CSE 403
Lecture 7

UML Class Diagrams

Reading:
UML Distilled, *Ch. 3,* M. Fowler

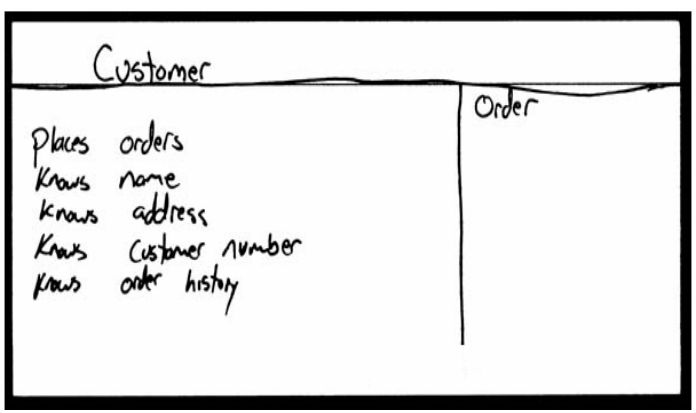
slides created by Marty Stepp
http://www.cs.washington.edu/403/

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 card exercises
  - 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)

- UML
  - class diagrams (today)
  - sequence diagrams
  - ...



Introduction to UML

- **Unified Modeling Language (UML)**: depicts an OO system
  - programming languages are not abstract enough for OO design
  - UML is an open standard; lots of companies use it
    - many programmers either know UML or a "UML-like" variant

- UML is ...
  - a *descriptive* language: rigid formal syntax (like programming)
  - a *prescriptive* language: shaped by usage and convention

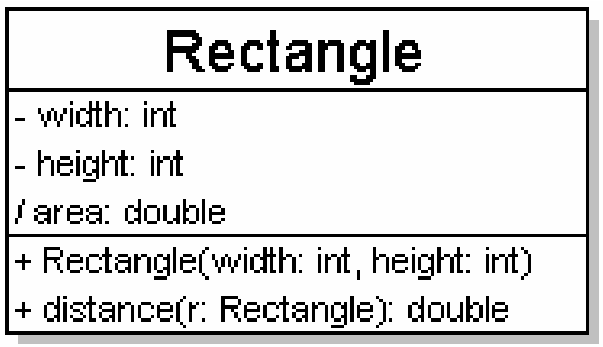
  - UML has a rigid syntax, but some don't follow it religiously
  - it's okay to omit things from UML diagrams if they aren't needed by team/supervisor/instructor

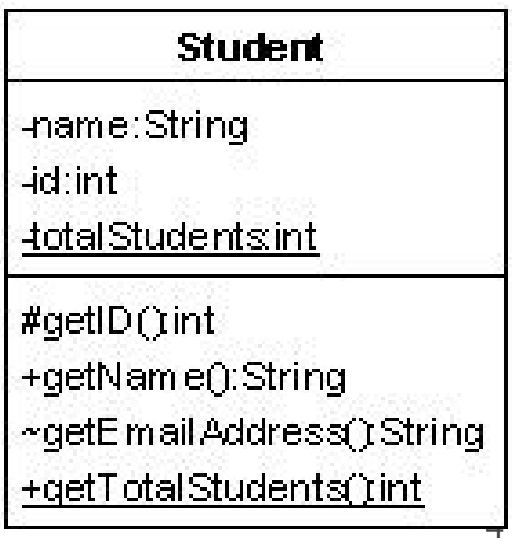
Diagram of one class

- **class name** in top of box
  - write **<<interface>>** on top of interfaces' names
  - use *italics* for an *abstract class* name

- **attributes**
  - should include all fields of the object
  - also includes derived "properties"

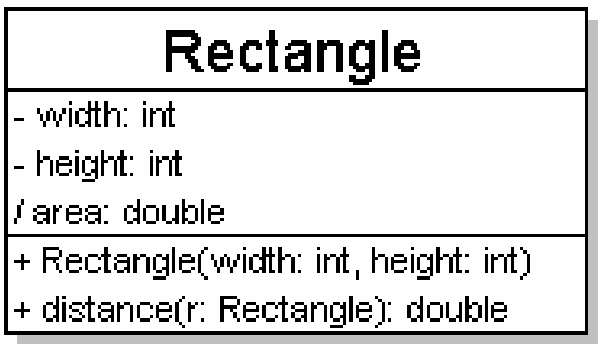
- **operations** / methods
  - may omit trivial (get/set) methods
    - but don't omit any methods from an interface!
  - should not include inherited methods

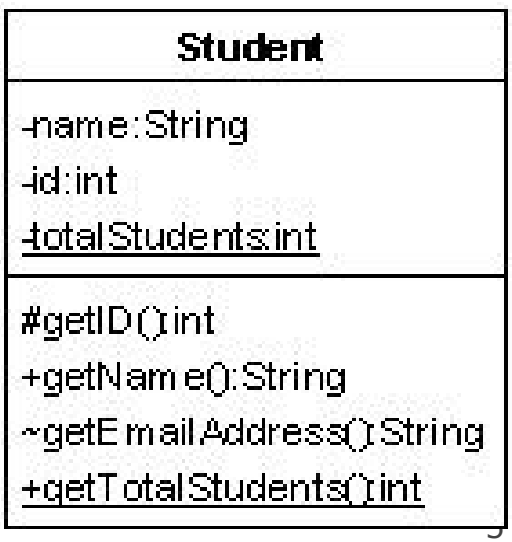




Class attributes

- attributes (fields, instance variables)
  - *visibility name* : *type* [*count*] = *defaultValue*
  - visibility: + public
    # protected
    – private
    ~ package (default)
    / derived
  - underline static attributes
  - **derived attribute**: not stored, but can be computed from other attribute values
  - attribute example:
    - balance : double = 0.00





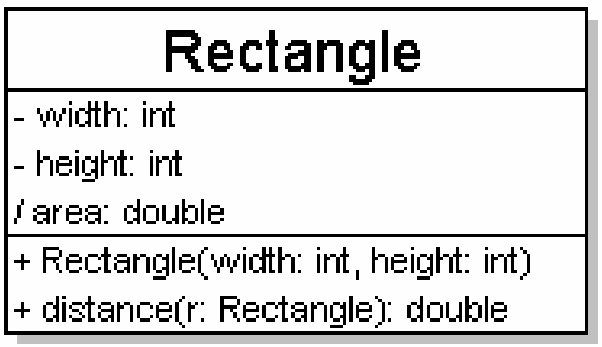
Class operations / methods

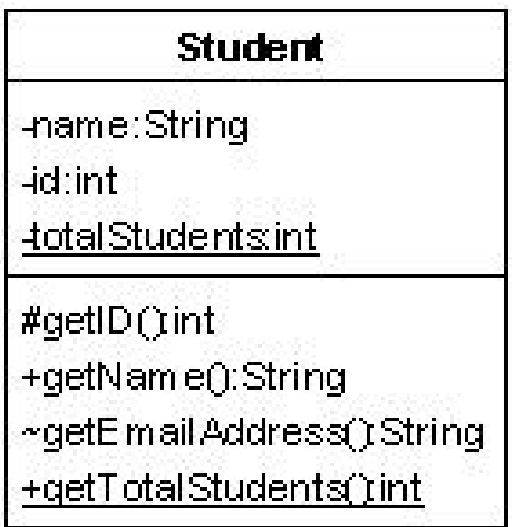
- operations / methods
  - *visibility name* (*parameters*) : *returnType*

  - underline static methods
  - parameter types listed as (name: type)
  - omit *returnType* on constructors and when return is `void`

  - method example:

    + distance(p1: Point, p2: Point): double



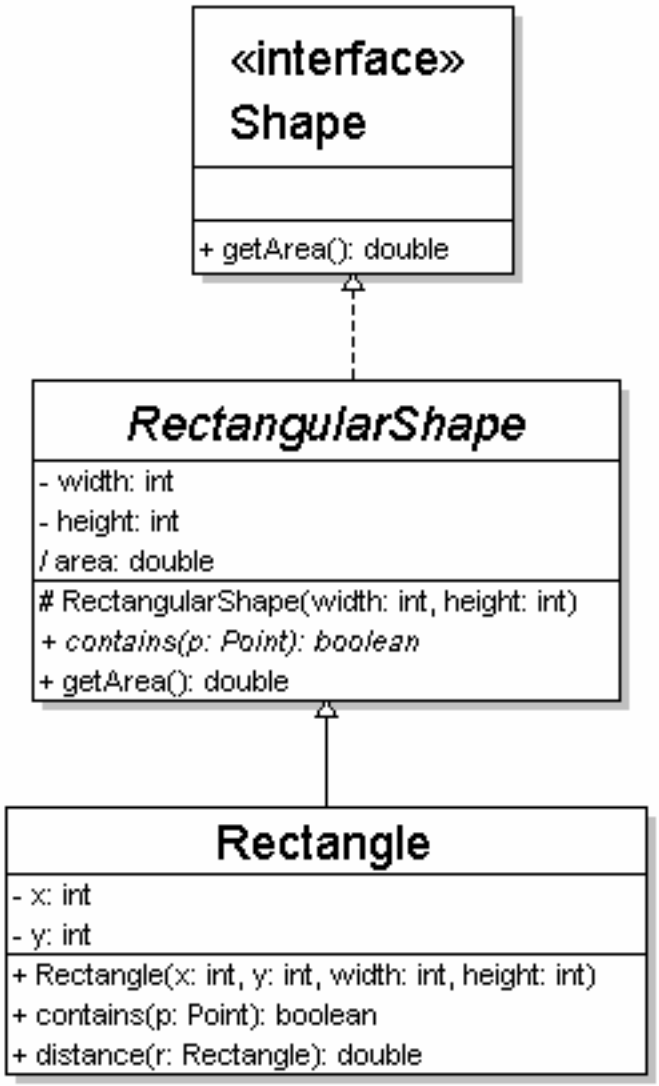


Inheritance relationships

- hierarchies drawn top-down with arrows pointing upward to parent

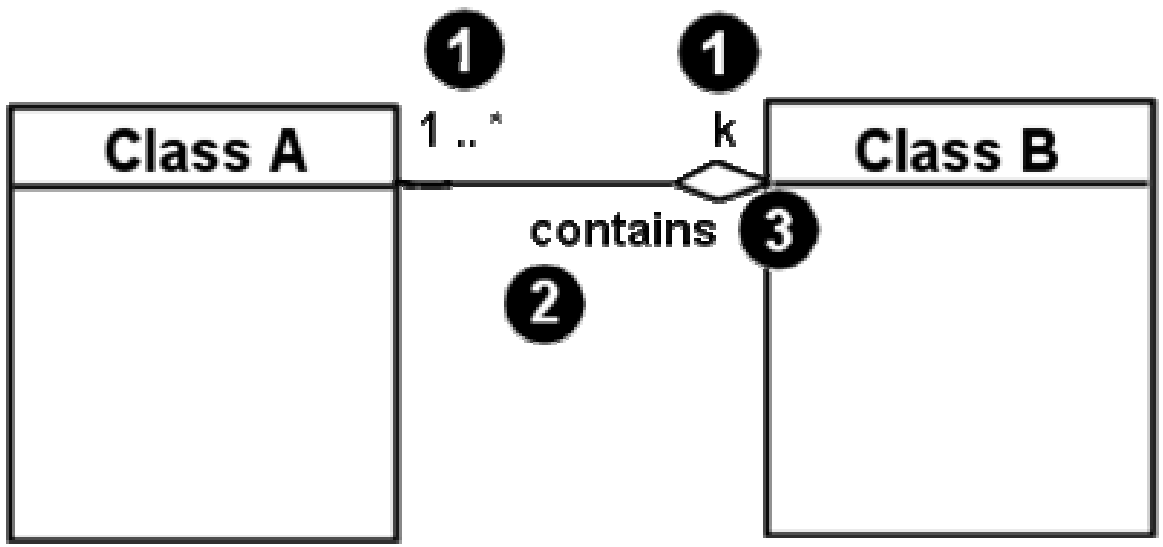
- line/arrow styles differ based on parent:
  - *class* : solid, black arrow
  - *abstract class* : solid, white arrow
  - *interface* : dashed, white arrow

- we often don't draw trivial / obvious relationships, such as drawing the class `Object` as a parent



Associational relationships

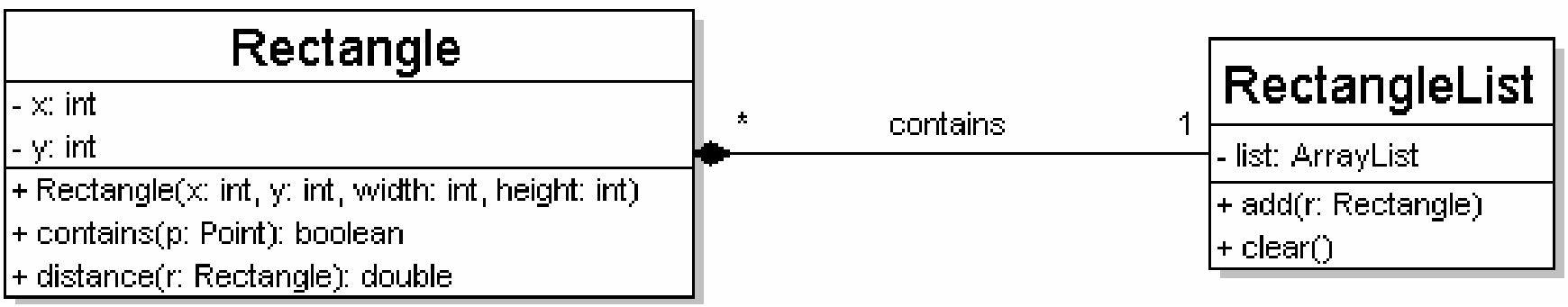
1. multiplicity (how many are used)
   - * ⇒ 0, 1, or more
   - 1 ⇒ 1 exactly
   - 2..4 ⇒ between 2 and 4, inclusive
   - 3..* ⇒ 3 or more
2. name (what relationship the objects have)
3. navigability (direction)



Multiplicity

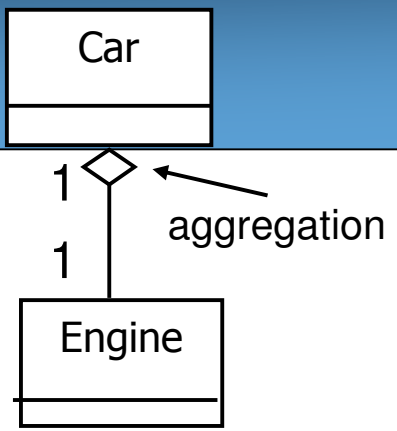
- one-to-one
  - each student must have exactly one ID card

- one-to-many
  - a RectangleList can contain 0, 1, 2, ... rectangles

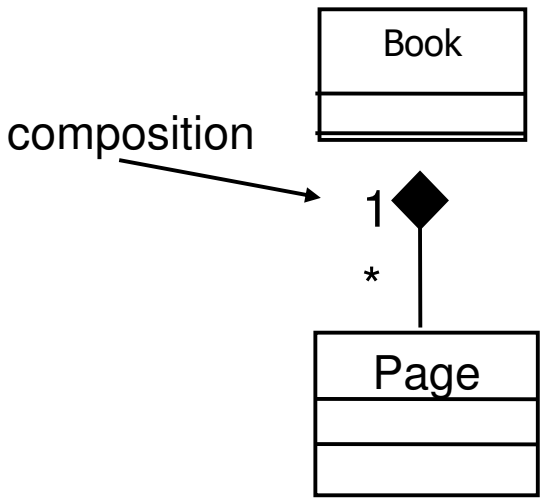


Association types

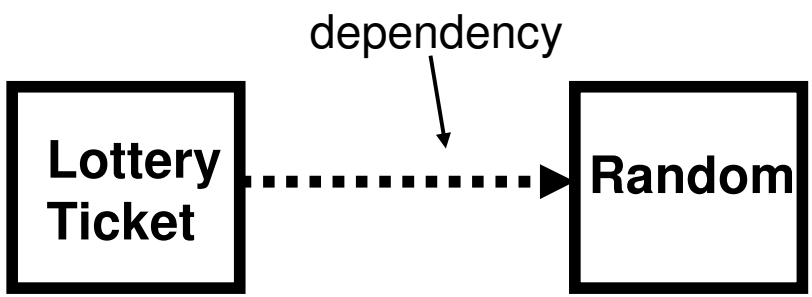
- **aggregation**: "is part of"
  - clear white diamond



- **composition**: "is entirely made of"
  - stronger version of aggregation
  - the parts live and die with the whole
  - black diamond



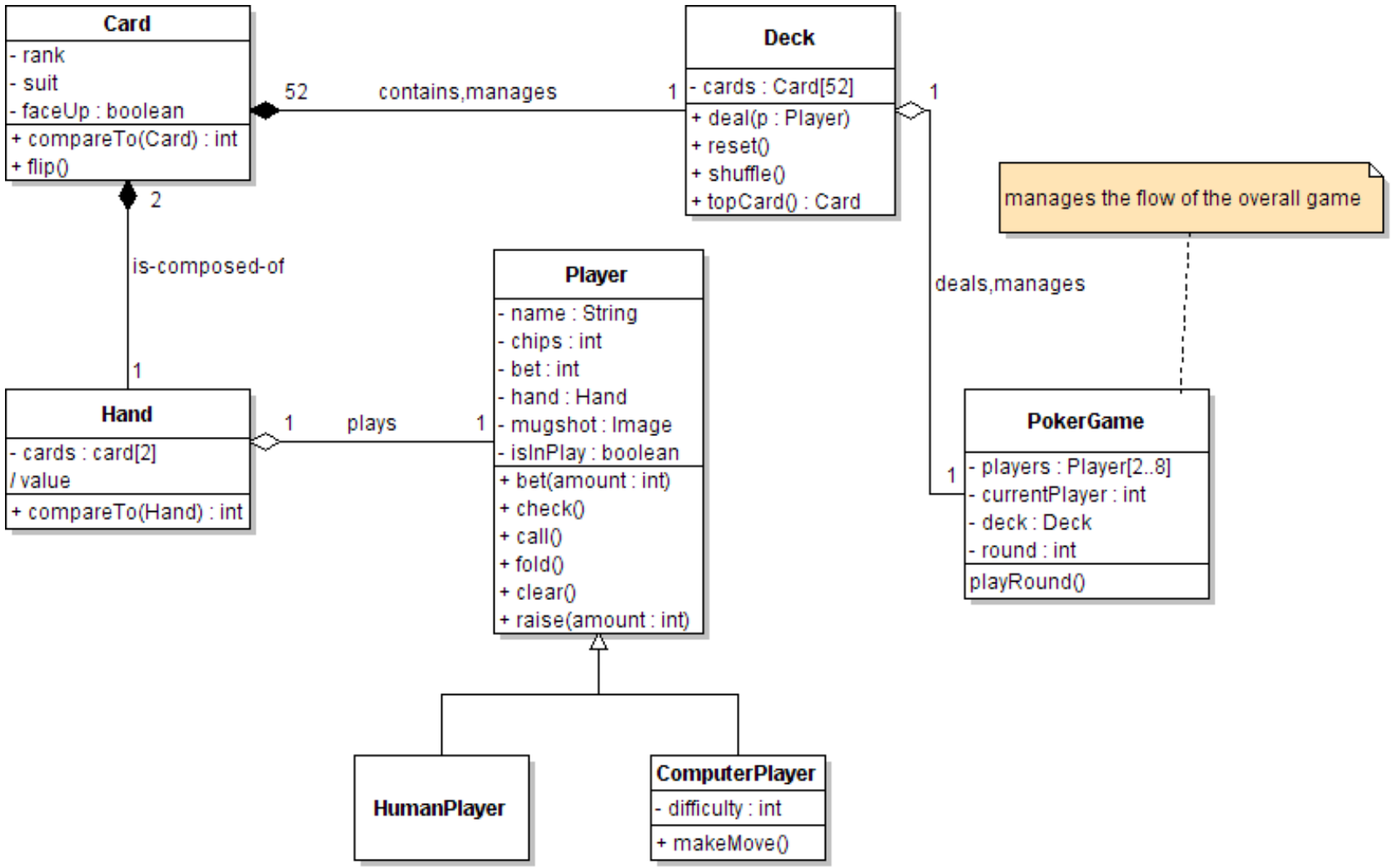
- **dependency**: "uses temporarily"
  - dotted line or arrow
  - often is an implementation detail, not an intrinsic part of that object's state



Class design exercise

- Consider this Texas Hold 'em poker game system:
  - 2 to 8 human or computer players
  - Each player has a name and stack of chips
  - Computer players have a difficulty setting: easy, medium, hard
  - Summary of each hand:
    - Dealer collects ante from appropriate players, shuffles the deck, and deals each player a hand of 2 cards from the deck.
    - A betting round occurs, followed by dealing 3 shared cards from the deck.
    - As shared cards are dealt, more betting rounds occur, where each player can fold, check, or raise.
    - At the end of a round, if more than one player is remaining, players' hands are compared, and the best hand wins the pot of all chips bet.
  - What classes are in this system? What are their responsibilities? Which classes collaborate?
  - Draw a class diagram for this system. Include relationships between classes (generalization and associational).

Poker class diagram



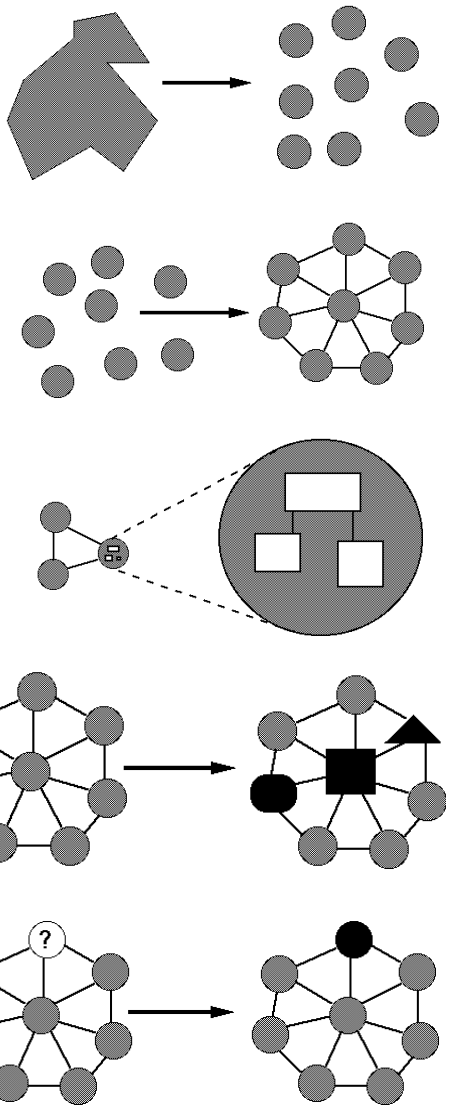
Class diag. pros/cons

- Class diagrams are great for:
  - discovering related data and attributes
  - getting a quick picture of the important entities in a system
  - seeing whether you have too few/many classes
  - seeing whether the relationships between objects are too complex, too many in number, simple enough, etc.
  - spotting dependencies between one class/object and another

- Not so great for:
  - discovering algorithmic (not data-driven) behavior
  - finding the flow of steps for objects to solve a given problem
  - understanding the app's overall control flow (event-driven? web-based? sequential? etc.)

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

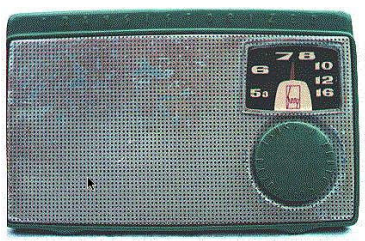
- **Heuristic 2.1**: All data should be hidden within its class.
- **Heuristic 2.2**: Users of a class must be dependent on its public interface, but a class should not be dependent on its users.
- **Heuristic 2.3**: Minimize the number of messages in the protocol of a class.
- **Heuristic 2.4**: Implement 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.

Interface/implementation

- **public interface**: visible data/behavior of an object
  - can be seen and executed externally

- **private implementation**: internal data/methods in an object
  - helps implement the public interface; cannot be directly accessed

- **client**: code that uses your class/subsystem



  - Example: *radio*
    - public interface is the speaker, volume buttons, station dial
    - private implementation is the guts of the radio; the transistors, capacitors, frequencies, etc. that user should not see

Poker design question 1

- 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()`, 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)

- 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

- Use a Java interface with only the needed methods, and then refer to your class by the interface type in client code.

Poker design question 2

- 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 the other `Player`s in the game, and a `Deck` reference to draw cards.
  - 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.

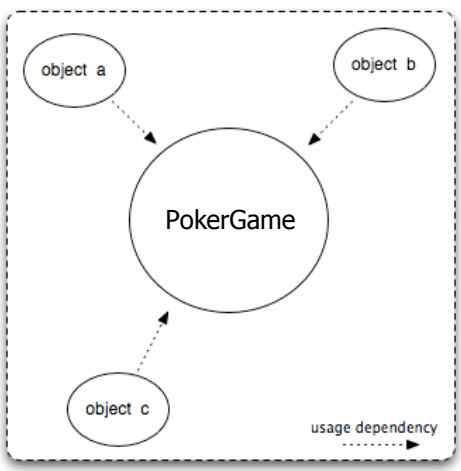
Heuristics 3 quick reference

- **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 3

- 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 at His computer

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

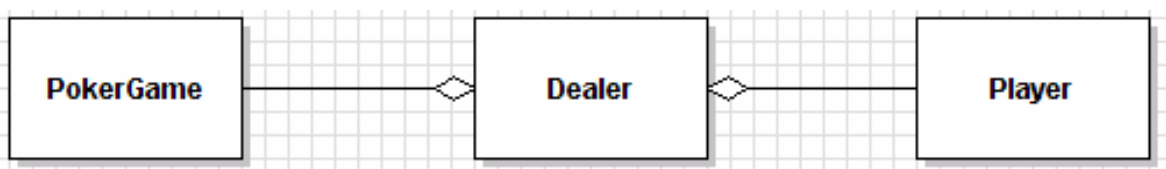
  - **Heuristic 2.8**: A class should capture one and only one key abstraction.

  - **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 3.4**: Beware of classes that have too much non-communicating behavior, that is, methods that operate on a proper subset of the data members of a class. God classes often exhibit much non-communicating behavior.

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?



Poker design question 5

- 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.

Poker design question 6

- 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?
  - Design 1: `Player` class remembers whether that player is in the game, what that player's 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.
  - 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.

Related data and behavior

- **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 and behavior are not being kept in one place.
  - "**policy**" behavior should be where that policy is enforced/enacted