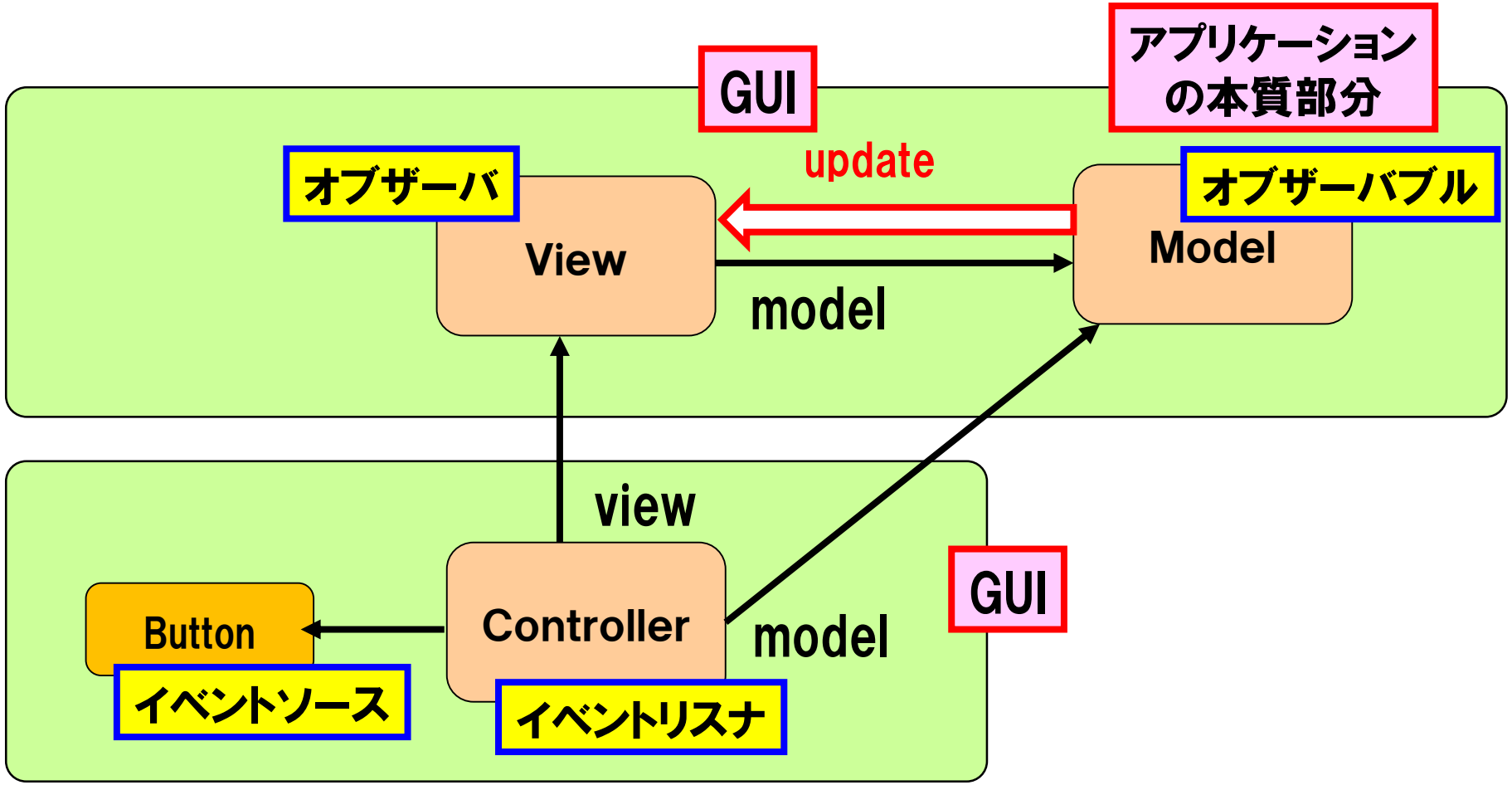
Observer/Observableパターンとイベント処理（08/11）

- Observer/Observableパターンとイベント処理：　【更新通知】



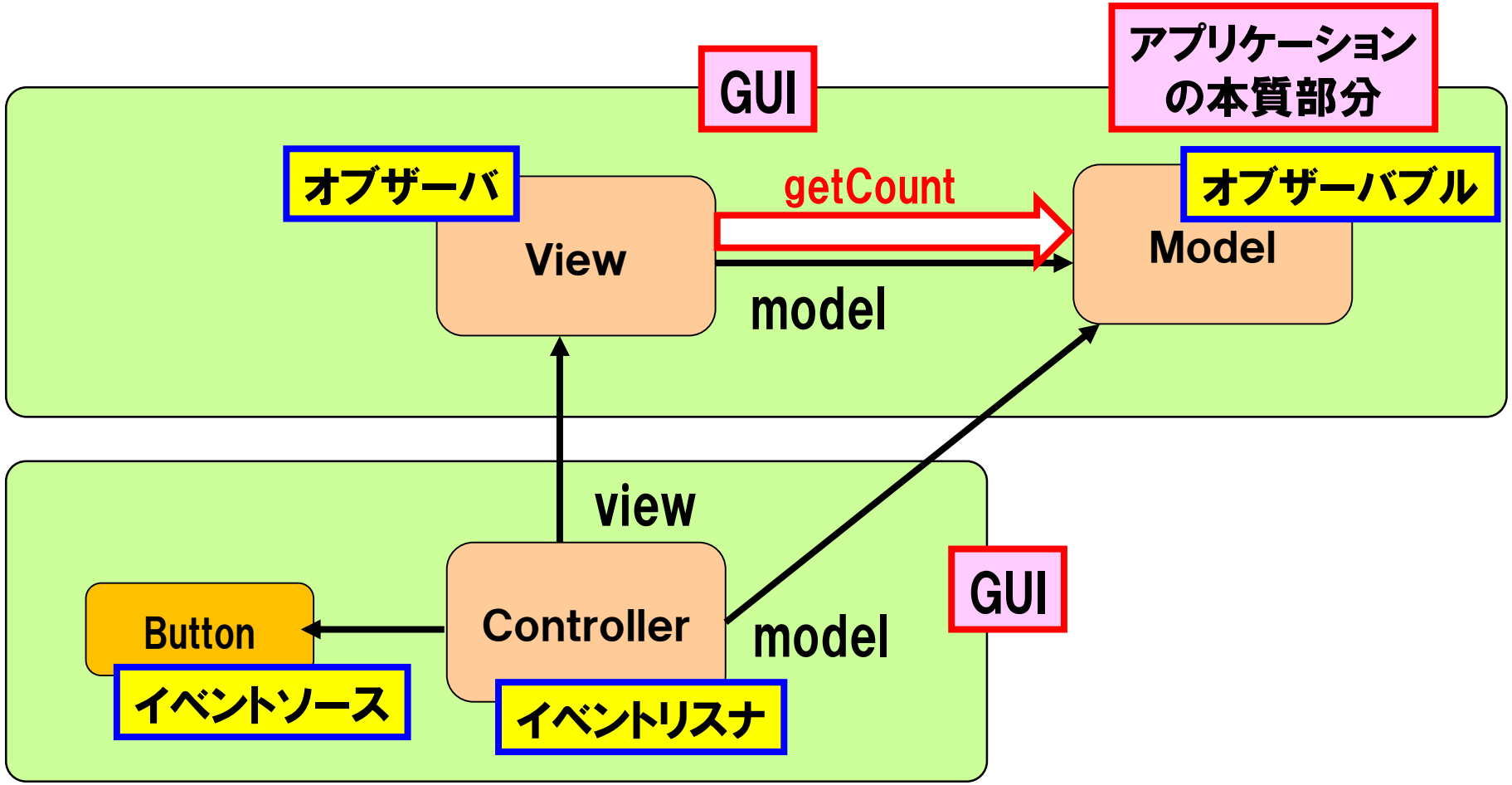
Observer/Observableパターンとイベント処理（09/11）

- Observer/Observableパターンとイベント処理：　【ビューの更新指示】



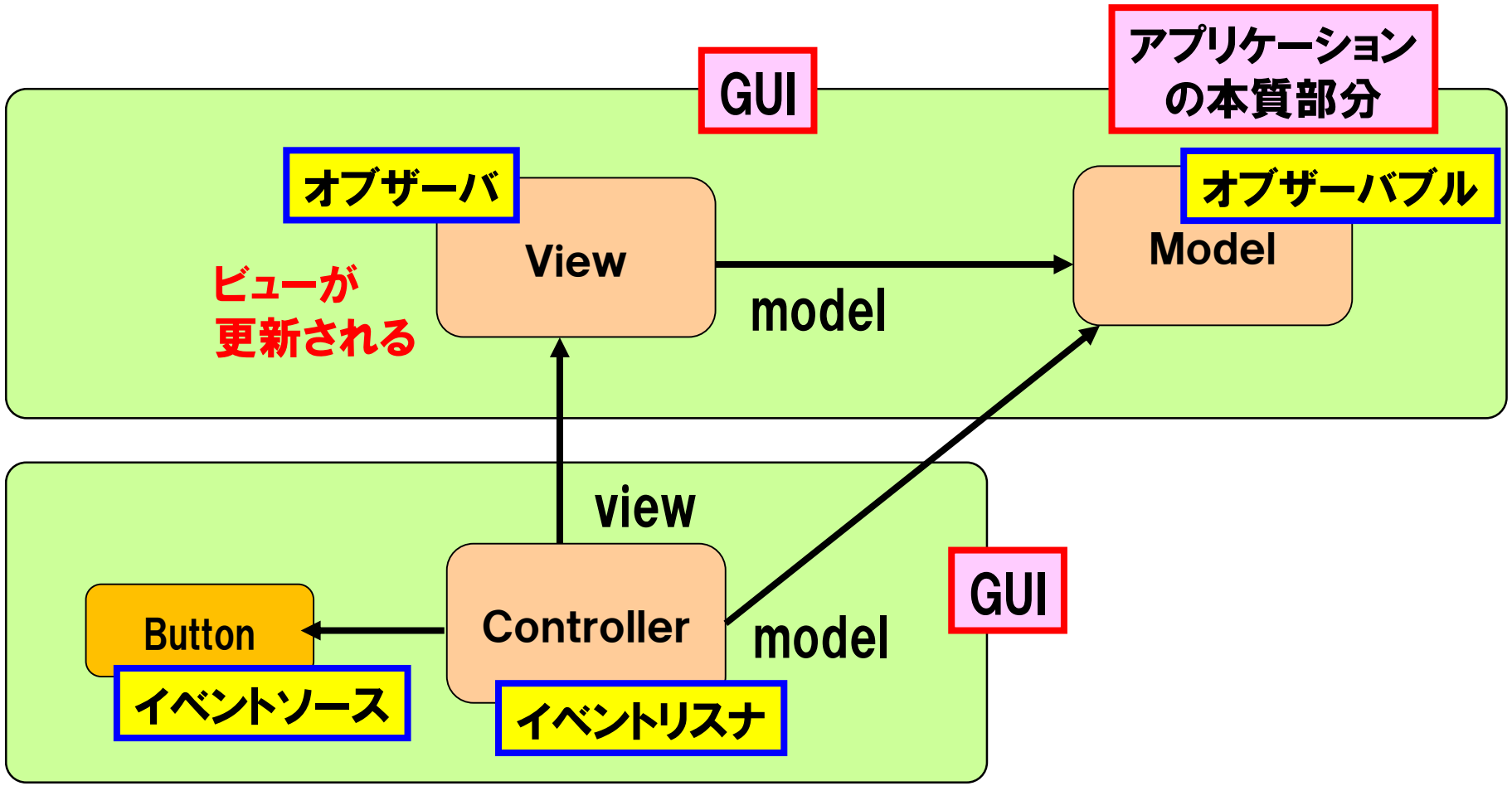
Observer/Observableパターンとイベント処理（10/11）

- Observer/Observableパターンとイベント処理：　【モデルの内部状態を取得】



Observer/Observableパターンとイベント処理（11/11）

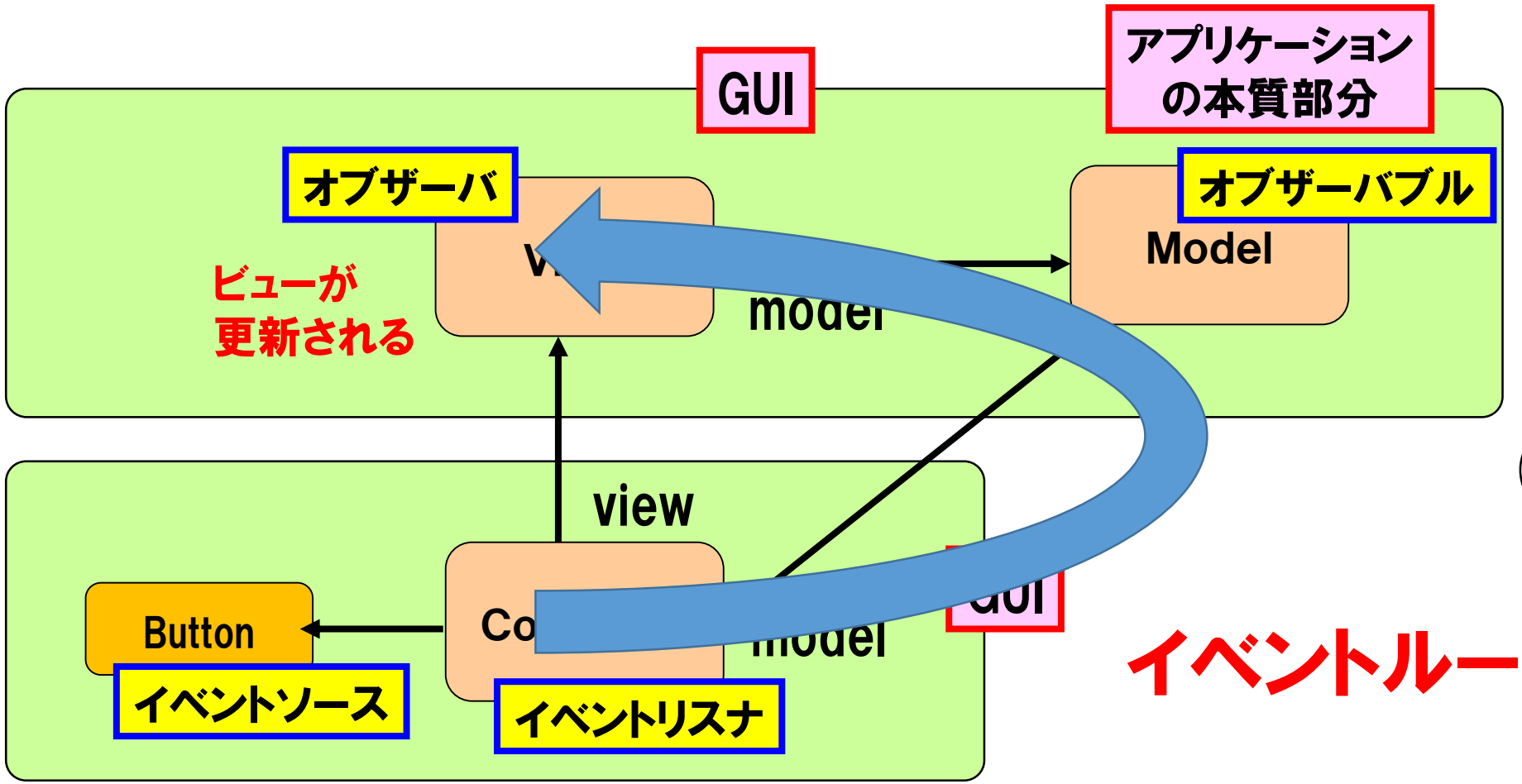
- Observer/Observableパターンとイベント処理：　【ビューが更新される】



iijima@ae.keio.ac.jp

Observer/Observableパターンとイベント処理（11/11）

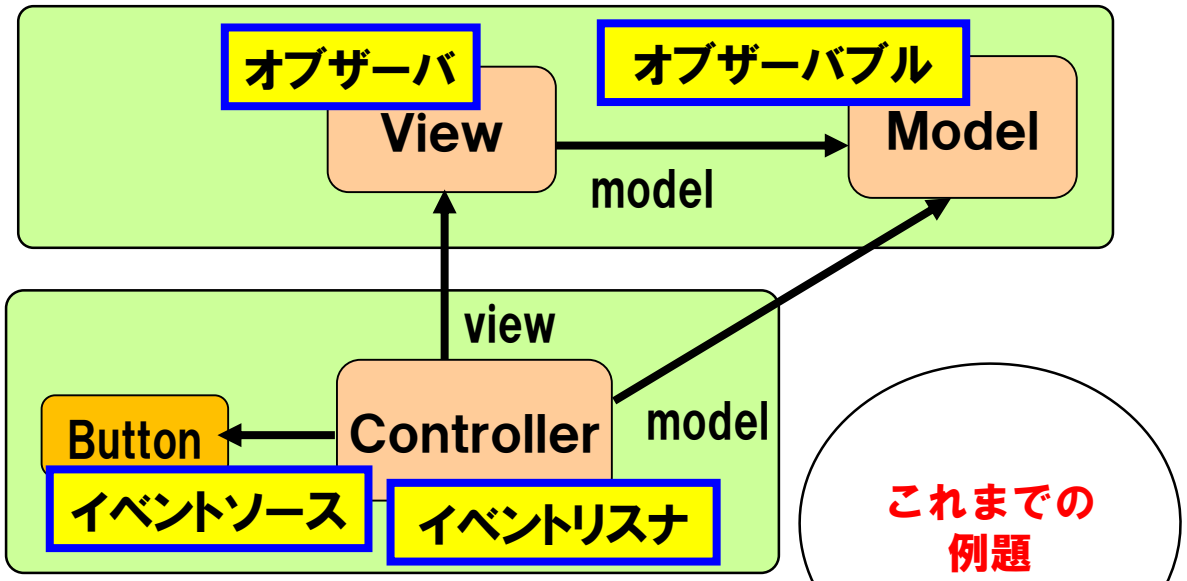
- Observer/Observableパターンとイベント処理：　【ビューが更新される】

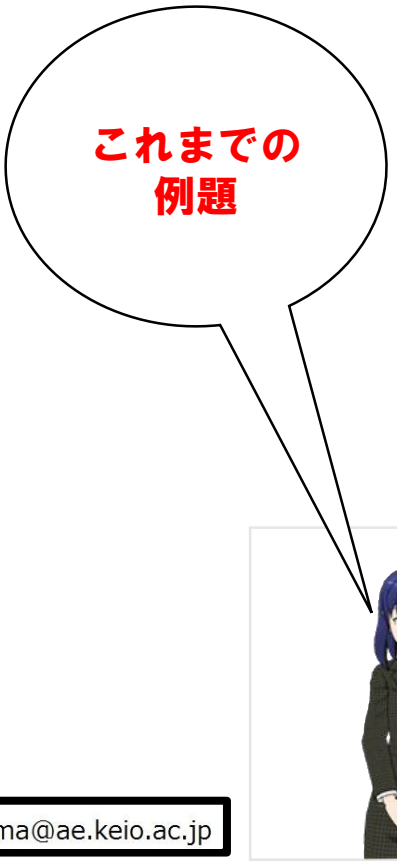


簡単な例題: 動的なモデル

動的モデルとMVC

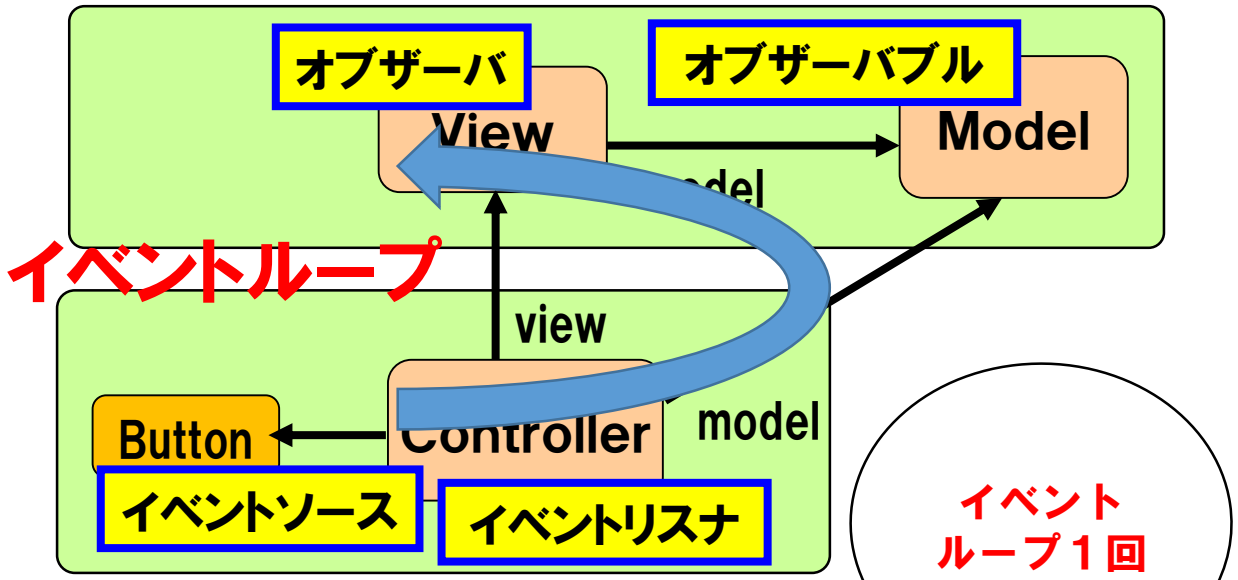
- これまでの例題では…
  - イベントが発生すると，
    モデルが操作され，
    その操作終了後に，
    ビューを更新して，
    またイベントの発生を待つ

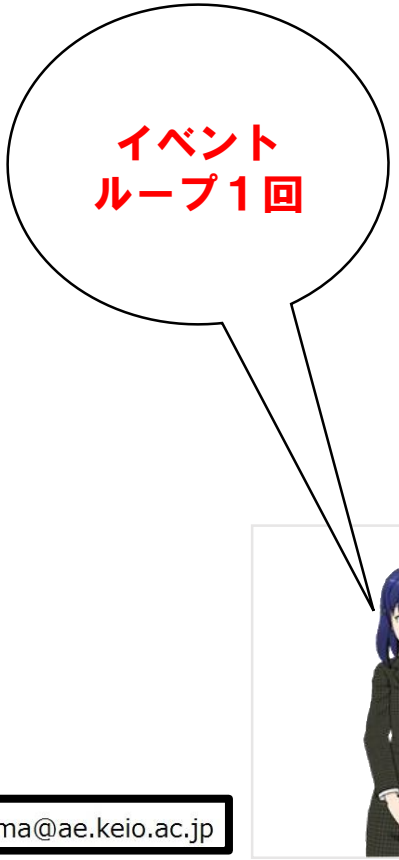




動的モデルとMVC

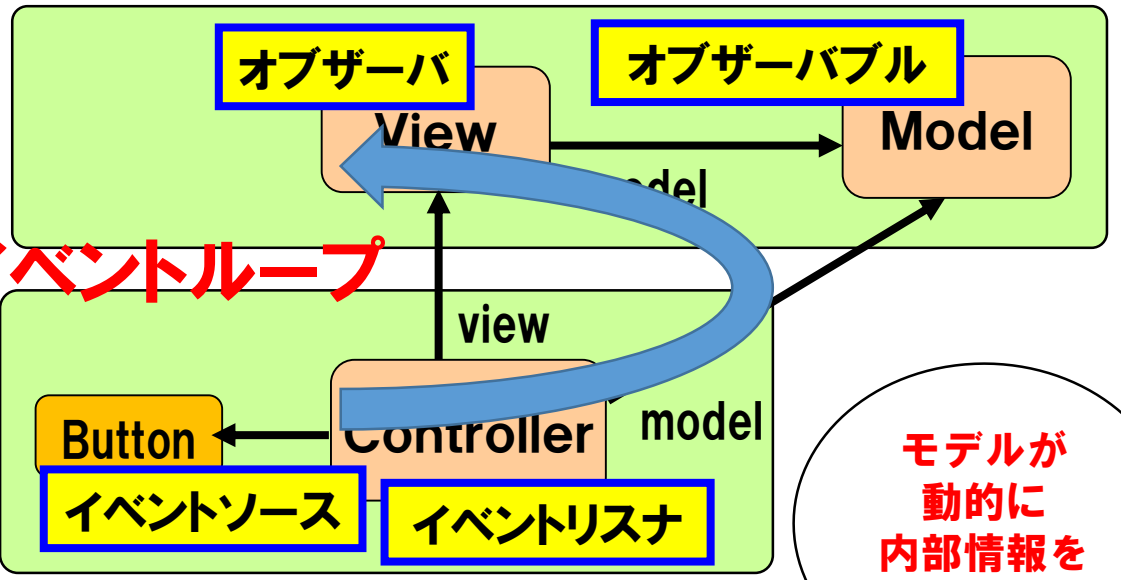
- これまでの例題では…
  - イベントが発生すると，
    モデルが操作され，
    その操作終了後に，
    ビューを更新して，
    またイベントの発生を待つ
  - イベントループ一周するだけ



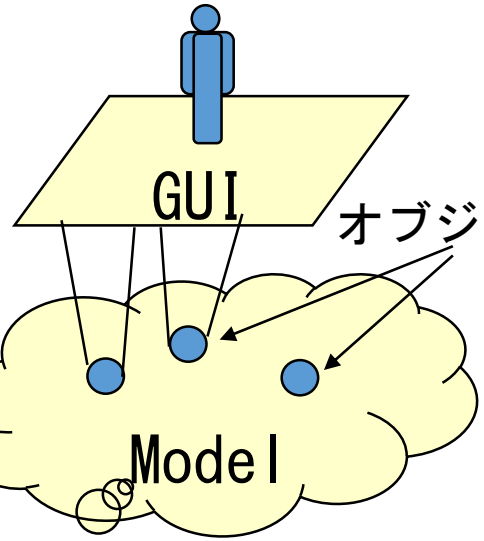


動的モデルとMVC

- これまでの例題では…
  - イベントが発生すると，
    モデルが操作され，
    その操作終了後に，
    ビューを更新して，
    またイベントの発生を待つ
  - イベントループ一周するだけ



- 動的モデル
  - M（Model）が，**時間経過に沿って，動的に自分の内部状態をどんどん変えていく**
  - V（View）は，**モデルの変化に，追従していく**
  - C（Controller）

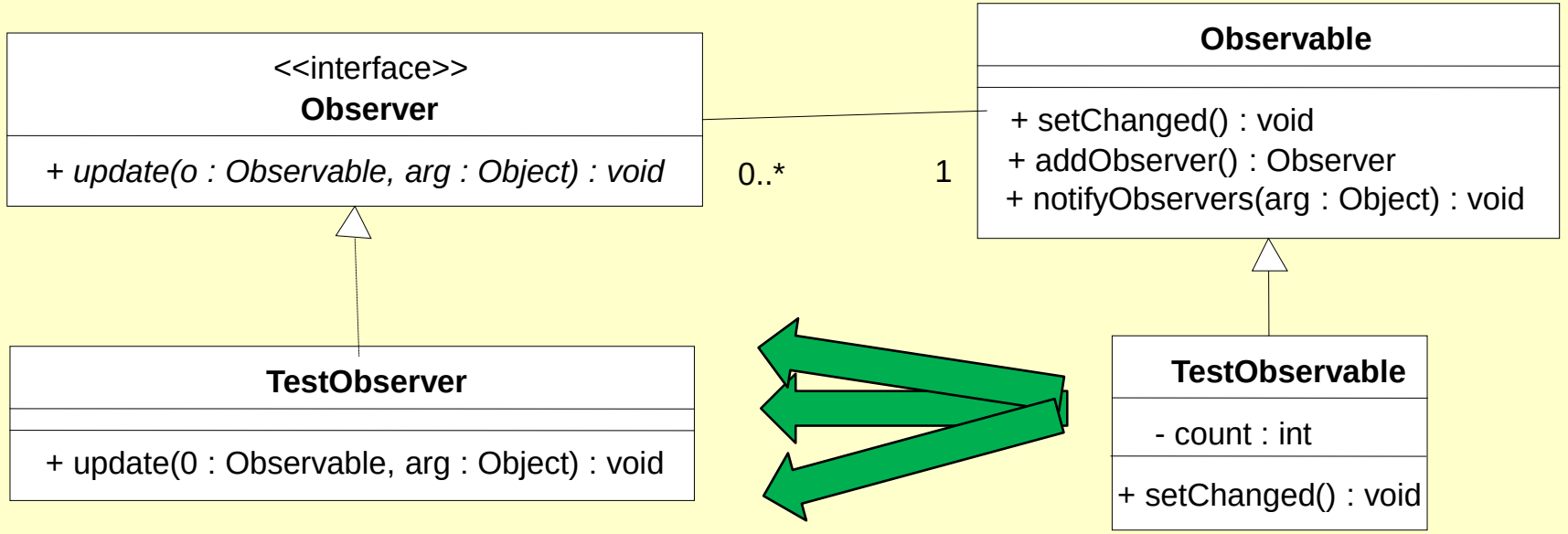


その振舞いを
状態遷移図等で
モデル化…

例題：UMLのクラス図（いずれやります）

監視者（複数可）

監視対象

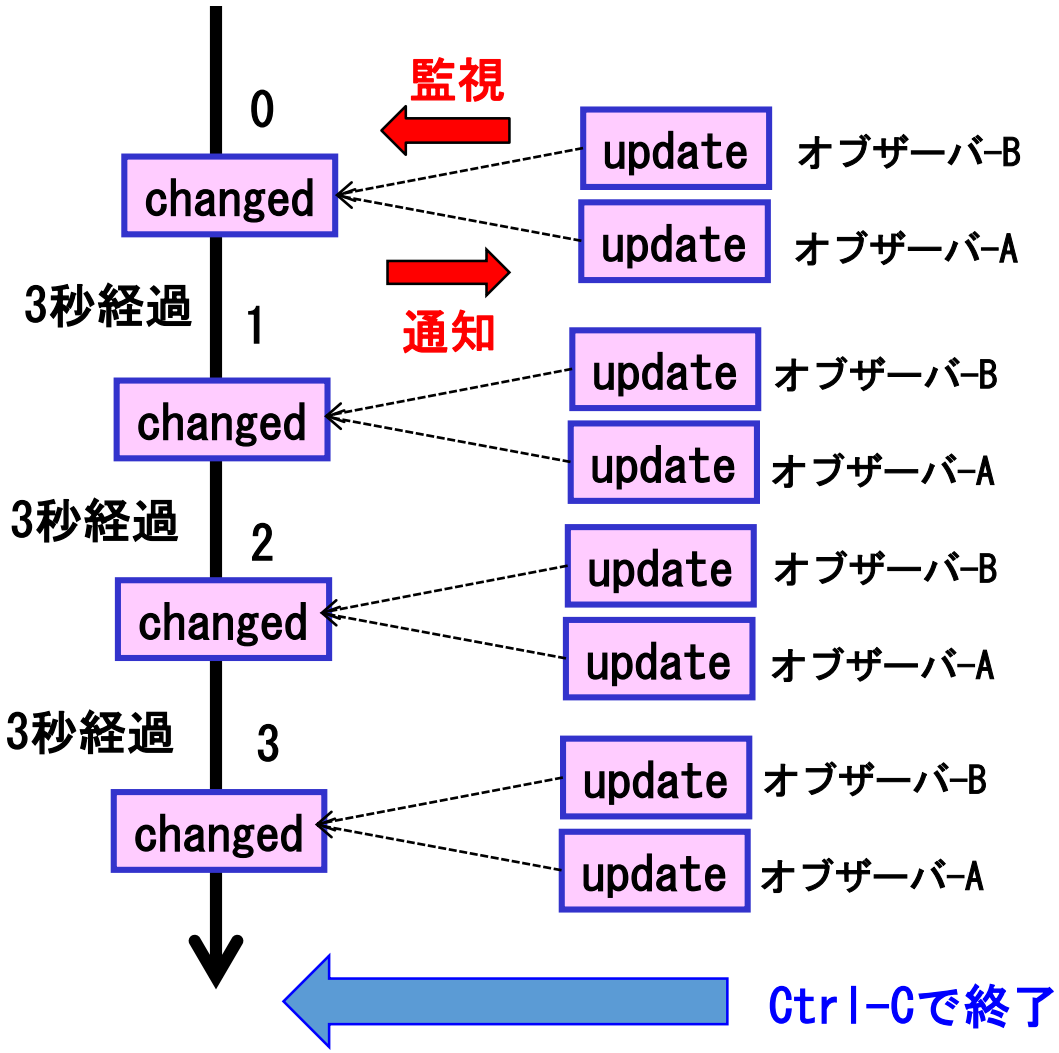


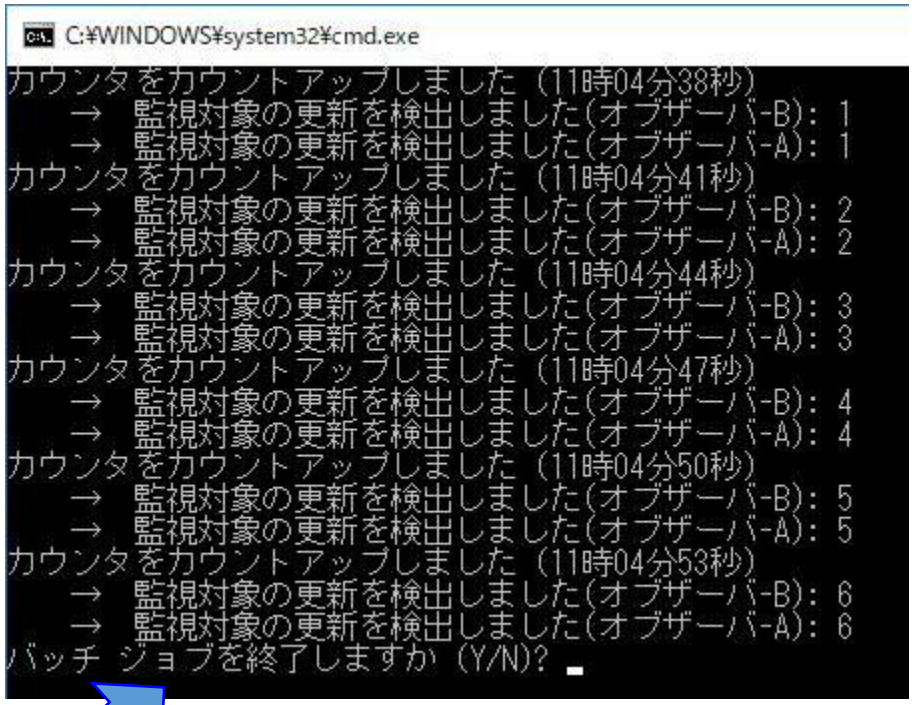
非明示的な
メッセージ送信
↓
イベント通知

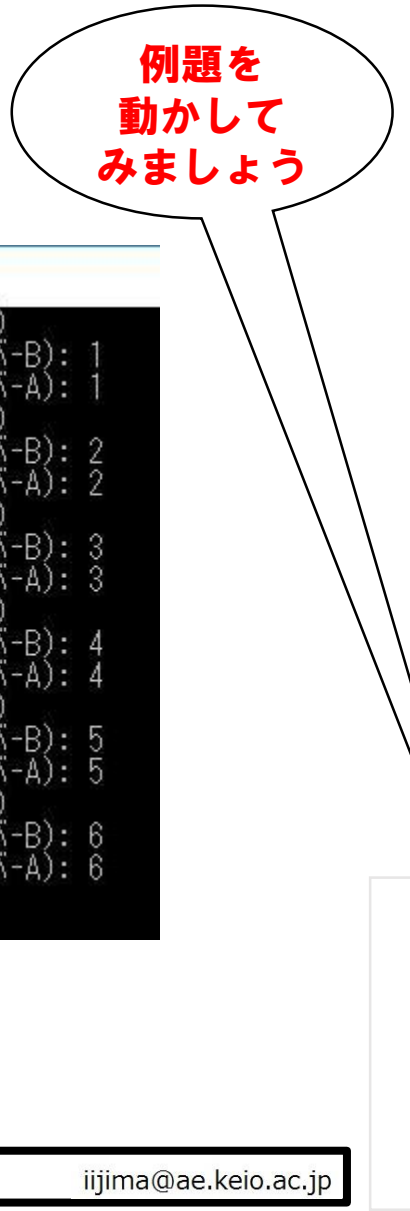
UMLの
クラス図

iijima@ae.keio.ac.jp

3秒に1度内部状態を変化させる動的モデル







Main.java

```
package p06;
public class Main  {
    public static void main ( String [] args )  {
        // ------------------------------------------------
        TestObservable o = new TestObservable ();
        // ------------------------------------------------
        o.addObserver ( new TestObserver ( "オブザーバ-A" ) );
        o.addObserver ( new TestObserver ( "オブザーバ-B" ) );
        // ------------------------------------------------
        o.run ();
        // ------------------------------------------------
    }
}
```

iijima@ae.keio.ac.jp

TestObserver.java: push型Observable

```
package p06;
import java.util.Observer;
import java.util.Observable;
public class TestObserver implements Observer {
private String name = null;
    public TestObserver ( String name ) {
        this.name = name;
    }
    public void update ( Observable o, Object arg ) {
        System.out.println ( "    →   監視対象の更新を検出しました ("
                + name +") : "
                + ( (Integer) arg).toString () ); // 被監視オブジェクトから
                                                  // pushされたカウント値を表示します.
    }
}
```

監視対象から
pushされた
カウント値を
表示します

TestObserver.java: pull型Observer

```
package p06;
import java.util.Observer;
import java.util.Observable;
public class TestObserver implements Observer {
private String name = null;
    public TestObserver ( String name ) {
        this.name = name;
    }
    public void update ( Observable o, Object arg ) {
        System.out.println ( "    →   監視対象の更新を検出しました ("
                + name +") : "
                + ( (TestObservable) o) .getCount () ); // カウント値を被監視オブジェクトから
                                                        // pullして表示する.
    }
}
```

TestObservable.java（1/3）：pull型Observer


```
package p06;
import java.util.Observer;
import java.util.Observable;
import java.util.Calendar;
import java.text.SimpleDateFormat;

public class TestObservable extends Observable {
    private int count = 0;
    private Calendar cal = null;
    private SimpleDateFormat dateFormat = null;

    public TestObservable () {
        count = 0;
        dateFormat = new SimpleDateFormat ("hh時mm分ss秒");
    }
   void changed () {
          setChanged ();
          notifyObservers ();
   }
```

オブザーバが
カウント値を
pullしますので
通知の際に
push
する必要が
ありません.

TestObservable.java（1/3）：push型Observable

```
package p06;
import java.util.Observer;
import java.util.Observable;
import java.util.Calendar;
import java.text.SimpleDateFormat;

public class TestObservable extends Observable {
    private int count = 0;
    private Calendar cal = null;
    private SimpleDateFormat dateFormat = null;

    public TestObservable () {
        count = 0;
        dateFormat = new SimpleDateFormat ("hh時mm分ss秒");
    }
    void changed () {
        setChanged ();
        notifyObservers ( new Integer ( count ) );
    }
```

オブザーバへ
カウント値を
プッシュします

TestObservable.java（2/3）

```
void run () {
    count = 0;
    while ( true ) {
        count++;      // カウントアップ
        cal = Calendar.getInstance ();
        System.out.printf ( "カウンタをカウントアップしました (%s) ¥n",
                            dateFormat.format (cal.getTime () ) );
        changed ();    // イベント通知
        try {
            Thread.sleep ( 3000 ); // 3秒休止
        } catch ( InterruptedException ex ) {
            System.exit ( 1 );
        }
    }
}
```

3秒ごとにカウントアップを繰り返す

Ctrl-Cで終了

3秒に一度，
カウントアップします．
マルチスレッドという
仕掛けを使っています．

iijima@ae.keio.ac.jp

TestObservable.java（3/3）


```
    public int getCount()  {
        return ( count );
    }
}
```