CSE 331

Object-Oriented Design Heuristics

slides created by Marty Stepp
based on materials by M. Ernst, S. Reges, D. Notkin, R. Mercer
http://www.cs.washington.edu/331/

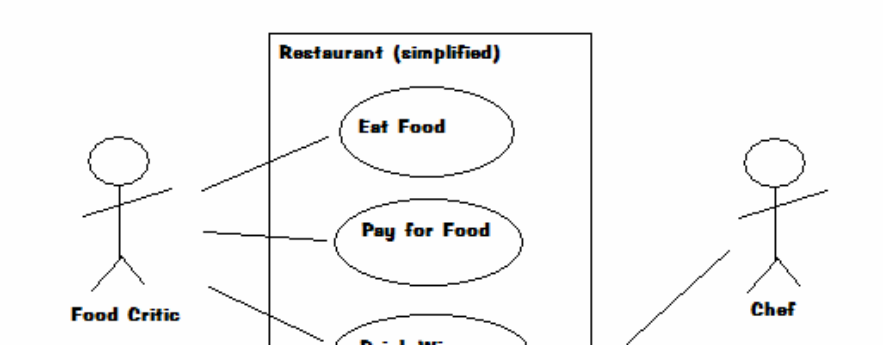
Design phase

- During the **design phase**, we identify classes, responsibilities, and collaborations between objects in our system.

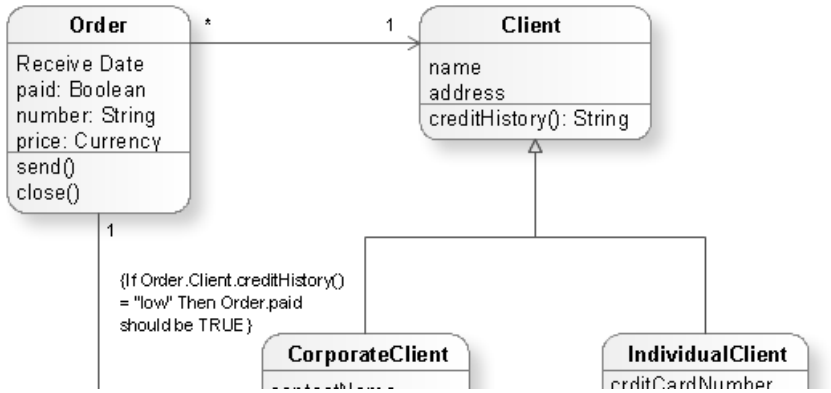
- Possible artifacts produced during design phase:
  - text descriptions of classes and their responsibilities
  - diagrams of relationships between classes
  - diagrams of usage scenarios
  - diagrams of a particular object's state over its lifetime

Design diagrams

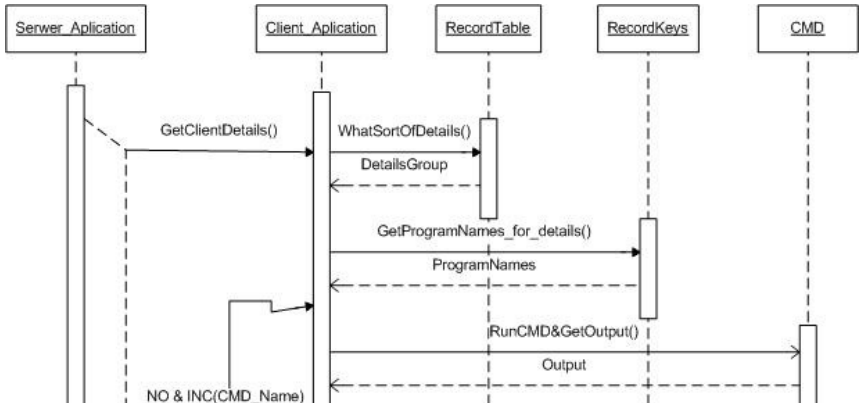
- **use case diagram**: Shows major ways system will be used, and actors who use it.



- **class diagram**: Shows all classes in the system and their state/ behavior, connections to others.



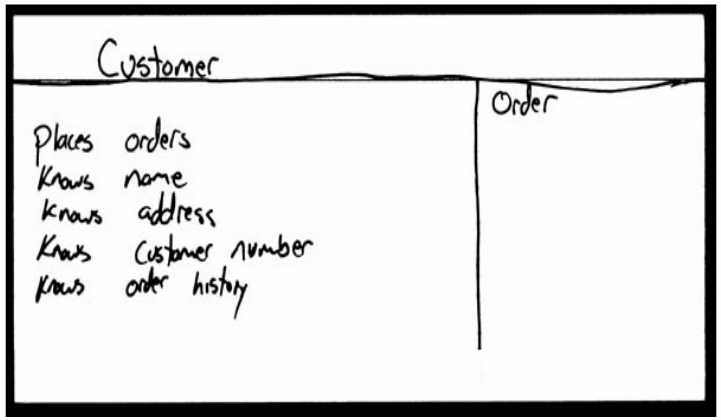
- **sequence diagram**: Shows the messages passed between objects to accomplish a task.



How do we design classes?

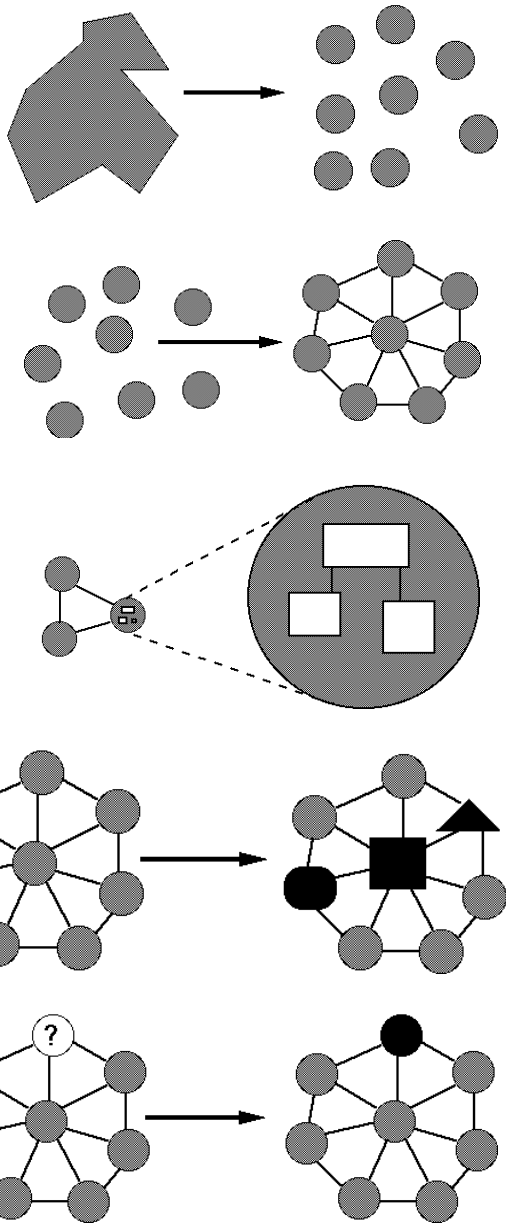
- Class identification from project spec / requirements:
  - nouns are potential classes, objects, fields
  - verbs are potential methods or responsibilities of a class

- **CRC cards** are one way to brainstorm classes in a system.
  - Write down classes' names on index cards:
  - Next to each **class**, list the following:
    - **responsibilities**: Problems to be solved; short verb phrases.
    - **collaborators**: Other classes that are sent messages by this class (asymmetric)



Qualities of modular software

- decomposable
  - can be broken down into pieces
- composable
  - pieces are useful and can be combined
- understandable
  - one piece can be examined in isolation
- has continuity
  - reqs. change affects few modules
- protected / safe
  - an error affects few other modules



Heuristics 2 quick reference

- **Heuristic 2.1**: All data should be hidden within its class.
- **Heuristic 2.2**: Users of a class must be dependent on its public interface, but a class should not be dependent on its users.
- **Heuristic 2.3**: Minimize the number of messages in the protocol of a class.
- **Heuristic 2.4**: Implement a minimal **public interface** that all classes understand.
- **Heuristic 2.5**: Do not put implementation details such as common-code private functions into the public interface of a class.
- **Heuristic 2.6**: Do not clutter the public interface of a class with items that users of that class are not able to use or are not interested in using.
- **Heuristic 2.7**: Classes should only exhibit nil or export **coupling** with other classes, that is, a class should only use operations in the public interface of another class or have nothing to do with that class.
- **Heuristic 2.8**: A class should capture one and only one key **abstraction**.
- **Heuristic 2.9**: Keep related data and behavior in one place.
- **Heuristic 2.10**: Spin off non-related behavior into another class (i.e., non-communicating behavior).
- **Heuristic 2.11**: Be sure the abstractions that you model are classes and not simply the roles objects play.

Poker design question

- Poker `Deck` class stores a list of cards; the game needs to be able to shuffle and draw the top card.
  - We give the `Deck` class the following methods: `add(Card)`, `add(index, Card)`, `getCard(int)`, `indexOf(Card)`, `remove(index)`, `shuffle()`, `drawTopCard()`, `isEmpty()`, `set(index, Card)`, etc.
    - What's wrong with this design?

  - **Heuristic 2.3**: Minimize the # of messages in the protocol of a class.
  - **Heuristic 2.5**: Do not put implementation details such as common-code private functions into the public interface of a class.
  - **Heuristic 2.6**: Do not clutter the public interface of a class with items that users of that class are not able to use or are not interested in using.

Minimizing public interface

- Make a method private unless it needs to be public.
- Supply getters (not setters) for fields if you can get away with it.
  - example: `Card` object with rank and suit (`get`-only)

- Be mindful of the design of "collectiony" classes.
  - **"data structure wrapper" anti-pattern**: If a class does nothing more than wrap a collection's methods, maybe you should just use that collection and forego the class entirely.
  - In a class that stores a data structure, don't replicate that structure's entire API; only expose the parts clients need.
  - example: If `PokerGame` has an inner set of `Players`, supply just an `iterator` or a `getPlayerByName(String)` method.

Poker design question

- Proposed fields in various poker classes:
  - A `Hand` stores 2 cards and the `Player` whose hand it is.
  - A `Player` stores his/her `Hand`, last bet, a reference to all the other `Player`s in the game, and a reference to all past betting rounds.
  - The `PokerGame` stores an array of all `Player`s, the `Deck`, and an array of all players' last bets.

  - What's wrong with this design?

Cohesion and coupling

- **cohesion**: how complete and related things are in a class

  (a good thing)

- **coupling**: when classes connect to / depend on each other

  (too much can be a bad thing)

  - **Heuristic 2.7**: Classes should only exhibit nil or export coupling with other classes; that is, a class should only use operations in the public interface of another class or have nothing to do with that class.
    - (in other words, minimize unnecessary coupling)

Reducing coupling

- combine 2 classes if they don't represent a whole abstraction
  - example: `Bet` and `PlayRound`

- make a coupled class an inner class
  - example: list and list iterator; binary tree and tree node
  - example: GUI window frame and event listeners

- provide simpler communication between subsystems
  - example: provide methods (`newGame`, `reset`, ...) in `PokerGame` so that clients do not need to manually refresh the players, bets, etc.

Poker design question

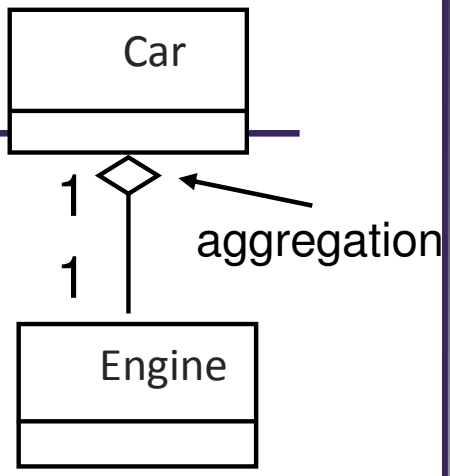
- Example: A poker Player needs to draw cards from a Deck. Should a Player object store a Deck object as a field?

- Q: If objects of class A "sort of relate to" objects of class B, but not for most of the class's behavior, then:
  - Should the A actually store a reference to the B?

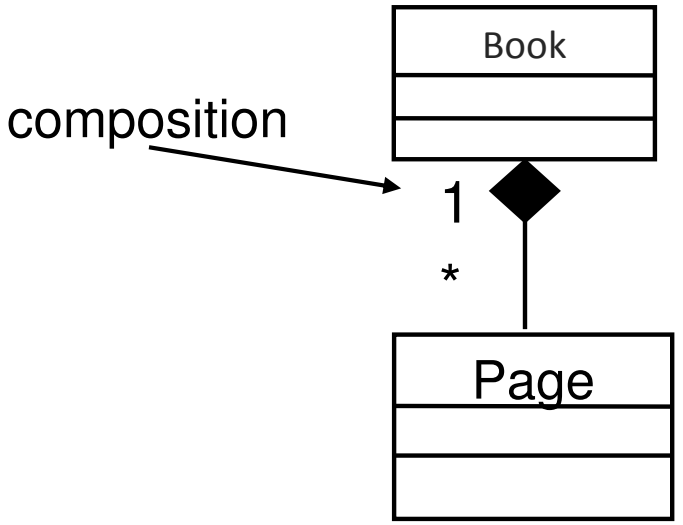
- A: If possible, no. Alternatives:
  - A can store some kind of *key* (name, ID #) that could later allow a person to look up the B.
  - An outside source can pass the B to A as a *parameter* when needed.
    - Turns an *aggregation* into a *dependency*.

Class relationships

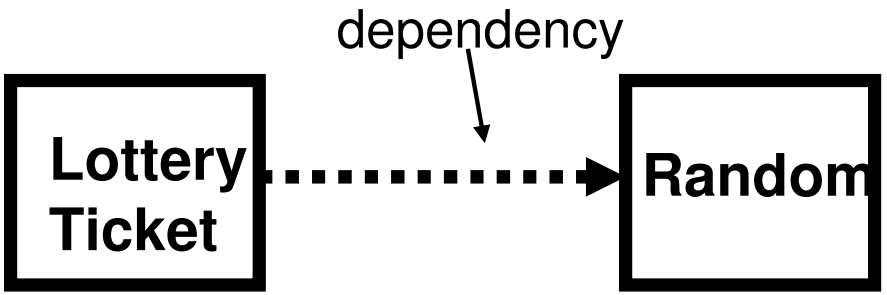
- **aggregation**: "is part of"
  - A field that is an important sub-part.



- **composition**: "is entirely made of"
  - A field that is the true essence of the state of the object containing it.
  - The parts live and die with the whole.
  - Stronger than aggregation.



- **dependency**: "uses temporarily"
  - Often a parameter, not a field.
  - Usually an implementation detail, not an intrinsic part of that object's permanent state



Heuristics 3 quick reference

- **Heuristic 3.1**: Distribute system intelligence horizontally as uniformly as possible, that is, the top-level classes in a design should share the work uniformly.
- **Heuristic 3.2**: Do not create **god classes**/objects in your system. Be very suspicious of a class whose name contains Driver, Manager, System, or Subsystem.
- **Heuristic 3.3**: Beware of classes that have many accessor methods defined in their public interface.
- **Heuristic 3.4**: Beware of classes that have too much **noncommunicating behavior**.
- **Heuristic 3.5**: In applications that consist of an object-oriented model interacting with a user interface, the **model** should never be dependent on the interface.
- **Heuristic 3.6**: Model the real world whenever possible.
- **Heuristic 3.7**: Eliminate irrelevant classes from your design.
- **Heuristic 3.8**: Eliminate classes that are outside the system.
- **Heuristic 3.9**: Do not turn an operation into a class. Be suspicious of any class whose name is a verb or is derived from a verb, especially those that have only one piece of meaningful behavior (don't count set, get, print).
- **Heuristic 3.10**: **Agent classes** are often placed in the analysis model of an application. During design time, many agents are found to be irrelevant and should be removed.

Poker design question

- Q: Should the PokerGame contain a list of the Players, or should each Player contain a field that refers to the PokerGame? Or both?

- A: The players should not have a reference to the game. Player is the "smaller" or "contained" object, and the relationship should go inward from large to small.

  - **Heuristic 4.1**. If a class contains objects of another class, then the containing class should be sending messages to the contained objects, not vice versa.

  - **Heuristic 4.2**. A class must know what it contains, but should never know who contains it.

Client dependency

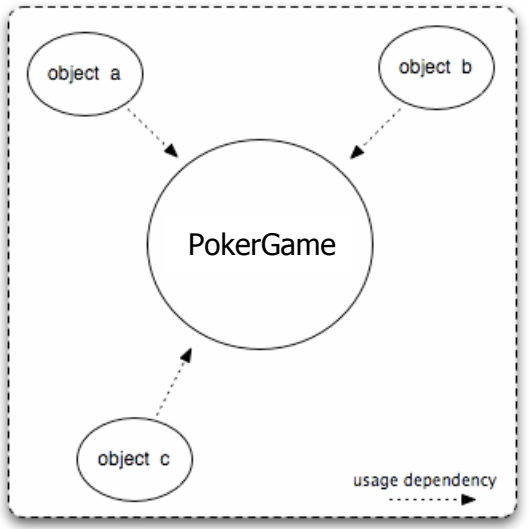
- Q: My PokerGame has an update method that needs to be called after each betting round to make sure the right amount of money is in each player's hand and on the table. The client must call nextBetRound and then call update once, in that order.
  - What is wrong with this design?

- A: A class should not depend on its clients.
  - If possible, don't depend on your methods being called in a certain order, or being called exactly a certain number of times.
  - e.g. HW2 Schedule shouldn't depend on GUI stopping at 5:30pm.

Poker design question

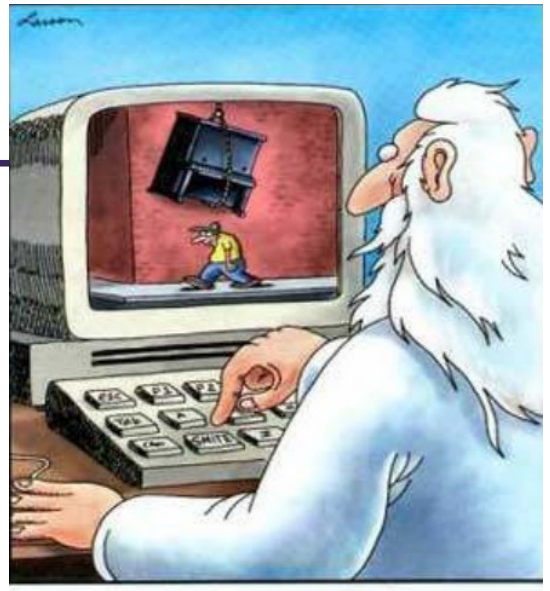
- Our `PokerGame` class:
  - stores all the players
  - stores an array of cards representing the card deck
  - stores all bets and money
  - does the logic for each betting round
  - performs the AI for each computer player's moves

  - What's wrong with this design?



God classes

- **god class**: a class that hoards too much of the data or functionality of a system.



God at His computer

- **Heuristic 2.8**: A class should capture one and only one key abstraction.
- **Heuristic 3.2**: Do not create god classes in your system. Be suspicious of a class whose name contains Driver, Manager, System, or Subsystem.
- **Heuristic 3.4**: Beware of classes that have too much non-communicating behavior, that is, methods that operate on a *proper subset* of the data of a class. God classes often exhibit much non-communicating behavior.
- **Heuristic 4.2**. Classes should not contain more objects than a developer can fit in his or her short-term memory. A favorite value for this number is six.

Design question

- Q: A player may bet only as much $ as they have; and if a prior player has made a "call", the current player cannot raise.
  - Where should these policies be enforced?

- Q: If we are writing a course registration system, students can only enroll in courses if they have taken all prerequisites for the course. Which class should check and enforce this rule? Student? Course? CourseOffering? Instructor? RegistrationSystem? ...

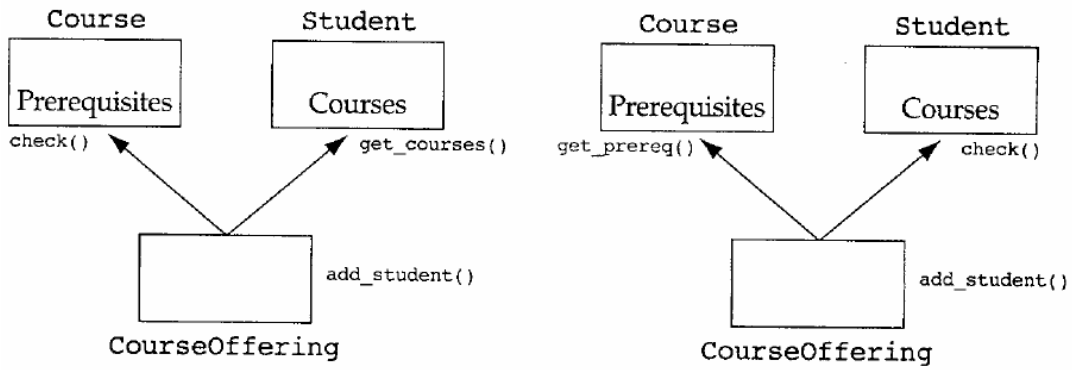


Figure 3.5 Two designs for checking student prerequisites.

Possible designs

- Poker Design 1: `Player` class remembers whether that player is in the game, what the current bet is, whether it is his turn, etc.
  - `Player` checks whether a "call" has been made.
  - `Player` checks whether he/she has enough to make a given bet.

- Poker Design 2:
  - `PokerGame` class remembers who is in the game.
  - `Betting` class remembers every player's current bets, checks $.
  - `Dealer` class remembers whose turn it is.

- Which is better? Is there a third option?

Related data and behavior

- "**policy**" behavior should be where that policy is enforced/enacted.
  - Make the class with the most relevant data enforce the policy.
  - **Heuristic 2.9**: Keep related data and behavior in one place.
    - avoids having to change two places when one change is needed
  - **Heuristic 3.3**: Beware of classes that have many accessor methods ... [This] implies that related data+behavior are not kept in one place.

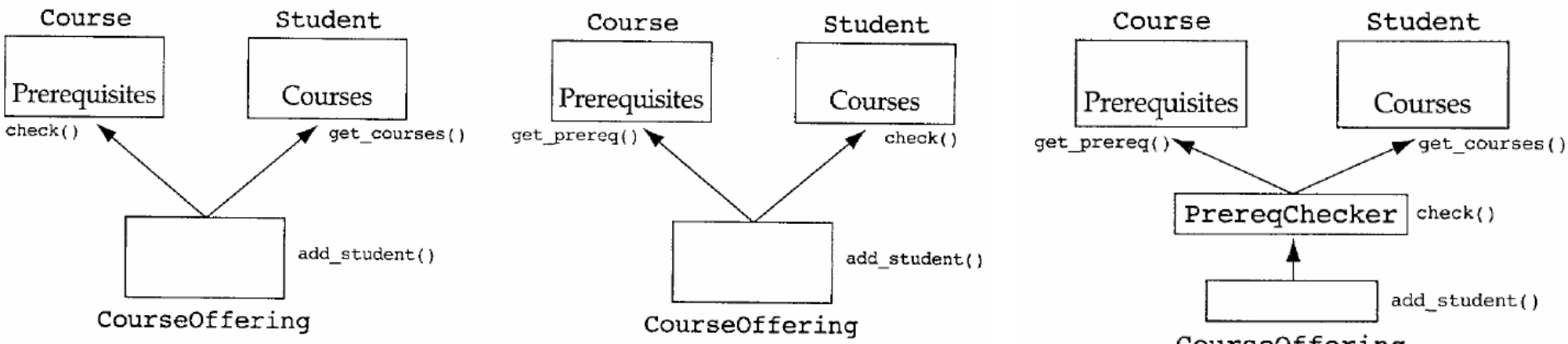
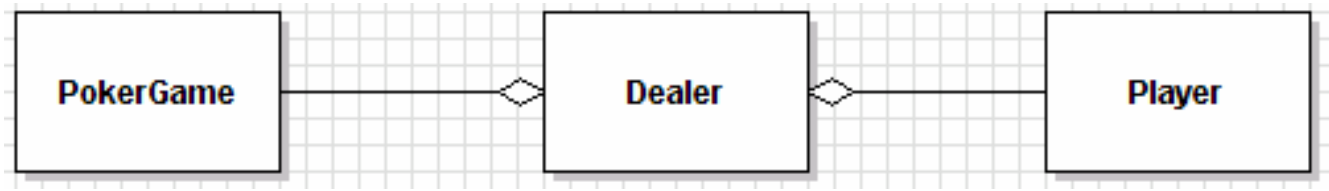


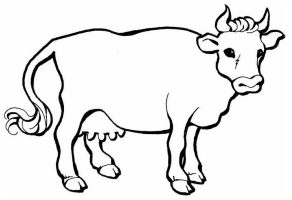
Figure 3.5 Two designs for checking student prerequisites.

Poker design question 4

- Each new game round, the `PokerGame` wants to deal cards to each player. During the game, players draw additional cards.
  - We will create a `Dealer` class that the `PokerGame` asks to deal the cards to every player.
  - `Player` objects will store a reference to the `Dealer`.
    During the game, they will talk to the dealer to draw their cards.
    The `Dealer` will notify the Game once all players have drawn.
  - What's wrong with this design?



Agent classes



- **agent class**: a class that acts as a middle-man to help two or more other classes communicate.
  - example: Farmer class to link Cow and Milk classes
  - example: Librarian class to link Book and Shelf classes

  - **Heuristic 3.10**: Agent classes are often placed in the analysis model of an application. During design time, many agents are found to be irrelevant and should be removed.

  - What defines whether an agent is relevant?
    - A relevant agent must have some other behavior beyond simply being a middle-man; it must have some useful purpose of its own as well.

Poker design question

- Cards belong to one of four suits. So we have created classes `Club`, `Diamond`, `Heart`, `Spade` class to represent each suit.

- In each game round, one player is the dealer and one is the first better. Also each turn there is a next better waiting. So we have created classes `Dealer`, `NextBetter`, `FirstBetter`.

- Every game has several betting rounds, each round consisting of several bets. So we have created classes `Bet` and `CurrentBettingRound`.

- What's wrong with this design?

Proliferation of classes

- **proliferation of classes**: too many classes that are too small in size/scope; makes the system hard to use, debug, maintain
  - **Heuristic 2.11**: Be sure the abstractions that you model are classes and not simply the roles objects play.
  - **Heuristic 3.7**: Eliminate irrelevant classes from your design.
    - often have only data and `get`/`set` methods; or only methods, no real data
  - **Heuristic 3.8**: Eliminate classes that are outside the system.
    - don't model a Blender just because your company sells blenders;
      don't necessarily model a User just because the system is used by somebody
  - **Heuristic 3.9**: Do not turn an operation into a class.
    - Be suspicious of any class whose name is a verb, especially those that have only one piece of meaningful behavior. Move the behavior to another class.

Design question

- Q: If my object's data comes primarily from a file, should my constructor take the file as a parameter?
  - In other words, it's one thing to decide that the object stores a certain collection of data. But how does that data get put into the object?
- Q: Does the object need all of that data given to it initially, at the start of its lifetime?
  - Or can it be passed in incrementally over time?
- A: Constructors should almost never accept files as parameters.
  - Parsing a file is a "heavy-duty" operation; constructors are light.
  - Instead, make a (static?) producer method that parses the file, then passes the data (now a data structure) to your object.