CMSC 132: OBJECT-ORIENTED PROGRAMMING II

Design

Department of Computer Science
University of Maryland, College Park

Few Things About Projects

- Remember that we take academic integrity very seriously. We have software tools that allow us to:
  - Compare all students projects (even across sections)
  - Changing variable names, and spacing is something our tools recognize
- You should try to submit your project often
  - Even though through CVS you can get previous project versions, using the submit server is easier

About JUnit Tests

- Remember: you need to bring StudentTest to office hours
- Study public tests so you understand what they are testing
- Expected results are in the actual tests or in text files that are part of your project
- You can add output statements so you can see the your program results

```
public void testSumBasic() {
    /* test code goes here */
    output += result[result.length-1];

    /* We don't need to print the result */
        /* Just to show we can see results from our code */
    System.out.println(output);

    assertEquals("1,3,6,10,15,21", output);
}
```

- Be careful and don’t modify public tests (copy tests to StudentTest file)
- You can step through tests using the debugger
- **Note:** We cannot disclose information about release tests or secret tests. After a project has been graded you can see a TA in order to see why you failed any release or secret tests

Applying Object-Oriented Design

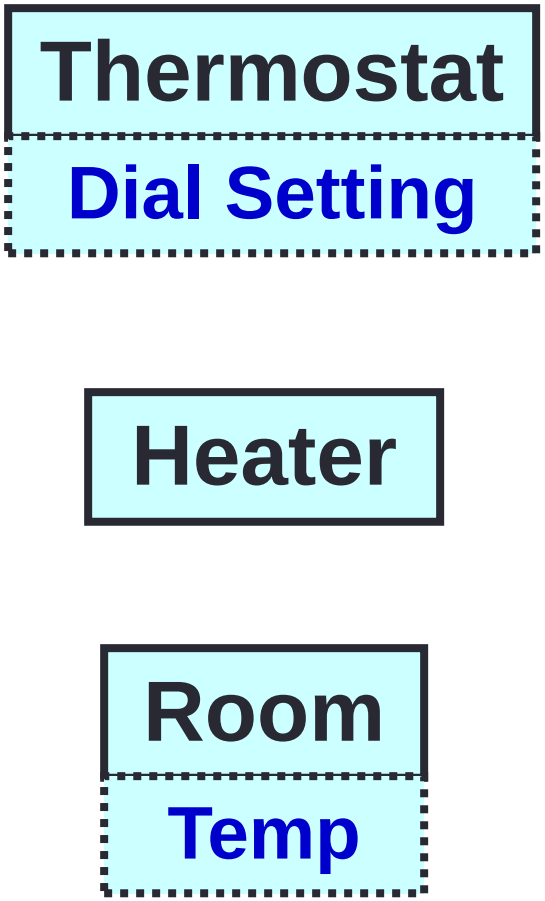
- We can use the term “message” to describe the interaction between objects. Let’s see an example
- When designing a system based on a problem statement:
  - Look at objects participating in system
    - Find nouns in the problem statement (requirements & specifications)
    - Noun may represent class/variables needed in the design
    - Relationships (e.g., “has” or “belongs to”) may represent fields
  - 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

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 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)



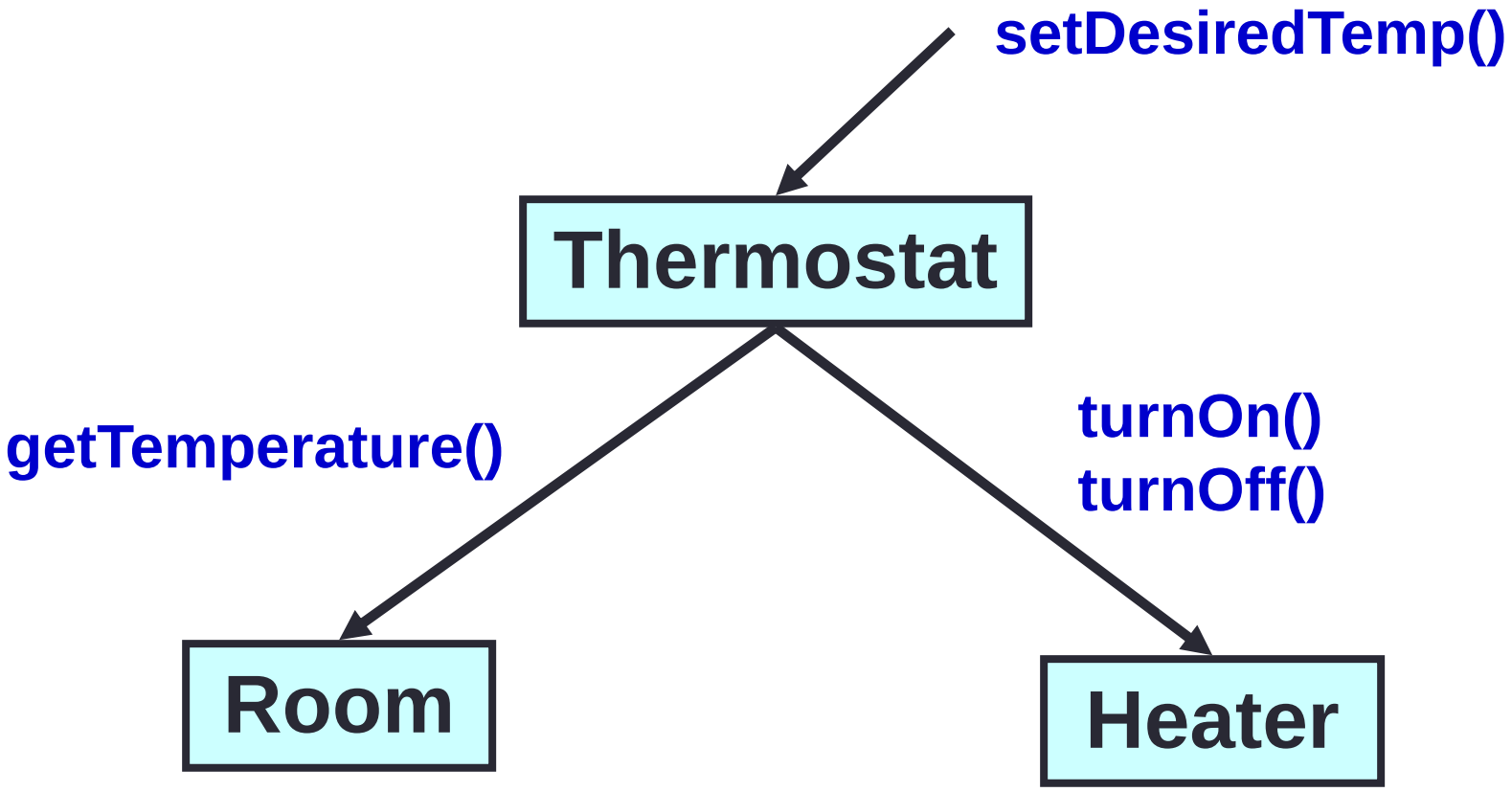
2) Finding Messages

- Thermostat uses dial setting to control a heater to maintain constant temperature in room
- Verbs
  - Uses
  - Control
  - Maintain
- Analyze each verb
  - Does 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



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

Immutable

- Define a class as immutable if possible
- Later on we will see the advantages of immutable classes

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
- Using is-a restricts you from having a car with more than one engine, or with no engine
- 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 of where we have an Employee class and we need to kinds of Employee: salaried and hourly

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