CMSC 132: OBJECT-ORIENTED PROGRAMMING II

Design

Department of Computer Science
University of Maryland, College Park

Applying Object-Oriented Design

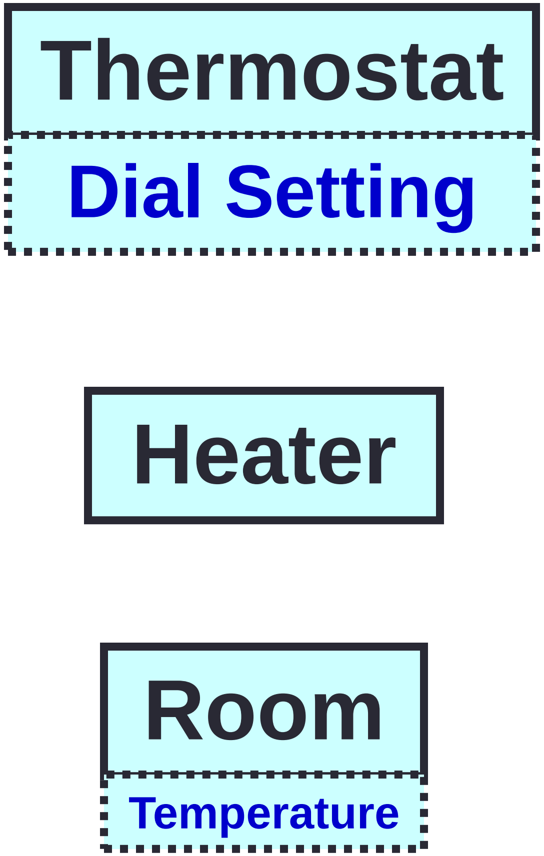
- We can use the term “message” to describe the interaction between objects. **Let’s see an example**
- When designing a system based on a problem statement:
  - Look at objects participating in system
    - Find nouns in the problem statement (requirements & specifications)
    - Noun may represent class/variable(s) needed in the design
    - Relationships (e.g., “has” or “belongs to”) may represent instance variables
  - Look at interactions between objects
    - Find verbs in problem statement
    - Verb may represent message between objects
  - Design classes accordingly
    - Determine relationship between classes
    - Find state & methods needed for each class

Step #1: Finding Classes

- **Problem Statement**
  - Thermostat uses dial setting to control a heater to maintain constant temperature in room
- **Nouns**
  - Thermostat
  - Dial setting
  - Heater
  - Temperature
  - Room
- **Analyze Each Noun**
  - Does noun represent a class needed in the 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)



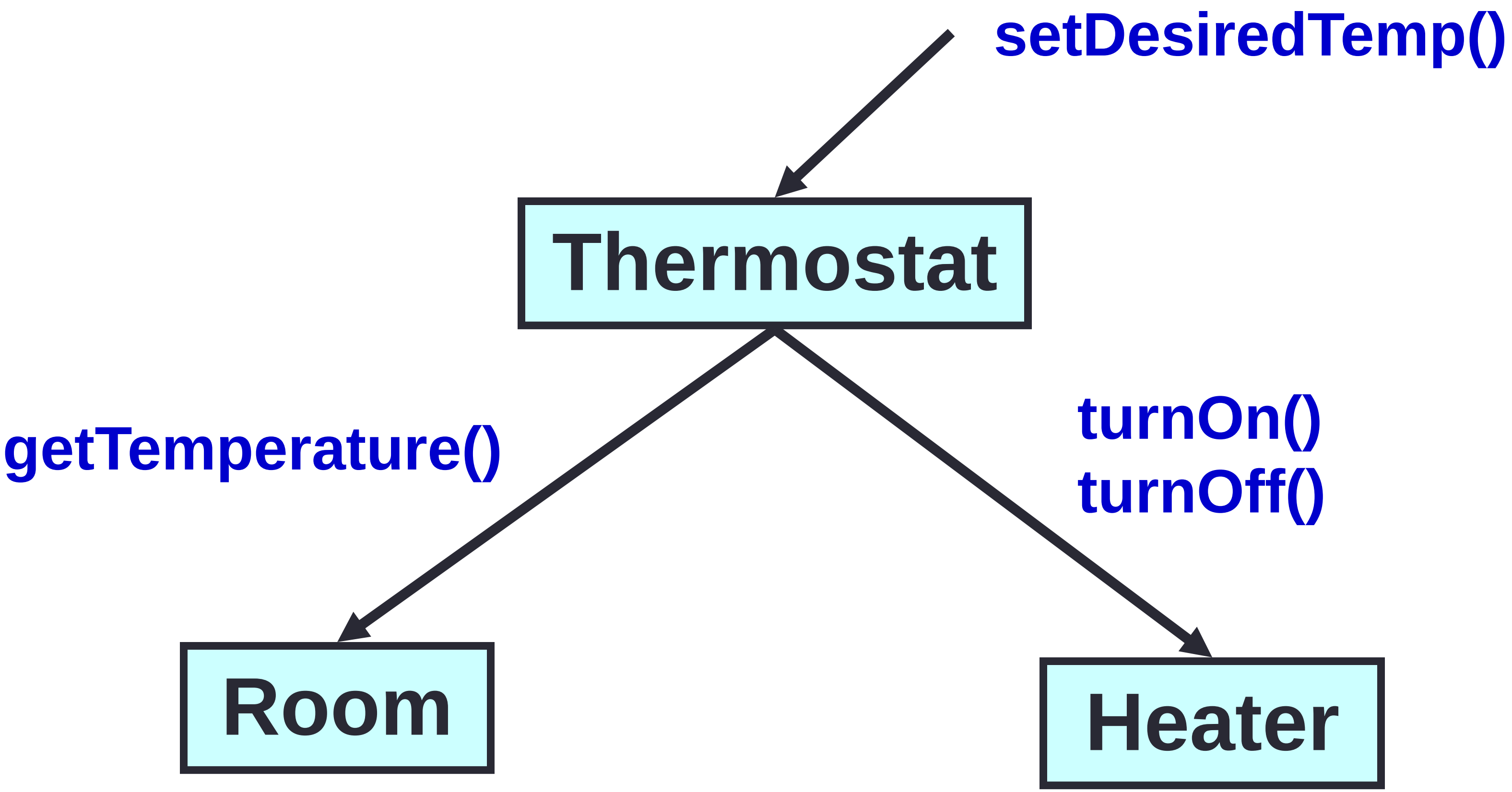
Step #2: Finding Messages

- Thermostat uses dial setting to control a heater to maintain constant temperature in room
- **Verbs**
  - Uses
  - Control
  - Maintain
- Analyze each verb
  - Does the verb represent interaction between objects?
- For each interaction
  - Assign methods to classes to perform interaction

Analyzing Verbs

- Uses
  - "Thermostat uses dial setting…"
  - $\Rightarrow$ Thermostat.setDesiredTemp(int degrees)
- Control
  - "To control a heater…"
  - $\Rightarrow$ Heater.turnOn()
  - $\Rightarrow$ Heater.turnOff()
- Maintain
  - "To maintain constant temperature in room"
  - $\Rightarrow$ Room.getTemperature()

Example Messages



Resulting Classes

- **Thermostat**
  - State - dialSetting
  - Methods - setDesiredTemp()
- **Heater**
  - State - heaterOn
  - Methods - turnOn(), turnOff()
- **Room**
  - State - temp
  - Methods - getTemperature()
- The above design could have been described using UML Class Diagrams

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 over Inheritance

- 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
- Tempting to use subtyping in places where it doesn’t really make conceptual sense to avoid having to delegate methods
  - Don’t
- Let’s see an example where we have an Employee class and we need to kinds of employees: salaried and hourly
  - Should we use composition or inheritance?

Immutable

- Define a class as immutable if possible
  - Do not add set methods by default
- You have already seen how sharing of immutable objects simplifies object duplication. Later on we will see additional advantages of immutable classes when threads are interacting with objects

Pseudocode

- How about pseudocode?

UML Class Diagrams

- Allow us to represent classes in our design
- There are Eclipse plugins for the generation of UML Diagrams