COMP2511 State Pattern

Prepared by

Dr. Ashesh Mahidadia

State Pattern

These lecture notes are from the wikipedia page at: https://en.wikipedia.org/wiki/Finite-state_machine

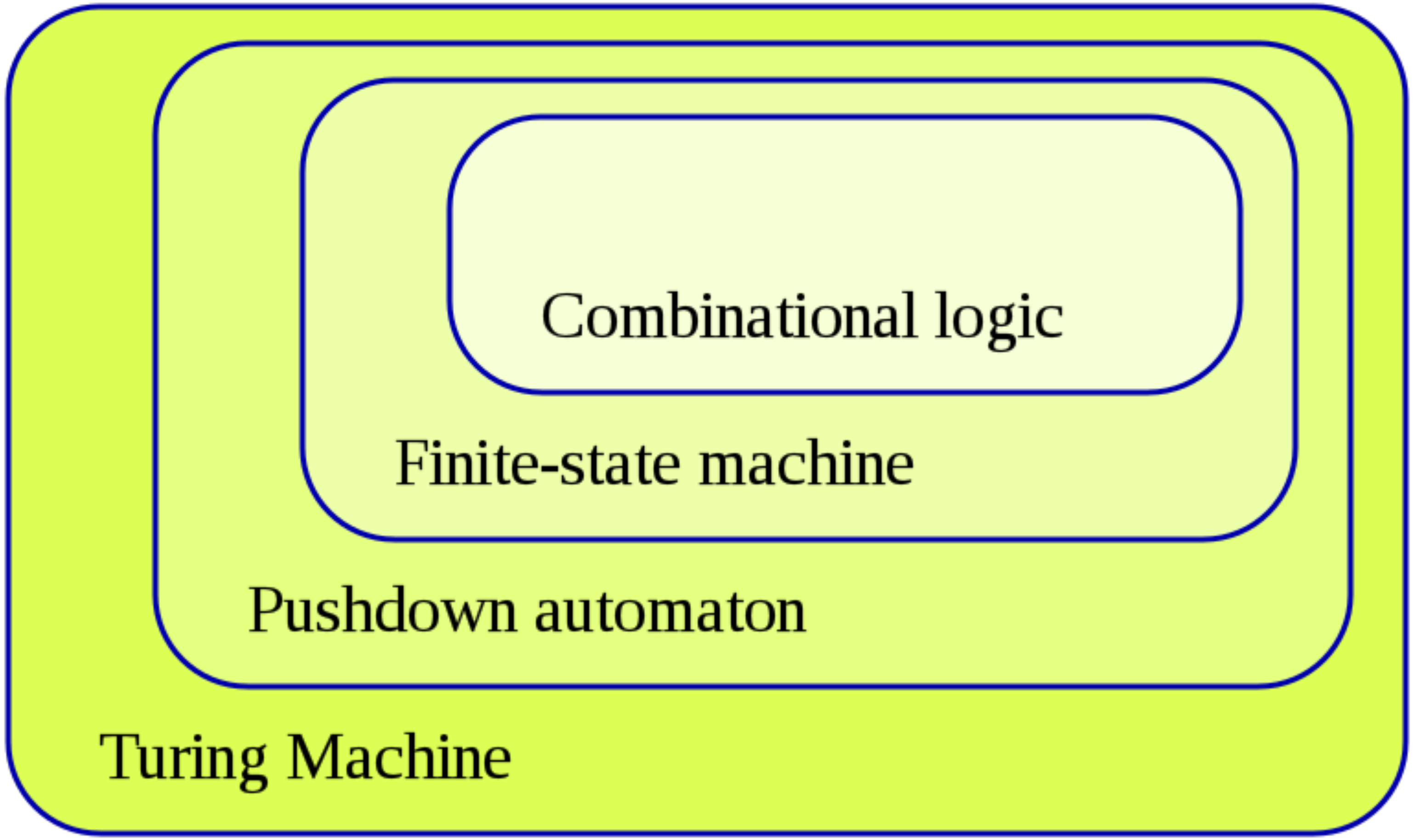
And

the reference book "Head First Design Patterns".

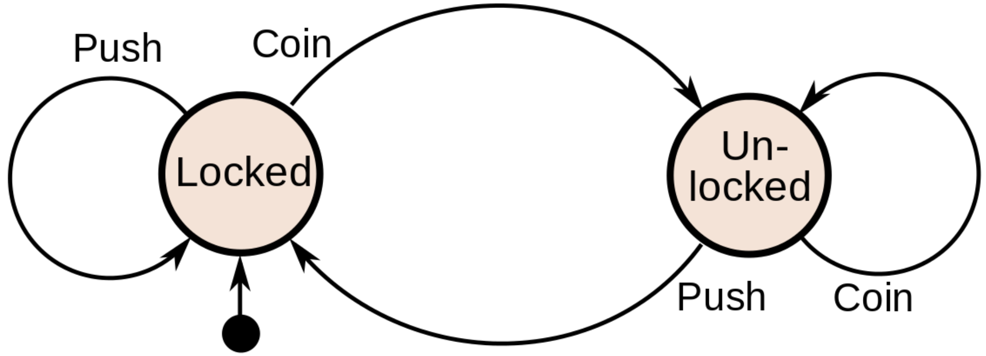
Finite-state Machine

- A **finite-state machine (FSM)**, is an abstract machine that can be in exactly **one** of a finite number of ***states*** at any given time.
  - the finite-state machine can change from one state to another in response to some external ***inputs***.
  - the change from one state to another is called a ***transition***.
- An finite-state machine is **defined** by
  - a list of its **states**
  - the conditions for each **transition**
  - its **initial** state
- Finite-state machine also refer to as finite-state automaton, finite automaton, or state machine

Automata theory



Example: coin-operated turnstile



Current State	Input	Next State	Output
Locked	coin	Unlocked	Unlocks the turnstile so that the customer can push through.
	push	Locked	None
Unlocked	coin	Unlocked	None
	push	Locked	When the customer has pushed through, locks the turnstile.

State Transition Table: shows for each possible state, the transitions between them (based upon the inputs given to the machine) and the outputs resulting from each input

State Machines: Simple examples

- **vending machines**, which dispense products when the proper combination of coins is deposited,
- **elevators**, whose sequence of stops is determined by the floors requested by riders,
- **traffic lights**, which change sequence when cars are waiting,
- **combination locks**, which require the input of combination numbers in the proper order.

State Machine: Terminology

- A ***state*** is a description of the status of a system that is waiting to execute a *transition*.
- A ***transition*** is a set of actions to be executed when a condition is fulfilled or when an event is received.
- **Identical stimuli trigger different actions depending on the current state.**
- For example,
  - when using an audio system to listen to the radio (the system is in the "radio" state), receiving a "**next**" stimulus results in moving to the next station.
  - when the system is in the "CD" state, the "**next**" stimulus results in moving to the next track.
- Often, the following are also associated with a state:
  - an ***entry action***: performed *when entering* the state, and
  - an ***exit action***: performed *when exiting* the state.

Representations

- The most common representation is shown below:

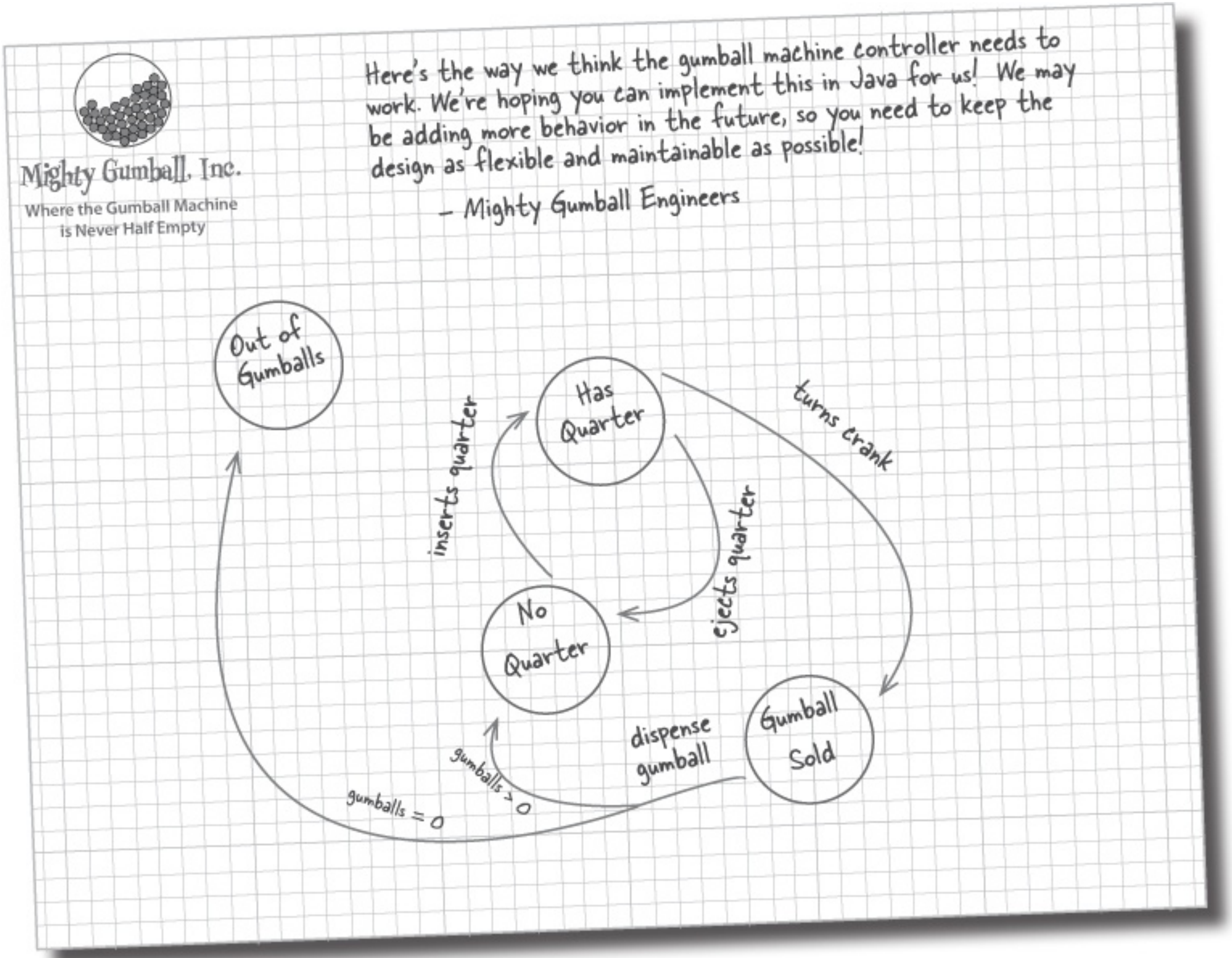
State transition table

Current state / Input	State A	State B	State C
Input X	…	…	…
Input Y	…	State C	…
Input Z	…	…	…

State Machines for UI

- Examples …

Gumball Machine!



From the reference book "head First Design Patterns"

State machines 101

How are we going to get from that state diagram to actual code? Here's a quick introduction to implementing state machines:

- ① First, gather up your states:

- 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.

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

inserts quarter

turns crank

ejects quarter

dispense

These actions are the gumball machine's interface – the things you can do with it.

Looking at the diagram, invoking any of these actions causes a state transition.

Dispense is more of an internal action the machine invokes on itself.

Read the example code provided for this week

From the reference book "head First Design Patterns"

- 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. For instance, for the insert quarter action, we might write a method like this:

```
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");

    }
}
```

Each possible state is checked with a conditional statement...

...and exhibits the appropriate behavior for each possible state...

...but can also transition to other states, just as depicted in the diagram.

Read the example code provided for this week

From the reference book "head First Design Patterns"

With that quick review, let's go implement the Gumball Machine!

```java
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 has the gumball!

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

The customer tries to turn the crank...

```java
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.

```java
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");
    }
}

// other methods here like toString() and refill()
}
```

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.

Read the example code provided for this week

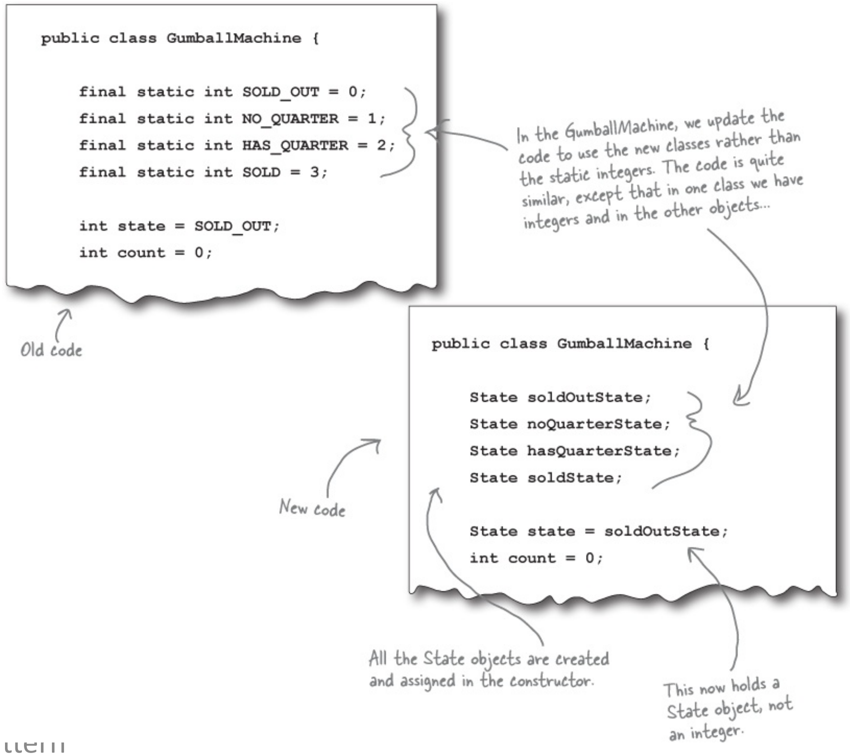
From the reference book “head First Design Patterns”

The new design

It looks like we've got a new plan: instead of maintaining our existing code, we're going to rework it to encapsulate state objects in their own classes and then delegate to the current state when an action occurs.

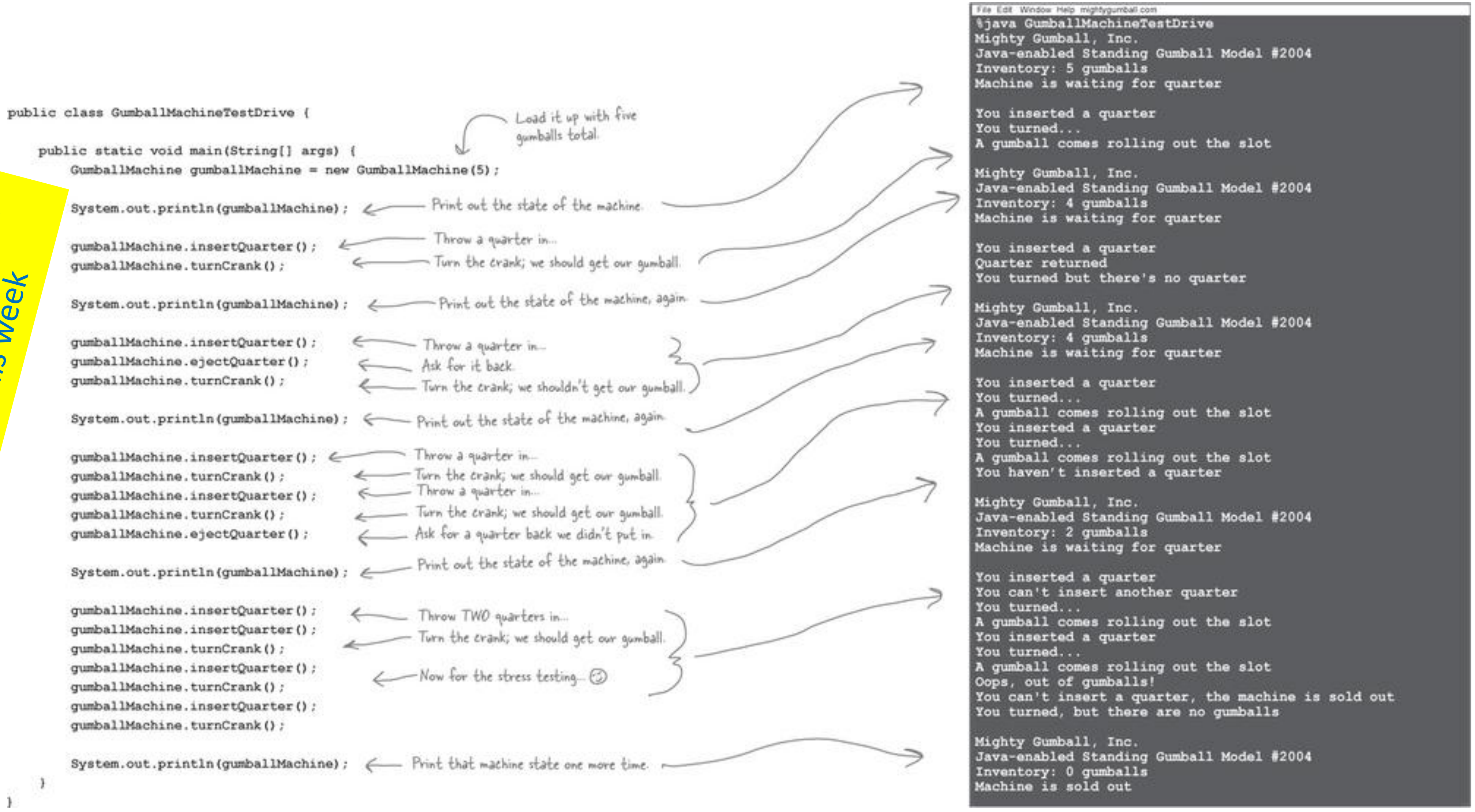
We're following our design principles here, so we should end up with a design that is easier to maintain down the road. Here's how we're going to do it:

- 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.**



From the reference book
"head First Design Patterns"

Read the example code provided for this week


```java
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();

        System.out.println(gumballMachine);
    }
}
```



```
%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
```

From the reference book
“head First Design Patterns”

Demo …

- Demo of Gumball, from the reference book “Head First Design Patterns”.

BULLET POINTS

- The State Pattern allows an object to have many different behaviors that are based on its internal state.
- Unlike a procedural state machine, the State Pattern represents state as a full-blown class.
- The Context gets its behavior by delegating to the current state object it is composed with.
- By encapsulating each state into a class, we localize any changes that will need to be made.
- The State and Strategy Patterns have the same class diagram, but they differ in intent.
- Strategy Pattern typically configures Context classes with a behavior or algorithm.
- State Pattern allows a Context to change its behavior as the state of the Context changes.
- State transitions can be controlled by the State classes or by the Context classes.
- Using the State Pattern will typically result in a greater number of classes in your design.
- State classes may be shared among Context instances.

From the reference book
“head First Design Patterns”