

CMSC 132: Object-Oriented Programming II



Object-Oriented Design II

Department of Computer Science
University of Maryland, College Park

Overview

- Object-oriented design
 - Objects, methods ⇒ Last lecture
 - Classes, inheritance ⇒ This lecture
- Applying object-oriented design

Elements of Object-Oriented Design

- Objects
 - Entities in program
- Methods
 - Functions associated with objects
- Classes
 - Groups of objects with similar properties
- Inheritance
 - Relationship between classes

Classes

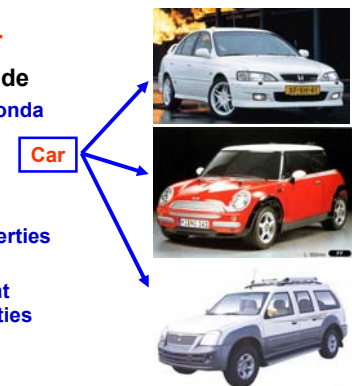
- Definition
 - Group of objects with same state & behavior
 - Abstract description of a group of objects
- Similar to data types
 - Type is a set of data values & their operations
 - Example ⇒ integer, real, boolean, string
 - Can view classes as types for objects

Classes

- Properties
 - Classes provides classification for objects
 - Every object belongs to some class
 - Objects ⇒ instances (instantiations) of a class

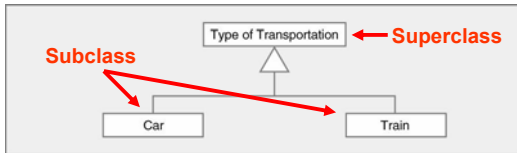
Example Class

- Given a class **Car**
- Objects can include
 - MyHonda, YourHonda
 - HerMiniCooper
 - HisSUV
- All **Car** objects
 - Share same properties & behavior
 - May have different values for properties



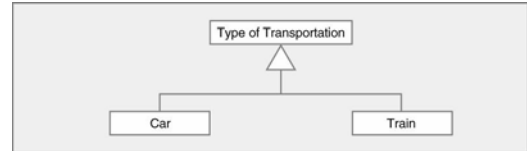
Inheritance

- **Definition**
 - Relationship between classes when state and behavior of one class is a subset of another class
- **Terminology**
 - Superclass / parent ⇒ More general class
 - Subclass ⇒ More specialized class



Inheritance

- **Properties**
 - Subclass **inherits** state & behavior of superclass
 - “**Is-a**” relationship exists between inherited classes
 - Example – train is a type of transportation

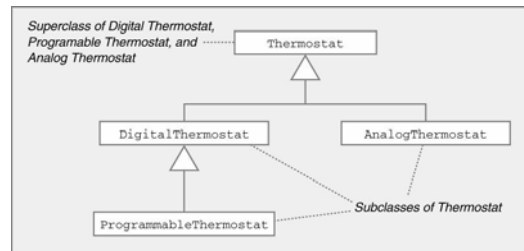


Inheritance

- **Inheritance forms a hierarchy**
 - Helps organize classes
- **Inheritance is transitive**
 - Class inherits state & behavior from all ancestors
- **Inheritance promotes code reuse**
 - Reuse state & behavior for class

Inheritance Hierarchy Example

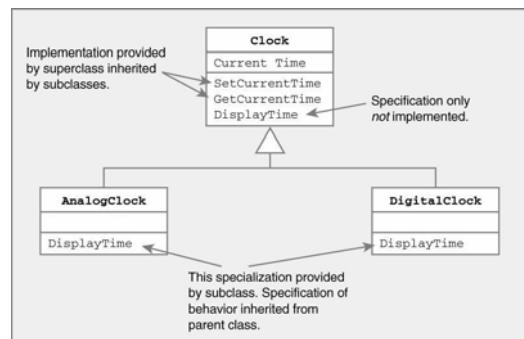
- **Classes**
 - Thermostat
 - Digital thermostat
 - Analog thermostat
 - Programmable thermostat



Forms of Inheritance

- **Specification**
 - Defines behavior implemented only in subclass
 - Guarantees subclasses implement same behavior
 - In Java → abstract method in superclass
- **Specialization**
 - Subclass is customized
 - Still satisfies all requirements for parent class
 - In Java → override method

Specialization Example

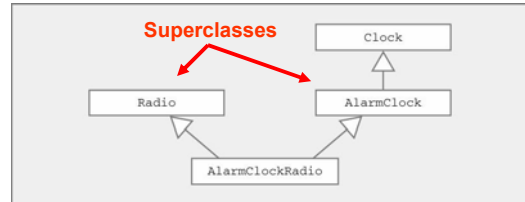


Forms of Inheritance

- **Extension**
 - Adds new functionality to subclass
 - In Java → new method
- **Limitation**
 - Restricts behavior of subclass
 - In Java → override method, throw exception
- **Combination**
 - Inherits features from multiple superclasses
 - Also called **multiple inheritance**
 - Not possible in Java
 - In Java → implement interface instead

Multiple Inheritance Example

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



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
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

1) Finding Classes

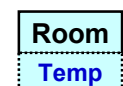
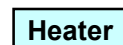
- **Thermostat** uses **dial setting** to control a **heater** to maintain constant **temperature** in **room**
- **Nouns**
 - Thermostat
 - Dial setting
 - Heater
 - Temperature
 - Room

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
- **Dial setting**
 - State in class (Thermostat)
- **Heater**
 - Class in model
- **Room**
 - Class in model
- **Temperature**
 - State in class (Room)

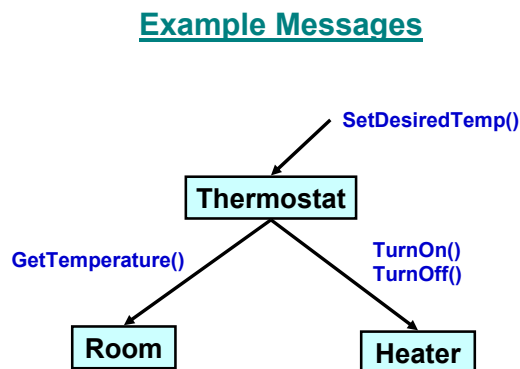


Finding Classes

- 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 \Rightarrow more abstraction, flexibility
 - Fewer classes \Rightarrow less complexity, overhead
 - Choice (somewhat) depends on personal preference
- Avoid making functions into classes
 - Examples – class `ListSorter`, `NameFinder`

Finding Messages

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



2) Finding Messages

- Thermostat **uses** dial setting to **control** a heater to **maintain** constant temperature in room
- Verbs
 - Uses
 - Control
 - Maintain

Analyzing Verbs

- Uses
 - “Thermostat **uses** dial setting...”
 - \Rightarrow `Thermostat.SetDesiredTemp()`
- Control
 - “to **control** a heater...”
 - \Rightarrow `Heater.TurnOn()`
 - \Rightarrow `Heater.TurnOff()`
- Maintain
 - “to **maintain** constant temperature in room”
 - \Rightarrow `Room.GetTemperature()`

Resulting Classes

- Thermostat
 - State – `DialSetting`
 - Methods – `SetDesiredTemp()`
- Heater
 - State – `HeaterOn`
 - Methods – `TurnOn()`, `TurnOff()`
- Room
 - State – `Temp`
 - Methods – `GetTemperature()`