More on Design

- States
- Delegation

First, a digression...

Design a wall following behavior for a robot

- **Problem Statement:**

  When awakened, the robot should move forward until it reaches a wall. Once it reaches a wall, the robot should follow that wall, keeping the wall to its right.

- **Sensors:**

  The robot has sensors that indicate whether a wall is in front of or to the right of the robot. (All other sensors are currently broken.)

```
boolean frontOK = _sensor.isClear(FRONT)
boolean rightOK = _sensor.isClear(RIGHT)
```

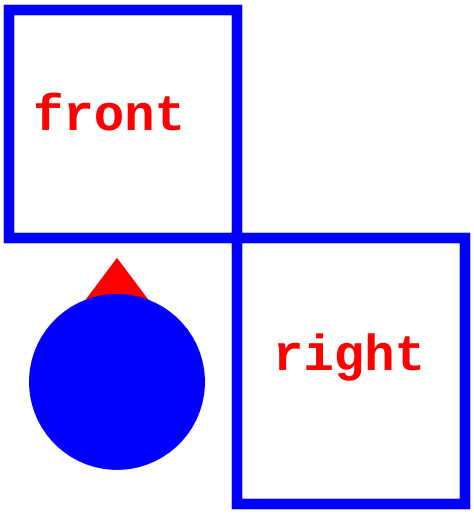
- **Actions:**

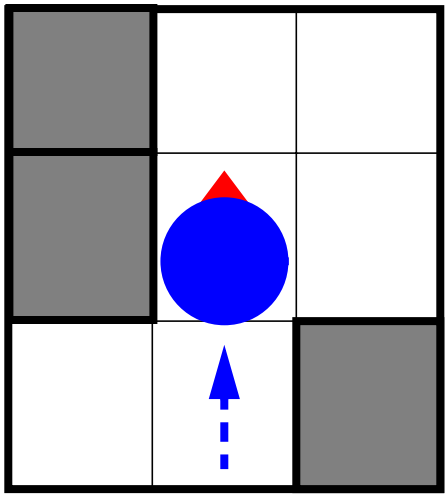
  The robot lives in a grid world. It can move in the following ways: turn in place (90 degrees left or right), and go forward one grid spacing.

```
_robot.turnLeft()
_robot.turnRight()
_robot.goForward()
```

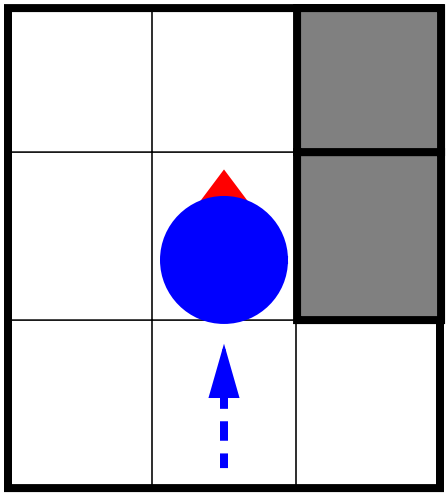
Example - Sensors

```
boolean frontOK = _sensor.isClear(FRONT)
boolean rightOK = _sensor.isClear(RIGHT)
```

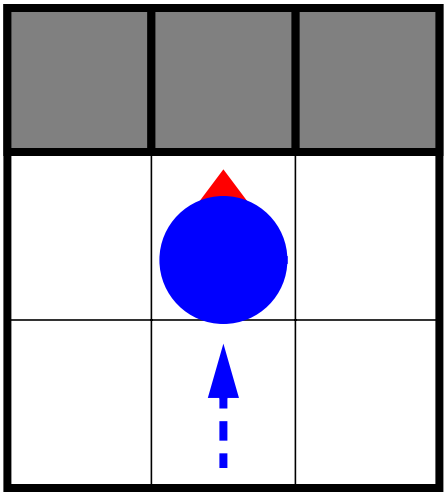




(frontOK & rightOK)

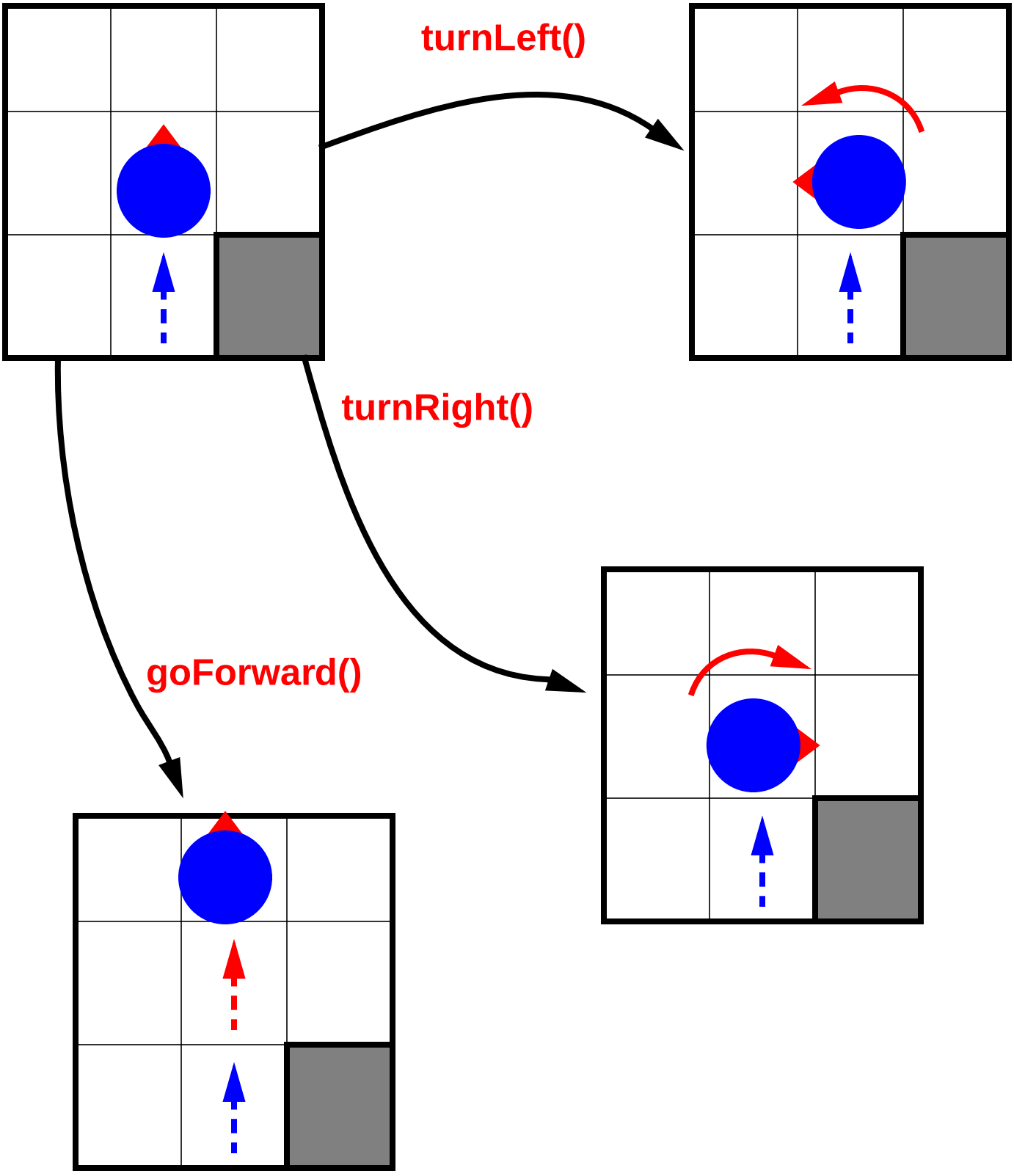


(frontOK)

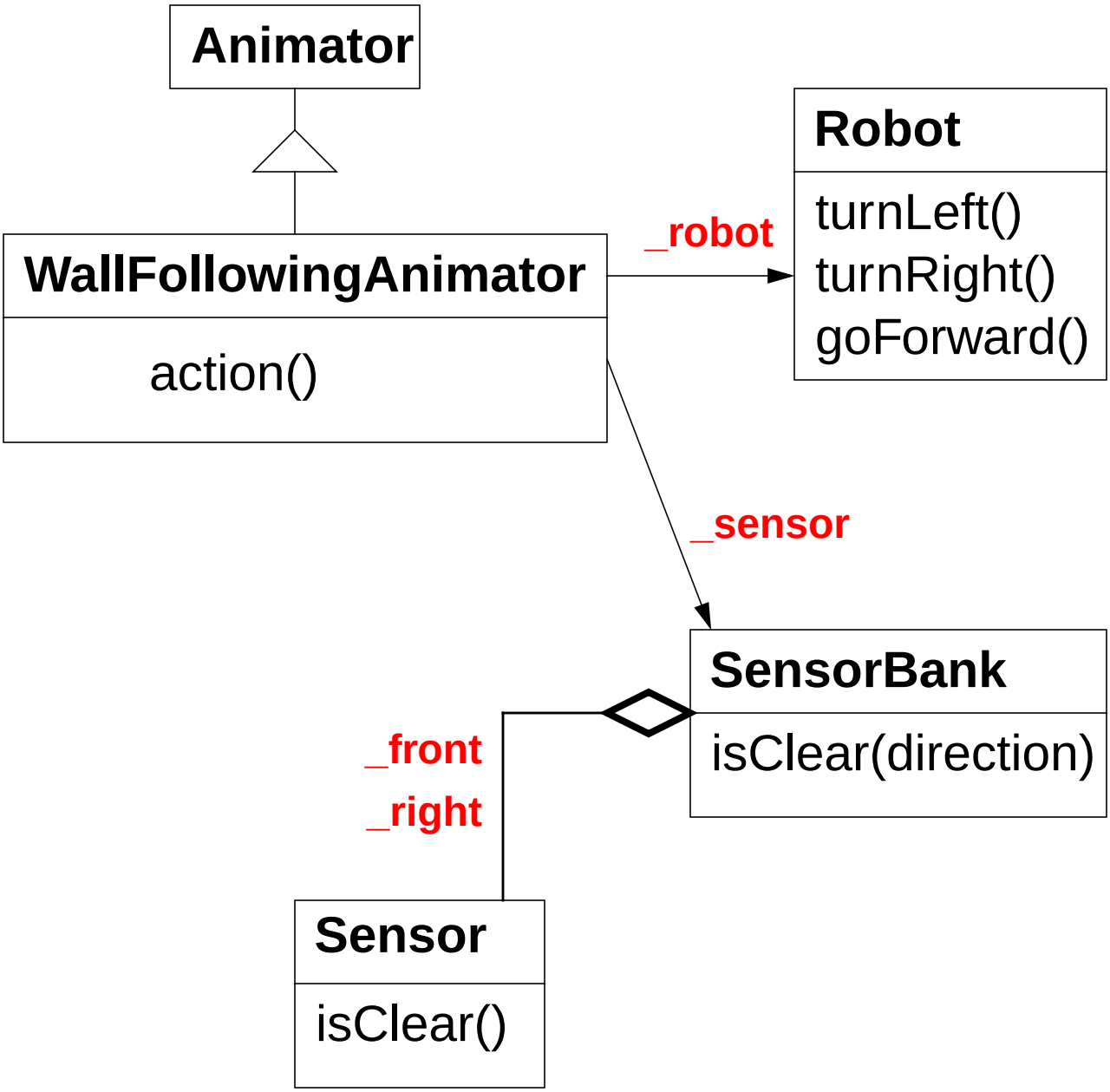


(rightOK)

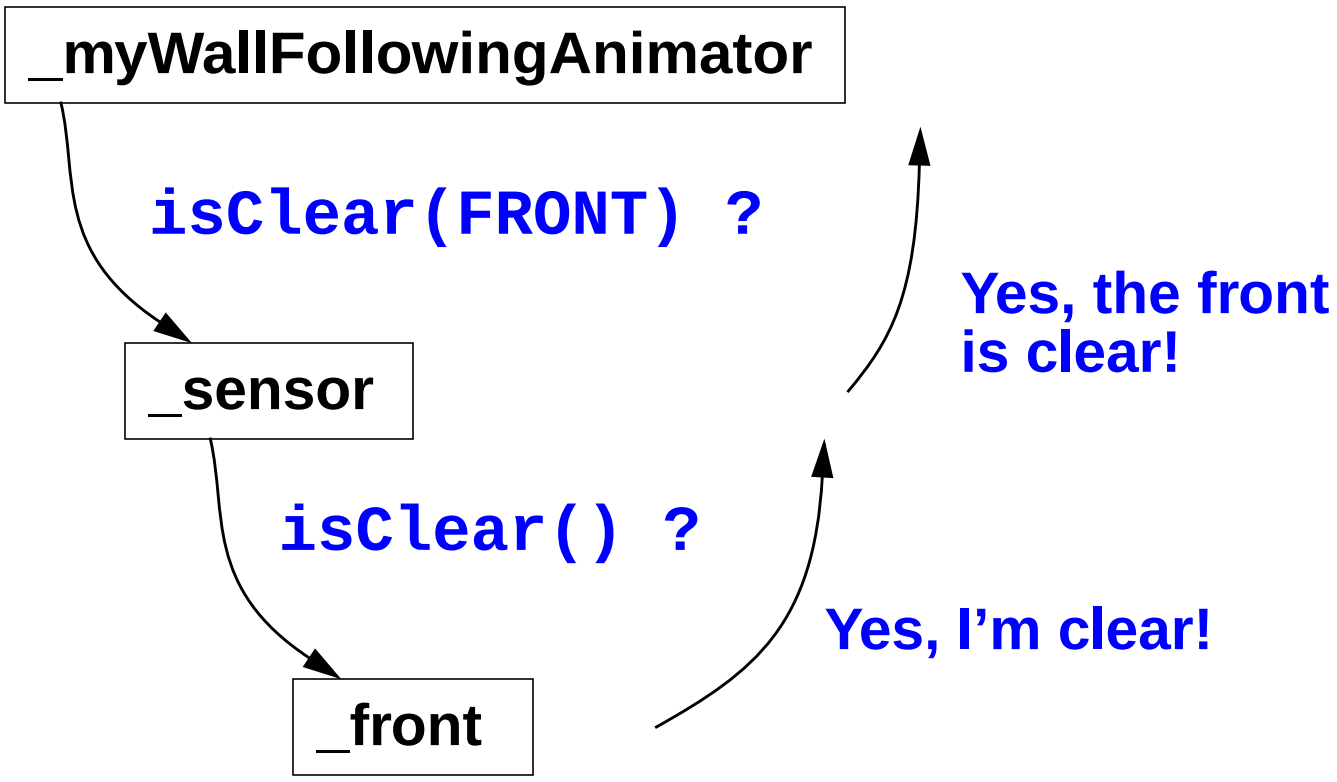
Example - Actions



Let's create a WallFollowingAnimator..



Delegation!



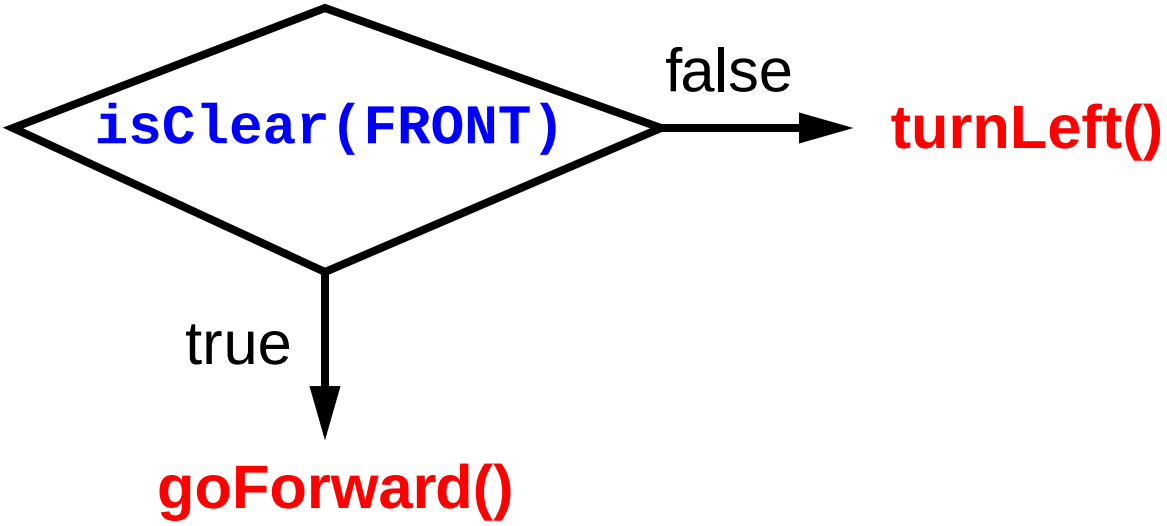
Why is this helpful?

- We can plug in any kind of sensor we have (sonar, infrared, camera)

- The only requirement is that it must be able to process its data and tell us whether there is anything directly in front of it (i.e. implement the isClear method).

What does “action()” do?

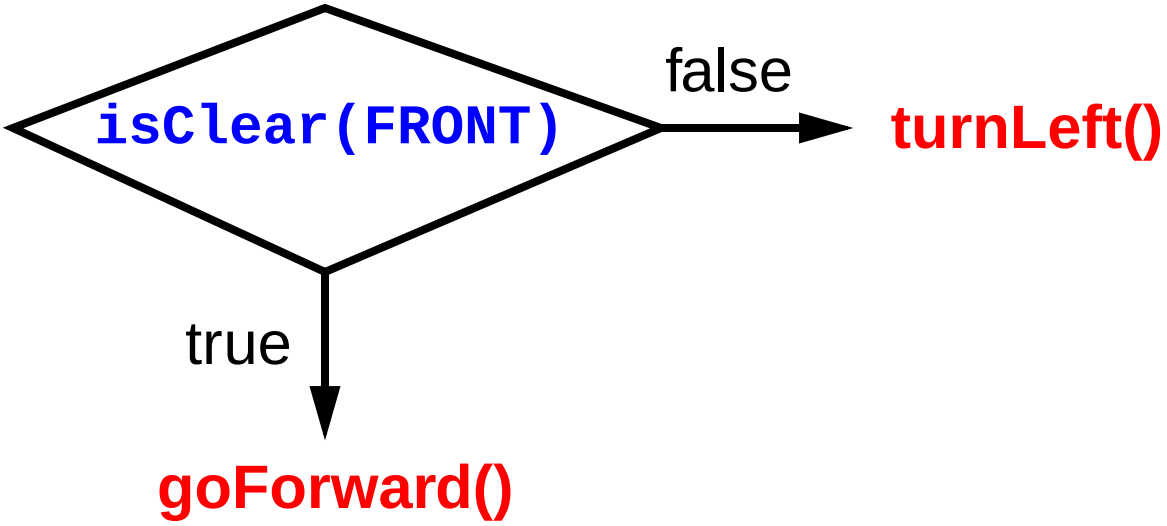
This design will not work



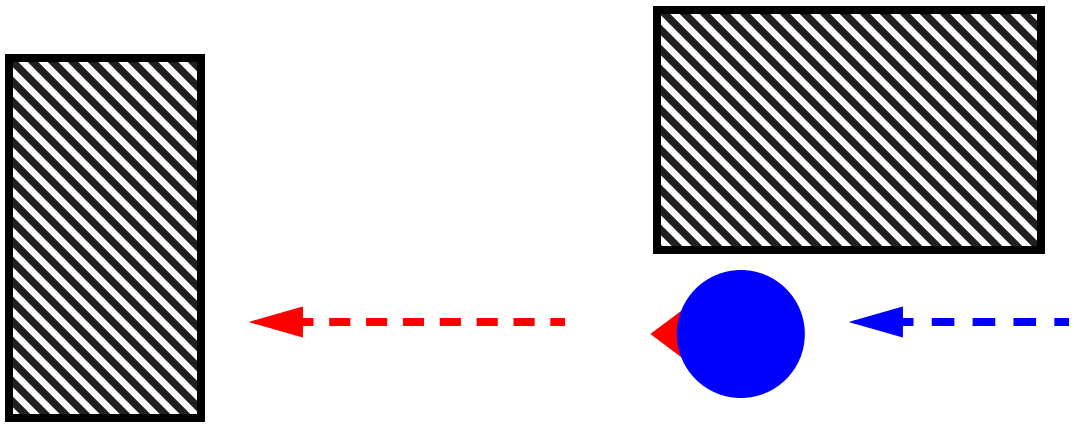
When does it fail?

What does “action()” do?

This design will not work



Problem: the robot will miss the right turn...

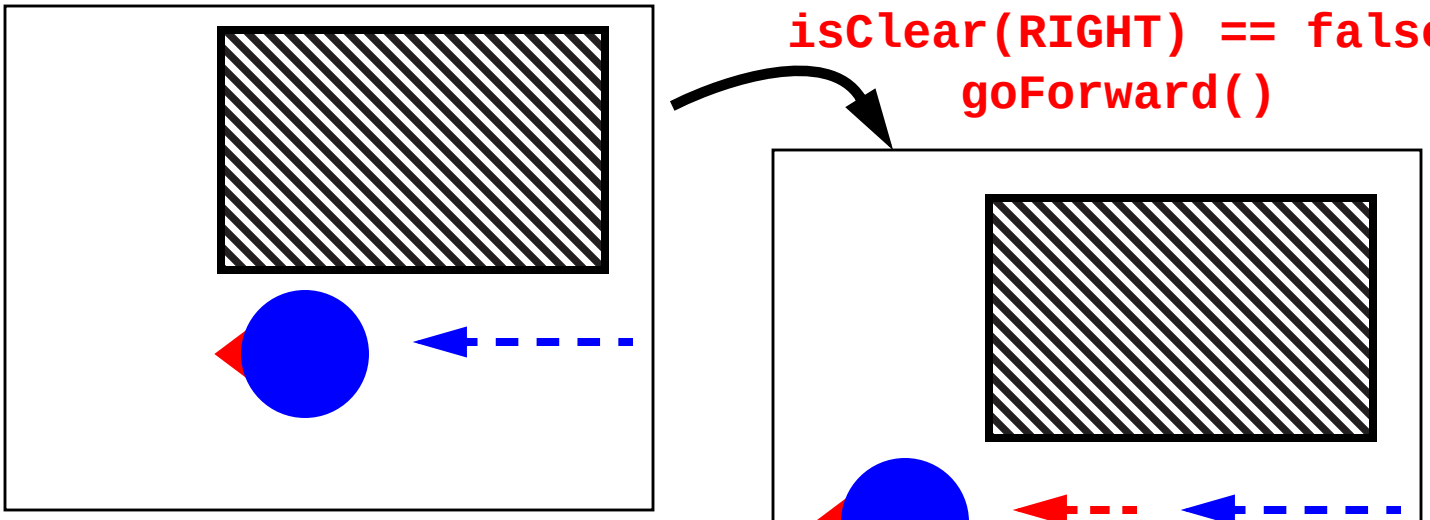


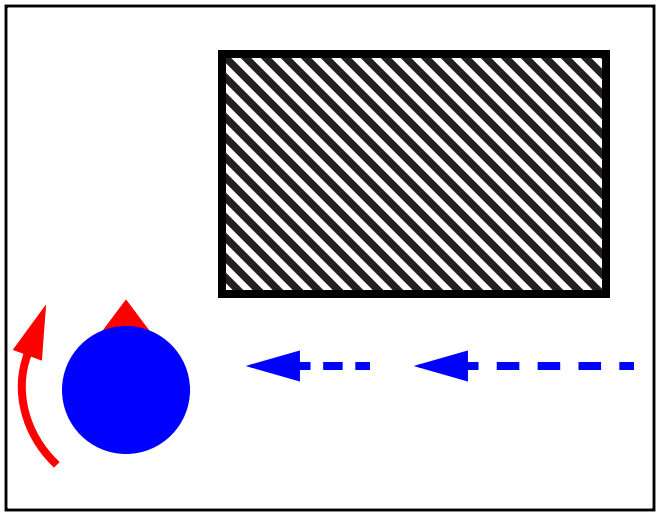
How can we fix this?

Solution

- Add a “following” state variable to indicate whether the robot was following the wall before the last action.

```
following == true
isClear(RIGHT) == false
goForward()
```



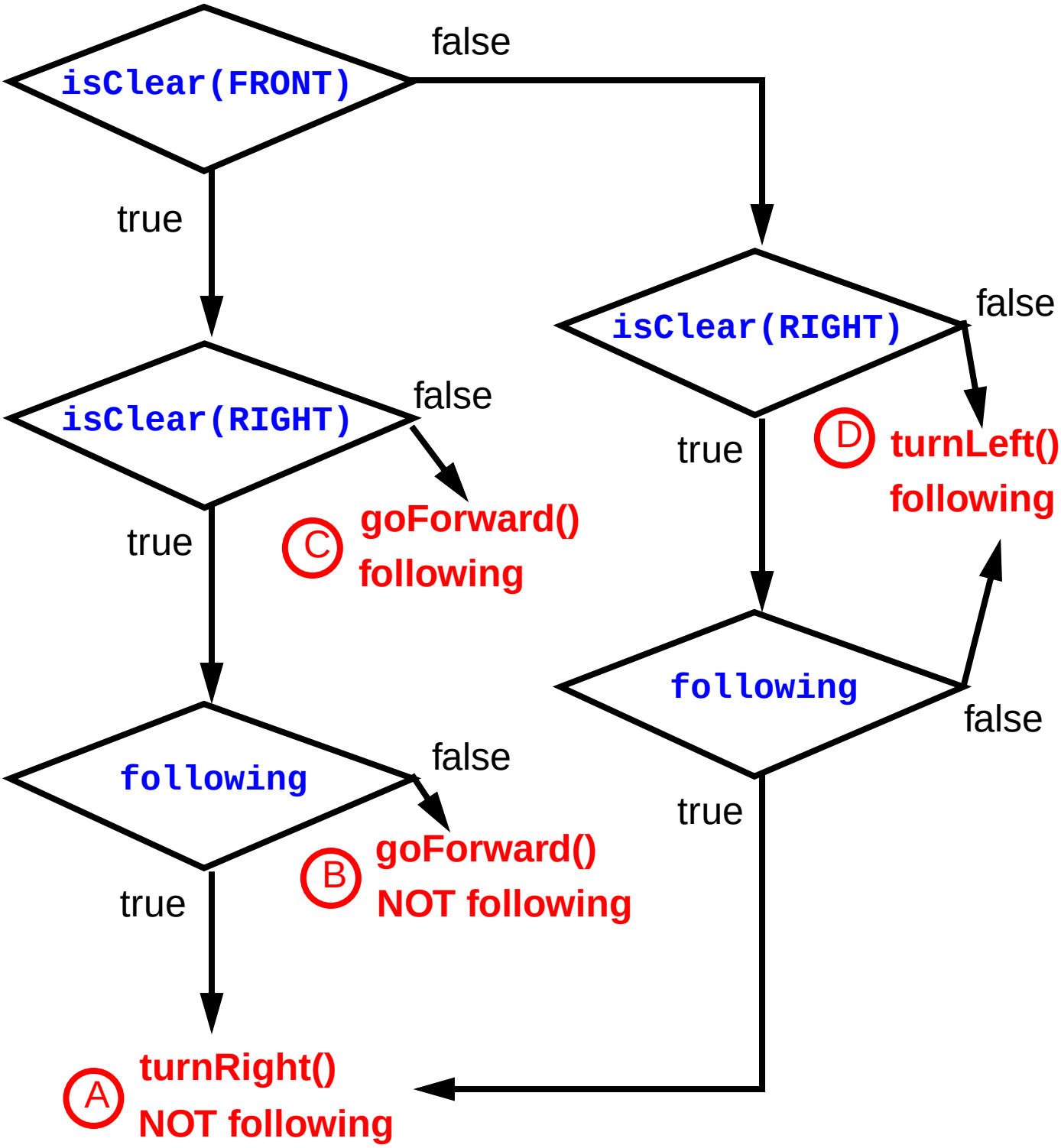


```
following == true
isClear(RIGHT) == true
turnRight()
following = false
```

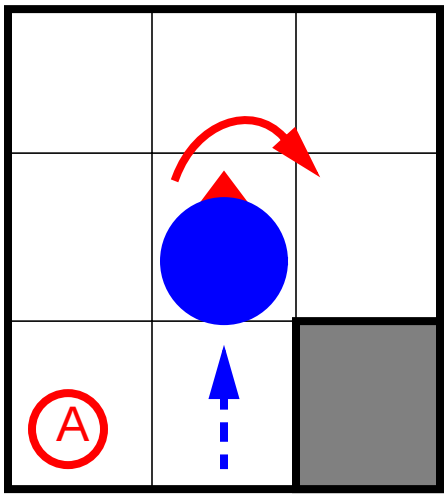
The wall should be over to the right. Turn right and begin to look for it again.

Here's the Flow Chart..

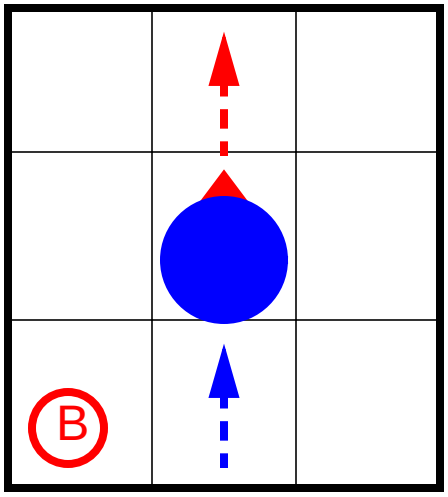
Four different situations can be distinguished



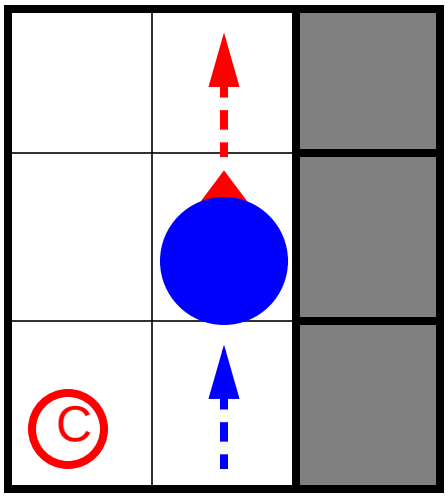
Examples



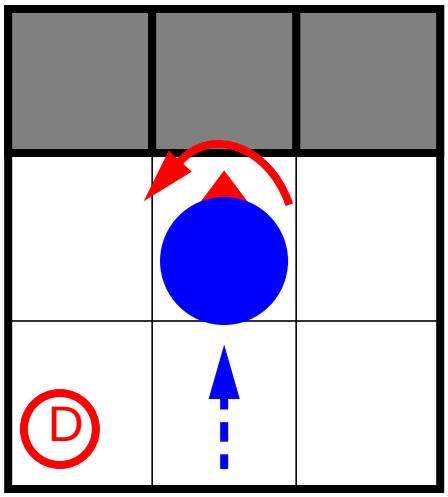
JUST_LOST_WALL



FIND_THE_WALL



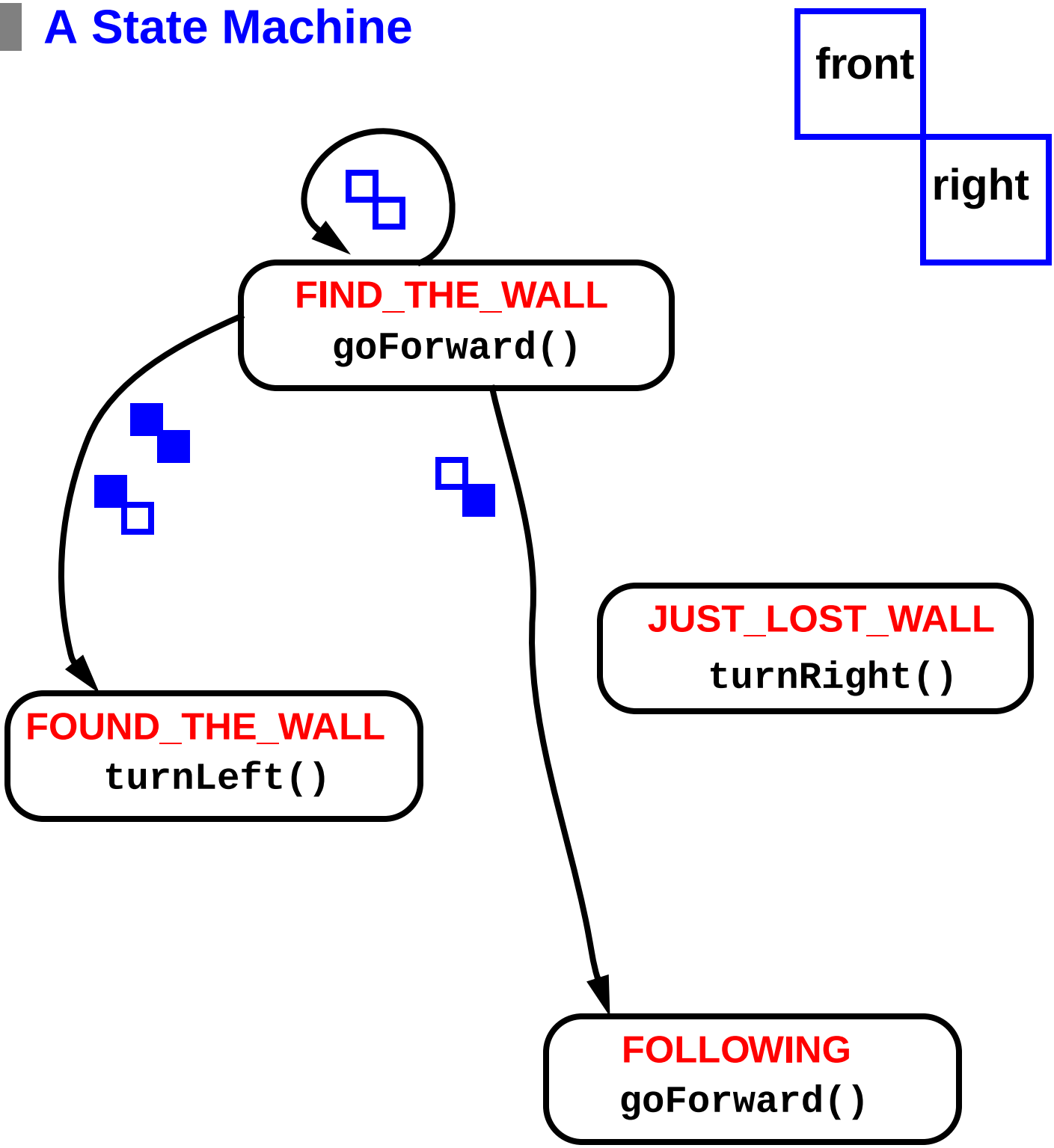
FOLLOWING



FOUND_THE_WALL

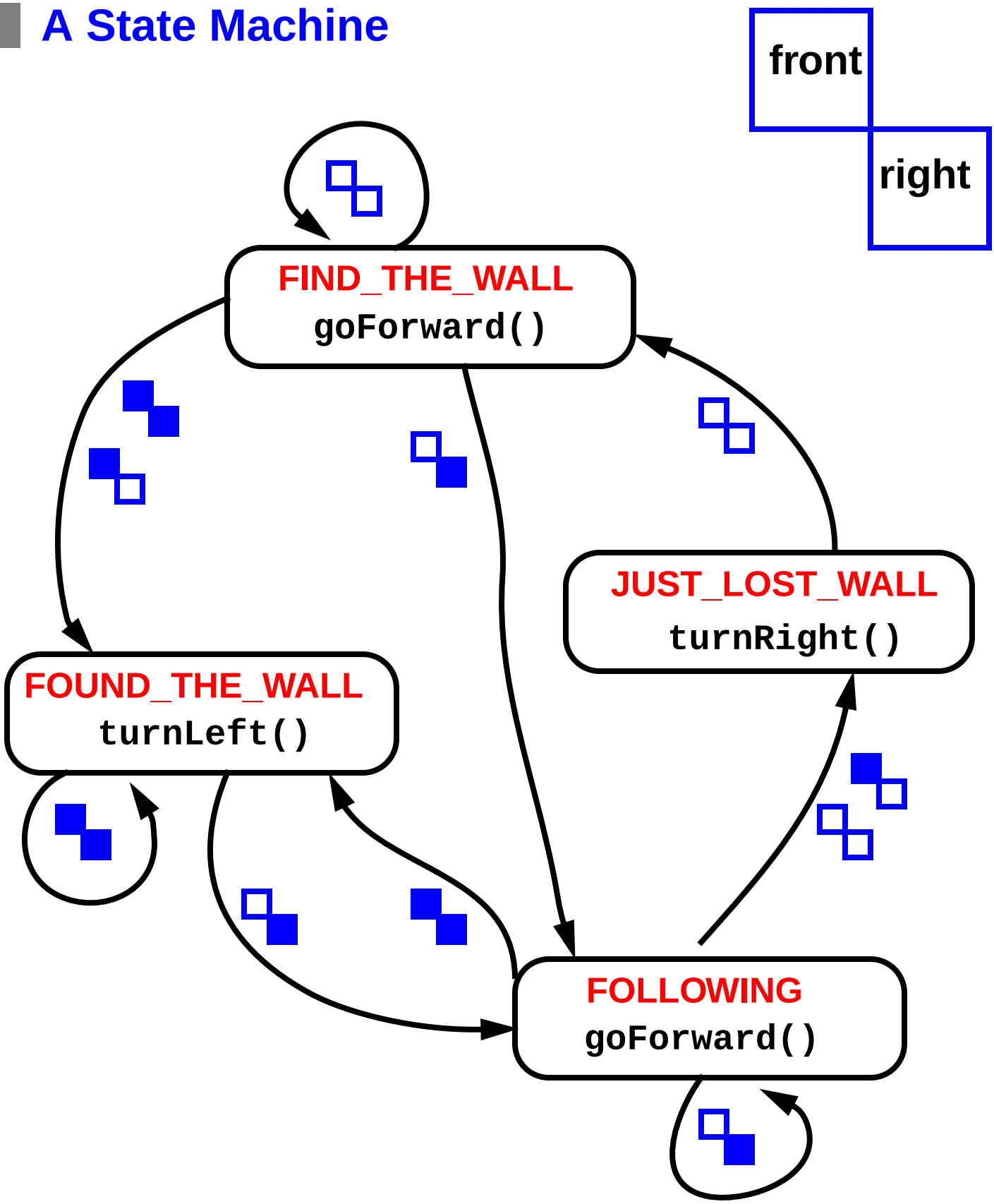
- We can give each of these situations names.
- We call each different situation a state.
- Here, each state is associated with a specific, simple action, but this is not always the case. The action can be more complex.

A State Machine



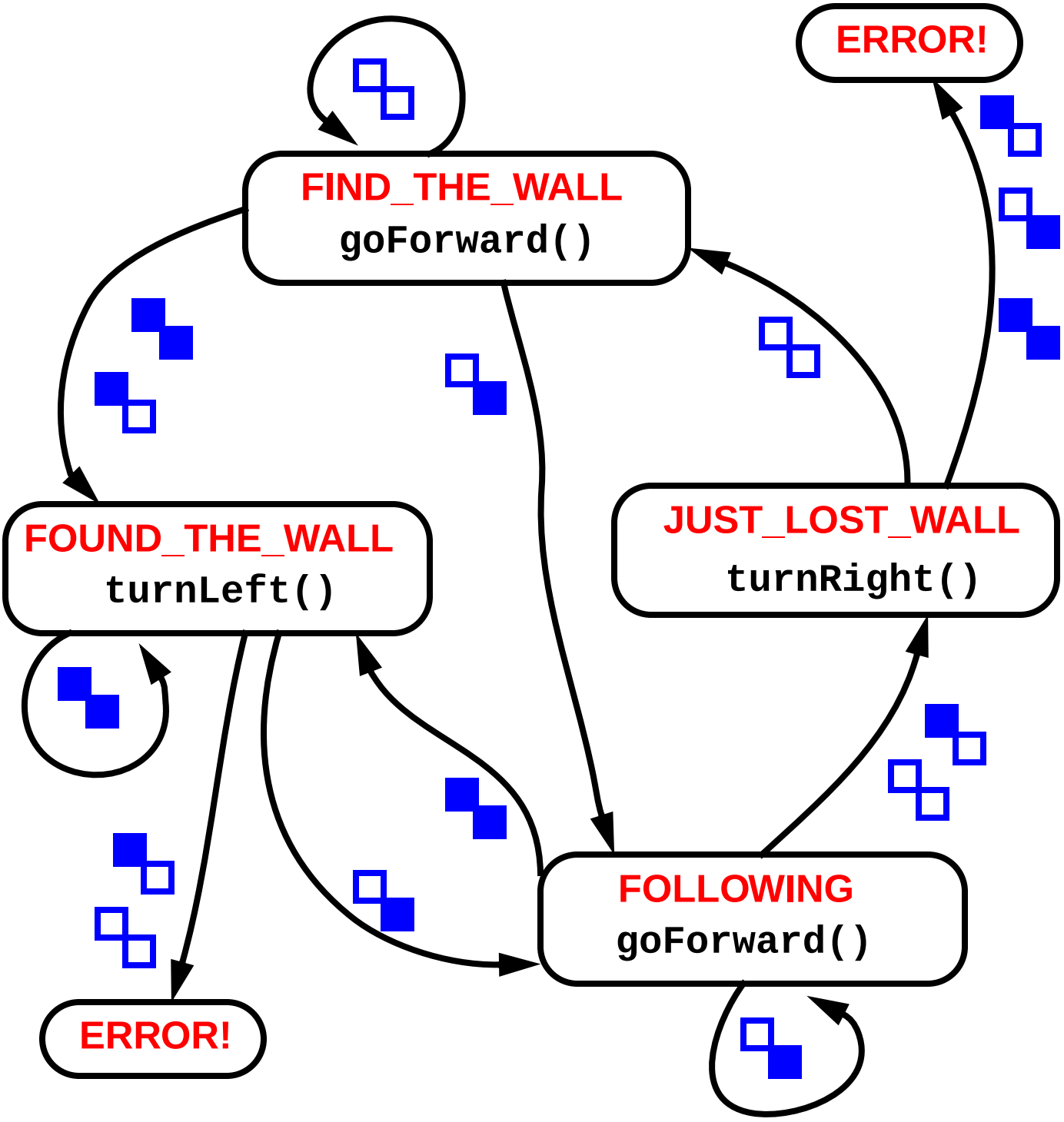
- Begin constructing a state machine by adding transitions from each state to the others.
- Here, only the transitions from the FIND_THE_WALL state have been added.

A State Machine



- Here is the complete state machine.

A State Machine



- State transitions are based on sensor values.
- Some sensor values are unexpected.
- Error states can be added for easier debugging and more reliable code.

Implementation

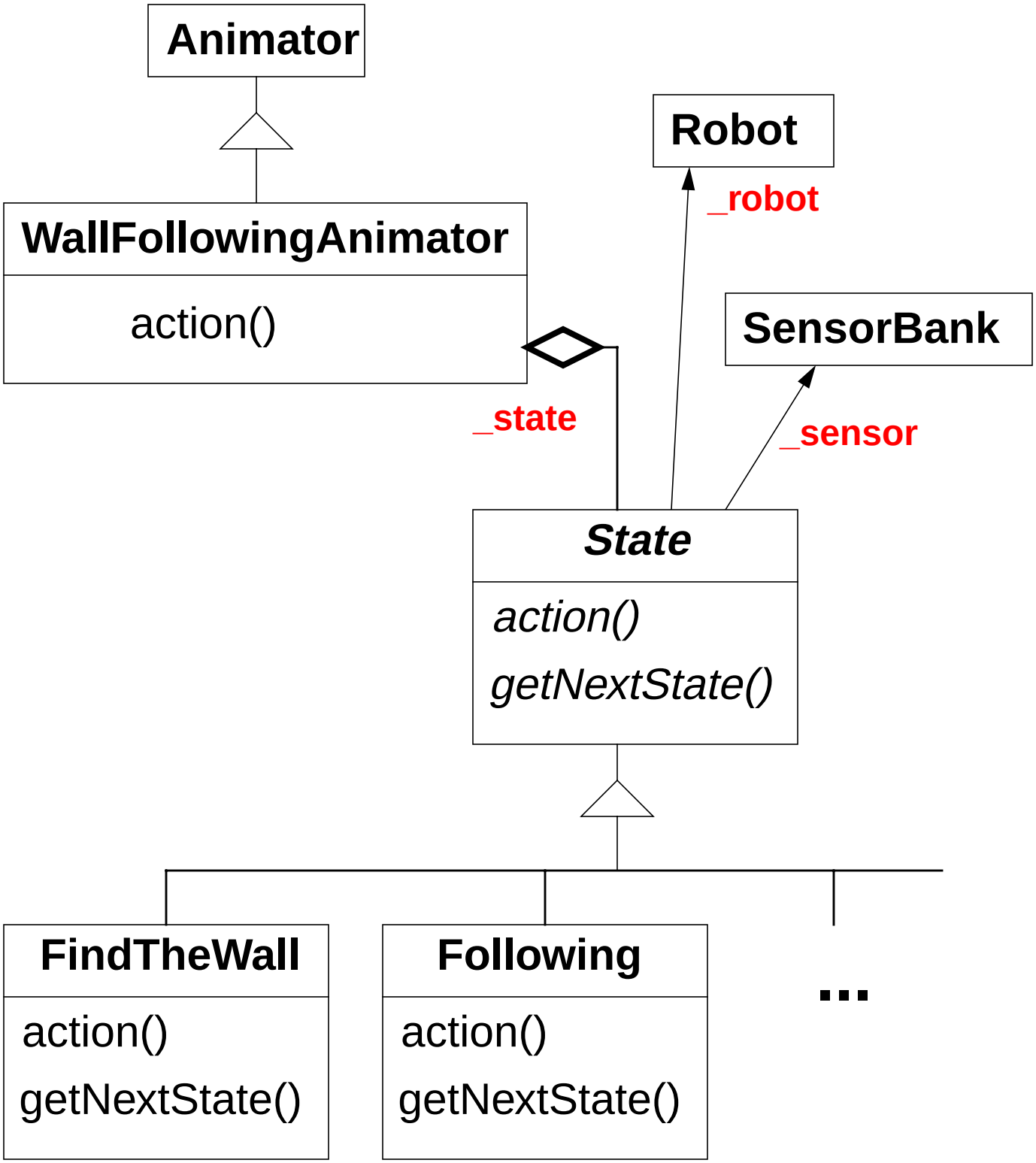
- WallFollowingAnimator can have a state variable, and select the proper action based on the value of the sensors and the value of that state variable.

- This could get messy!

- An alternative is to create a separate object for each state, with a much simpler action.

Implementation

- We can use the State Pattern!



Implementation - Let's see some code

WallFollowingAnimator

```
public void action() {
   _state = _state.getNextState();
   _state.action();
}
```

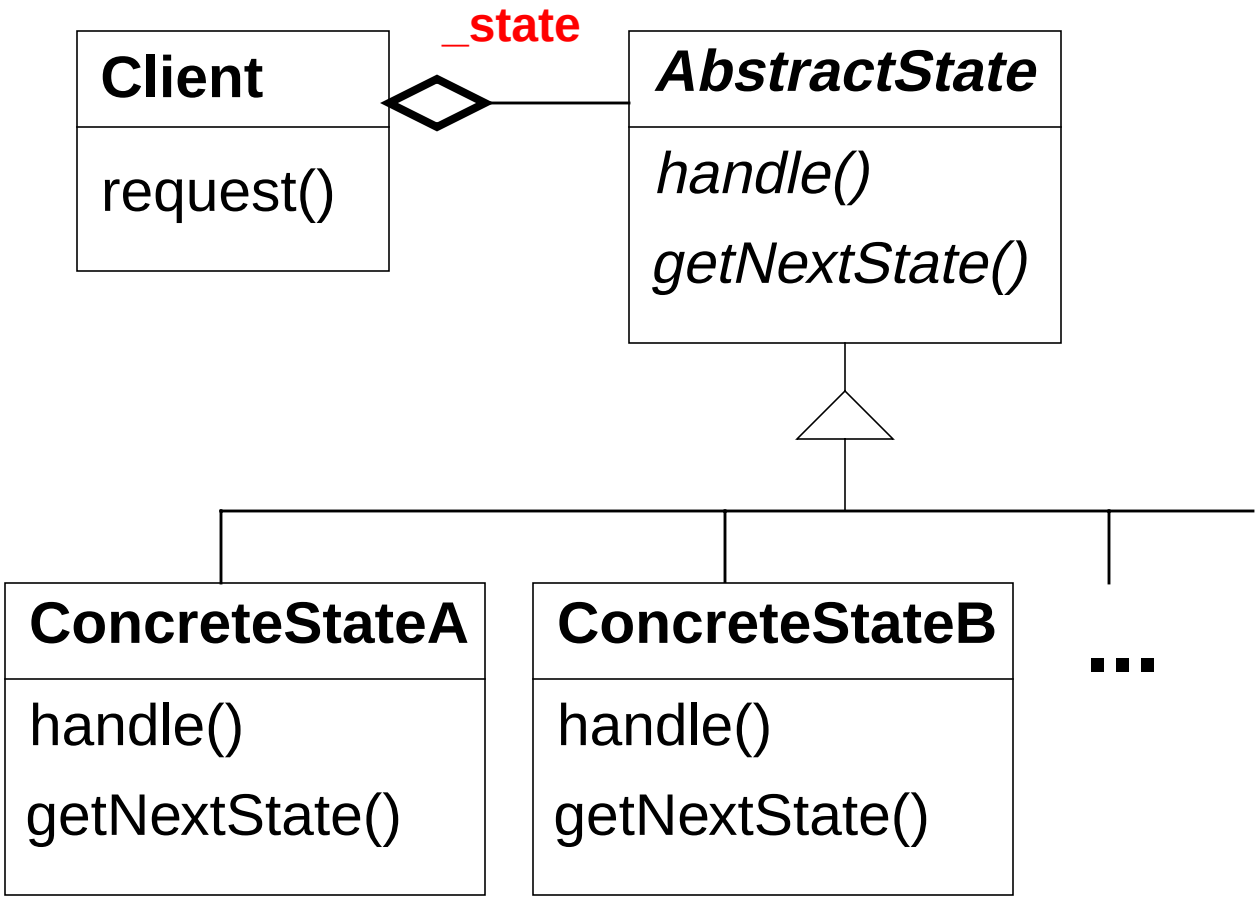
FindTheWall

```
public void action() {
   _robot.goForward();
 }


public State getNextState() {
   frontClear = _sensor.isClear(FRONT);
   rightClear = _sensor.isClear(RIGHT);
   if (frontClear & !rightClear) {
      return new Following();
   }

      ...
 }
```

The Generic Pattern



- The Client request() method asks _state to handle() the request.

- It also swaps in a new state by replacing _state with _state.getNextState()

Questions

- This implementation creates a new state object every time there is a transition from one state to the next.
  - What if you don’t want to constantly create new states?

- The WallFollowingAnimator calls _state.getNextState() in this example.
  - This means that WallFollowingAnimator has to reach in and control the actions of the state machine.
  - But WallFollowingAnimator really shouldn’t have to know that there is a state machine involved!
  - How can we encapsulate the state machine functionality entirely within the state machine?
  - The solution should reduce the possibility of errors and make the implementation of the state machine much more flexible.