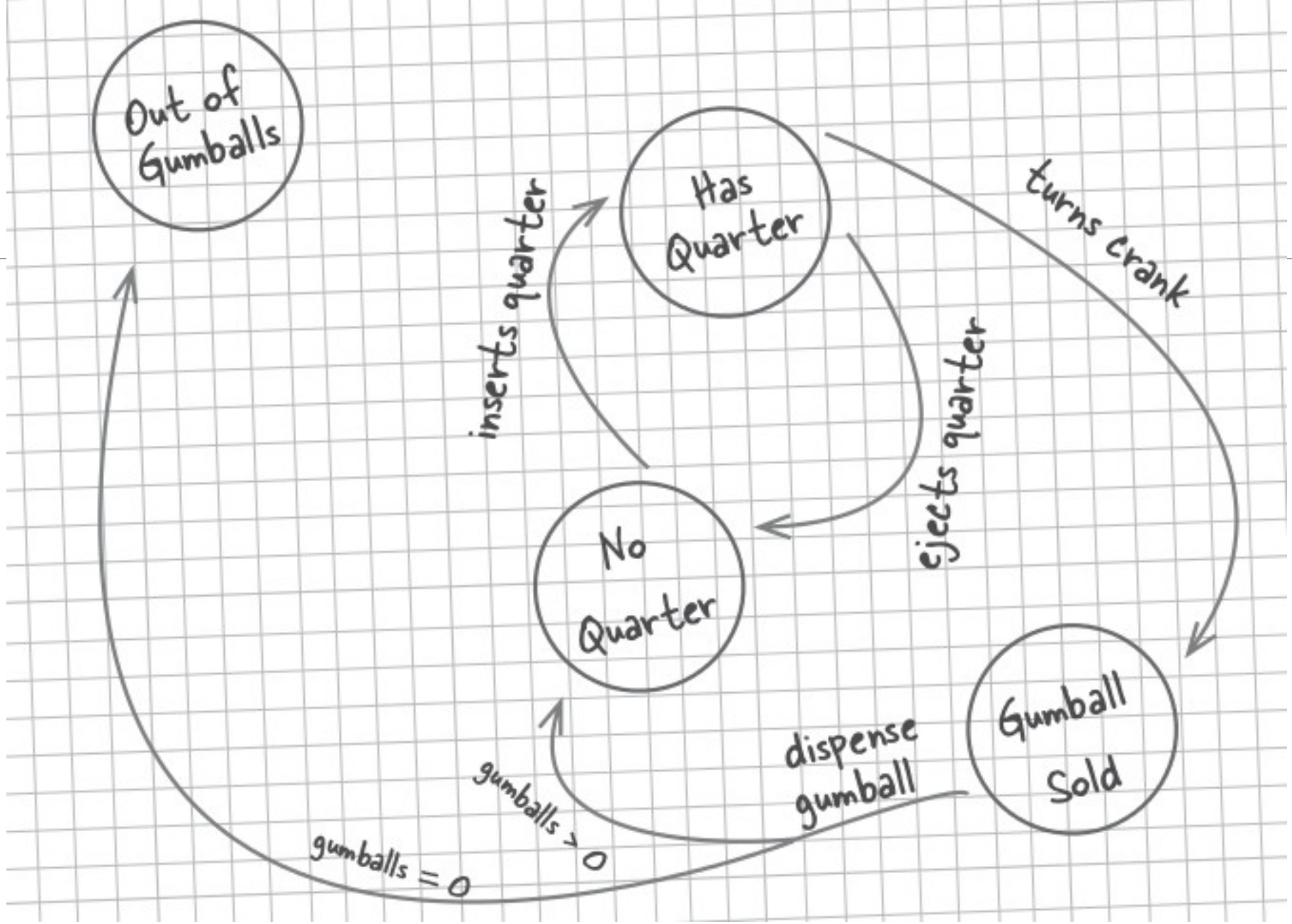
Patrón de diseño “State”

Problema: Maquina de dulces

Una empresa que fabrica máquinas de dulces te pide que le agregues una pantalla para indicar instrucciones a los clientes, les pides que te indiquen como funciona la máquina y te dan algo así.

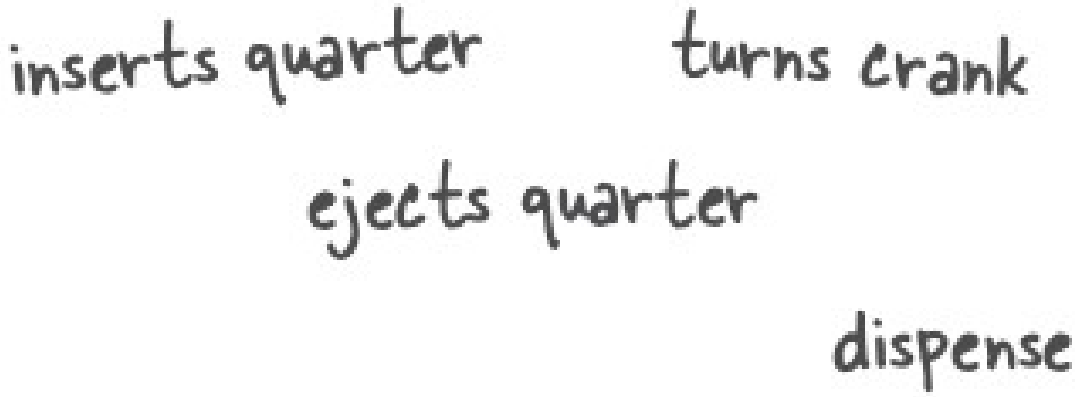

Out of Gumballs
Has Quarter
No Quarter
Gumball Sold
inserts quarter
turns crank
ejects quarter
dispense gumball
gumballs > 0
gumballs = 0

¿Qué hacemos con eso?

El diagrama no son especificaciones, es un diagrama de estados, así que hay que analizarlo por partes.

Los estados

Las acciones



Identificando los estados

Como primera idea pongamos cada estado como un entero y guardemos el “estado” actual en otro entero.

```
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;
```

Implementando las acciones

Las acciones las implementamos como métodos, pero hay que evitar que se realicen acciones en estados que no están permitidas esa acciones.

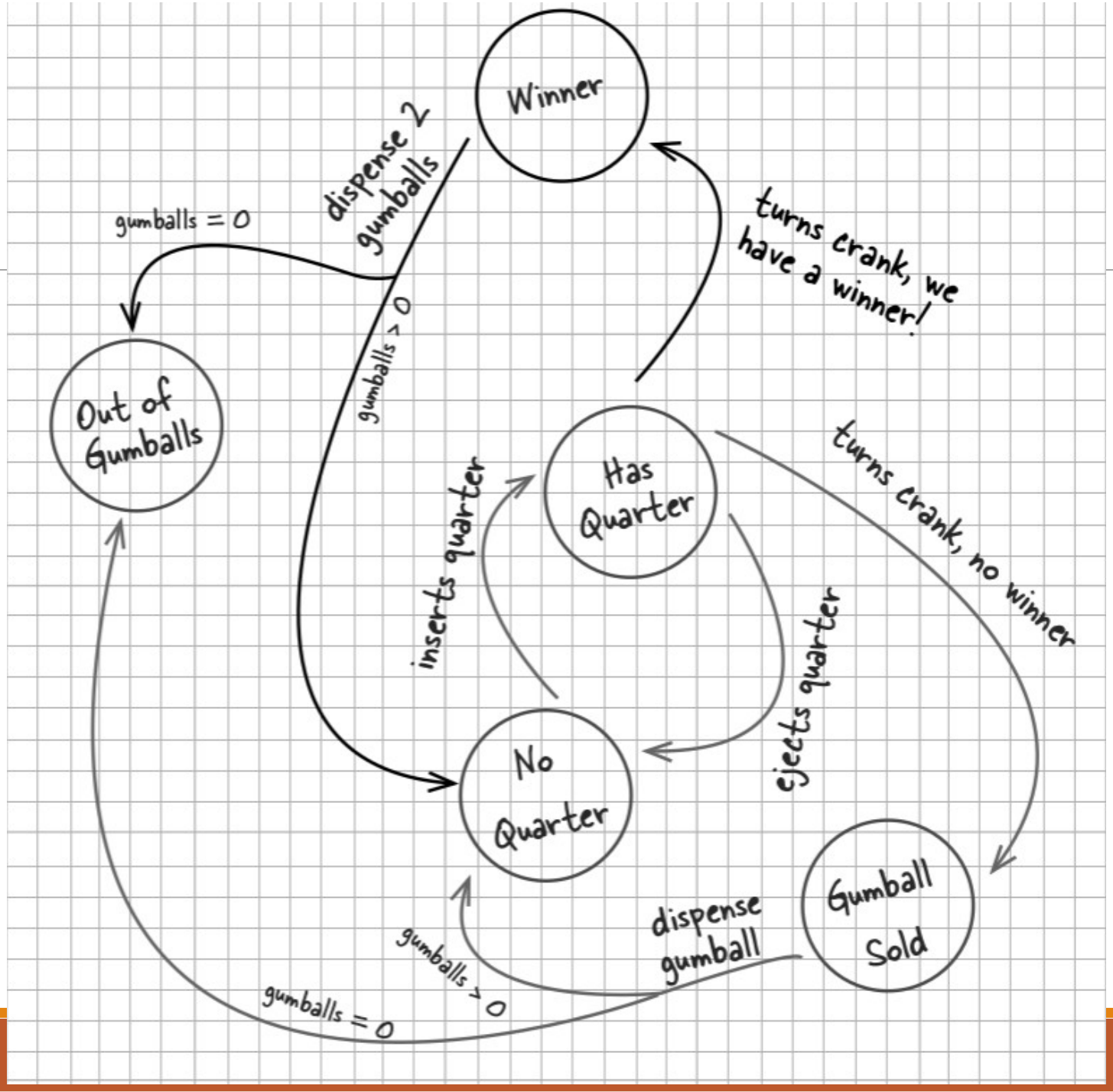
```
public void insertQuarter() {

    if (state == HAS_QUARTER) {

        System.out.println("You can't insert another 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");

    } else if (state == NO_QUARTER) {

        state = HAS_QUARTER;
        System.out.println("You inserted a quarter");

    }
}
```

Y así hacemos para los otros métodos.
FIN


TE LA CREISTE
WE XD
memegenerator.net

Y ahora… ¡cambios!

El código parece funcionar bien, pero ahora el cliente quiere agregar una promoción, una de cada diez veces la máquina entregará dos dulces en lugar de uno.


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

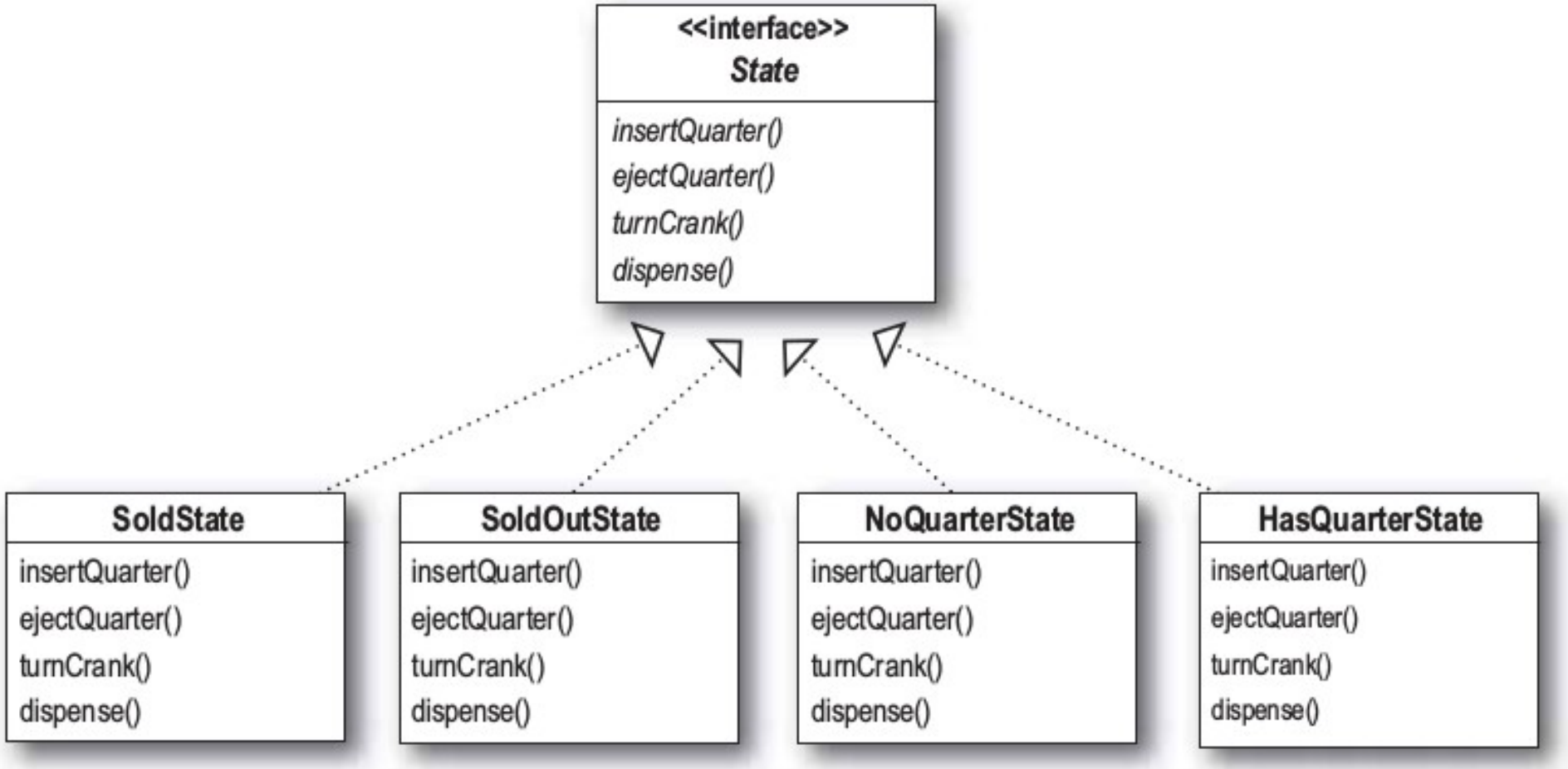
¿Cuál es el problema?

Para cada estado que se agregue hay que agregar condicionales en cada método que existe, además de agregar la transición de estados y el nuevo estado.

¿Cómo arreglamos esto?

Recuerden los principios de diseño hay que encapsular lo que cambia, en este caso lo que cambia son las acciones a realizar dependiendo del estado.

Diagrama de estados



```
public class GumballMachine {

    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;
    int count = 0;
```

```
public class GumballMachine {

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

    State state = soldOutState;
    int count = 0;
```

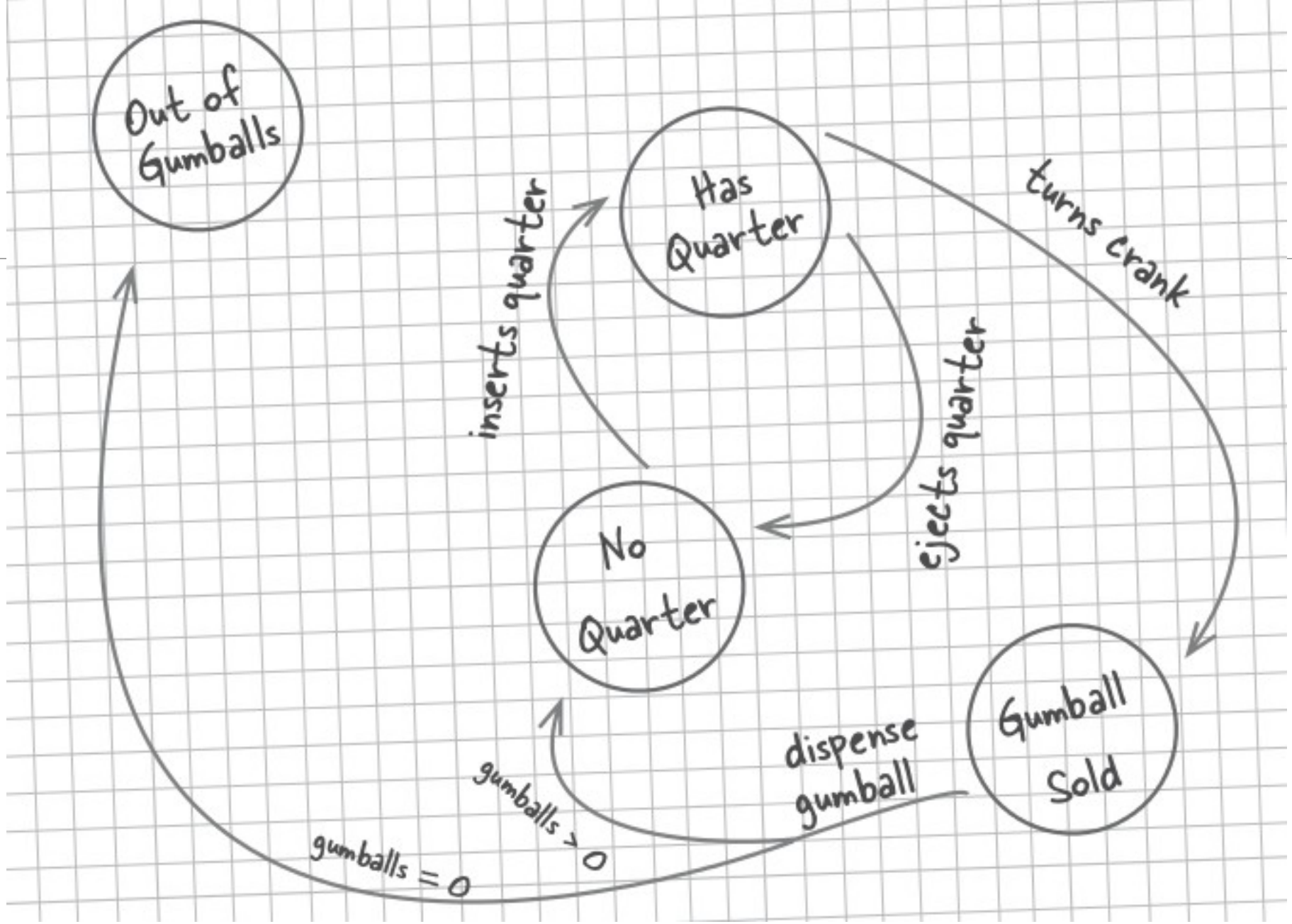
```
public class GumballMachine {

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

    State state;
    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;
        } else {
            state = soldOutState;
        }
    }

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

```
public interface State {

    public void insertQuarter();
    public void ejectQuarter();
    public void turnCrank();
    public void dispense();
}
```

```java
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");
    }
```

```
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");
    }
```

```
public class SoldState implements State {

    GumballMachine gumballMachine;

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

    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());
        }
    }
```

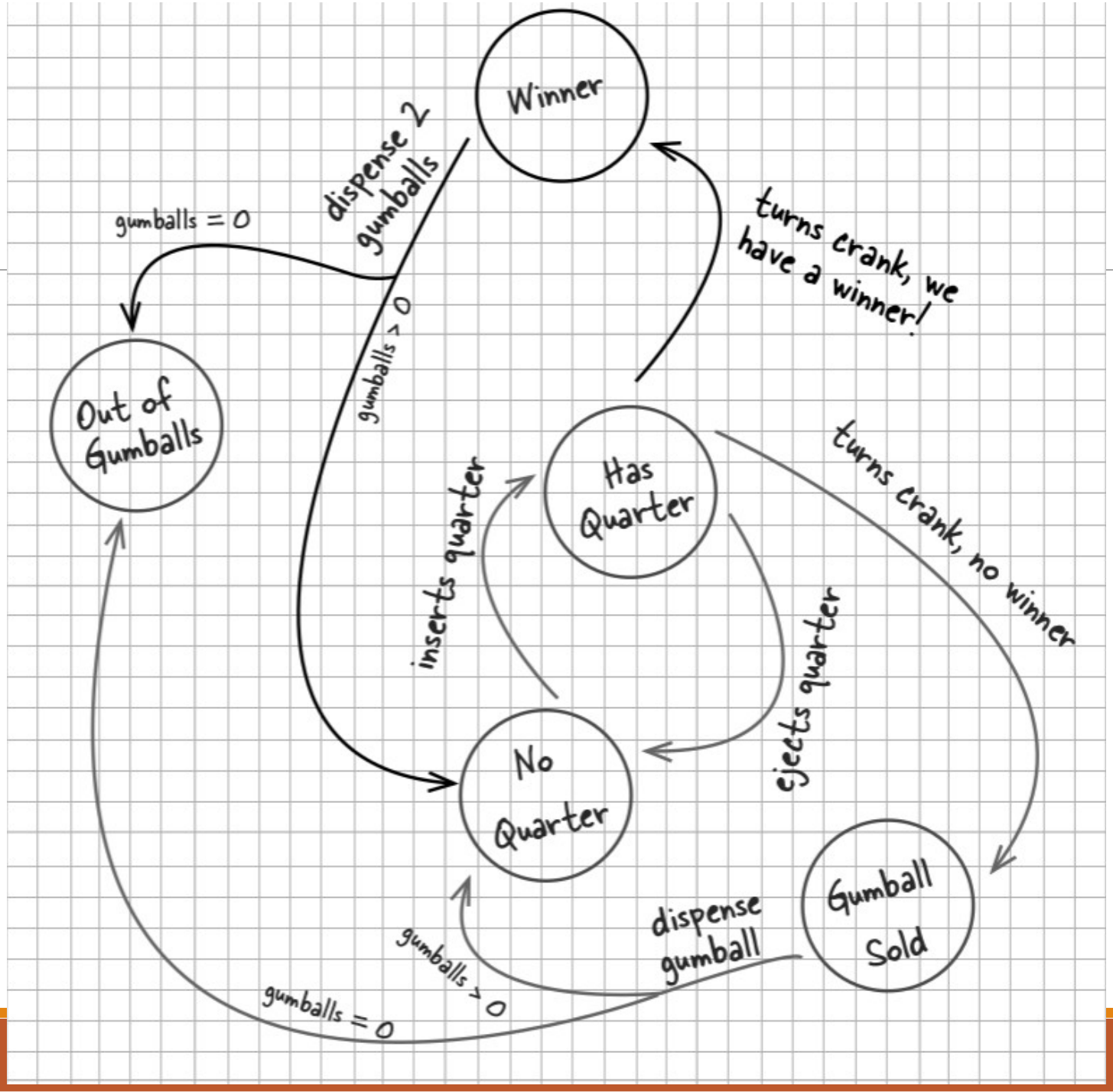

Out of Gumballs
Has Quarter
No Quarter
Gumball Sold
inserts quarter
turns crank
ejects quarter
dispense gumball
gumballs > 0
gumballs = 0

```
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");
    }
```

Ahora si van los cambios para implementar la

Cambios en la clase principal

```
public class GumballMachine {

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

    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);
        winnerState = new WinnerState(this);
```


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

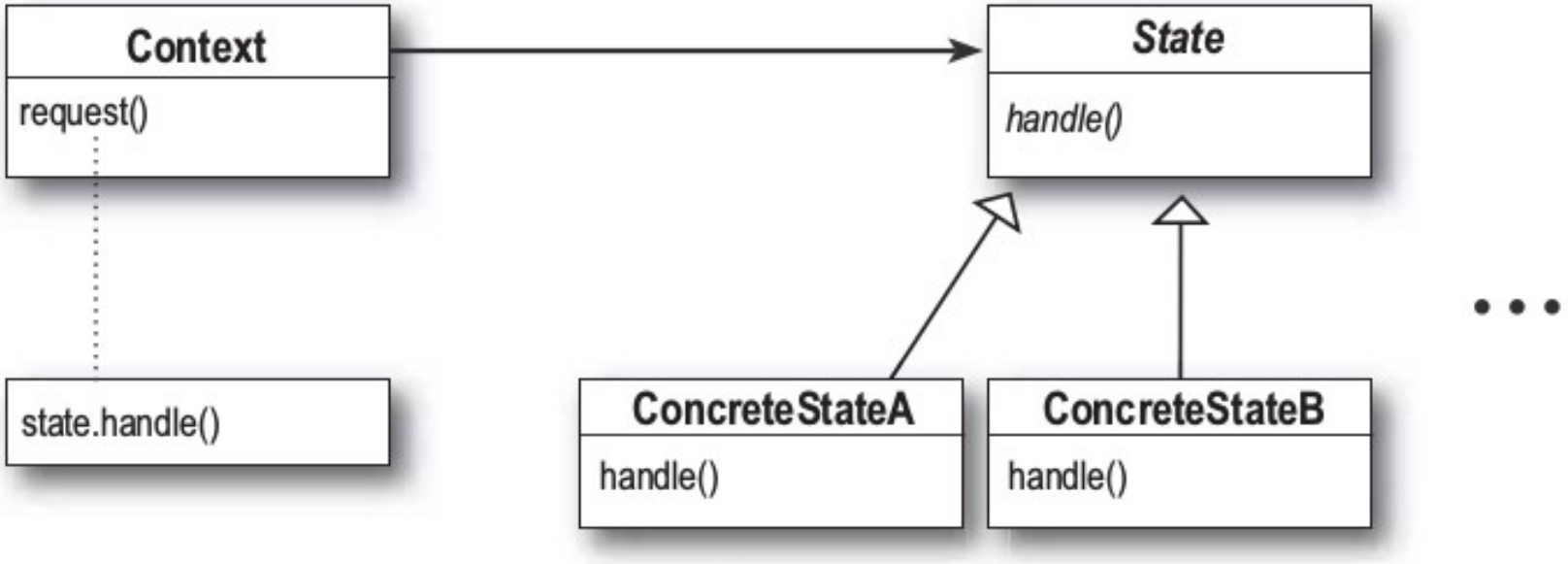
```
public class HasQuarterState implements State {
    Random randomWinner = new Random(System.currentTimeMillis());
    GumballMachine gumballMachine;

    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");
    }
```

```java
public class WinnerState implements State {
    GumballMachine gumballMachine;

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

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

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

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

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

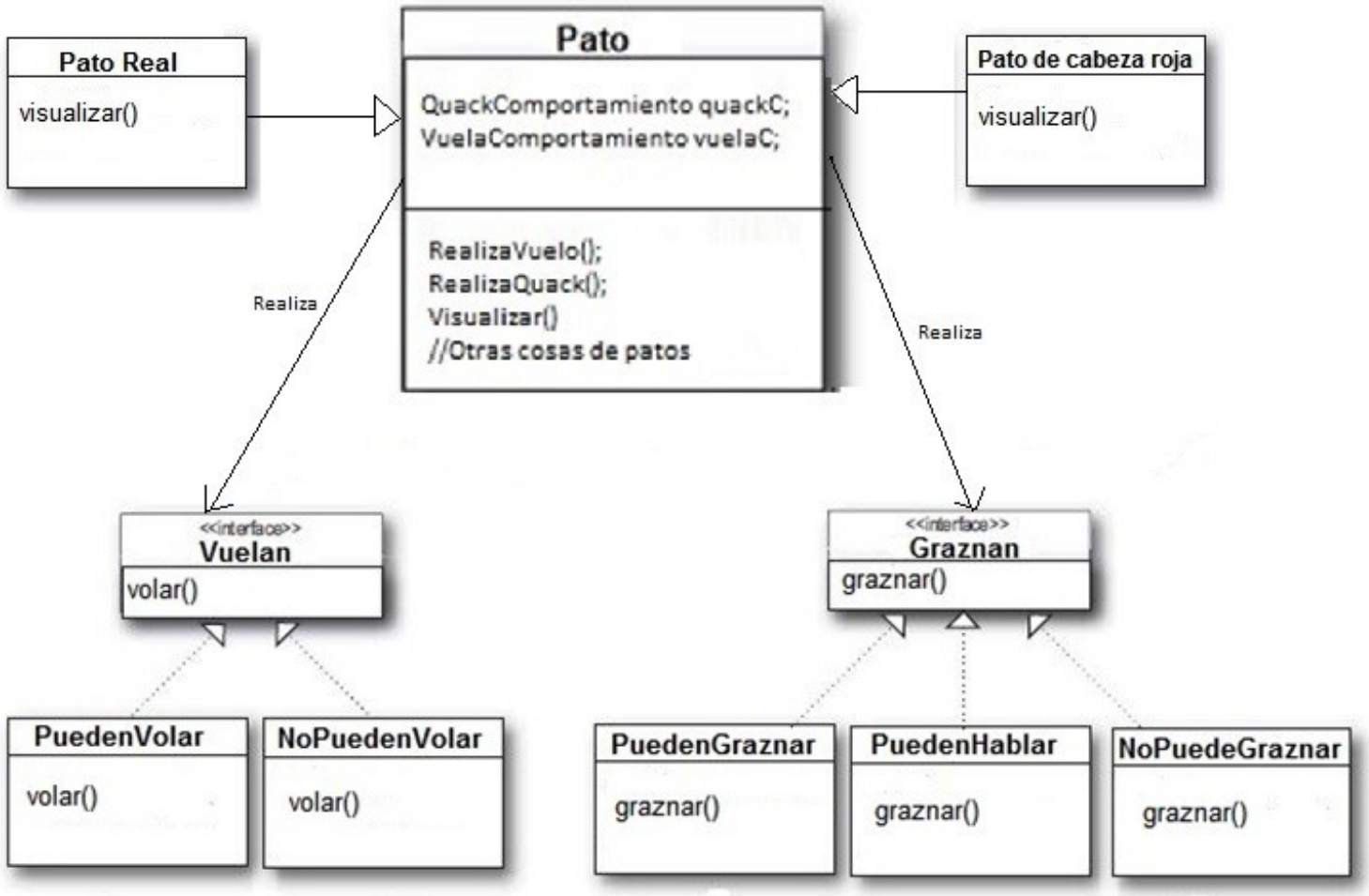
El patrón “State” definido

El patrón “state” permite a un objeto alterar su comportamiento cuando su estado interno cambia, el objeto aparenta cambiar su clase.

Diagrama general



¿Recuerdan a los patos?



Diferencias con “Strategy”

La diferencia es la intención:

Diferencias con “Strategy”

La diferencia es la intención:

- El patrón de diseño “strategy” se usa cuando tienes uno o varias clases que comparten cierta similitud en su comportamiento, y se puede cambiar entre estos, pero sin importar un estado interno.

- El patrón de diseño “state” se usa cuando una clase tiene distintas fases (estados) y en cada una de estas su comportamiento es distinto