State Pattern

Idaho State University | Computer Science

Isaac Griffith

CS 2263
Department of Informatics and Computer Science
Idaho State University

ROAR

Outcomes

After today’s lecture you will be able to:

- Understand the use of the State Design Pattern
- Use and implement the State Pattern

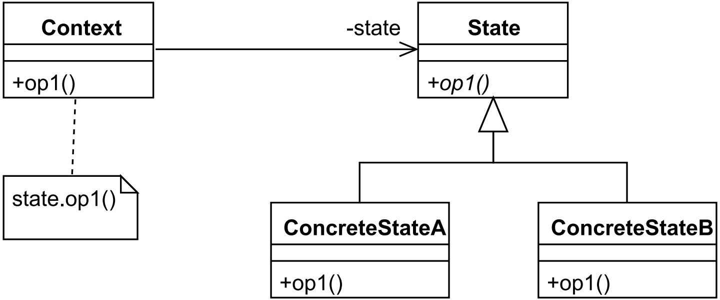
Inspiration

“Einstein argued that there must be simplified explanations of nature, because God is not capricious or arbitrary. No such faith comforts the software engineer.” – Fred Brooks

State Pattern: Definition

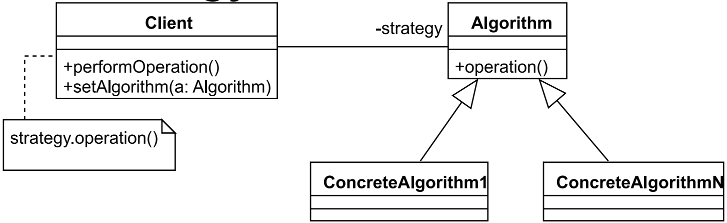
- Allows objects to vary behavior based on “state”
  - The object’s public interface doesn’t change, but
  - each method’s behavior differs as it’s internal state changes
- **Definition:** The State Pattern allows an object to alter its behavior when its internal state changes. The object will appear to change its class.
  - If we associate a class with behavior, then
    - Allows an object to dynamically alter behavior
    - Seems to change classes as it changes state

State Pattern: Structure



Look Familiar?

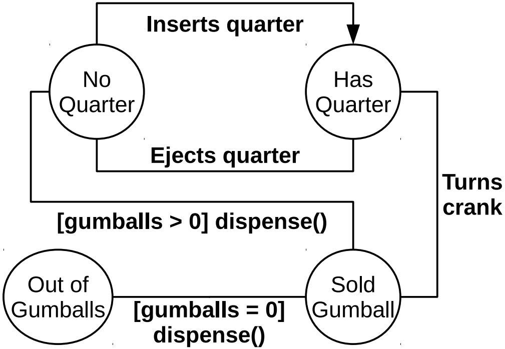
Strategy Pattern: Structure



Strategy and State Patterns: Separated at Birth?!

- Strategy and State are structurally equivalent, but
- Their intent is quite different.
- Strategy shares behavior between classes without inheritance
  - allowing dynamic run-time behavior configuration and changes
- State has a very different purpose, as we shall see.

Example: State Machines for Gumball Machines



- Each circle is a state of the gumball machine
- Each label corresponds to an event (method call)

Modeling State without State Pattern

- Create instance variable to track current state
  - Define constants: one for each state
    - For example:

      ```
      final static int SOLD_OUT = 0;
      int state = SOLD_OUT;
      ```

- Create class to act as a state machine
  - One method per state transition
  - Inside each method:
    - Code the behavior that transition for each state
    - We do this using conditional statements
  - **Demonstration**

Seemed Like a Good Idea At The Time...

- This approach is intuitive but naive
- Change requests makes problems apparent
  - Each change causes update to the state machine

Gumball Example

- You get the following request:
  - There is a 10% chance of dispensing two gumballs
  - “Has Quarter” now transitions as follows:
    - 90% -> “turns crank” leads to “Gumball Sold”
    - 10% -> “turns crank” leads to “Winner”
- The problem? Added one new state but need to update each action.

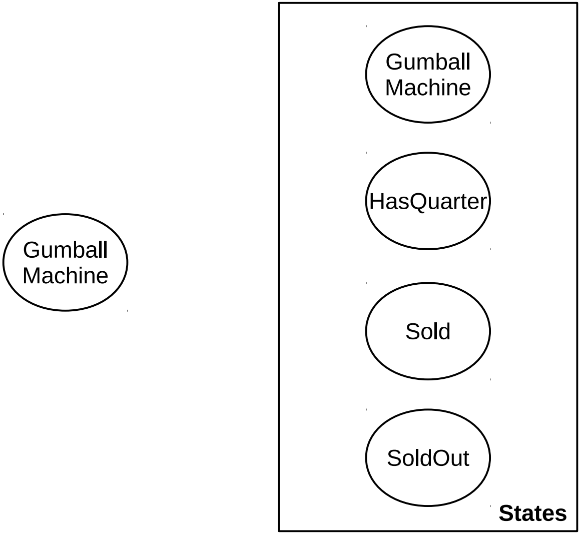
Design Problems with First Attempt

- Does not support **Open Close Principle**
  - Changes to the state machine require class changes
    - New behavior cannot be put in a subclass
- The design is not very OO:
  - No objects except for the `GumballMachine` (the `StateMachine`).
  - State transitions are not explicit
    - they are hidden amongst the conditional code
  - We have not “encapsulated what varies”

2nd Attempt: Use State Pattern

- Create a State interface that has one method per state transition
- Create one class per state of the state machine.
  - Each class implements the State interface
  - Each class provides the correct behavior in that state
- Changes to `GumballMachine` class:
  - Point to instances of the State implementations
  - Delegate all calls to that State instance.
  - Each action updates current state pointer to next instance
- **Demonstration**

State Pattern in Action



Gumball Machine

Gumball Machine

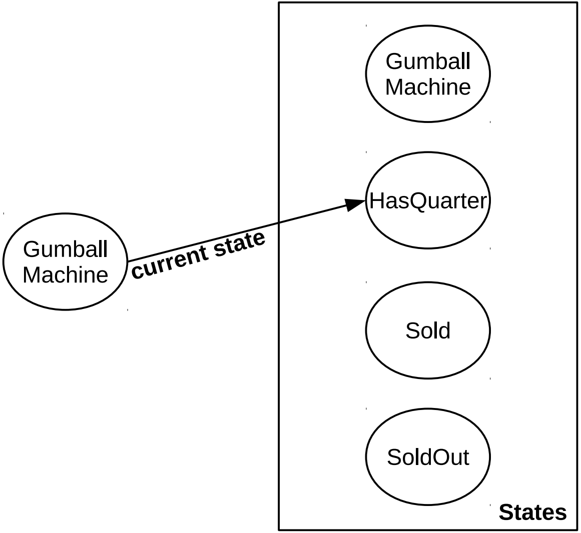
HasQuarter

Sold

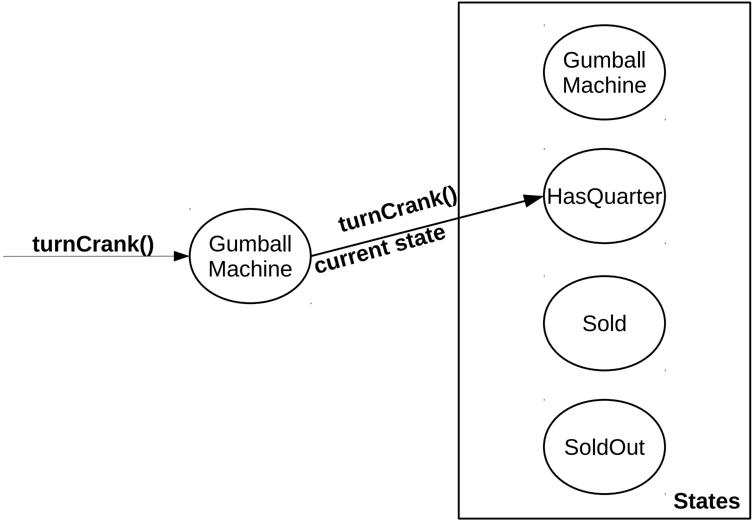
SoldOut

States

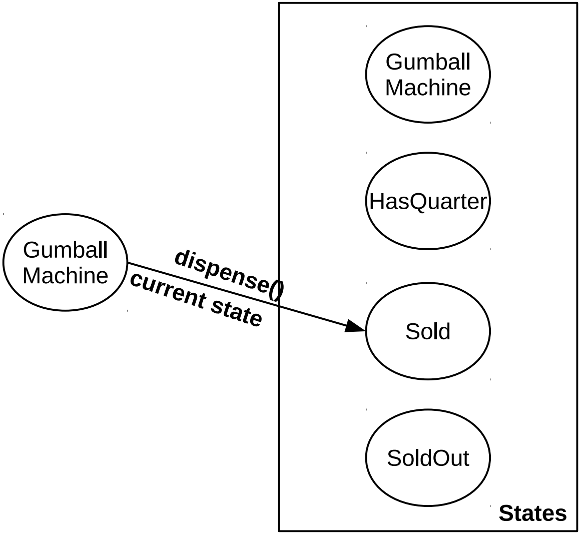
State Pattern in Action



State Pattern in Action



State Pattern in Action



Third Attempt: Implement 1 in 10 Game

- Demonstrates flexibility of State Pattern
  - Add a new State implementation: `WinnerState`
    - `WinnerState` extends `SoldState` and overrides `dispense()`
    - ensure the gumball machine has >= two gumballs
    - dispense two gumballs from the machine
- Update `HasQuarterState` to generate random number between 1 and 10
  - `if number == 1`: use an instance of `WinnerState`
  - `else`: use an instance of `SoldState`
- **Demonstration**

Wrapping Up

- 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 state machine object gets its behavior by delegating to its current state object
  - Each state object has the power to change the state of the state machine object, aka context object

Are there any questions?