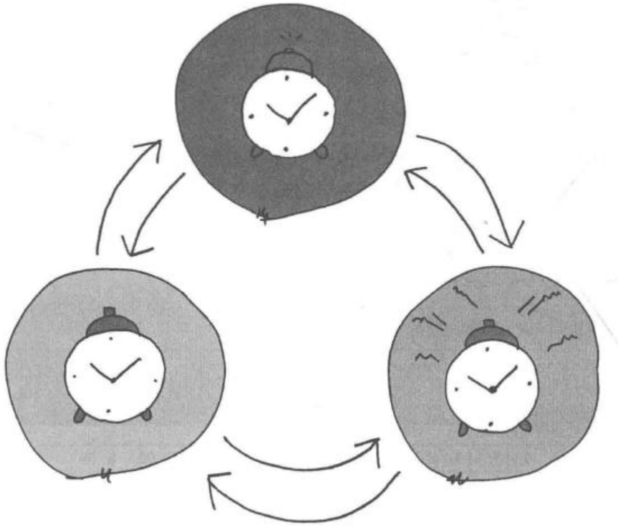
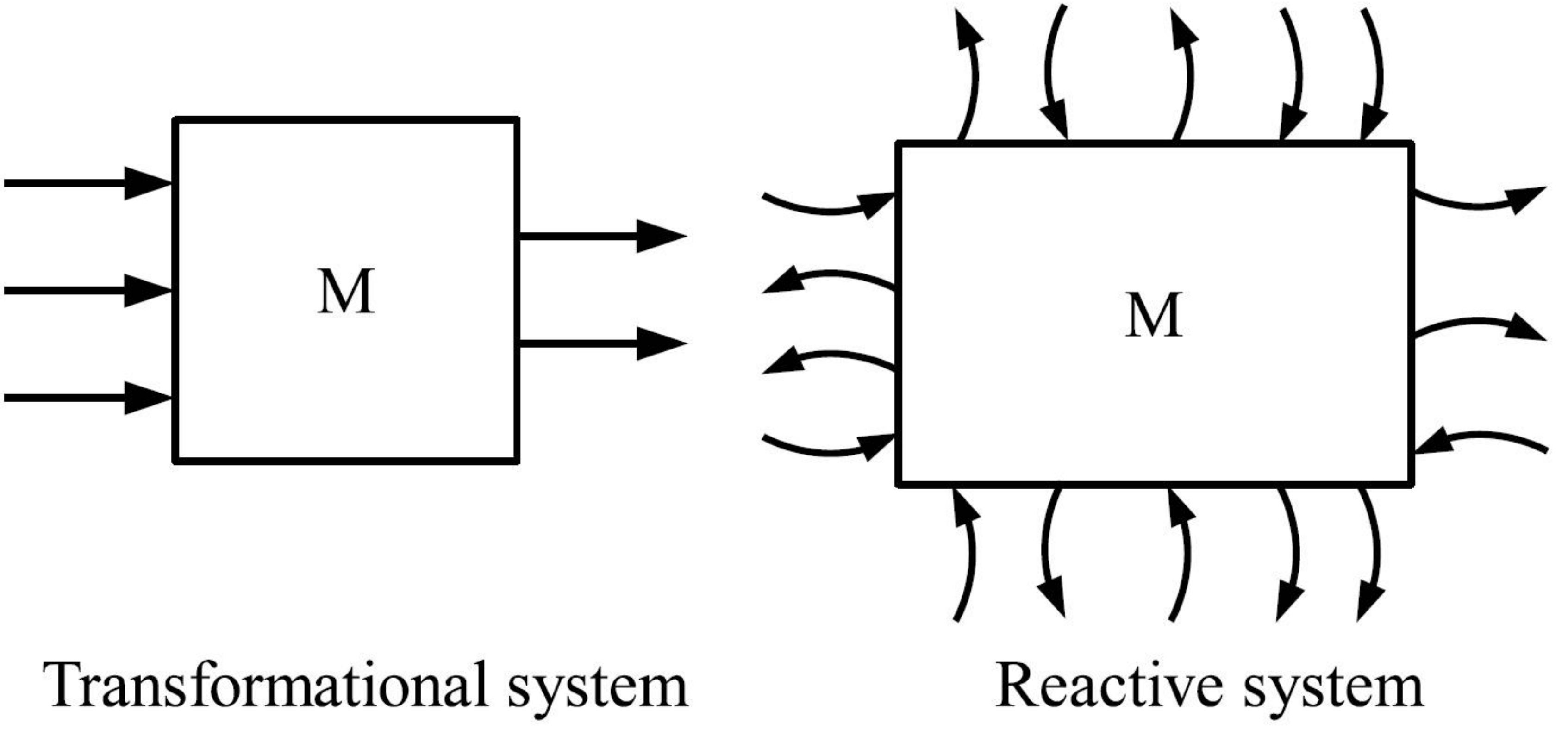
State
(状态,
Behavioral Pattern)



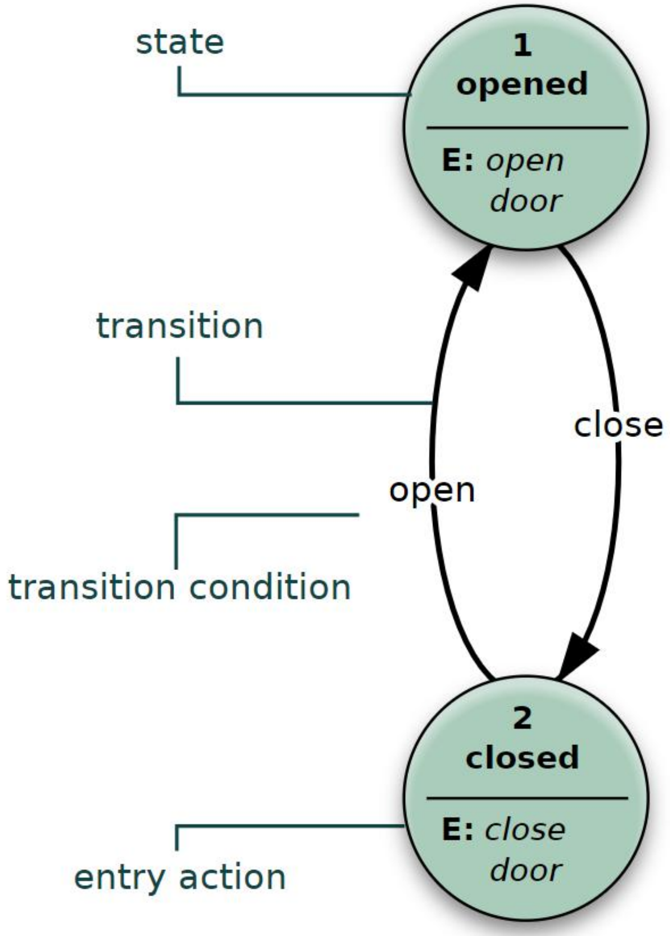
Kai SHI

Reactive Systems [Harel and Pnueli 1985]



State Diagram [Harel]

- A state diagram is a type of diagram used in computer science and related fields to describe the behavior of systems.
- Part of the UML



STATEMATE: A working environment for the development of complex reactive systems [Harel, et al 1990]

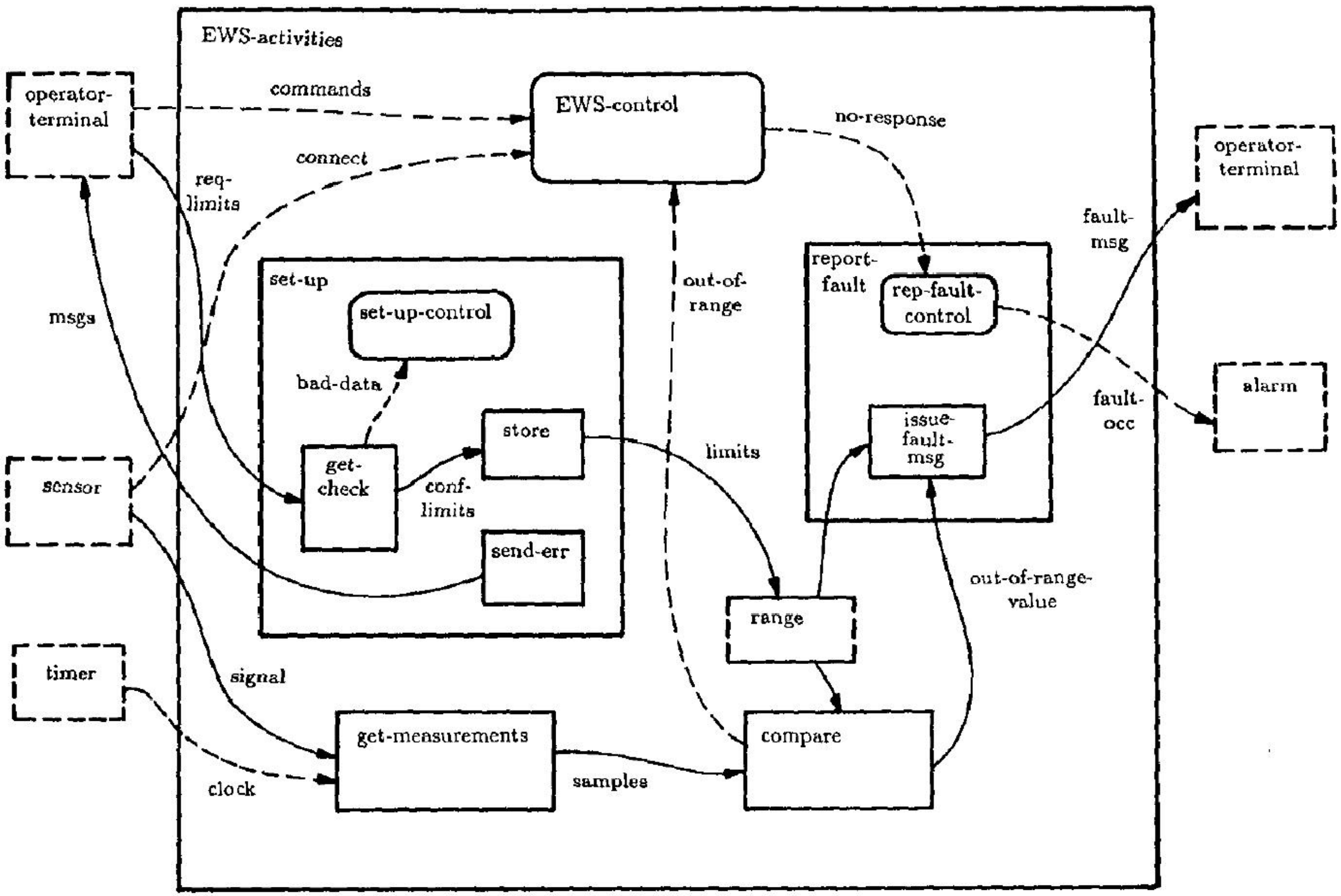
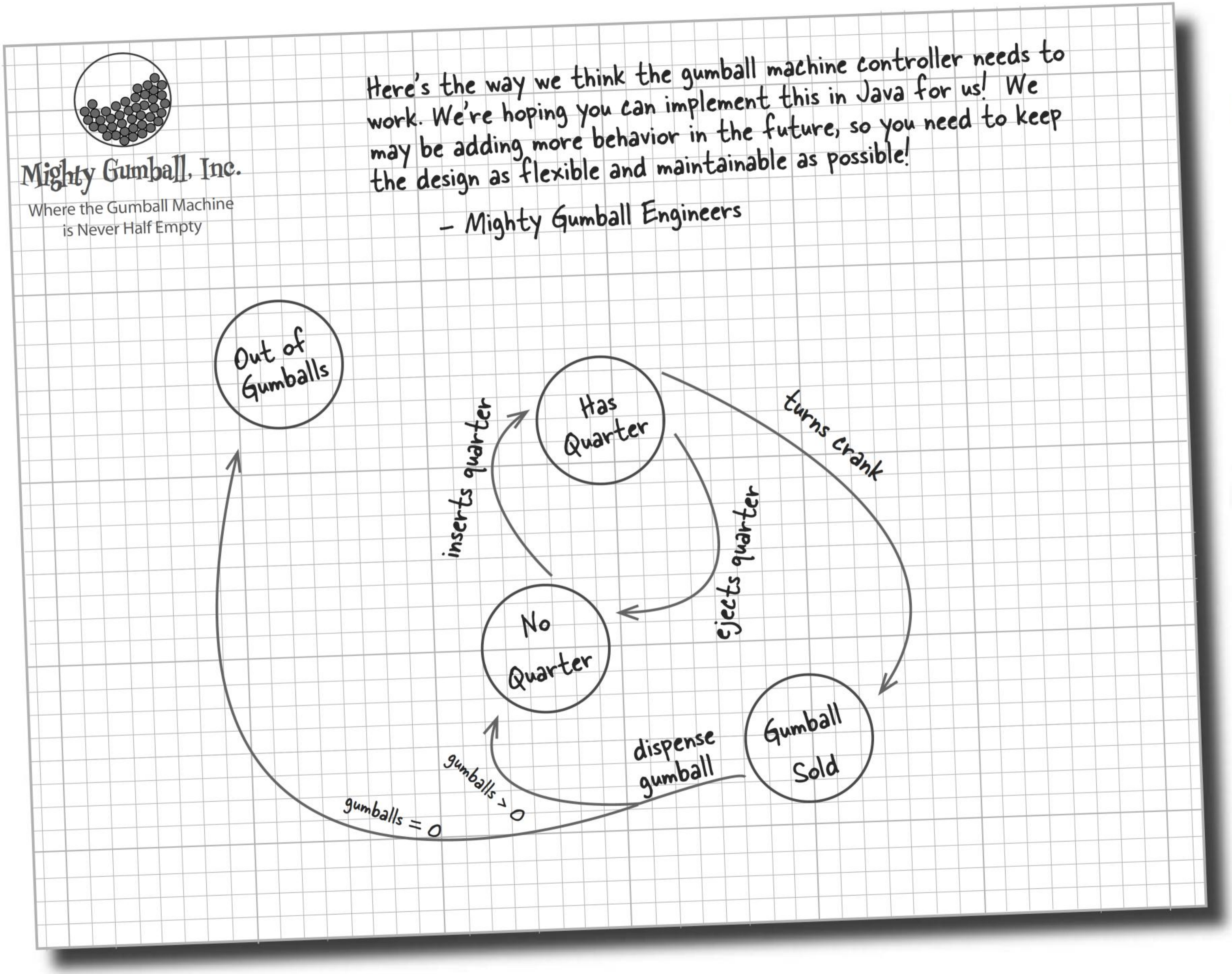
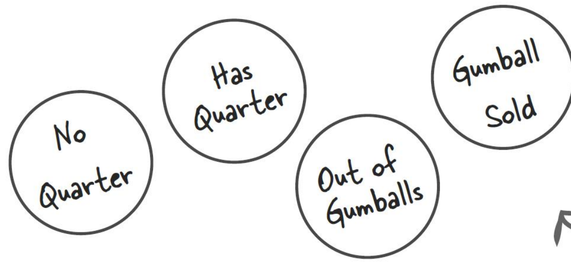


Figure 7: Activity-chart of the early warning system

Problem: Think about the solution



1 First, gather up your states:



Here are the states – four in total.

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.

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

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

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 == 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");

    }
}
```

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

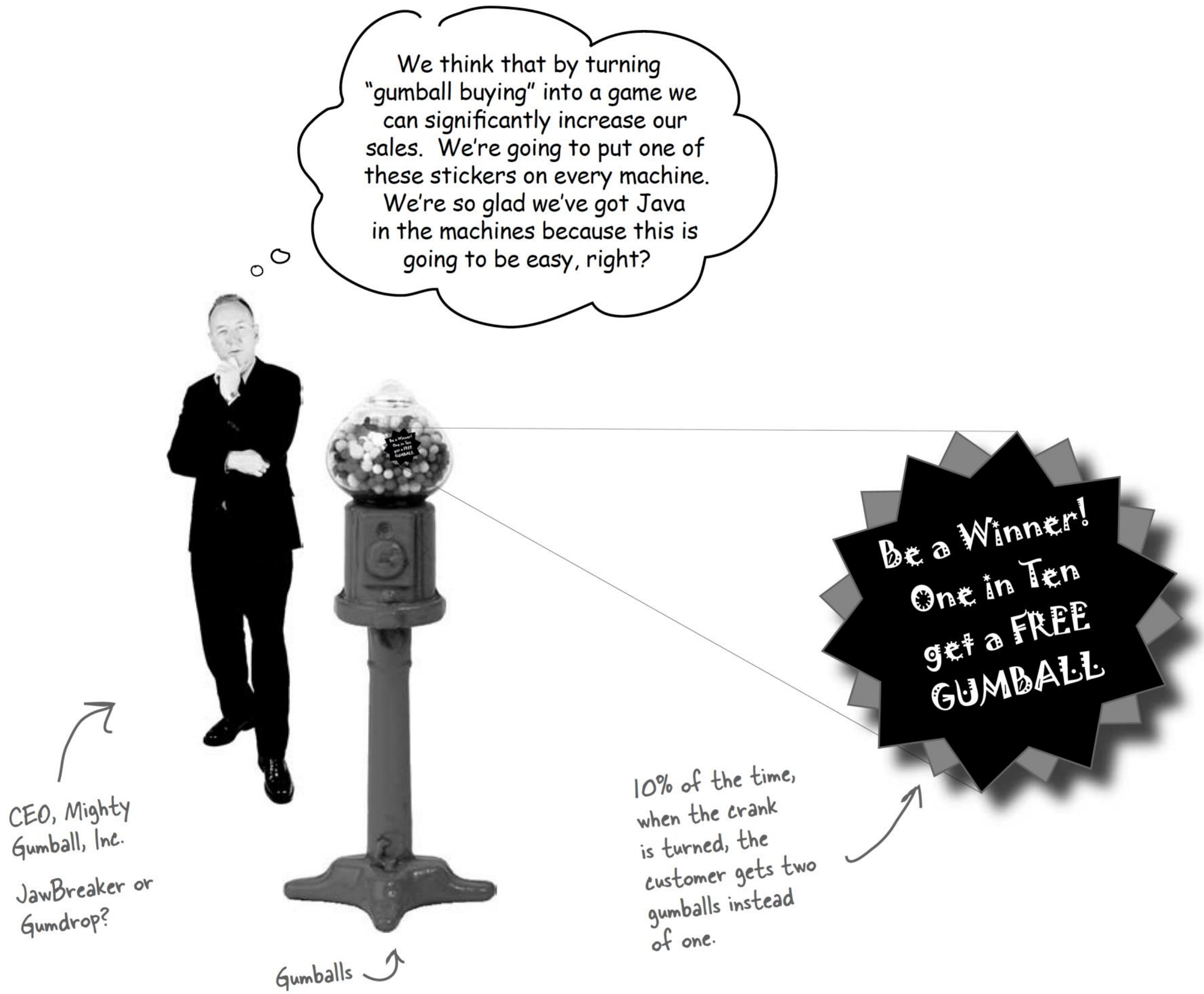
...and exhibits the appropriate behavior for each possible state...

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

The Code

- net.dp.state.gumball.GumballMachine
- net.dp.state.gumball.GumballMachineTestDrive

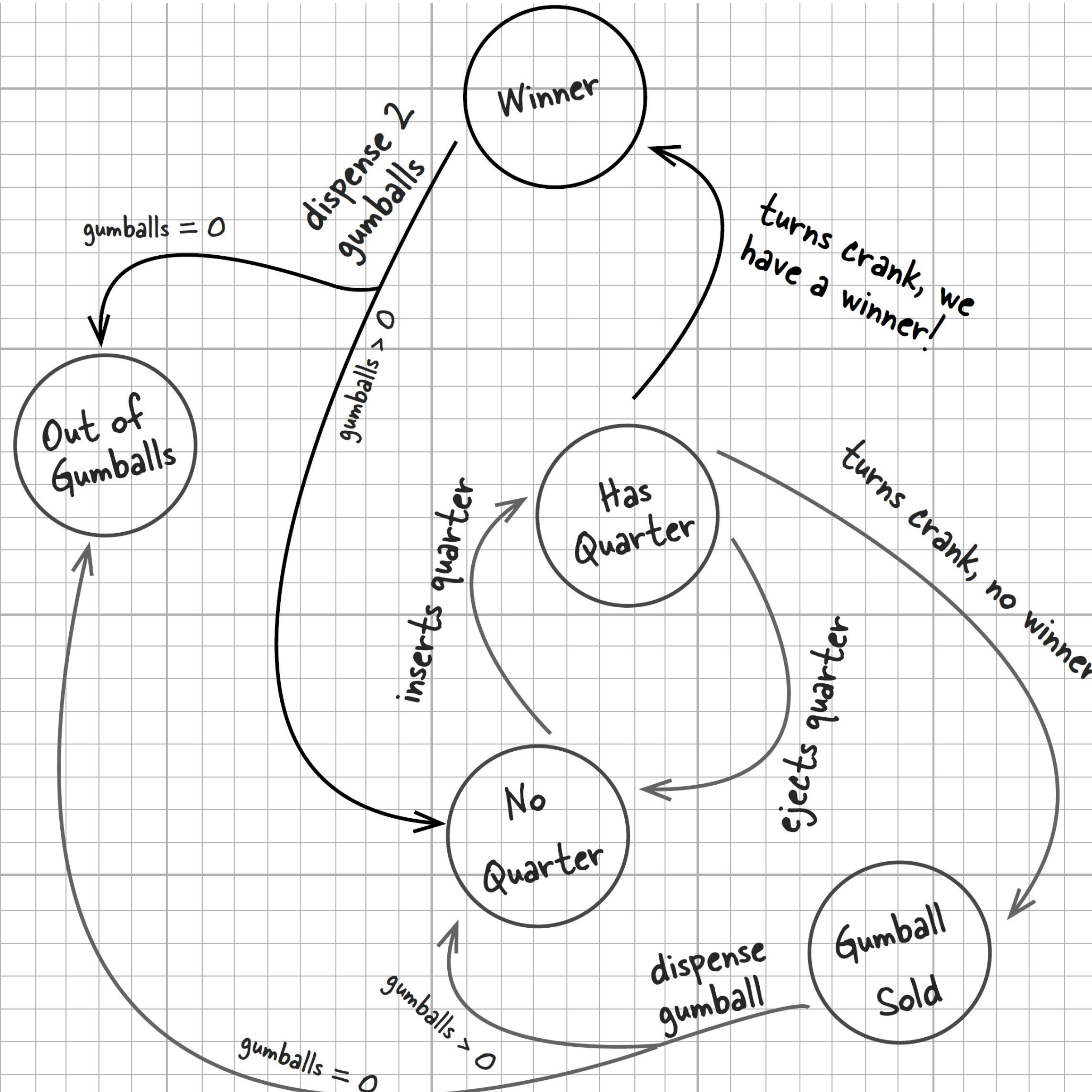
Requirements Change



- Draw the state diagram

Mighty Gumball, Inc.

Where the Gumball Machine is Never Half Empty



But you have to modify... Bad!

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

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

```
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
}
```

... 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.

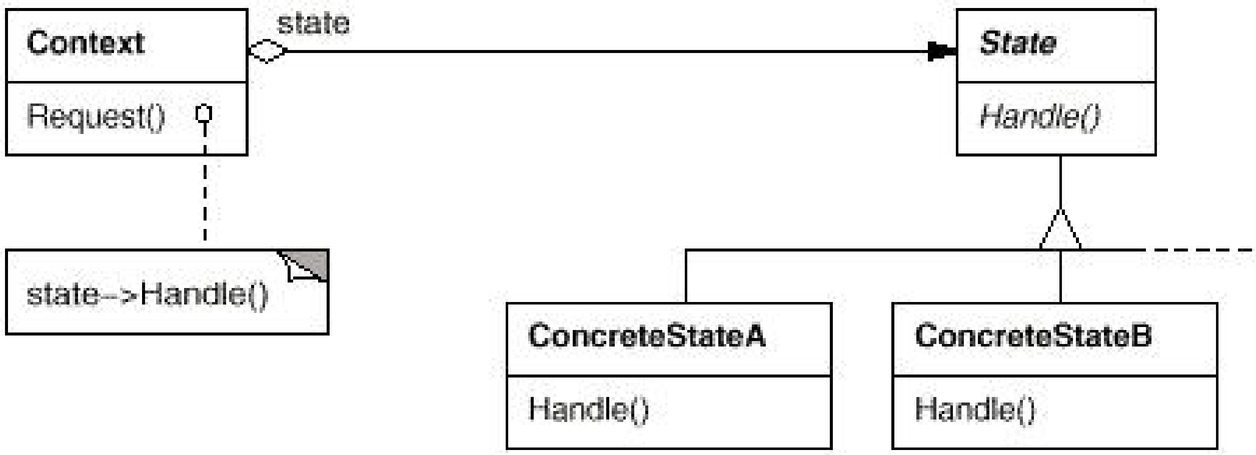
Intent

- Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
- 状态模式允许一个对象在其内部状态改变的时候 改变其行为。这个对象看上去就像改变了它的类一样。

Applicability: Use the State pattern in either of the following cases:

- An object's behavior depends on its state, and it must change its behavior at run-time depending on that state.
- Operations have large, multipart conditional statements that depend on the object's state.

Structure



Participants

- Context: Defines the interface of interest to clients; maintains an instance of a ConcreteState that defines the current state.
- State: Defines an interface for encapsulating the behavior associated with a particular state of the Context.
- ConcreteState: each implements a behavior associated with a state of the Context.

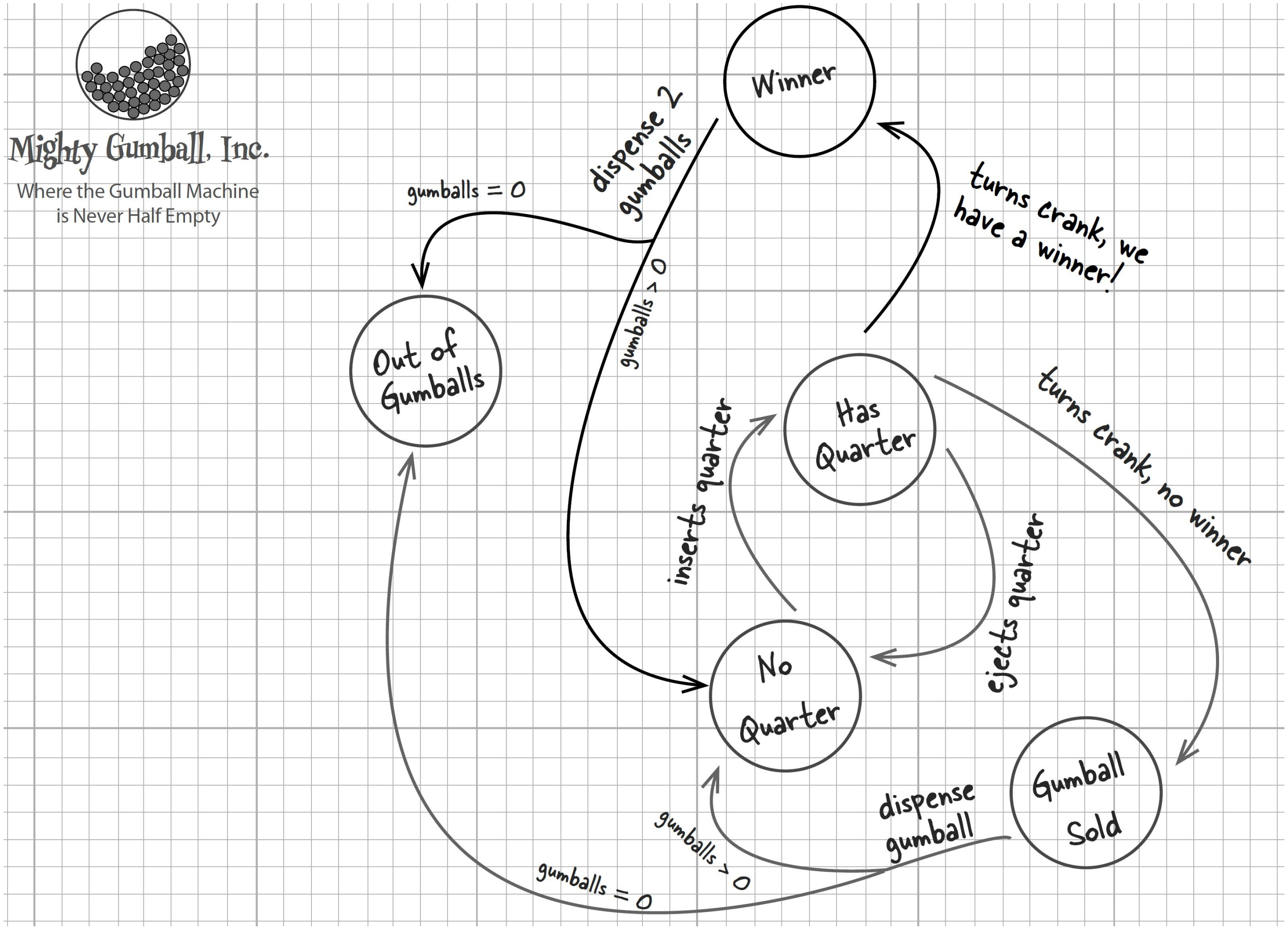
Collaborations

- Context delegates state-specific requests to the current ConcreteState object.
- A context may pass itself as an argument to the State object handling the request. This lets the State object access the context if necessary.
- Context is the primary interface for clients. State objects can be configured to context. Once a context is configured, its clients don't have to deal with the State objects directly.
- Either Context or the ConcreteState can decide which state succeeds another and under what circumstances (情况).

Consequences

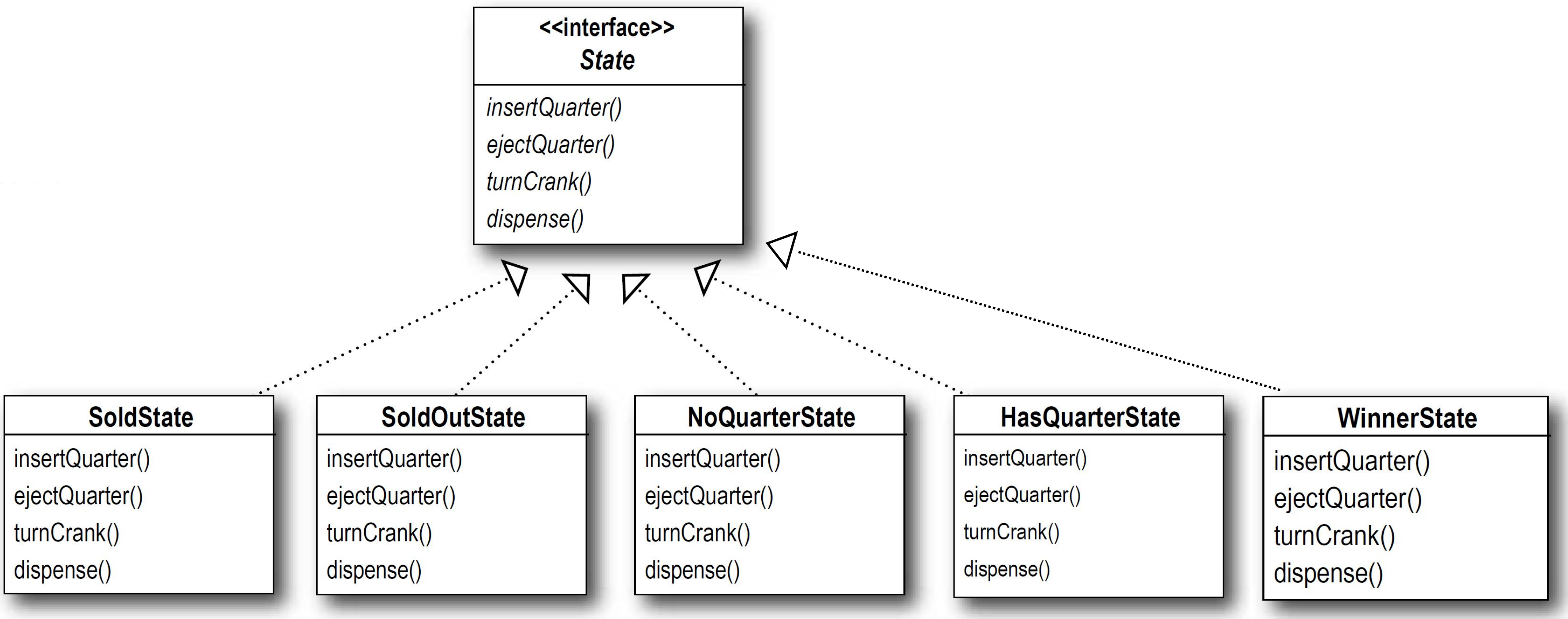
- It localizes state-specific behavior and partitions behavior for different states.
- It makes state transitions explicit.
  - When an object defines its current state by internal data values, its state transitions have no explicit representation; they only show up as assignments to some variables.
- State objects can be shared (Flyweight).

Refine the Solution

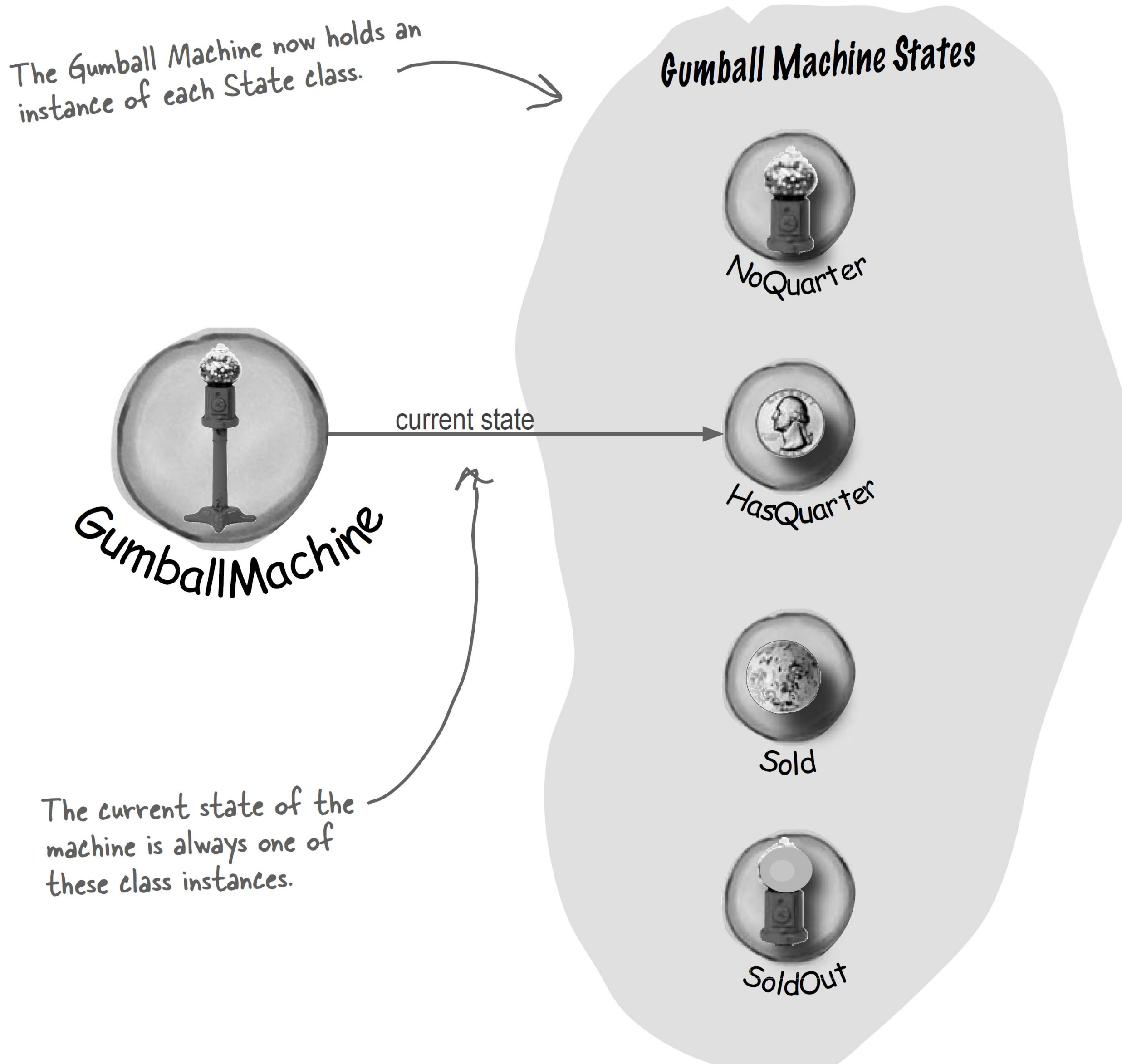


- Draw the class diagram

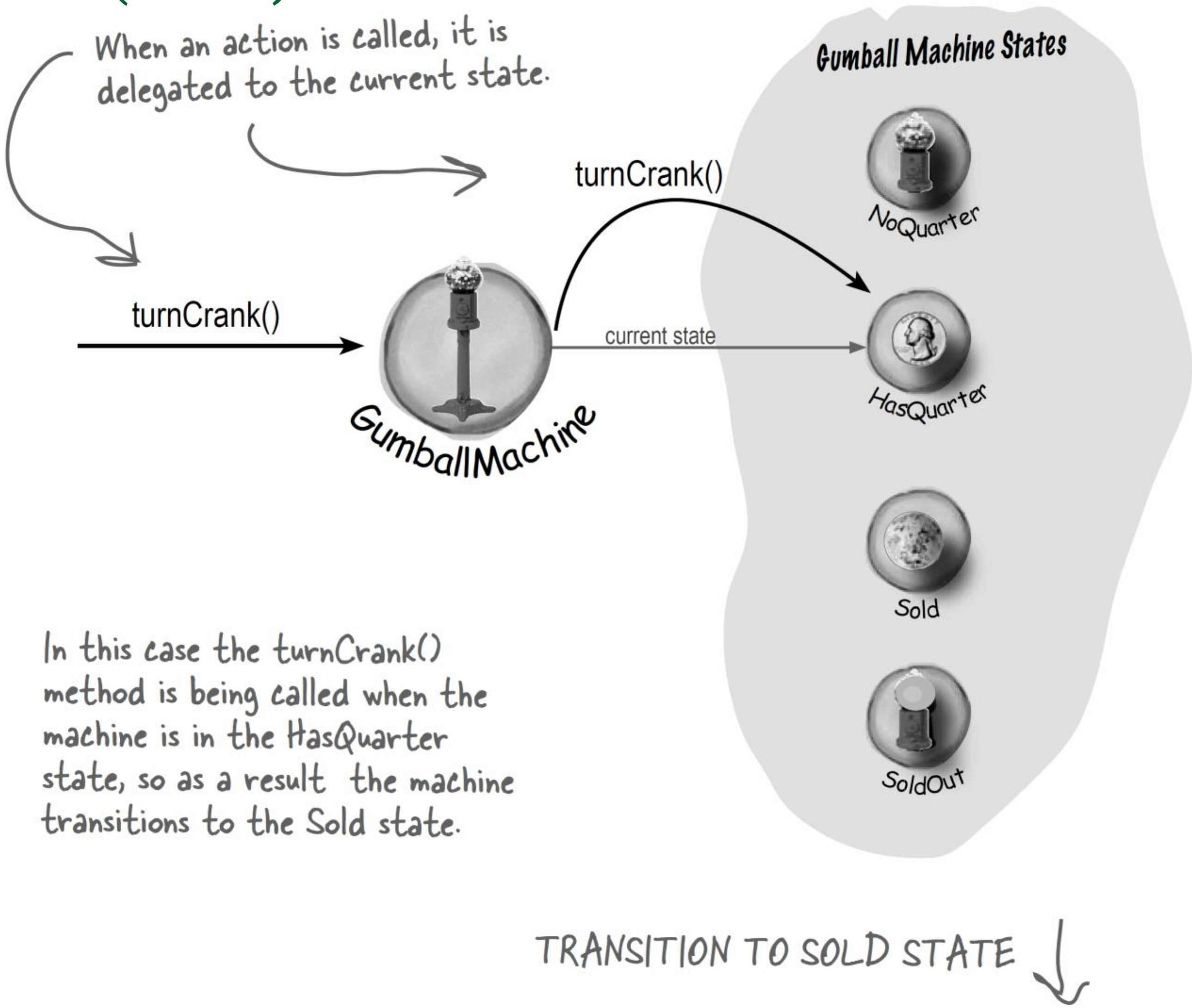
Class Diagram



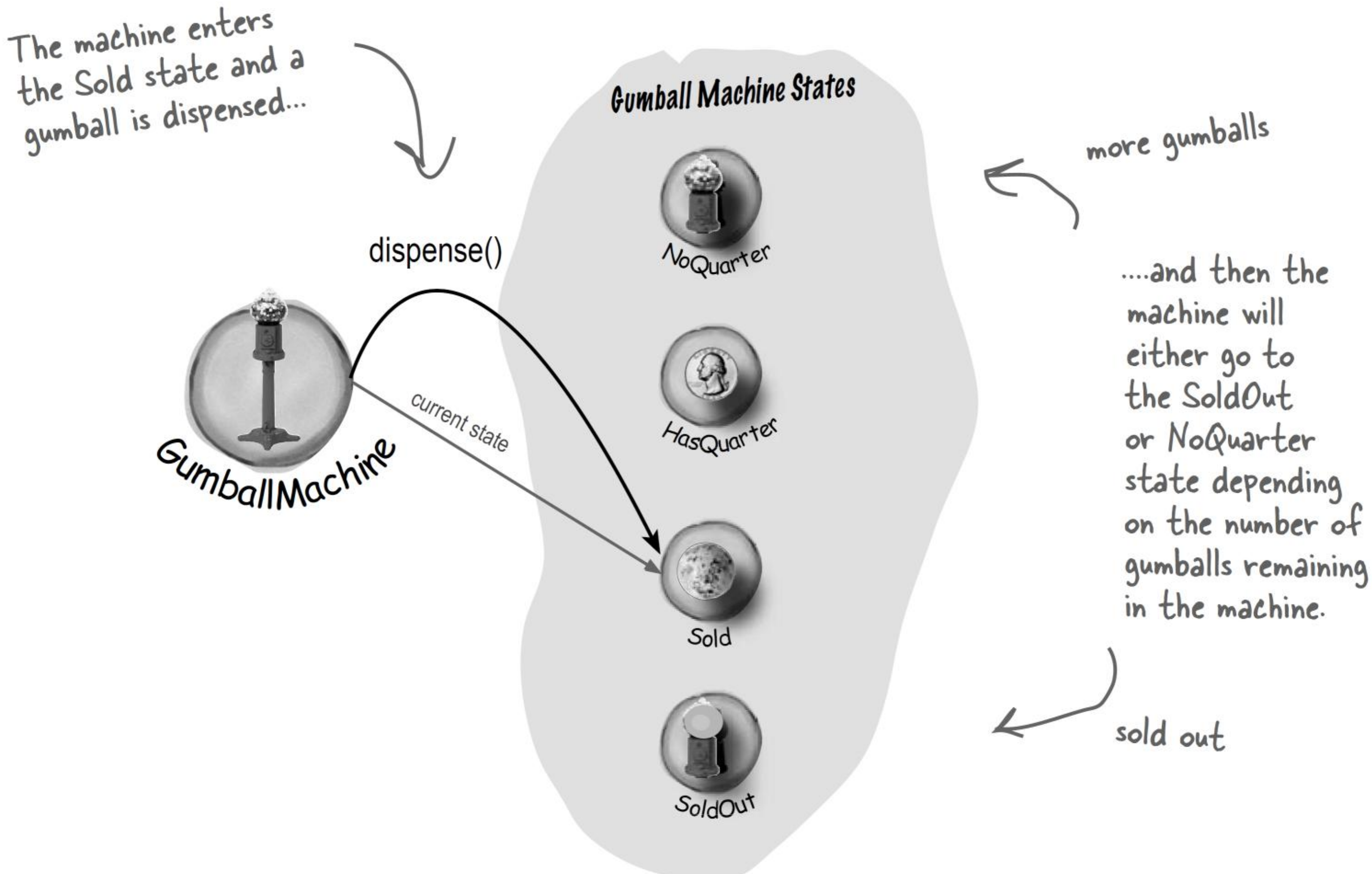
Details (1/3) (Without WinnerState)



Details (2/3)


When an action is called, it is delegated to the current state.
turnCrank()
turnCrank()
current state
GumballMachine
Gumball Machine States
NoQuarter
HasQuarter
Sold
SoldOut
In this case the turnCrank() method is being called when the machine is in the HasQuarter state, so as a result the machine transitions to the Sold state.
TRANSITION TO SOLD STATE

Details (3/3)


The machine enters the Sold state and a gumball is dispensed...
Gumball Machine States
more gumballs
dispense()
NoQuarter
GumballMachine
current state
HasQuarter
....and then the machine will either go to the SoldOut or NoQuarter state depending on the number of gumballs remaining in the machine.
Sold
sold out
SoldOut

Code

- Without WinnerState version
  - net.dp.state.gumballstate.GumballMachineTestDrive
- WinnerState version
  - net.dp.state.gumballstatewinner.GumballMachineTestDrive

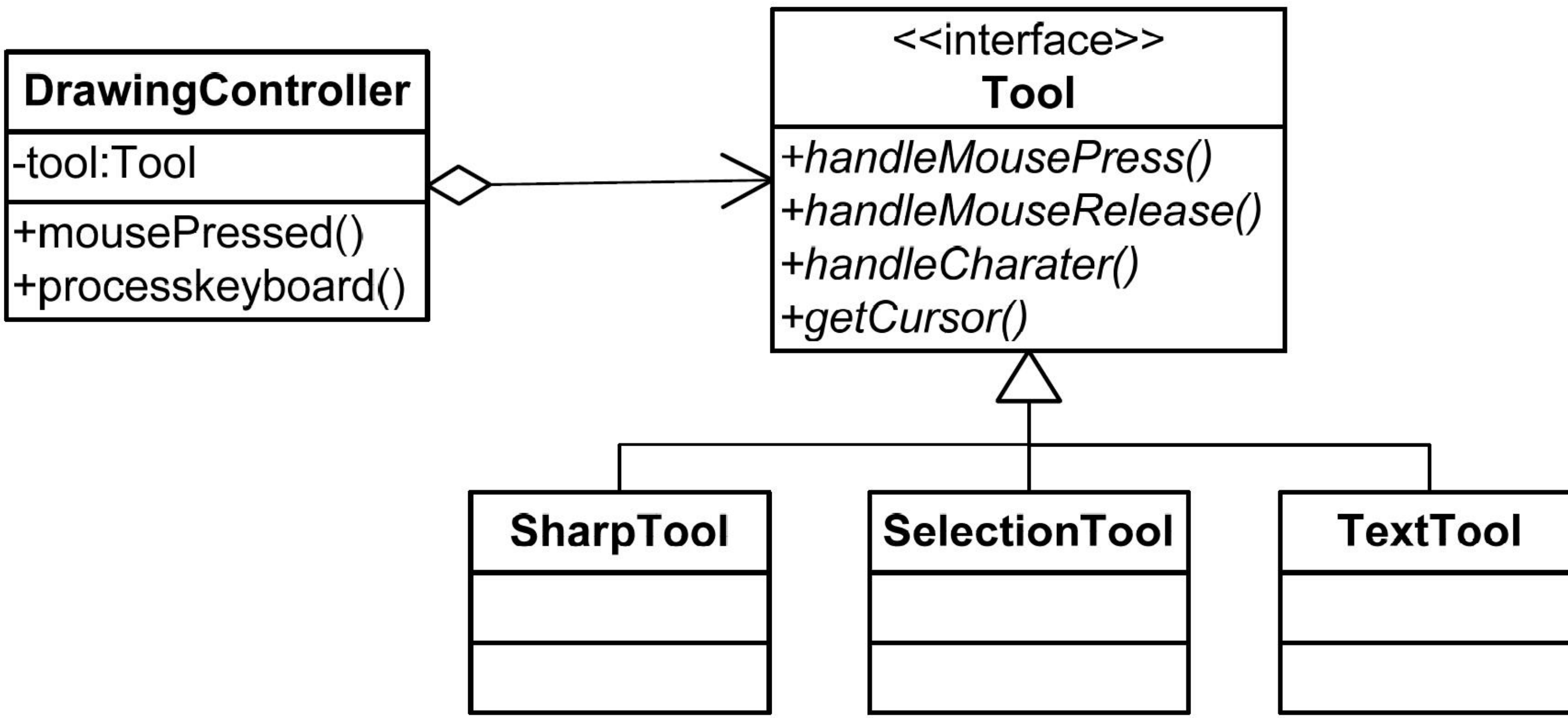
Implementation 1: Who defines the state transitions?

- The State pattern does not specify which participant defines the criteria for state transitions.
- If the criteria are fixed, then they can be implemented entirely in the Context.
- It is generally more flexible and appropriate to let the State subclasses themselves specify their successor state and when to make the transition.
  - It is easy to modify or extend the logic by defining new State subclasses.
  - A disadvantage is State subclass will have knowledge of at least one other, which introduces implementation dependencies between subclasses.

Implementation 2: Creating and destroying State objects.

- A common implementation trade-off worth considering is whether:
- Lazy: to create State objects only when they are needed and destroy them thereafter.
  - When the states that will be entered aren't known at runtime, and contexts change state infrequently.
- Eager: creating them ahead of time and never destroying them.
  - When state changes occur **rapidly**.

Example



DrawingController
-tool:Tool
+mousePressed()
+processkeyboard()
<<interface>>
Tool
+handleMousePress()
+handleMouseRelease()
+handleCharater()
+getCursor()
SharpTool
SelectionTool
TextTool

Extension: Table-driven approach (1/2)

- Using tables to map inputs to state transitions. For each state, a table maps every possible input to a succeeding state.
  - This approach converts conditional code into a table look-up.
- The main advantage of tables is their regularity: You can change the transition criteria by modifying data instead of changing program code.

Extension: Table-driven approach (2/2)

- Disadvantages
  - A table look-up is often less efficient than a function call.
  - Less explicit and harder to understand.
  - It's usually difficult to add actions to accompany the state transitions.
- The key difference between table-driven and the State pattern
  - The **State** pattern models state-specific **behavior**.
  - The **table**-driven approach focuses on defining **state transitions**.

Strategy VS State

- Strategy: **One** state with many algorithms;
- State: **many** States with different behaviors.