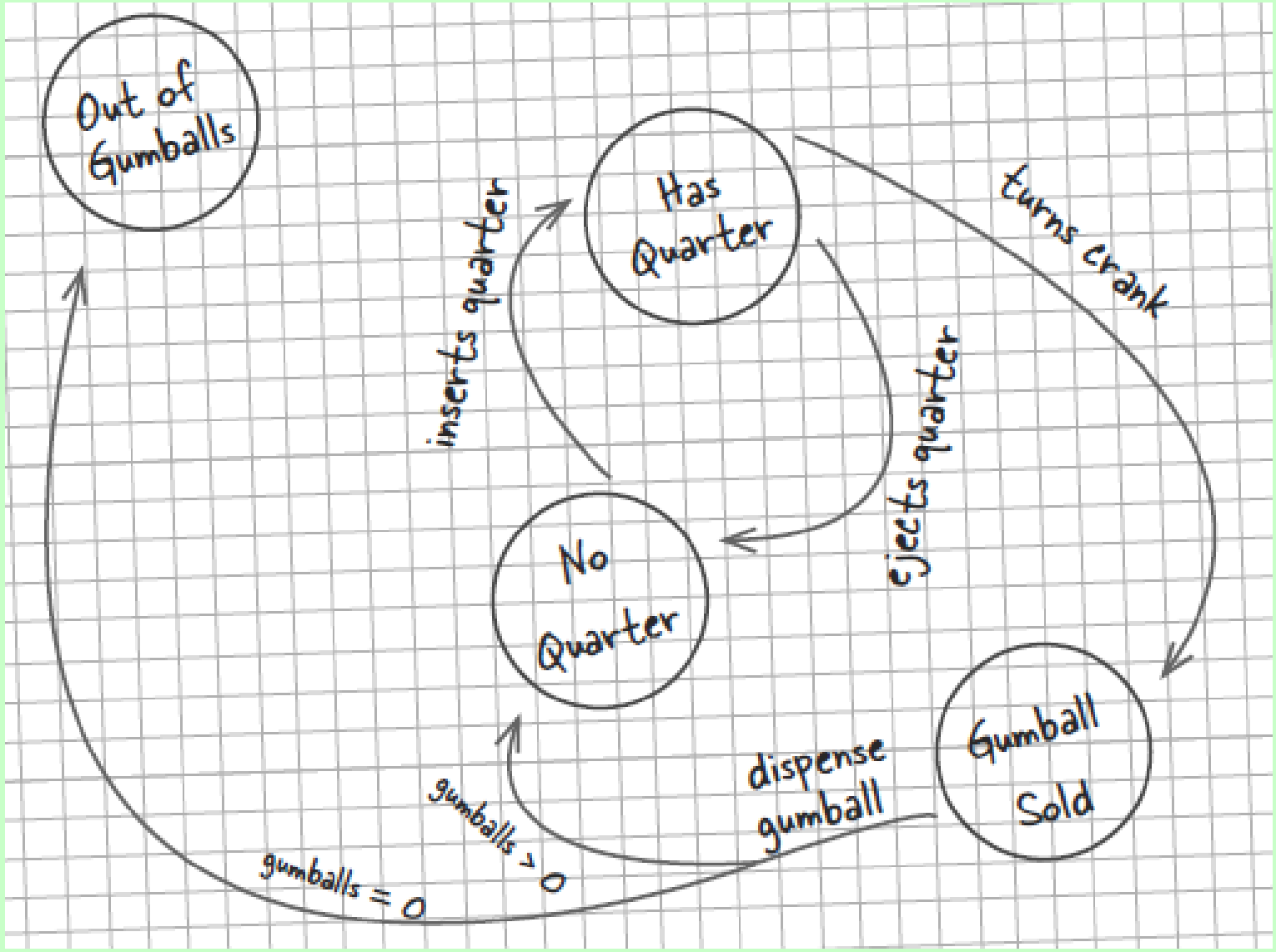
State Pattern

Problem

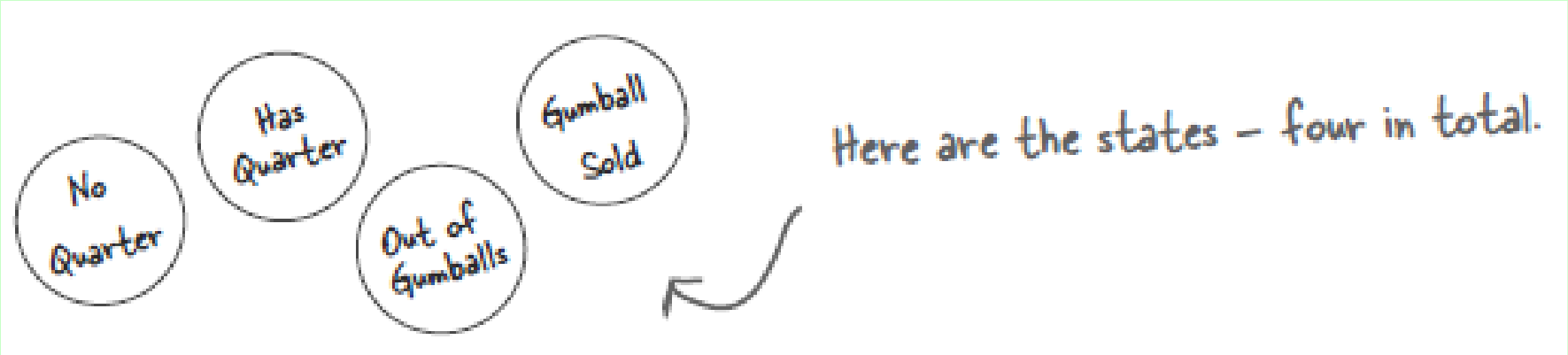
- Implement gumball machine
- The major manufacturers have found that by putting CPUs into their machines,
  - they can increase sales,
  - monitor inventory over the network
  - and measure customer satisfaction more accurately.

Gumball Machine State Diagram



From State Diagram to Code

1. First, gather up your states:



From State Diagram to Code

2. Next, create an instance variable to hold the current state, and define values for each of the states:

Let's just call "Out of Gumballs" "Sold Out" for short.

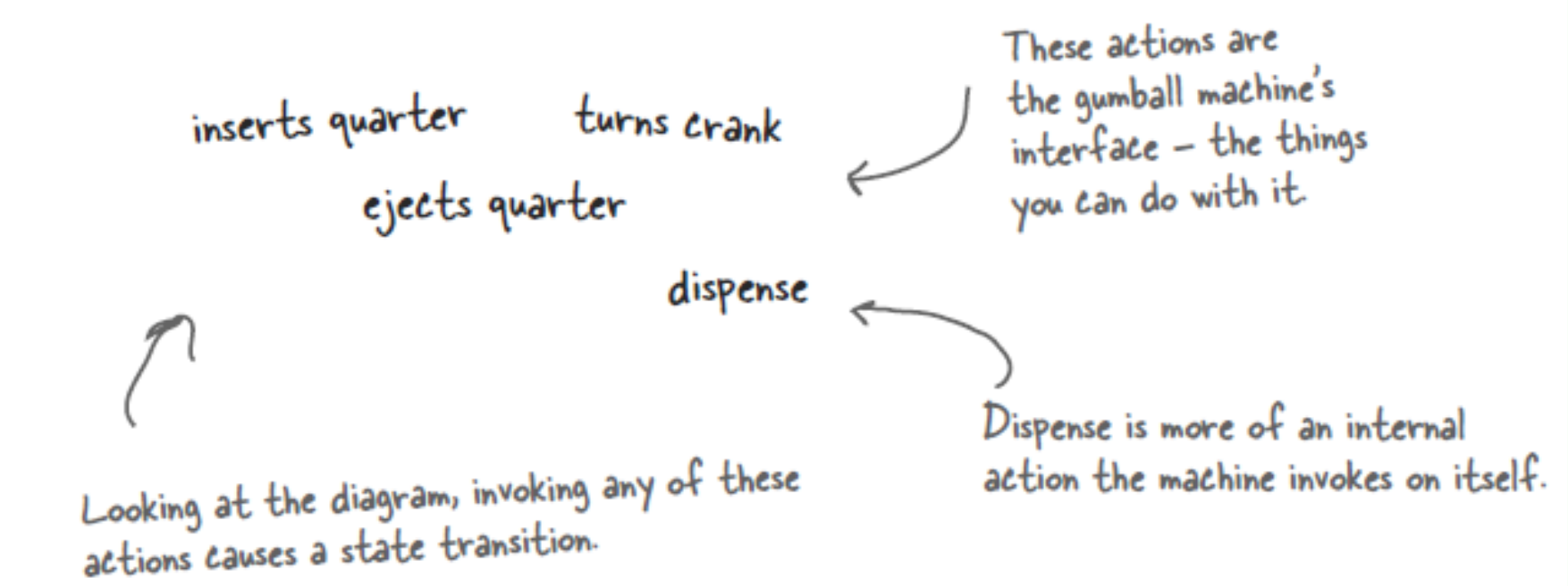
```
final static int SOLD_OUT = 0;
final static int NO_QUARTER = 1;
final static int HAS_QUARTER = 2;
final static int SOLD = 3;

int state = SOLD_OUT;
```

Here's each state represented as a unique integer...

...and here's an instance variable that holds the current state. We'll go ahead and set it to "Sold Out" since the machine will be unfilled when it's first taken out of its box and turned on.

From State Diagram to Code

3. Now we gather up all the actions that can happen in the system:



From State Diagram to Code

4. Now we create a class that acts as the state machine.
   - For each action, we create a method that uses conditional statements to determine what behavior is appropriate in each state.

   - public void insertQuarter()
   - public void ejectQuarter()
   - public void turnCrank()
   - public void dispense()

Constructor

- Set the number all gumballs and set the state

```
public GumballMachine(int count) {
    this.count = count;
    if (count > 0) {
        state = NO_QUARTER;
    }
}
```

The constructor takes an initial inventory of gumballs. If the inventory isn't zero, the machine enters state NO_QUARTER, meaning it is waiting for someone to insert a quarter, otherwise it stays in the SOLD_OUT state.

insertQuarter()

Now we start implementing the actions as methods....

When a quarter is inserted, if....

```
public void insertQuarter() {
    if (state == HAS_QUARTER) {
        System.out.println("You can't insert another quarter");
    } else if (state == NO_QUARTER) {
        state = HAS_QUARTER;
        System.out.println("You inserted a quarter");
    } else if (state == SOLD_OUT) {
        System.out.println("You can't insert a quarter, the machine is sold out");
    } else if (state == SOLD) {
        System.out.println("Please wait, we're already giving you a gumball");
    }
}
```

a quarter is already inserted we tell the customer;

otherwise we accept the quarter and transition to the HAS_QUARTER state.

If the customer just bought a gumball he needs to wait until the transaction is complete before inserting another quarter.

and if the machine is sold out, we reject the quarter.

ejectQuarter()

```
public void ejectQuarter() {
    if (state == HAS_QUARTER) {
        System.out.println("Quarter returned");
        state = NO_QUARTER;
    } else if (state == NO_QUARTER) {
        System.out.println("You haven't inserted a quarter");
    } else if (state == SOLD) {
        System.out.println("Sorry, you already turned the crank");
    } else if (state == SOLD_OUT) {
        System.out.println("You can't eject, you haven't inserted a quarter yet");
    }
}
```

Now, if the customer tries to remove the quarter...

If there is a quarter, we return it and go back to the NO_QUARTER state.

Otherwise, if there isn't one we can't give it back.

If the customer just turned the crank, we can't give a refund; he already

You can't eject if the machine is sold out, it doesn't accept quarters!

turnCrank()

The customer tries to turn the crank...

give a refund; he already has the gumball!

```
public void turnCrank() {
    if (state == SOLD) {
        System.out.println("Turning twice doesn't get you another gumball!");
    } else if (state == NO_QUARTER) {
        System.out.println("You turned but there's no quarter");
    } else if (state == SOLD_OUT) {
        System.out.println("You turned, but there are no gumballs");
    } else if (state == HAS_QUARTER) {
        System.out.println("You turned...");
        state = SOLD;
        dispense();
    }
}
```

Someone's trying to cheat the machine.

We need a quarter first.

We can't deliver gumballs; there are none.

Success! They get a gumball. Change the state to SOLD and call the machine's dispense() method.

Called to dispense a gumball.

dispense()

Called to dispense a gumball.

machine's dispense() method.

```
public void dispense() {
    if (state == SOLD) {
        System.out.println("A gumball comes rolling out the slot");
        count = count - 1;
        if (count == 0) {
            System.out.println("Oops, out of gumballs!");
            state = SOLD_OUT;
        } else {
            state = NO_QUARTER;
        }
    } else if (state == NO_QUARTER) {
        System.out.println("You need to pay first");
    } else if (state == SOLD_OUT) {
        System.out.println("No gumball dispensed");
    } else if (state == HAS_QUARTER) {
        System.out.println("No gumball dispensed");
    }
}
```

We're in the SOLD state; give 'em a gumball!

Here's where we handle the "out of gumballs" condition: If this was the last one, we set the machine's state to SOLD_OUT; otherwise, we're back to not having a quarter.

None of these should ever happen, but if they do, we give 'em an error, not a gumball.

Testing the Machine

```
public class GumballMachineTestDrive {
    public static void main(String[] args) {
        GumballMachine gumballMachine = new GumballMachine(5);

        System.out.println(gumballMachine);

        gumballMachine.insertQuarter();
        gumballMachine.turnCrank();

        System.out.println(gumballMachine);

        gumballMachine.insertQuarter();
        gumballMachine.ejectQuarter();
        gumballMachine.turnCrank();

        System.out.println(gumballMachine);

        gumballMachine.insertQuarter();
        gumballMachine.turnCrank();
        gumballMachine.insertQuarter();
        gumballMachine.turnCrank();
        gumballMachine.ejectQuarter();

        System.out.println(gumballMachine);

        gumballMachine.insertQuarter();
        gumballMachine.insertQuarter();
        gumballMachine.turnCrank();
        gumballMachine.insertQuarter();
        gumballMachine.turnCrank();
        gumballMachine.insertQuarter();
        gumballMachine.turnCrank();
```

Load it up with five gumballs total.

Print out the state of the machine.

Throw a quarter in...

Turn the crank; we should get our gumball.

Print out the state of the machine, again.

Throw a quarter in...

Ask for it back.

Turn the crank; we shouldn't get our gumball.

Print out the state of the machine, again.

Throw a quarter in...

Turn the crank; we should get our gumball

Throw a quarter in...

Turn the crank; we should get our gumball

Ask for a quarter back we didn't put in.

Print out the state of the machine, again.

Throw TWO quarters in...

Turn the crank; we should get our gumball.

Now for the stress testing... ☺

File Edit Window Help mightygumball.com

```
%java GumballMachineTestDrive

Mighty Gumball, Inc.
Java-enabled Standing Gumball Model #2004
Inventory: 5 gumballs
Machine is waiting for quarter

You inserted a quarter
You turned...
A gumball comes rolling out the slot

Mighty Gumball, Inc.
Java-enabled Standing Gumball Model #2004
Inventory: 4 gumballs
Machine is waiting for quarter

You inserted a quarter
Quarter returned
You turned but there's no quarter

Mighty Gumball, Inc.
Java-enabled Standing Gumball Model #2004
Inventory: 4 gumballs
Machine is waiting for quarter

You inserted a quarter
You turned...
A gumball comes rolling out the slot
You inserted a quarter
You turned...
A gumball comes rolling out the slot
You haven't inserted a quarter

Mighty Gumball, Inc.
Java-enabled Standing Gumball Model #2004
Inventory: 2 gumballs
Machine is waiting for quarter

You inserted a quarter
You can't insert another quarter
You turned...
A gumball comes rolling out the slot
You inserted a quarter
You turned...
A gumball comes rolling out the slot
Oops, out of gumballs!
You can't insert a quarter, the machine is sold out
You turned, but there are no gumballs

Mighty Gumball, Inc.
Java-enabled Standing Gumball Model #2004
Inventory: 0 gumballs
Machine is sold out
```

Change

- "No man ever steps in the same river twice."
  - » Heraclitus of Ephesus
- "Chage is the only constant."

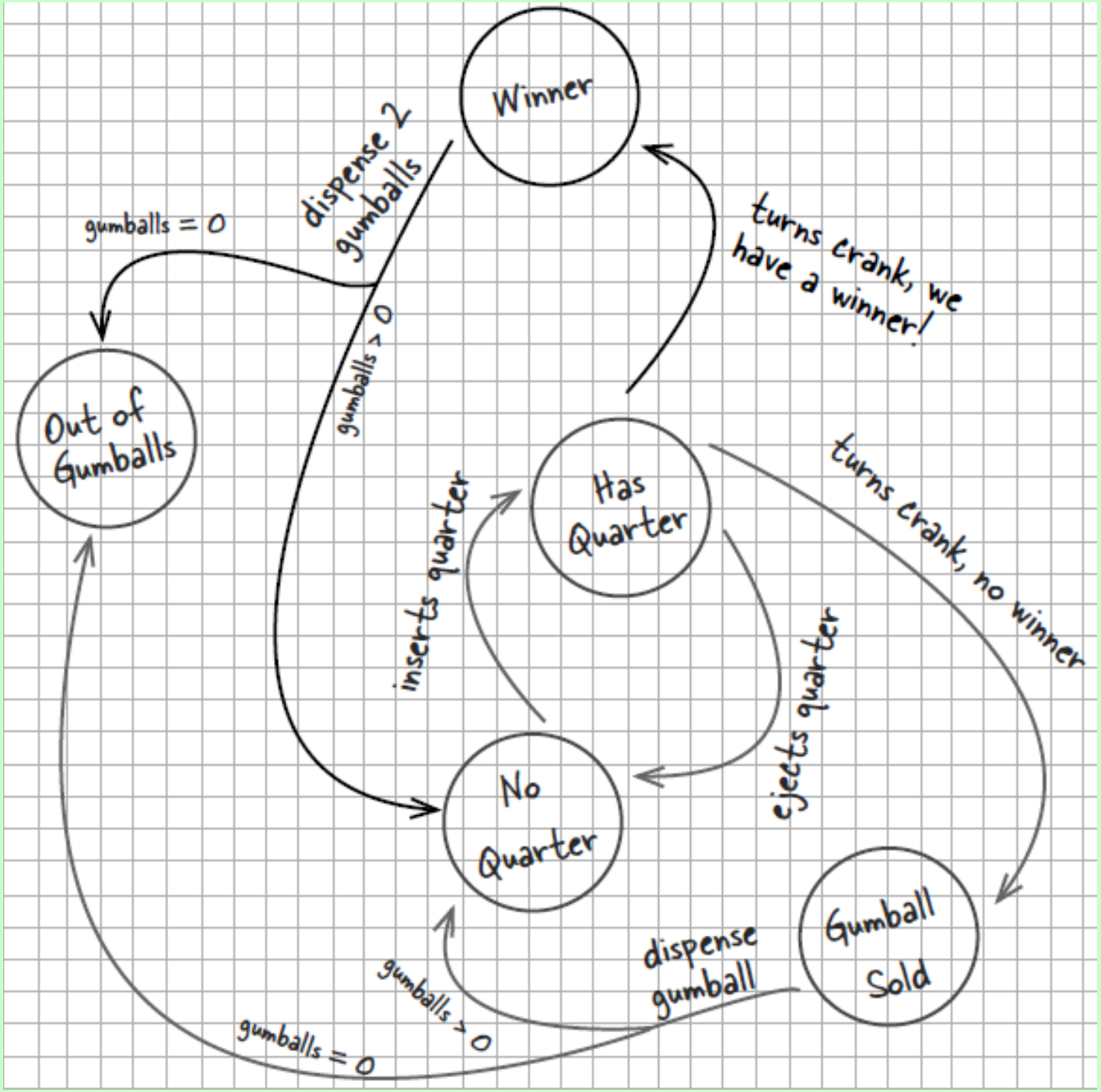
We think that by turning "gumball buying" into a game we can significantly increase our sales. We're going to put one of these stickers on every machine. We're so glad we've got Java in the machines because this is going to be easy, right?

Be a Winner! One in Ten get a FREE GUMBALL

CEO, Mighty Gumball, Inc.

JawBreaker or Gumdrop?

10% of the time, when the crank is turned, the customer gets two gumballs instead of one.


Winner
dispense 2 gumballs
gumballs = 0
gumballs > 0
turns crank, we have a winner!
Out of Gumballs
Has Quarter
turns crank, no winner
inserts quarter
ejects quarter
No Quarter
Gumball Sold
dispense gumball
gumballs > 0
gumballs = 0

Messy STATE of things

```
final static int SOLD_OUT = 0;
final static int NO_QUARTER = 1;
final static int HAS_QUARTER = 2;
final static int SOLD = 3;


public void insertQuarter() {
    // insert quarter code here
}

public void ejectQuarter() {
    // eject quarter code here
}

public void turnCrank() {
    // turn crank code here
}

public void dispense() {
    // dispense code here
}
```

First, you'd have to add a new WINNER state here. That isn't too bad...

... but then, you'd have to add a new conditional in every single method to handle the WINNER state; that's a lot of code to modify.

turnCrank() will get especially messy, because you'd have to add code to check to see whether you've got a WINNER and then switch to either the WINNER state or the SOLD state.

Observations

- This code certainly isn’t adhering to the Open Closed Principle!
- This design isn’t even very object oriented.
- State transitions aren’t explicit; they are buried in the middle of a bunch of conditional code.
- We haven’t encapsulated anything that varies here.
- Further additions are likely to cause bugs in working code.

New Design

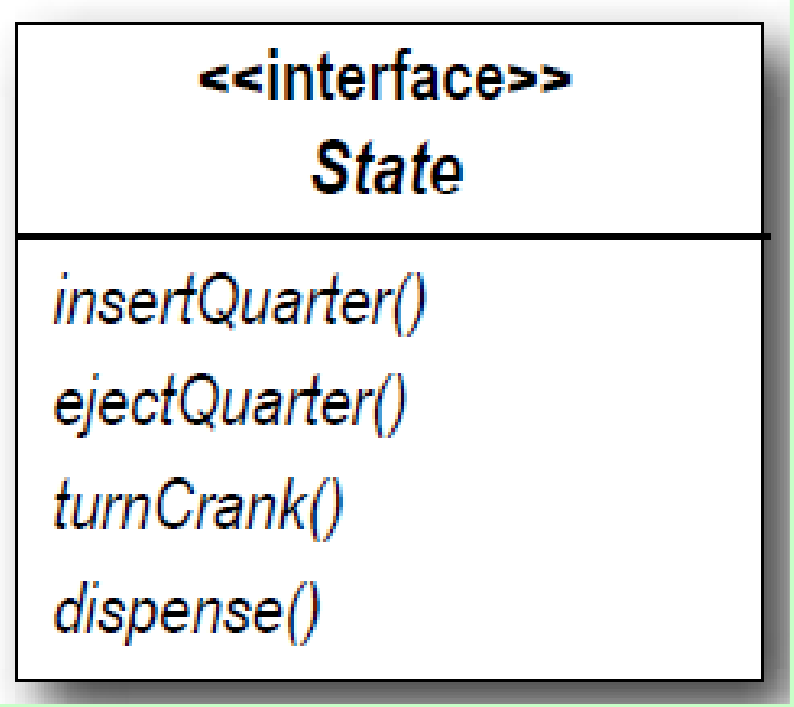
1. First, we’re going to define a State interface that contains a method for every action in the Gumball Machine.

2. Then we’re going to implement a State class for every state of the machine. These classes will be responsible for the behavior of the machine when it is in the corresponding state.

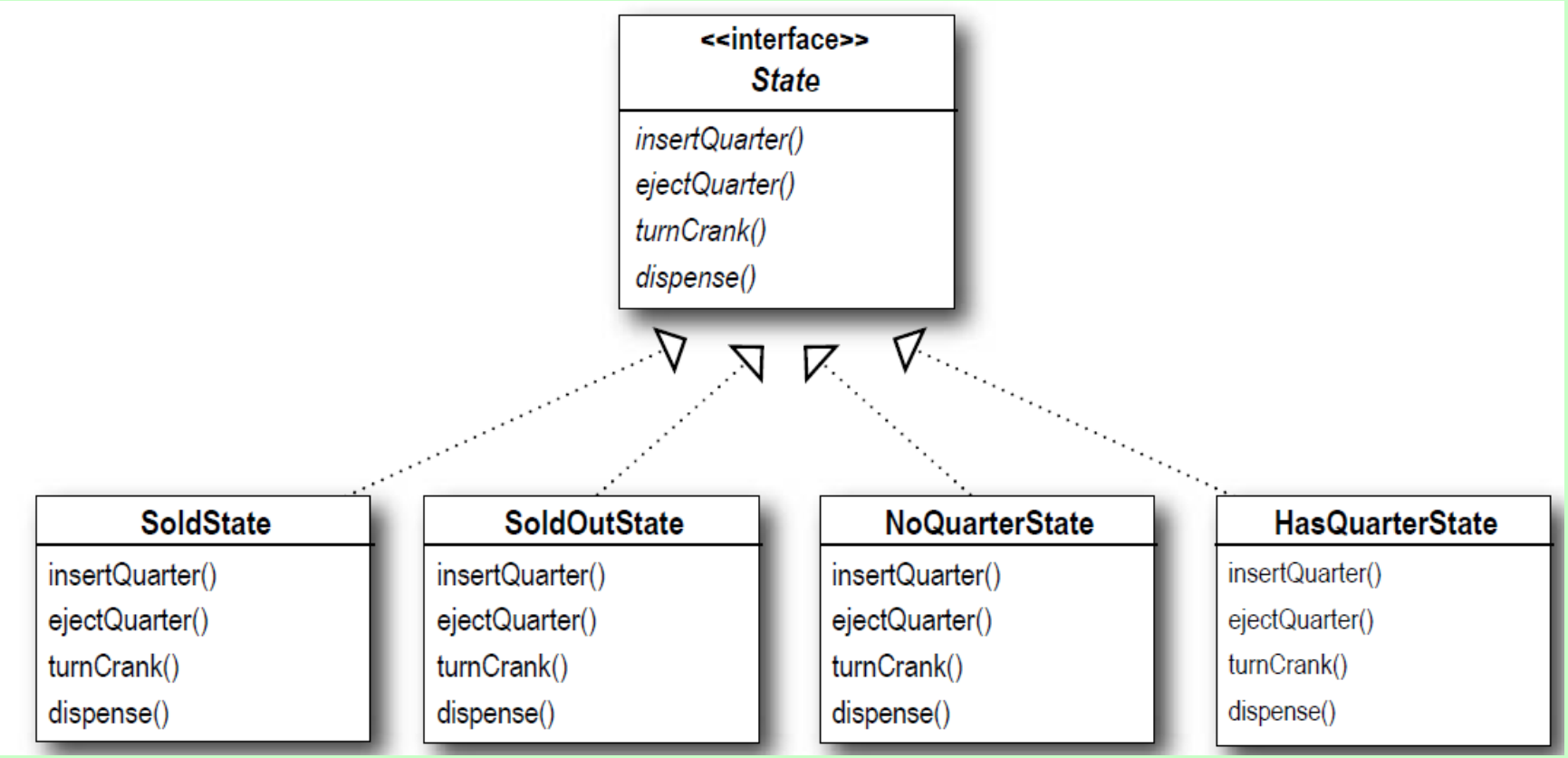
3. Finally, we’re going to get rid of all of our conditional code and instead delegate to the state class to do the work for us.

Defining the State Interface

- First let’s create an interface for State, which all our states implement
- The methods map directly to actions that could happen to the Gumball Machine

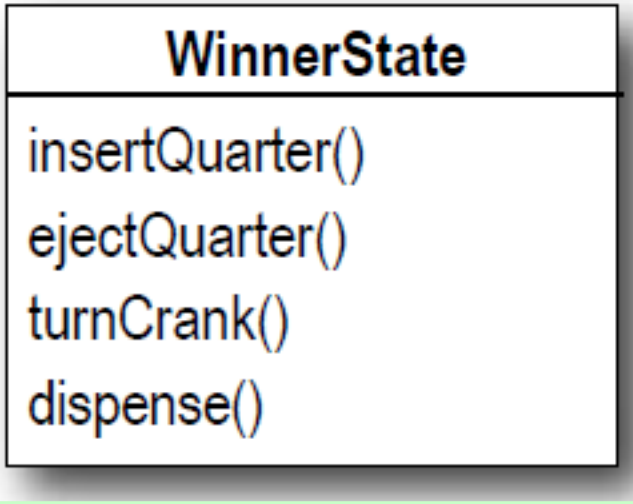


Defining the State Classes



Defining the State Classes

Don't forget, we need a new "winner" state too that implements the state interface. We'll come back to this after we reimplement the first version of the Gumball Machine.



Implementing our State classes

First we need to implement the State interface.

We get passed a reference to the Gumball Machine through the constructor. We're just going to stash this in an instance variable.

```
public class NoQuarterState implements State {
    GumballMachine gumballMachine;

    public NoQuarterState(GumballMachine gumballMachine) {
        this.gumballMachine = gumballMachine;
    }

    public void insertQuarter() {
        System.out.println("You inserted a quarter");
        gumballMachine.setState(gumballMachine.getHasQuarterState());
    }

    public void ejectQuarter() {
        System.out.println("You haven't inserted a quarter");
    }

    public void turnCrank() {
        System.out.println("You turned, but there's no quarter");
     }

    public void dispense() {
        System.out.println("You need to pay first");
    }
}
```

If someone inserts a quarter, we print a message saying the quarter was accepted and then change the machine's state to the HasQuarterState.

You'll see how these work in just a sec...

You can't get money back if you never gave it to us!

And, you can't get a gumball if you don't pay us.

We can't be dispensing gumballs without payment.

Implementing our State classes

```
public class HasQuarterState implements State {
    GumballMachine gumballMachine;

    public HasQuarterState(GumballMachine gumballMachine) {
        this.gumballMachine = gumballMachine;
    }

    public void insertQuarter() {
        System.out.println("You can't insert another quarter");
    }

    public void ejectQuarter() {
        System.out.println("Quarter returned");
        gumballMachine.setState(gumballMachine.getNoQuarterState());
    }

    public void turnCrank() {
        System.out.println("You turned...");
        gumballMachine.setState(gumballMachine.getSoldState());
    }
    public void dispense() {
        System.out.println("No gumball dispensed");
    }
}
```

When the state is instantiated we pass it a reference to the GumballMachine. This is used to transition the machine to a different state.

An inappropriate action for this state.

Return the customer's quarter and transition back to the NoQuarterState.

When the crank is turned we transition the machine to the SoldState state by calling its setState() method and passing it the SoldState object. The SoldState object is retrieved by the getSoldState() getter method (there is one of these getter methods for each state).

Another inappropriate action for this state.

Implementing our State classes

```
public class SoldState implements State {
    //constructor and instance variables here

    public void insertQuarter() {
        System.out.println("Please wait, we're already giving you a gumball");
    }

    public void ejectQuarter() {
        System.out.println("Sorry, you already turned the crank");
    }

    public void turnCrank() {
        System.out.println("Turning twice doesn't get you another gumball!");
    }

    public void dispense() {
        gumballMachine.releaseBall();
        if (gumballMachine.getCount() > 0) {
            gumballMachine.setState(gumballMachine.getNoQuarterState());
        } else {
            System.out.println("Oops, out of gumballs!");
            gumballMachine.setState(gumballMachine.getSoldOutState());
        }
    }
}
```

Here are all the inappropriate actions for this state

And here's where the real work begins...

We're in the SoldState, which means the customer paid. So, we first need to ask the machine to release a gumball.

Then we ask the machine what the gumball count is, and either transition to the NoQuarterState or the SoldOutState.

Implementing our State classes

In the Sold Out state, we really can't do anything until someone refills the Gumball Machine.

```
public class SoldOutState implements State {
    GumballMachine gumballMachine;

    public SoldOutState(GumballMachine gumballMachine) {
        this.gumballMachine = gumballMachine;
    }

    public void insertQuarter() {
        System.out.println("You can't insert a quarter, the machine is sold out");
    }

    public void ejectQuarter() {
        System.out.println("You can't eject, you haven't inserted a quarter yet");
    }

    public void turnCrank() {
        System.out.println("You turned, but there are no gumballs");
    }

    public void dispense() {
        System.out.println("No gumball dispensed");
    }

}
```

GumballMachine class

```
public class GumballMachine {

    State soldOutState;
    State noQuarterState;
    State hasQuarterState;
    State soldState;

    State state = soldOutState;
    int count = 0;

    public GumballMachine(int numberGumballs) {
        soldOutState = new SoldOutState(this);
        noQuarterState = new NoQuarterState(this);
        hasQuarterState = new HasQuarterState(this);
        soldState = new SoldState(this);
        this.count = numberGumballs;
        if (numberGumballs > 0) {
            state = noQuarterState;
        }
    }
```

Here are all the States again...

...and the State instance variable.

The count instance variable holds the count of gumballs – initially the machine is empty.

Our constructor takes the initial number of gumballs and stores it in an instance variable.

It also creates the State instances, one of each.

If there are more than 0 gumballs we set the state to the NoQuarterState.

GumballMachine class

```
public void insertQuarter() {
    state.insertQuarter();
}

public void ejectQuarter() {
    state.ejectQuarter();
}

public void turnCrank() {
    state.turnCrank();
    state.dispense();
}

void setState(State state) {
    this.state = state;
}

void releaseBall() {
    System.out.println("A gumball comes rolling out the slot...");
    if (count != 0) {
        count = count - 1;
    }
}
```

Now for the actions. These are VERY EASY to implement now. We just delegate to the current state.

Note that we don't need an action method for dispense() in GumballMachine because it's just an internal action; a user can't ask the machine to dispense directly. But we do call dispense() on the State object from the turnCrank() method.

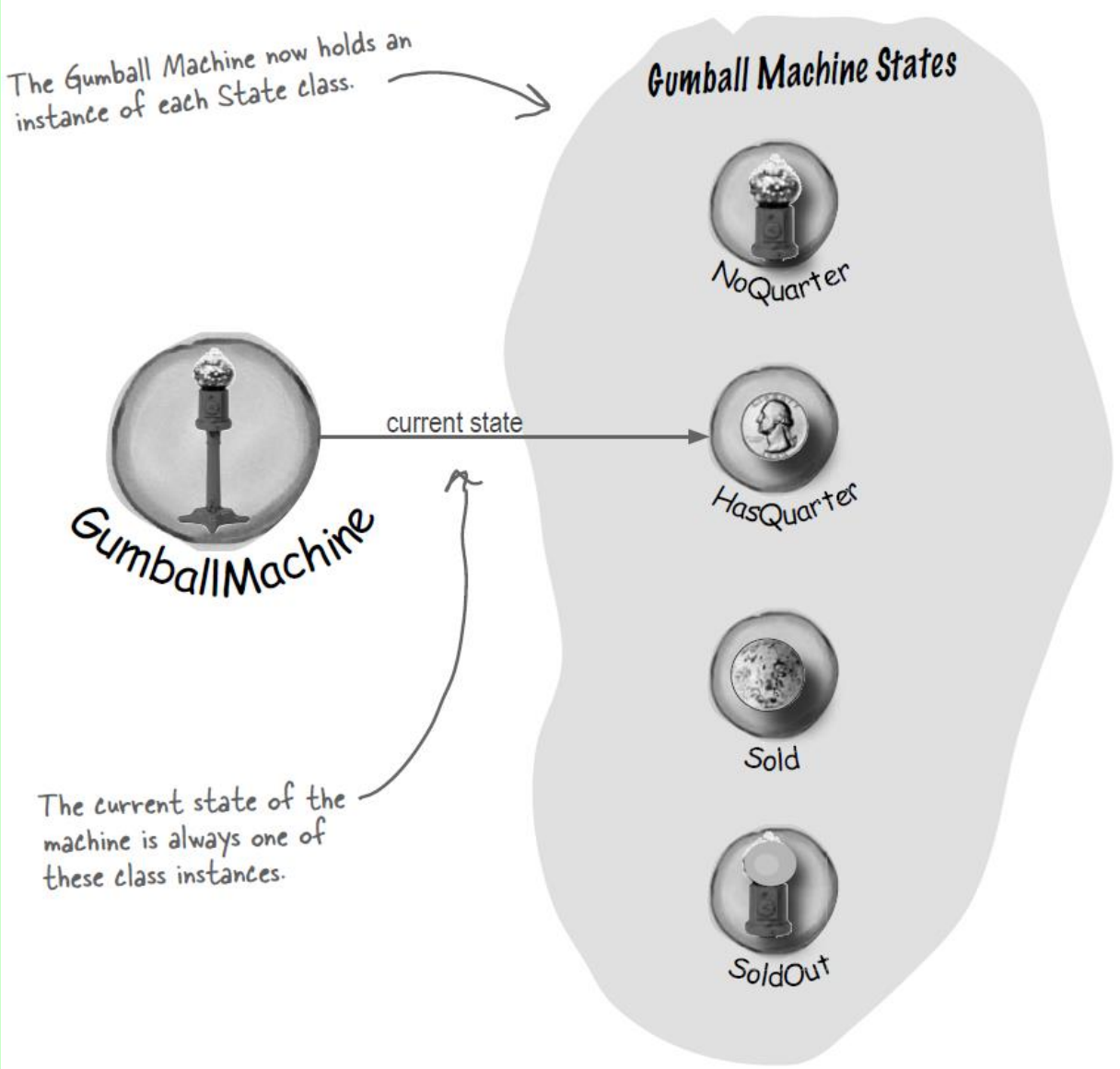
This method allows other objects (like our State objects) to transition the machine to a different state.

The machine supports a releaseBall() helper method that releases the ball and decrements the count instance variable.

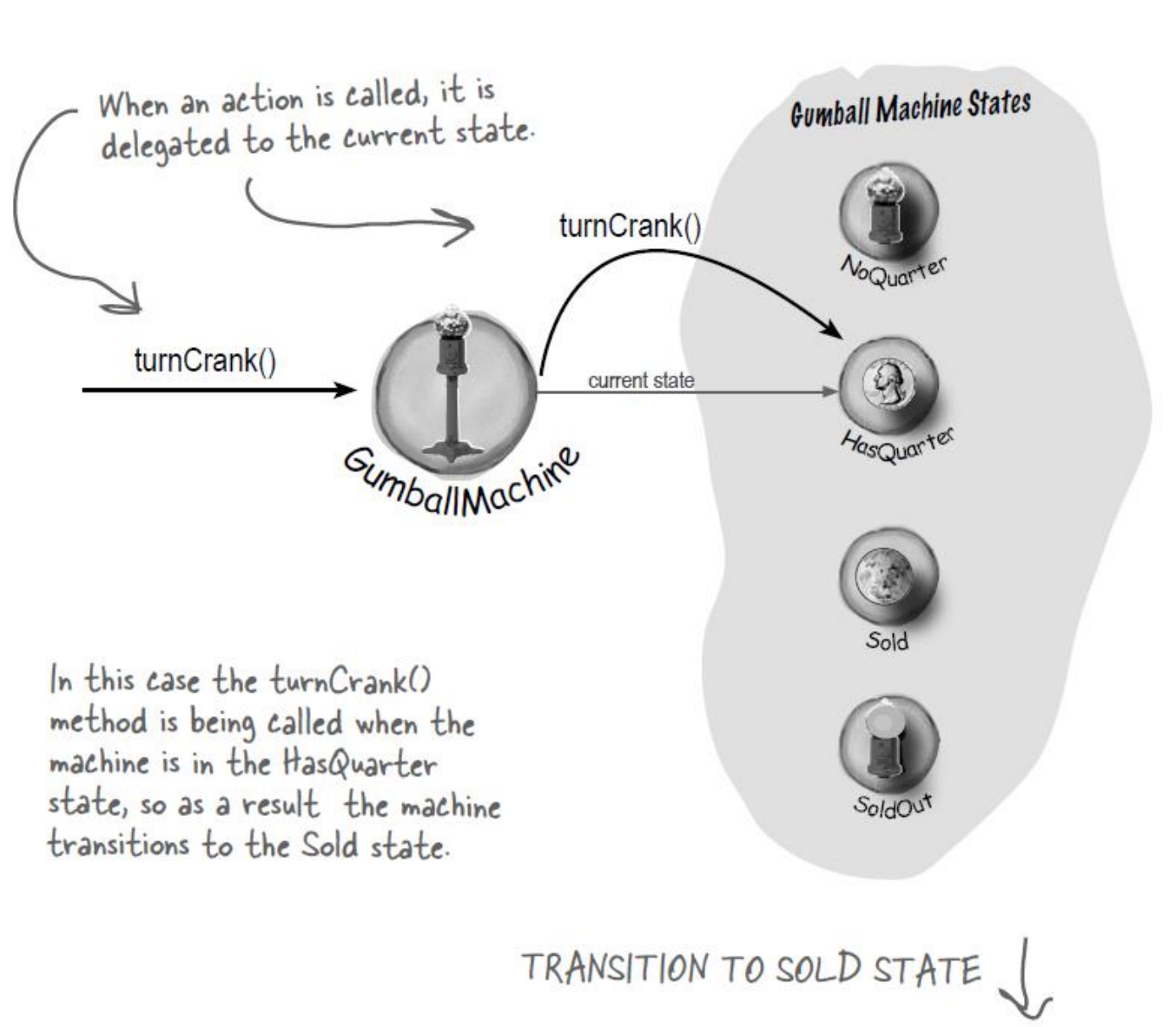
What we’ve done so far

- Achieve the same functionality with different structure
  - Localized the behavior of each state into its own class.
  - Removed all the troublesome if statements that would have been difficult to maintain.
  - Closed each state for modifi cation, and yet left the Gumball Machine open to extension by adding new state classes
  - Created a code base and class structure that maps much more closely to the Mighty Gumball diagram and is easier to read and understand.

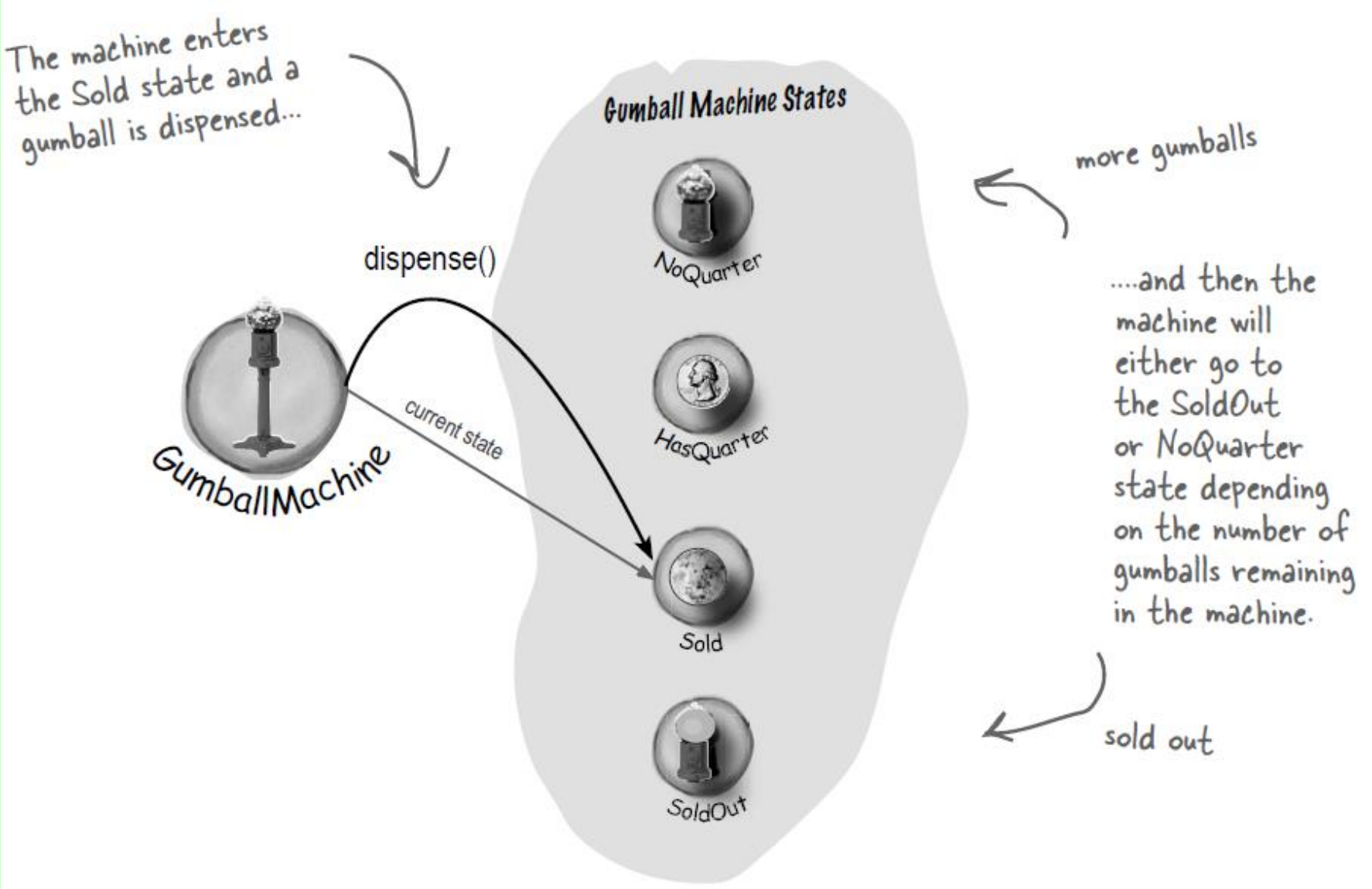
What we’ve done so far



When an Action is called



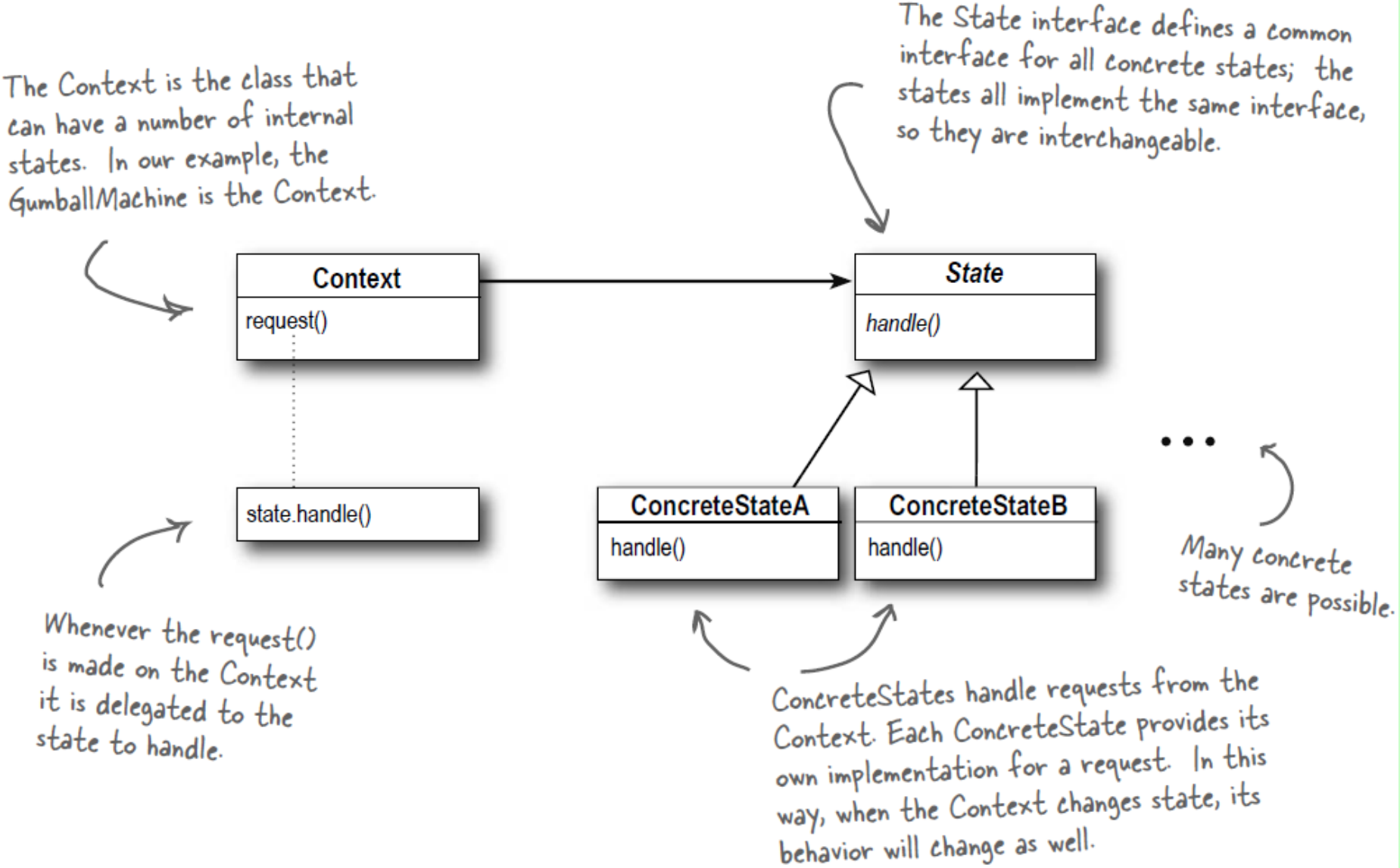
When an Action is called



The State Pattern defined

- The State Pattern allows an object to alter its behavior when its internal state changes. The object will appear to change its class.

Class Diagram



State Pattern vs Strategy Pattern

- With Strategy, the client usually specifies the strategy object that the context is composed with. Mostly used to set context objects with appropriate strategies.

- With the State Pattern, we have a set of behaviors encapsulated in state objects; at any time the context is delegating to one of those states.

Bonus Gumball 1 in 10 game

```
public class GumballMachine {

    State soldOutState;
    State noQuarterState;
    State hasQuarterState;
    State soldState;
    State winnerState;

    State state = soldOutState;
    int count = 0;

    // methods here
}
```

All you need to add here is the new WinnerState and initialize it in the constructor.

Don't forget you also have to add a getter method for WinnerState too.

WinnerState

```
public class WinnerState implements State {

    // instance variables and constructor

    // insertQuarter error message

    // ejectQuarter error message

    // turnCrank error message

    public void dispense() {
        System.out.println("YOU'RE A WINNER! You get two gumballs for your quarter");
        gumballMachine.releaseBall();
        if (gumballMachine.getCount() == 0) {
            gumballMachine.setState(gumballMachine.getSoldOutState());
        } else {
            gumballMachine.releaseBall();
            if (gumballMachine.getCount() > 0) {
                gumballMachine.setState(gumballMachine.getNoQuarterState());
            } else {
                System.out.println("Oops, out of gumballs!");
                gumballMachine.setState(gumballMachine.getSoldOutState());
            }
        }
    }
}
```

Just like SoldState.

Here we release two gumballs and then either go to the NoQuarterState or the SoldOutState.

As long as we have a second gumball we release it.

Setting the Winner

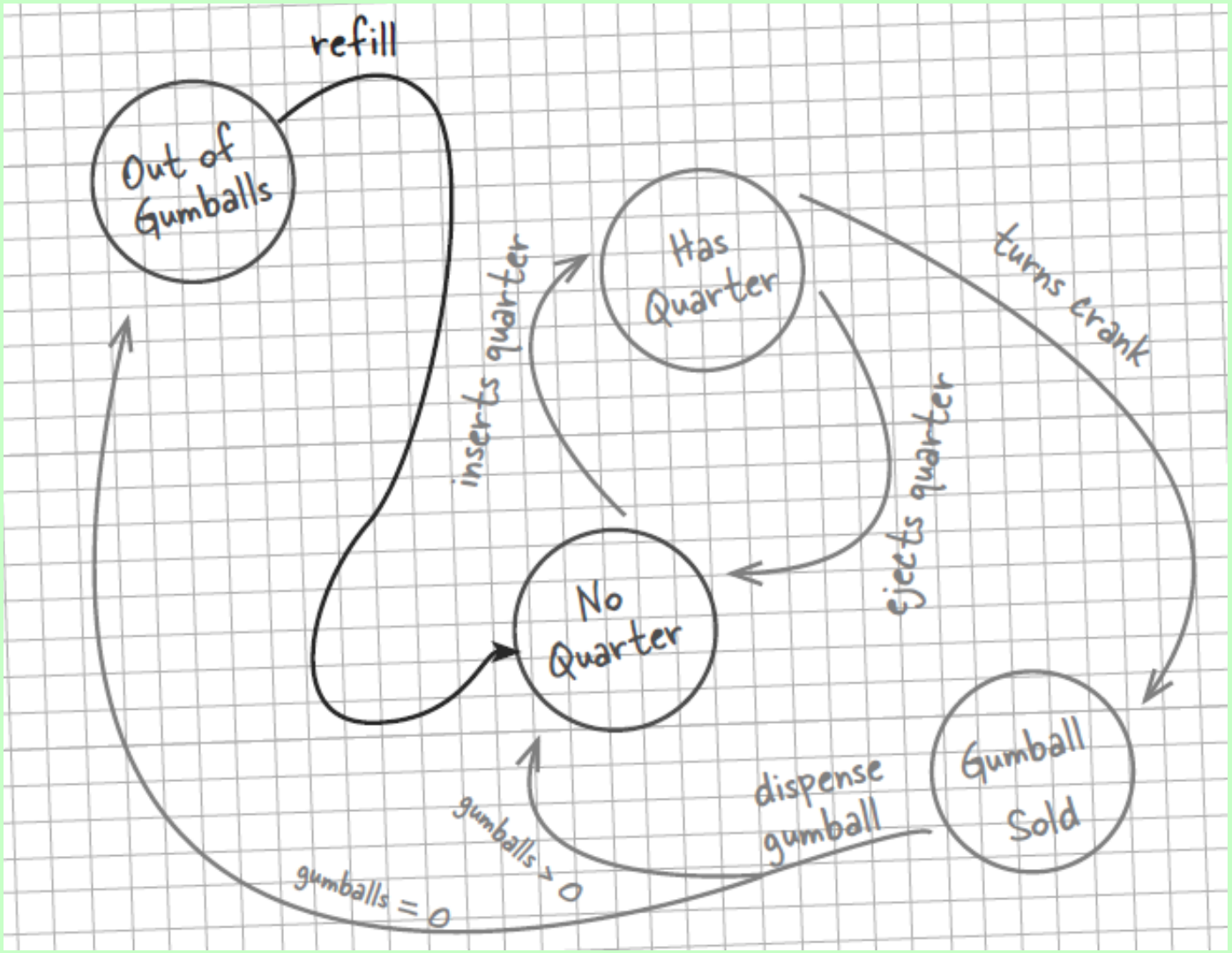
```
public class HasQuarterState implements State {
    Random randomWinner = new Random(System.currentTimeMillis());
    GumballMachine gumballMachine;

    public HasQuarterState(GumballMachine gumballMachine) {
        this.gumballMachine = gumballMachine;
    }
```

First we add a random number generator to generate the 10% chance of winning...

```
    public void turnCrank() {
        System.out.println("You turned...");
        int winner = randomWinner.nextInt(10);
        if ((winner == 0) && (gumballMachine.getCount() > 1)) {
            gumballMachine.setState(gumballMachine.getWinnerState());
        } else {
            gumballMachine.setState(gumballMachine.getSoldState());
        }
    }
    public void dispense() {
        System.out.println("No gumball dispensed");
    }
}
```

If they won, and there's enough gumballs left for them to get two, we go to the WinnerState; otherwise, we go to the SoldState (just like we always did).

Missing Action - refill



Missing Action - refill

```
void refill(int count) {
    this.count = count;
    state = noQuarterState;
}
```

References

- Design Patterns: Elements of Reusable Object-Oriented Software
  By Gamma, Erich; Richard Helm, Ralph Johnson, and John Vlissides (1995). Addison-Wesley. ISBN 0-201-63361-2.
- **Head First Design Patterns** By Eric Freeman, Elisabeth Freeman, Kathy Sierra, Bert Bates
  First Edition October 2004
  ISBN 10: 0-596-00712-4