オブジェクト指向開発論

2020年6月25日

海谷 治彦

目次

- モデリングの視点 復習
- クラス図
  - インタフェースに関する補足も含む
- シーケンス図
- 実例

復習

モデリングの代表的な側面

- 構造的側面
  - 現実世界のどんなモノが当面コンピュータで行いたいことに関係するか，それらの(静的な)関係は何かを明確にする.
  - 通常，クラス図を利用.
- 機能的側面
  - 現実世界の事象(コンピュータへの入力)に対して，コンピュータは何を起こすかを明確にする.
  - 通常，ユースケースモデルを利用.
- 振る舞い的側面
  - 機能の実行順序をモデル化.
  - 通常，ステートマシン図，アクティビティ図，シーケンス図を利用.

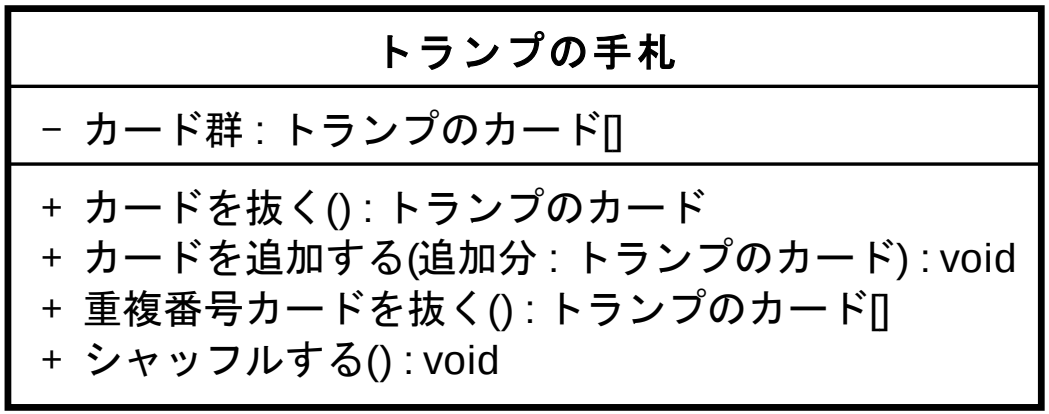
クラスとは？

- オブジェクト指向のプログラムでの基本単位.
- 形式的には，データ(属性)群と，それに対する操作(メソッド，関数)群のセットとなっている.
- 意味的には，特定の役割を担うモノ(名詞)をあらわしている.
- 役割(一般)を表記しているものであり，個々の具体的な事例を示しているわけではない.
  - クラスとインスタンスの違い

- JavaやC#等の場合，モデリングでのクラスが，ほぼそのまま，プログラムのクラスとなる.

クラス図の表記

- 既に出てきているが，箱を三分割した表記をとる.
- 上の区画がクラスの名前，中が属性群，下がメソッド群となる.



クラス図

- クラス間の関連を付記したクラス群の図.
- 関連を線で書く.
- 線でつながっているクラス間で，メソッドの呼び出しを行うことができる.
  - 自身への関連は特別な場合以外はかかない.
  - 方向性を書くこともある，その場合，メソッド呼び出しは一方的.

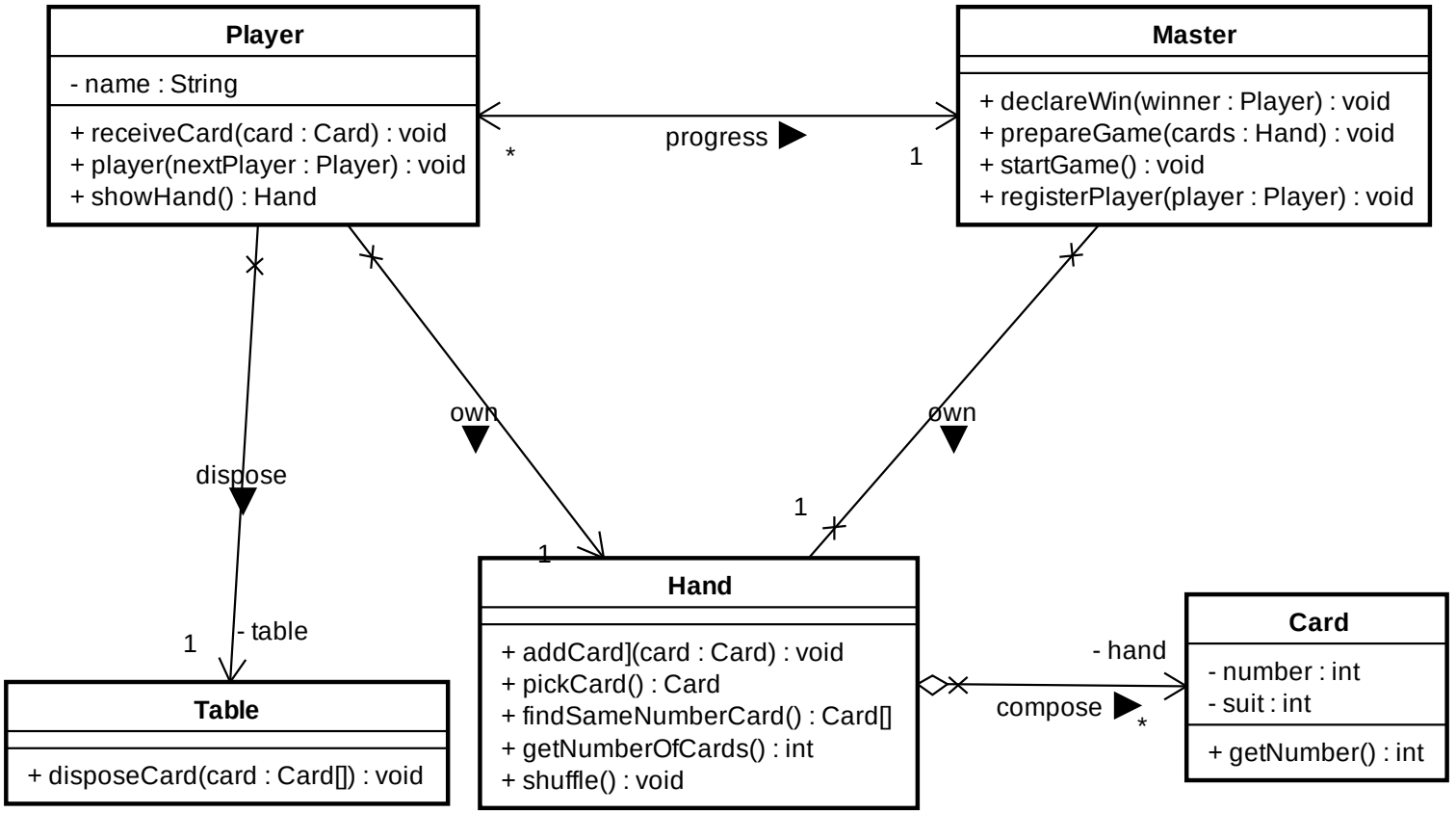
メッセージ vs メソッド呼び出し

- オブジェクト指向は，本来，オブジェクト(クラス)間で，メッセージを送りあうことでオブジェクトが情報交換をするという考え方だった.
  - Smalltalk, objective-C, Ruby にはメッセージがあるらしい.
- JavaやC#等のオブジェクト指向言語にはメッセージという概念は無い.
  - 基本，C言語から進化したからだと思う.
- そこで，Java等では，メッセージの代わりに，
  - メッセージを送る先のオブジェクトのメソッドを呼び出す

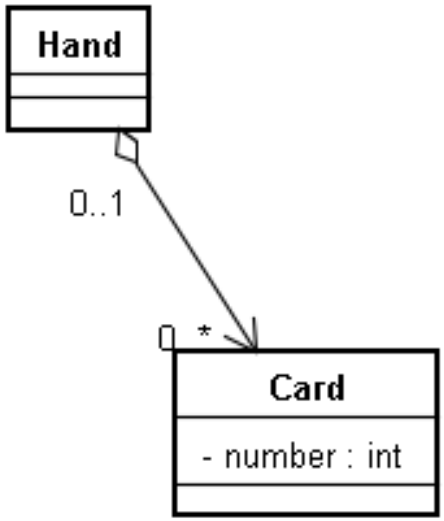
  という形で，情報交換を実現している.

再掲載

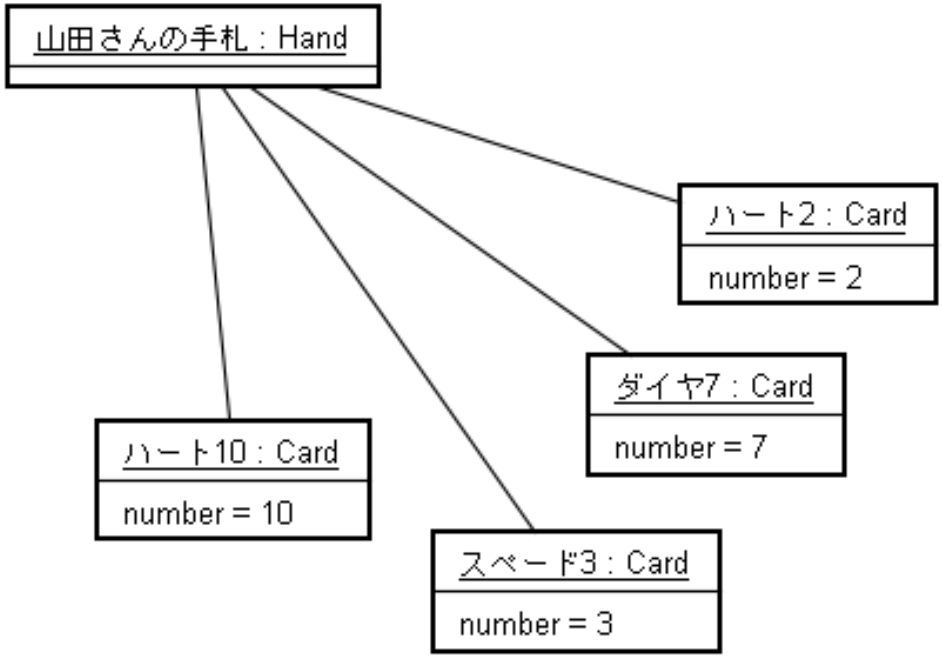
クラス図の例


Player
- name : String
+ receiveCard(card : Card) : void
+ player(nextPlayer : Player) : void
+ showHand() : Hand
Master
+ declareWin(winner : Player) : void
+ prepareGame(cards : Hand) : void
+ startGame() : void
+ registerPlayer(player : Player) : void
progress
*
1
own
own
dispose
1
1
1
- table
Table
+ disposeCard(card : Card[]) : void
Hand
+ addCard](card : Card) : void
+ pickCard() : Card
+ findSameNumberCard() : Card[]
+ getNumberOfCards() : int
+ shuffle() : void
- hand
compose
*
Card
- number : int
- suit : int
+ getNumber() : int

クラス図 vs オブジェクトのイメージ

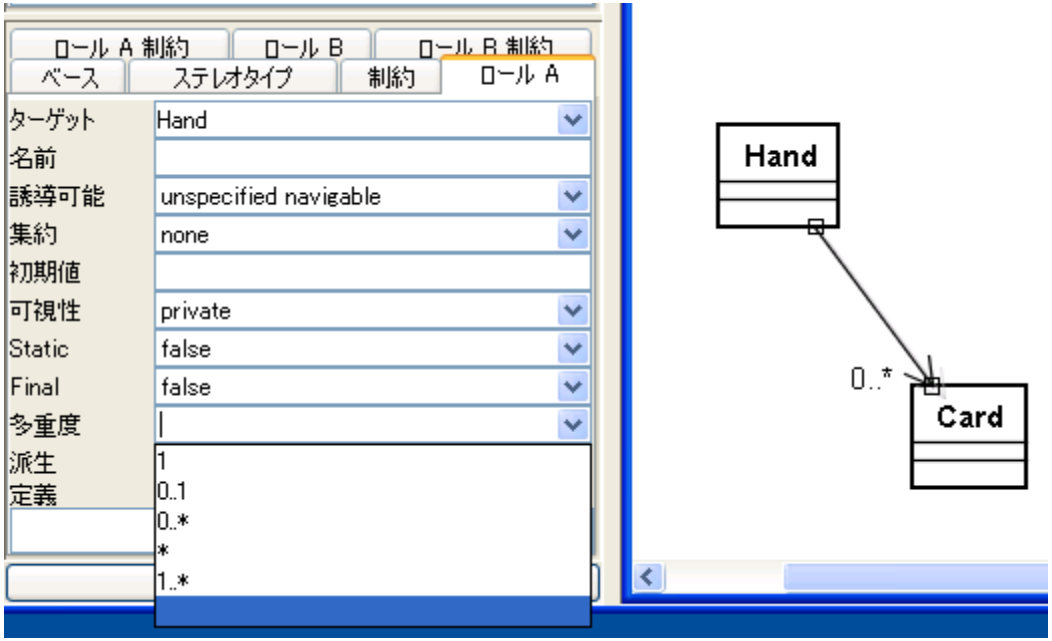


クラス図
(クラス・レベル)



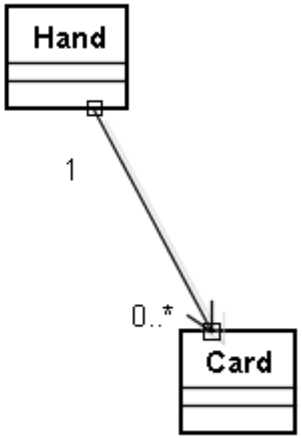
オブジェクト図
(インスタンス・レベル)

多重度



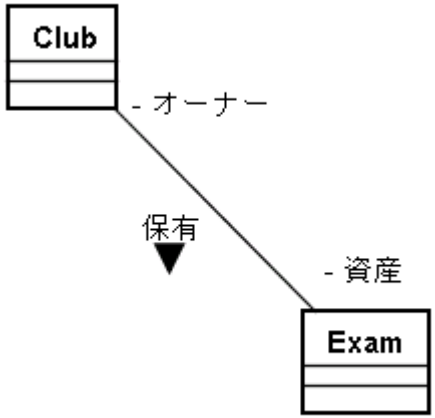
- 1対1，多対一等の対応を示す．
- 1が一
- *はゼロ以上

誘導可能性 navigable



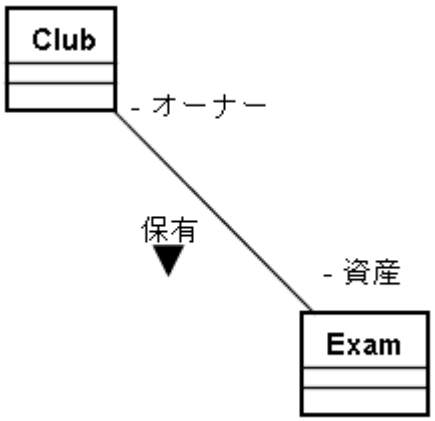
- 関連に方向性がある場合，→頭を書く.
- 尾から頭に向けて参照できる.
- 尾が頭の参照を持っている.

関連名



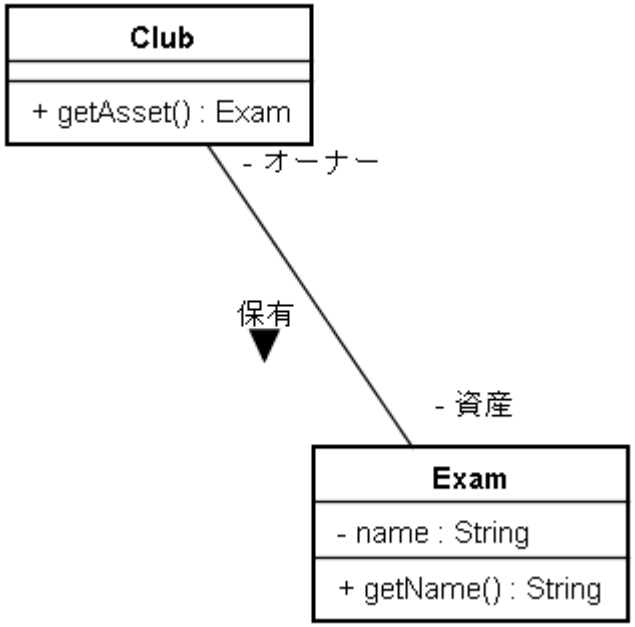
- 関連の意味に名前をつける.
- 後述の関連端名(ロール名)と区別するために▲をつける.

関連端名 (ロール名)



- 名がついてない方から見て，名がついてる方にどうかかわるかを記述.

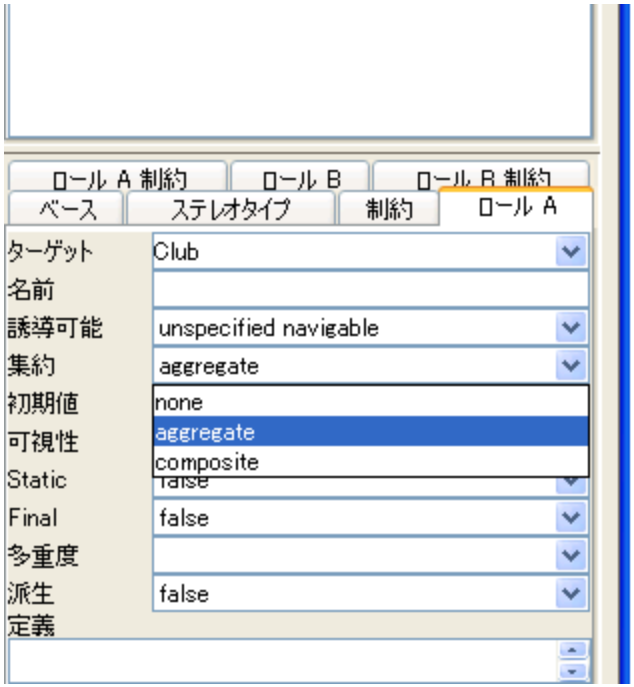
+と-

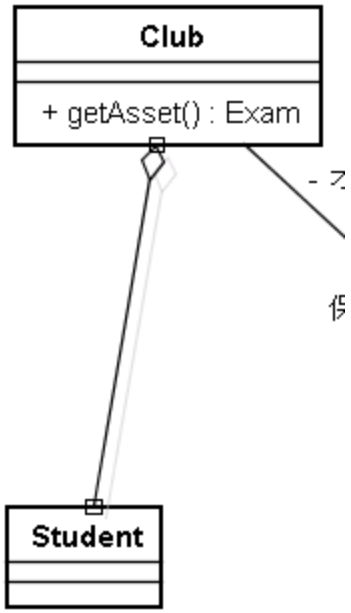


- +はpublic
- -はprivate

集約 Aggregation

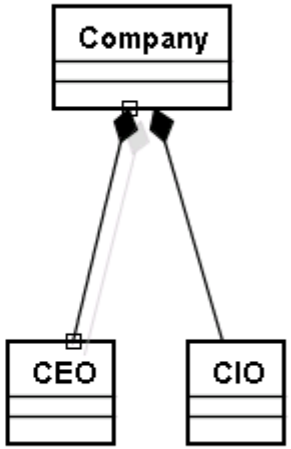
- 部分-全体関係
- 全体が消えても部分は消える必要はない.



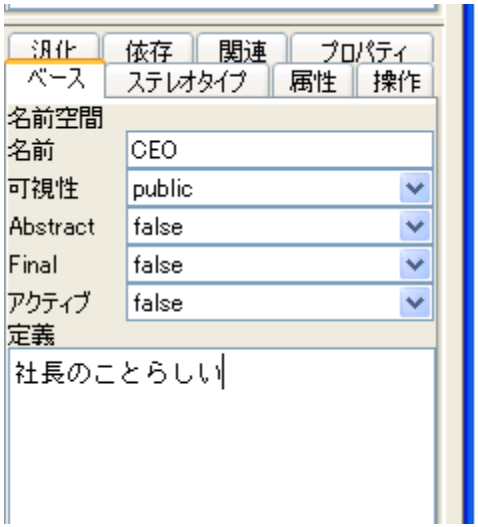


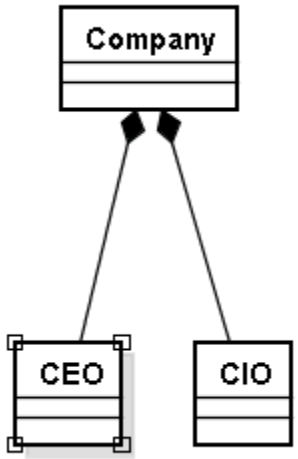
コンポジション Composition

- 全体が消えれば部分も消える関係.

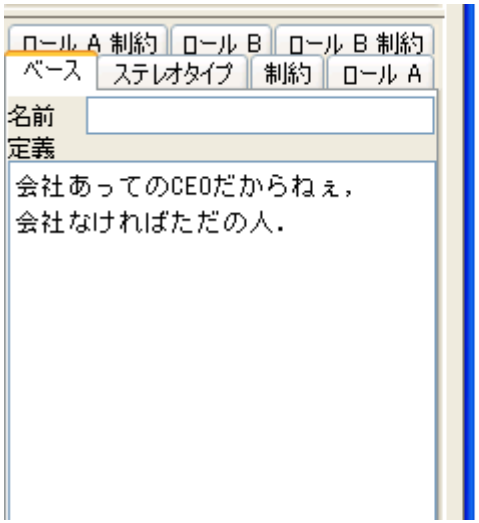


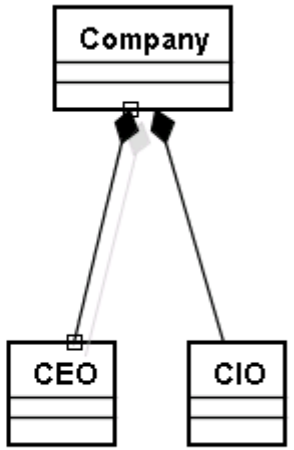
定義説明



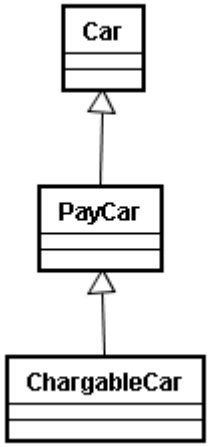


- クラスや関連には説明書きが書ける.
- 意味不明にならないように書いておこう.
- Javaソースを生成するとコメント文に入る.





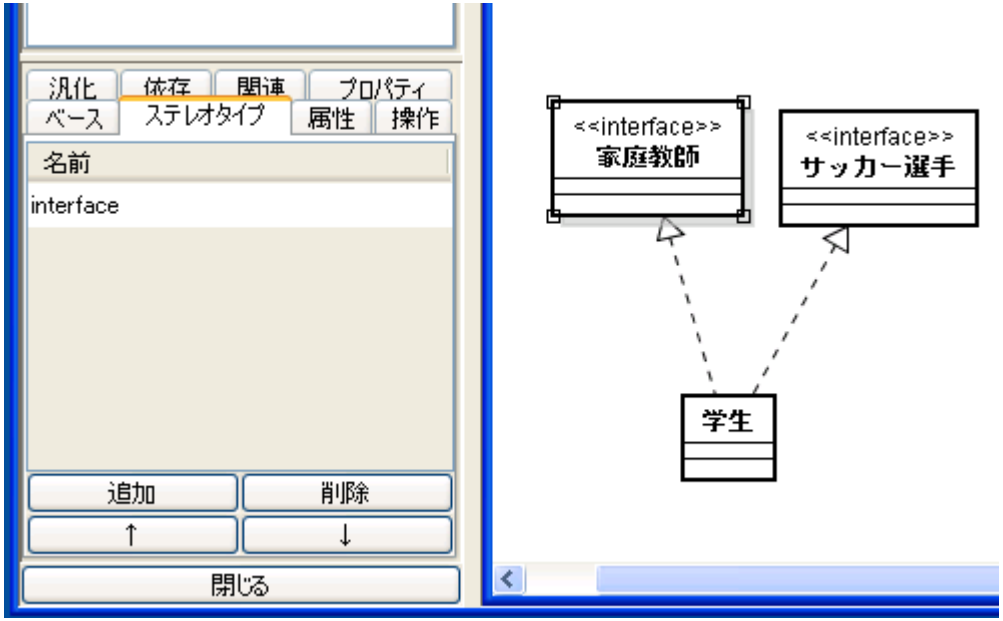
汎化・継承



- △で書きます.
- Javaでいうところのextends

インタフェースの実装

- △と点線で書く.
- インタフェース側は<<interface>>というステレオタイプを付けないといけない.
- ステレオタイプ: クラス等の種類の分類タグと思えばよい.

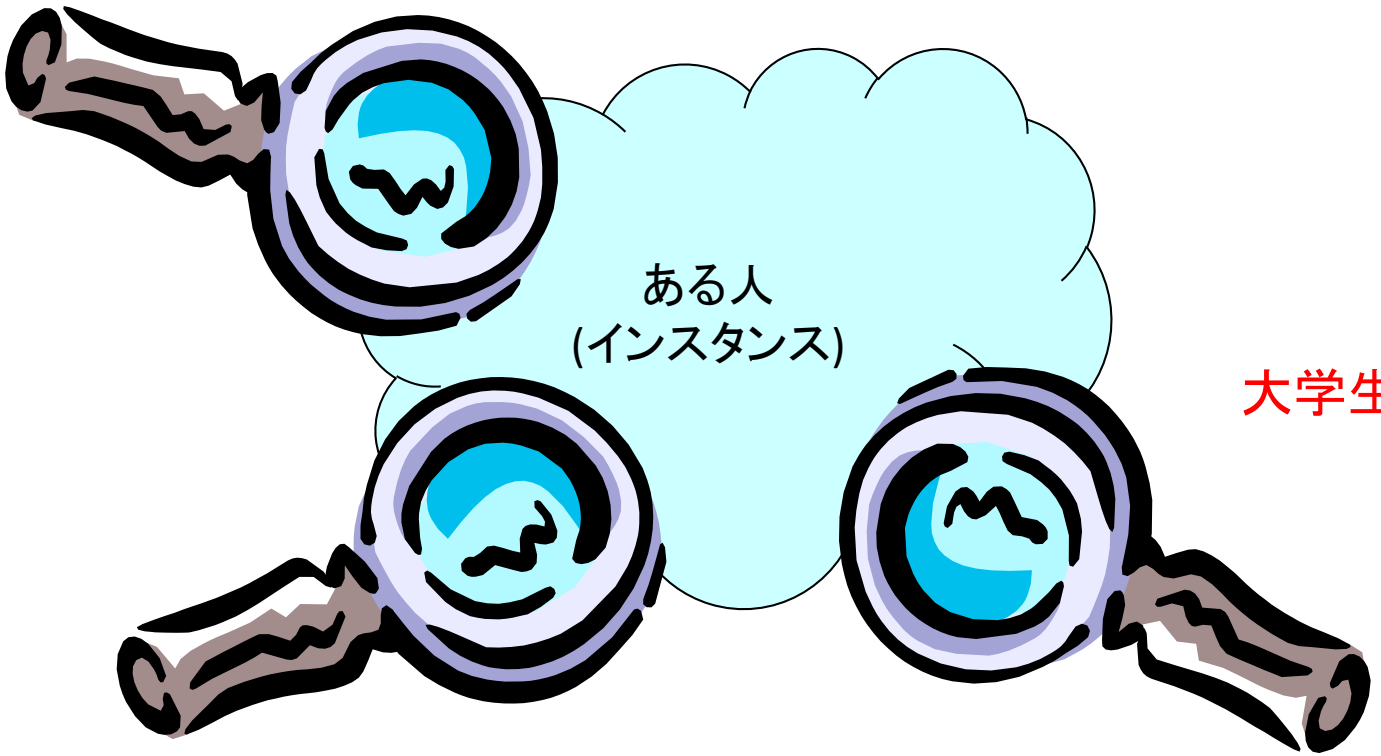


Interface

- あるクラスの多面性(ま，別に一面でもいいけど)を明示的に記述したもの.
- あるクラスでinterfaceを実装することで，そのクラスはinterfaceで定義されたメソッドを提供することを保障する.
- interfaceの意味通り，(interfaceを実装した)クラスを使う(他の)クラスのための接点を提供するもの.

例 (というか比喩)

サッカー選手であって,
サッカーする.



大学生であって,
勉強する.
実験する.

家庭教師であって,
教える.

interfaceの記述

```
public interface FootballPlayer{
  public void select(Game g);
}
```

```
public interface Student{
  public void teach(Field f);
}
```

```
public interface PrivateTeacher{
  public void learn();
  public void invite();
  public void emply(Pupil p);
}
```

interfaceの実装

```
class Gakusei implements FootballPlayer, Student, PrivateTeacher{
  public void select(Game g){
      // 実際あるゲームgへの参加選抜をされる際の処理をかく
  }
  public void teach(Field f){
     // 実際にある分野fを教わる場合の処理を書く
  }
  public void employ(Pupil p){
    // 実際にある生徒pに雇われる際の処理を書く
  }
  public void learn(){
    // 実際に雇われている生徒が学ぶ際の処理を書く
  }
}
```

interfaceの効用

- (前述のように)あるクラスの持っている側面に明示的に名をつけて区別することができる.
- クラスを使う側からすれば，クラスではなくインタフェースを指定することで，クラス間の関連を低くすることができる.

  ⇒ あるインタフェースを実装したクラスなら，なんでも使えるような汎用性の高いクラスを書ける.

Interfaceで指定

家庭教師を雇う方にすれば，その機能を提供するものなら誰でも良い(はず).

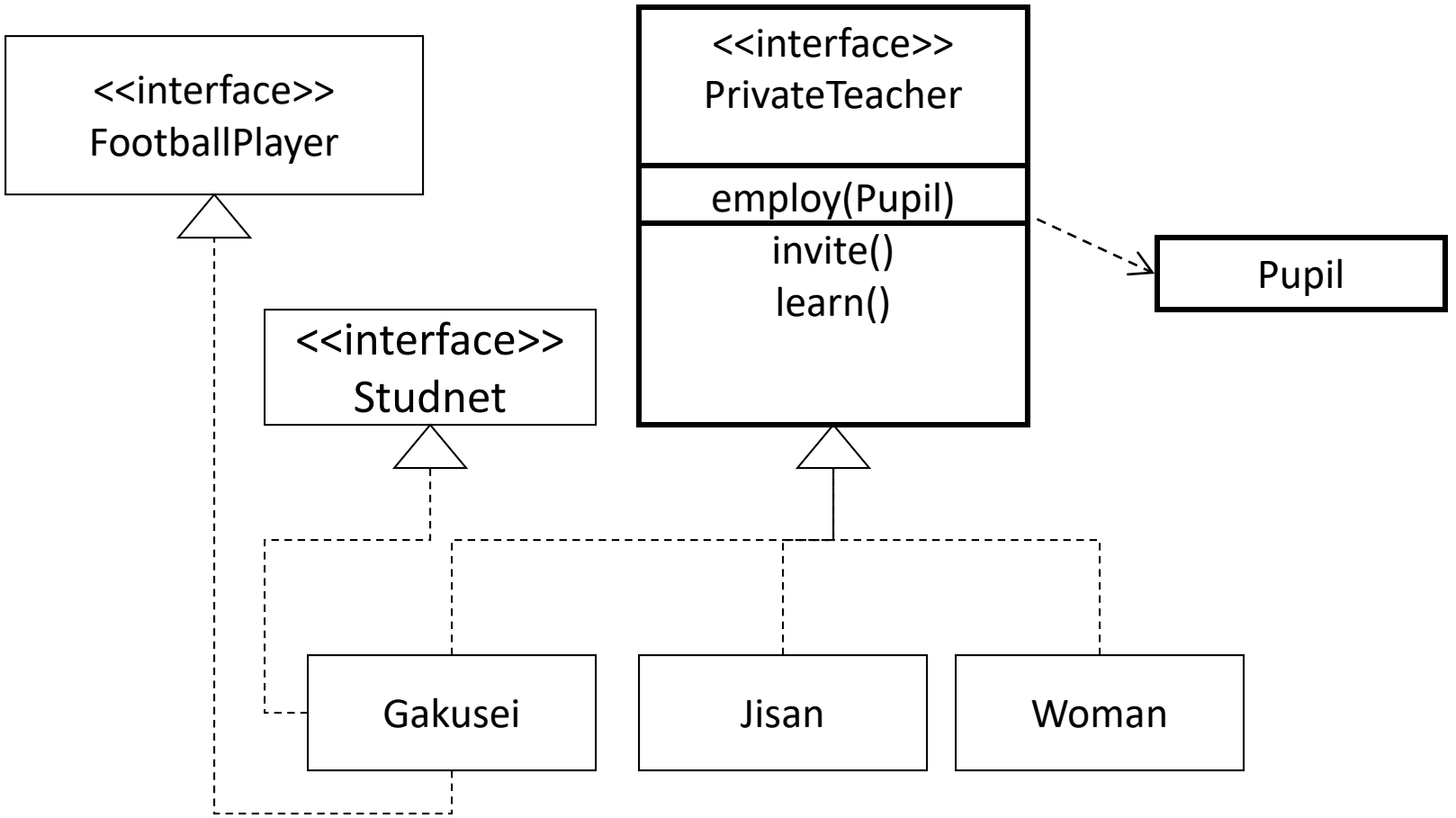
```
class Pupil {
  .... educate(PrivateTeacher pt){
      pt.employ(this);
      pt.invite();
      pt.lean();
  }
}
```

```
class Woman implements
  PrivateTeacher, HouseWife{
   ...
}
```

```
class Jisan implements
    PrivateTeacher, Niwashi{
   ...
}
```

```
class Gakusei implements
   FootballPlayer, Student, PrivateTeacher{ ...
}
```

クラス図で書いてみると



現実的なinterface

- Javaの標準API (Application Programming Interface, 標準的に利用できるクラスライブラリのこと)には，多数のinterfaceと多数のinterfaceを実装したクラスがある.

Interface Serializable

- java.io パッケージ内に定義
- このinterfaceが実装されているクラスのインスタンスは，ファイルにしまったり，ネットワーク上にデータとして転送できたりする.
  - 逆にこれが実装されていないクラスのインスタンスはファイル保存等ができない.
- String, Vector, Integer等，データ指向のクラスでは大抵実装されている.

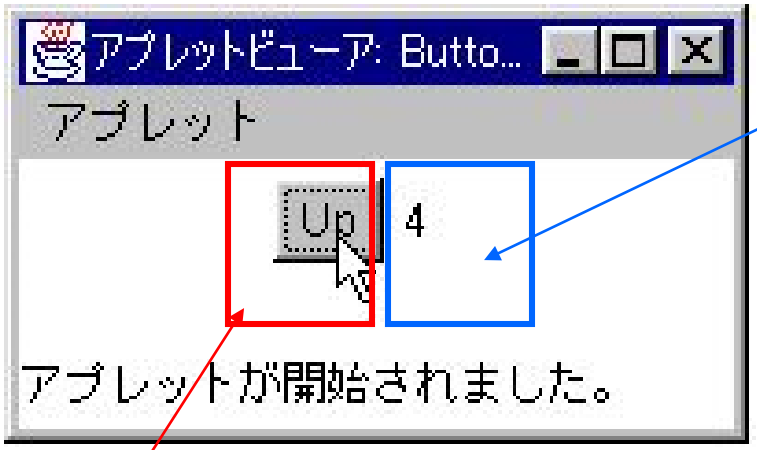
interface Runnable

- run + able すなわち「実行可能」を示すinterface.
- スレッド(プログラム内の並行処理の1つ)を実現するためには，このインタフェースに，処理ループを書くのが普通.
- public void run() メソッドの実装を指示.

MouseListener

- GUIにおいてマウスの動作に伴い発生するデータ(event)を拾い，それに反応するためのクラスは大抵，コレを実装している.
- ボタン等は，通常，特定のMouseListenerを実装したクラスが結びついているが，その結びつきを変えることで，簡単にボタンを押した際の振る舞いを変えることができる.

例: リスナを使ったイベント駆動



イベントリスナ:
発生したイベントに対応してある処理をする部品

イベントソース:
イベントを発生する部品

この例では，ボタンを押すとラベルの数値が増える，という単純なもの．

例: ソースコード

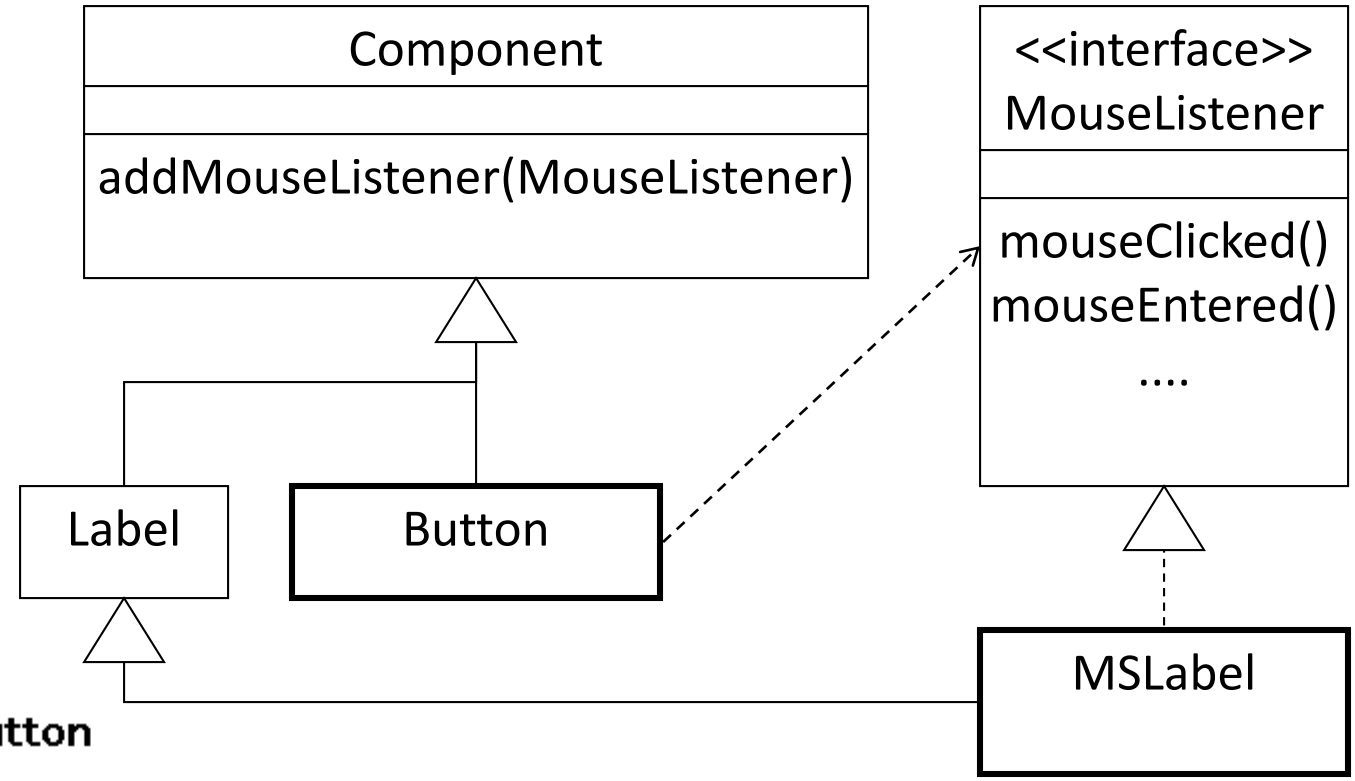
```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class ButtonListen extends Applet{

        public void init(){
         Button b=new Button("Up");
         MSLabel ms=new MSLabel(0);
         b.addMouseListener(ms);
         this.add(b);
         this.add(ms);
        }
}
```

```
class MSLabel extends Label implements MouseListener{
private int;
        MSLabel(int initn){
                n=initn;
                setText(n+"");
        }
        public void mouseClicked(MouseEvent e){
                n++;
                setText(n+"");
        }
        public void mouseEntered(MouseEvent e){}
        public void mouseExited(MouseEvent e){}
        public void mousePressed(MouseEvent e){}
        public void mouseReleased(MouseEvent e){}
}
```

ボタンbのイベントを
ラベルmsが聞くように指示

ボタン系のイベントに対応して行う処理を，
ラベル(リスナー)内に実装．

クラス図



java.awt
クラス Button

```
java.lang.Object
  |
  +--java.awt.Component
        |
        +--java.awt.Button
```

すべての実装インタフェース:
Accessible, ImageObserver, MenuContainer, Serializable

InterfaceのTIPS

- ClassよりむしろInterfaceのほうが役割という意味に近い.
- 「視点」と考えてもよい.
- インスタンスは実装されているInterfaceで参照できる.
  - Interfaceで参照するとアクセス可能なメソッドは減る可能性がある.
- Classを使うことを想定せず，Interfaceを使うことを想定したクラスのほうが柔軟.

インタフェースは振舞は規定しない

- 「オブジェクトに対する操作方法と，それに対応する振る舞い」を規定と解説する本もある.

- 操作法は規定している.
- 振る舞いは規定されているとは言えない.
  - メソッドの名前から直感的な振る舞いはわかる.
  - しかし，そのメソッドが直感的な振る舞い通り実装されているかをインタフェースは保障できない.
- 詳細は反例 Calcable にて

反例 Calcable

```
/**
 計算可能な者とみなせるものが
持つべき機能を規定
 */

public interface Calcable{
 /** 足し算 */
 public Calcable add(Calcable b);
 /** 引き算 */
 public Calcable sub(Calcable b);
 /** 値を文字列で返す */
 public String value();
}
```

```
/**  おかしな振る舞いの整数  */

public class FunnyInt implements Calcable {
 private int v=0;

 public FunnyInt(int x){ v=x; }

 /** 名前に反して引き算結果を返す */
 public Calcable add (Calcable b){
  int y=Integer.parseInt(b.value());

  return new FunnyInt( v-y );
 }

 /** 名前に反して足し算結果を返す */
  public Calcable sub (Calcable b){
   int y=Integer.parseInt(b.value());

   return new FunnyInt( v+y) ;
 }
以下，略
```

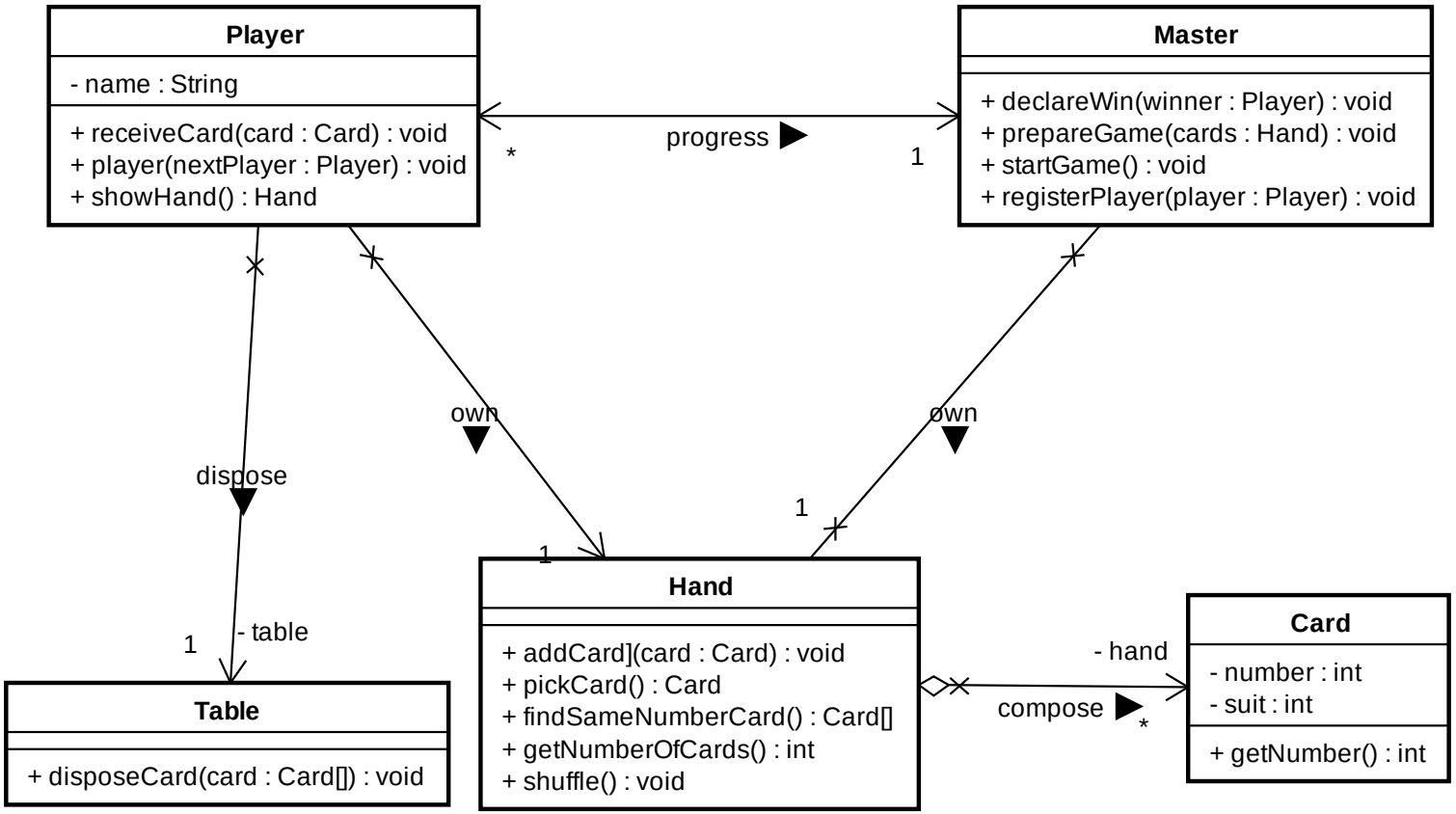
TIPS

- 多重度，Nabigability, Aggregation, Composition 等は面倒なら記載しなくてもよい.
- 重要なのはクラスとその関連をちゃんとつけること.
- クラスや関連の名前は意味にあったものを選ぶこと.

具体例:「ばば抜き」のモデリング

- 参考図書「なぜJava」からの引用.
- プレーヤーの人数
  - プレイヤーは2人以上とする.
- ゲームの準備
  - 進行役は，ジョーカー1枚を含む53枚のトランプをよくシャッフルし，参加するすべてのプレイヤーに等しく配る.
  - プレイヤーは配られたカードを手札に加える．このとき，手札の中に同じ数の組み合わせがある場合，その組み合わせのカードをテーブルに捨てることができる.
- ゲームの開始
  - 進行役はプレイヤーを順に指名する．指名されたプレイヤーは隣のプレイヤーの手札から任意の1枚を引き，自分の手札へ加える．このとき，手札の中に同じ数の組み合わせがある場合，その組み合わせのカードをテーブルに捨てることができる.
  - これを繰り返し，手札をすべてなくしたプレイヤーが上がりとなる．最終的にジョーカーを残したプレイヤーが負けとなる.

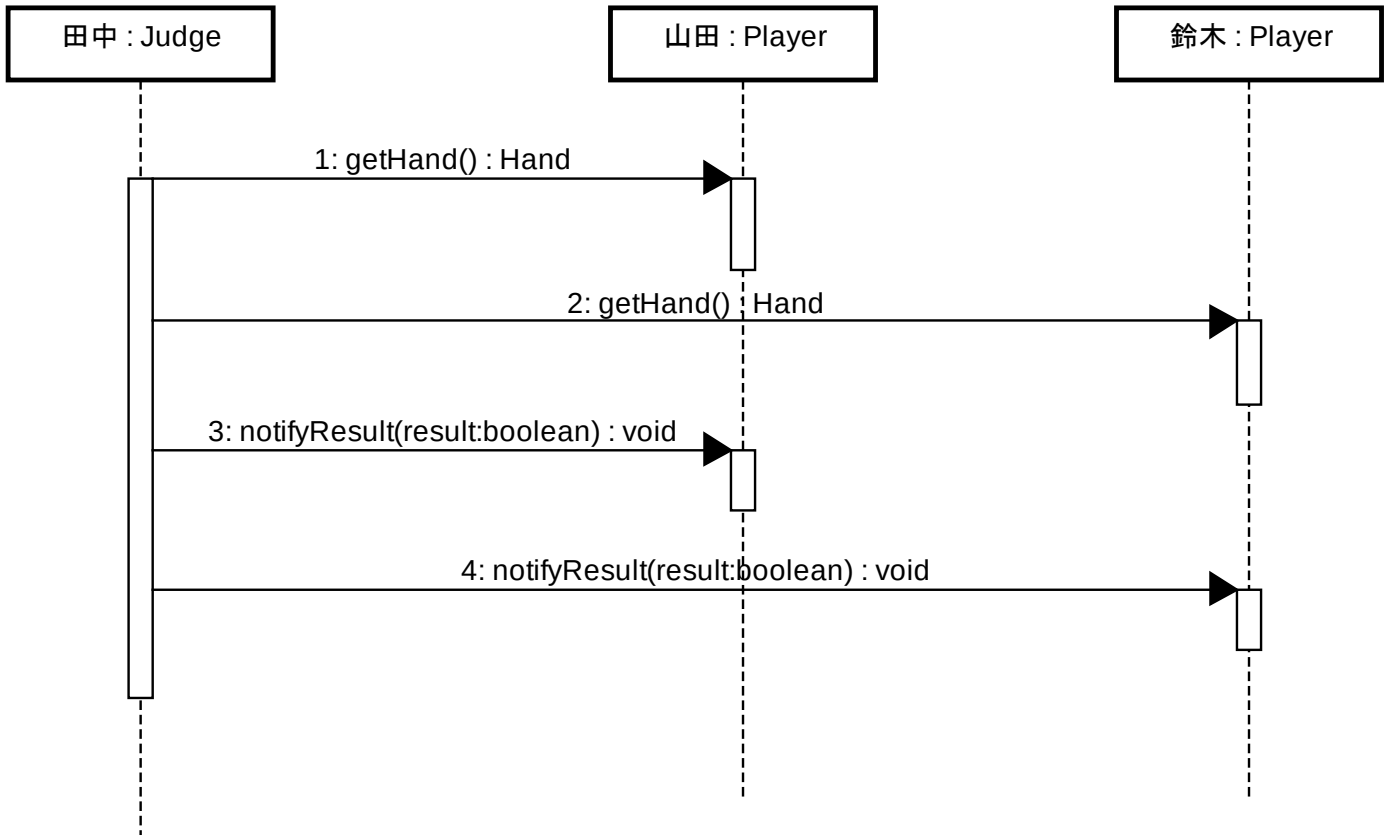
クラス図 oo07_6d.asta



シーケンス図とは？

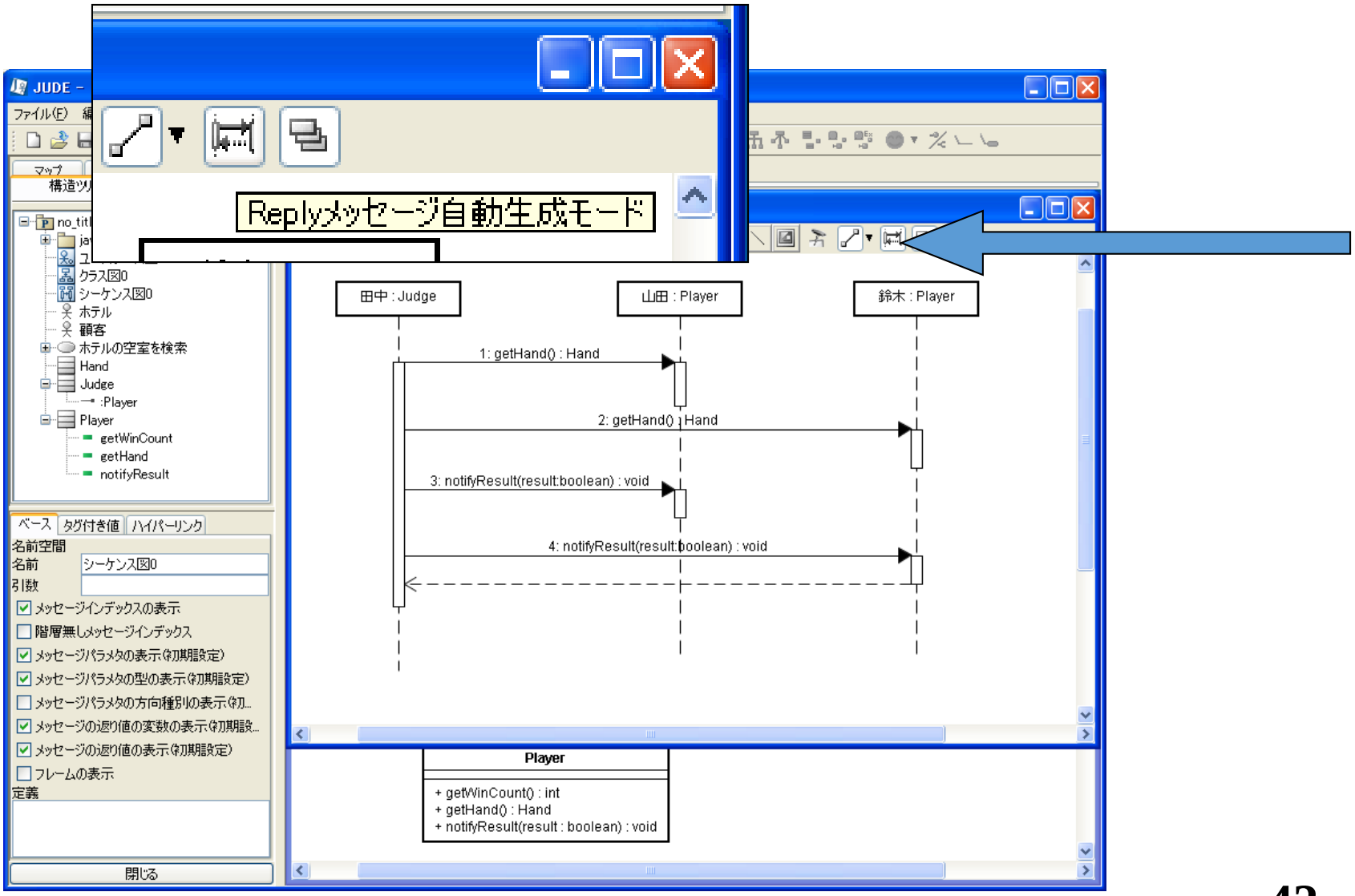
- 実際にクラス間のメソッド呼び出し関係の例を書いた図.
- あるシーケンスの例に過ぎず，個々のシーケンス図は，全ての流れを網羅してはいないし，網羅するのはよくない.
- 次回，話すがシーケンス図を書きながら，
  - クラス間の関連の有無
  - クラスの持つべきメソッド
  - を決めてゆくのが普通.
- 最終的には，クラス図をもとに，あらゆるメソッドの呼び出し関係を網羅できないといけない.
  - そーじゃないと，漏れのあるプログラムとなってしまう.

シーケンス図 じゃんけんの例

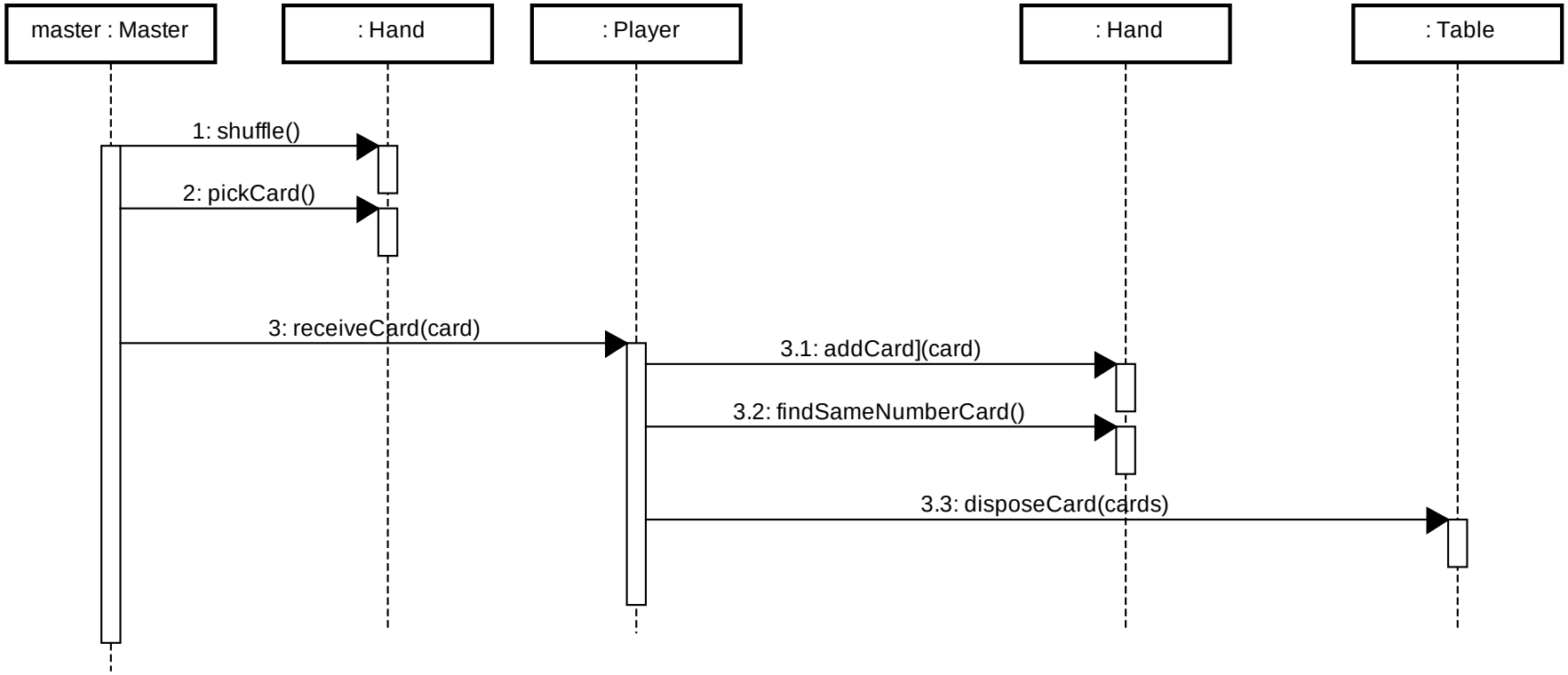


何故Javaにはreplyメッセージ(逆方向の点線矢印)があるが，省略してよい．

replyメッセージを書きたい場合

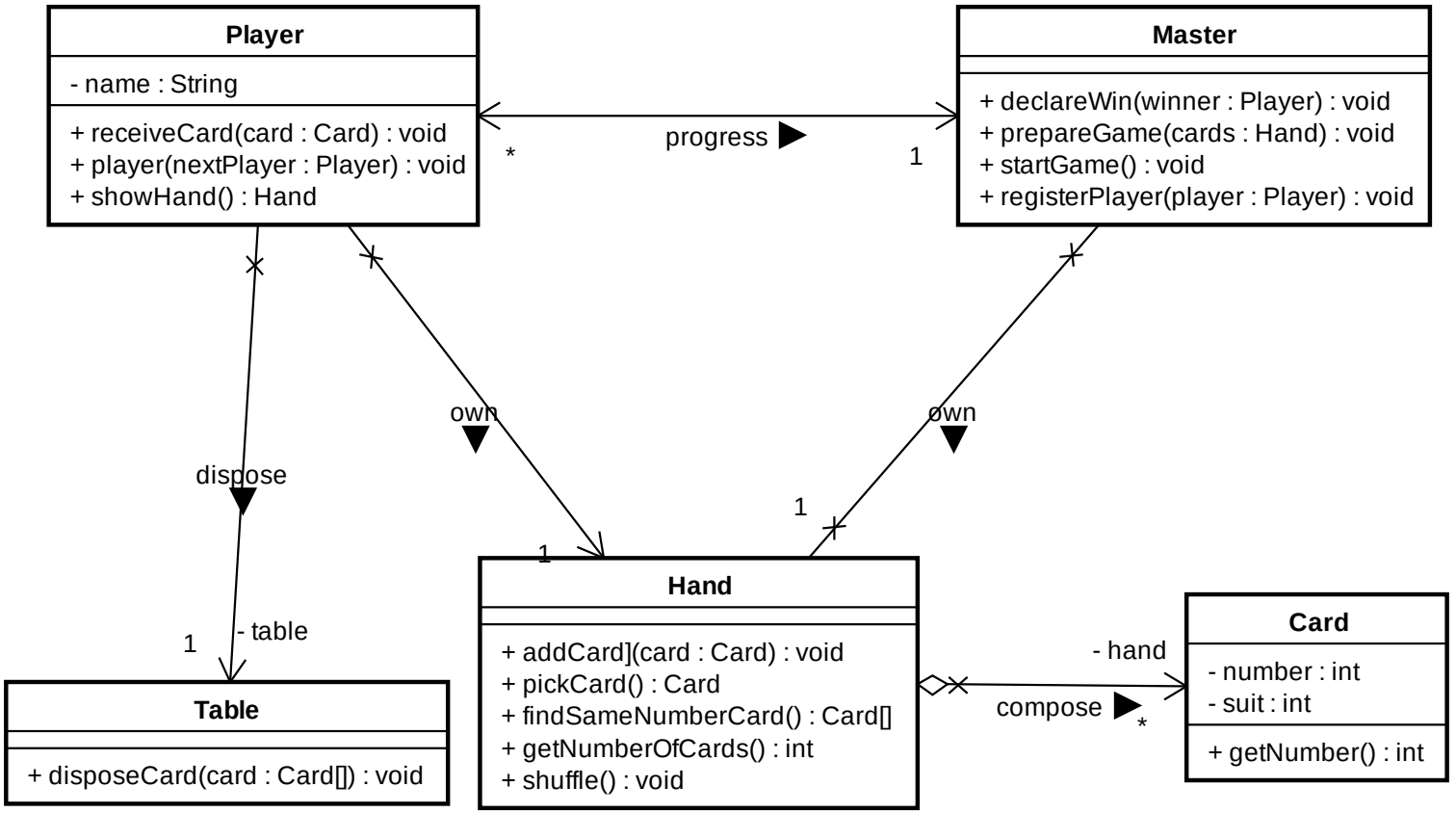


トランプを配るシーケンス



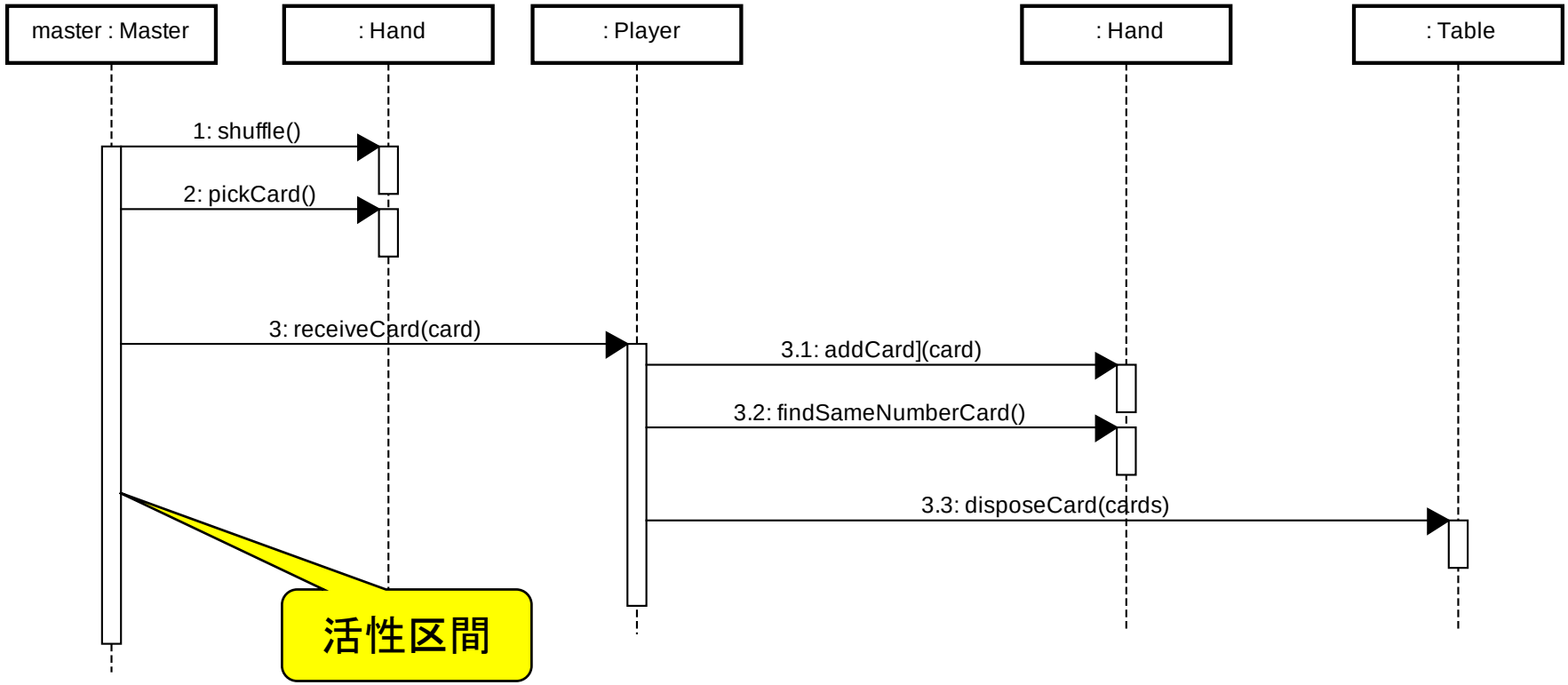
oo07_6d.asta より

前ページは以下をもとにしている



活性区間は気にしなくてよい

- astahを使うと必ず「活性区間」という矩形がかかれれてしまいます.
- コレの長さや，存在は気にしないでください.



デモ

- もとになるクラス図はsample07.zipにある oo07_6d.asta を参照してください.
- トランプゲームのものです.
- これをもとにシーケンス図を色々かけます.
- 想定されるシーケンスがかけない場合,
  - クラス, メソッドが不足している.
  - 関連が不足している.

  のどちらかです.

その他，例題

- 問題とクラス図，シーケンス図は，

http://www.sci.kanagawa-u.ac.jp/info/kaiya/oo/

の第7回のところからダウンロードしてください.

- 生命保険会社の業務支援システム
- 薬局の販売支援システム
- パソコン上の音楽ファイル再生アプリ

- astahを調べてみたらクラスは5千個以上だった！
  - 設計しないと破綻するゾ.

いままでのICONIXの苦労は？

- もし，いきなりクラス図を作れるのであれば，ユースケース，ドメインモデル，ロバストネス図等の苦労は不要である．
- カードゲームやじゃんけん等の簡単な教室での例題なら，おそらく，いきなりクラス図を書くことも可能．
- しかし，現実の業務アプリ等のクラス図を何の準備も無く書くのは不可能．
  - すくなくとも，漏れや抜けがあとから，ぼろぼろ出てくる．
- よって，ICONIXの苦労は，現実の開発では必要である．

次の話題へ

目次

- クラス図の再考 – ICONIXにおいて
- シーケンス図を書く理由
- シーケンス図を書く場合のガイドライン
- 事例

クラス図 -- そもそもの目標

- ソフトウェア開発の主たる目標は実行できるコードを作ることである.
- JavaやC#等のオブジェクト指向言語を想定する場合,

  クラスとクラス間の関連(クラス図)を決めることが,

  コードの骨組み(仕様)を決めることである.
- もし, いきなりクラス図が書けるなら, いままでのロバストネス図等のお絵かきは不要となる.
- クラス図に入れるべき情報を漏れなく集めるために, 仕方無く, なんとか図を書いているのである.

クラス図ができたら

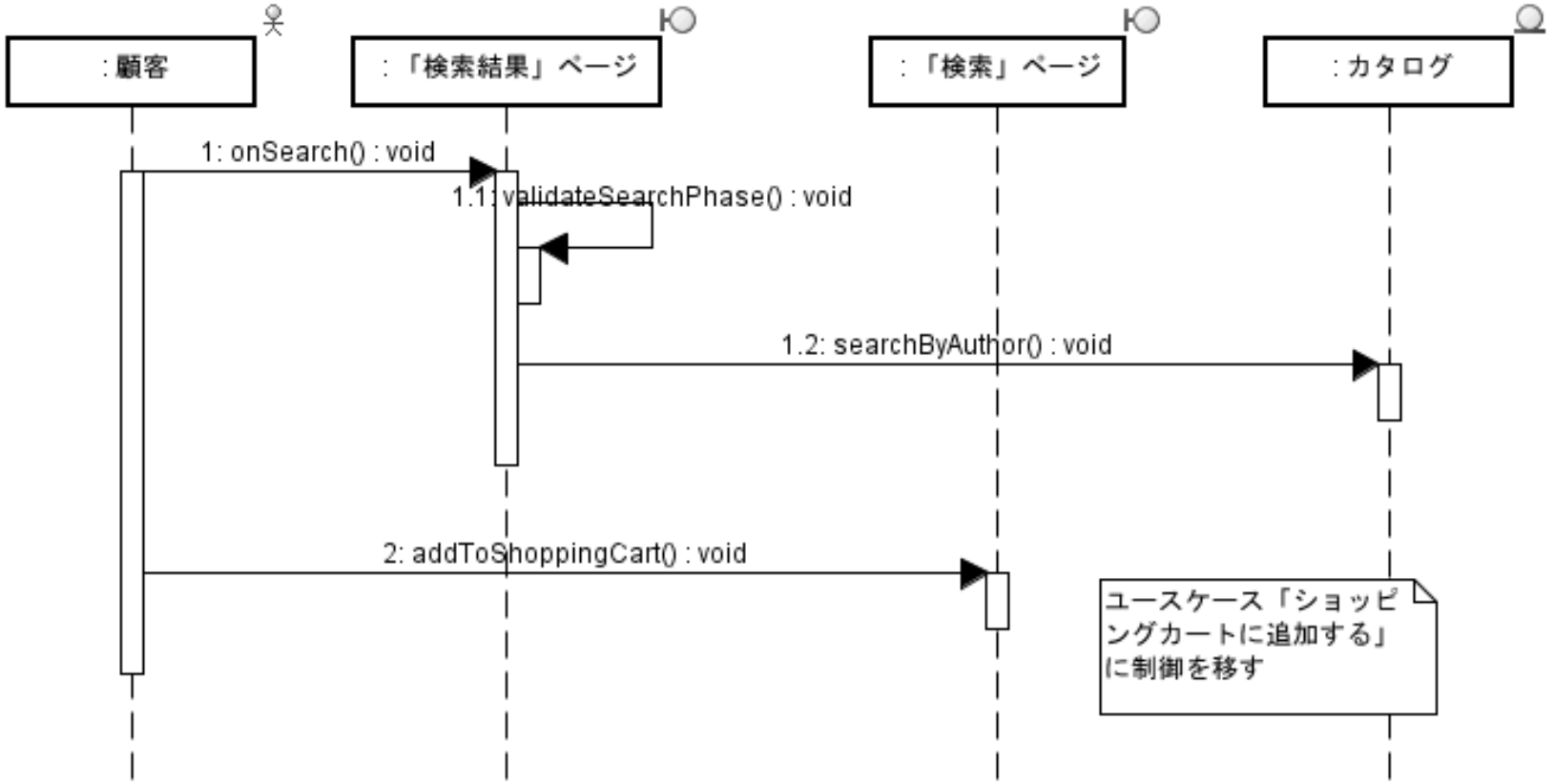
- 基本，クラス毎にコードを書けばよい．
- 各クラスが持つべきメソッドと，その意味は仕様化されているので，それにしたがって，コードを書く．
- 必要に応じて属性を追加してゆく．
- 後の改造や機能追加を見越して，クラス図の構造を修正する．

等

シーケンス図とその役割

- シーケンス図そのものについては，前のほうのスライド参照.
- シーケンス図を書く理由は，クラス毎に必要なメソッドを識別してゆくことである.

シーケンス図の例

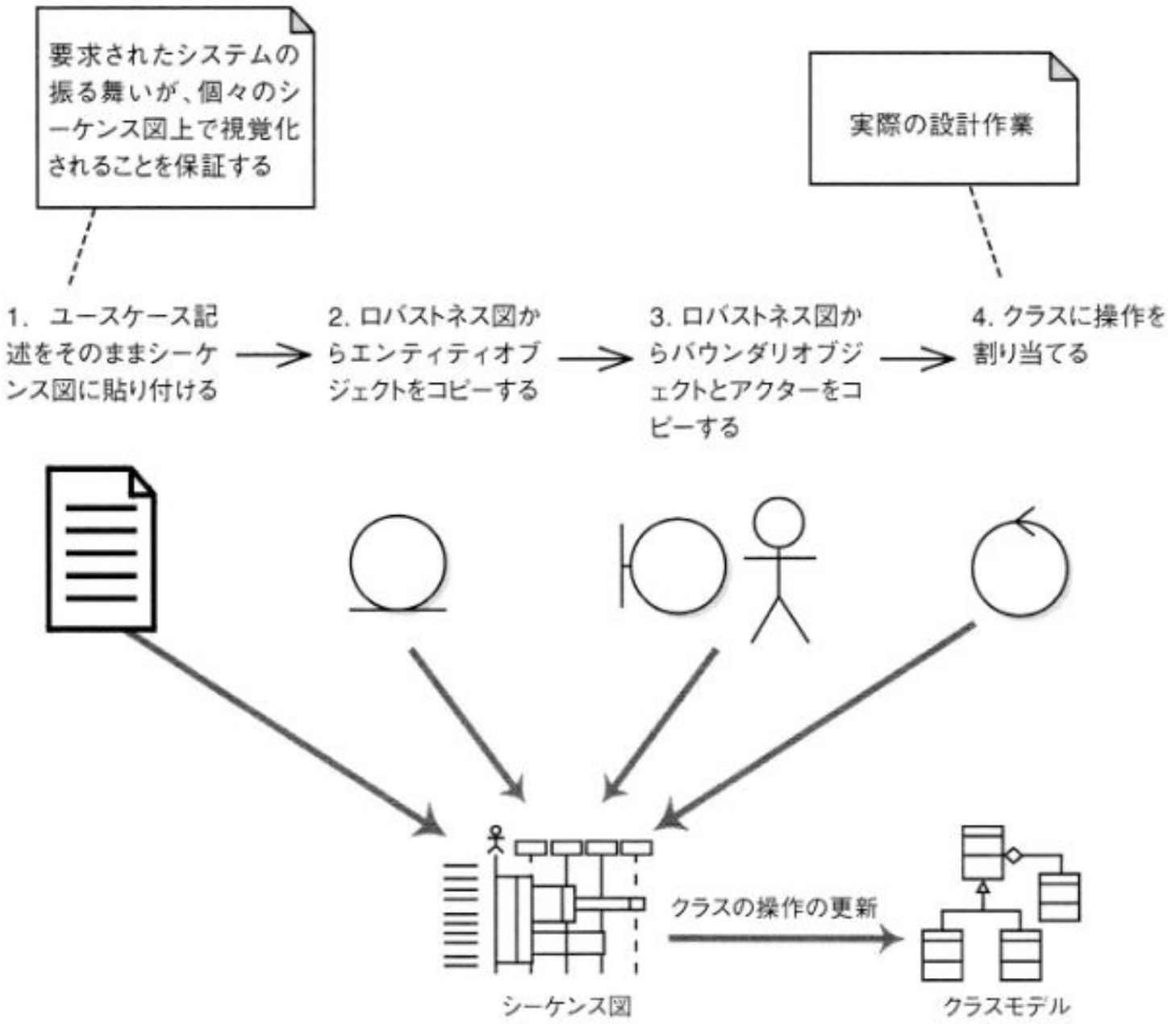


P211.asta

TIPS for astah

シーケンス図の登場人物は？

- 基本，ロバストネス図におけるアクター，バウンダリ，エンティティがライフラインを構成する.
- コントロールは，上記どれかのメソッドとなる場合が多い.
- コントロールもクラスとなりライフラインを構成する場合もある.


要求されたシステムの振る舞いが、個々のシーケンス図上で視覚化されることを保証する
実際の設計作業
1. ユースケース記述をそのままシーケンス図に貼り付ける
2. ロバストネス図からエンティティオブジェクトをコピーする
3. ロバストネス図からバウンダリオブジェクトとアクターをコピーする
4. クラスに操作を割り当てる
クラスの操作の更新
シーケンス図
クラスモデル

シーケンス図ガイドライン 1/2

1. なぜシーケンス図を書くか良く理解してかきなさい.
2. すべてのユースケースに対して，基本，代替，例外のシーケンス図を一つのシーケンス図に書きなさい.
3. シーケンス図の作成は，バンダリ，エンティティ，アクターそしてロバストネス分析の結果を反映したユースケース記述から始めなさい.
4. シーケンス図はユースケース図の振る舞い(すなわちコントローラー)をオブジェクトがどのように達成するかを示す道具として使いなさい.
5. ユースケース記述が，シーケンス図上でやり取りされるメッセージと対応付けられるかどうかを確認しなさい．記述とメッセージのやり取りとを並べてみるとよいでしょう.

ガイドライン 2/2

6. 活性区間に対する検討に長時間費やさないでください.
7. メッセージを書くことによって，操作をクラスに割り当てなさい.
8. 全ての操作が正しいクラスに割り当てられるように，操作の割り当てを行っている間はクラス図を繰り返しレビューしなさい.
9. コーディングを始める前に，シーケンス図上に描かれた設計をプレファクタリングしなさい.
10. 詳細設計のレビューを行う前に，静的モデルを整理しなさい.

1. シーケンス図を書く理由

- クラスに責務を割り当てること
  - コントローラーに相当する機能を，どのクラスが実施するかを明確にする.
  - 一つのコントローラーを一つのクラスが責任を持つとは限らない.
- あるユースケース中にクラス間がどのように相互作用するか明確にすること
- クラスに操作を割り当てる
  - 誰が誰に対して何をするか？を明確にする.
  - 結果として，クラス メソッド クラスの関係が明確になる.
  - JavaやC#にはメッセージの概念が無いので，メッセージを操作呼び出しに翻訳する感じ.

2. 基本，代替，例外
全て詰まったシーケンス図

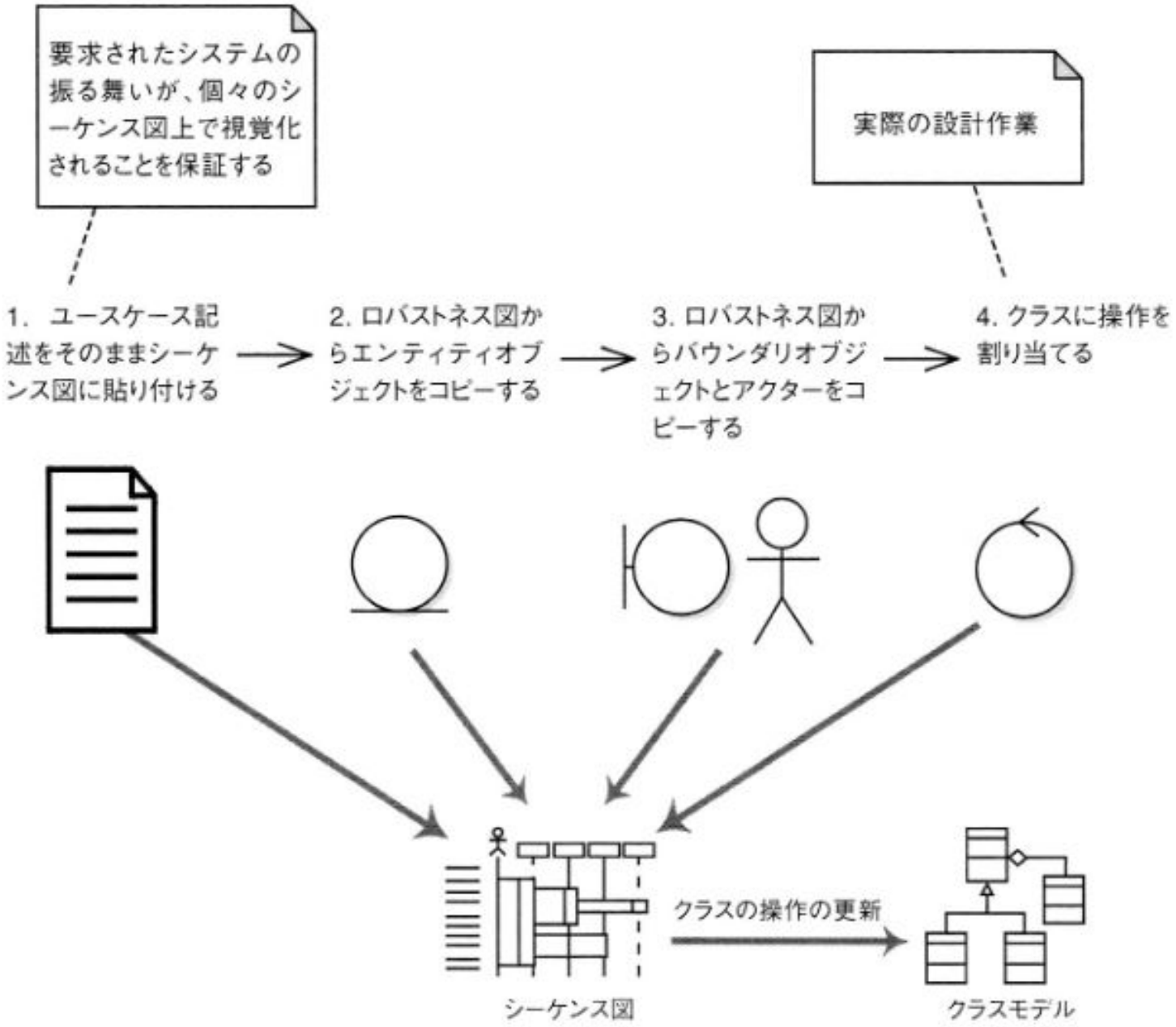
- 基本，代替，例外を別のシーケンス図には描かない．
- 全て詰まったシーケンス図が巨大になった場合，むしろ，もとのユースケースを分割すべき．

- 個人的には異論がある･･･

プレファクタリングとは？

- メソッドの命名変更，メソッドの他のクラスへの移動，メソッドをクラスに置き換える，条件記述の統合等をシーケンス図上で行うこと.
- これをコードを書き始めてからやることを，リファクタリングと呼ぶ.
- 一般にリファクタリングのほうがコストがかからない.

再掲載

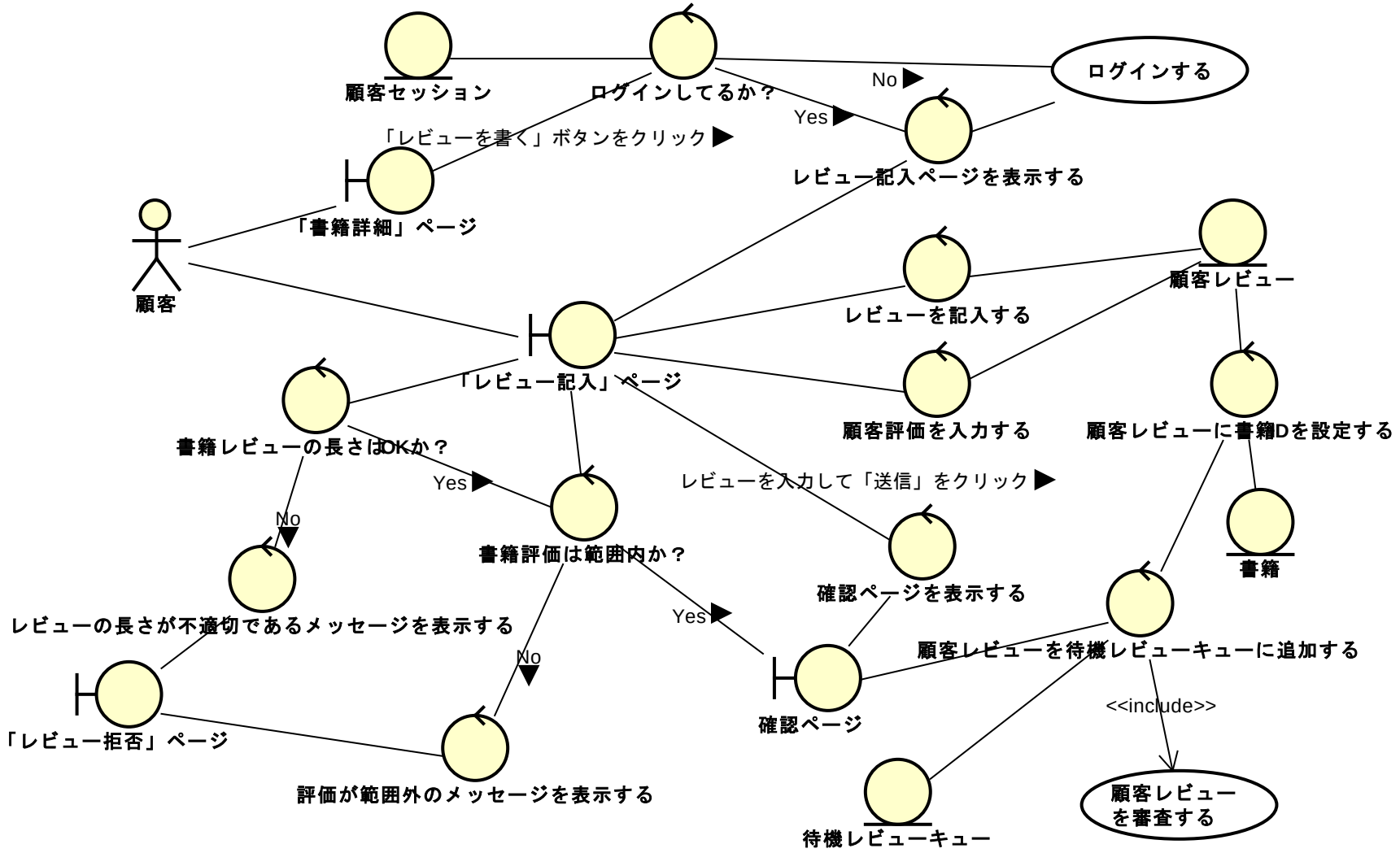

要求されたシステムの振る舞いが、個々のシーケンス図上で視覚化されることを保証する
実際の設計作業
1. ユースケース記述をそのままシーケンス図に貼り付ける
2. ロバストネス図からエンティティオブジェクトをコピーする
3. ロバストネス図からバウンダリオブジェクトとアクターをコピーする
4. クラスに操作を割り当てる
クラスの操作の更新
シーケンス図
クラスモデル

前頁の手順を実際にみてゆく

- 例題「顧客レビューを書く」ユースケース
- ステップ1は省略.
- ところどころ英語になっている.

前々回より

顧客レビューを書く最終版

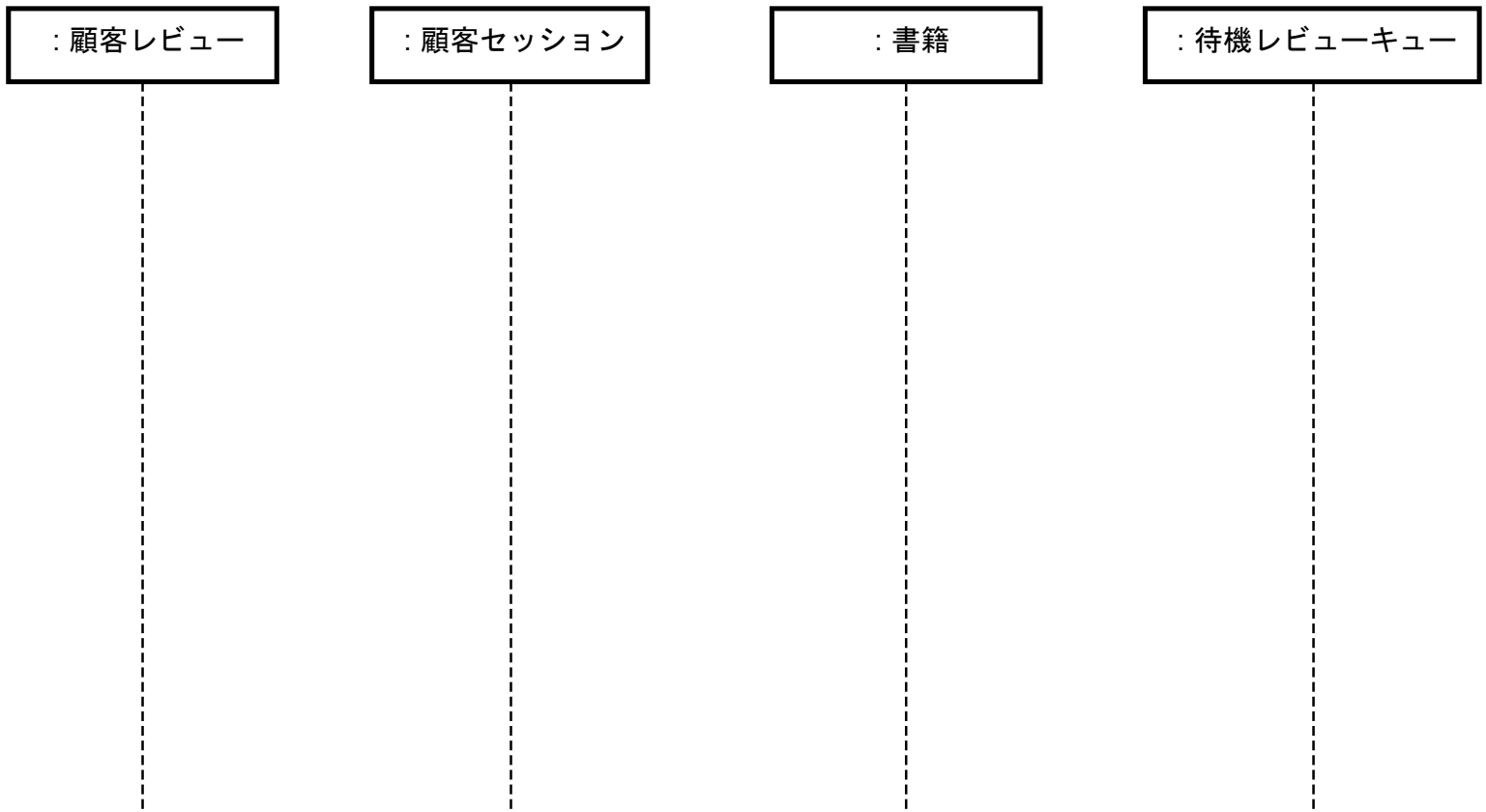


対応するユースケース記述は dotcampus p176ucd.xlsx より参照のこと.

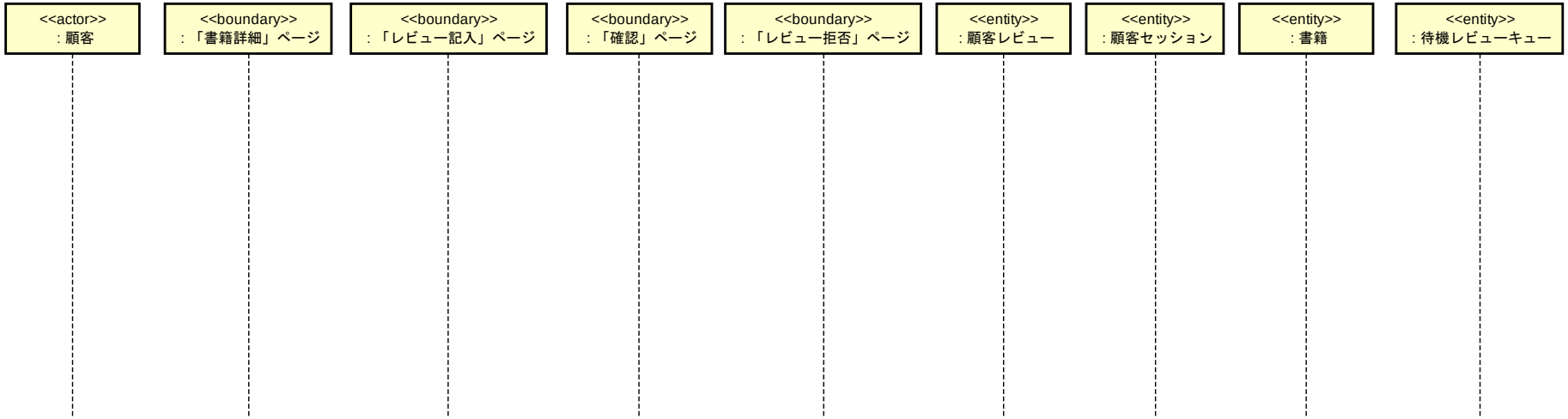
ユースケース記述

顧客レビューを書く	
基本コース	
顧客は現在表示されている書籍の「書籍詳細」ページで，「レビューを書く」ボタンをクリックする.	
システムは顧客がログイン済が否かをチェックするために顧客セッションをチェックする.	+
システムは「レビュー記入」ページを表示する.	
顧客は書籍レビューを入力し，書籍評価を5つ星までの範囲で指定し，「送信」ボタンをクリックする.	
システムはレビューが短すぎたり長すぎたりしないか，書籍評価が1～5の間となっているかどうかを確認する.	
システムは確認ページを表示する.	*
システムは顧客レビューを審査のために待機レビューキューに格納する.	+
このキューはユースケース「顧客レビューを審査する」で処理される.	+
例外コース	
顧客がログインしていない場合	
顧客はまずログイン画面を表示し，ログインしてからもう一度「レビューの記入」ページに移動する.	
顧客が書いたレビューが長すぎる(1MBより長い文章)場合	
システムはレビューを拒絶し，その理由を説明するメッセージを表示する.	
レビューが短すぎる(10文字未満)の場合	
システムはレビューを拒絶する.	

Step 2 エンティティを拾う

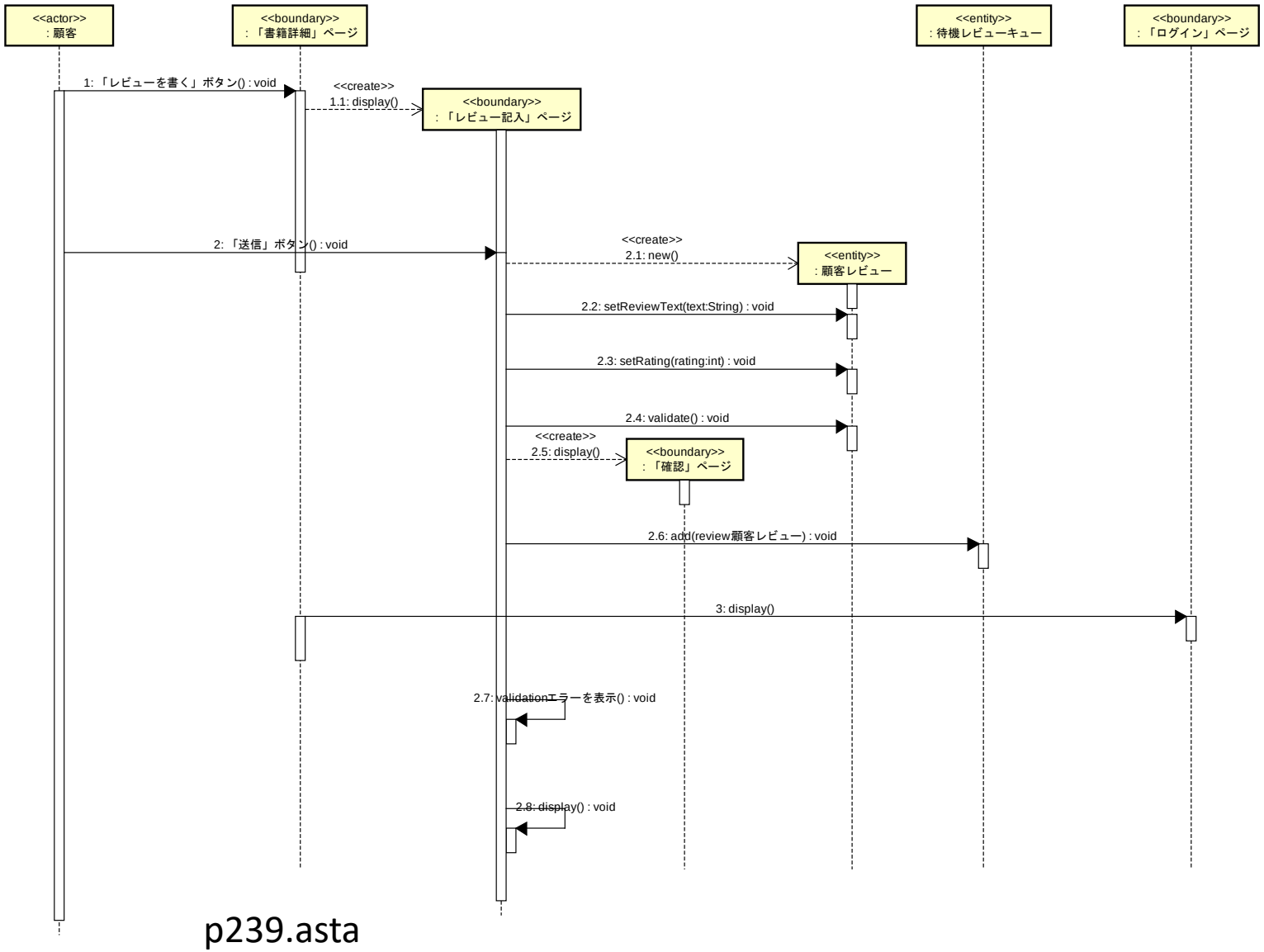


Step 3 アクター，バンダリーも



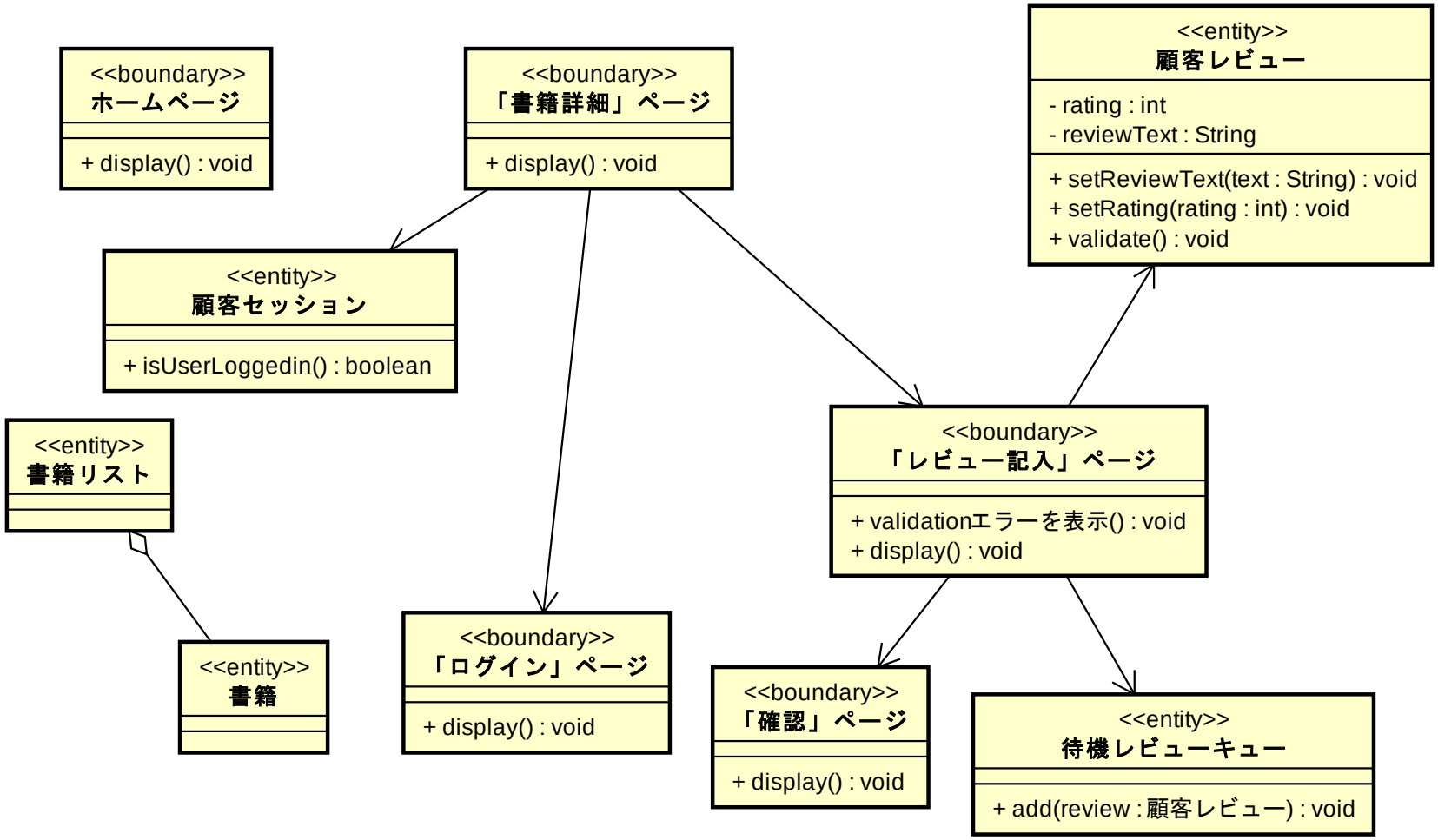
p226.asta をみてください

完成シーケンス図の例



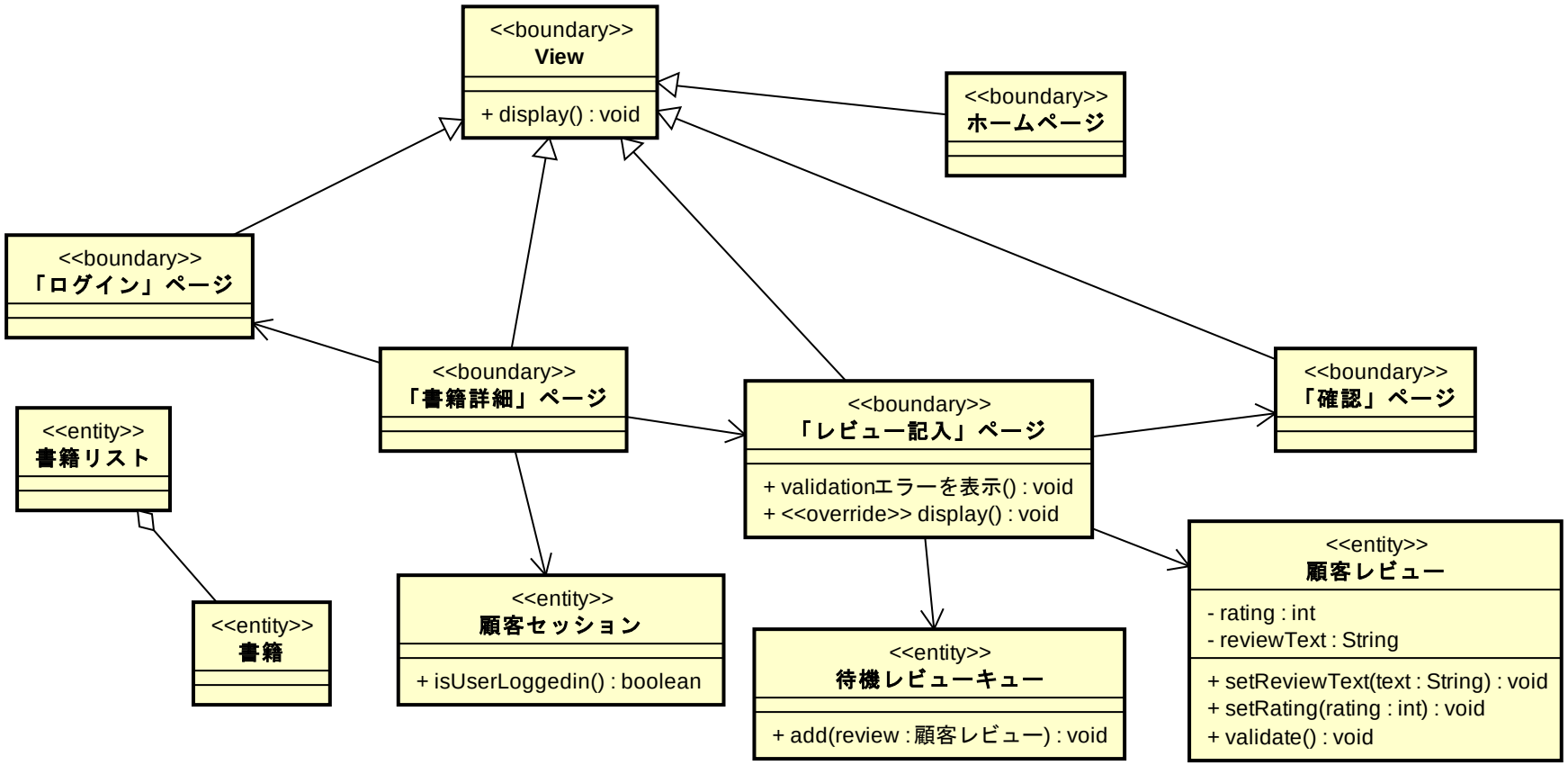
p239.asta

シーケンス図に基づくクラス図



プレファクタリングした

- やったことはページまわりのスーパークラスを作っただけ.
- 一部，サブクラスを残して中身の再定義(override)を行う.



本日は以上