Object-Oriented Design

Department of Computer Science
University of Maryland, College Park
Applying Object-Oriented Design

1. Look at objects participating in system
   - Find **nouns** in problem statement (requirements & specifications)
   - Noun may represent class needed in design
   - Relationships (e.g., “has” or “belongs to”) may represent fields

2. Look at interactions between objects
   - Find **verbs** in problem statement
   - Verb may represent message between objects

3. Design classes accordingly
   - Determine relationship between classes
   - Find state & methods needed for each class
Classes

- A class or interface defines and describes a set of objects.
- It describes a set of methods or messages that the object responds to:
  - Not only the name and signature of the method, but the contract the method respects.
- Classes also provide/describe fields and method implementations.
1) Finding Classes

Thermostat uses dial setting to control a heater to maintain constant temperature in room

Nouns

- Thermostat
- Dial setting
- Heater
- Temperature
- Room
Finding Classes

Analyze each noun

- Does noun represent class needed in design?
- Noun may be outside system
- Noun may describe state in class
Analyzing Nouns

- Thermostat
  - Central class in model
  - State in class (Thermostat)
- Dial setting
  - State in class (Thermostat)
- Heater
  - Class in model
- Room
  - Class in model
- Temperature
  - State in class (Room)
Finding Classes

- Decision not always clear
  - Possible to make everything its own class
    - Approach taken in Smalltalk
    - Overly complex
      - $2 + 3 = 5$ vs. NUM2.add(NUM3) = NUM5
  - Impact of design
    - More classes ⇒ more abstraction, flexibility
    - Fewer classes ⇒ less complexity, overhead
  - Choice (somewhat) depends on personal preference
Singleton classes

- A Singleton class is a class for which there will only ever be one instance
- Makes sense if the class is a subclass of another class
  - For example, you might have a class Person, and a singleton subclass Elvis
- Avoid making verbs/functions into classes
  - Examples – class ListSorter, NameFinder
  - Unless you might have multiple verb classes that all implement a common interface
    - The Strategy design pattern
Thermostat uses dial setting to control a heater to maintain constant temperature in room

Verbs
- Uses
- Control
- Maintain
Finding Messages

- Analyze each verb
  - Does verb represent interaction between objects?
- For each interaction
  - Assign methods to classes to perform interaction
Analyzing Verbs

- **Uses**
  - “Thermostat *uses* dial setting…”
  - ⇒ Thermostat.setDesiredTemp(int degrees)

- **Control**
  - “To *control* a heater…”
  - ⇒ Heater.turnOn()
  - ⇒ Heater.turnOff()

- **Maintain**
  - “To *maintain* constant temperature in room”
  - ⇒ Room.getTemperature()
Example Messages

Thermostat

- getTemperature()
- setDesiredTemp()
- turnOn()
- turnOff()

Room

Heater
Resulting Classes

- **Thermostat**
  - State – dialSetting
  - Methods – setDesiredTemp()

- **Heater**
  - State – heaterOn
  - Methods – turnOn(), turnOff()

- **Room**
  - State – temp
  - Methods – getTemperature()
If a class Y extends class X and implements interface A
then Y is a subtype of both X and A

If Q is a subtype of P, then Q satisfies P’s contract
Anyone who expects a P can be given a Q

This is known as the Liskov Substitution Principle (named for Prof. Barbara Liskov)

Not always strictly followed, but an ideal to approach
For example, some iterators don’t support remove
Which could be a subtype?

**Class B**

```java
/** Search for x in a,
 * return location of first occurrence,
 * -1 if not found */
int search(int x, int a[]) { ... }
```

**Class C**

```java
/** .. same as above...
 * uses binary search for speed;
 * the array a must be sorted */
int search(int x, int a[]) { ... }
```
Which could be a subtype?

Class B {
/** Search for x in a, *
 * return location of first occurrence,
 * -1 if not found */
 int search(int x, int a[]) { … } }

Class C {
/** * Search for x in a, *
 * return location of any occurrence,
 * -1 if not found */
 int search(int x, int a[]) { … } }
is-a vs. has-a

Say we have two classes, Engine and Car

Two possible designs

- A Car object has a reference to an Engine object
  - has-a

- The Car class is a subtype of Engine
  - is-a
Prefer Composition to Subtyping

- Generally, prefer composition/delegation (has-a) to subtyping (is-a)
  - Subtyping is very powerful, but easy to overuse and can create confusion and lead to mistakes
- Using is-a restricts you from having a car with more than one engine, or with no engine
- Tempting to use subclassing in places where it doesn’t really make conceptual sense to avoid having to delegate methods
  - Don’t
Forms of Inheritance

- **Extension**
  - Adds new functionality to subclass
    - In Java $\rightarrow$ new method

- **Limitation**
  - Restricts behavior of subclass
    - In Java $\rightarrow$ override method, throw exception

- **Combination**
  - Inherits features from multiple superclasses
  - Also called *multiple inheritance*
  - Not possible in Java
    - In Java $\rightarrow$ implement interface instead
Multiple Inheritance Example

- **Combination**
  - AlarmClockRadio has two parent classes
  - State & behavior from both Radio & AlarmClock

![Diagram of Multiple Inheritance Example](diagram.png)