Life is great
VIA University College

Software Development with UML and Java 2

Strategy design pattern

What’s it about?

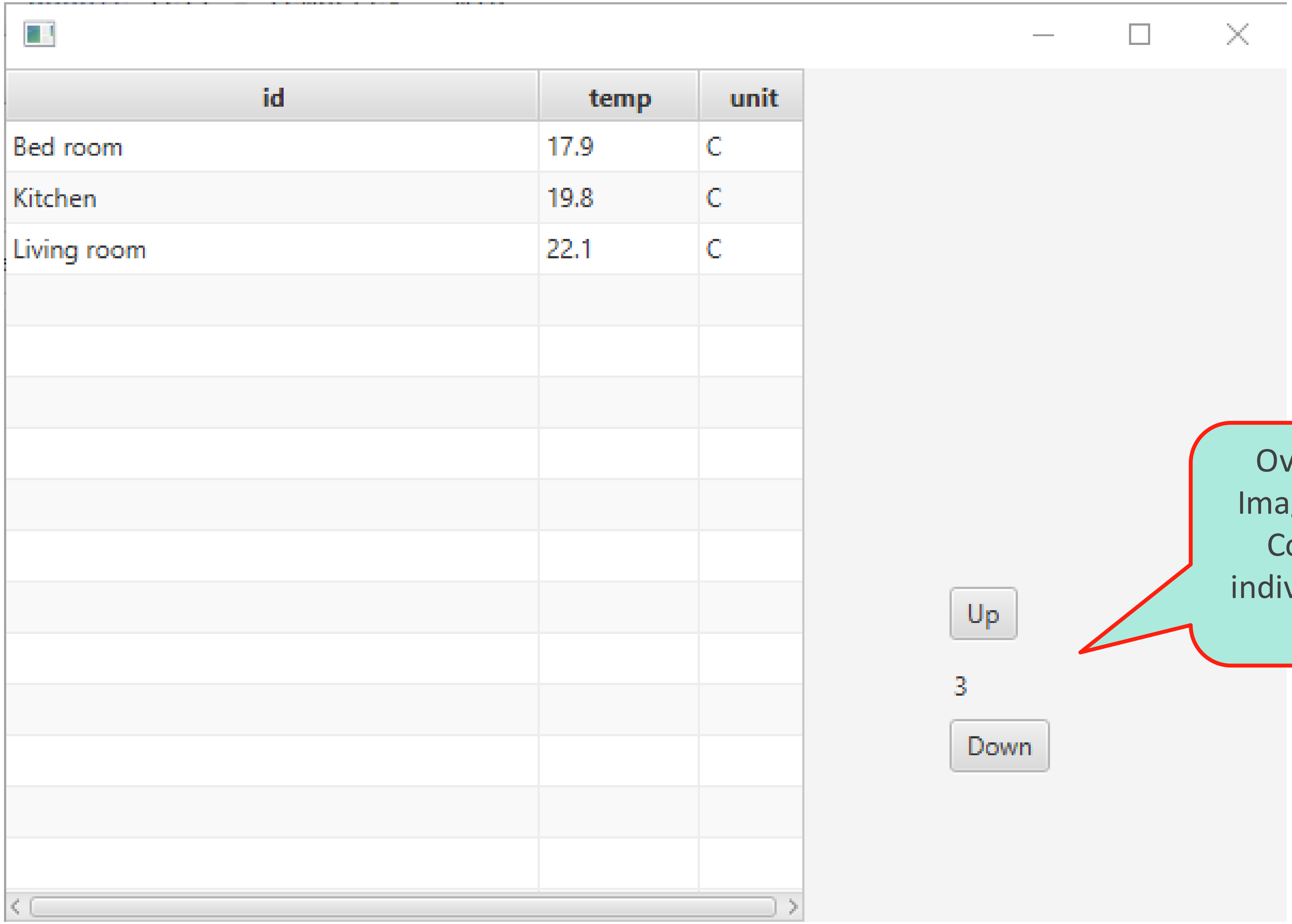
- It looks similar to State pattern, but different intent and behavior

- Example: Some class needs to calculate something, but based on conditions the result is calculated using different methods.

Agenda

- Example introduction, the thermostat
- New requirement, solving it with
  - if-statements
  - inheritance
  - composition
- Composition vs inheritance
- The Strategy pattern
  - General UML
  - Pros/cons
- State vs Strategy
- Assignment 2 at 11:20

Thermostat example (thermometer from previous session)

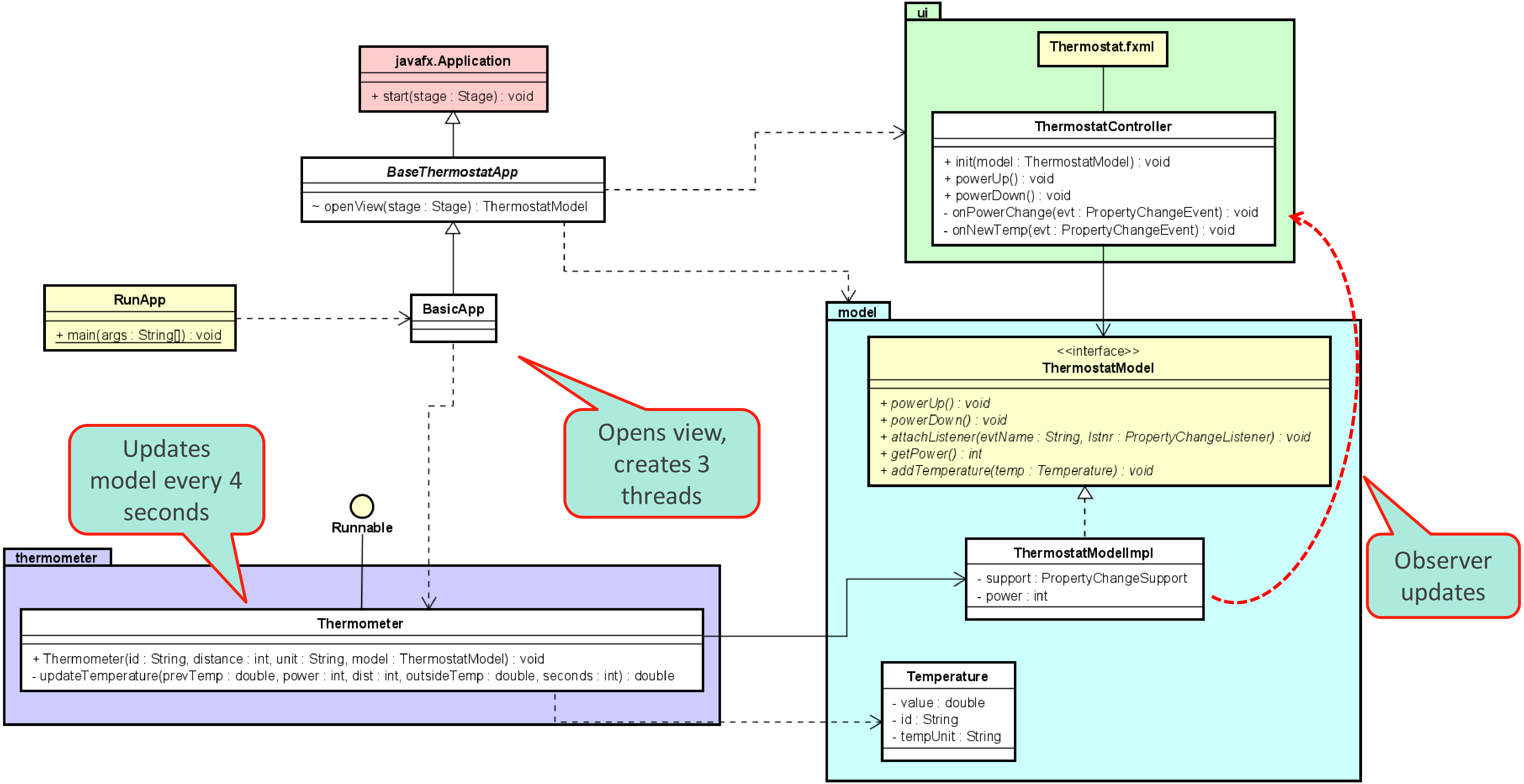


id	temp	unit
Bed room	17.9	C
Kitchen	19.8	C
Living room	22.1	C

Different thermometers shown here

Overall heat power. Imagine floor heating. Could have added individual radiators per room.

Thermostat example (thermometer from previous session)



javafx.Application
+ start(stage : Stage) : void

BaseThermostatApp
~ openView(stage : Stage) : ThermostatModel

RunApp
+ main(args : String[]) : void

BasicApp

Runnable

thermometer

Thermometer
+ Thermometer(id : String, distance : int, unit : String, model : ThermostatModel) : void
- updateTemperature(prevTemp : double, power : int, dist : int, outsideTemp : double, seconds : int) : double

ui

Thermostat.fxml

ThermostatController
+ init(model : ThermostatModel) : void
+ powerUp() : void
+ powerDown() : void
- onPowerChange(evt : PropertyChangeEvent) : void
- onNewTemp(evt : PropertyChangeEvent) : void

model

<<interface>>
ThermostatModel
+ *powerUp() : void*
+ *powerDown() : void*
+ *attachListener(evtName : String, lstnr : PropertyChangeListener) : void*
+ *getPower() : int*
+ *addTemperature(temp : Temperature) : void*

ThermostatModelImpl
- support : PropertyChangeSupport
- power : int

Temperature
- value : double
- id : String
- tempUnit : String

Updates model every 4 seconds

Opens view, creates 3 threads

Observer updates

Thermometer code

Boring constructor, initializing fields

```
public class ThermometerBasic implements Runnable {

    private double temp;
    private final String id;
    private final int distance;
    private final String unit;
    private final ThermostatModel model;

    public ThermometerBasic(String id, int distance, String unit, ThermostatModel model) {
        this.id = id;
        this.distance = distance;
        this.unit = unit;
        this.model = model;
    }
```

Thermometer code

```
public class ThermometerBasic implements Runnable {

    private double temp;
    private final String id;
    private final int distance;
    private final String unit;
    private final ThermostatModel model;

    public ThermometerBasic(String id, int distance, String unit, ThermostatModel model) {
        this.id = id;
        this.distance = distance;
        this.unit = unit;
        this.model = model;
    }

    @Override
    public void run() {
        while(true) {

            temp = updateTemperature(temp, model.getPower(), distance, 0, 4);
            model.addTemperature(new Temperature(temp, id,unit ));
            try {
                Thread.sleep(4000);
            } catch (InterruptedException ignored) {
            }
        }
    }
```

Core behavior of the thermometer

Thermometer code

Calculation used by core behaviour

```
            } catch (InterruptedException ignored) {
            }
        }
    }

    private double updateTemperature(double prevTemp, int power, int dist, double outsideTemp, int seconds) {
        double tMax = Math.min(11 * power + 10, 11 * power + 10 + outsideTemp);
        tMax = Math.max(Math.max(prevTemp, tMax), outsideTemp);
        double heaterTerm = 0;
        if (power > 0) {
            double den = Math.max((tMax * (20 - 5 * power) * (dist + 5)), 0.1);
            heaterTerm = 30 * seconds * Math.abs(tMax - prevTemp) / den;
        }
        double outdoorTerm = (prevTemp - outsideTemp) * seconds / 250.0;
        prevTemp = Math.min(Math.max(prevTemp - outdoorTerm + heaterTerm, outsideTemp), tMax);
        return prevTemp;
    }
}
```

Agenda

- Example introduction, the thermostat
- New requirement, solving it with
  - if-statements
  - inheritance
  - composition
- Composition vs inheritance
- The Strategy pattern
  - General UML
  - Pros/cons
- State vs Strategy

The problem

- New requirement:
  - I want to add support for simulating a **static** outside temperature, e.g. just hardcoded to 0.

- Solution:
  - Add a condition in the run() method, to check if the thermometer should simulate indoor or outside temperature.

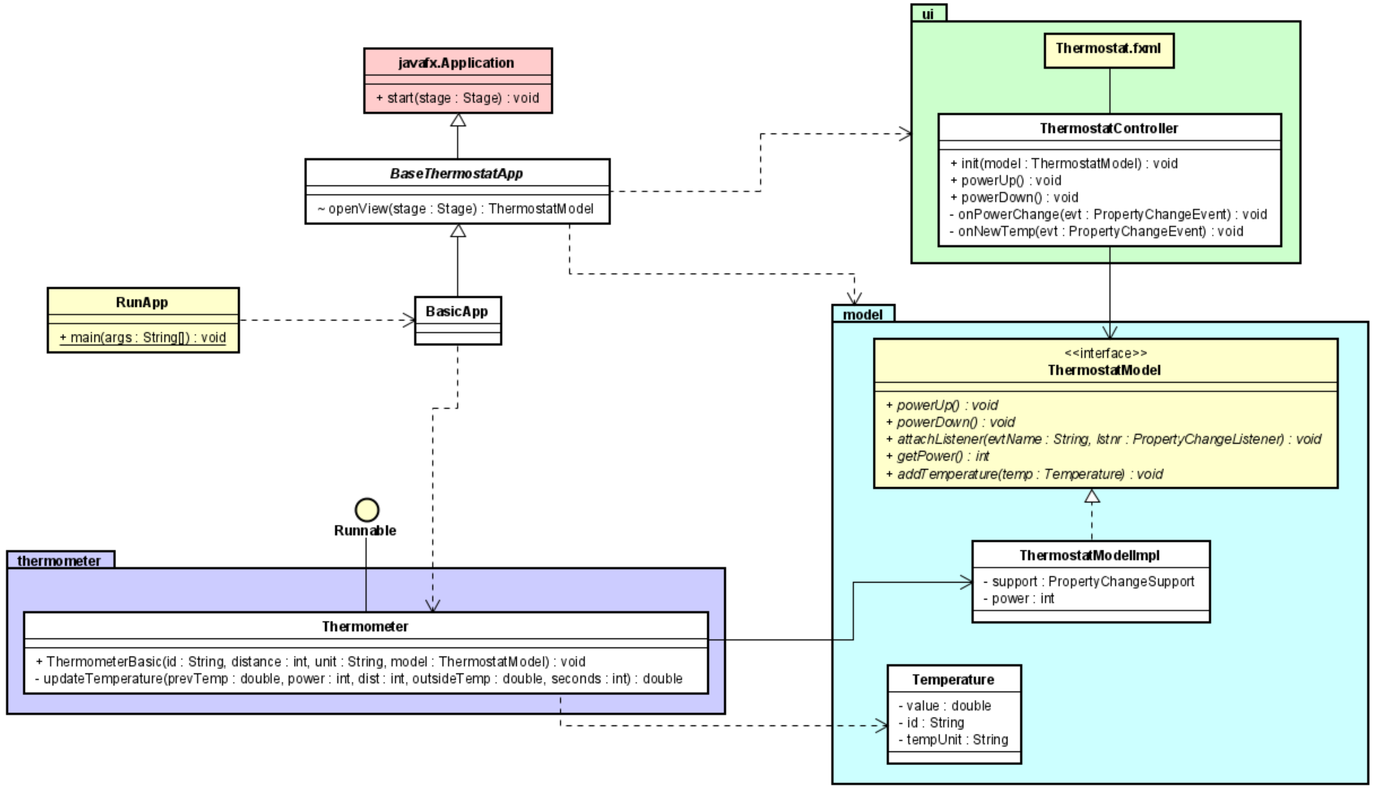
Solution with if statement

```
public ThermometerSwitch(String id, int distance, String unit,
                         ThermostatModel model, boolean isOutside) {
  this.id = id;
  this.distance = distance;
  this.unit = unit;
  this.model = model;
  this.isOutside = isOutside;
}

@Override
public void run() {
  while (true) {

    if (isOutside) { // static outside temp
      temp = 0;
    } else {
      temp = updateTemperature(temp, model.getPower(), distance, 0, 4);
    }

    model.addTemperature(new Temperature(temp, id, unit));
    try {
      Thread.sleep(4000);
    } catch (InterruptedException ignored) {
    }
  }
}
```

Extra argument

Setting field

condition



ui
Thermostat.fxml
ThermostatController
+ init(model : ThermostatModel) : void
+ powerUp() : void
+ powerDown() : void
- onPowerChange(evt : PropertyChangeEvent) : void
- onNewTemp(evt : PropertyChangeEvent) : void
javafx.Application
+ start(stage : Stage) : void
BaseThermostatApp
~ openView(stage : Stage) : ThermostatModel
RunApp
+ main(args : String[]) : void
BasicApp
model
<<interface>>
ThermostatModel
+ powerUp() : void
+ powerDown() : void
+ attachListener(evtName : String, lstnr : PropertyChangeListener) : void
+ getPower() : int
+ addTemperature(temp : Temperature) : void
ThermostatModelImpl
- support : PropertyChangeSupport
- power : int
Runnable
thermometer
Thermometer
+ ThermometerBasic(id : String, distance : int, unit : String, model : ThermostatModel) : void
- updateTemperature(prevTemp : double, power : int, dist : int, outsideTemp : double, seconds : int) : double
Temperature
- value : double
- id : String
- tempUnit : String

New requirement

- I want to add support for simulating a **dynamic** outside temperature, e.g. not just hardcoded to 0.

- Solution:
  - Add **another** condition in the run() method, to check if the thermometer should simulate
    - Indoor
    - Static outside
    - Dynamic outside
  - Boolean not enough, introduce enum.

```java
private final ThermostatModel model;
private final Type type;

public enum Type {
    INSIDE,
    OUTSIDE_STATIC,
    OUTSIDE_DYNAMIC
}

public ThermometerSwitch(String id, int distance, String unit,
                        ThermostatModel model, Type type) {
    this.id = id;
    this.distance = distance;
    this.unit = unit;
    this.model = model;
    this.type = type;
}

@Override
public void run() {
    while (true) {

        if (type == Type.OUTSIDE_STATIC) {                          // static outside temp
            temp = 0;
        } else if (type == Type.OUTSIDE_DYNAMIC) {                  // dynamic outside
            temp = externalTemperature(temp, -5, 5);                // min and max temp
        } else {                                                    // inside temperature
            temp = updateTemperature(temp, model.getPower(), distance, 0, 4);
        }

        model.addTemperature(new Temperature(temp, id, unit));
```

```java
public void run() {
    while (true) {

        if (type == Type.OUTSIDE_STATIC) {                            // static outside temp
            temp = 0;
        } else if (type == Type.OUTSIDE_DYNAMIC) {                    // dynamic outside
            temp = externalTemperature(temp, -5, 5);                  // min and max temp
        } else {                                                      // inside temperature
            temp = updateTemperature(temp, model.getPower(), distance, 0, 4);
        }

        model.addTemperature(new Temperature(temp, id, unit));
        try {
            Thread.sleep(4000);
        } catch (InterruptedException ignored) {
        }
    }
}

public double externalTemperature(double tempPrev, double min, double max) {

    double left = tempPrev - min;

    double right = max - tempPrev;

    int sign = Math.random() * (left + right) > left ? 1 : -1;

    tempPrev += sign * Math.random();

    return tempPrev;
```

```java
public class ThermometerSwitch implements Runnable {

    private double temp;
    private final String id;
    private final int distance;
    private final String unit;
    private final ThermostatModel model;
    private final Type type;

    public enum Type {
        INSIDE,
        OUTSIDE_STATIC,
        OUTSIDE_DYNAMIC
    }

    public ThermometerSwitch(String id, int distance, String unit,
                       ThermostatModel model, Type type) {
        this.id = id;
        this.distance = distance;
        this.unit = unit;
        this.model = model;
        this.type = type;
    }

    @Override
    public void run() {
        while (true) {

            if (type == Type.OUTSIDE_STATIC) {                  // static outside temp
                temp = 0;
            } else if (type == Type.OUTSIDE_DYNAMIC) {          // dynamic outside
                temp = externalTemperature(temp, -5, 5);        // min and max temp
            } else {                                            // inside temperature
                temp = updateTemperature(temp, model.getPower(), distance, 0, 4);
            }

            model.addTemperature(new Temperature(temp, id, unit));
            try {
                Thread.sleep(4000);
            } catch (InterruptedException ignored) {
            }
        }
    }

    public double externalTemperature(double tempPrev, double min, double max) {

        double left = tempPrev - min;

        double right = max - tempPrev;

        int sign = Math.random() * (left + right) > left ? 1 : -1;

        tempPrev += sign * Math.random();

        return tempPrev;

    }

    private double updateTemperature(double tempPrev, int power, int dist, double outsideTemp, int seconds) {

        double tMax = Math.min(11 * power + 10, 11 * power + 10 + outsideTemp);

        tMax = Math.max(Math.max(tem pPrev, tMax), outsideTemp);

        double heaterTerm = 0;

        if (power > 0) {

            double den = Math.max((tMax * (20 - 5 * power) * (dist + 5)), 0.1);

            heaterTerm = 30 * seconds * Math.abs(tMax - tempPrev) / den;

        }

        double outdoorTerm = (tempPrev - outsideTemp) * seconds / 250.0;

        tempPrev = Math.min(Math.max(tem pPrev - outdoorTerm + heaterTerm, outsideTemp), tMax);

        return tempPrev;

    }
}
```

Fields

Enum

constructor

Behaviour

calculation

calculation

New requirement

- The company would like to sell these thermometers in both Europe and USA, i.e. we need to be able to show °C and °F.
  To prepare for the future, some scientists might prefer °K (kelvin)

- Solution:
  - More ifs, to check which unit to use, and then convert

Solution, code

Getting complicated

```
public void run() {
    while (true) {

        if (type == Type.OUTSIDE_STATIC) {
            if (unit.equals("F")) {
                temp = 32;
            } else if(unit.equals("K")) {
                temp = 273.15;
            } else {
                temp = 0;
            }
        } else if (type == Type.OUTSIDE_DYNAMIC) {
            if(unit.equals("F")) {
                temp = 32 + externalTemperature(temp, -5, 5) * 1.8;
            } else if(unit.equals("K")) {
                temp = externalTemperature(temp, -5, 5) + 273.15;
            } else {
                temp = externalTemperature(temp, -5, 5);
            }
        } else {
            if(unit.equals("F")) {
                temp = 32 + updateTemperature(temp, model.getPower(), distance, 0, 4) * 1.8;
            } else if (unit.equals("K")) {
                temp = updateTemperature(temp, model.getPower(), distance, 0, 4) + 273.15;
            } else {
                temp = updateTemperature(temp, model.getPower(), distance, 0, 4);
            }
        }

        model.addTemperature(new Temperature(temp, id, unit));
        try {
```

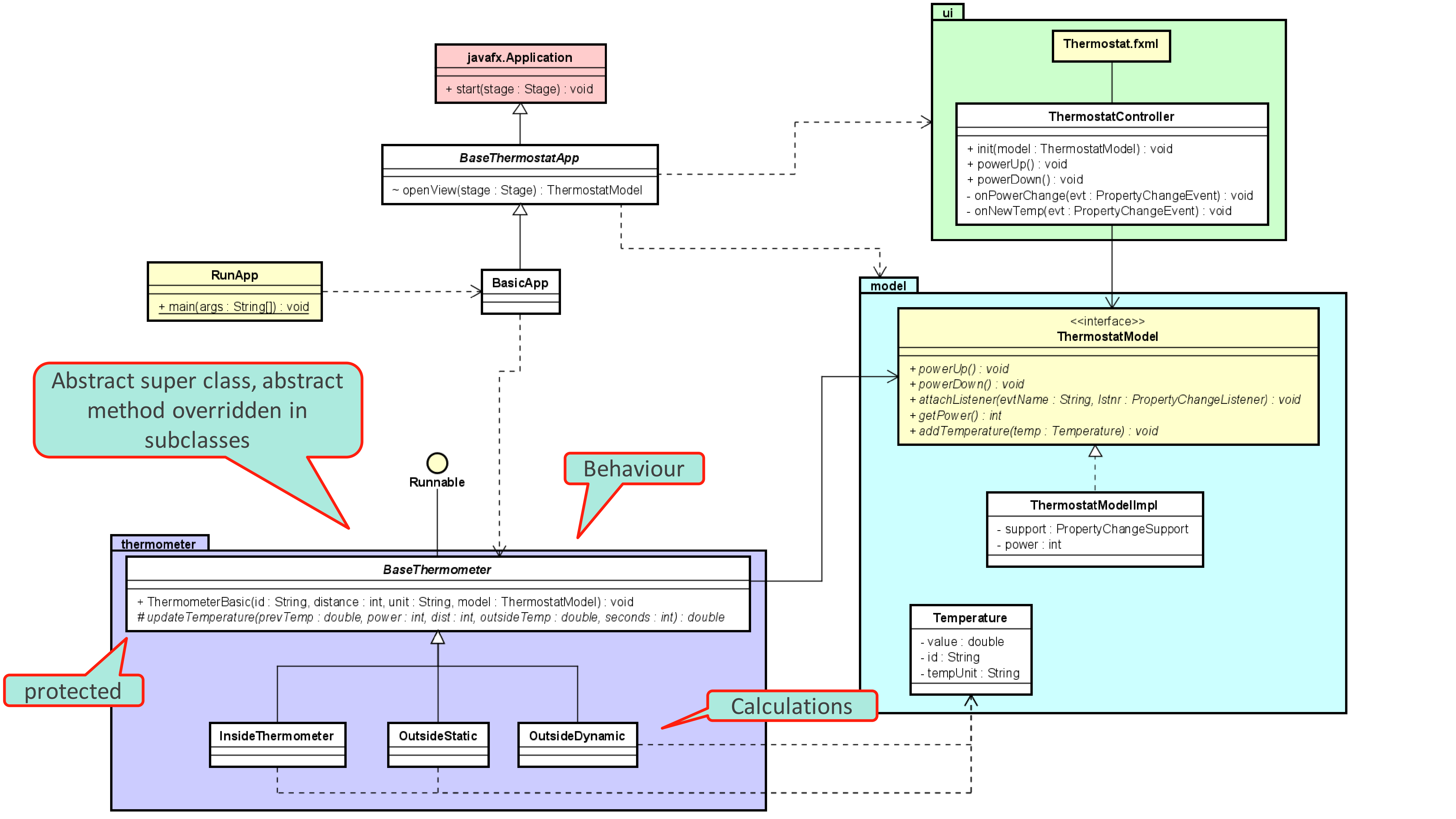
Agenda

- Example introduction, the thermostat
- New requirement, solving it with
  - if-statements
  - inheritance
  - composition
- Composition vs inheritance
- The Strategy pattern
  - General UML
  - Pros/cons
- State vs Strategy

Problem

- We’ve bloated our code with conditionals. It might have been structured slightly more clever, but still.
- Harder to maintain, harder to understand, harder to expand upon. (remember the same problem with state pattern)

- What now?
  - Refactor to use inheritance
  - Incapsulate specific behavior in subclasses
  - (ignore unit for now, for simplicity)



javafx.Application

+ start(stage : Stage) : void

BaseThermostatApp

~ openView(stage : Stage) : ThermostatModel

RunApp

+ main(args : String[]) : void

BasicApp

ui

Thermostat.fxml

ThermostatController

+ init(model : ThermostatModel) : void
+ powerUp() : void
+ powerDown() : void
- onPowerChange(evt : PropertyChangeEvent) : void
- onNewTemp(evt : PropertyChangeEvent) : void

model

<<interface>>
ThermostatModel

+ powerUp() : void
+ powerDown() : void
+ attachListener(evtName : String, lstnr : PropertyChangeListener) : void
+ getPower() : int
+ addTemperature(temp : Temperature) : void

ThermostatModelImpl

- support : PropertyChangeSupport
- power : int

Temperature

- value : double
- id : String
- tempUnit : String

Runnable

thermometer

BaseThermometer

+ ThermometerBasic(id : String, distance : int, unit : String, model : ThermostatModel) : void
# updateTemperature(prevTemp : double, power : int, dist : int, outsideTemp : double, seconds : int) : double

InsideThermometer

OutsideStatic

OutsideDynamic

Abstract super class, abstract method overridden in subclasses

Behaviour

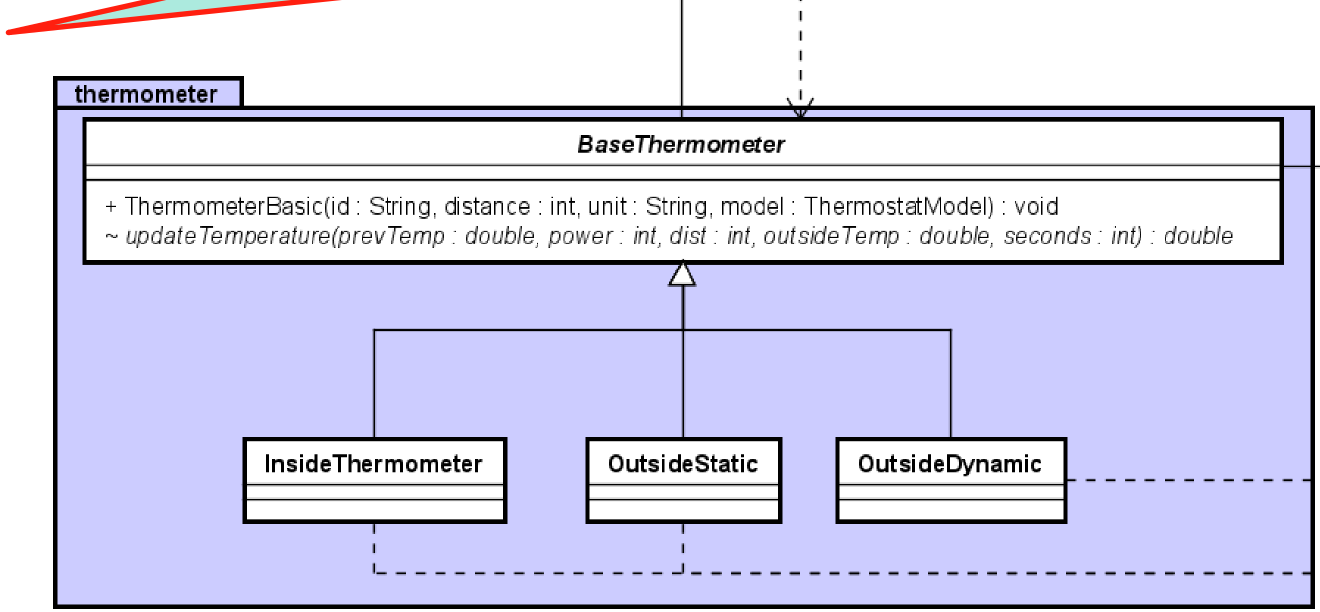
protected

Calculations

Code, BaseThermometer

```java
public abstract class BaseThermometer implements Runnable {

   private double temp;
   private final String id, unit;
   private final int distance;
   private final ThermostatModel model;

   public BaseThermometer(String id, int distance, String unit, ThermostatModel model) {
      this.id = id;
      this.distance = distance;
      this.unit = unit;
      this.model = model;
   }

   protected abstract double updateTemperature(double tempPrev, int power, int dist, double outsideTemp, int seconds);

   @Override
   public void run() {
      while (true) {

         temp = updateTemperature(temp, model.getPower(), distance, 0, 4);

         model.addTemperature(new Temperature(temp, id, unit));
         try {
            Thread.sleep(4000);
         } catch (InterruptedException ignored) {
         }
      }
   }
}
```

Update method implemented in sub-classes


thermometer
BaseThermometer
+ ThermometerBasic(id : String, distance : int, unit : String, model : ThermostatModel) : void
~ updateTemperature(prevTemp : double, power : int, dist : int, outsideTemp : double, seconds : int) : double
InsideThermometer
OutsideStatic
OutsideDynamic

Code, Sub-thermometers

```
public class DynamicOutsideThermometer extends BaseThermometer{

    private double min;
    private double max;

    public DynamicOutsideThermometer(String id, int distance, String unit, ThermostatModel model, double min, double max) {
        super(id, distance, unit, model);
        this.min = min;
        this.max = max;
    }

    @Override
    protected double updateTemperature(double tempPrev, int power, int dist, double outsideTemp, int seconds) {
        double left = tempPrev - min;

        double right = max - tempPrev;

        int sign = Math.random() * (left + right) > left ? 1 : -1;

        tempPrev += sign * Math.random();

        return tempPrev;
    }
}
```

```
public class StaticOutsideThermometer extends BaseThermometer{
    public StaticOutsideThermometer(String id, int distance, String unit, ThermostatModel model) {
        super(id, distance, unit, model);
    }

    @Override
    protected double updateTemperature(double tempPrev, int power, int dist, double outsideTemp, int seconds) {
        return 0;
    }
}
```

Code, Sub-thermometers

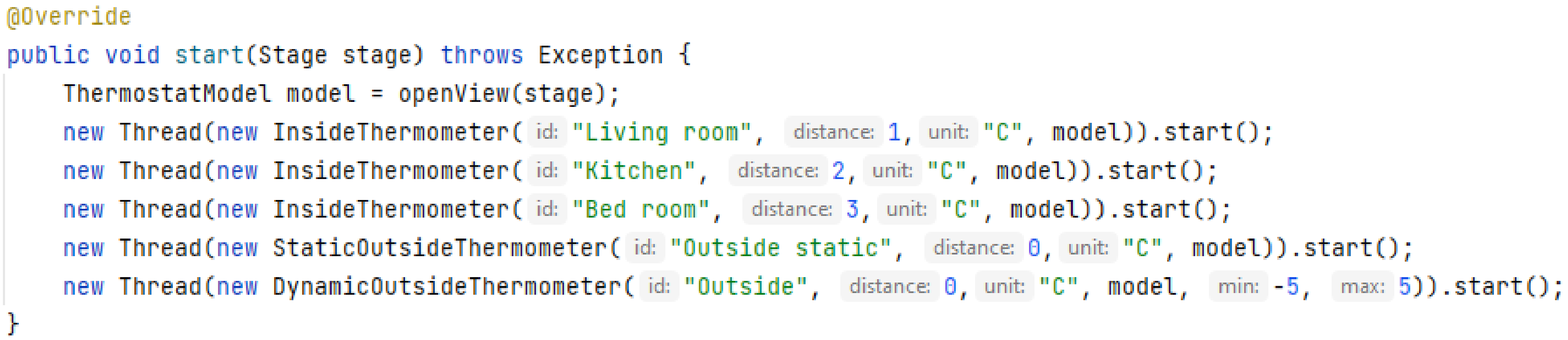
```
public class InsideThermometer extends BaseThermometer{

    public InsideThermometer(String id, int distance, String unit, ThermostatModel model) {
        super(id, distance, unit, model);
    }

    @Override
    protected double updateTemperature(double tempPrev, int power, int dist, double outsideTemp, int seconds) {
        double tMax = Math.min(11 * power + 10, 11 * power + 10 + outsideTemp);

        tMax = Math.max(Math.max(tempPrev, tMax), outsideTemp);

        double heaterTerm = 0;

        if (power > 0) {

            double den = Math.max((tMax * (20 - 5 * power) * (dist + 5)), 0.1);

            heaterTerm = 30 * seconds * Math.abs(tMax - tempPrev) / den;

        }

        double outdoorTerm = (tempPrev - outsideTemp) * seconds / 250.0;

        tempPrev = Math.min(Math.max(tempPrev - outdoorTerm + heaterTerm, outsideTemp), tMax);

        return tempPrev;
    }
}
```

Good solution?

- Can be just fine.
- Code is isolated
- Easier to maintain because of separation
- Easier to expand with more sub-classes, if needed.
- Support for other units?
  Either:
  1. Introduce if-statements in each current class
  2. Create 3*3 classes, i.e. 3 types of thermometer times 3 types of units. Can grow large
  3. Introduce an abstract Unit converter class, with subclasses that converts to C, K, or F. (out of scope)
  4. Template method design pattern (out of scope)

Bad solution?

- We’ve locked our inheritance. May not be a problem, might be in the future.
- Inheritance can be un-flexible, rigid.
  - What if the program was started with StaticOutdoor, but we would at some point want to switch?



```
@Override
public void start(Stage stage) throws Exception {
    ThermostatModel model = openView(stage);
    new Thread(new InsideThermometer( id: "Living room",  distance: 1, unit: "C", model)).start();
    new Thread(new InsideThermometer( id: "Kitchen",  distance: 2, unit: "C", model)).start();
    new Thread(new InsideThermometer( id: "Bed room",  distance: 3, unit: "C", model)).start();
    new Thread(new StaticOutsideThermometer( id: "Outside static",  distance: 0, unit: "C", model)).start();
    new Thread(new DynamicOutsideThermometer( id: "Outside",  distance: 0, unit: "C", model,  min: -5,  max: 5)).start();
}
```

  - I would need to terminate current thread, and start a new DynamicOutsideThermometer. This can be cumbersome

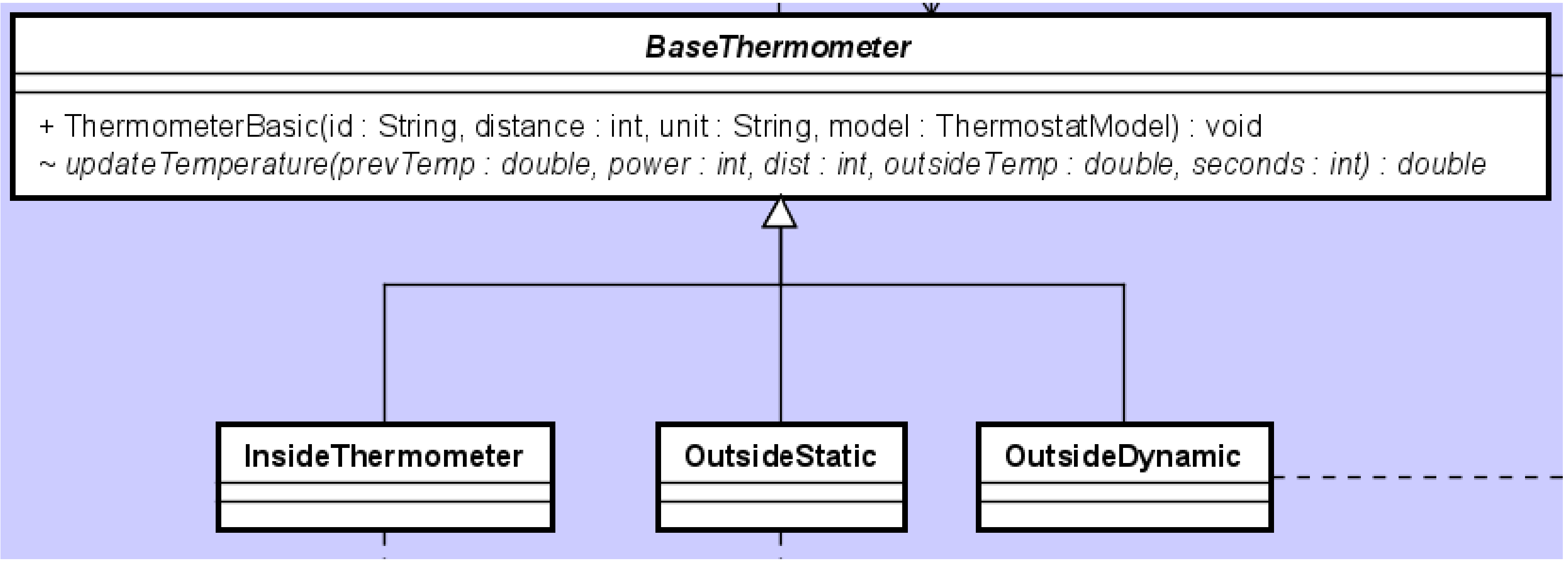
- Changes to super class can cause problems for sub-class

Agenda

- Example introduction, the thermostat
- New requirement, solving it with
  - if-statements
  - inheritance
  - composition
- Composition vs inheritance
- The Strategy pattern
  - General UML
  - Pros/cons
- State vs Strategy

Composition, idea

– Currently the behavior is in the super-class, and the calculations are specified in the sub-class.

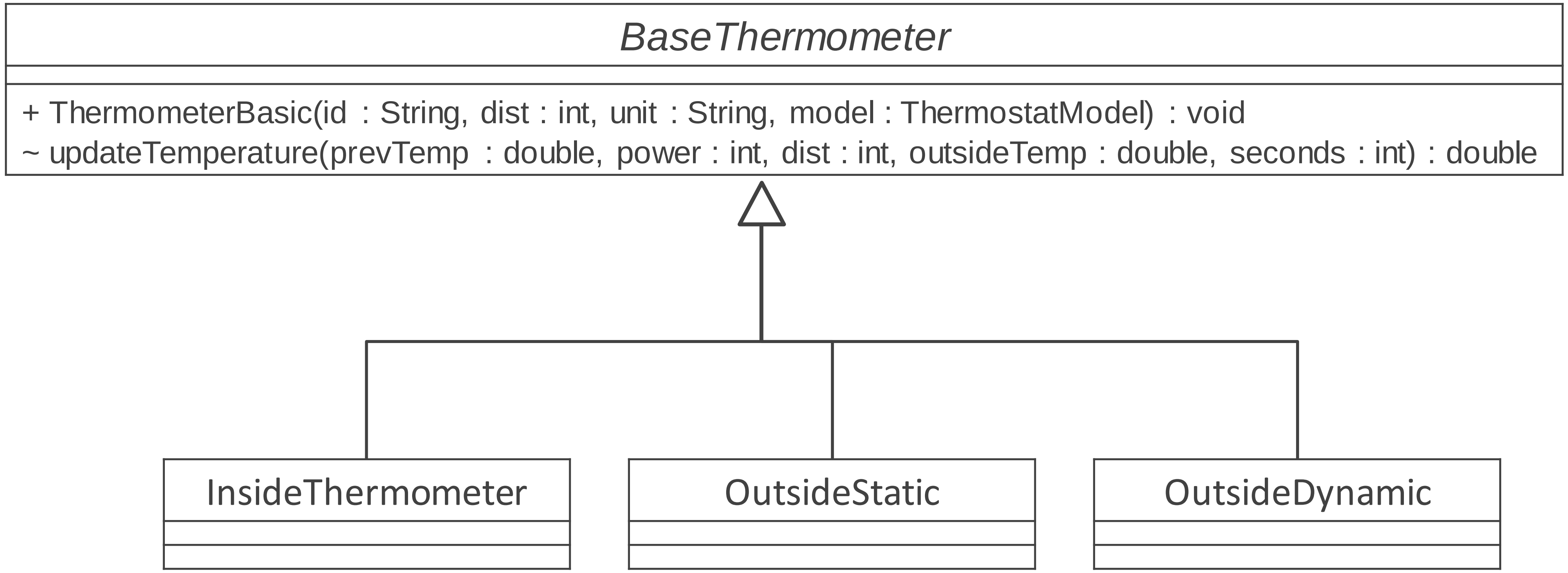


BaseThermometer
+ ThermometerBasic(id : String, distance : int, unit : String, model : ThermostatModel) : void
~ updateTemperature(prevTemp : double, power : int, dist : int, outsideTemp : double, seconds : int) : double
InsideThermometer
OutsideStatic
OutsideDynamic

Composition, idea

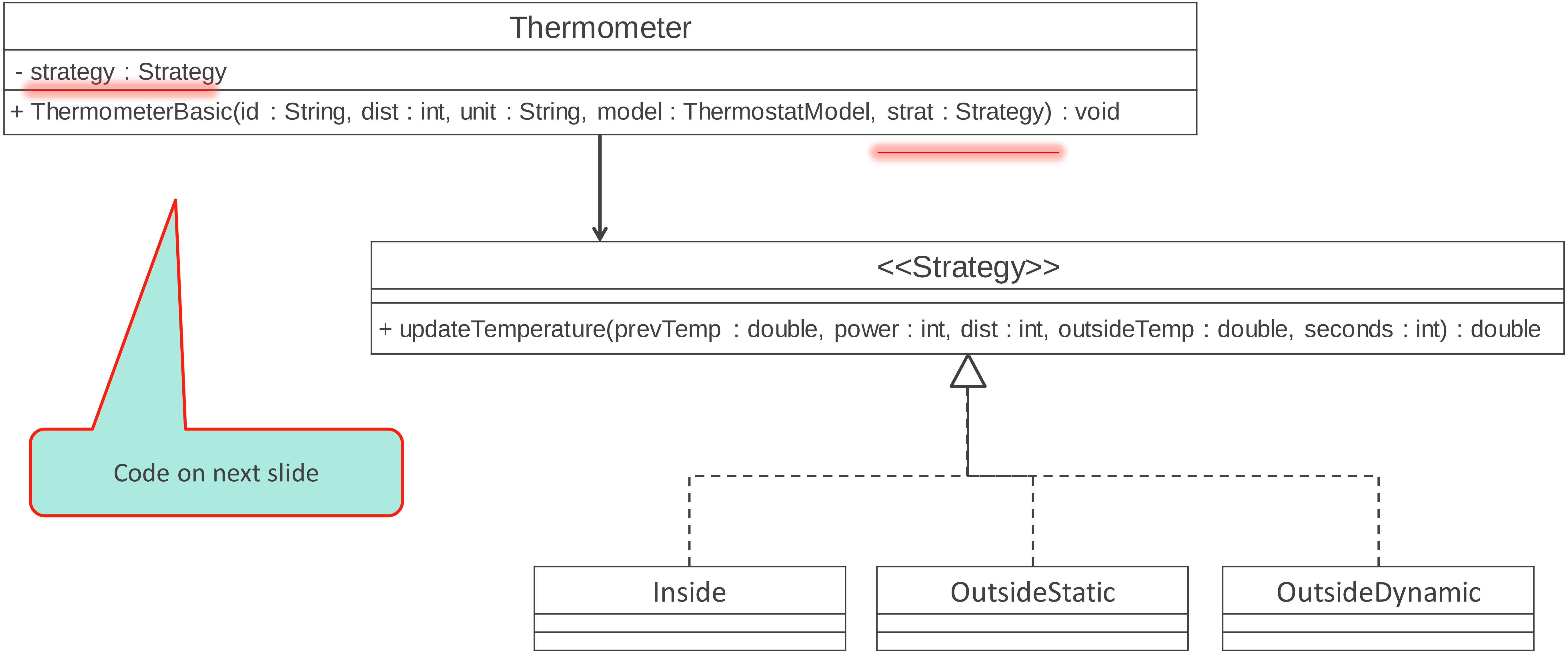
- Let’s introduce an interface with the calculation-method. Classes can then implement how the calculation is handled.
- Our thermometer can delegate the calculation to this interface.
(sort of similar to how StateContext delegated to state implementations)

Current setup



BaseThermometer
+ ThermometerBasic(id : String, dist : int, unit : String, model : ThermostatModel) : void
~ updateTemperature(prevTemp : double, power : int, dist : int, outsideTemp : double, seconds : int) : double
InsideThermometer
OutsideStatic
OutsideDynamic

New setup, using composition


Thermometer
- strategy : Strategy
+ ThermometerBasic(id : String, dist : int, unit : String, model : ThermostatModel, strat : Strategy) : void
<<Strategy>>
+ updateTemperature(prevTemp : double, power : int, dist : int, outsideTemp : double, seconds : int) : double
Inside
OutsideStatic
OutsideDynamic

Code on next slide

Code

Inheritance

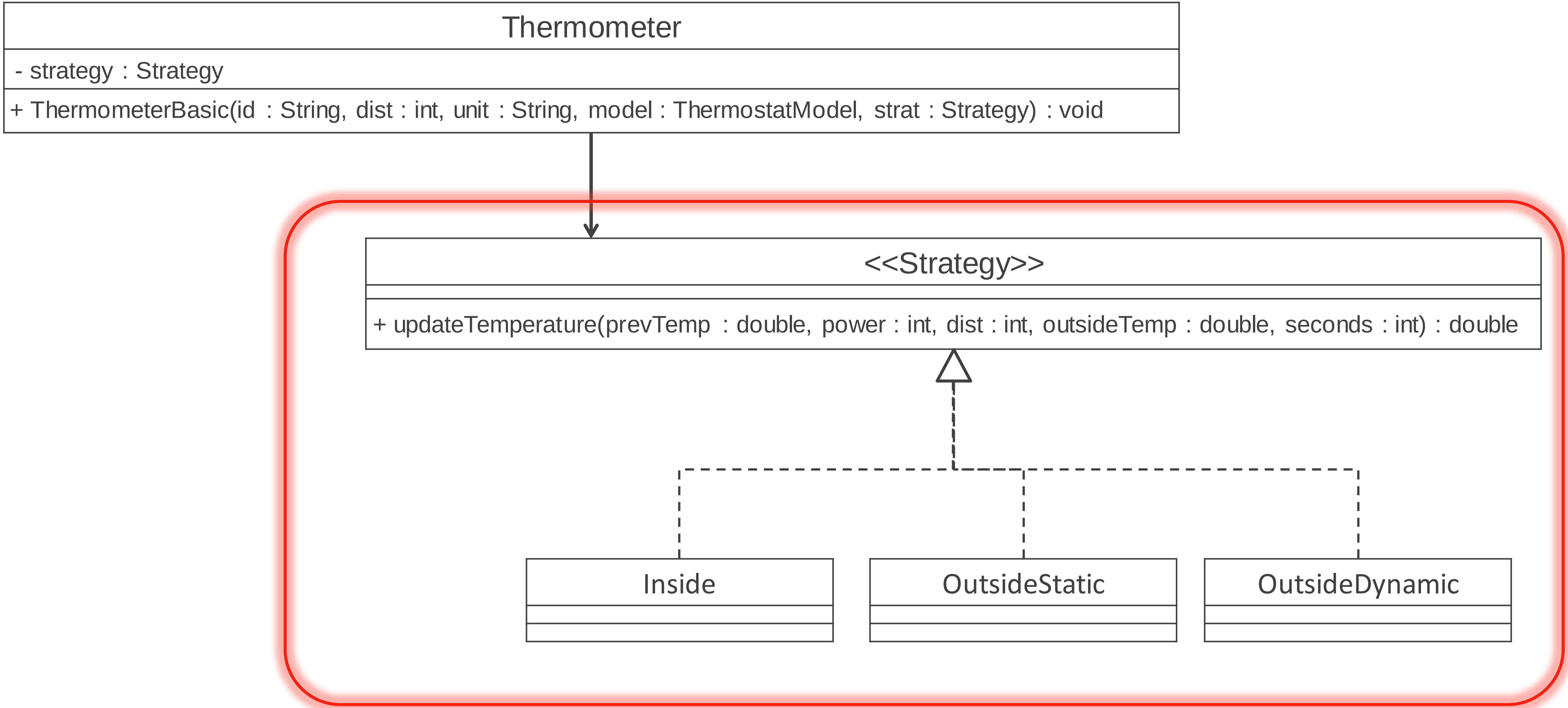
```
public abstract class BaseThermometer implements Runnable {

  private double temp;
  private final String id, unit;
  private final int distance;
  private final ThermostatModel model;

  public BaseThermometer(String id, int distance, String unit, ThermostatModel model) {
    this.id = id;
    this.distance = distance;
    this.unit = unit;
    this.model = model;
  }

  protected abstract double updateTemperature(double tempPrev, int power,
                                 int dist, double outsideTemp, int seconds);

  @Override
  public void run() {
    while (true) {

      temp = updateTemperature(temp, model.getPower(), distance, 0, 4);

      model.addTemperature(new Temperature(temp, id, unit));
      try {
        Thread.sleep(4000);
      } catch (InterruptedException ignored) {
      }
    }
  }
}
```

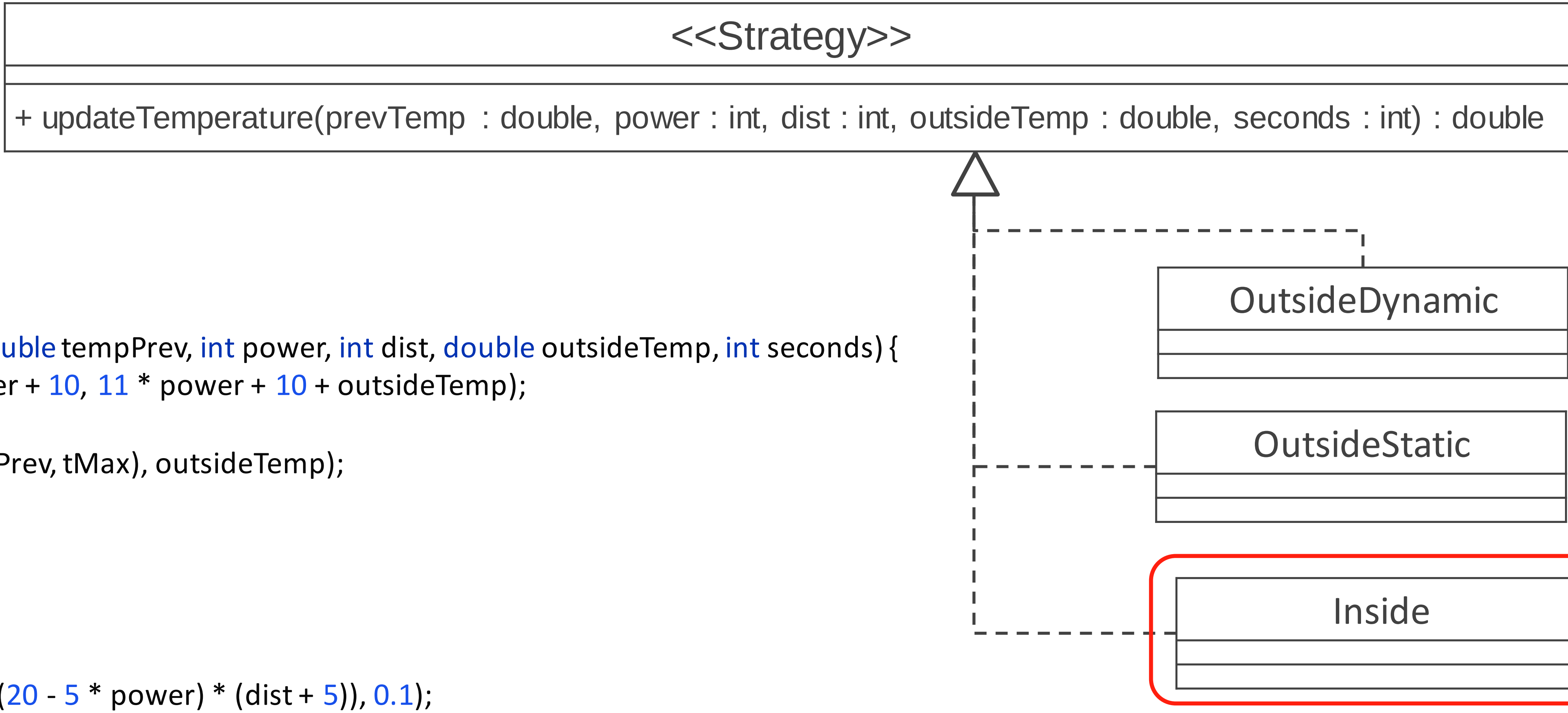
Composition

```
public interface ThermometerStrategy {
  double updateTemperature(double tempPrev, int power,
                    int dist, double outsideTemp, int seconds);
}
```

```
public class Thermometer implements Runnable {

  private double temp;
  private final String id, unit;
  private final int distance;
  private final ThermostatModel model;
  private ThermometerStrategy strategy;

  public Thermometer(String id, int distance, String unit,
               ThermostatModel model, ThermometerStrategy strategy) {
    this.id = id;
    this.distance = distance;
    this.unit = unit;
    this.model = model;
    this.strategy = strategy;
  }


  @Override
  public void run() {
    while (true) {

      temp = strategy.updateTemperature(temp, model.getPower(), distance, 0, 4);

      model.addTemperature(new Temperature(temp, id, unit));
      try {
        Thread.sleep(4000);
      } catch (InterruptedException ignored) {
      }
    }
  }
```

Code for strategy implementations


Thermometer
- strategy : Strategy
+ ThermometerBasic(id : String, dist : int, unit : String, model : ThermostatModel, strat : Strategy) : void
<<Strategy>>
+ updateTemperature(prevTemp : double, power : int, dist : int, outsideTemp : double, seconds : int) : double
Inside
OutsideStatic
OutsideDynamic



<<Strategy>>

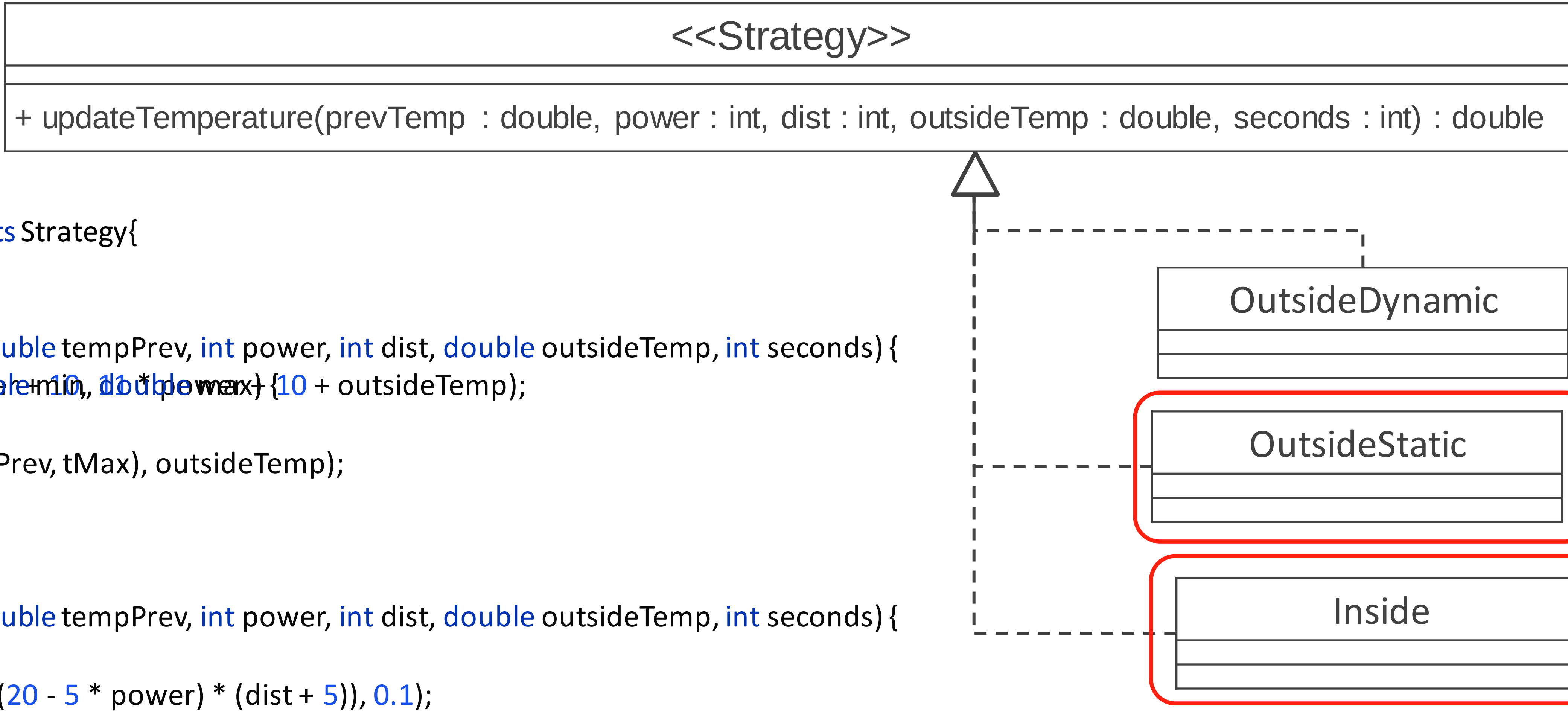
+ updateTemperature(prevTemp : double, power : int, dist : int, outsideTemp : double, seconds : int) : double

OutsideDynamic

OutsideStatic

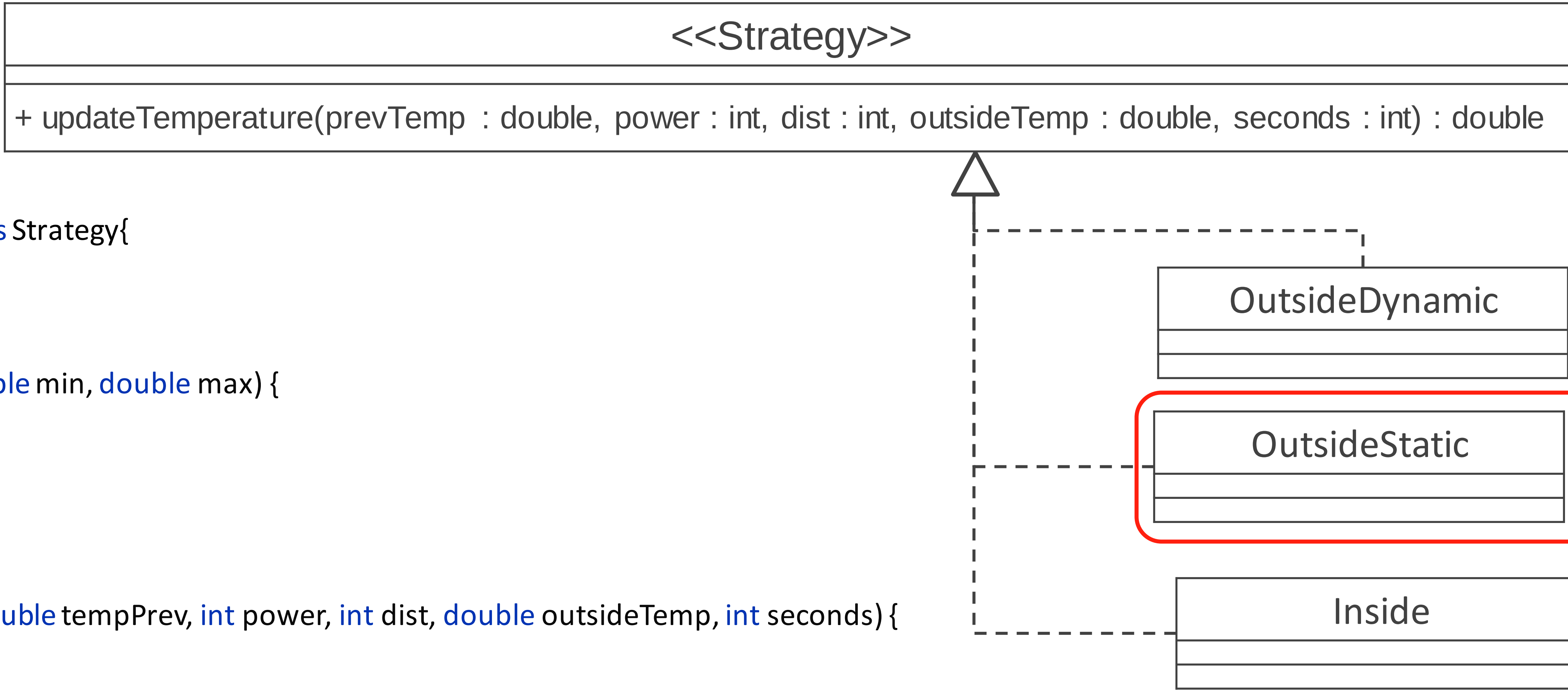
Inside

```java
public class Inside implements Strategy
{
  @Override
  public double updateTemperature(double tempPrev, int power, int dist, double outsideTemp, int seconds) {
    double tMax = Math.min(11 * power + 10, 11 * power + 10 + outsideTemp);

    tMax = Math.max(Math.max(tempPrev, tMax), outsideTemp);

    double heaterTerm = 0;

    if (power > 0) {

      double den = Math.max((tMax * (20 - 5 * power) * (dist + 5)), 0.1);

      heaterTerm = 30 * seconds * Math.abs(tMax - tempPrev) / den;

    }

    double outdoorTerm = (tempPrev - outsideTemp) * seconds / 250.0;

    tempPrev = Math.min(Math.max(tempPrev - outdoorTerm + heaterTerm, outsideTemp), tMax);

    return tempPrev;
  }
}
```


<<Strategy>>
+ updateTemperature(prevTemp : double, power : int, dist : int, outsideTemp : double, seconds : int) : double
OutsideDynamic
OutsideStatic
Inside

```
public class OutsideDynamic implements Strategy{
{
    private final double min, max;
    public OutsideDynamicStrategy(double min, double max) {
        this.min = min;
        this.max = max;
    }
    @Override
    public double updateTemperature(double tempPrev, int power, int dist, double outsideTemp, int seconds) {
        double left = tempPrev - min;
        double right = max - tempPrev;
        int sign = Math.random() * (left + right) > left ? 1 : -1;
        tempPrev += sign * Math.random();
        return tempPrev;
    }
}
```

```
public class Inside implements Strategy{
{
    @Override
    public double updateTemperature(double tempPrev, int power, int dist, double outsideTemp, int seconds) {
        double tMax = Math.min(11 * power + 10, 10 + outsideTemp);
        tMax = Math.max(Math.max(tempPrev, tMax), outsideTemp);
        double heaterTerm = 0;
        if (power > 0) {
            double den = Math.max((tMax * (20 - 5 * power) * (dist + 5)), 0.1);
            heaterTerm = 30 * seconds * Math.abs(tMax - tempPrev) / den;
        }
        double outdoorTerm = (tempPrev - outsideTemp) * seconds / 250.0;
        tempPrev = Math.min(Math.max(tempPrev - outdoorTerm + heaterTerm, outsideTemp), tMax);
        return tempPrev;
    }
}
```



<<Strategy>>
+ updateTemperature(prevTemp : double, power : int, dist : int, outsideTemp : double, seconds : int) : double
OutsideDynamic
OutsideStatic
Inside

```
public class OutsideDynamicimplements Strategy{

   private final double min, max;

   public OutdoorDynamicStrategy(double min, double max) {
      this.min = min;
      this.max = max;
   }

   @Override
   public double updateTemperature(double tempPrev, int power, int dist, double outsideTemp, int seconds) {
      double left = tempPrev - min;

      double right = max - tempPrev;

      int sign = Math.random() * (left + right) > left ? 1 : -1;

      tempPrev += sign * Math.random();

      return tempPrev;
   }
}
```

Using it?

```
@Override
public void start(Stage stage) throws Exception {

    ThermostatModel model = openView(stage);

    Strategy indoorStrategy = new IndoorStrategy();
    Strategy outDynamic = new OutdoorDynamicStrategy(-5, 5);
    Strategy outStatic = new OutdoorStaticStrategy();

    new Thread(new Thermometer("Living room", 1,"C", model, indoorStrategy)).start();
    new Thread(new Thermometer("Kitchen", 2,"C", model, indoorStrategy)).start();
    new Thread(new Thermometer("Bed room", 3,"C", model, indoorStrategy)).start();
    new Thread(new Thermometer("Outside static", 0,"C", model, outStatic)).start();
    new Thread(new Thermometer("Outside", 0,"C", model, outDynamic)).start();

}
```

Agenda

- Example introduction, the thermostat
- New requirement, solving it with
  - if-statements
  - inheritance
  - composition
- Composition vs inheritance
- The Strategy pattern
  - General UML
  - Pros/cons
- State vs Strategy

Composition vs inheritance

“has a” vs “is a”

- It’s a design principle: prefer composition over inheritance (but not always)
- Inheritance is limited: no multiple inheritances in Java. No such problem with composition
- Loose coupling vs tight coupling
- Composition improves testability with unit tests (later in course)
- More flexible: I could change the strategy on the fly, but the same change with inheritance would require stopping the current thread and starting a new.
- A change in a super class affects the sub class. This can lead to problems.
- Design a class on what it does *versus* design a class on what it is

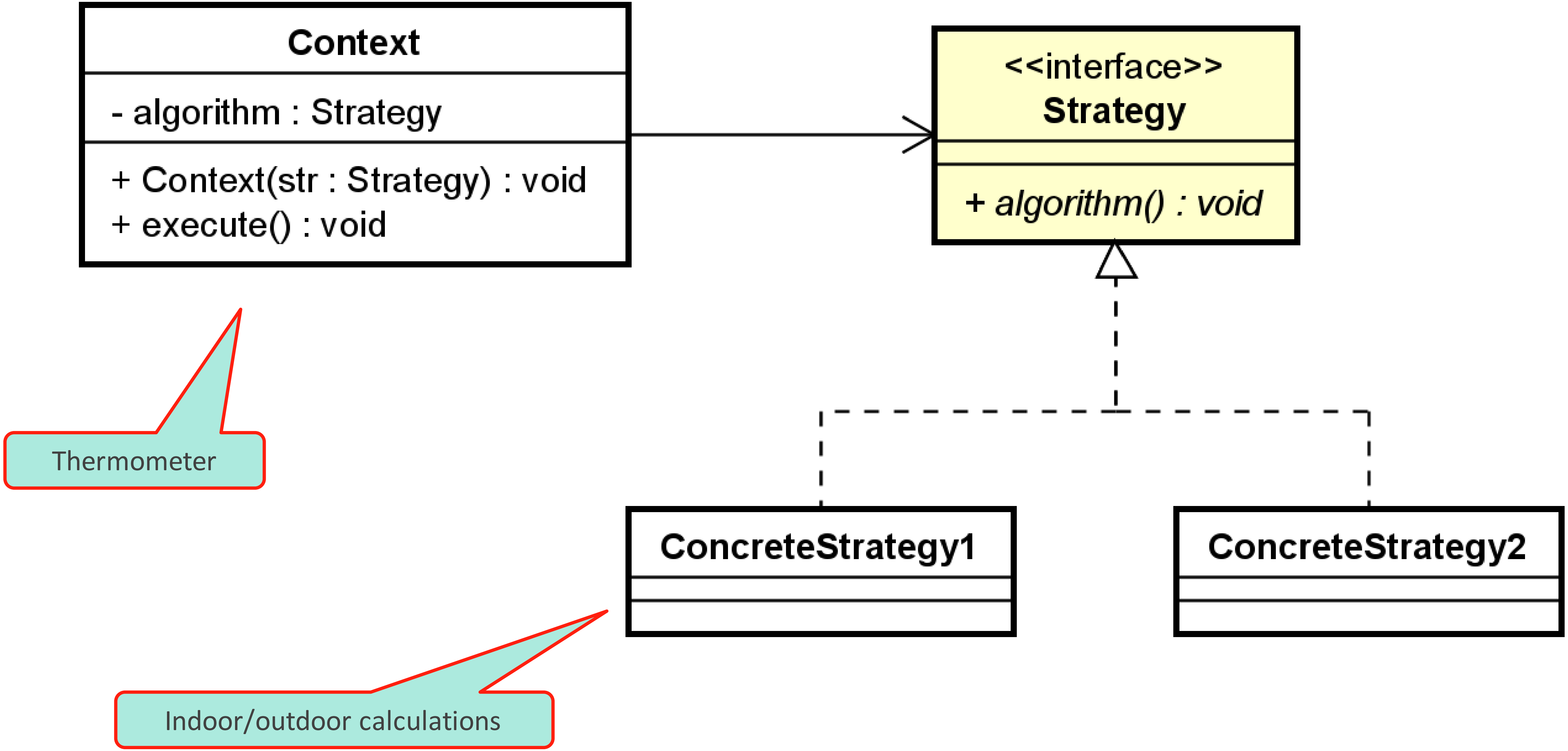
Agenda

- Example introduction, the thermostat
- New requirement, solving it with
  - if-statements
  - inheritance
  - composition
- Composition vs inheritance
- The Strategy pattern
  - General UML
  - Pros/cons
- State vs Strategy

Strategy pattern

- Problem:
  - Your product must support variable algorithms or business rules, and you want a flexible and reliable way of controlling the variability.
- Intent
  - Define a family of algorithms, encapsulate each one, and make them interchangeable
  - Easily change algorithms at runtime
- Solution
  - Encapsulate each algorithm in a class, all sharing an interface. Let the context-class delegate calculations to this interface, so that the implementations are interchangeable.
- Examples:
  - Change tax calculation
  - Change what happens when you click on something (button, etc)
  - Change input field validation
  - Used very often in game development
  - Change google maps route planning (car, bike, walk, public transport)
  - Data export: to JSON, XML, binary, etc.
  - Collections.sort(…) ← you implement your sorting criteria here

Strategy general UML diagram



Context

- algorithm : Strategy

+ Context(str : Strategy) : void
+ execute() : void

<<interface>>
Strategy

+ algorithm() : void

ConcreteStrategy1

ConcreteStrategy2

Thermometer

Indoor/outdoor calculations

Pros and cons

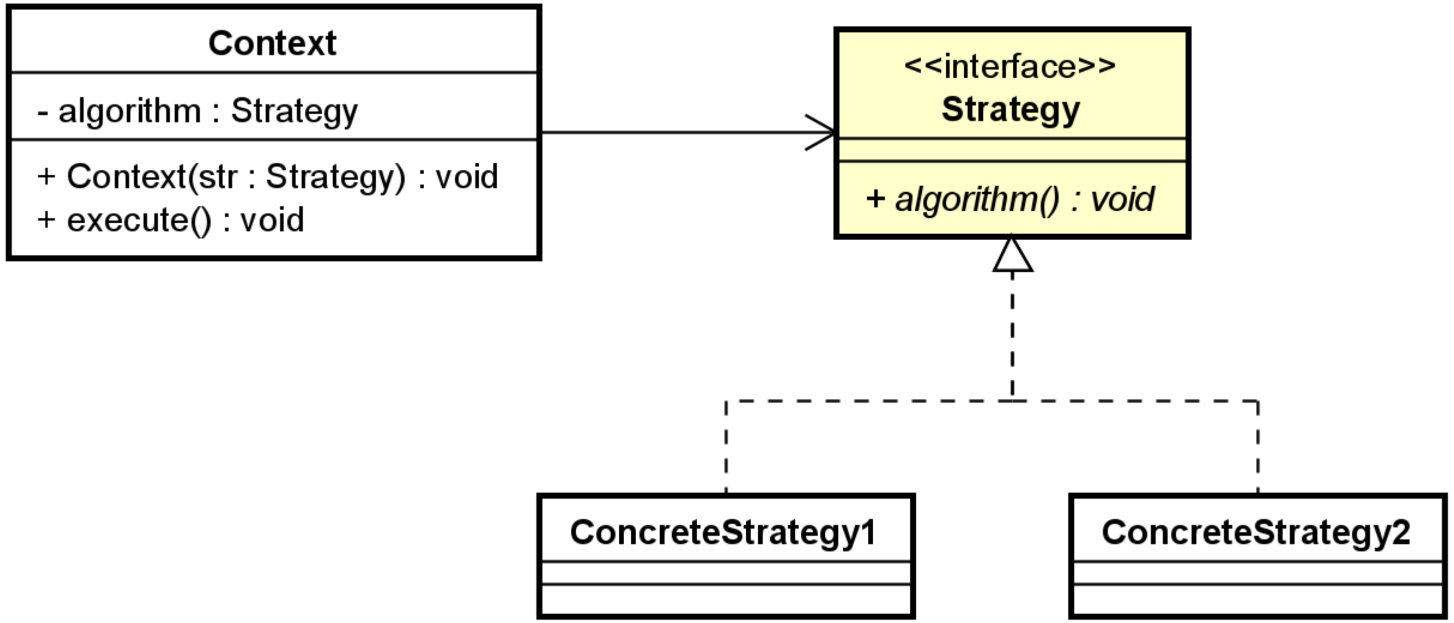
- ✓ Swap algorithms used inside an object at runtime
- ✓ Isolate the implementation details of an algorithm rom the code that uses it
- ✓ Composition over inheritance
- ✓ Open/closed principle
- ✓ Single responsibility regarding algorithms

- ✗ Can overcomplicate a simple class
- ✗ Users of the context must be aware of strategies to select proper one
- ✗ Lambda expression might be simpler, reduces number of classes.

Agenda

- Example introduction, the thermostat
- New requirement, solving it with
  - if-statements
  - inheritance
  - composition
- Composition vs inheritance
- The Strategy pattern
  - General UML
  - Pros/cons
- State vs Strategy

Strategy vs state



Context

- algorithm : Strategy

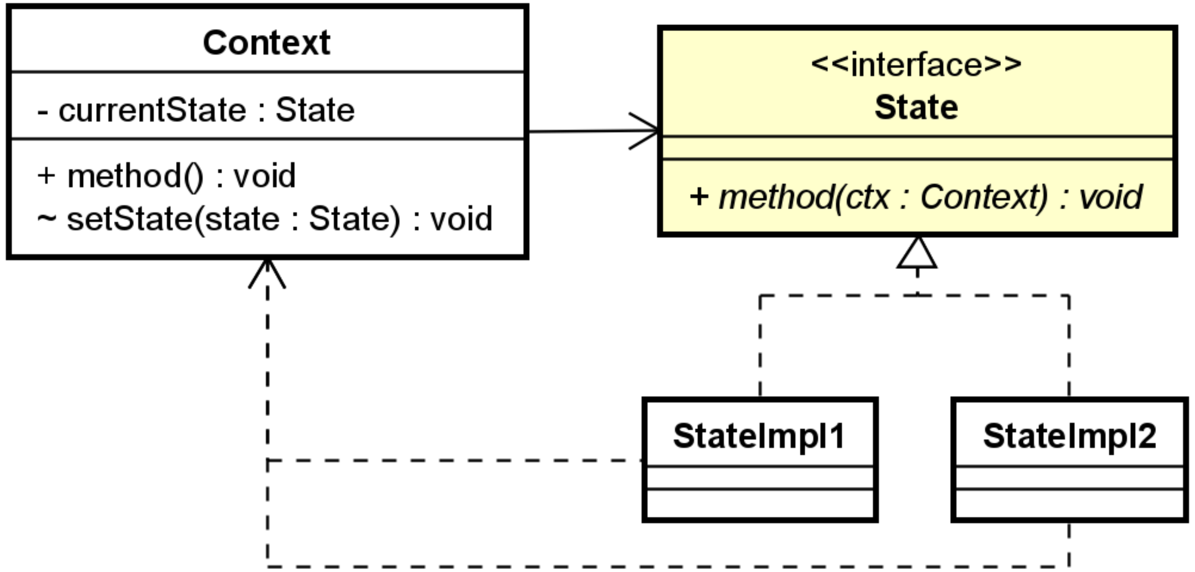
+ Context(str : Strategy) : void
+ execute() : void

<<interface>>
Strategy

+ algorithm() : void

ConcreteStrategy1

ConcreteStrategy2



Context

- currentState : State

+ method() : void
~ setState(state : State) : void

<<interface>>
State

+ method(ctx : Context) : void

StateImpl1

StateImpl2

Strategy vs state, similar in structure, different in intent.

– State:

1. Internal workings determine the state, often quite dynamic
2. State implementations are responsible for changing state.
3. States may work on the Context, i.e. modify data
4. Handles multiple actions per state, i.e. multiple methods
5. Internal Context data may affect the current State
6. States may depend on/reference each other
7. Can be expressed with a state machine
8. State focused
9. ***What*** an object does ***when*** it’s in a given state
10. User of Context knows nothing of internal states

Strategy:

1. Algorithm is set from outside, i.e. as constructor arguments
2. Strategies should not change to other strategies. It’s done from outside the Context
3. Strategies doesn’t know the Context
4. Often more isolated to a single calculation/method. Usually single, specific task.
5. Internal Context data should not affect which strategy is used
6. Strategies are isolated
7. Cannot be expressed with a state machine
8. Algorithm focused
9. ***How*** an object does something
10. User of Context sets the internal strategy

Agenda

- Example introduction, the thermostat
- New requirement, solving it with
  - if-statements
  - inheritance
  - composition
- Composition vs inheritance
- The Strategy pattern
  - General UML
  - Pros/cons
- State vs Strategy

Assignment 2 at 11:20