

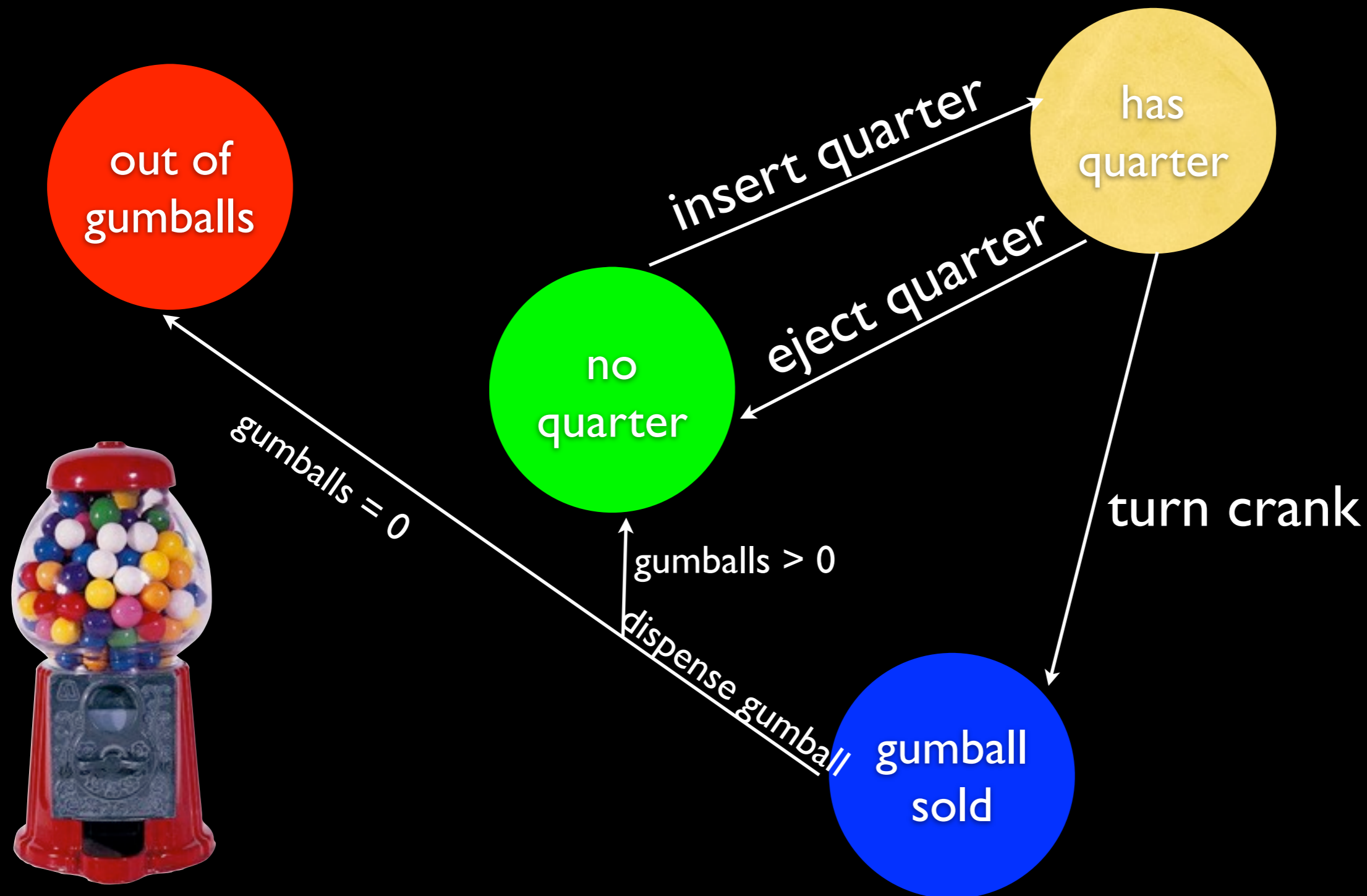
A brief introduction to the State Pattern

(a software engineering digression)

Emily Fortuna

(motivating example borrowed from Head First Design Patterns)

Suppose **electronic gumball machines** are the next big thing...



Great, let's write some code!

(Note: this method is **not** our final solution)



gather all of the states, and create instance variables to hold the states:

SOLD_OUT = 0

NO_QUARTER = 1

HAS_QUARTER = 2

SOLD = 3

Now, what are the **actions** that can happen in this system?

insert quarter **eject quarter** **turn crank**
dispense gumball

So these are our methods:

```
def insertQuarter
  if (state == HAS_QUARTER)
    puts "you can't insert another quarter"
  elsif (state == SOLD_OUT)
    puts "You can't insert a quarter; U no can haz gumballs :-("
  elsif (state == SOLD)
    puts "Please wait as we're already giving you a gumball... OM NOM"
  elsif (state == NO_QUARTER)
    state = HAS_QUARTER
    puts "You inserted a quarter!"
  end
end
```

But that's only one method!

```
class GumballMachine
  def insertQuarter
    # insert quarter code
  end

  def ejectQuarter
    # eject quarter code
  end

  def turnCrank
    # turn crank code
  end

  def dispense
    # dispense code
  end
end
```

But that's only one method!

```
class GumballMachine
  def insertQuarter
    # insert quarter code
  end

  def ejectQuarter
    # eject quarter code
  end

  def turnCrank
    # turn crank code
  end

  def dispense
    # dispense code
  end
end
```

**OMG so
many if/else
cases!!!**

It gets **WORSE** if you need to add more states:

```
SOLD_OUT = 0  
NO_QUARTER = 1  
HAS_QUARTER = 2  
SOLD = 3
```

```
def insertQuarter  
  if (state == HAS_QUARTER)  
    puts "you can't insert another quarter"  
  elsif (state == SOLD_OUT)  
    puts "You can't insert a quarter; U no can haz gumballs :-( "  
  elsif (state == SOLD)  
    puts "Please wait as we're already giving you a gumball... OM NOM"  
  elsif (state == NO_QUARTER)  
    state = HAS_QUARTER  
    puts "You inserted a quarter!"  
  end  
end
```

It gets **WORSE** if you need to add more states:

```
SOLD_OUT = 0  
NO_QUARTER = 1  
HAS_QUARTER = 2  
SOLD = 3  
WINNER = 4 # winner gets all gumballs!
```

```
def insertQuarter  
  if (state == HAS_QUARTER)  
    puts "you can't insert another quarter"  
  elsif (state == SOLD_OUT)  
    puts "You can't insert a quarter; U no can haz gumballs :-( "  
  elsif (state == SOLD)  
    puts "Please wait as we're already giving you a gumball... OM NOM"  
  elsif (state == NO_QUARTER)  
    state = HAS_QUARTER  
    puts "You inserted a quarter!"  
  elsif (state == WINNER)  
    puts "Please wait as we're giving you all of the gumballs... OM NOM NOM NOM"  
end
```


It gets **WORSE** if you need to add more states:

SOLD_OUT = 0
NO_QUARTER = 1
HAS_QUARTER = 2
SOLD = 3
WINNER = 4 # winner gets all gumballs!

Now you have to add a new case to every
other method, too! Do it three more times!

Mwahahaha!

```
def insertQuarter
  if (state == HAS_QUARTER)
    puts "you can't insert another quarter"
  elsif (state == SOLD_OUT)
    puts "You can't insert a quarter; U no can haz gumballs :-( "
  elsif (state == SOLD)
    puts "Please wait as we're already giving you a gumball... OM NOM"
  elsif (state == NO_QUARTER)
    state = HAS_QUARTER
    puts "You inserted a quarter!"
  elsif (state == WINNER)
    puts "Please wait as we're giving you all of the gumballs... OM NOM"
  end
end
```



Instead, let's create a bunch of State objects, that each know how to respond to different situations. For example:

```
class NoQuarterState
  def initialize(gumballMachine)
    @machine = gumballMachine
  end

  def insertQuarter
    @machine.setState(HasQuarterState.new(@machine))
  end

  def ejectQuarter
    puts "You haven't inserted a quarter yet" #here the state doesn't change
  end

  def turnCrank
    puts "You turned, but there's no quarter" #state stays the same
  end

  def dispense
    puts "You have to pay first!"
  end
end
```

Old GumballMachine class

```
class GumballMachine
  def insertQuarter
    # insert quarter code
  end

  def ejectQuarter
    # eject quarter code
  end

  def turnCrank
    # turn crank code
  end

  def dispense
    # dispense code
  end
end
```

Our lovely GumballMachine class now:

```
class GumballMachine
  def initialize
    @state = NoQuarterState.new(self)
  end

  def insertQuarter
    @state.insertQuarter
  end

  def ejectQuarter
    @state.ejectQuarter
  end

  def turnCrank
    @state.turnCrank
  end

  def dispense
    @state.dispense
  end
end
```



this concludes another edition of:

DESIGN PATTERNS TO THE RESCUE!

thank you for watching.