Object Oriented Software Engineering (COMP2003)

Lecture 7: State

Updated: 7th October, 2015

David Cooper
Department of Computing
Curtin University

Outline

State

- "State" has two closely-related meanings:
  1. Any runtime information.
  2. A specific combination of runtime information that causes the system to behave in a particular way. (We'll come back to this.)
- Runtime information:
  - The values of variables and fields.
  - Object types – i.e. what kind of object is it *really*? (As opposed to the declared reference type.)
  - The "instruction pointer" – i.e. which line of code is currently executing.
- Mostly internal to your program, but can also be stored elsewhere:
  - In a temporary file.
  - In a library/framework/component that your software uses.
  - In the OS/hardware – e.g. the current time of day.

Discrete States

- Sometimes we talk about *a* state (singular), or states (plural).
- We're usually, actually taking about a particular *behaviour*.
- Some systems behave differently under in different conditions.
  - e.g. websites with logins, traffic lights, etc.
- Inevitably, systems use state to decide their behaviour.
  - e.g. a "logged-in" flag, a "colour" variable, etc.
- A *state transition* happens when a change in state causes a change in behaviour.
  - e.g. logging in or out, lights changing colour, etc.

State Transitions

- States change as a result of:
  - A stimulus (e.g. the user logging in), or
  - A timer expiry.
- These state transitions are *part* of the state's behaviour.
  - e.g. A log-out action can only happen in the logged-in state.
  - A traffic light only changes to green from the red state.
- Each state has:
  - A set of possible *incoming* transitions from other states.
  - A set of possible *outgoing* transitions to other states.
- Transitions are instantaneous (theoretically).
  - Of course, this is physically impossible.
  - Transitions do take some time, because everything does.
  - But they are fast enough that we don't notice, or don't care.
- If we *do* care what happens "during" a transition, we instead view it as a separate, intermediate state.

System State vs. Object State

- States can represent the whole application, or individual parts of it.
- Take the traffic light example:
  - A single traffic light can be red, green or amber.
  - But a traffic light is always part of a set, which together have a more complex set of states.
- High level or low level?
  - States and state transitions are part of the software design vocabulary.
  - In other words, use them to communicate ideas.
  - Sometimes these ideas concern the application as a whole.
  - Sometimes they concern individual objects.

Object State – GUI Buttons Example

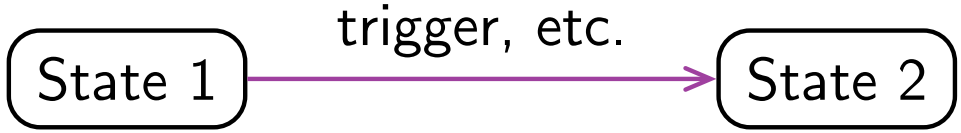
- GUI Buttons have several states:
  - Disabled – greyed-out; not available.
  - Enabled but not selected.
  - Selected but not pressed – say, if you press TAB enough times.
  - Pressed – in the act of being clicked.
- These states are *mutually exclusive* – only one applies at a time.
- However, these states are not the only state.
  - Remember the two meanings of "state".
  - Buttons also have: a textual label, an icon, a keyboard shortcut (hotkey), colour, size, location, etc.
  - This is state information too.
  - But it does not lead to different button behaviours.

What States are Not

- States are not operations.
- Operations (tasks/actions/processes/etc.) are just a bundle of instructions. Some take time, but many are almost instantaneous.
- States are *modes* of operation – different ways in which the system behaves.
- States last for a non-trivial amount of time.
  - They don't just finish once all the work is done.
  - Something must cause them to change.

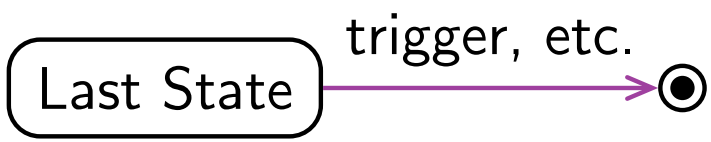
UML State Diagrams (or Statecharts)

- State diagrams represent discrete states graphically.
- States are represented by rounded boxes: (The State)
- Arrows represent transitions between states:



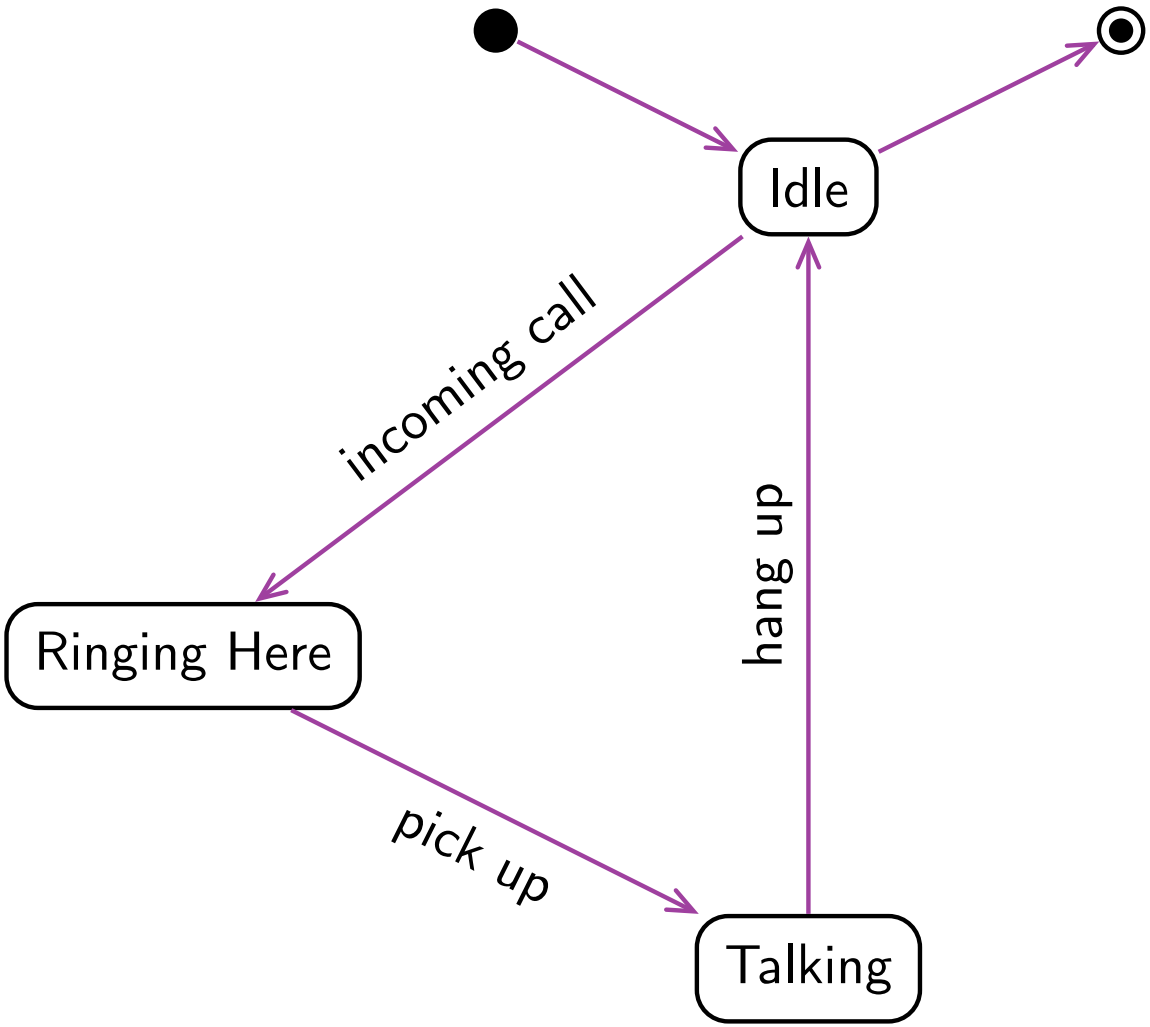
- There is always an "initial pseudostate": "●".

- There is *sometimes* a "final pseudostate": "◉".



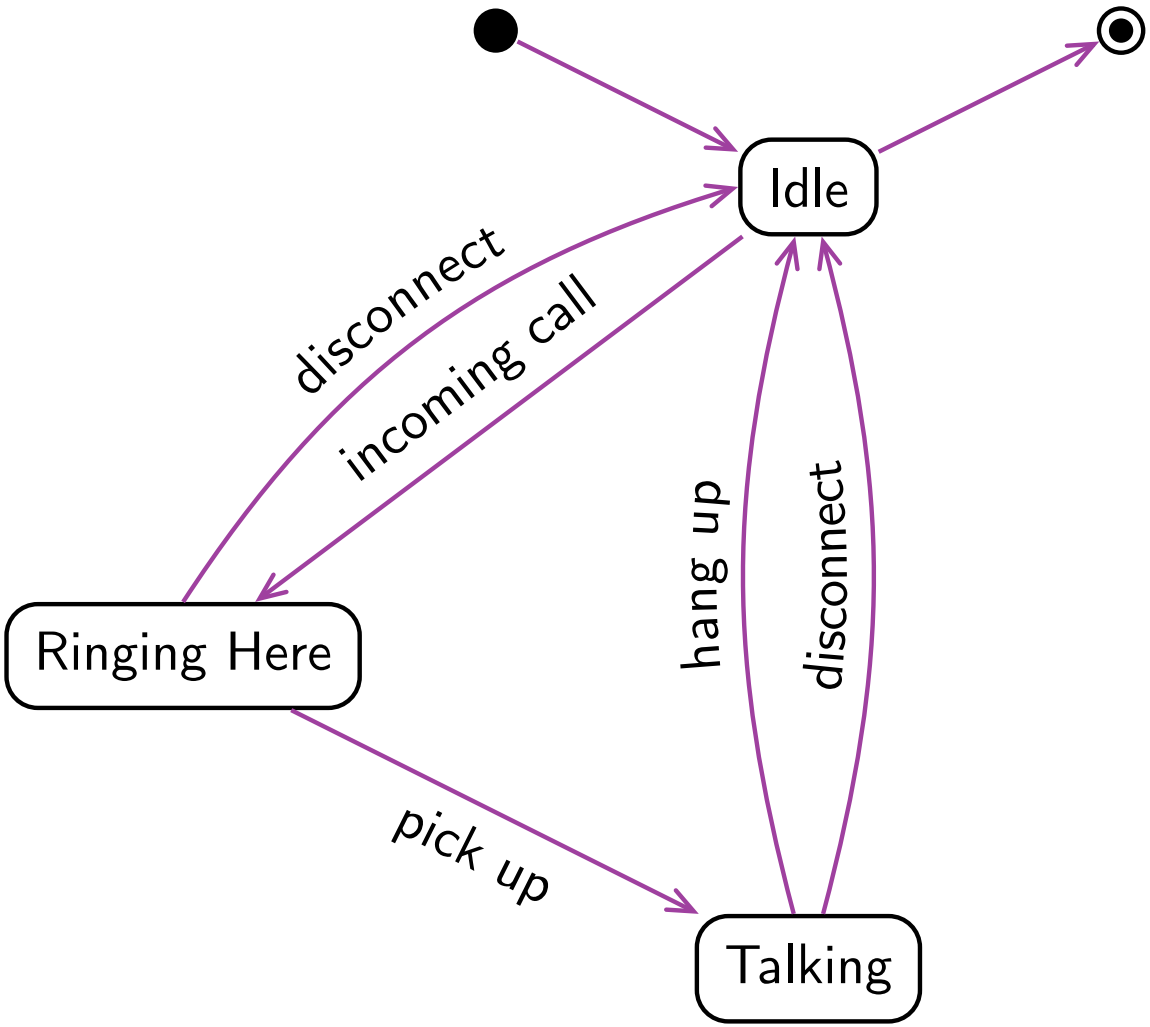
- ● and ◉ are not real states.
  - They just indicate where things start and stop.

Phone Example



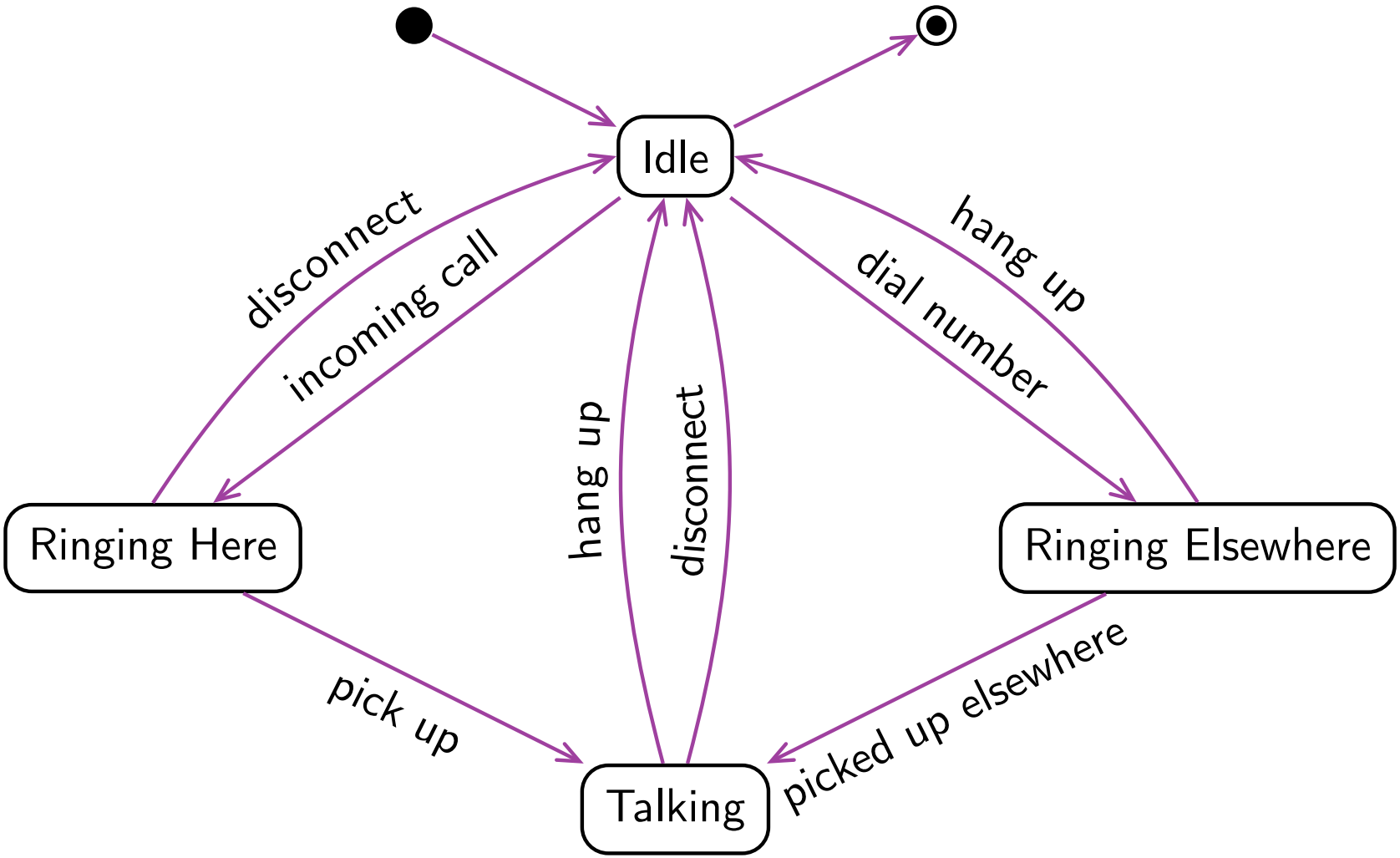
(Using informal transition labels.)

Phone Example



(Using informal transition labels.)

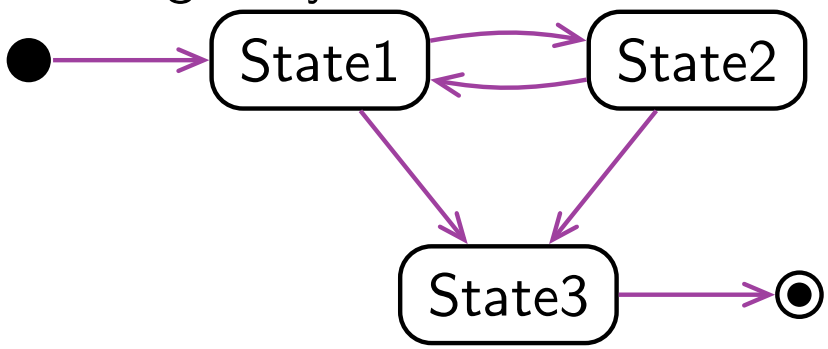
Phone Example



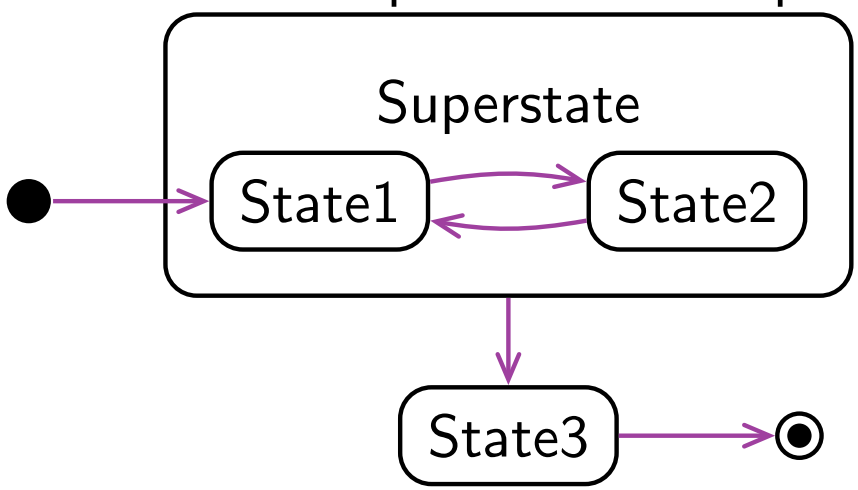
(Using informal transition labels.)

Superstates and Substates

- Sometimes, you find that several states have similarities.
  - e.g. they have some of the same outgoing transitions.

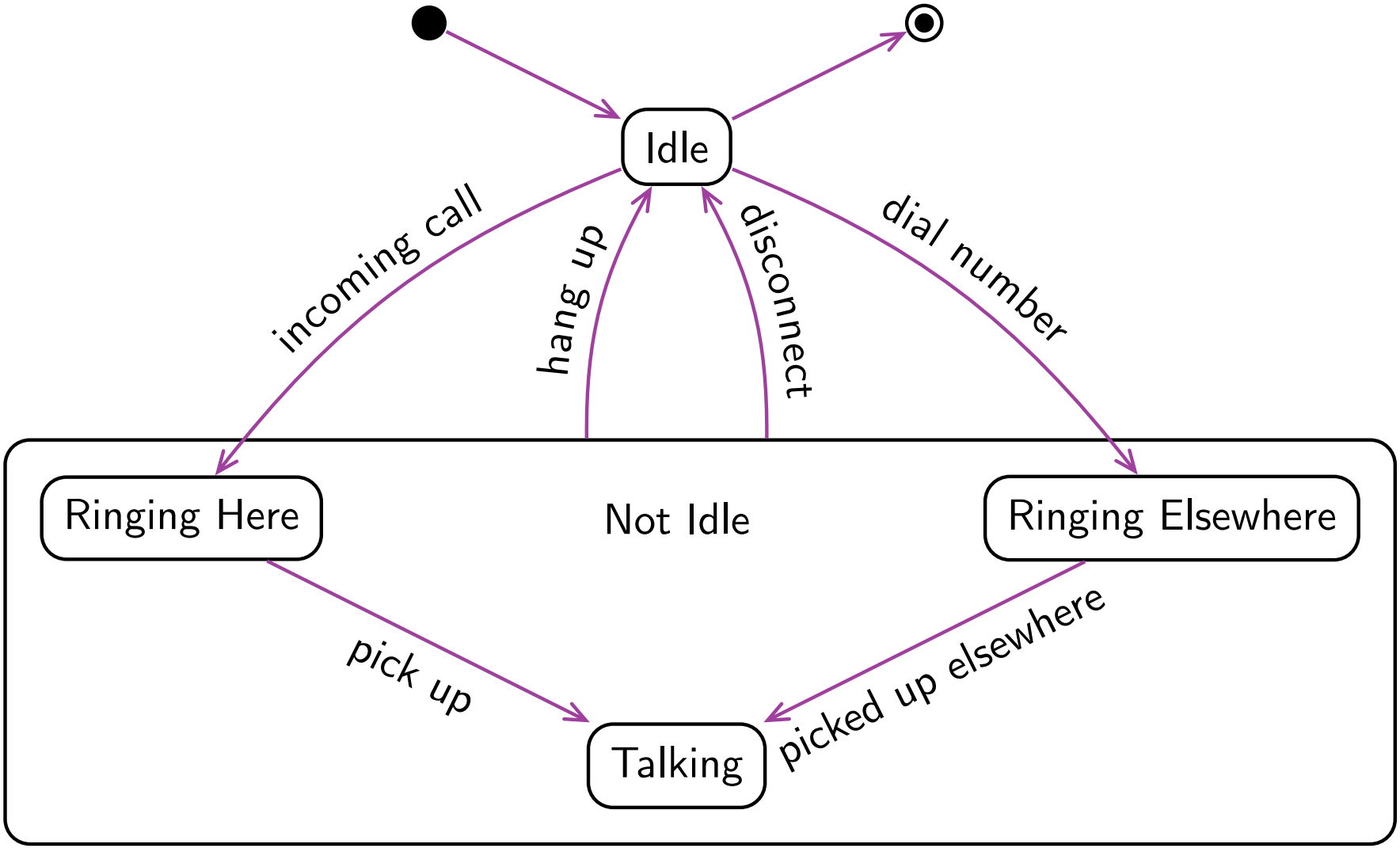


- We can use "superstates" to represent this:



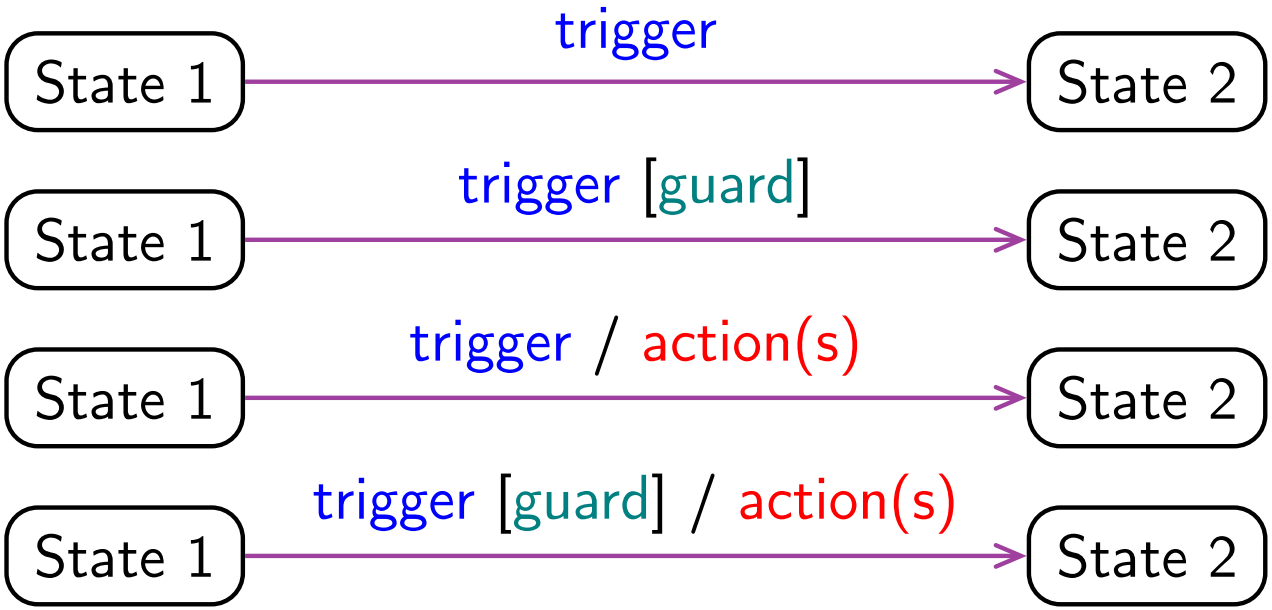
(Transition labels omitted here to avoid clutter.)

Superstates – Phone Example Again



UML State Transition Labels

State transitions have a particular structure in UML:



trigger: the event that **causes** the transition to happen.

guard: a boolean condition that, if false, **prevents** the transition.

action(s): events **caused by** the transition (an effect of it).

Triggers

- On previous slides, our triggers were informal.
- In UML, a trigger has one of three types:

  - The transition is triggered by the condition becoming true.
  - (Unlike a guard condition, which can only *prevent* a transition.)

  - The transition occurs after a period of time elapses.
  - The elapsed time is counted from the last transition.

  - The transition is triggered by a method call.

Method Triggers – Implementation

- What does this state transition mean?
- Here's the equivalent in pseudo-Python:

```
class MyClass:
    def __init__(self, ...):
        initialise object to state 1

    def someMethod(self):
        if this object is in state 1:
            change to state 2
```

- We wait for someMethod to be called, then check what to do.

States – Implementation

- Ahh, so what do "state 1" and "state 2" mean?
- Here's *one possible* (simplistic) implementation:

```
class MyClass:
    def __init__(self, ...):
        self.state = 1

    def someMethod(self):
        if self.state == 1:
            self.state = 2
```

- The state is determined by a "state variable".
- Not always so simplistic.

Guards and Actions – Implementation

- How to deal with guard conditions and actions?

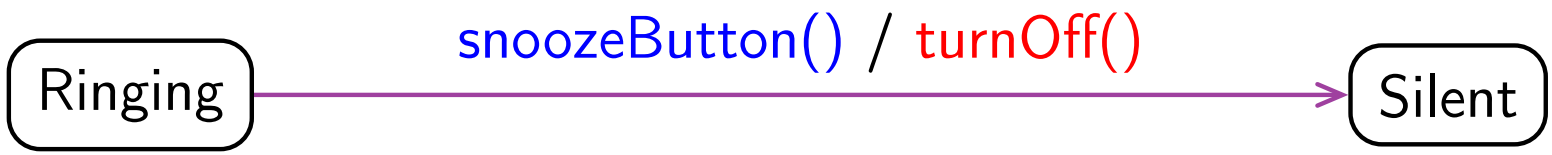
```
class MyClass:
    def __init__(self, ...):
        self.state = 1
        self.ferrets = ... # Some initial value

    def check(self):
        if self.state == 1 and self.ferrets > 25:
            self.state = 2
            self.runAway() # Perform action
```

- We presume that `ferrets` and `runAway` are both in the same class, unless we say otherwise.

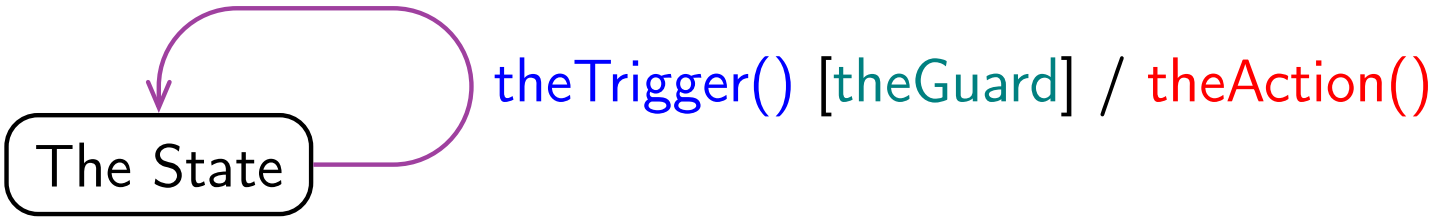
Guards! Guards!

- Guard conditions are not always needed.
- Simply *being in a state* is itself a condition that the system must check.
- You *don't* need to say this:

  - [isRinging()] is redundant. We *already know* that it's true.
  - All we need is this:



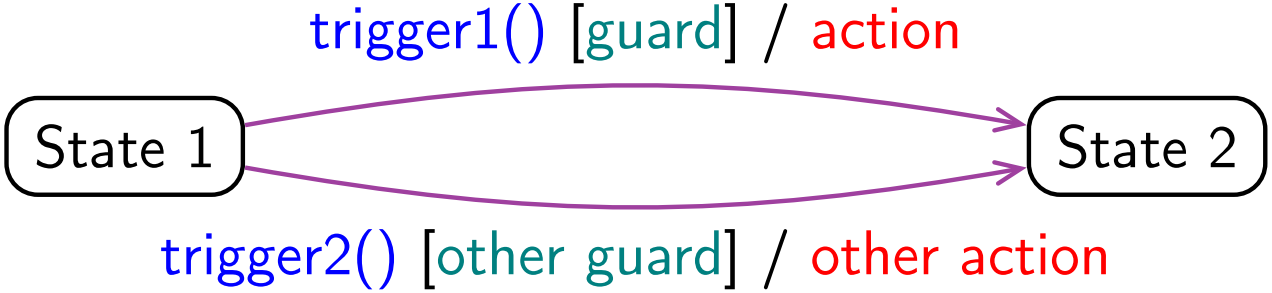
Self-Transitions



- It may seem useless, but a state can transition to itself.
- This has two effects:
  1. You reset the timer for any "after" triggers.
  2. You can describe actions performed while in a state.
- In pseudo-Python:

```
def theTrigger(self):
    if (we are in The State) and (theGuard is true):
        self.theAction()
        # But don't change state
```

Multiple Transitions

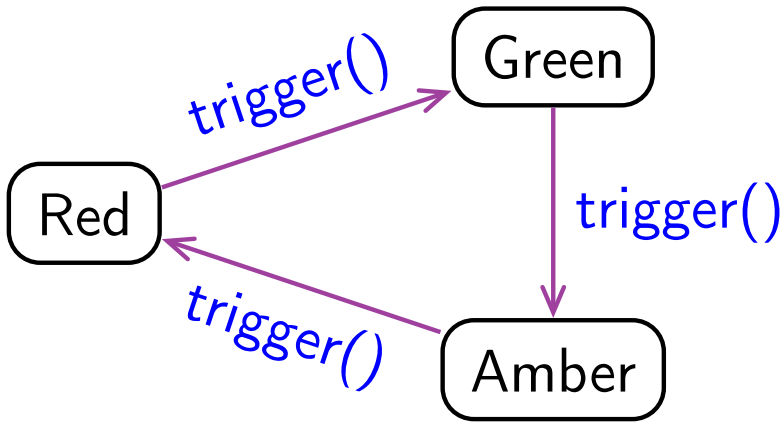


- Multiple transitions between the same start and end states may also be useful.
- You can describe different ways of achieving a similar effect.

```
class MyClass:
    ...
    def trigger1(self):
        if (in state 1) and (guard condition): ...

    def trigger2(self):
        if (in state 1) and (other guard cond.): ...
```

Same Trigger, Different States

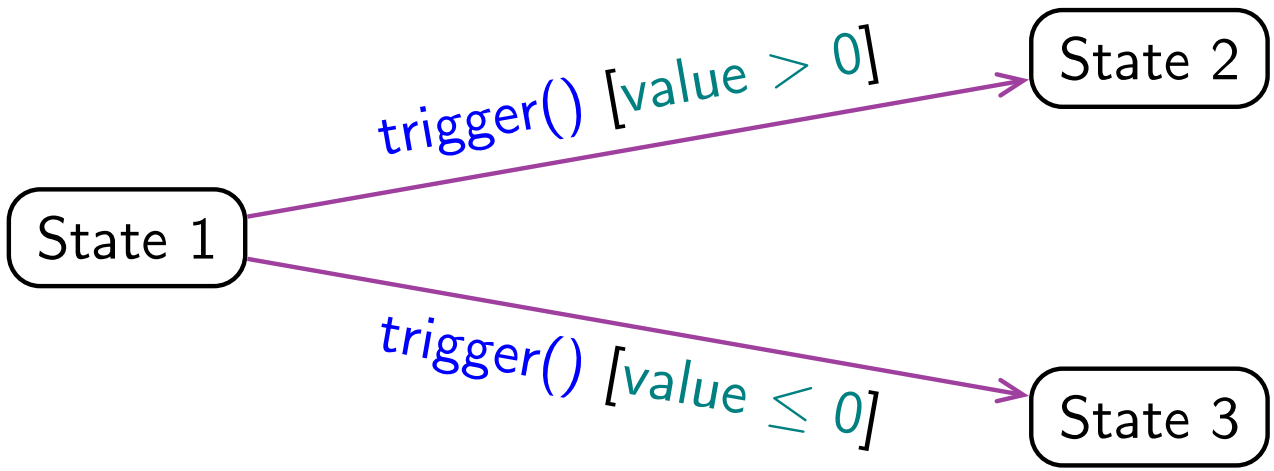
- The same trigger can be reused in different states.




```
def trigger(self):
    if self.state == "Red":     self.state = "Green"
    elif self.state == "Green": self.state = "Amber"
    elif self.state == "Amber": self.state = "Red"
```

Same Trigger, Same State, Different Guards

- Several transitions from the same state with the same trigger:
  - This is a logical contradiction...
  - *Unless* they have completely separate guard conditions.



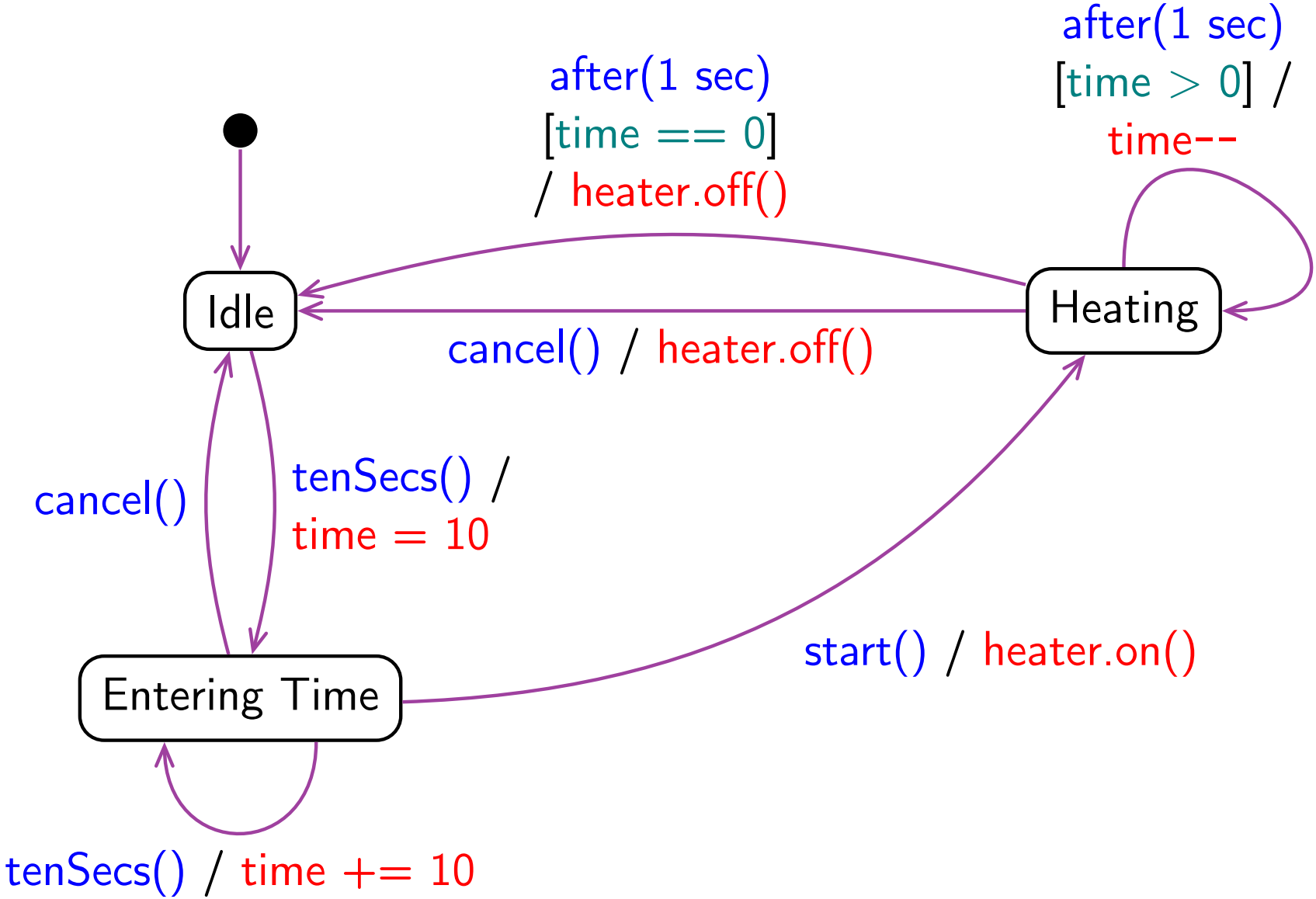
- Without "value > 0" and "value ≤ 0", which state would you go to?
  - Only one transition can be triggered at a time.
  - You can only be in one state at a time.

Microwave Example

- Say we're desiging a simple microwave.
- It will have three buttons: 10 seconds, start and cancel.
- Our Controller class might look like this:

```
public class MicrowaveController
{
    private int time = 0;
    public MicrowaveController() {}
    public void tenSecs() {...}
    public void start() {...}
    public void cancel() {...}
}
```

Microwave Example – UML



Microwave Example – Discussion

- The microwave begins idle.
  - The "cancel" button goes back here from both other states.
  - Heating is turned off if necessary.
- In the Entering Time state, we count up.
  - We get to this state by pressing "10 seconds".
  - Each time we press "10 seconds", we add to the time.
  - Pressing "start" starts heating.
- In the Heating state, we count each second.
  - If time > 0, we subtract one, and keep heating.
  - If time $== 0$, we stop heating and go to Idle.
- Some other things to consider:
  - Do we need the "Entering Time" state? (Could we combine it with "Idle"?)
  - How do we *display* the time?
  - What if the door is opened?

Microwave Example – Code for tenSecs()

```
public void tenSecs()
{
    switch(state)
    {
        case IDLE: // Symbolic constant
            time = 10;
            state = ENTERING_TIME;
            break;

        case ENTERING_TIME:
            time += 10;
            break;
    }
} // Ignored in the HEATING state
```

Microwave Example – Code for cancel()

```
public void cancel()
{
    switch(state)
    {
        case ENTERING_TIME:
            state = IDLE;
            break;

        case HEATING:
            state = IDLE;
            heater.off(); // Action
            break;
    }
} // Ignored in the IDLE state
```

Microwave Example – Timer code, part 1

```
public class MicrowaveController extends TimerTask {
    ...
    @Override
    public void run() // Called once per second
    {                 // (must be set up using Timer)
        if(state == HEATING} {   // Original state
            if(time > 0) {       // Guard condition 1
                time--;          // Action 1
            }
            else { // time == 0  // Guard condition 2
                heater.off();    // Action 2
                state = IDLE;    // New state
                timer.cancel();
            } // Stops the run() method being called.
        }
    }
}
```

Microwave Example – Timer code, part 2

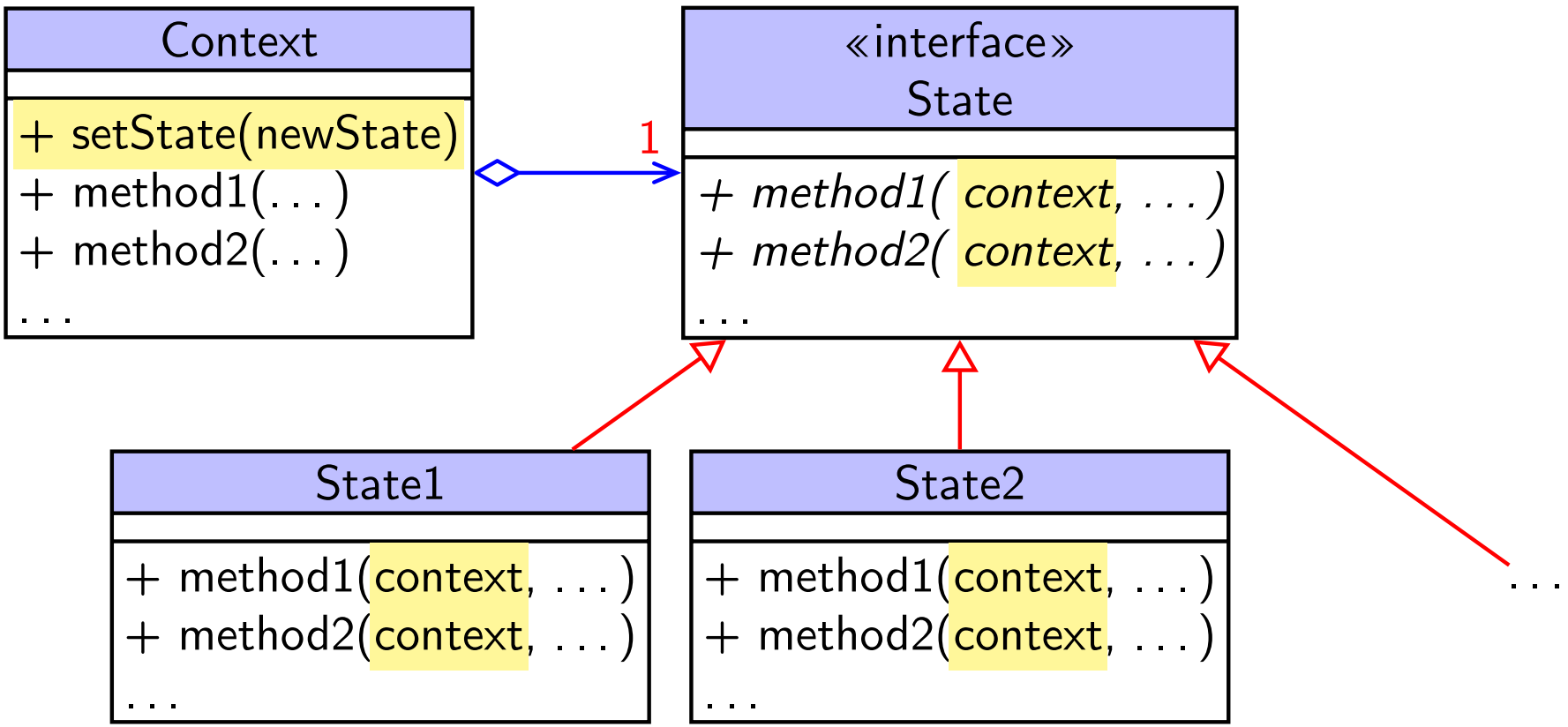
```
public class MicrowaveController extends TimerTask {
    private Timer timer = new Timer();
    ...
    public void start()
    {
        if(state == ENTERING_TIME) // Original state
        {
            heater.on();              // Action
            state = HEATING;          // New state
            timer.scheduleAtFixedRate(self, 0, 1000);
        } // Sets up the run() method to be called
    }     // once per second.
}
```

- Timer and TimerTask are standard Java API classes.
- *Not* critical that you know how to use them – this is just to illustrate how you *could* implement "after(1 second)".

The State Pattern

- For complex states and state transitions.
- Breaks up an object into its component *states*.
- A main/context object will own a state object, representing one possible state.
  - The rest of the program *only* sees the context object, not the state object.
- When a state transition occurs:
  - A new state object will be created.
  - The old state object will be destroyed.
- The main object delegates all its functionality to the current state object.
  - The main object's methods will simply call a corresponding state object method.

State Pattern – Generic UML



- The context object's method1 just calls State's method1, etc.
- Each subclass implements method1.
- These implementations have access *back* to the context.

State Pattern – What the UML Doesn't Say

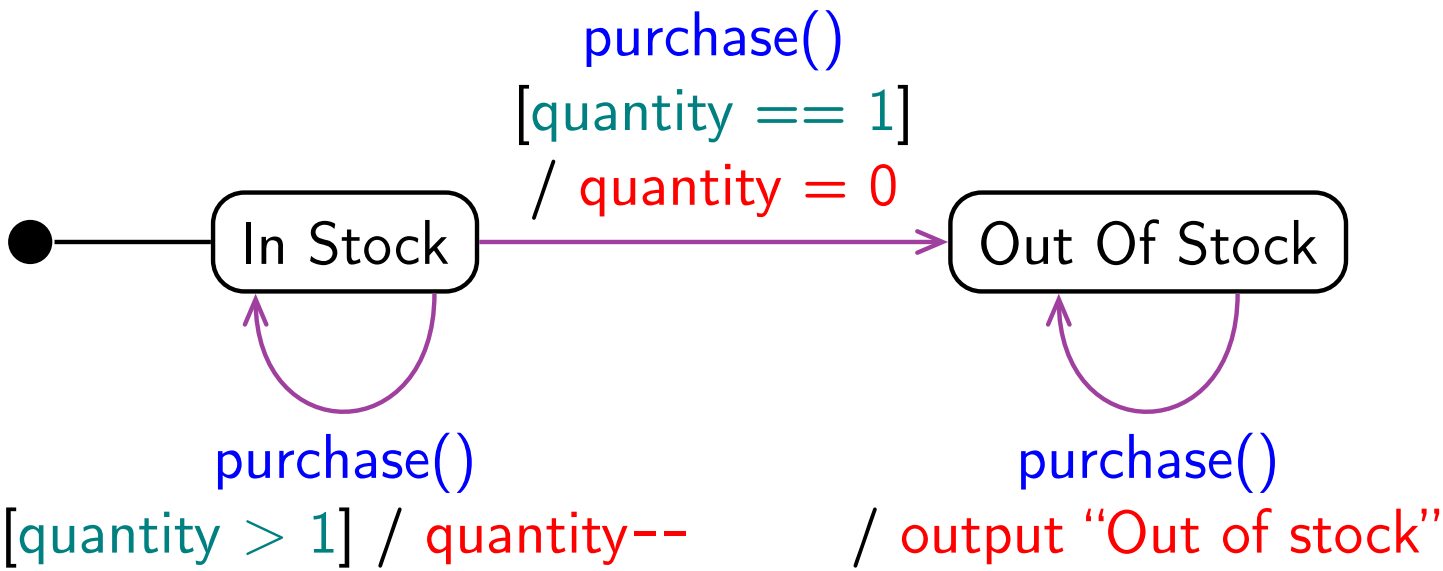
- When the Context.method1 is called, it does this:

```
currentState.method1(this, ...);
```

  - `currentState` is a field in the context object.
  - `method1` is a method in the state object.
  - "`...`" are any extra parameters.
- The state object's `method1` does all the hard work.
- The state object's `method1` has a reference to the context.
  - Thanks to its first parameter ("this", from the context's point of view).
- The state object can replace itself with another state object, when required.

```
context.setState(new State2());
```

Example – Products For Sale (1)



- A fairly simple state diagram.
- We'll use it to illustrate the State Pattern.

Example – Products For Sale (2)

```
public class Product {
    private String name = "Sprockets";
    private int quantity = 5;
    private ProductState state = new InStock();
    public Product() {}

    public void setState(ProductState newState) {
        state = newState;
    }

    public void purchase() {    // State-dependent
        state.purchase(this);   // behaviour
    }
    ... // Plus getQuantity and setQuantity
}
```

Example – Products For Sale (3)

```
public interface ProductState
{
    void purchase(Product context);
}
```

```
public class OutOfStock implements ProductState
{
    public void purchase(Product context)
    {
        System.out.println("Out of stock");
    }
}
```

Example – Products For Sale (4)

```
public interface ProductState
{
    void purchase(Product context);
} // As before
```

```
public class InStock implements ProductState
{
    public void purchase(Product context)
    {
        int quantity = context.getQuantity() - 1;
        context.setQuantity(quantity);
        if(quantity == 0)
            context.setState(new OutOfStock());
    }
}
```

Example – Products For Sale (5)

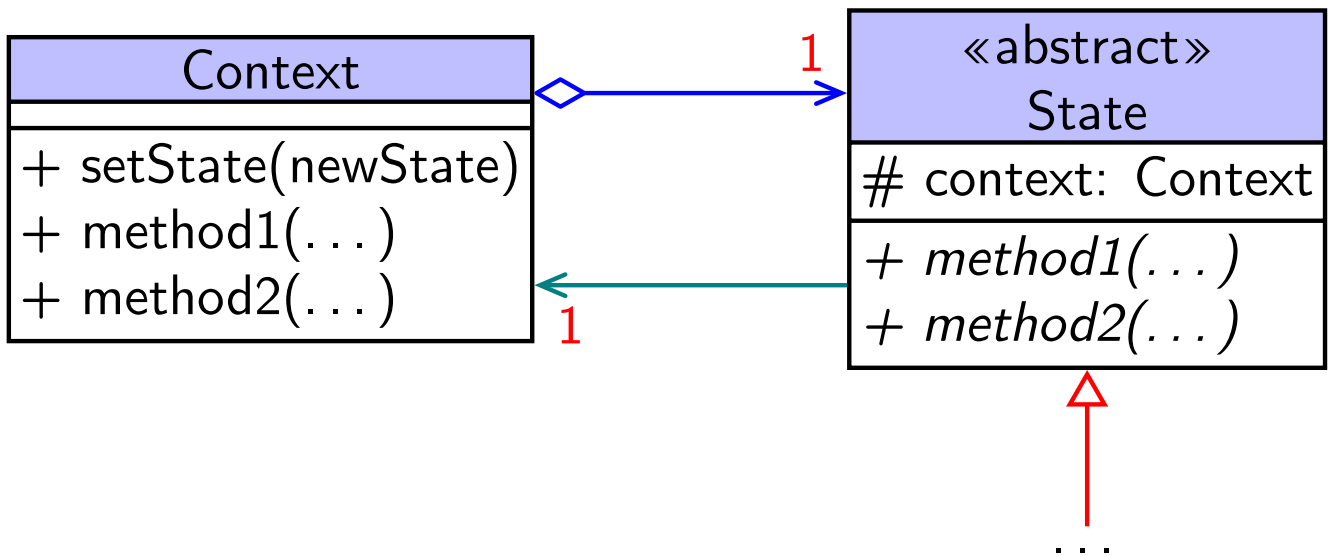
- Our `Product` object begins with `quantity == 5`, and the `InStock` state.
- `Product.purchase` (when called) calls `InStock.purchase`.
- `InStock.purchase` reduces the quantity by one.
- When `quantity == 0`, `InStock.purchase` triggers a state transition to `OutOfStock`:

```
context.setState(new OutOfStock());
```

- From then on, `Product.purchase` will call `OutOfStock.purchase`.

Context and State – A Two-Way Relationship

- Every method in every state object takes a context parameter.
  - A reference to the context object, as mentioned.
- Or does it?
  - Consider the State interface.
  - What if it was an abstract class instead?
  - You could have a context field, set by its constructor.



- Here, each state object method can call `getContext` instead.