Components

Hardware analogy: Consider a PC

- Consists of mechanical and electronic components (drives, boards, etc.) held together by standard interfaces (e.g. PCI bus).
- Boards consist of ICs conforming to standard interfaces (sockets).
- ICs often consist of numerous standard cells.
- Each standard cell consists of transistors.

Components are units of: Design, Sale, Reusability, Maintenance, Documentation, Responsibility, Change, Sharing.

[0]Thanks to Theodore S. Norvell for much of this lecture.

Interface

Set of assumptions that the "clients" of a component may make about how to "use" the component.
Consider a particular IC — a 7400 TTL in a DIP package.
Aspects of its interface:

- Physically it has 14 pins in precise physical positions.
- Pin 7 is ground. Pin 14 is 5VCC.
- Voltage levels are as defined in the TTL standard.
- Pins 3, 6, 8, 11 are outputs.
- All other pins are inputs.

In a sense the above are "syntactic" aspects of the interface.

Syntax and Semantic Interface

The semantics (behaviour) of the 7400:
The output on pin 3 is (after a given maximum delay) the NAND of the inputs on pins 1 and 2. And likewise for pins 6, 4, & 5, pins 8, 9, & 10, and pins 11, 12, & 13.
This distinguishes the 7400 from other ICs with similar "syntax" (e.g., 7408).
Syntax — form.
Semantics — meaning.

- The use of a part with the wrong syntactic interface can be detected with a minimum of intelligence.
- Detecting the use of a part with the wrong semantic interface requires an understanding of the system and some intelligence.

Implementation

The interface of the IC does not tell us

- Whether it was designed with VHDL, Verilog, or etc.
- Whether the IC contains Bipolar or field effect transistors (except indirectly as this affects voltage levels, current drain, and other observable quantities.)
- How many transistors it has.

These do not affect the suitability of the IC for its purpose and thus are not considered part of the interface.
They are considered matters of *implementation*.
Separating matters of interface from matters of implementation is a key concept in (software) engineering.
The interface abstracts what we need to know to use the device.

Components in Software

Various entities may serve as components.[1]

- Source files
- Variables
- Statements
- Subroutines (procedures, functions , methods, operations)
- Classes / Types / Interfaces
- Modules / Packages / Libraries
- Subsystems
- Programs
- Resources (e.g. icons and other data)

[1] Some writers use other terms for this (e.g., *module*) and some mean something particular by *component*.

Component Hierarchy

Many components contain other components. For example:

- Programs contain subsystesm or packages
- Subsystems contain packages
- Packages contain other packages, classes, types, variables, subroutines
- Classes contain variables (members), subroutines (methods), & other classes
- Subroutines contain variables and (depending on language) other subroutines

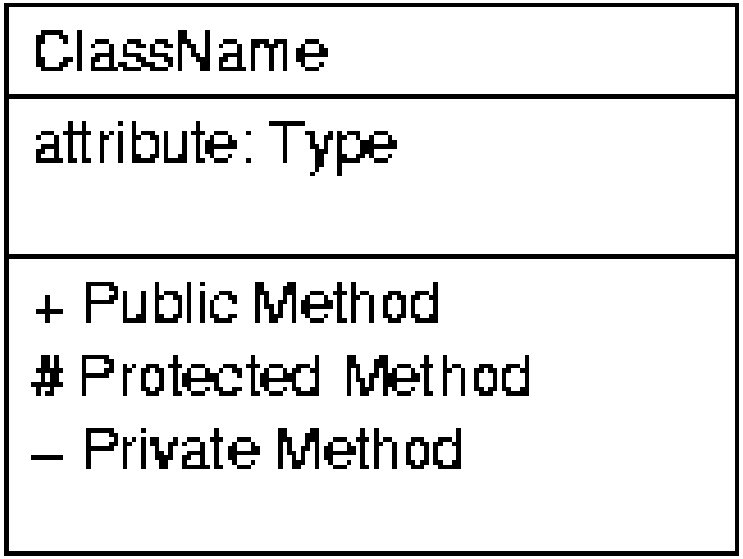
Is this hierarchy a tree?
Often the same component will be included in multiple programs — sharing.
But generally speaking the hierachy is a tree.

Class Design

- In Object-Oriented design a fundamental component is *objects*.
- Objects represent the key nouns (roles, physical objects, concepts) in the problem (or solution).
- A *class* is a specification for objects:
  - Name
  - A set of *attributes* (a.k.a. *fields*).
  - A set of *operations* (a.k.a. *methods*).
    - *Constructors*: initialize the object state
    - *Accessors*: report on the object state
    - *Mutators*: alter the object state
    - *Destructors*: clean up

Class Diagram

In UML a class is represented by a box with three sub-sections:



Note that UML can be used for many purposes:

software design — classes usually correspond to classes in the code.

domain (requirements) analysis — classes represent real objects in the problem domain.

Class in Java

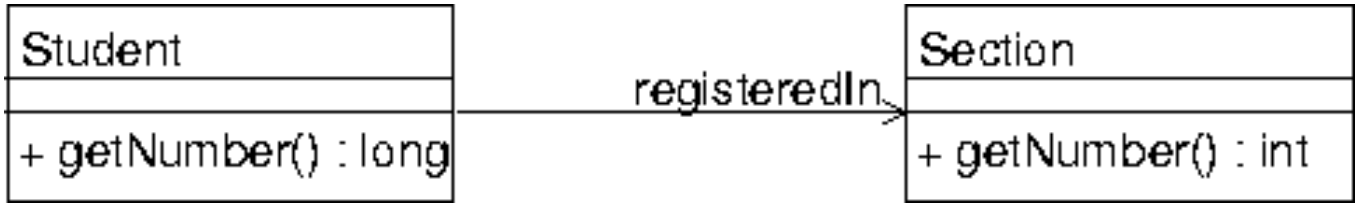
```
// File Student.java
class Student { // class name is Student
  // Attributes
  private long studNum;
  private String name;

  // Operations
  public Student(long sn, String nm) {
    studNum = sn; name = nm;
  }

  public String getName() { return name; }
  public long getNumber() { return studNum; }
}
```

Relationships between Classes

- Associations
- Aggregation
- Composition
- Dependence
- Generalization

Association

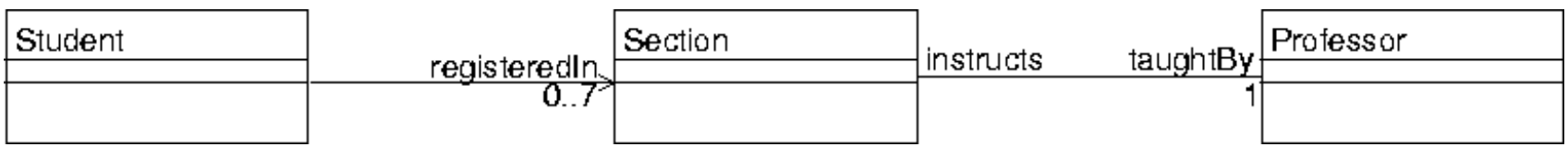
- If one or more instances of one class play a role w.r.t. instances of another class then the classes are associated.
- Can be helpful to name the role, but only do it when it adds clarity.
- Arrowheads indicate navagability (none indicates both directions).



Multiplicities

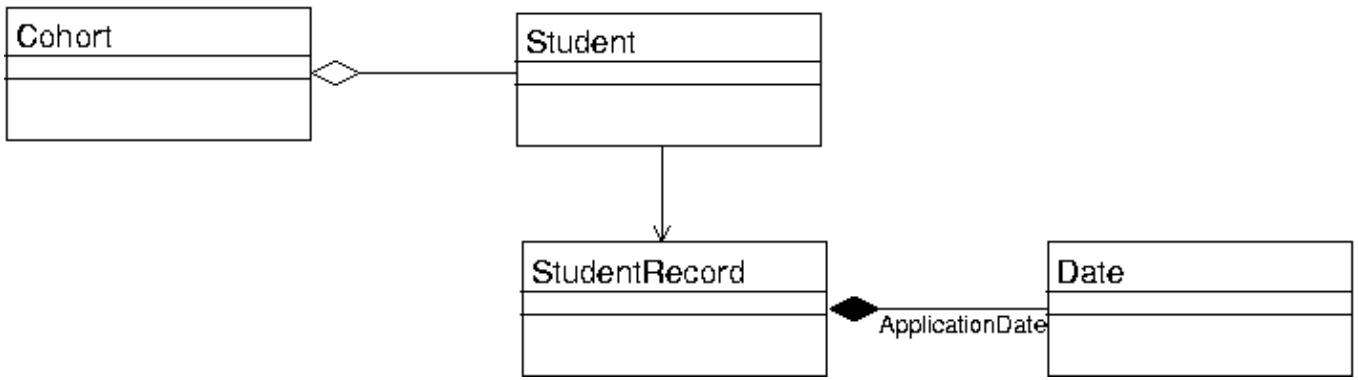
Indicate how many objects play a particular role:

Notation	Meaning
n	Exactly n
$*$	Any number
0..1	Optional (0 or 1)
$n..m$	Between n and m



Aggregation and Composition

- Both can be said "is part of."
- An object should only be part of at most one object.
- Composition indicates that one cannot exist without the other — the lifetime of the component and that of the whole are the same.
- (Rose doesn't support composition.)



Choosing Classes

- Candidates are nouns in the problem/solution description.
- Refine this by assigning *responsibilities* to each class:
  - Information hiding — a design decision is encapsulated in the implementation.
  - Services — classes provide services to each other (verbs in the methods).
- Identify *collaborations* — interactions with other classes to carry out its responsibility.

Class-Responsibility-Collaboration (CRC) Cards

- Use index cards or post-it notes etc. to illustrate each class.
- Lines (e.g., on a whiteboard) of position (on a table) to indicate collaborations
- Not part of UML, but a very useful technique.

Class Name	
responsibility	collaboration
responsibility	collaboration
...	...

Kinds of Classes

1. Information storage

   Service: An abstract interface to the information.
   Secret: The data structures used.
   Examples: Classes representing sequences, sets, finite functions, chess boards, maps of rooms.

2. Algorithmic

   Service: Performing some manipulation of data.
   Secret: The algorithm used.
   Examples: Sort a sequence, find shortest path in a graph, determine best move in a chess game, plot best course through a room.

Kinds of Classes (cont'd)

3. Device interface
   - Service: Provide access to an external device.
   - Secret: The nature of the device and the protocols for using it.
   - Examples: Sensing the input from a touch sensitive chess board, moving a robotic arm to a specific location, outputting to a console (or console window).
4. OS interface
   - Service: Services provided via the OS.
   - Secret: The OS being used and the means to communicate with it.
   - Examples: Interacting with the file system or window system or process system.

Kinds of Classes (cont'd)

5. Formatting and Parsing Classes

   Service: Input and output of data in specific formats.
   Secret: The format.
   Examples: Input and output filters in word-processors. Configuration file reading and writing.

6. Adaptation Classes

   Service: Representing other classes in a way conforming to a given interface.
   Secret: Which classes are being represented.
   Examples: Event listener classes (e.g. ActionListeners in Java). Iterator classes (e.g. Enumerations in Java).

Kinds of Classes (cont'd)

7. Structural classes
   - Service: Providing a single interface to a number of objects.
   - Secret: The details of those objects.
   - Examples: Façade classes. Container classes in Java's AWT.
8. Creators
   - Service: Creating objects needed by other classes.
   - Secret: The method of creation and the concrete class of the objects.
   - Examples: Factory classes (later).

Kinds of Classes (cont'd)

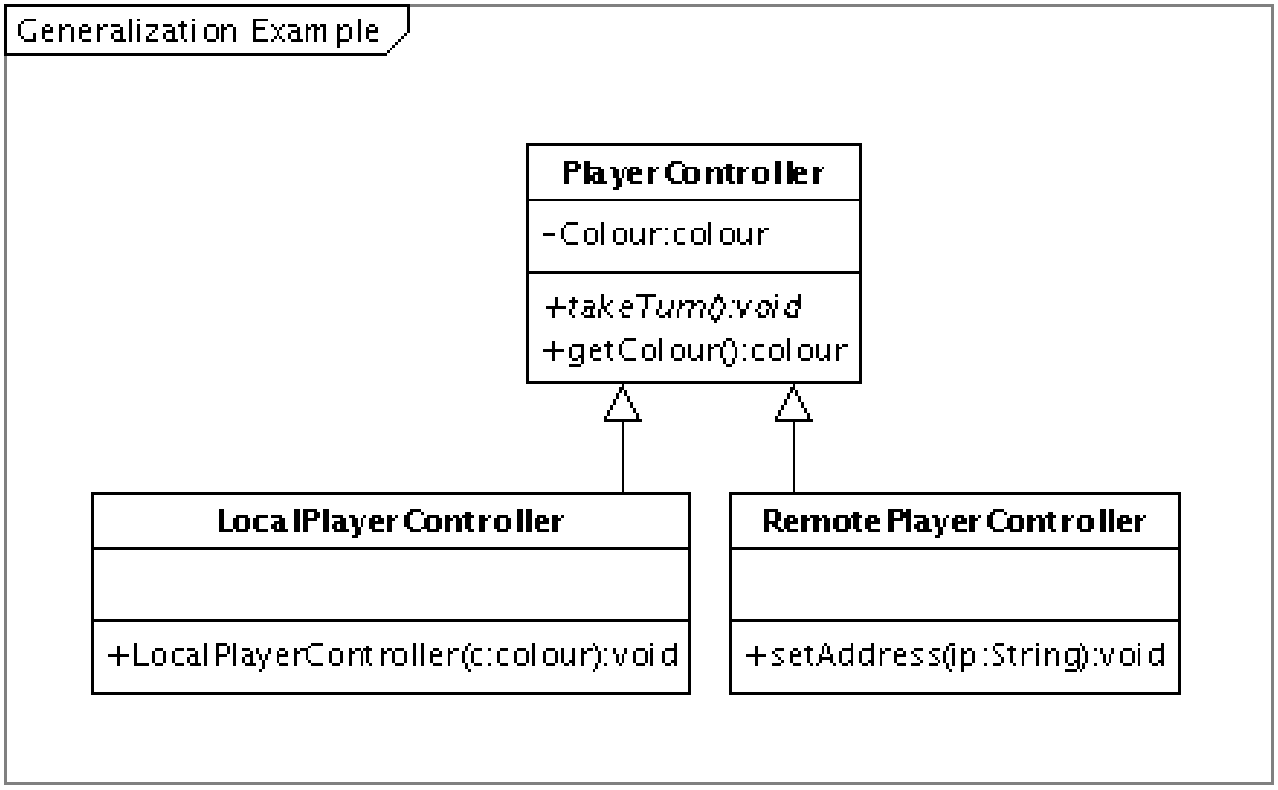
9. Arbitrary facts

   Examples: Numerical constants, rules for financial calculations, rules or games, rules for formatting dates and monetary amount, strings corresponding to messages, names of provinces and countries, rules for checking validity (e.g. of postal codes, credit card number), etc. These facts tend to change as software is used in other countries and languages (internationalization).

Generalization

- Sub/super-type relationship (i.e., "is a kind of").
- Attributes and operations of the super-class are *inherited* by the sub-class.
- Sub-class *extends* (adds to) the behaviour/interface of the super class.
- Perhaps the most talked about association in early OO design.
- Use with caution:
  - The "wrong" generalization can make system difficult to modify.
  - Be wary of too deep generalization structures.
  - Principle of substitutability
    - Any instance of the sub-class should be substitutable for an instance of the super-class.
    - Any statement that you'd make about the super-class should be true of all sub-classes.

Generalization Example



Generalization Example

```
public class RemotePlayerController
  extends controller.PlayerController {

    public void takeTurn() {
        // implementation of takeTurn
    }

    public void setAddress(String ip) {
        // implementation of setAddress
    }
}
```

Design By Contract

- Principle is that the interface to a component forms a contract between the client and supplier.
- The implementer of other components (client) is told what she must supply, the component implementer (supplier) says what will be the supplied.
- For classes, the public interface declaration is the syntactic interface.
- The semantic (behavioural) interface is given by pre- and post-conditions for each public method.
  - *Pre-condition* is required to be true before a method starts execution — it is the client's duty to ensure this.
  - *Post-condition* is assured to be true when a method finishes execution — it is the supplier's duty to ensure this.

Design by Contract (cont'd)

- (Class) *invariant* — a condition that is required to be true in all consistent states of the component (i.e., when no methods of the component are executing).
  - Can help implementer and future modifiers to understand component (and therefore get it right).
  - Conjoin (logical **and**) with pre-condition and post-condition.

Substitutability and DBC

Susbstitutability implies that:

- Assertions can only increase the responsibility of a sub-class.
- Sub-class can:
  - weaken pre-conditions.
  - strengthen post-conditions.
  - strengthen invariants.