Inheritance and Design Examples

CS193J Section: Friday, February 5

Blackjack

You're tired of hanging out in Sweet Hall and have decided to seek your fortune as a card shark and reap wealth and glory in Reno. The only problem is that you don't really know how to play Blackjack and need to get some air time in before you run off and squander your savings. You also need to test out various strategies to find out what might be a good one to put into practice.

Rules of Blackjack

Blackjack is played with an ordinary deck of 52 cards. Each card has a rank (Ace through King, or equivalently 1 to 13) and a suit (Spaces, Diamonds, Hearts, Clubs). In this game, a human player and various other computer players are pitted against a computer dealer. (The players don't play against each other; each player is simultaneously playing a one-on-one game against the dealer.) A round of Blackjack is played according to these rules:

1. The dealer gives each player two cards: one face down and one face up. Players can look at their own face-down card, but not other players' face-down card.

2. Each player's goal is to get more points than the dealer, but not to exceed 21 points. A player's points are determined by summing the contributions of their cards:
    a. Face cards (rank = 11, 12, and 13) are worth 10 points.
    b. An ace (rank = 1) is worth either 1 point or 11 points.
    c. All other cards are worth their face value.

3. After dealing out the starting cards, each player gets a turn to accumulate a hand as close as they can to 21 without going over. A player can draw additional cards one by one from the deck until they no longer want any more. If at any point the player's hand exceeds 21, they have "busted" and are done for this round.

4. After all of the players have had their turn and drawn as many cards as they want, the dealer draws his cards.

5. After all cards are drawn, the face-down cards are revealed and the dealer announces the winners and losers.

Betting

At the beginning of a round, before any cards are dealt, each player is asked how much they would like to bet on this round. If they win (i.e. beat the dealer), the player will double their money, if they lose, they lose the entire bet. If they tie or both bust, no money exchanges hands. The minimum bet is $1. Our casino is very generous with credit, so players can bet themselves deep into the hole if they like.

Players

Dealer There is always exactly one dealer per game. The dealer plays by the house rules. The dealer never bets any money of his own.

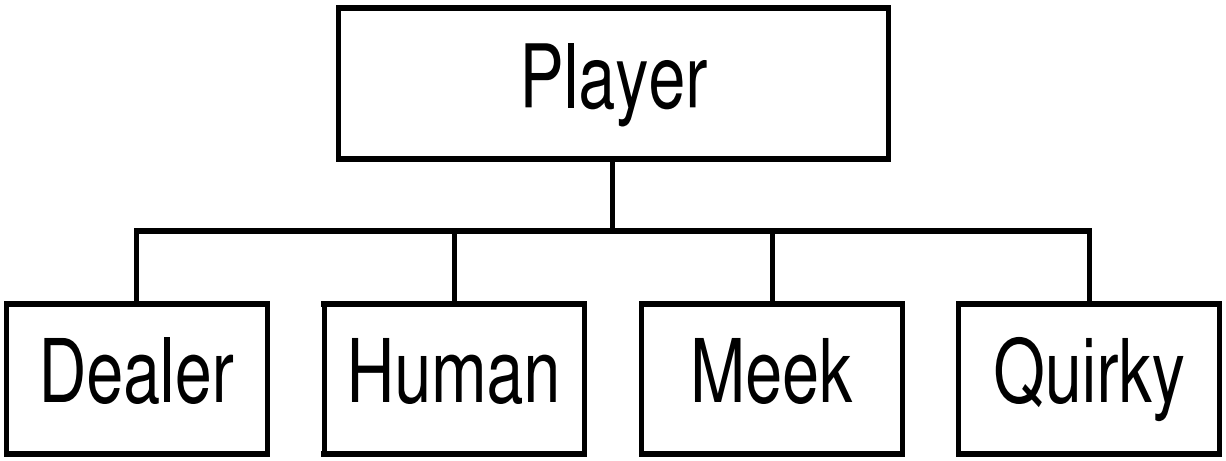
Human A human player makes choices by asking the user questions and getting answers using the IO.

Meek This computer player is a rather meek soul. He usually bets just $2 every round. If he wins three times in a row (ignoring ties), he doubles his current betting amount. The next time he busts (i.e. his card total goes over 21), he goes back to his $2 bet. He draws a card whenever he has an even number of points, he stays whenever he has an odd total.

Quirky This random computer player usually bets a random amount between $1 and $20. If he has 12-17 points, he always draws. When he has more than 18 points, he never draws. However, as one of his quirks, if he has the eight of hearts in his hand, no matter what his total, he never draws.

Classes

It is natural to come up with the following simple class hierarchy. We have an abstract base class Player, and four subclasses for the four types of players.



Player Data and Methods

int **money:** How much money a player has.

Vector **cards**: What cards a player has.

bet(): Bet some amount of money for a round of game.

takeTurn (): Take a turn in the game. Decide whether to stay or hit.

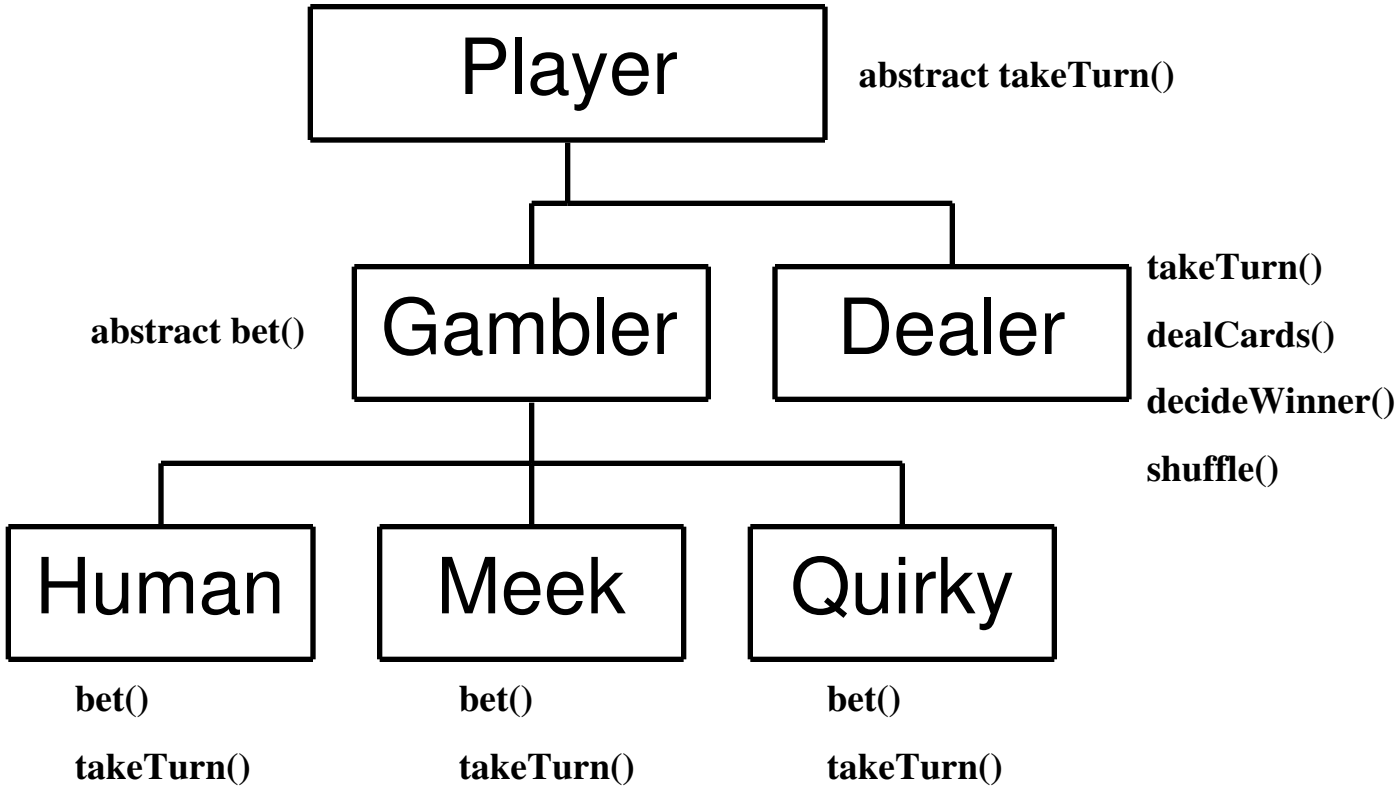
Dealer

For Dealer, we need some additional data and methods:

Deck **deck**:	The deck of cards being played.
dealCards():	Deal two cards to each player for a round.
decideWinner():	Decide the winners after each player plays. Dealer pays winners and collect money from losers.
reshuffle ():	Reshuffle the cards.

Also, note that dealer does not have to bet. We can see that dealer behaves quite different from other players.

New Class Hierarchy



Any Code Factoring?

Note: all players implement takeTurn(). Do they have any common code? Let's look at what takeTurn() may do.

```
void takeTurn() {

   //Each player has a different strategy
   decide whether to take a card;

   if(take) {  // All code down below is the
               // same for all players
      take a card;
      if(not busted)
         takeTurn();
   }
}
```

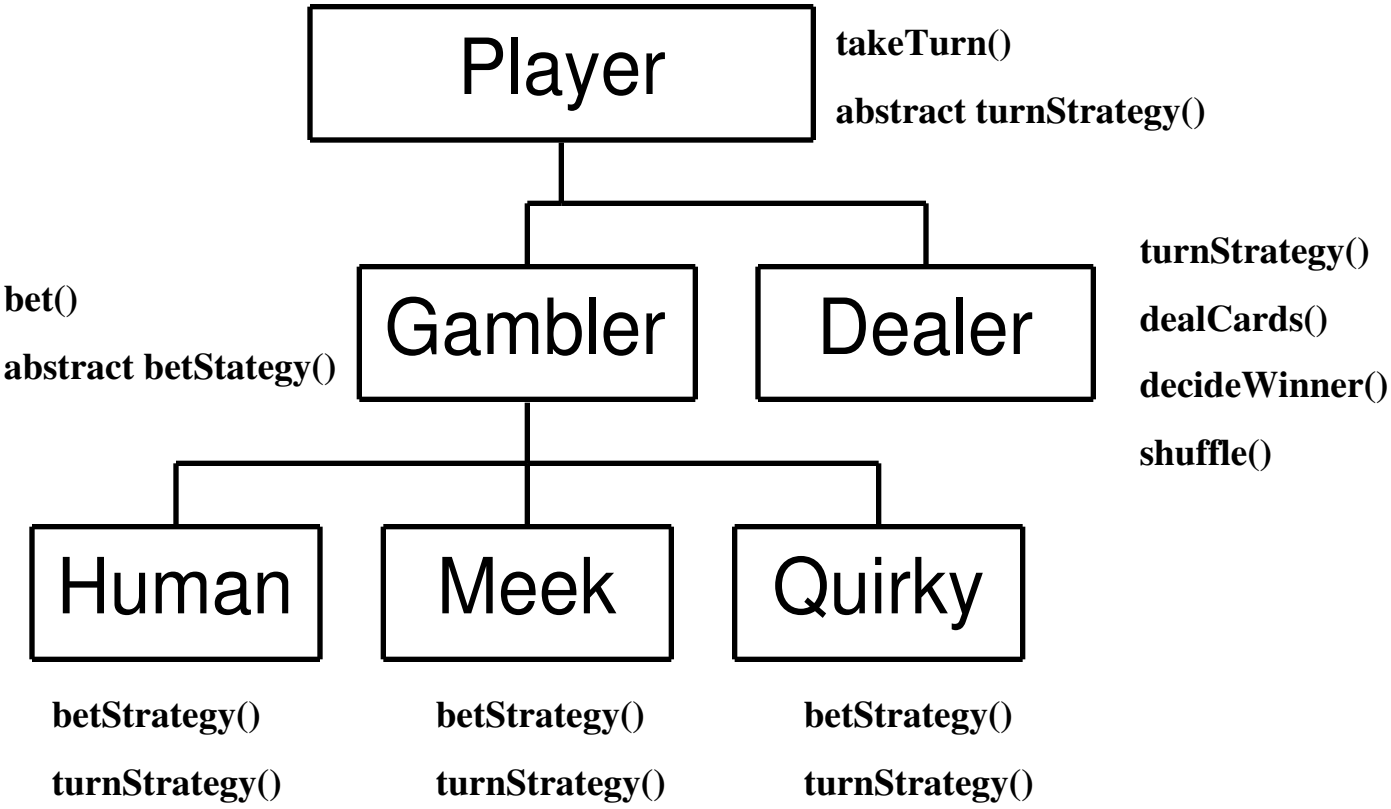
Factor takeTurn()

We have a new method **turnStrategy** () to decide whether to take a card in a turn. turnStrategy() is different for each playe takeTurn() calls turnStrategy(). Now takeTurn() is the same for all players, so it can be pushed up the class hierarchy. Similar factoring can also be applied to bet().

```
void takeTurn() {
   boolean take = turnStrategy(); // new method

   if(take) {
      take a card;
      if(not busted)
         takeTurn();
   }
}
```

Improved Class Hierarchy



Improved Class Hierarchy

In the improved class hierarchy, bet() is pushed up to the Gambler class, and takeTurn() is pushed up to the Player class. This way, we shared more common code between similar classes.

More Factoring?

Money can be transferred between different players in various ways. For instance, the dealer needs to give money to the winners in a round. Meek may randomly give money to other gamblers out of meekness. A new method **giveMoney**() will be necessary.

However, it is difficult to push **giveMoney**() up the class hierarchy. Neither Gambler nor Player is a good base class to implement **giveMoney**().

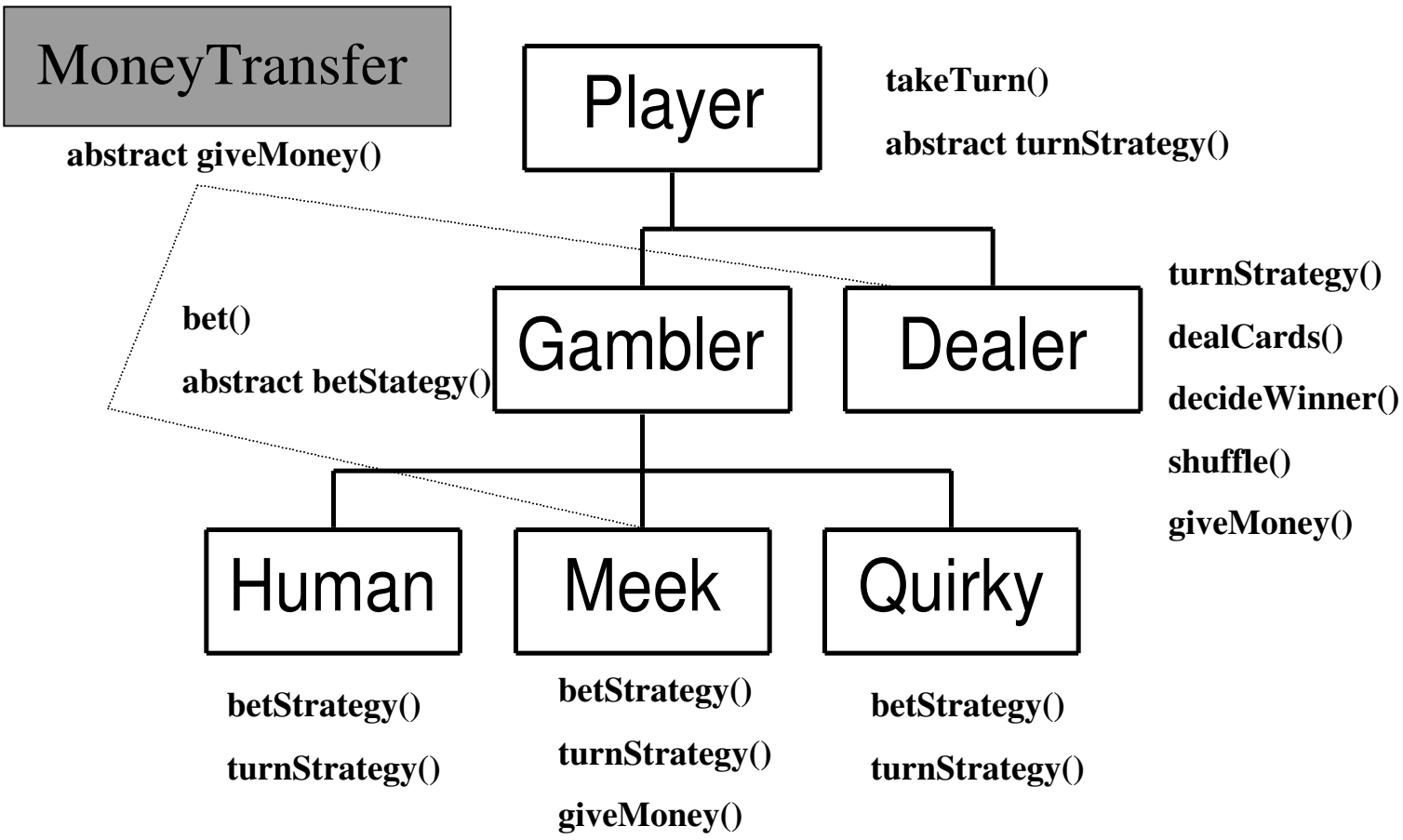
Sounds like a great opportunity to try an interface...

MoneyTransfer Interface

```
interface MoneyTransfer {
    public void giveMoney(Player who);
}
```

Both Dealer and Meek can implement giveMoney(), but no other players will need to. A Dealer gives money to the winners based on how much they bet. Meek gives some random amount of money to other Gamblers.

Improved Class Hierarchy(2)



Appendix: Class Declarations

Only the declarations of the classes mentioned above are included here. Some class variables are not shown.

```
interface MoneyTransfer {
   public void giveMoney(Player who);
}

abstract class Player {
   protected Vector cards;
   protected int money;

   public Player(int initialMoney) {
      money = initialMoney;
      cards = new Vector(10);
   }

   public abstract boolean turnStrategy(); // hit/stay?
   public void takeTurn(boolean hit) {
      // code to take a turn
   }
}
```

```
public class Dealer extends Player implements MoneyTransfer{
   private Deck deck; // deck of cards

   public void giveMoney(Player who) {
      // code to give a player money after she/he wins
   }

   public boolean turnStrategy() {
      // code to decide hit/stay for the dealer
   }

   public void dealCards() {
      // code to deal initial two cards to each player
   }

   public void decideWinner() {
      // code to decide winner of a round of game and
      // settle the bet
   }

   public void shuffle() {
      // code to shuffle the deck when necessary
   }
}
```

```
public abstract class Gambler extends Player {
   protected int betAmount;

   public abstract int betStrategy(); // return bet amount

   public void bet(int amount) {
      // code to bet
   }
}

public class Human extends Gambler {
   public int betStrategy() {
      // code to decide bet strategy(from input)
   }

   public boolean turnStrategy() {
      // code to decide whether to hit or stay(from input)
   }
}
```

```
public class Quirky extends Gambler {
   public int betStrategy() {
      // code to decide bet strategy
   }

   public boolean turnStrategy() {
      // code to decide whether to hit or stay
   }
}

public class Meek extends Gambler implements MoneyTransfer{
   public int betStrategy() {
      // code to decide bet strategy
   }

   public boolean turnStrategy() {
      // code to decide whether to hit or stay
   }

   public void giveMoney(Player who) {
      // code to give a player random amount of money
   }
}
```

Interface and Nested Class Example

CS193J Section: Friday, February 5

The SortableVector

One class which might have come in really handy for assignment #1 is a SortableVector class. As you probably have already discovered, there is no built-in class which has this behavior. But we will set out to remedy this situation in today's section…

Let's extend java.util.Vector to make our own SortableVector class!

Whose Responsibility is Sorting?

Let's say there is a client program which is using our SortableVector class to store a bunch of Objects. Where is it possible to put the responsibility of sorting?

The SortableVector: The SortableVector clearly has access to all the Objects in the vector, but has no idea how to compare these Objects or even what type of Objects they are.

The Objects: Surely the Objects themselves know what type they are, and could very well know how to compare themselves against other Objects.

The Client: The client also must know what type of objects are being stored and how to compare them.

Object Responsibility

First let's try giving the objects themselves the responsibility of comparison. We could make all of our objects implement a Comparable interface:

```
public interface Comparable {

        public bool lessThan(Object o);

}
```

As long as all the objects in the SortableVector implement this interface, the vector could call lessThan(…) on any pair of objects, and thus sort the whole vector!

What are some problems with this approach?

Problems With This Approach

- What if we want to store objects of some built-in class in a SortableVector?
- How do we know what implementation to use for lessThan()? Specifically, what if some times we want to sort by last name, but some times we want to sort by birth date?

Client Responsibility

What about making comparison the responsibility of the client? We could have the client pass an object which implements a Comparison interface to the SortableVector!

```
public interface Comparison {

    //returns true if o1 < o2

    public bool lessThan(Object o1, Object o2);

}
```

The client is basically just specifying a call-back method, much like what happens when you pass a function ptr to qsort in C.

SortableVector Class

Here's a very rough outline of our SortableVector class:

```
public class SortableVector extends Vector {

        // some constructors would go here

        public void sort(Comparison cmp) {
                // insert your favorite sorting
                // algorithm here
        }
}
```

Client Use -- Nested Class

Let's say we have a SortableVector of Student objects, and want to sort them by last name:

```
class StudentDB {
   void sortByLastName(SortableVector vector) {
      vector.sort(new LastNameComparison());
   }

   private static class LastNameComparison implements Comparison {
      public bool lessThan(Object o1, Object o2) {
         //check to make sure o1 and o2 are actually Students.
         String lastName1 = ((Student)o1).lastName();
         String lastName2 = ((Student)o2).lastName();
         return (lastName1.compareTo(lastName2) < 0);
      }
   }
}
```

Client Use 2 -- Anonymous Class

Here is the same basic code, but this time using an anonymous class:

```
class StudentDB {
   void sortByLastName(SortableVector vector) {
      vector.sort(new Comparison() {
         public bool lessThan(Object o1, Object o2) {
            return ((Student)o1).lastName().compareTo(
                   ((Student)o2).lastName() ) < 0;
            }
      });
   }
}
```

Can be pretty handy sometimes, but don't over use it!! The syntax is kind of strange, and your code can very quickly become unreadable.

Sorting Students by studentID

```
class StudentDB {
   //sortByLastName() and other existing methods are here

   void sortByStudentID(SortableVector vector) {
      vector.sort(new StudentIDComparison());
   }
   private static class StudentIDComparison implements Comparison {
      public bool lessThan(Object o1, Object o2) {
         //check to make sure o1 and o2 are actually Students.
         long studentID1 = ((Student)o1).studentID();
         long studentID2 = ((Student)o2).studentID();
         return (studentID1 < studentID2);
      }
   }
}
```