Simulated State and Real State

CS 5010 Program Design Paradigms
"Bootcamp"

Lesson 9.1

Goals of this lesson

- Understand the difference between a mathematical model of state and real state

Let's think about traffic lights

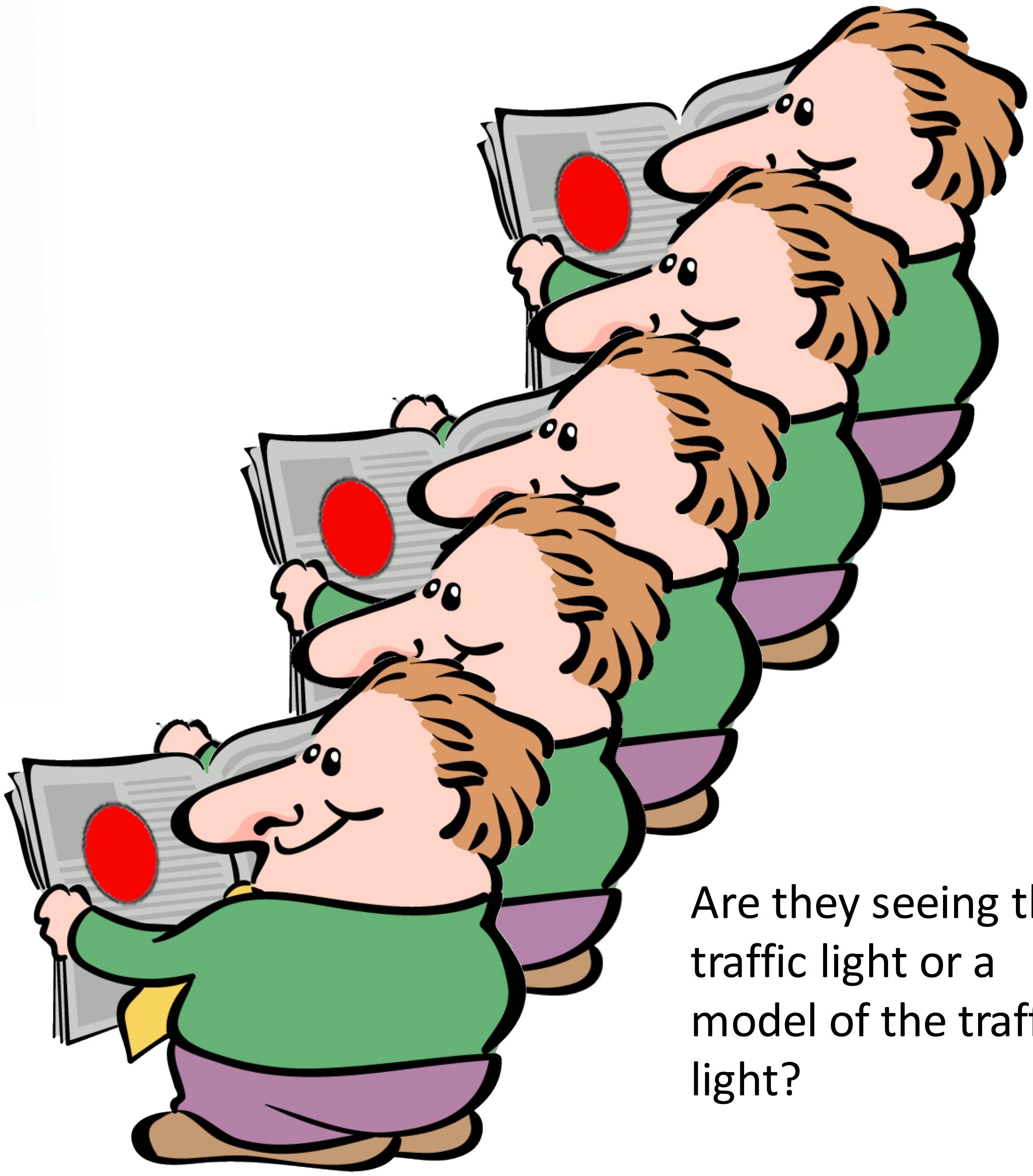
- nextstate : TLState → TLState
  - If the traffic light were in state s, what state should it be in next?
- This is a *specification* of how a traffic light should behave.
- This is simulated state: a mathematical function
- Real state: *make the traffic light change to the next state!*

Similarly for on-tick

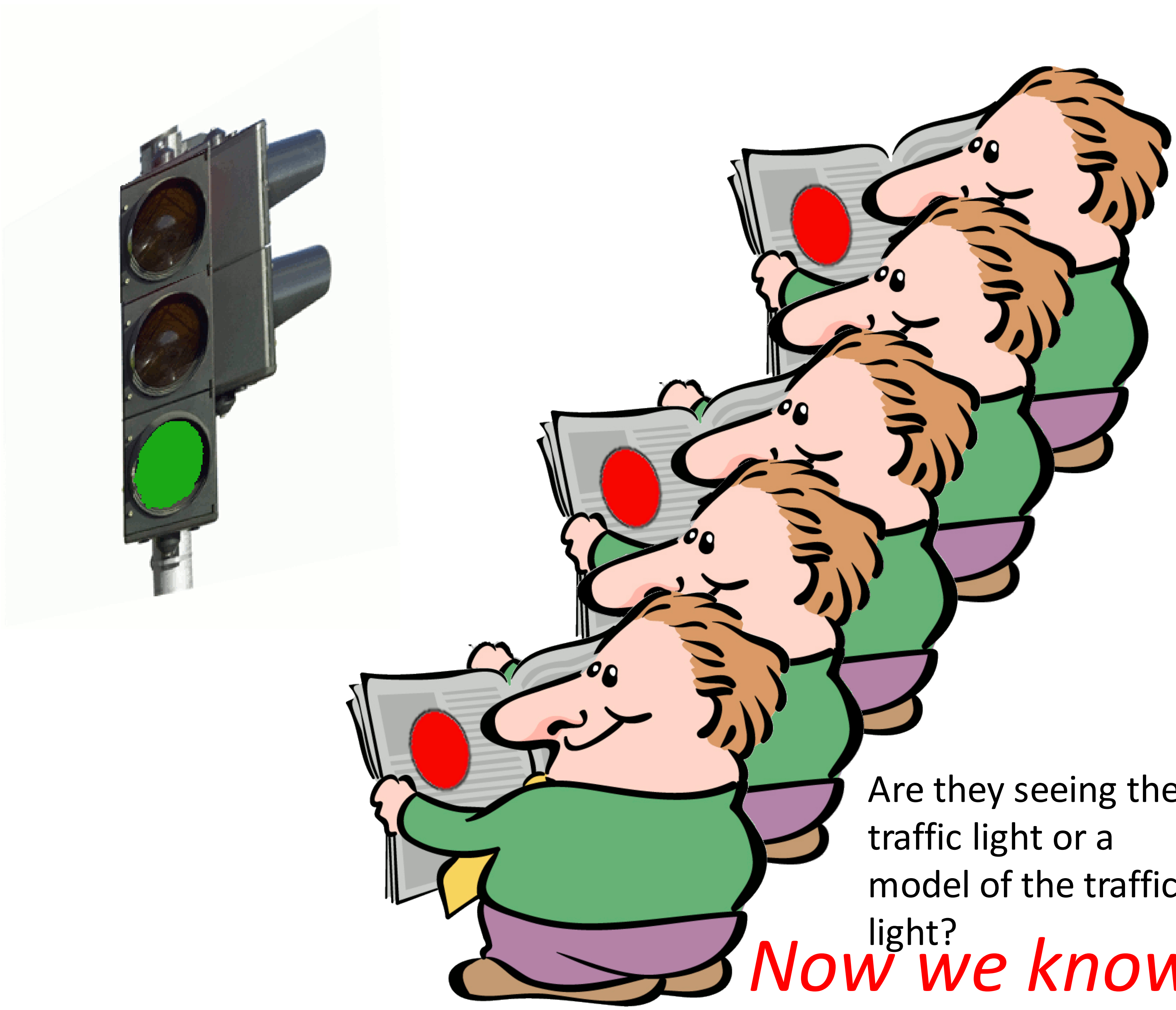
- on-tick : WorldState $\rightarrow$ WorldState
  - If the world is in state s, what state should it be in after the next tick?
- This is a specification of the desired behavior of the world.
- It's just a mathematical function.
- **`big-bang`** takes these functions and constructs a world that really behaves that way.

State is about sharing

- How can you tell the difference between a traffic light and a TLState?
- Ans: everybody sees the same traffic light.
- If its state changes everybody sees it.



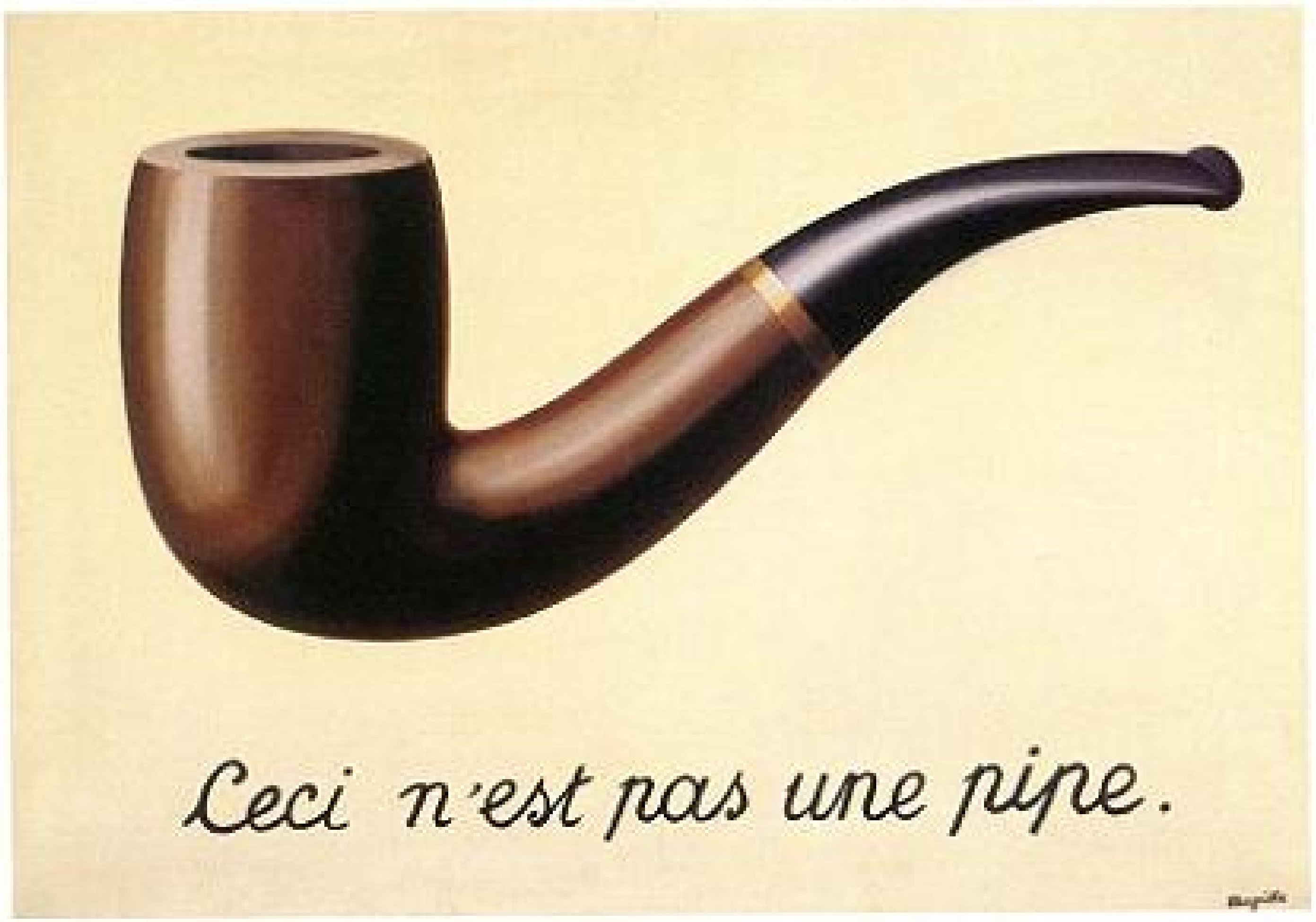
Are they seeing the traffic light or a model of the traffic light?



Are they seeing the traffic light or a model of the traffic light?

Now we know!

This is not a pipe



Blackboard metaphor

- State is like a blackboard: when I write on the blackboard, everybody sees it.
- I don't care who's in the room.
- I could distribute the changes by sending messages to each of you, but then I'd have to know who I needed to talk to.
  – That's what we do when we pass parameters in our programs!
- Blackboards foster collaboration

Where is the state?

- In World programs, there was just one piece of state– the world– and it was kept inside **`big-bang`**.
- In OO system, each object can have its own state.

An object is a little blackboard

- Lots of little blackboards!
- Of course you can't just look at a blackboard, you have to ask it questions, like

  ```
  (send traffic-light1 are-you-red?)
  ```
- This means we have to worry about patterns of collaboration again.

What has to be shared in our example?

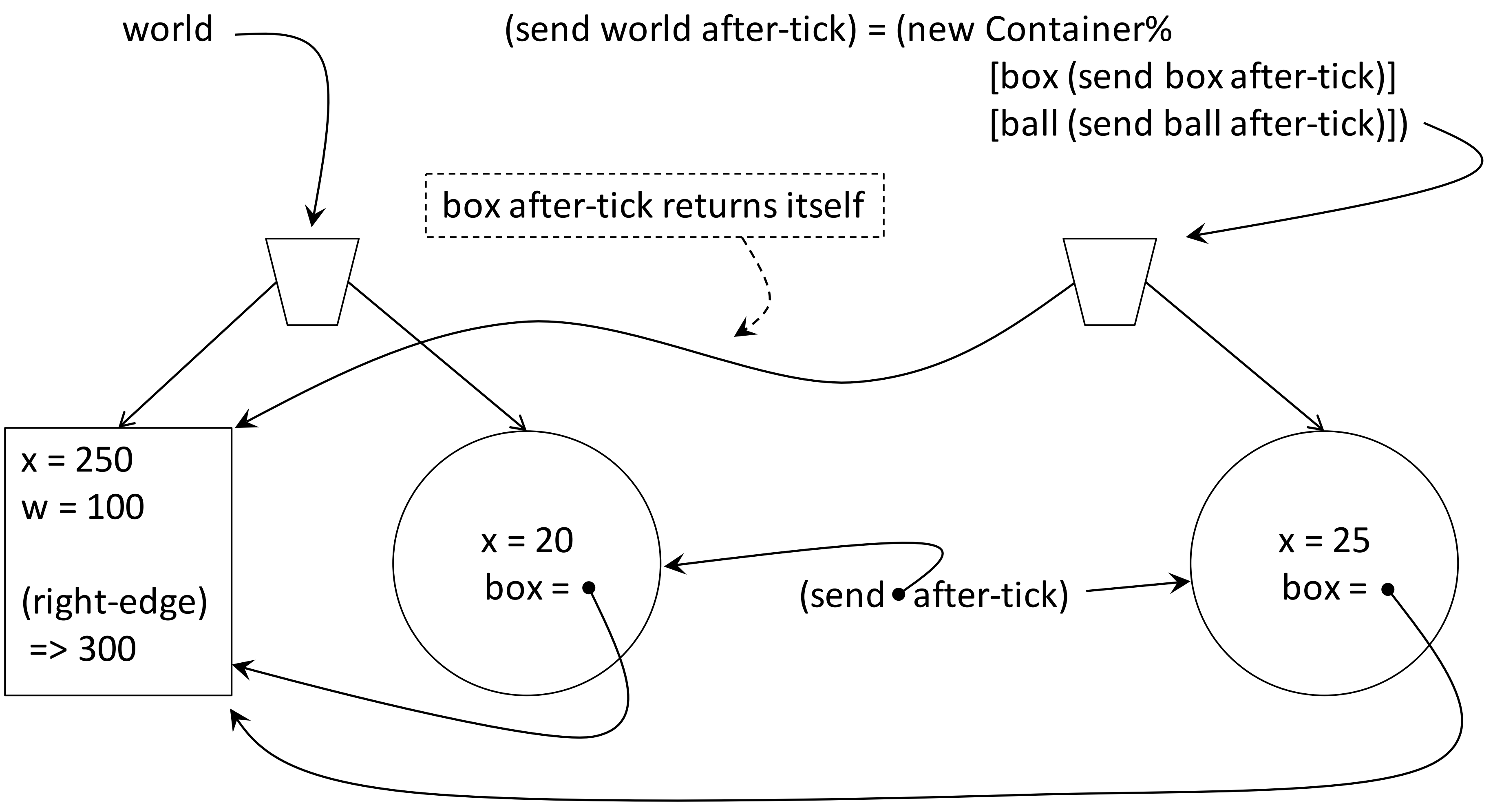
- When the frame changes, all the balls have to see it.
- The balls all have to look at the same frame.
- So let's make the frame stateful.
- [9-3-stateful-frame.rkt]
- Key contract (in Frame%):

```
on-mouse :
   Num Num MouseEvent -> Void
```

Let's compare these

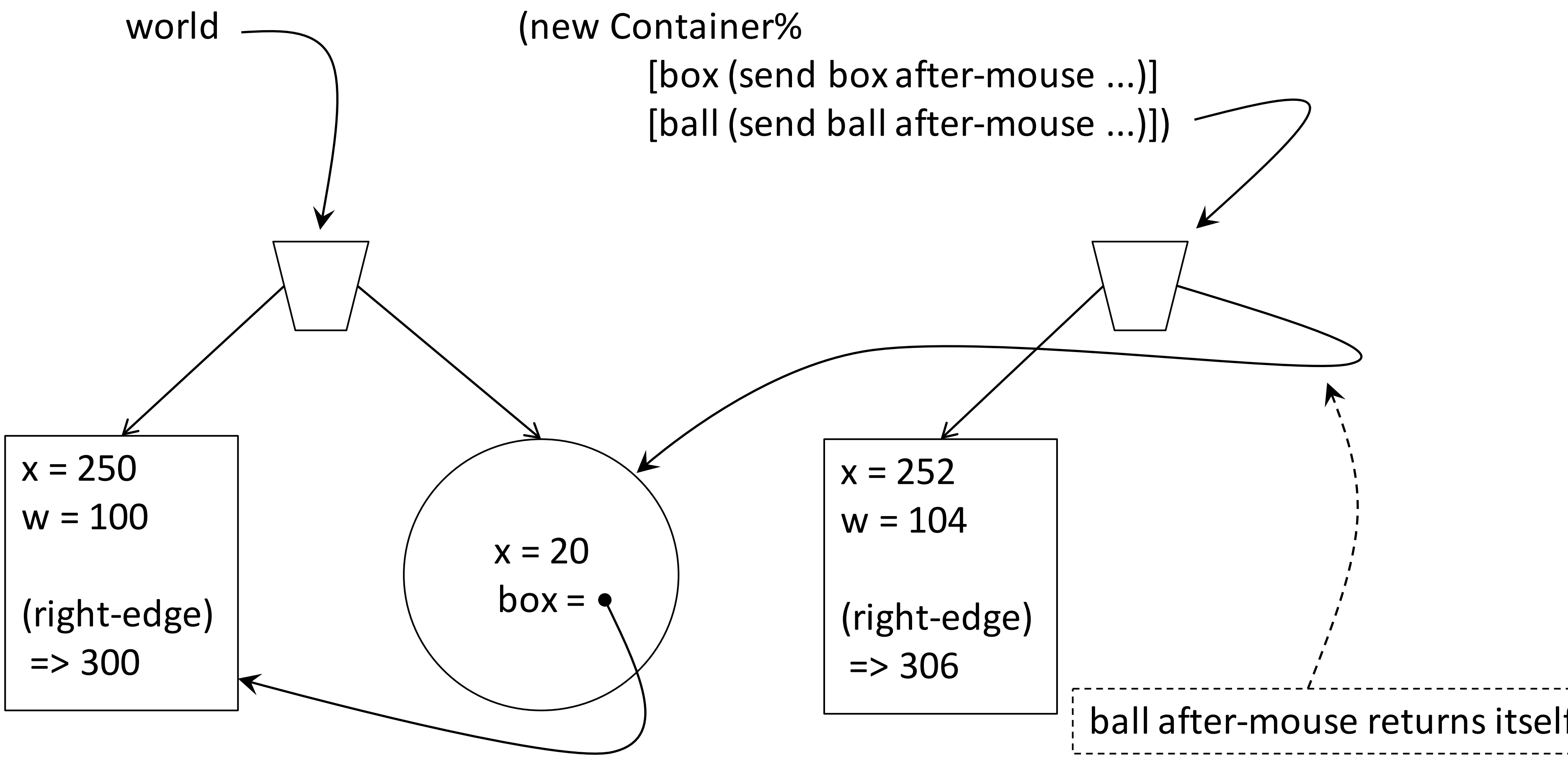
- We'll use a simple container with one box, called **`box`**, and one ball, called **`ball`**
- We'll call the world that big-bang sees **`world`**.

World-after-tick in 9-2


world
(send world after-tick) = (new Container%
[box (send box after-tick)]
[ball (send ball after-tick)])
box after-tick returns itself
x = 250
w = 100
(right-edge)
=> 300
x = 20
box =
(send • after-tick)
x = 25
box =

INVARIANT: the container's ball always points to the container's box

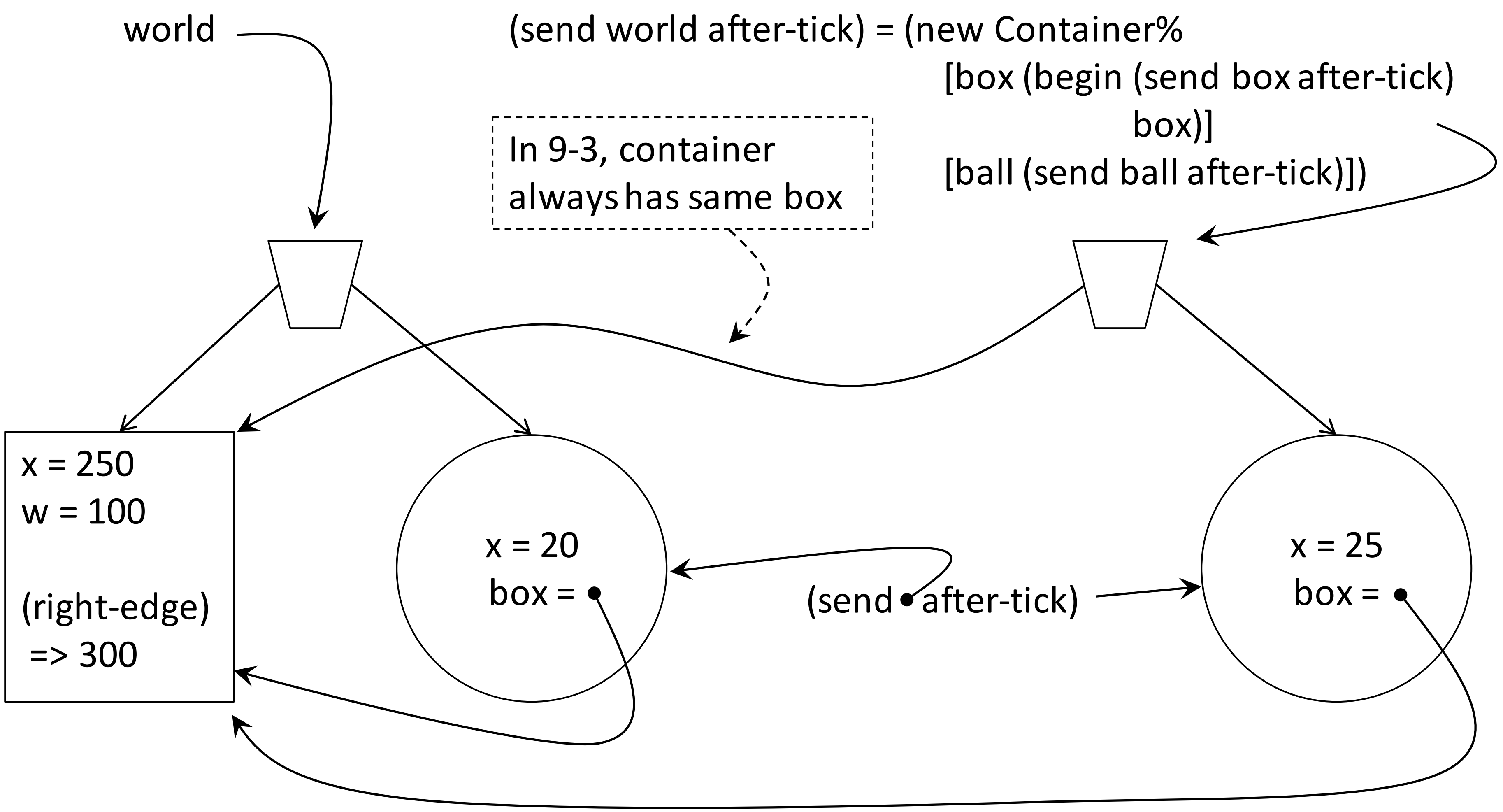
World after drag in 9-2


world
(new Container%
[box (send box after-mouse ...)]
[ball (send ball after-mouse ...)])
x = 250
w = 100
(right-edge)
=> 300
x = 20
box =
x = 252
w = 104
(right-edge)
=> 306
ball after-mouse returns itsel

INVARIANT: the container's ball always points to the container's box

FAIL!!!

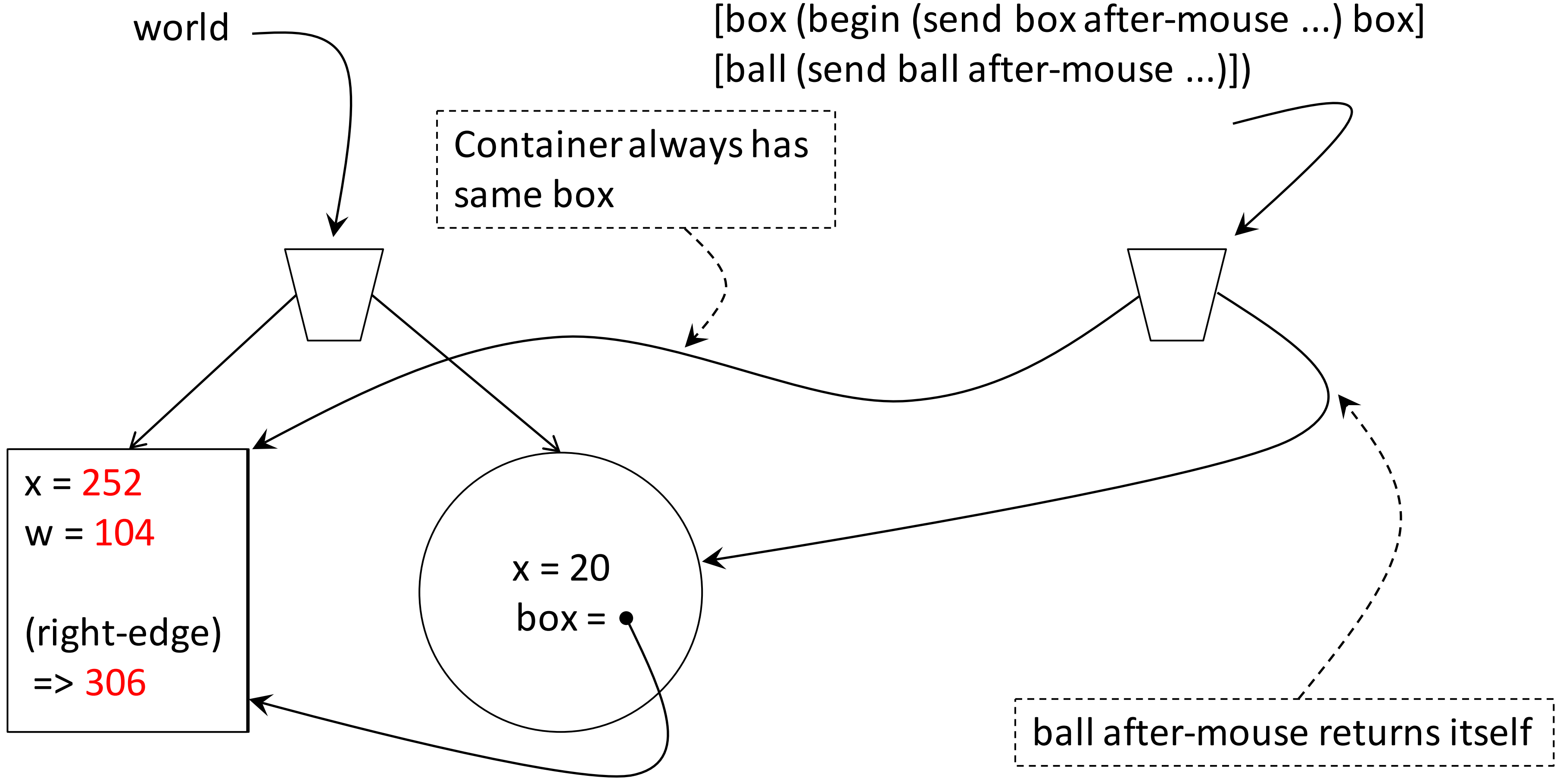
World-after-tick in 9-3


world
(send world after-tick) = (new Container%
[box (begin (send box after-tick)
box)]
[ball (send ball after-tick)])
In 9-3, container always has same box
x = 250
w = 100
(right-edge)
=> 300
x = 20
box =
(send after-tick)
x = 25
box =

INVARIANT: the container's ball always points to the container's box

World after drag in 9-3

```
(new Container%
         [box (begin (send box after-mouse ...) box]
         [ball (send ball after-mouse ...)])
```



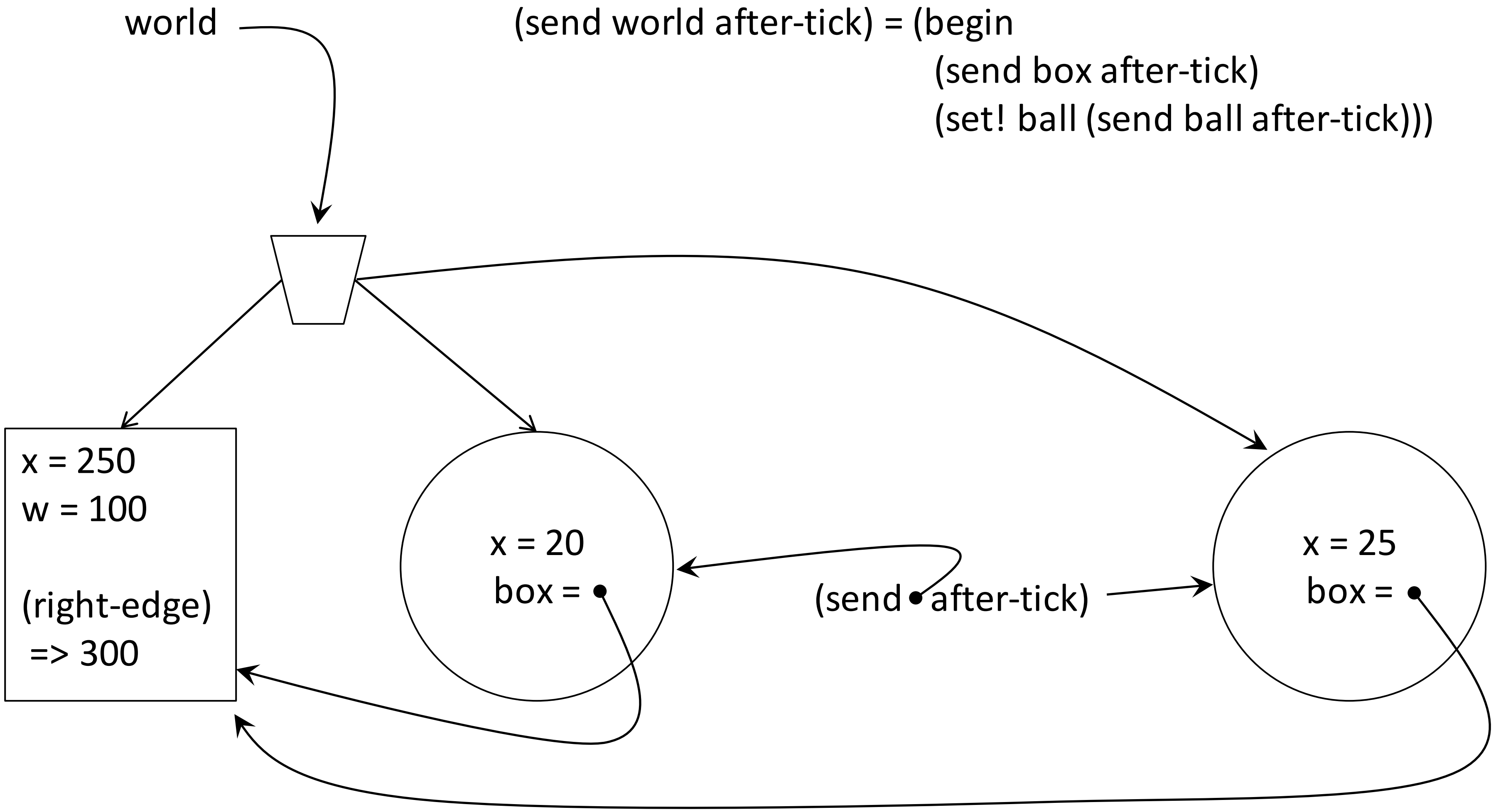
world
Container always has same box
x = 252
w = 104
(right-edge)
=> 306
x = 20
box =
ball after-mouse returns itself

INVARIANT: the container's ball always points to the container's box

WIN!

World-after-tick in 9-4

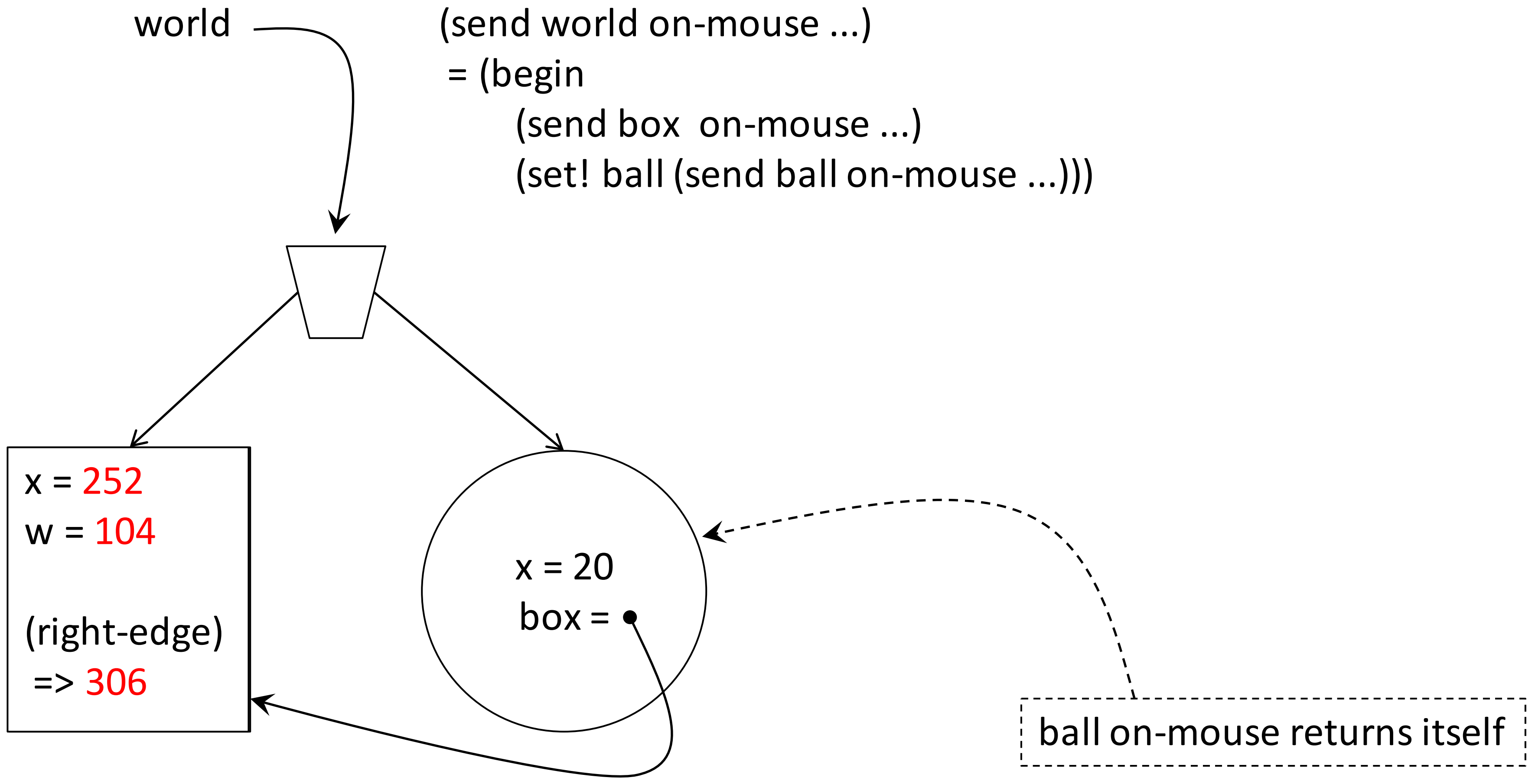
```
(send world after-tick) = (begin
                            (send box after-tick)
                            (set! ball (send ball after-tick)))
```



INVARIANT: the container's ball always points to the container's box

World after drag in 9-4

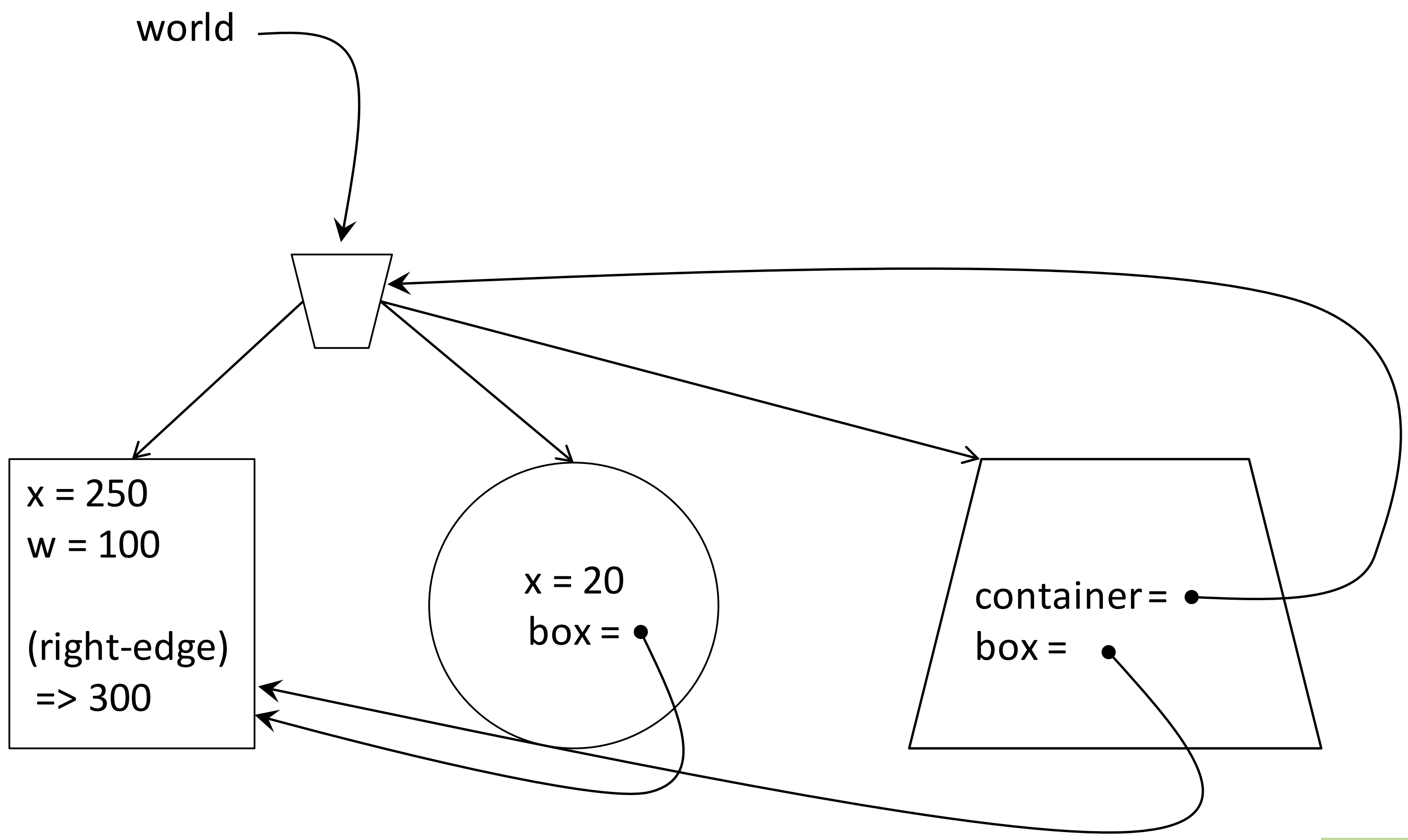
```
(send world on-mouse ...)
 = (begin
      (send box  on-mouse ...)
      (set! ball (send ball on-mouse ...)))
```



INVARIANT: the container's ball always points to the container's box

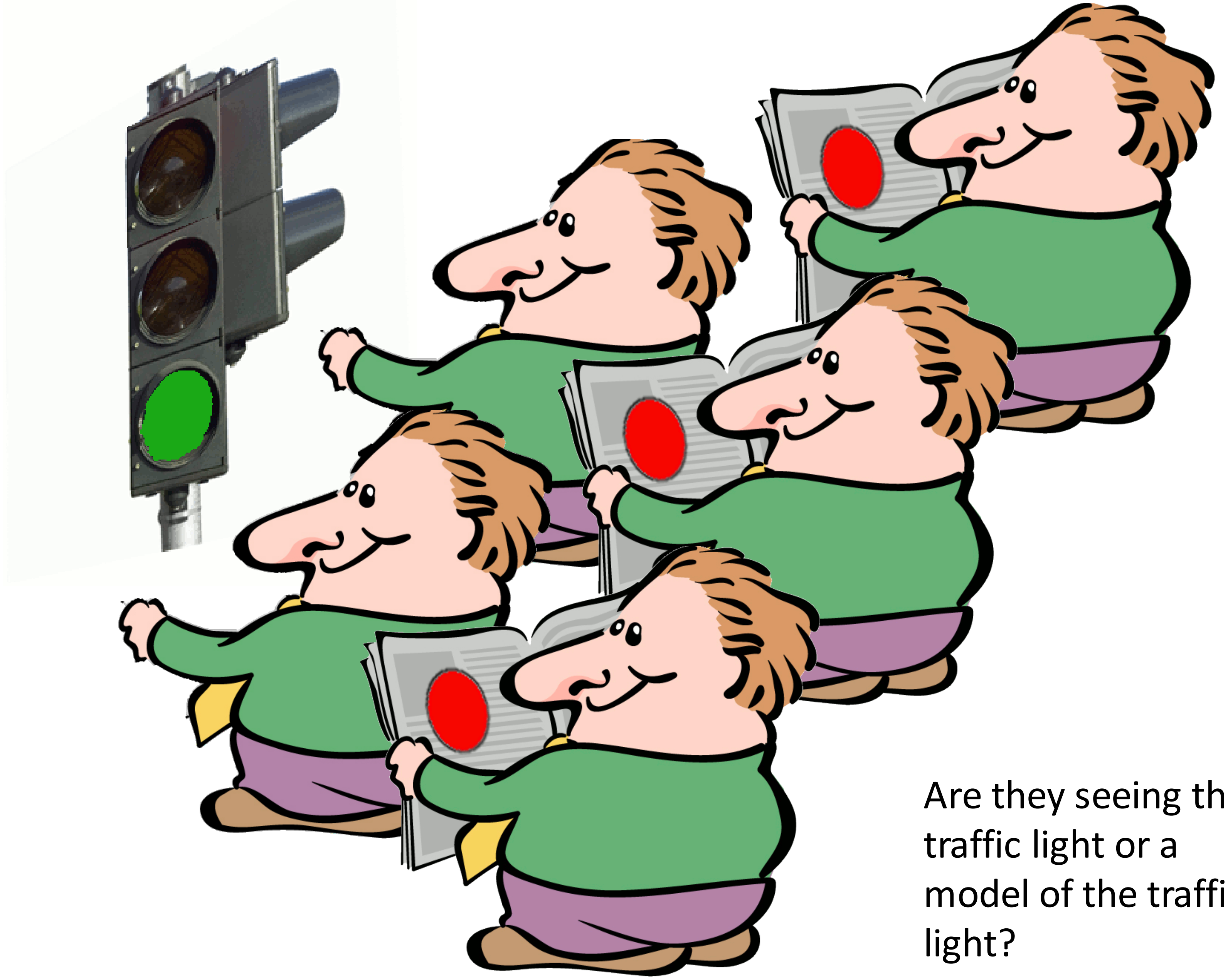
WIN!

Building the initial world in 9-5



INVARIANT: the container's ball always points to the container's box

WIN!



Are they seeing the traffic light or a model of the traffic light?

State makes testing harder

- You have to get things into the state you want
- Have observers that observe the relevant portions of the final state (at just the right time!)
- Then do tear-down
- Yecch!

Setting up a test scenario

```
(test-case
 "name of test"
 create objects for the test
 check to see that objects are initialized correctly
 (send obj1 method1 arg1 ...)
 check to see that objects have the right properties
 continue through sequence of events
 ...)
```

use probe methods

[09-5-ball-factory.rkt]

Java Guru on State:

Keep the state space of each object as simple as possible. If an object is immutable, it can be in only one state, and you win big. You never have to worry about what state the object is in, and you can share it freely, with no need for synchronization. If you can't make an object immutable, at least minimize the amount of mutation that is possible. This makes it easier to use the object correctly.

As an extreme example of what not to do, consider the case of java.util.Calendar. Very few people understand its state-space -- I certainly don't -- and it's been a constant source of bugs for years.

-- Joshua Bloch, Chief Java Architect, Google; author, *Effective Java*

Design Principle

- Use as little state as you can.
- Pass values whenever you can.

Summary

- We've studied the difference between a *value* (usually data) and a *state* (usually information)
- State enables objects to share information with objects that it doesn't know about.
- State makes testing and reasoning about your program harder.
- Use as little state as you can.
- Pass values whenever you can.