Software Design

Suppose you are designing and implementing a simplified version of one of your favorite games, Blackjack, where a single player plays against an automated dealer.

Here is a short description of the game:

> The objective of Blackjack is to obtain a score higher than the dealer. In each round the player and the dealer are initially dealt 2 cards, with one of the dealer's cards dealt face down. Each numerical card (2-9) is worth its face value, aces are worth either 1 or 11 (it is the player's choice), and all face cards (ten, jack, queen, and king) are worth 10.
>
> At the beginning of each round the player must place a bet; this amount will not change for the rest of the round. After the bet, the player repeatedly decides whether to *hit* (be dealt a new card, up to 5 cards in the player's hand) or *stand* (stop being dealt cards). If the player scores more than 21 points, the player loses her bet and the round.
>
> After the player stands, the dealer exposes her face-down card and must hit (be dealt cards) until the value of her cards is more than 16. If the player's score is more than the dealer's final score (but less than or equal to 21) then the player wins an amount equal to her bet. The player also wins if she is dealt 5 cards worth 21 or fewer points. Otherwise, the player loses her bet.
>
> More information can be found on Wikipedia's Blackjack page.

Object-Oriented Analysis

Build a vocabulary by creating a *domain model* for this problem. Document all relevant concepts you may need and include important attributes and associations. (Remember, a domain model is used to analyze the problem; it describes concepts and abstractions of the real world, not software classes.)

Object-Oriented Design

System Sequence Diagram

In software engineering, a *system sequence diagram* is a model that shows, for one scenario of use, the sequence of events that occur on the system's boundary or between subsystems.

A system sequence diagram should specify and show the following:

1. External actors
2. Messages (methods) invoked by these actors
3. Return values (if any) associated with previous messages
4. Indication of any loops or iteration area

To start designing a solution, think about how the game flows from an initial action (e.g., a user clicking a button, or part of the system requesting some information) to the individual steps following that action. Create a *system sequence diagram* to model the interactions of a player with the game of Blackjack. Carefully consider all the interactions a player can make with the system (e.g., the player takes a hit or the player makes a bet).

Interaction Diagrams

To start designing a solution, consider the scenarios listed below. For each scenario below create an *interaction diagram* (using a UML sequence diagram) that models the interaction among objects in that scenario. Note: To help distinguish this artifact from system sequence diagrams, we sometimes call this an *o*bject-level interaction diagram.

The scenarios are:

1. Determine the number of points of the player's current hand.
2. Initialize a new round with a given bet, which includes updating scores and dealing cards.
3. During the game, the player decides to take the 'hit' action, which includes dealing an additional card, checking whether the game is over, possibly updating scores, and so forth.

Object Model

Now that you are more familiar with Blackjack's concepts and behaviors, create an *object model* describing the software classes of the system as a UML class diagram. Consider the design goals you have seen in well-designed software to guide and justify your design. Try to assign responsibility to classes to create low coupling and high cohesion in your system. If you encounter nontrivial design decisions, carefully consider those decisions and explore alternatives for them. (Object models are closer to a possible implementation and assign responsibilities to classes. The methods in the object model should correspond to the methods called in interaction diagrams.)