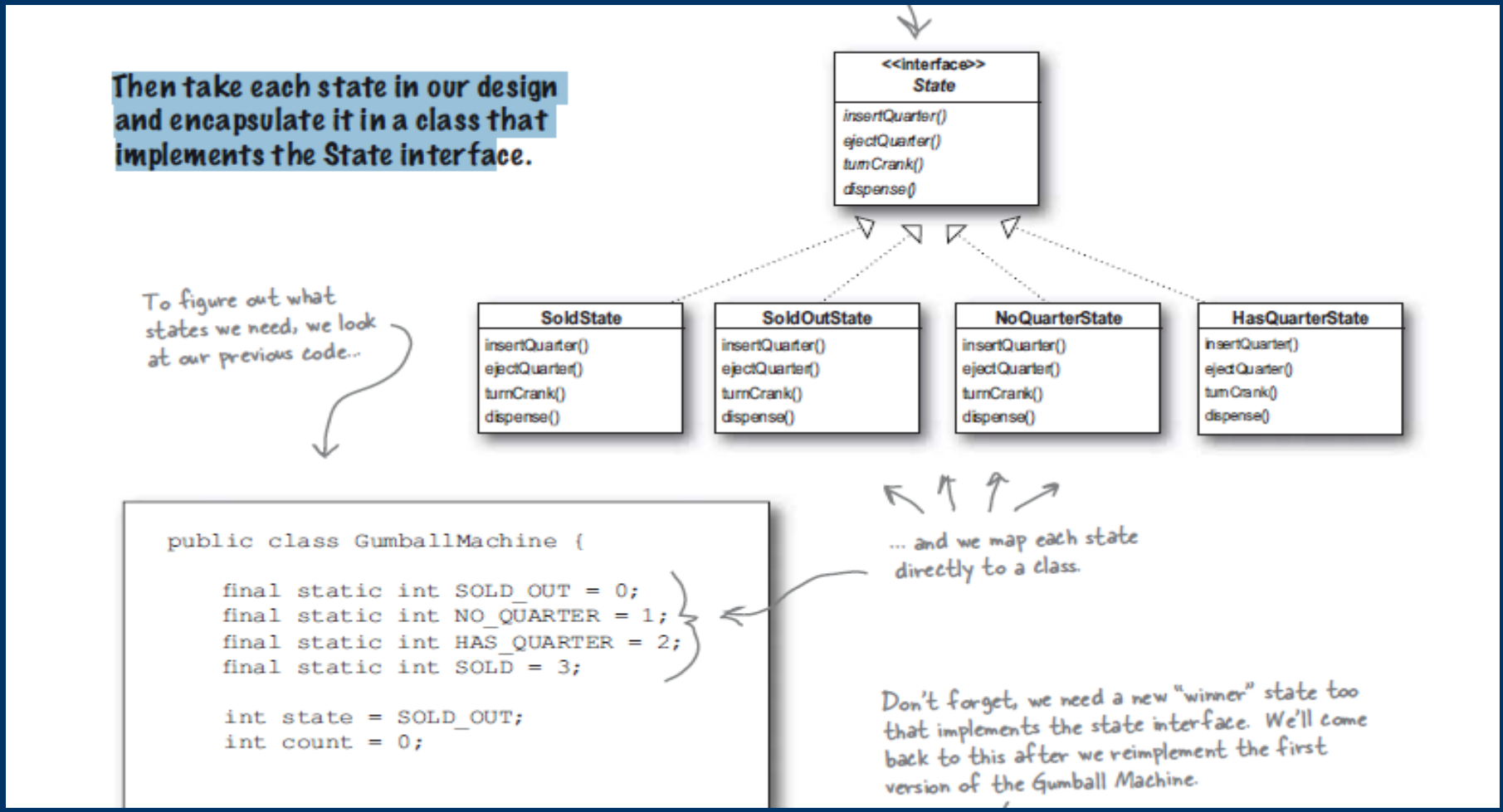
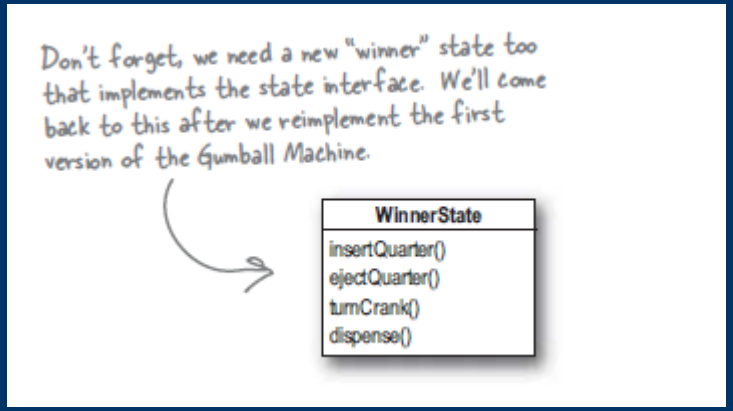
State Pattern(exersice)



State Pattern(exersice)



State Pattern(exersice)

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.

State Pattern

```
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;
        }
    }

    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...");
```

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.

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.

State Pattern

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

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

State Pattern

```
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.

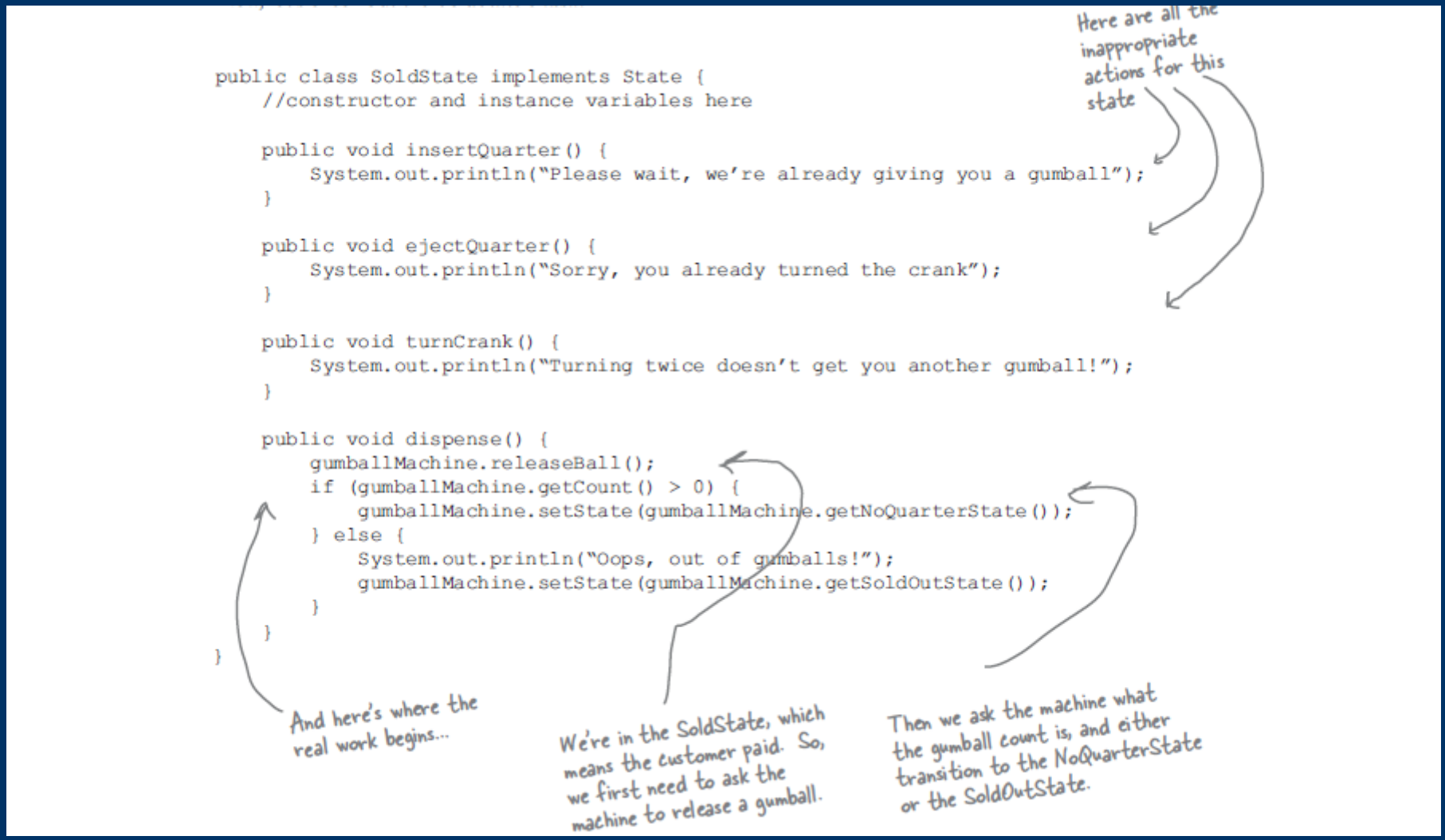
An inappropriate action for this state.

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.

State Pattern



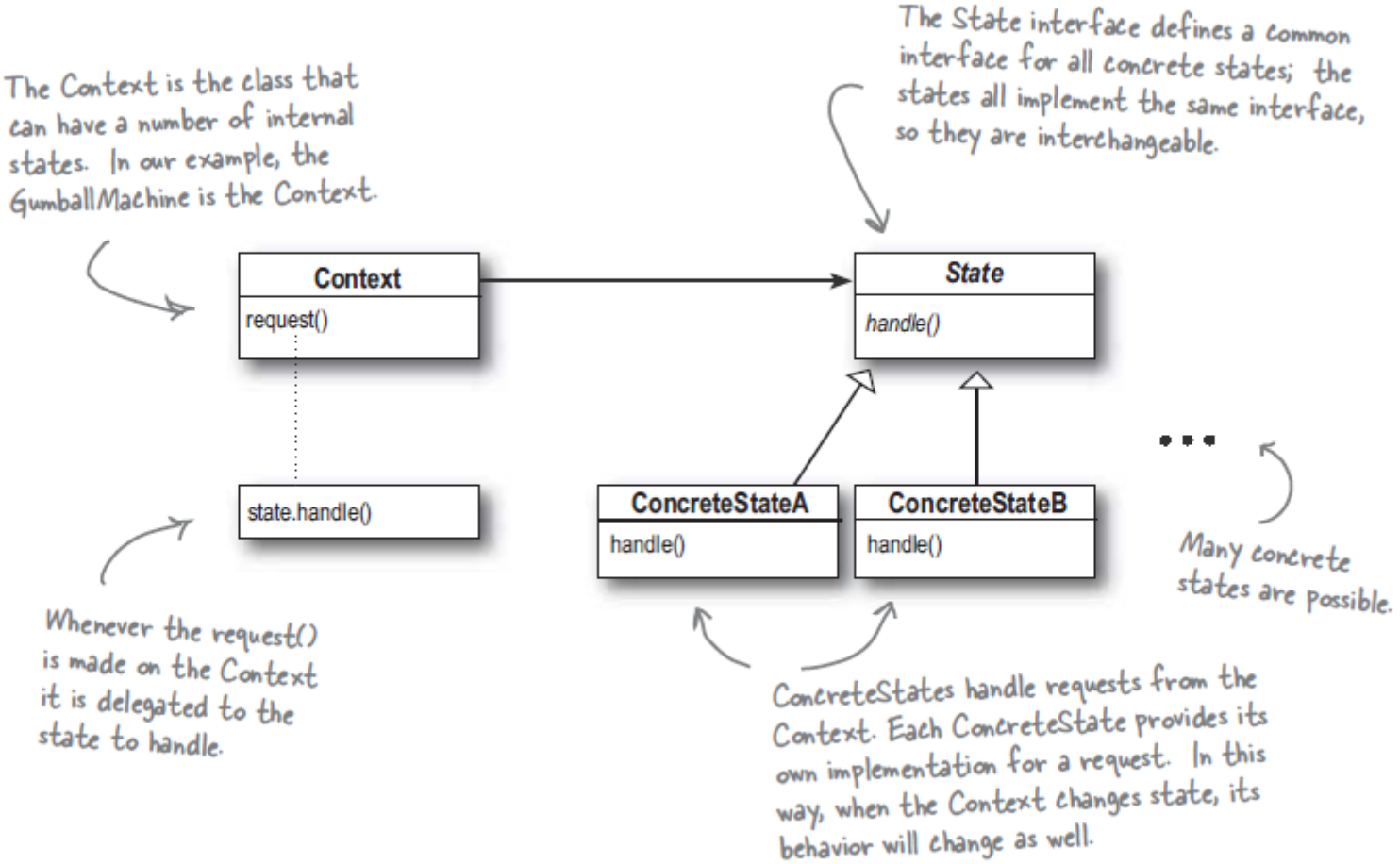
State Pattern

- What we have done ?
- Localized the behavior of each state into its own class.
- Removed all throublesome is statements that would have been difficult to maintain.
- Closed each state for modification, and yet left the GumbalMachine open to extension by adding new states classes.

State Pattern(definition)

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

State Pattern(diagram)



State Pattern(winner example)

```
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.

State Pattern(winner example)

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

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

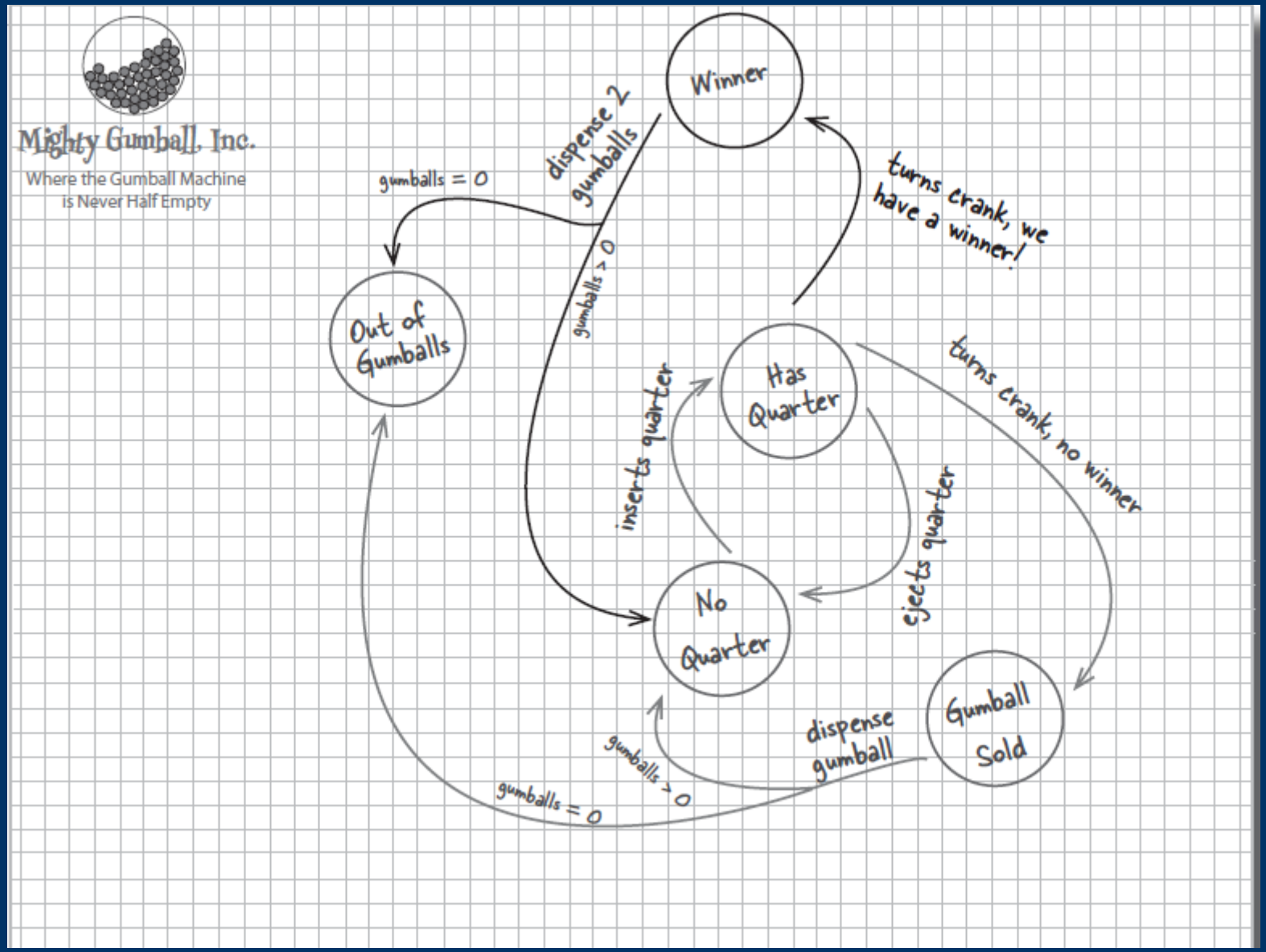
...then we determine if this customer won.

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).

State Pattern(conclusion)

- State patter allow an object to have different behavior that are based on its internal states.
- By encapsulate each state into a class, we localize any change , that we need to be made.
- The State and Strategy pattern has the same class diagram but differ in intent.
- State transition can be controlled by the state classes or by the Context classes.
- Using State pattern will typically result in a greater number of class in your design.

State Pattern(example)



Proxy Pattern

- Control and manage acces that is what proxies pattern do. As you are going to see, there are lot of ways in which proxies stand in for the object they proxy. Proxy has been known to houl entire method call over the internet for their proxied objects.

Proxy Pattern(example definition)

- We need a change in our Gumball Machine, we need to add monitoring. Get a report of inventory and machine states add a location as well.

Proxy Pattern(code example)

```
public class GumballMachine {
    // other instance variables
    String location;

    public GumballMachine(String location, int count) {
        // other constructor code here
        this.location = location;
    }

    public String getLocation() {
        return location;
    }

    // other methods here
}
```

A location is just a String.

The location is passed into the constructor and stored in the instance variable.

Let's also add a getter method to grab the location when we need it.

Proxy Pattern(code example)

```
public class GumballMonitor {
    GumballMachine machine;

    public GumballMonitor(GumballMachine machine) {
        this.machine = machine;
    }

    public void report() {
        System.out.println("Gumball Machine: " + machine.getLocation());
        System.out.println("Current inventory: " + machine.getCount() + " gumballs");
        System.out.println("Current state: " + machine.getState());
    }
}
```

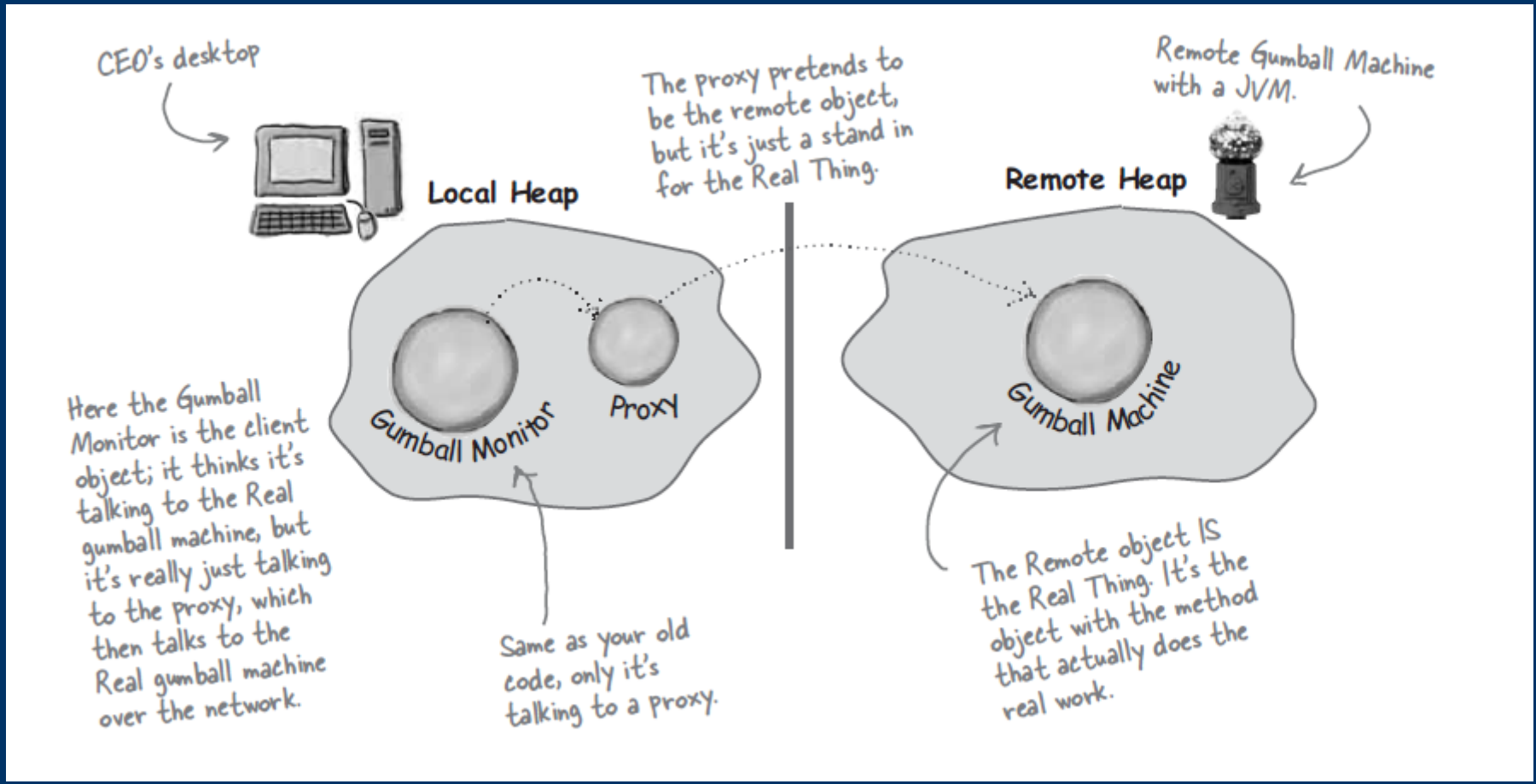
The monitor takes the machine in its constructor and assigns it to the machine instance variable.

Our report method just prints a report with location, inventory and the machine's state.

Proxy Pattern(example)

- New requirement is give access to monitor gumball machines REMOTELY.

Proxy Pattern(example)

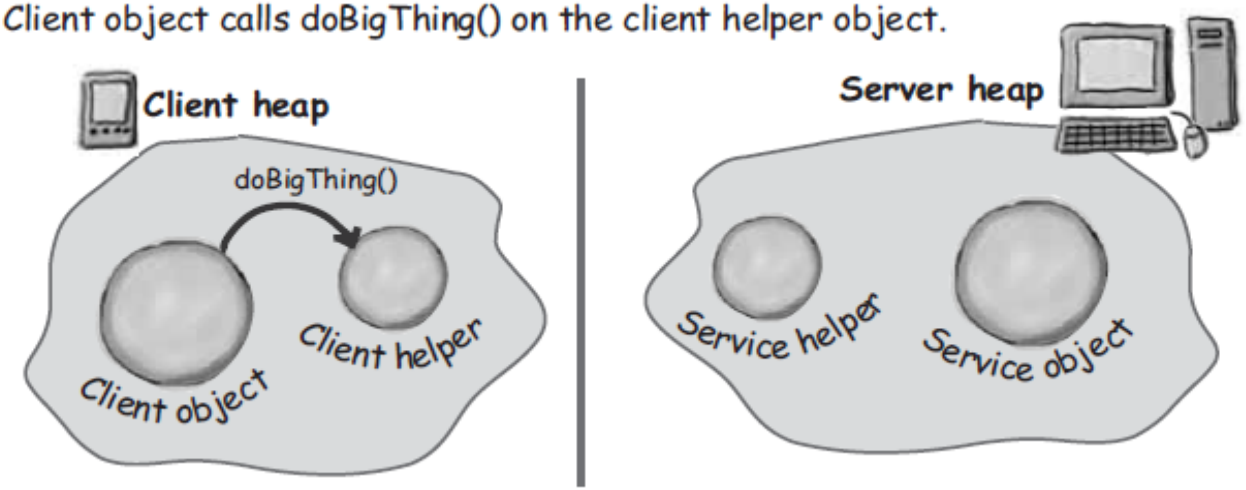


Proxy Pattern(example)

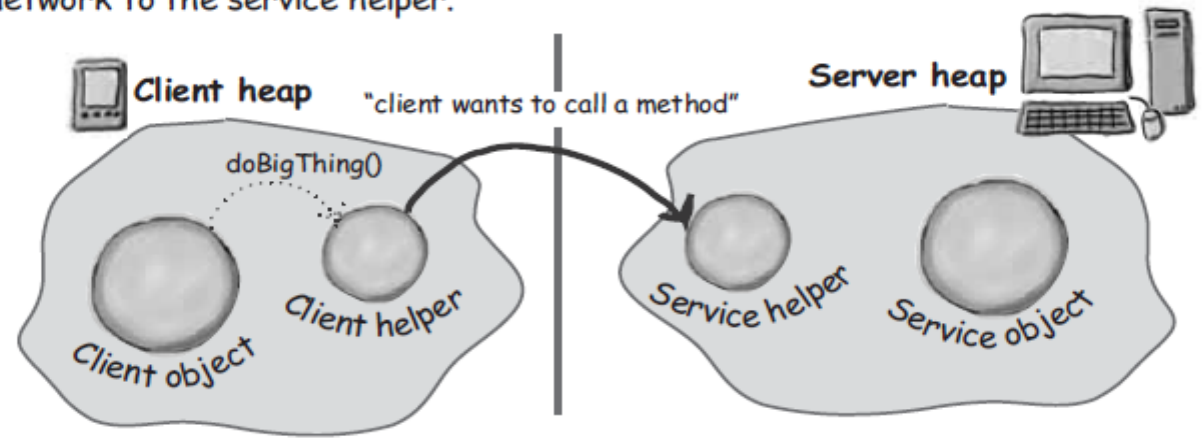
- How do we create a proxy that know how to invoke a method on an object that lives in another JVM ?

Proxy Pattern(RMI)

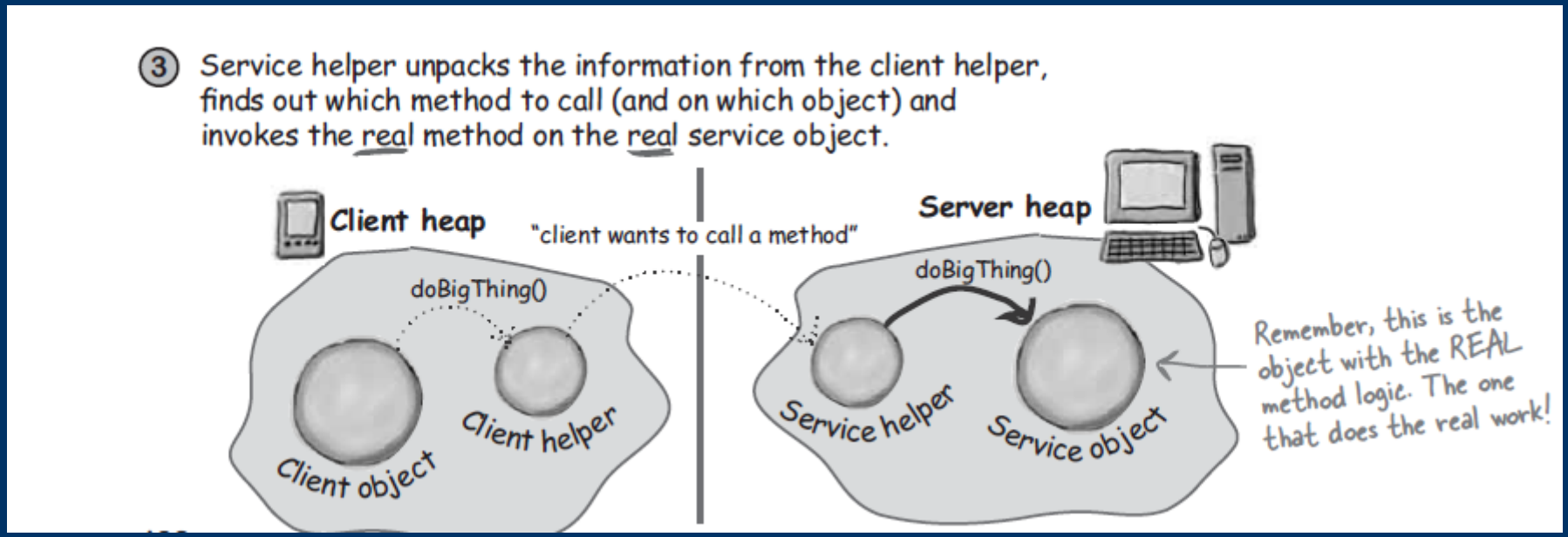
1 Client object calls doBigThing() on the client helper object.

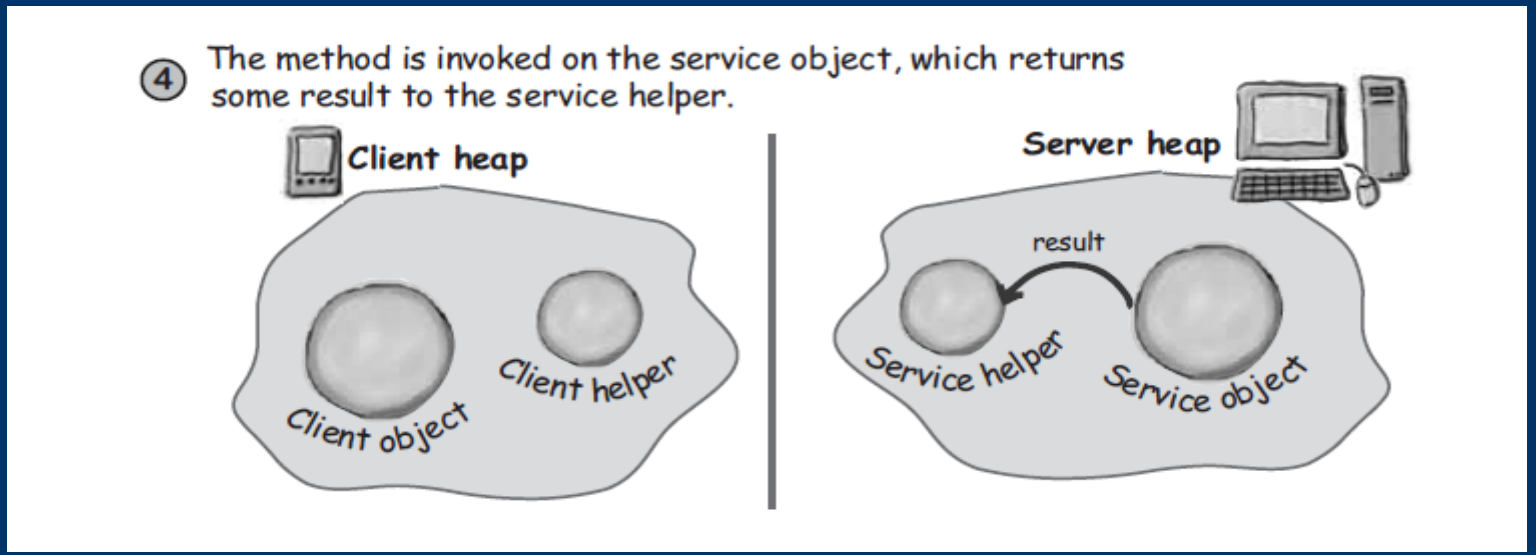


2 Client helper packages up information about the call (arguments, method name, etc.) and ships it over the network to the service helper.



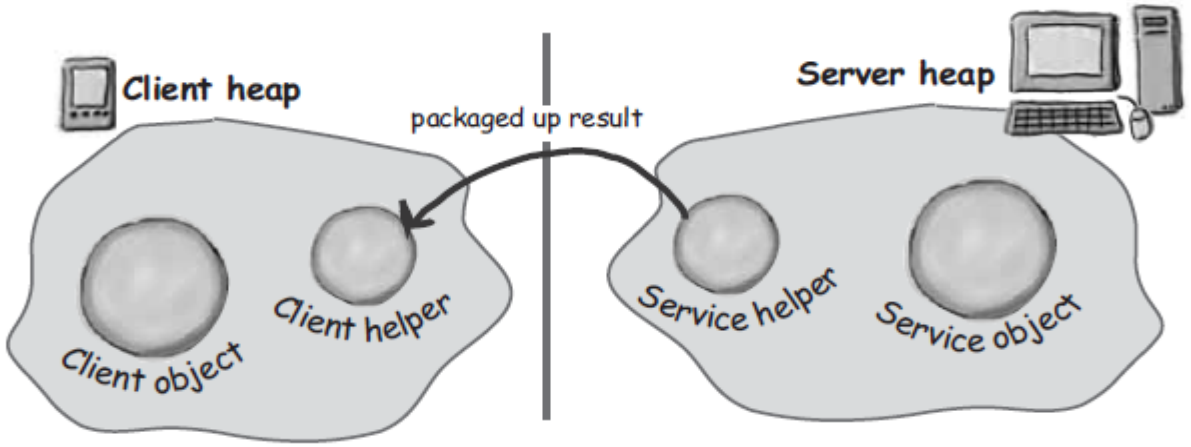
Proxy Pattern(RMI)



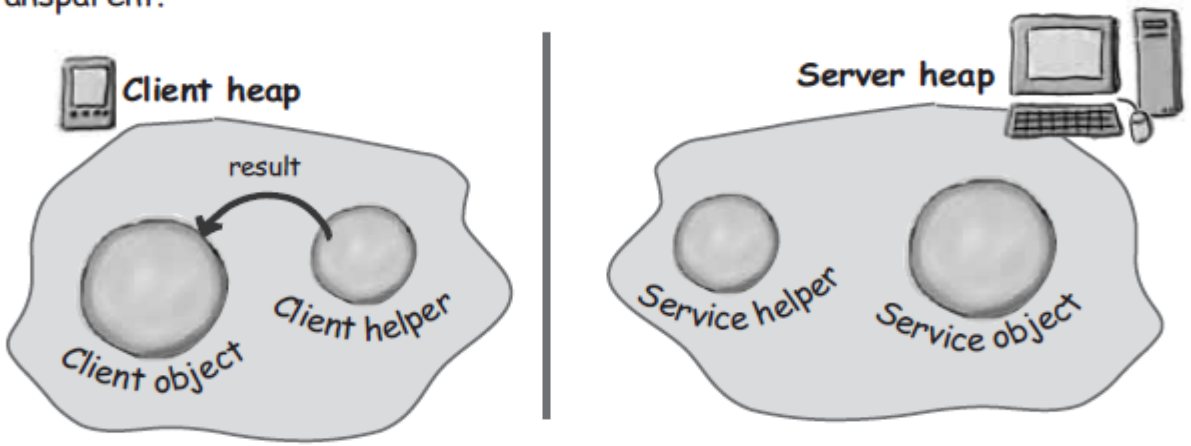


Proxy Pattern(RMI)

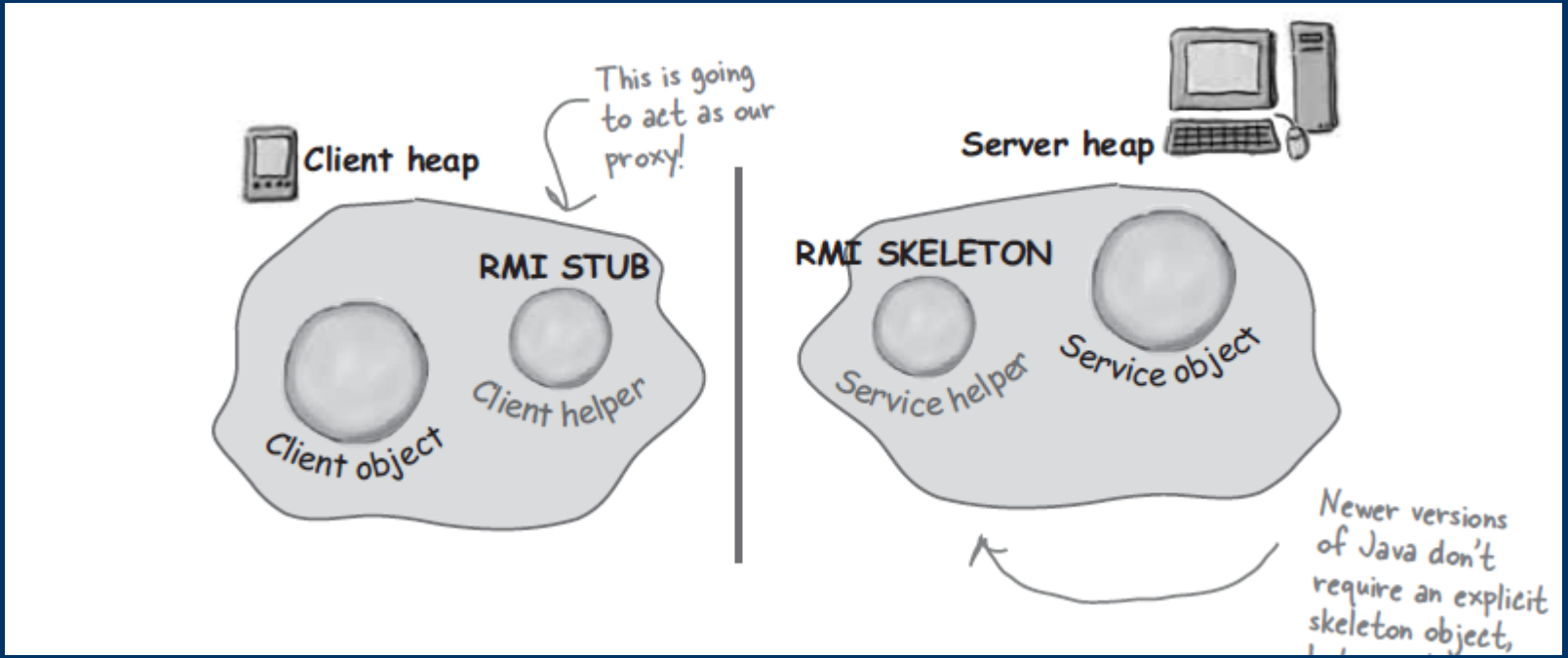
⑤ Service helper packages up information returned from the call and ships it back over the network to the client helper.



⑥ Client helper unpackages the returned values and returns them to the client object. To the client object, this was all transparent.



Proxy Pattern(RMI)

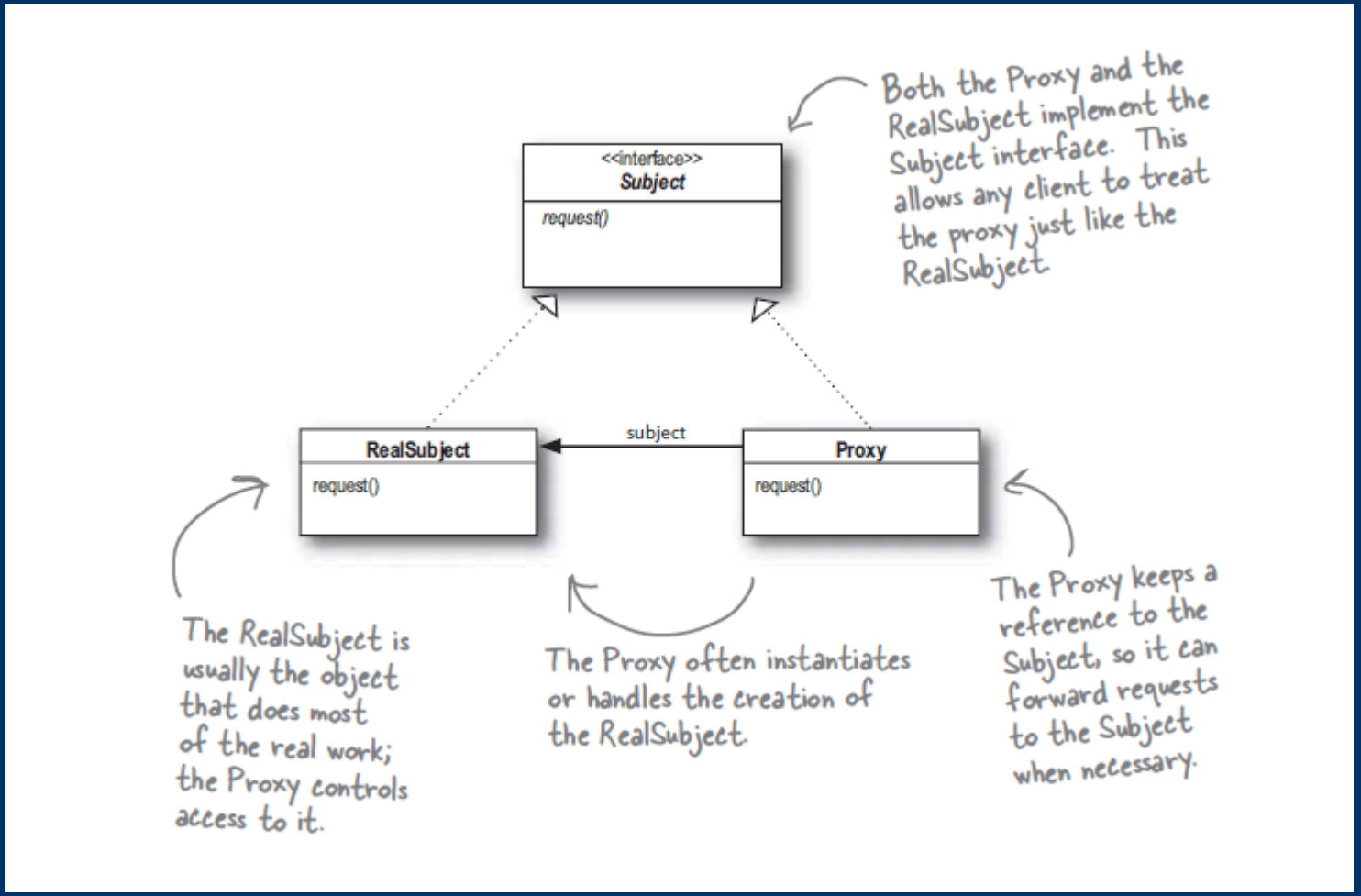


- In RMI the client HELPER is a stub and the service is a SKELETON.
- Optional – do the exersice using RMI

Proxy Pattern(definition)

- **Provides a surrogate or a placeholder for another object to control access to it.**
- Control access to an object which may be remote, expensive to create or in need of securing.
- In some case may be responsable of creating and destroying the RealSubject.
- Client interact with the RealSubject through the Proxy.

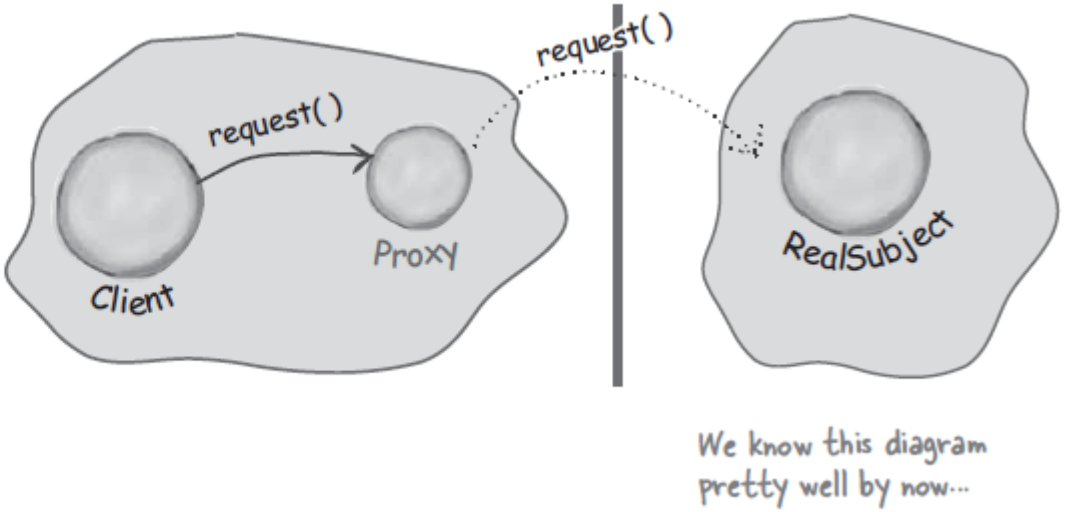
Proxy Pattern(diagram)



Proxy Pattern

Remote Proxy

With Remote Proxy, the proxy acts as a local representative for an object that lives in a different JVM. A method call on the proxy results in the call being transferred over the wire, invoked remotely, and the result being returned back to the proxy and then to the Client.

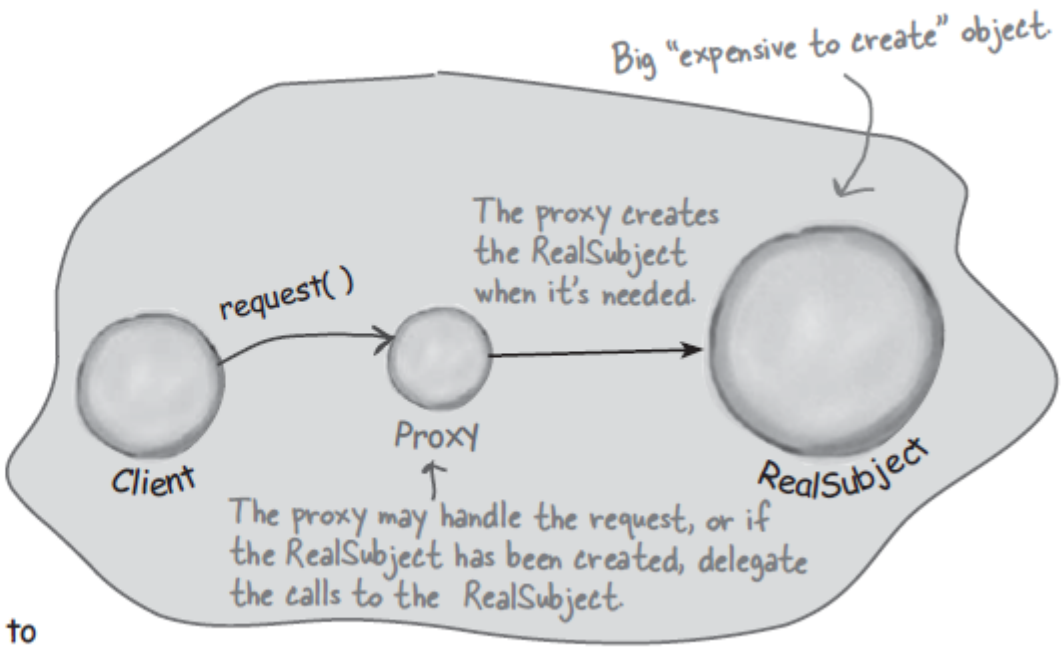


We know this diagram pretty well by now...

Proxy Pattern

Virtual Proxy

Virtual Proxy acts as a representative for an object that may be expensive to create. The Virtual Proxy often defers the creation of the object until it is needed; the Virtual Proxy also acts as a surrogate for the object before and while it is being created. After that, the proxy delegates requests directly to the RealSubject.



Proxy Pattern(variations)

- Firewall Proxy: Control access to a set of networks resources, protecting the subject from bad clients
- Smart Reference Proxy: Provides additional actions whenever a subject is referenced, just as counting the number of references to an object.
- Caching Proxy: Provides temporary storage for results of operations that are expensive. It can also allow multiple clients to share the results to reduce computation or network latency.
- Syncronization Proxy: Provide safe access to a subject from multiple threads.

Proxy Pattern(variations)

- Complexity hiding Proxy: Hides the complexity of and controls access to a complex set of classes. Sometimes called the Facade Proxy.
- Copy-on-write Proxy: controls the copying of an object by deferring the copy of an object until it is required by a client.

Proxy Pattern(conclusion)

- Provide a representative for an other object in other to control the client 's access to it.
- The Remote Proxy manages interaction between a client and a remote object.
- Virtual proxy control access to an object that is expensive to instantiate.
- Protection Proxy control access to the methods of an object based on the caller.
- Proxy is structurally similar to Decorator, but the two differ in theirs purpose.

Proxy Pattern(conclusion)

- Decorator Patterns adds behavior to an object, while a proxy control access.
- Like any wrapper, proxies will increase the number of classes and objects in your design.

Compound Pattern

- Some of the most powerful OO designs use several patterns together. The more you uses patterns the more you are going to see them showing up together in your design. We have a special name for a set of pattern that work together in a design that can be applied over many problems: a compound pattern.

Compound Pattern

- We are going to revisiting our friendly ducks from the duck simulator. The ducks will help us to understand how pattern can work together in the same solution. But remember that a real compound pattern should be apply to many problem.
- Compound pattern combines two or more patterns into a solution that solve a recurring or general problem.

Compound Pattern(exersice)

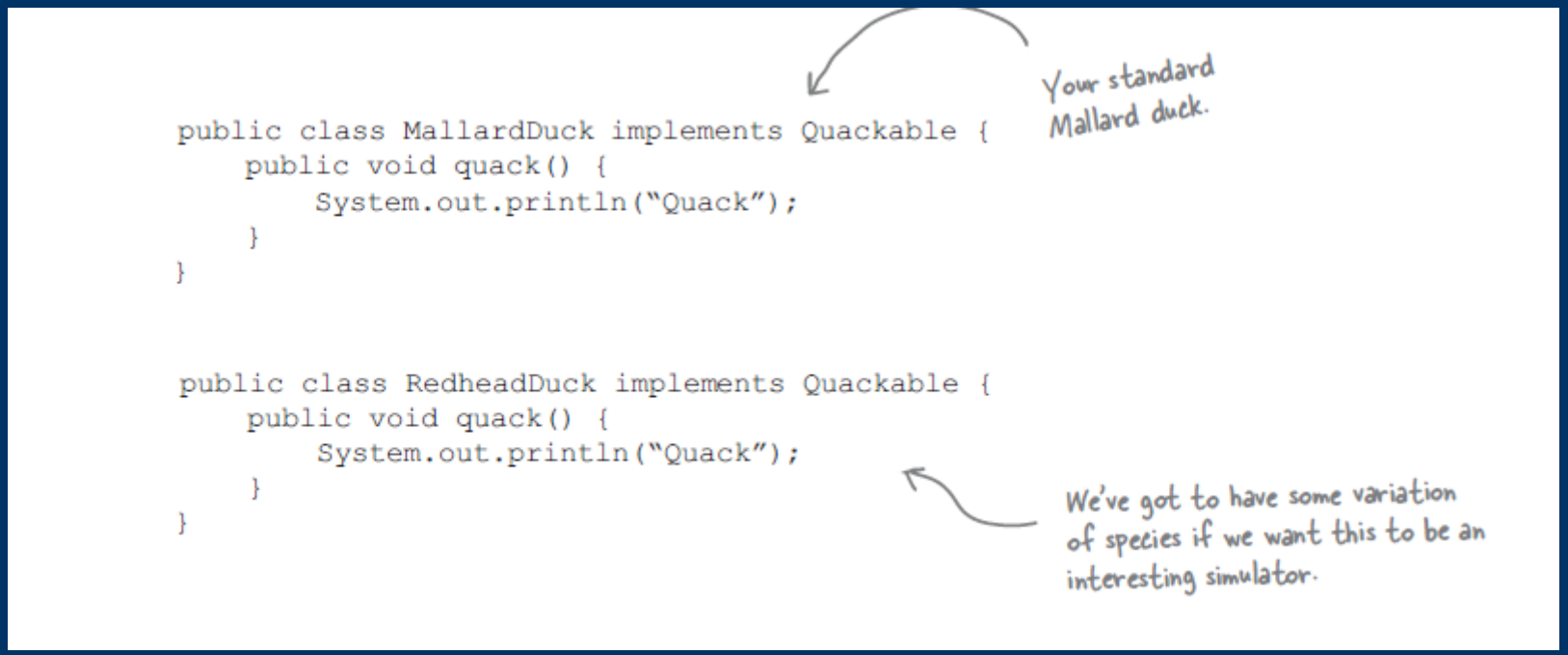
- Create a Quackable interface

  public interface Quackable {

    public void quack();

  }

- Now some duck that implement Quackable.



Compound Pattern(exersice)

- Some ducks.

```
public class DuckCall implements Quackable {
    public void quack() {
        System.out.println("Kwak");
    }
}
```

A DuckCall that quacks but doesn't sound quite like the real thing.

```
public class RubberDuck implements Quackable {
    public void quack() {
        System.out.println("Squeak");
    }
}
```

A RubberDuck that makes a squeak when it quacks.

Compound Pattern(exersice)

- Create a simulator

```
public class DuckSimulator {
    public static void main(String[] args) {
        DuckSimulator simulator = new DuckSimulator();
        simulator.simulate();
    }

    void simulate() {
        Quackable mallardDuck = new MallardDuck();
        Quackable redheadDuck = new RedheadDuck();
        Quackable duckCall = new DuckCall();
        Quackable rubberDuck = new RubberDuck();

        System.out.println("\nDuck Simulator");

        simulate(mallardDuck);
        simulate(redheadDuck);
        simulate(duckCall);
        simulate(rubberDuck);
    }

    void simulate(Quackable duck) {
        duck.quack();
    }
}
```

Here's our main method to get everything going.

We create a simulator and then call its simulate() method.

We need some ducks, so here we create one of each Quackable...

... then we simulate each one.

Here we overload the simulate method to simulate just one duck.

Here we let polymorphism do its magic: no matter what kind of Quackable gets passed in, the simulate() method asks it to quack.

Compound Pattern(exersice)

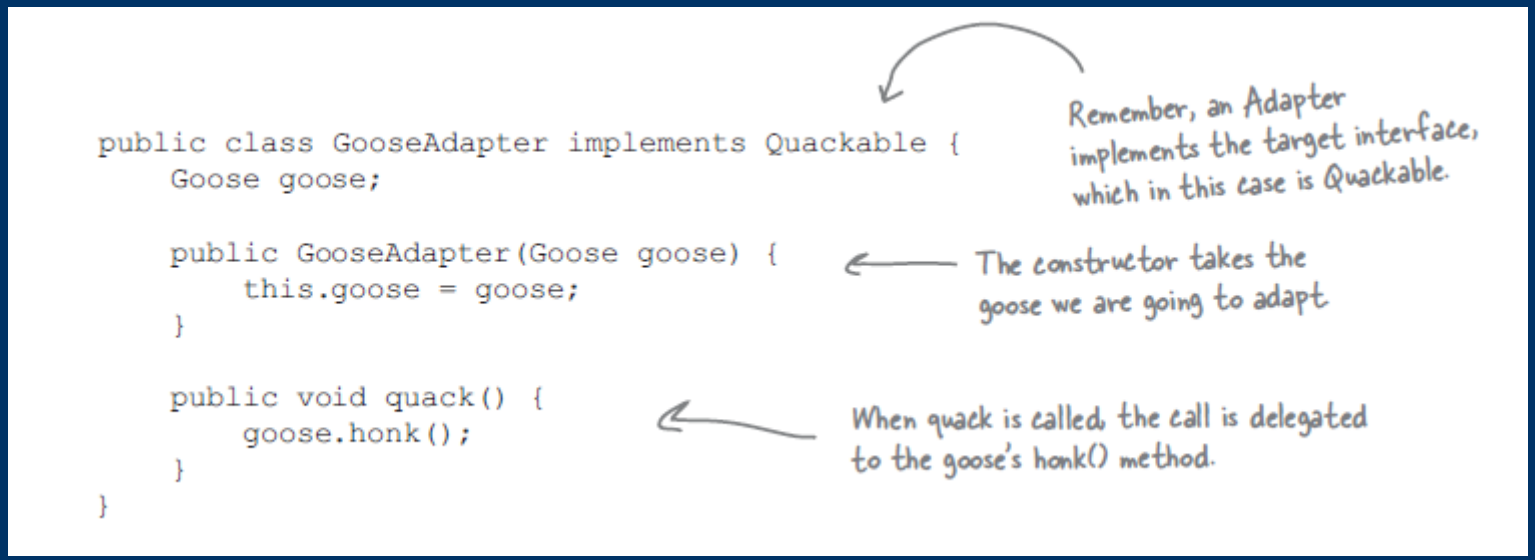
- We want to include Geese in the simulator and use it anywhere we would want to use a Duck. How would you do that ?

```
public class Goose {
    public void honk() {
        System.out.println("Honk");
    }
}
```

A Goose is a honker, not a quacker.

Compound Pattern(exersice)

- We need a goose adapter



Compound Pattern(exersice)

- Updating our Simulator.

```
public class DuckSimulator {
    public static void main(String[] args) {
        DuckSimulator simulator = new DuckSimulator();
        simulator.simulate();
    }
    void simulate() {
        Quackable mallardDuck = new MallardDuck();
        Quackable redheadDuck = new RedheadDuck();
        Quackable duckCall = new DuckCall();
        Quackable rubberDuck = new RubberDuck();
        Quackable gooseDuck = new GooseAdapter(new Goose());

        System.out.println("\nDuck Simulator: With Goose Adapter");

        simulate(mallardDuck);
        simulate(redheadDuck);
        simulate(duckCall);
        simulate(rubberDuck);
        simulate(gooseDuck);
    }

    void simulate(Quackable duck) {
        duck.quack();
    }
}
```

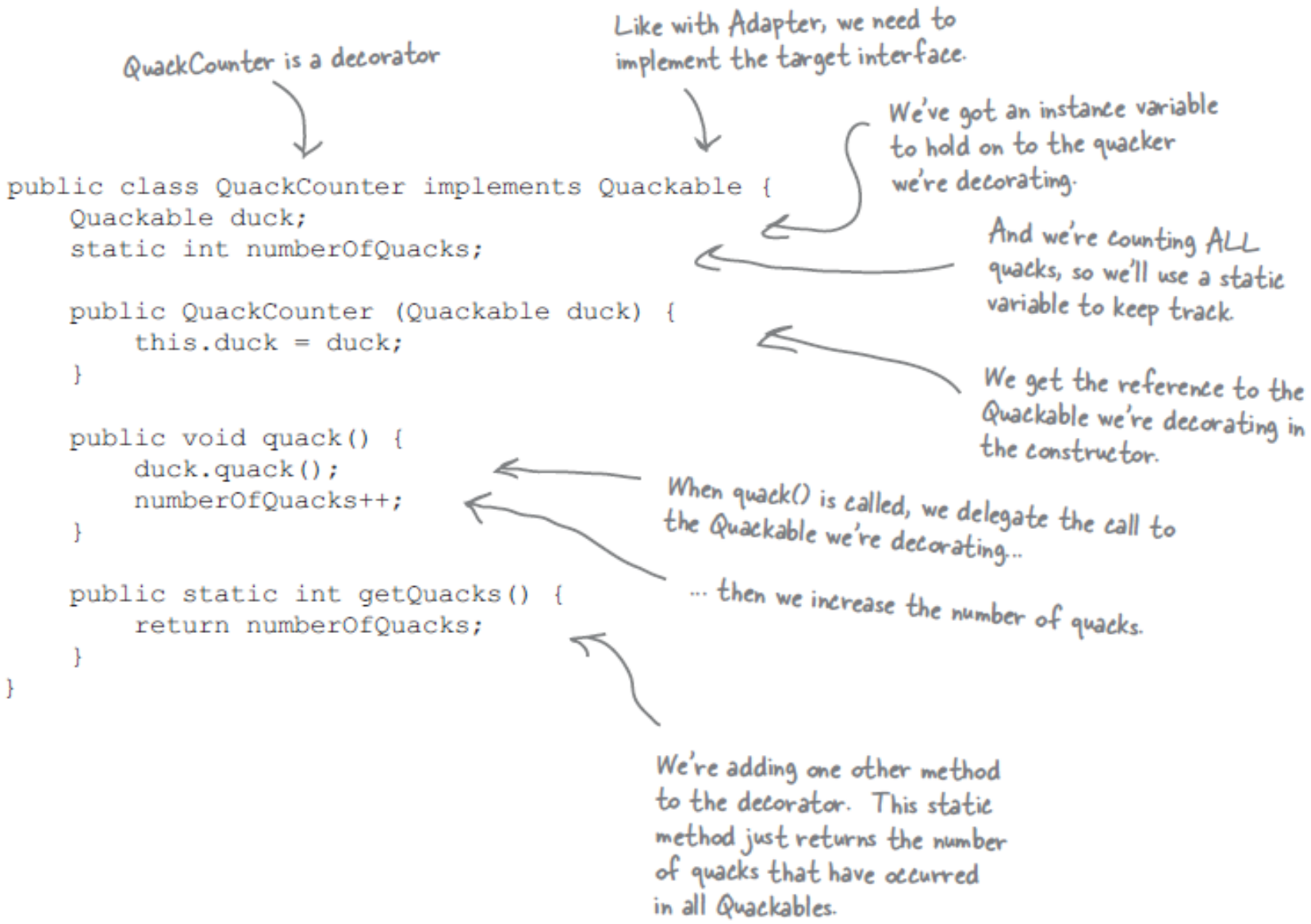
We make a Goose that acts like a Duck by wrapping the Goose in the GooseAdapter.

Once the Goose is wrapped, we can treat it just like other duck Quackables.

Compound Pattern(exersice)

- How can you add the ability to count duck quacks without having to change the duck classes ?

Compound Pattern(exersice)




```
public class QuackCounter implements Quackable {
    Quackable duck;
    static int numberOfQuacks;

    public QuackCounter (Quackable duck) {
        this.duck = duck;
    }

    public void quack() {
        duck.quack();
        numberOfQuacks++;
    }

    public static int getQuacks() {
        return numberOfQuacks;
    }
}
```