(5) クラス間の動き

(5．1)
レストランでの動き

(5．2)
注文

(5．3)
クローク

(5．4)
クラス間の動きに関する小さな整理

クラス間の動きとして3種類を見てみる。
それから小さな整理を行います。

(5．5)
ブラックジャックのシナリオ

(2)(3)章でみた、ブラックジャックを例に、アクティビティ図を学ぶ。

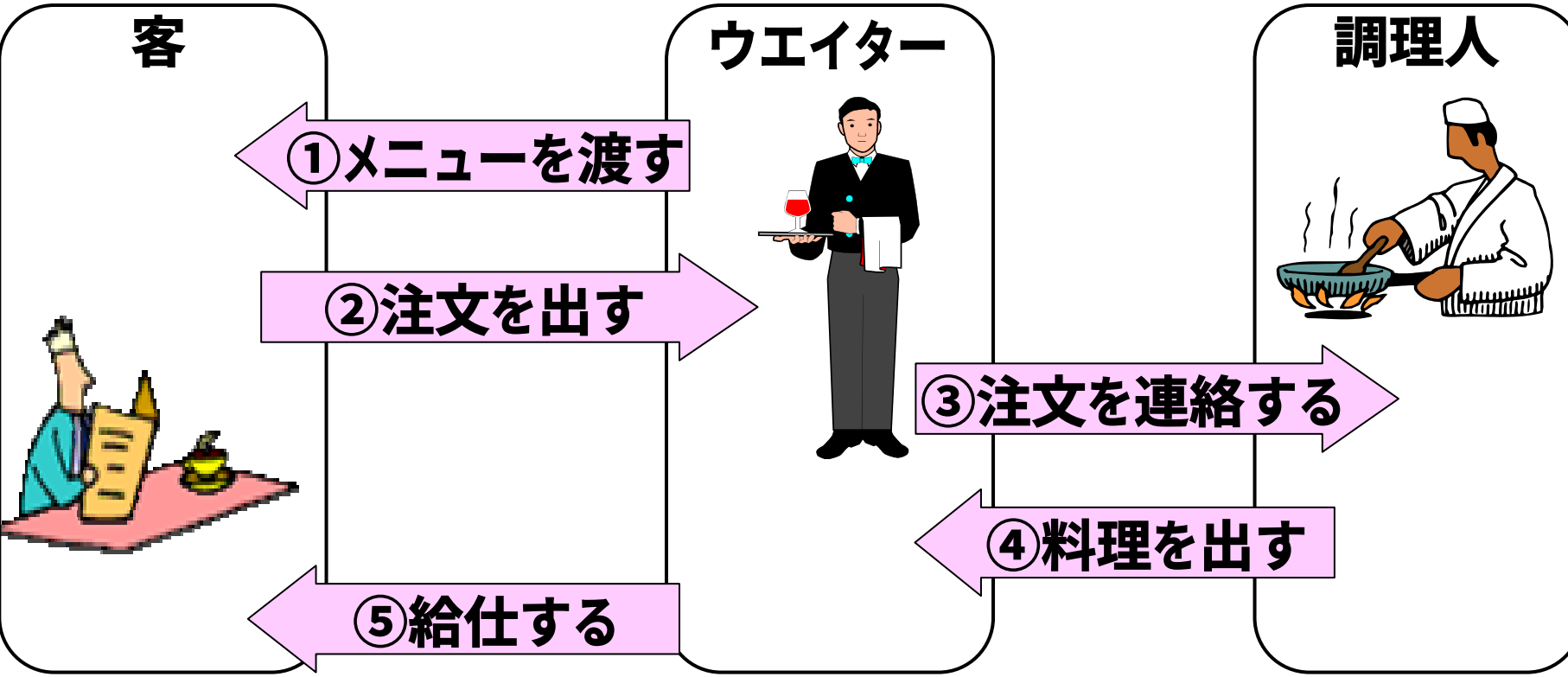
飛行機オブジェクト

(5．6)
ブラックジャックのシーケンス図

シーケンス図とプログラムとの対応を見る。

(5．1．1) レストラン

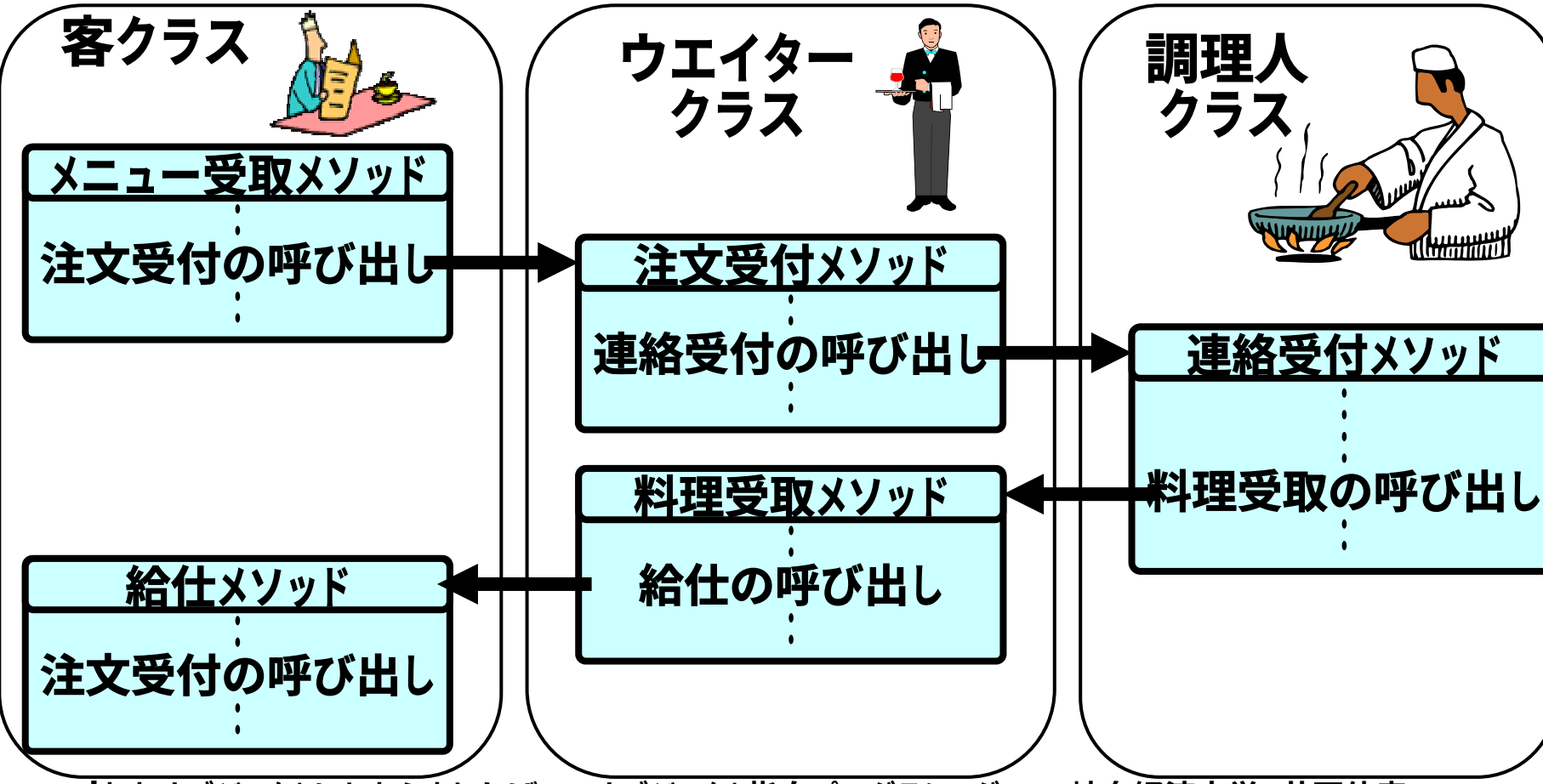
■レストランでの次のような動きを考えます。



■“客”、“ウエイター”、“料理人”の3つを、それぞれクラスと考えると、上記の図はプログラムでの動きに相当します。

(5．1．2) クラス間の動き

■それぞれのクラスにメソッドが用意されており、そのメソッドの中から次の動作にあたるメソッドが呼び出されます。



(5．1．3) プログラムのイメージ

■プログラムのレベルでも、各々の役割りを担うクラスが、メソッドを呼び合う関係はまったく同じです。

#客クラス
```
class customer{
public void menu(){
# メニュー受取メソッド
  waiter.order();
        :
```

```
public void serve(){
# 給仕メソッド
        :
}
```

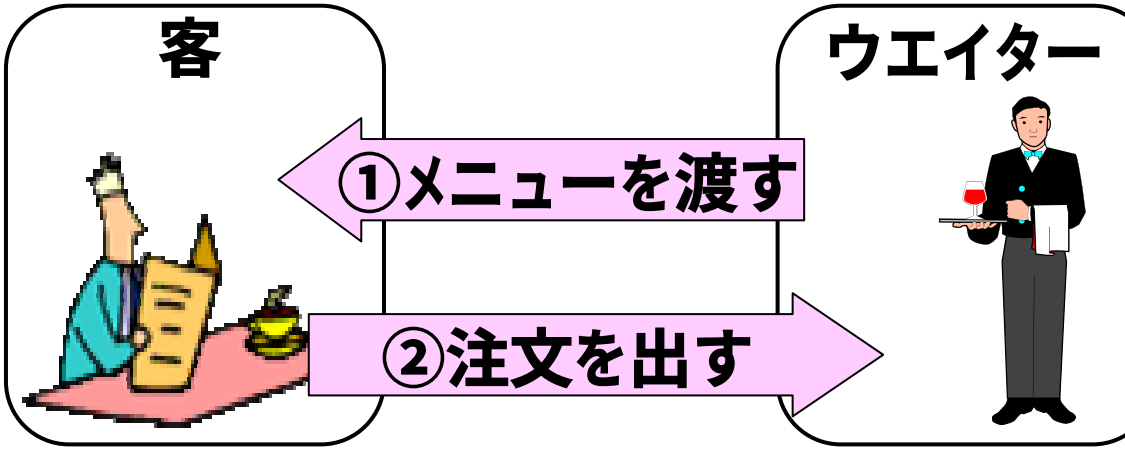
#ウエイタークラス
```
class waiter{
        :
public void order(){
# 注文受付メソッド
  cook.notice();
        :

public void dishes(){
# 料理受取メソッド
  customer.serve();
        :
}
```

#調理人クラス
```
class cook{
        :
public void notice(){
# 連絡受付メソッド
        :
  waiter.dishes();
        :
}
```

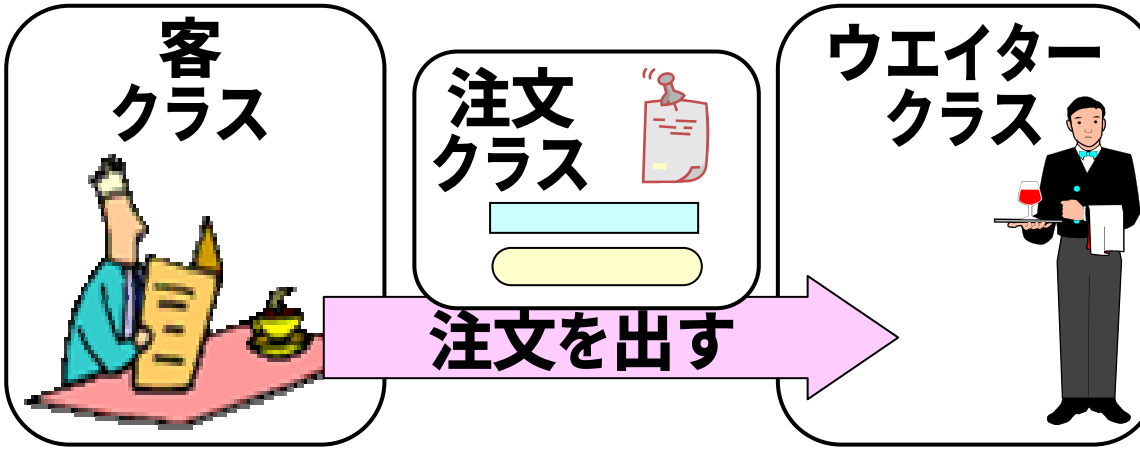
※ メソッドの呼び出し方については後述します。上の図中の記述は、イメージとして考えてください。

(5．2．1) “注文”はクラス

■スライド(5.1.1)中に出てきた“メニュー”や“注文”などの、クラス間で渡されるものについては、どう考えれば良いのでしょうか？

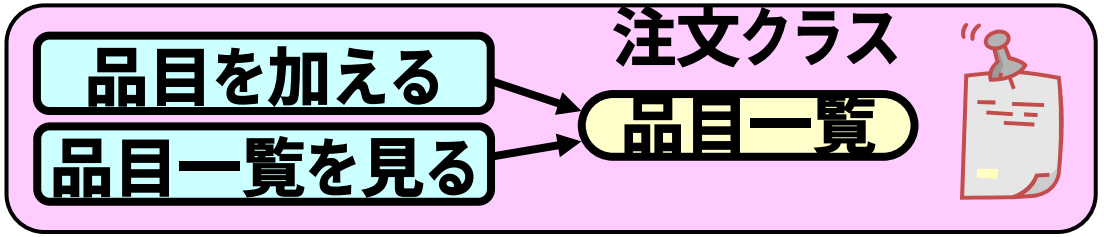


■“注文”も、やはり“もの”と扱いたい気がします。実際そうすることにしてみましょう。

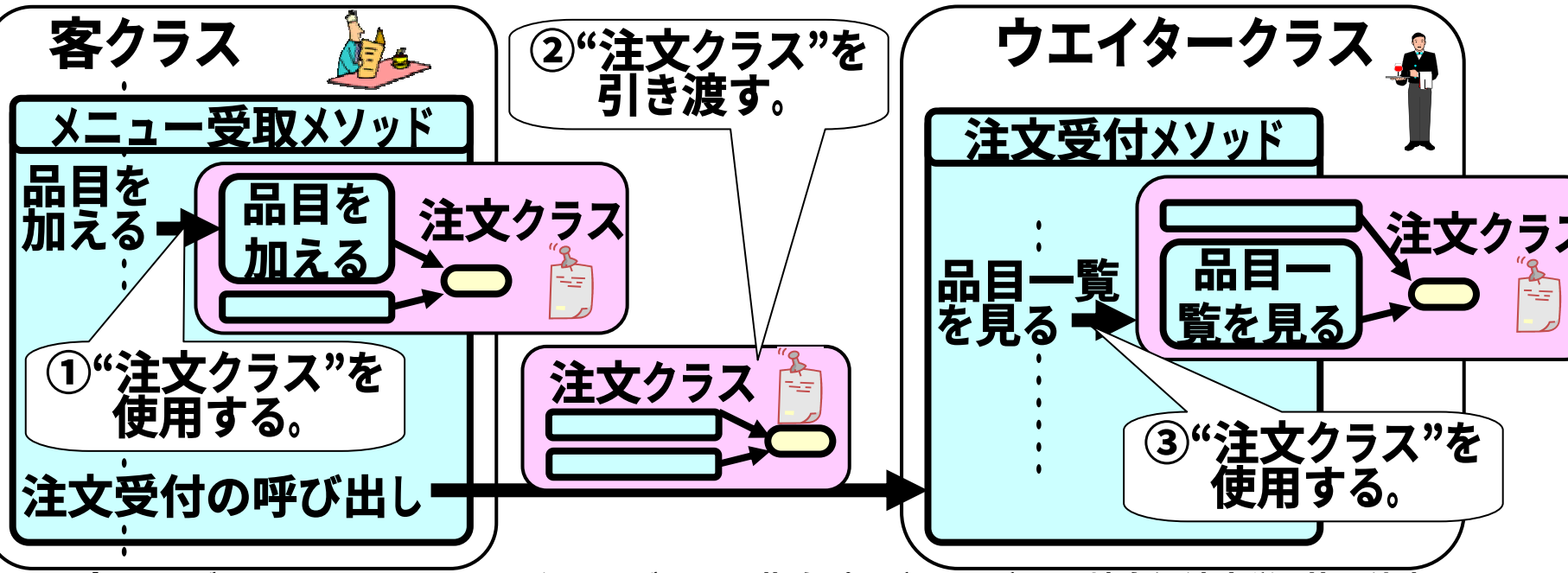


(5．2．2) 注文クラス

■注文をクラスと考え、次のようなメソッド・変数を持たせることとします。



■客クラスとウエイタークラスでは、次のように注文クラスを引渡し、双方で使うことになります。



(5．2．3) 注文クラス:プログラムのイメージ

■プログラムのレベルでも、注文クラスを引渡されて、客クラスとウエイターククラスとの双方で使われることは全く同じです。

```
#客クラス
class customer{
public void menu(){
# メニュー受取メソッド
        :
orders_i.put(“鍋物”);
        :
        :
        :
        :
        :
        :
waiter.order(orders_i);
        :
}
```

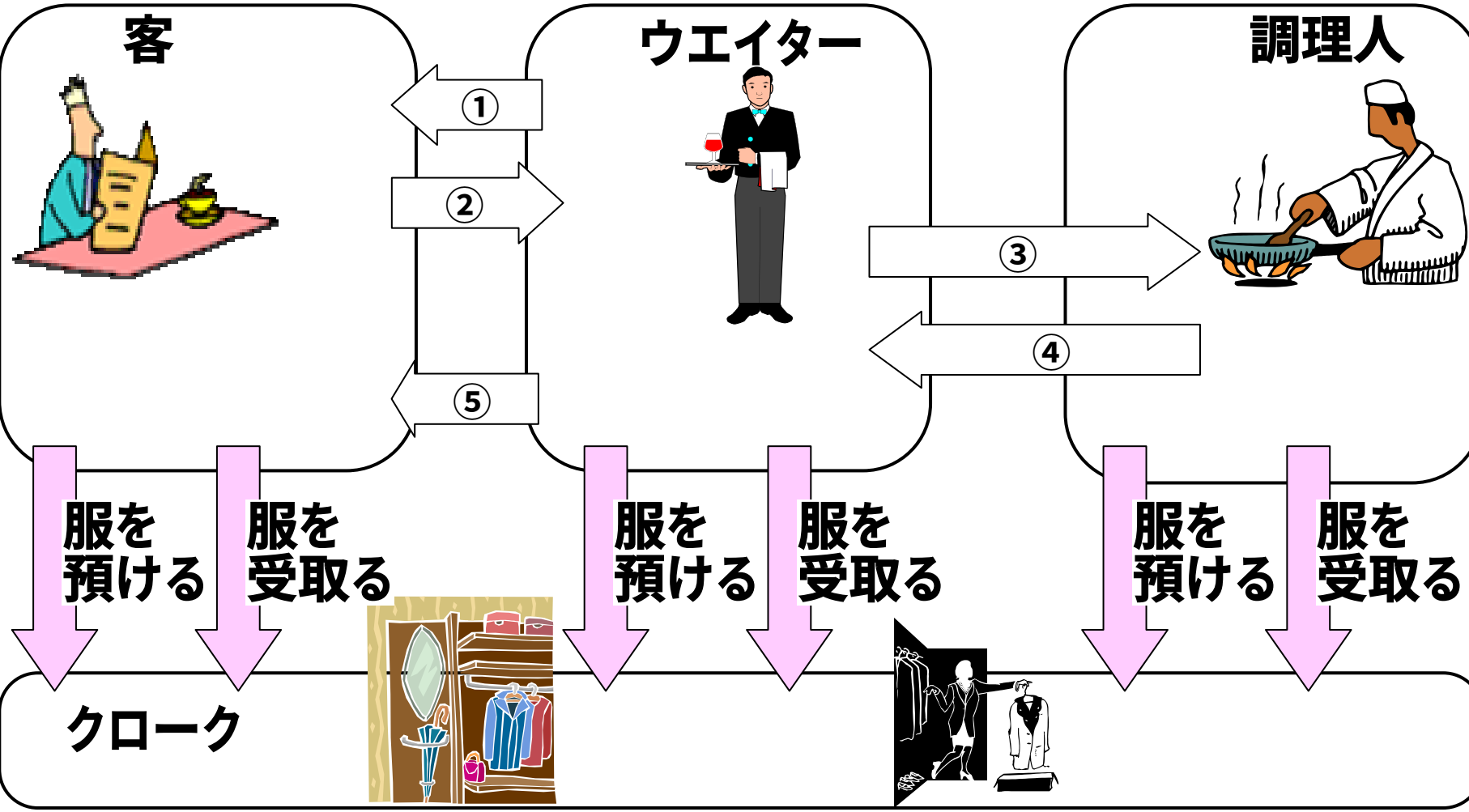
```
#注文クラス
class orders{
public void put(){
# 品目追加メソッド
:

public void get(){
# 品目一覧メソッド
:

}
```

```
#ウエイタークラス
class waiter{
        :
public void
order(orders orders_i){
# 注文受付メソッド
        :
item = orders_i.get();
        :

cook.notice();
        :

}
```

※ 引数については、イメージとして書いています。

(5．3．1)“クローク”

■少し不自然ですが、客もウエイターも調理人も、同じくクロークを使うこととします。



(5．3．2)“クローク”はクラス

■“クローク”も、やはりクラスとして扱うことになります。

#客クラス
```
class customer{
        :
        :
lavel=cloakroom.p
ut(“服”);
        :
suit=cloakroom.ge
t(“lavel”);
        :
}
```

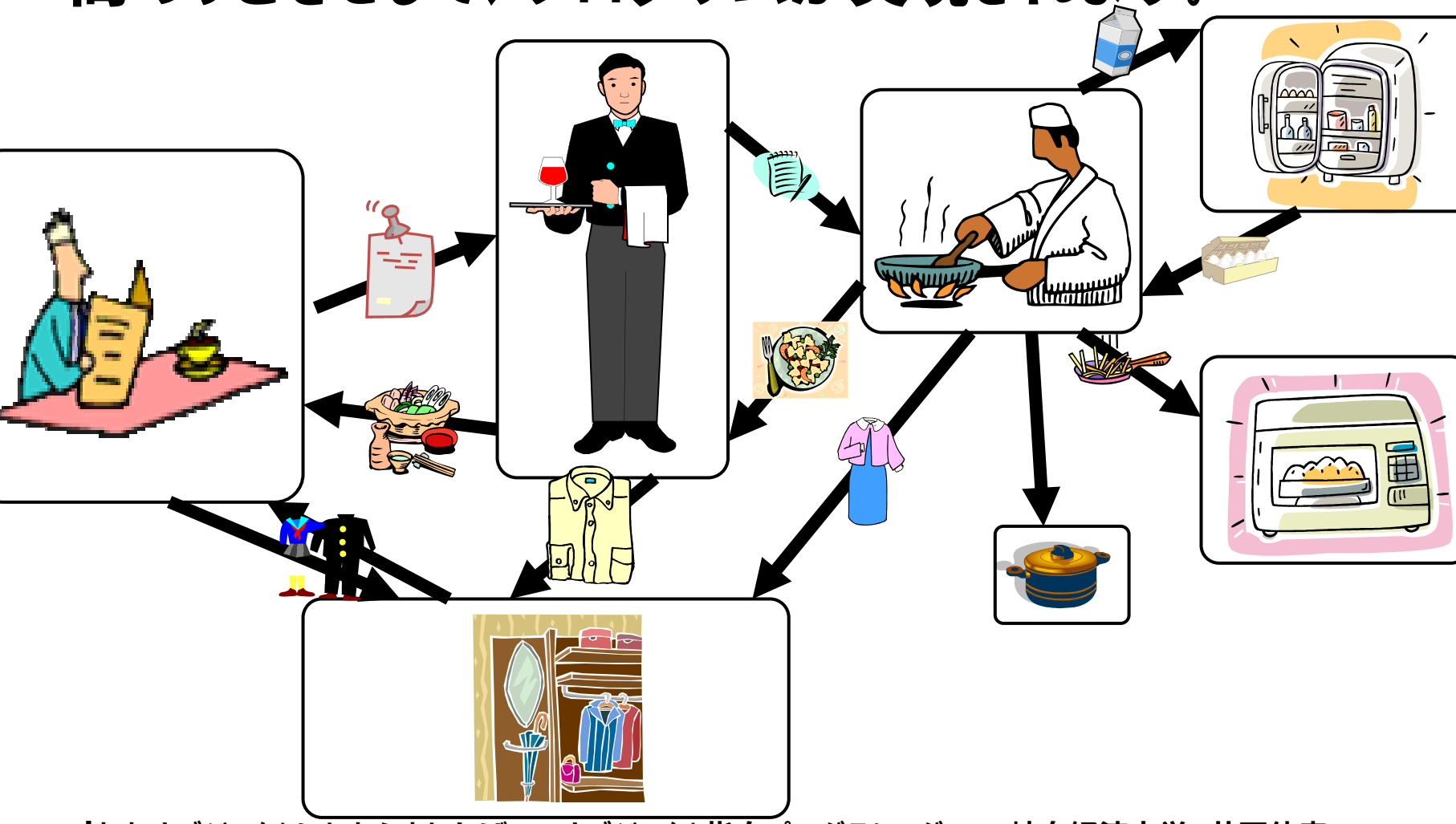
#ウエイタークラス
```
class waiter{
        :
        :
lavel=cloakroom.p
ut(“服”);
        :
suit=cloakroom.ge
t(“lavel”);
        :
}
```

#調理人クラス
```
class cook{
        :
        :
lavel=cloakroom.p
ut(“服”);
        :
suit=cloakroom.ge
t(“lavel”);
        :
}
```

#クロッククラス
```
class cloakroom{
        :
public int put(suit){
# 預かりメソッド
        :
public String get(suit){
# 受取メソッド
        :
}
```

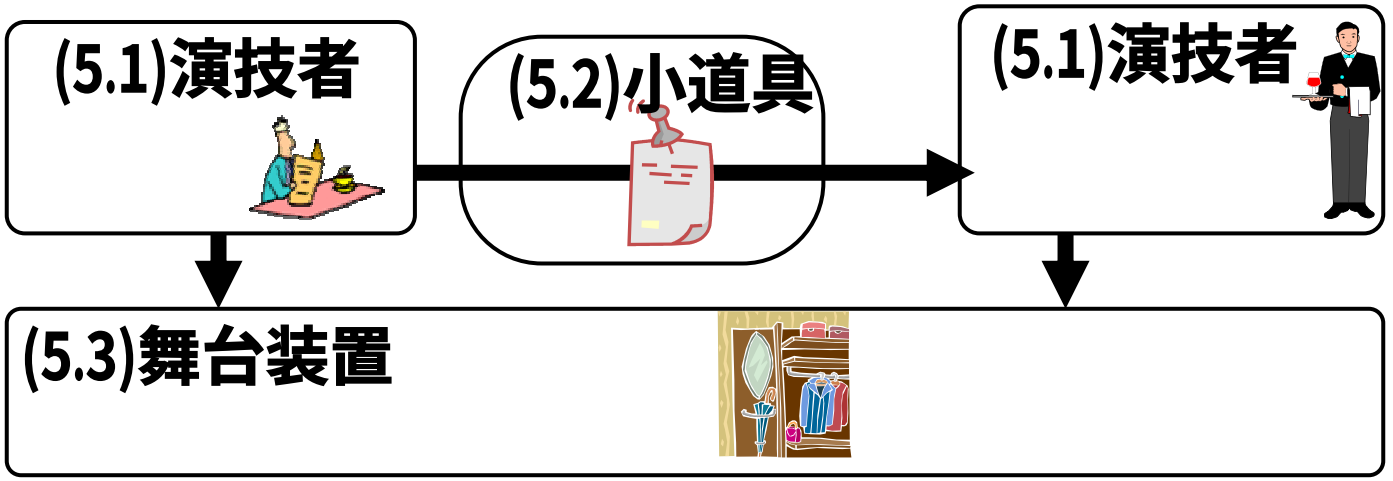
(5．4．1) クラスの動きに関する小さな整理

■結局のところ、さまざまな“もの”の間の動きが、クラス間のうごきとして、プログラムが実現されます。



(5．4．2) 劇に例える

■劇に例えると、スライド(5.1),(5.2),(5.3)で見たクラスは、それぞれ(5.1)演技者、(5.2)小道具、(5.3)舞台装置にあたるようです。



■このような分類が一般的にあるわけではありません。同じクラスが、演技者になったり小物に見られたりすることもあります。さまざまな役回りのある“クラス”があることの理解の一助としてください。

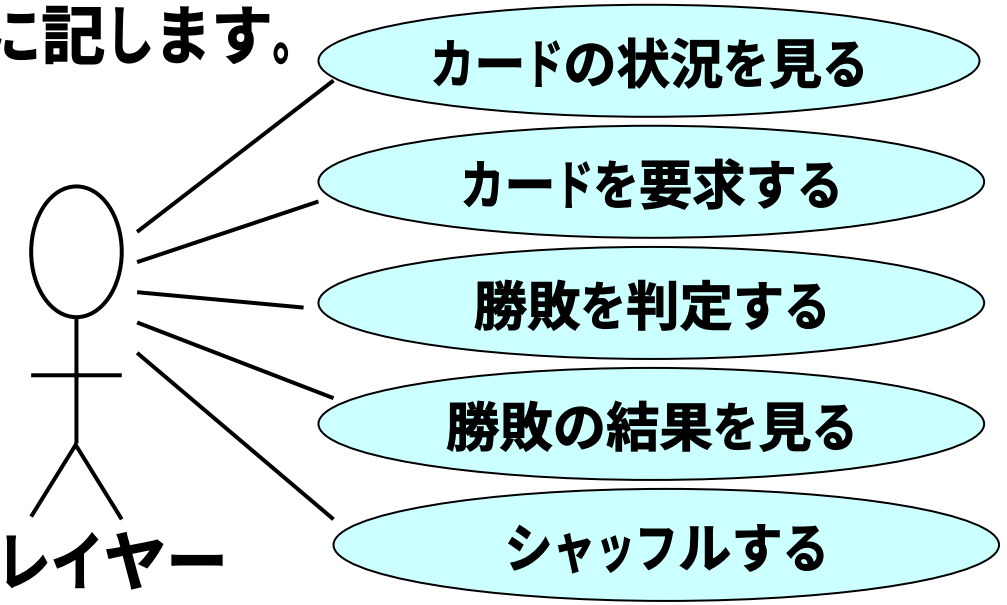
(5．5．1) ブラックジャックの全体像

■ブラックジャックでは、プレイヤー (競技者、player)がおり、様々な操作を行いゲームを進めていきます。

■操作には、次のようなものがありました。

- (配られた)カードの状況を見る　・カードを要求する (hit)
- 勝敗を判定する(stand)　　・勝敗の結果を見る
- (カードを) シャッフルする(shuffle)

■このような全体像を、UMLではユースケース図を使って、次のように記します。



※　ユースケース図については、別スライドにて再度勉強します。

(5．5．2) 各クラスの性格付け

■ブラックジャックでは、5つのクラスとしています。

ウインドウ(Window)クラス
・操作を行う画面のウインドウ
・ゲーム (カード配布や得点、勝ち負けを操作)

プレイヤー(player)クラス
・競技を行う人

カード(card)クラス
・1枚のカード

ディーラー(dealer)クラス
・プレイヤーと競って競技を行う人

カードデッキ(card deck)クラス
・競技者に配るカードを束ねたもの

■“ウインドウクラス”は、“もの”としてやや考えづらい点があります。現実的には、このようなクラスを作らざるを得ない場合があります。

■“カードクラス”は(5.2)小道具的、カードデッキは(5.3)舞台装置的、その他プレイヤーなどは(5.1)演技者的と言えます。

(5．5．3) クラスの属性

■各々のクラスには、おおよそ次のような属性 (データ、変数) があります。

ウインドウ(Window)クラス

画面表示のデータ

プレイヤー(player)クラス

所持カード一覧

ディーラー(dealer)クラス

所持カード一覧

カード(card)クラス

マーク

数字

カードデッキ(card deck)クラス

積み上げたカードの一覧

(5．5．4) クラス間の動作のシナリオ

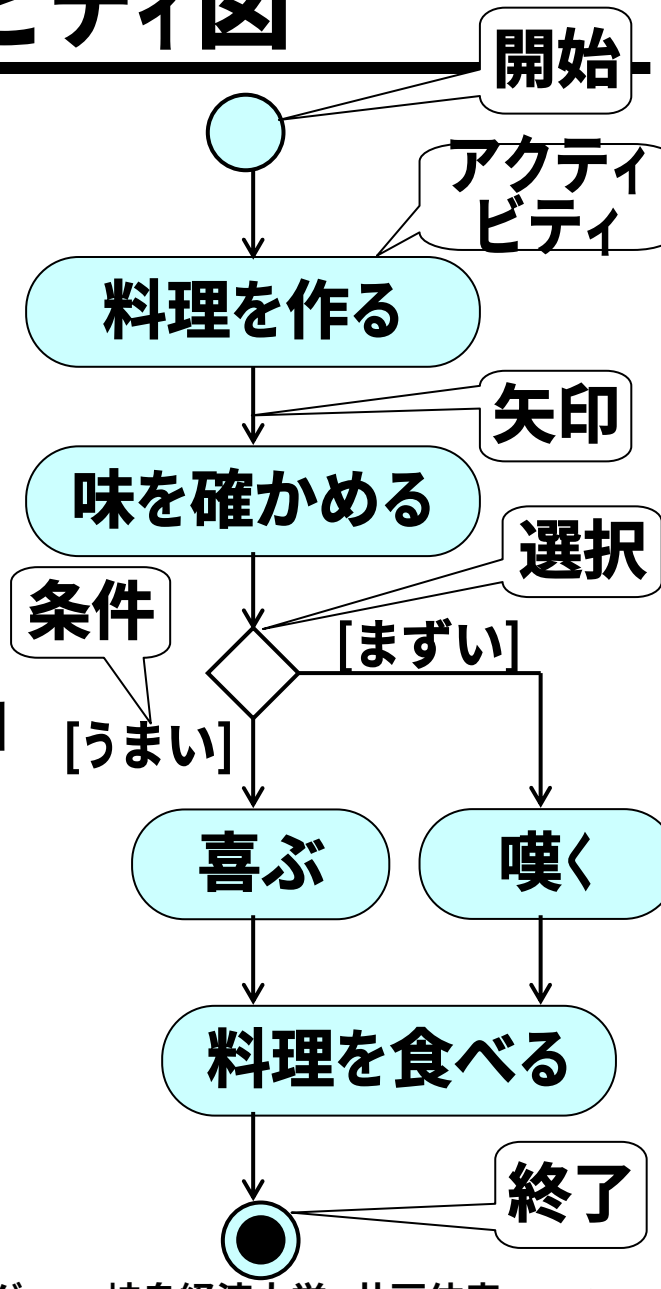
■スライド(2)(3)章でみた、ブラックジャックについてクラス間でどのような動作をするかを考えていきます。

■まず、言葉で説明してみましょう。

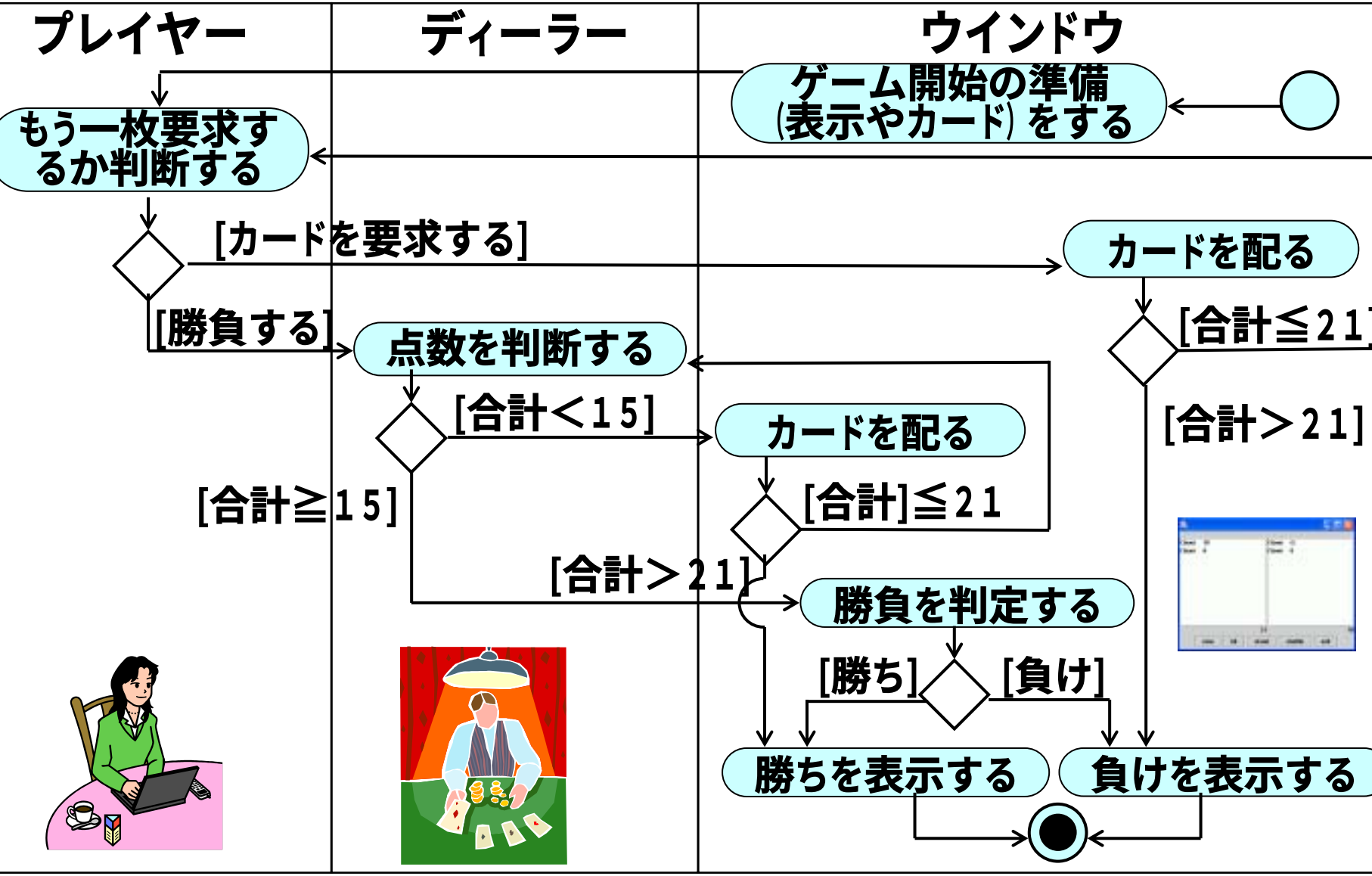
- コンピュータがカードを2枚配る。
- プレイヤーは、その2枚の持ち札を見て、さらにカードが必要かを判断する。
- もし必要なら新たにカードを要求し、必要でなければ、そのまま勝負を要求する(hit)。
- また、ユーザが勝負を要求する(stand)と、次はディーラーが手作りをする番になる。
- ディーラーは自分のポリシーをもとにカードの要求を行い、さらなるカードは不要と判断した時点(15点以上とします)で、勝負を要求する。
- すると、プレイヤーとディーラーの合計が比較され、勝敗が判定される。その結果を表示し、ゲームは終了となる。

(5．5．5) アクティビティ図

- クラス間のおおまかな動きは、アクティビティ図を使って記します。
- アクティビティとは、ある作業を行っていることを指します。
- アクティビティ図では、作業がどのように順次実行されていくのかを、アクティビティを矢印 (トランジッション) で結んでいくことで示します。
- 開始記号から始まり、終了記号で終わりとなります。
- 次のアクティビティを選ぶ判断として、選択条件も描けます。

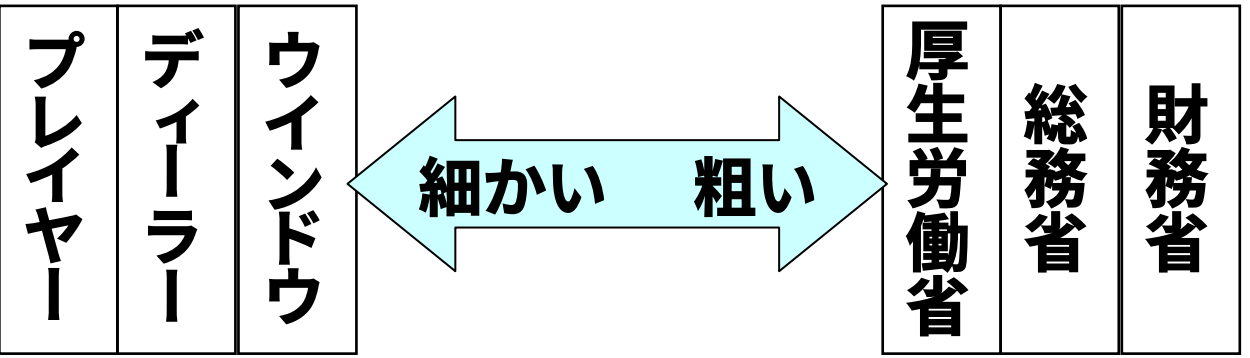


(5．5．6) 大まかな動き


プレイヤー
ディーラー
ウインドウ
ゲーム開始の準備
(表示やカード) をする
もう一枚要求す
るか判断する
[カードを要求する]
[勝負する]
カードを配る
[合計≦21]
[合計＞21]
点数を判断する
[合計＜15]
[合計≧15]
カードを配る
[合計]≦21
[合計＞21]
勝負を判定する
[勝ち]
[負け]
勝ちを表示する
負けを表示する

(5．5．7) スイムレーン

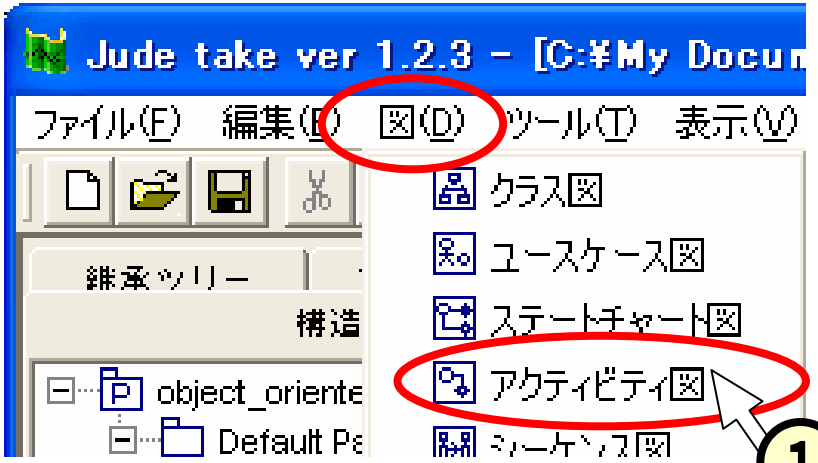
- スライド(5.5.6)のスライドでは、“プレイヤー”／“ディーラー”／“ウインドウ”の3つの区分に分けてアクティビティ図を描いていました。このような区分を“スイムレーン”(またはパーティション)といいます。
- 一般的には、スイムレーンはアクティビティをグループ化するものです。
- ブラックジャックは小さいプログラムです。
  - このため、アクティビティ図を作成すると、非常に詳細なレベルの記述になり、スイムレーンもクラス対応相当になっています。
  - アクティビティ図は作業を順次書いていくため、ブラックジャックの場合、(5.1)演技者に相当する、“プレイヤー”／“ディーラー”／“ウインドウ”の3つクラスを中心に描かれることになります。
- 大きなシステムの概要を示すアクティビティ図では、スイムレーンはもっと大きな区分となり、クラスとしては複数が対応するようなものになります。アクティビティ図には、記述のレベルがあるということです。

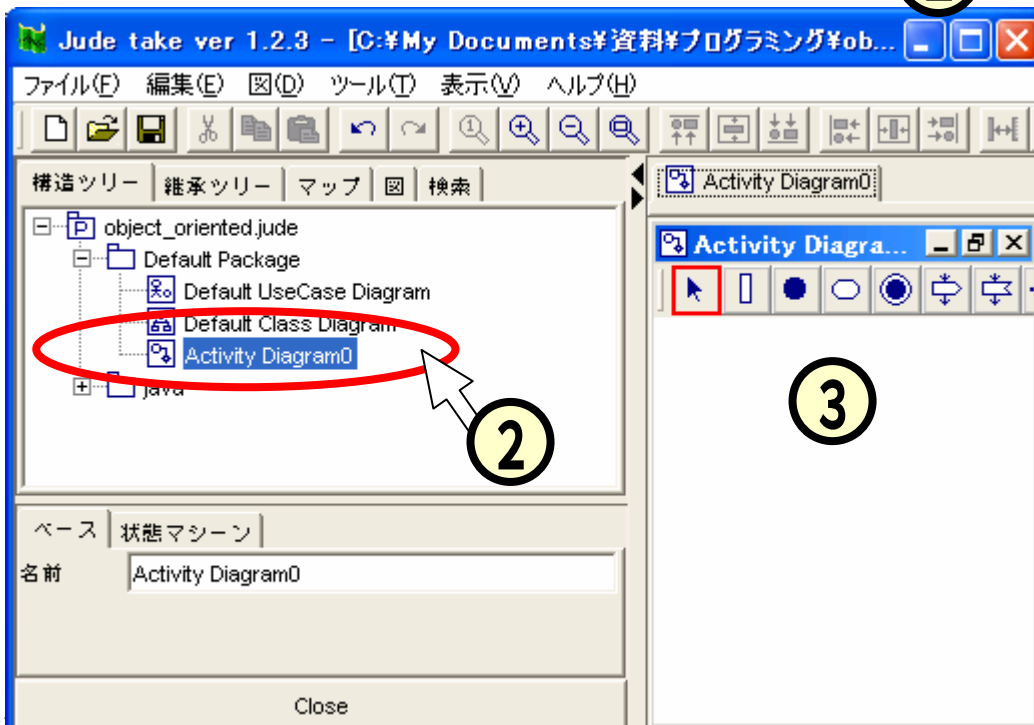


(5．5．8) 新規アクティビティ図を開く

■[図]- [アクティビティ図]をクリック (①) します。

■ツリー画面に新しいダイアログ (Activity Diagram) が表示され、これをクリック (②) してフォーカスすると、編集画面 (③) にて図の作成が可能になります。





(5．5．9) アクティビティ要素の配置

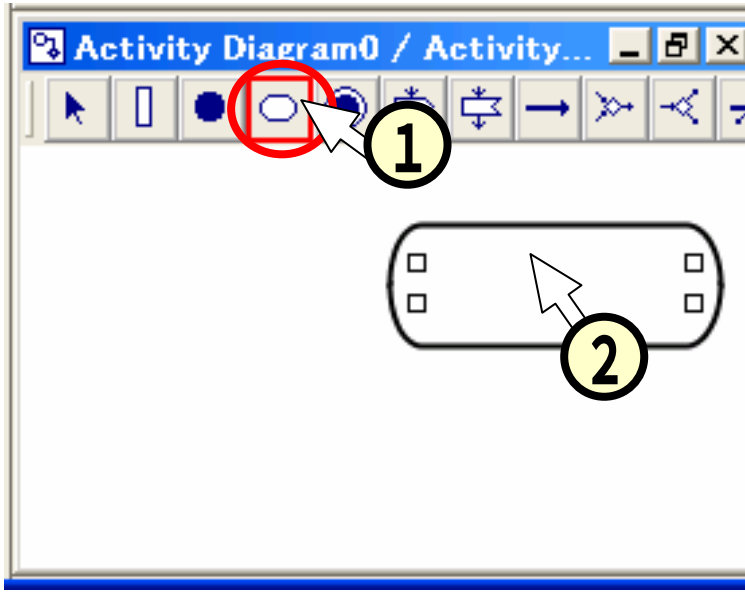
■ツールバーの[アクション状態]をクリック (①) して、編集画面上をクリック (②) します。

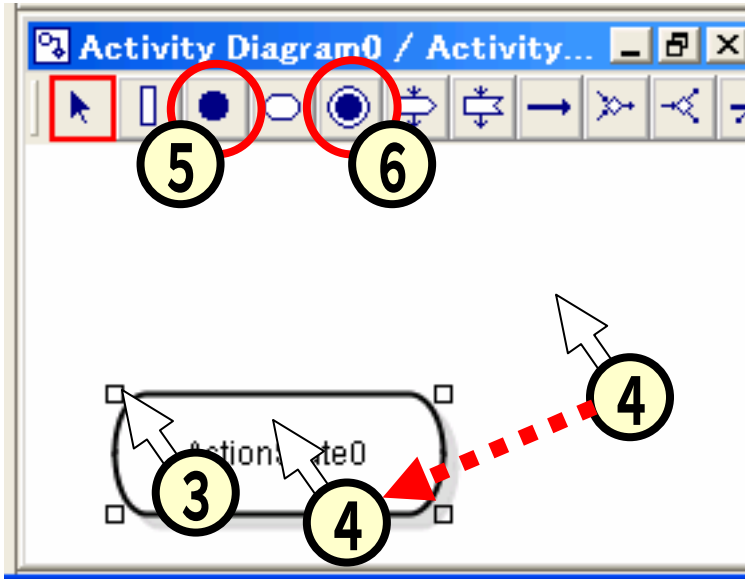
■アクティビティ要素が、名前の入力待ちの形で配置されます。ここで名前の入力が可能ですが、後から名前の欄をダブルクリックして入力することも出来ます。

■ドラッグ (③) することで、移動させます。また、要素の四隅のつまみをドラッグ (④) して、大きさを調整します。

■フォーカスされた状態 (＝四隅につまみが表示) で[Delete]キー押下すると、要素は削除されます。

■開始記号 (⑤)、終了記号 (⑥) も同様に配置します。



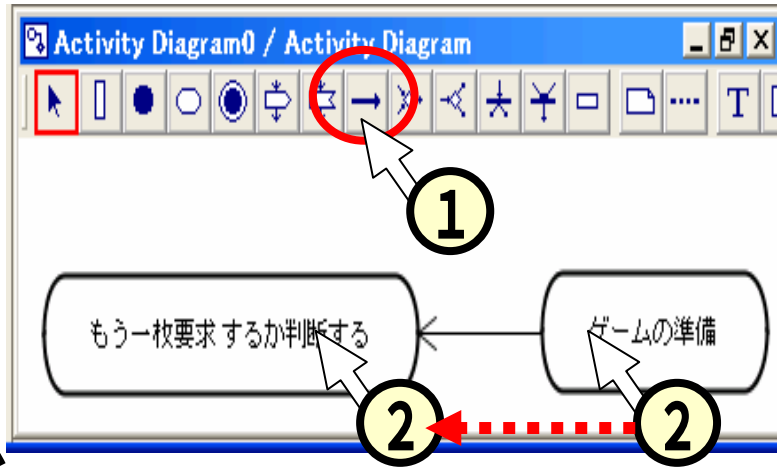


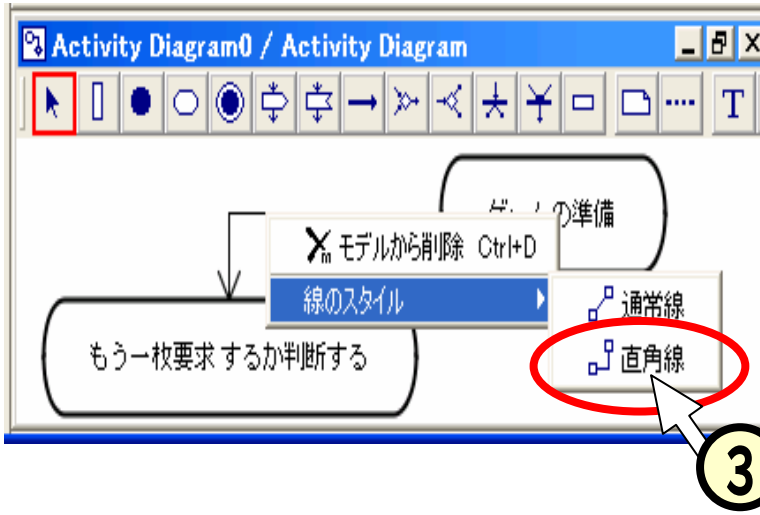
(5．5．10)トランジッション要素の配置

■スライド(5.5.9)のようにして、2つのエンティティ要素を作ったとします。この2つの間に、メッセージ要素を付け加えます

■ツールバーのメッセージ要素をクリック(①)して、メッセージ送信元のアクティビティ要素から受信先のアクティビティ要素へ、ドラッグ(②)します。

■トランジッションの矢印にも、直角線の指定(③)をすることが出来ます。

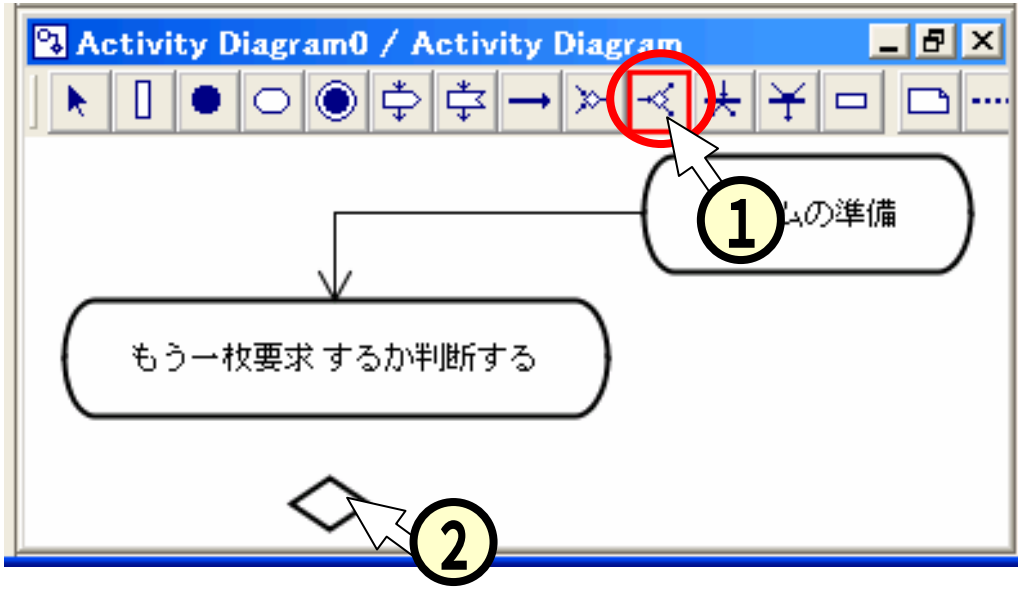


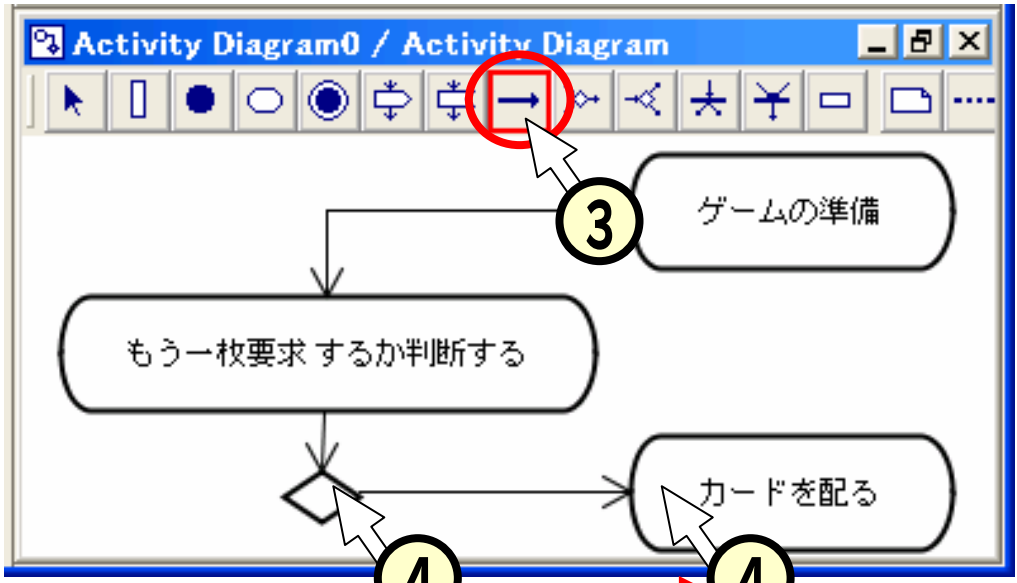


(5．5．11) 選択要素の配置

■選択要素を配置するには、ツールバーの[選択]をクリック(①)して、編集画面上をクリック(②)します。

■選択要素に対するトランザクションの配置は、アクティビティ要素への配置と同様に行えます(③、④)。



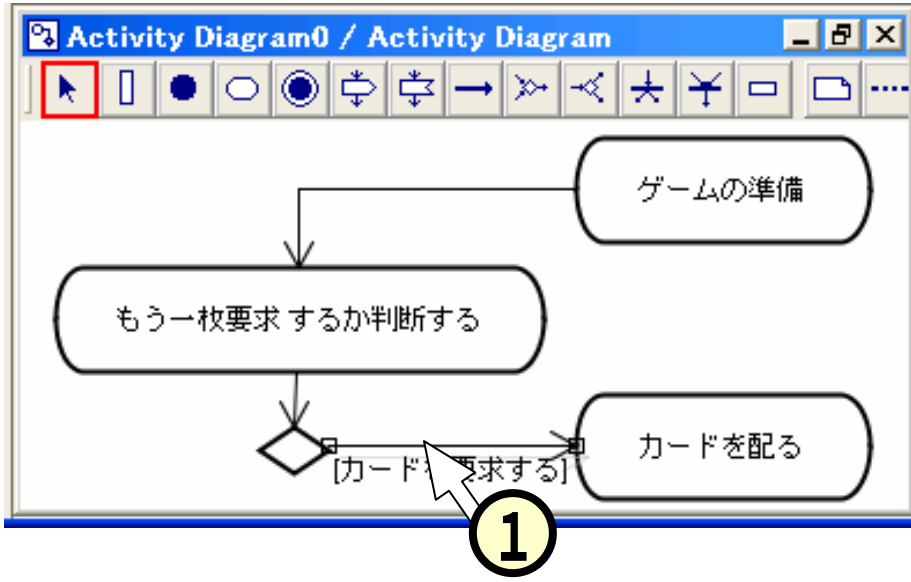


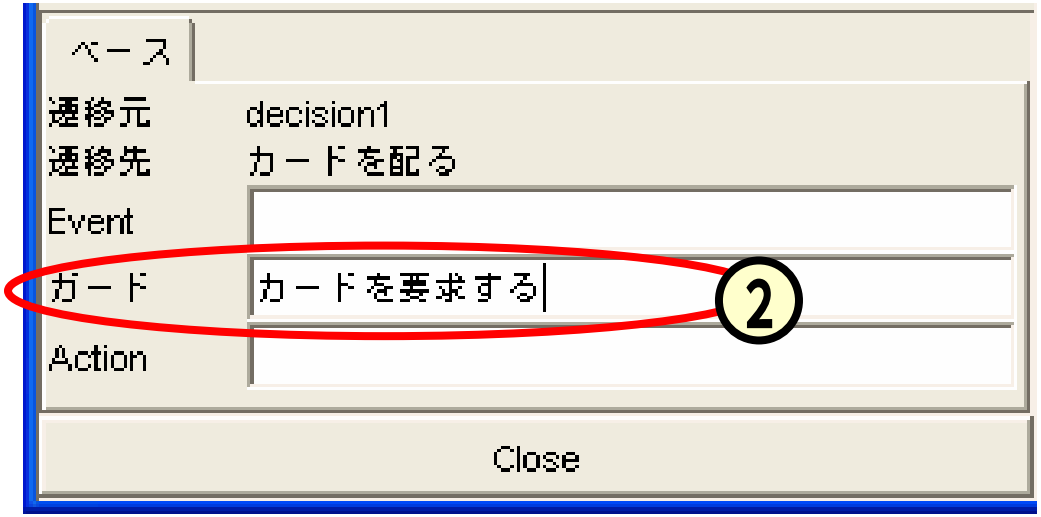
(5．5．12) ガード条件の記入

■トランジッションにガード条件を記入します。

■トランジッションをクリック (①) して選択します。

■詳細画面中で [ガード]の欄に入力 (②) します。



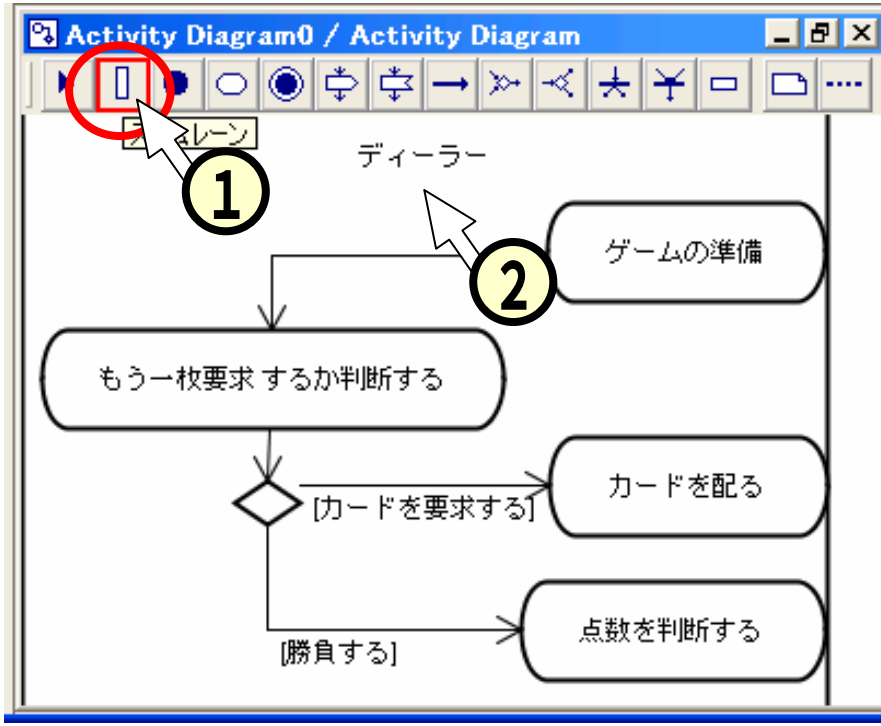


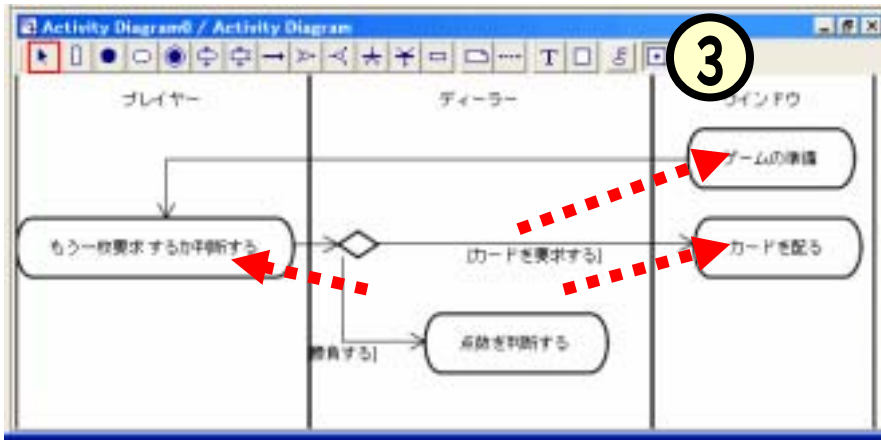
(5．5．13) スイムレーンの配置

■ツールバーの[スイムレーン]をクリック (①) して、編集画面上の適当な位置をクリック (②) します。

■スイムレーン要素が、名前の入力待ちの形で配置されます。ここで名前の入力が可能ですが、後から詳細画面中の“名前”にて変更することも出来ます。

■すでに描いた要素を選択して囲うようにスイムレーンを作成することは出来ません。まずスイムレーンを作成してから、アクティビティ要素などを再度配置 (③) します。





(5．5．14) スイムレーンの移動、サイズ変更

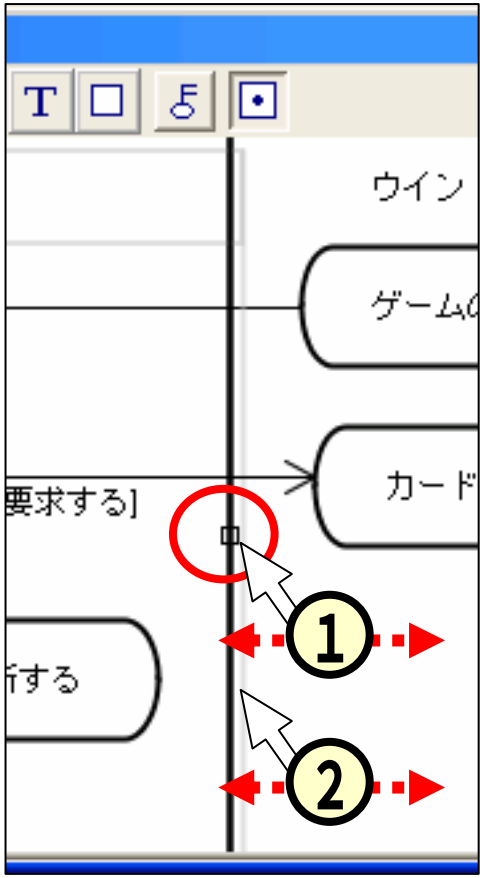
■スイムレーンの縦線をクリックします。

■サイズの変更

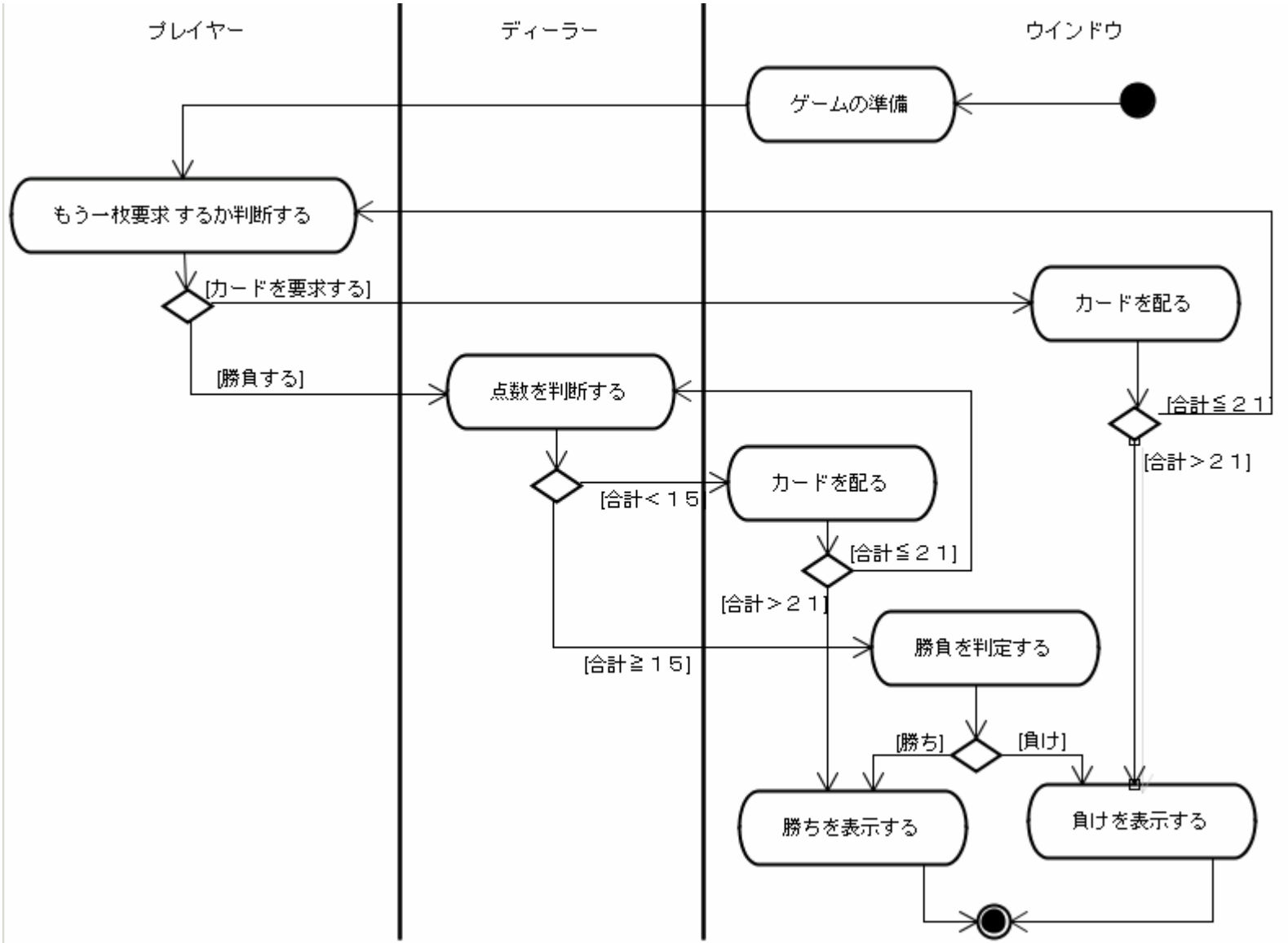
- 線上に現れた小さい四角をドラッグ(①)すると、サイズが変更できます。

■移動

- 小さい四角以外の部分の線をドラッグ(②)すると、スイムレーン自体が移動(③)します。



(5．5．15) Judeでのアクティビティ図



(5．5．16) 基本演習 (課題5－1)

■ (課題5－1) スライド(5.5.5)中のアクティビティ図を、Judeにて作成してください。

- “自炊”というスイムレーンに、全体を入れてください。
- Judeの起動方法および注意事項については、スライド「状態とデータ」(2.2)～(2.4)を参照してください。

(5．6) ブラックジャックのシーケンス図

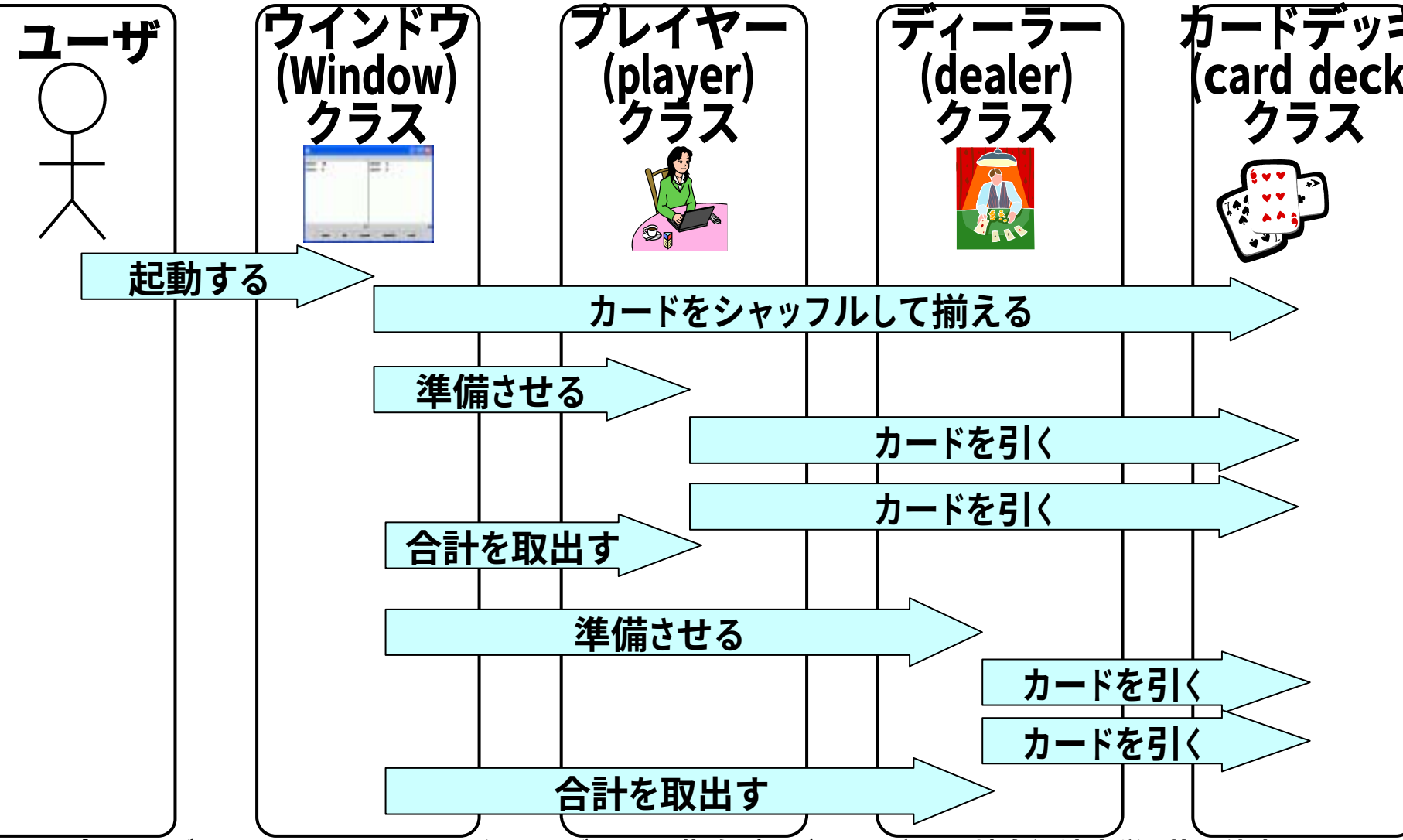
■シーケンス図については、以前、別スライドにて勉強しました。

■UMLでのシーケンス図は、クラス間のメッセージのやり取りを表現します。

■ここでは、シーケンス図とプログラムとの対応を見ていきます。

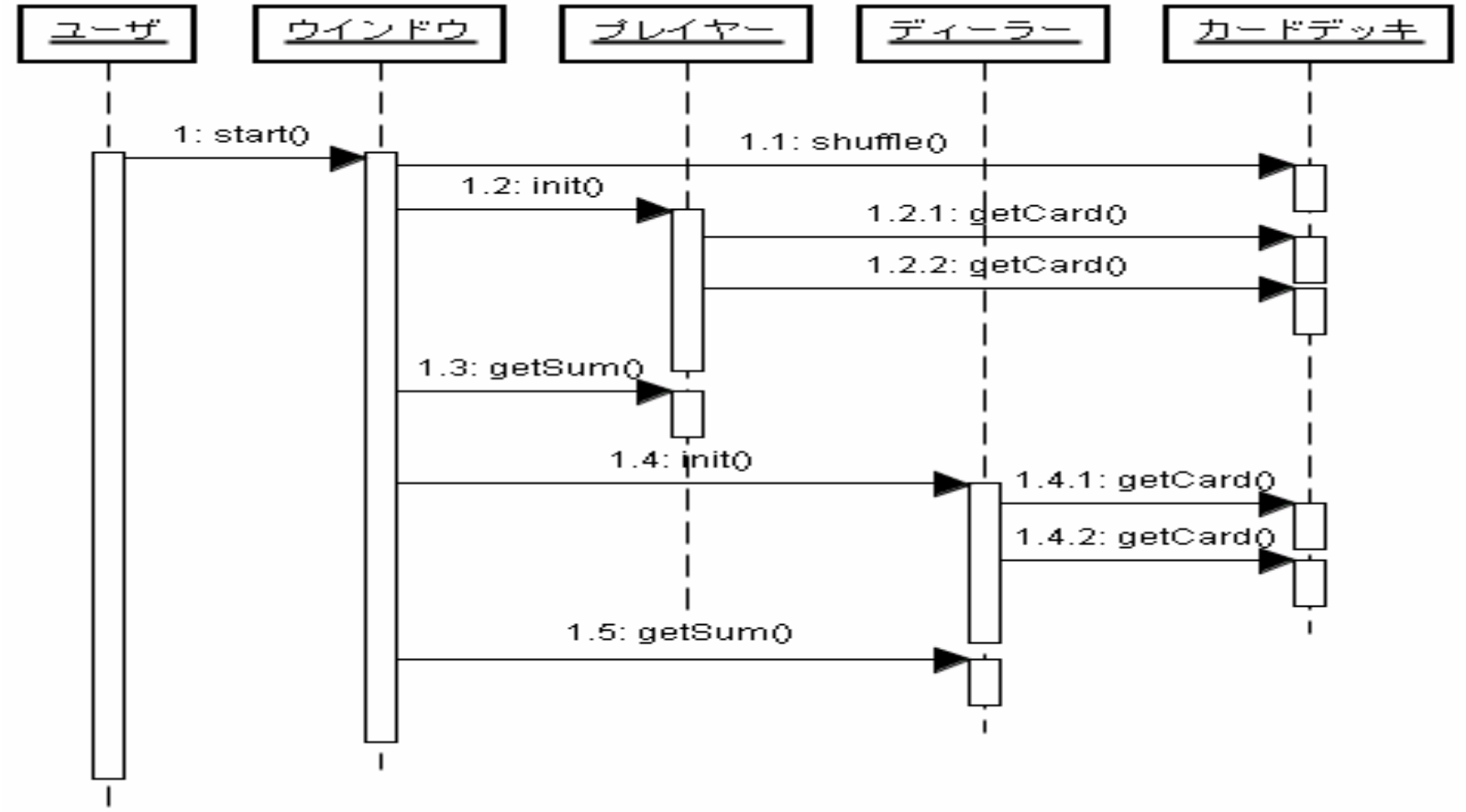
(5．6．5)ゲーム開始時の動作

■ゲームの開始時、プログラムは次のように動きます。



(5．6．6) 開始時のシーケンス図

■UMLのシーケンス図で表すと、次のようになります。



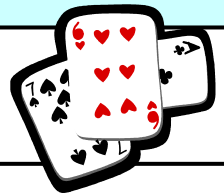
◆shuffle ： カードをシャッフルして揃える
◆init ： 準備させる　　◆getCard ： カードを引く
◆getSum ： 合計を取出す

(5．6．7) プログラムとの対応 (開始時)

```
# ウインドウクラス
class Window{
public Window(){
        :
myCardDeck.shuffle();
        :
        :
        :
        :
        :
        :
myPlayer.init();
myPlayer.getSum();
        :
}
```

```
# プレイヤークラス
class Player{
public void init(){
        :
addOne();
        :
}
public void addOne(){
        :
myCardDeck.getCard();
        :
}
public int getSum(){
        :
}
}
```

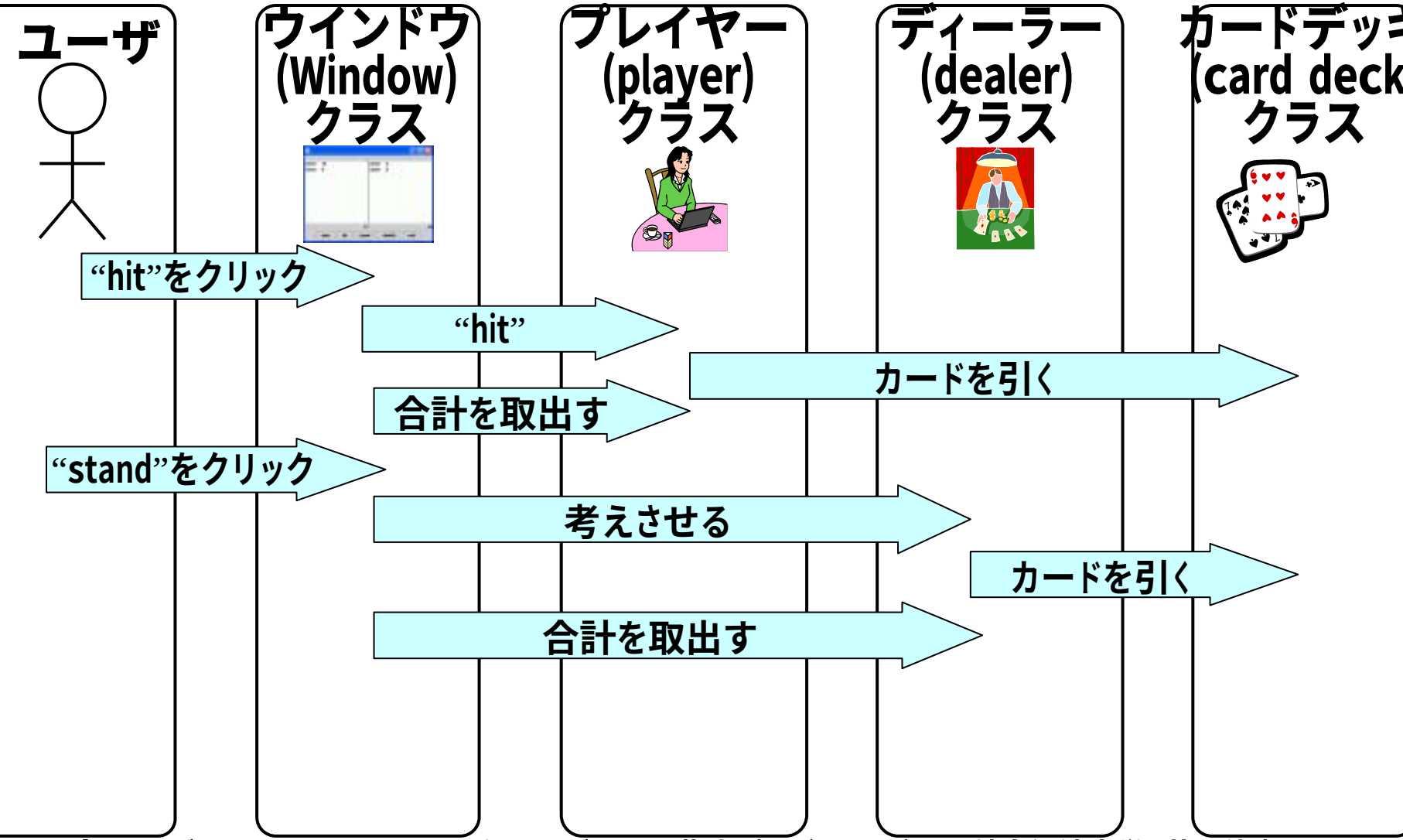
```
# カードデッキクラス
class cardDeck{
        :
public void shuffle(){
        :
}
public Card getCard(){
        :
}
}
```



※ ディーラークラスについては、(7) 章で説明します。

(5．6．8)ゲーム中終了までの動作

■プログラムを起動すると、次のように動きます。



(5．6．9) プログラムとの対応 (ゲーム中)

```
# ウインドウクラス
class Window{
        :
public void
ctionPerformed(){
        :
yPlayer.hit();
nt playerSum =
yPlayer.getSum();
        :
        :
        :
yDealer.seek();
nt dealerSum =
yDealer.getSum();
```

```
# プレイヤークラス
class Player{
        :
```

省略

```
}
```

```
# ディーラークラス
class Dealer{
        :
public void seek(){
        :
addOne();#結果として
        :
}
}
```

```
# カードデッキクラス
class cardDeck{
        :
public Card getCard(){
        :
}
}
```

■ディーラークラスの“getSum”については、(7) 章で説明します。

(5．7．1) 基本演習 (課題5－2、5－3)

- 演習1と同様に、“ブラックジャック”のプログラムを、井戸のネットワークドライブからダウンロードしてください。
- (課題5－2) ダウンロードしたプログラムを見て、スライド(5.6.7),(5.6.9)の内容を確認してください。
  - プログラム中にコメントで位置が示されています。
- (課題5－3) スライド(5.6.8)に対応するシーケンス図を作成してください。
  - シーケンス図の作成については、スライド「状態とデータ」(4.6.1)～(4.6.4)を参照してください。

(5．7．2) さいころゲーム:ゲームの説明

■次のゲームを考えます。

- ゲームを行う人が〃子〃、コンピュータ側が〃親〃となってゲームを始めます。
- 子と親との双方が、さいころを2個づつ持ちます。
- 子と親との双方が振ったさいころの出た目の合計が、7に近いほうが勝ちます。
- 合計と7との差が、子と親で同じ場合は、親の勝ちになります。
- まず、子も親もさいころを2個ずつ振ります。
- 次に、子は出た目を見て、2個のいずれかのさいころを振りなおすことが出来ます。振りなおさなくてもOKです。
- 次に、親は出た目を見て、2個のいずれかのさいころを振りなおすことが出来ます。振りなおさなくてもOKです。

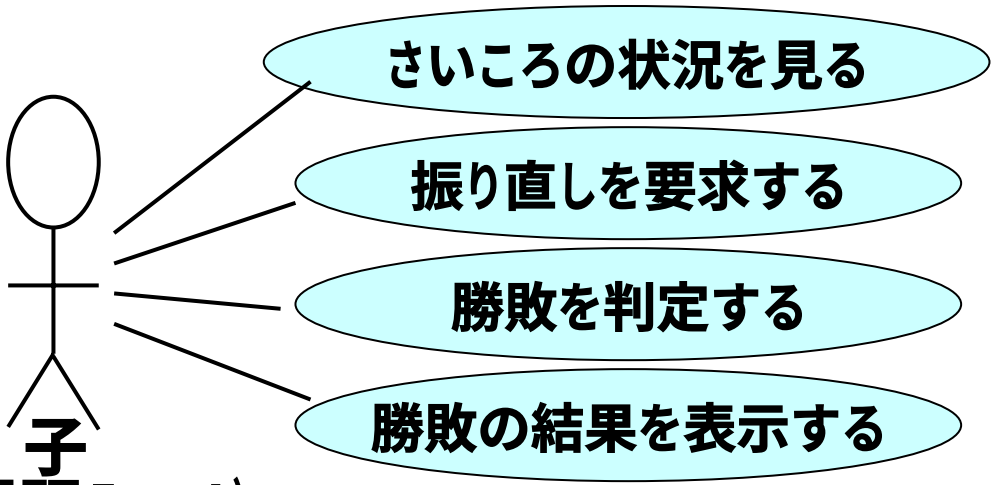
(5．7．3) 基本演習 (課題5－4，5－5)

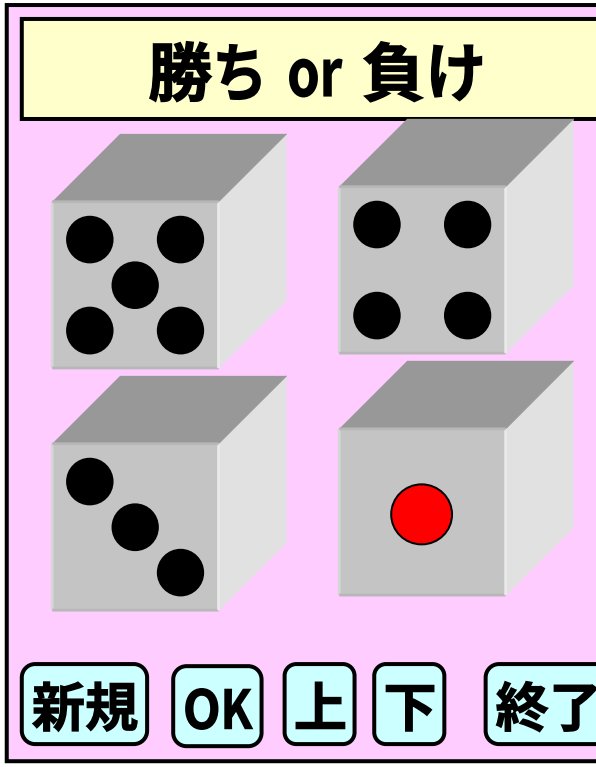
■右図のような操作画面を考えます。

■クラスとしては次の4つを考えます。

- 親、子、ウインドウ、さいころ

■次のようなユースケース図となります。





■ (課題5－4)

- アクティビティ図を作成してください。

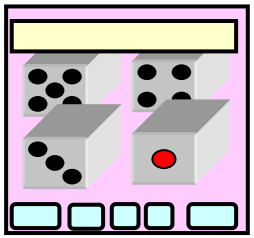
■ (課題5－5)

- シーケンス図を作成してください。

(5．7．4) 課題5－4：ヒント1

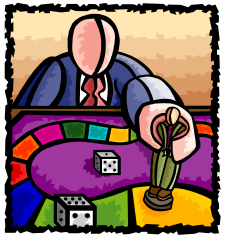
■クラスには、次のものを考えます。

ウインドウ (Window) クラス



子 (player) クラス

親 (dealer) クラス



さいころ (dice) クラス

■次スライド(5.7.5)に、未完成のアクティビティ図の描きました。完成させてください。

■シーケンス図について、どなたかホワイトボードに書いてみませんか？

(5．7．5) 課題5－4：ヒント2