

# Design Patterns

Engr. Abdul-Rahman Mahmood

DPM, MCP, QMR(ISO9001:2000)

 armahmood786@yahoo.com


 alphapeeler.sf.net/pubkeys/pkey.htm

 pk.linkedin.com/in/armahmood

 www.twitter.com/alphapeeler

 www.facebook.com/alphapeeler

 abdulmahmood-sss  alphasecure

 armahmood786@hotmail.com

 <http://alphapeeler.sf.net/me>

 alphasecure@gmail.com

 <http://alphapeeler.sourceforge.net>

 <http://alphapeeler.tumblr.com>

 armahmood786@jabber.org

 alphapeeler@aim.com

 mahmood\_cubix  48660186

 alphapeeler@icloud.com

 <http://alphapeeler.sf.net/acms/>

# State Design Pattern

# What is State pattern?

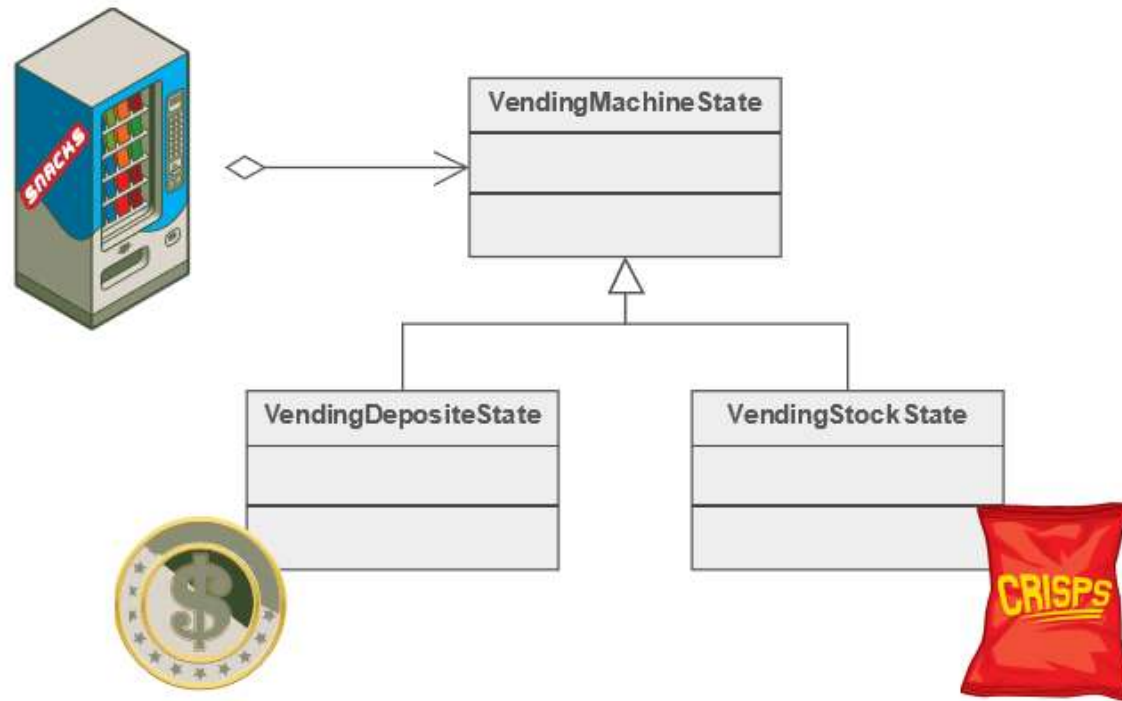
**State is one of the 23 Design Patterns which were selected by the GoF (Gang of Four).**

		Purpose		
		Creation	Structure	Behavior
Scope	Class	Factory Method		Interpreter Template
	Objects	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Façade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer <b>State</b> Strategy Visitor



# Example (a vending machine)

- When currency is deposited and a selection is made, a vending machine will either:
  - deliver a product and no change,
  - deliver a product and change,
  - deliver no product due to insufficient currency on deposit, or
  - deliver no product due to inventory depletion



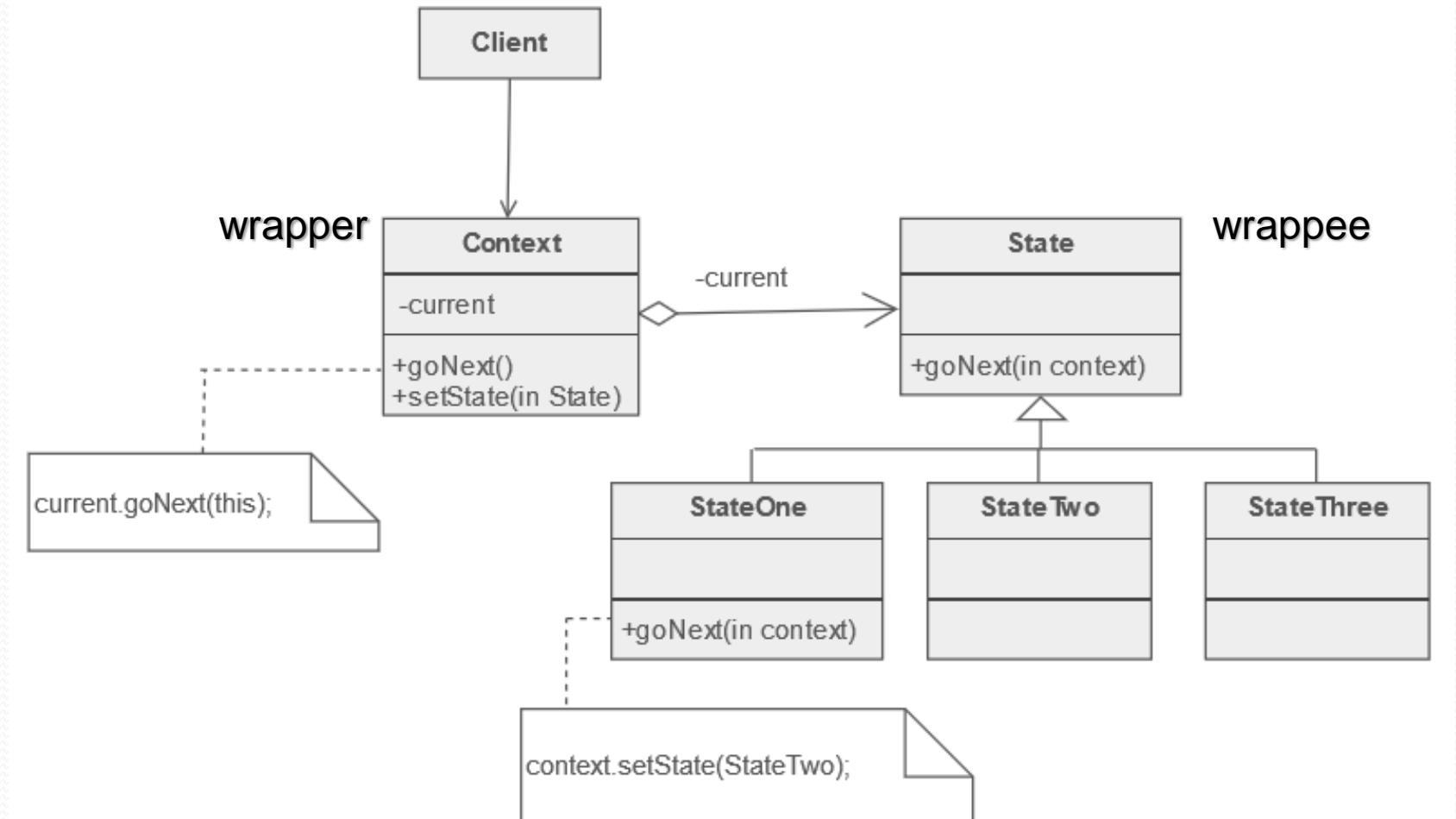
# Sate

- State: Allows an object to alter its behavior when its internal state changes. The object will appear to change its class.
- Most objects have state that changes
- State can become a prominent aspect of its behavior
- An object that can be in one of several states, with different behavior in each state
- An object-oriented state machine
- wrapper + polymorphic wrappee + collaboration

# State Design Pattern

- The State pattern is a solution to the problem of how to make behavior depend on state.
- Define a "**context**" (wrapper) class to present a single interface to the outside world.
- Define a **State** (wrappee) abstract base class.
- Represent the different "states" of the state machine as **derived classes** of the State base class.
- Define *state-specific behavior* in the appropriate State derived classes.
- Maintain a **pointer** to the current "state" in the "context" class.
- To change the state of the state machine, change the current "state" pointer.

# Structure



# Example: MP3Player

- Step 1: First we set up a context for our mp3 player.

```
public class MP3PlayerContext {  
    private State state;  
    public MP3PlayerContext(State state) {  
        this.state = state;  
    }  
    public void play() {  
        state.pressPlay(this);  
    }  
    public void setState(State state) {  
        this.state = state;  
    }  
    public State getState() {  
        return state;  
    }  
}
```

- Step 2: create state interface

```
public interface State {  
    public void pressPlay(MP3PlayerContext context);  
}
```



# Example: MP3Player

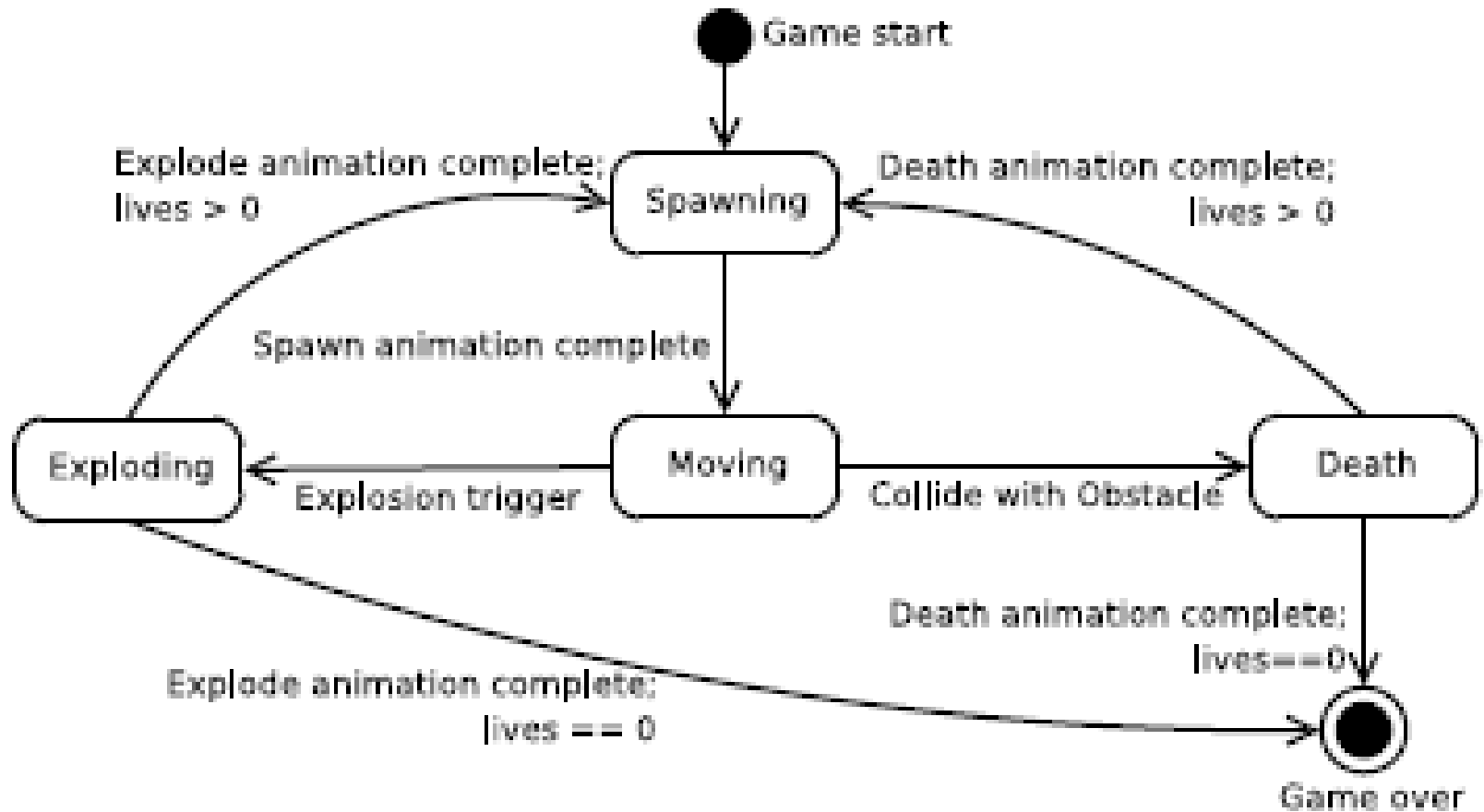
- Step3: creating a state for Standby and for Playing.

```
public class StandbyState implements State {  
    public void pressPlay(MP3PlayerContext context) {  
        context.setState(new PlayingState());  
    }  
}  
  
public class PlayingState implements State {  
    public void pressPlay(MP3PlayerContext context) {  
        context.setState(new StandbyState());  
    }  
}  
  
public class DemoApp {  
    public static void main(String[] args) {  
        StandbyState sb = new StandbyState();  
        MP3PlayerContext con = new MP3PlayerContext(sb);  
        System.out.println(con.getState().toString());  
        sb.pressPlay(con);  
        System.out.println(con.getState().toString());  
        PlayingState pl = new PlayingState();  
        pl.pressPlay(con);  
        System.out.println(con.getState().toString());  
    }  
}
```

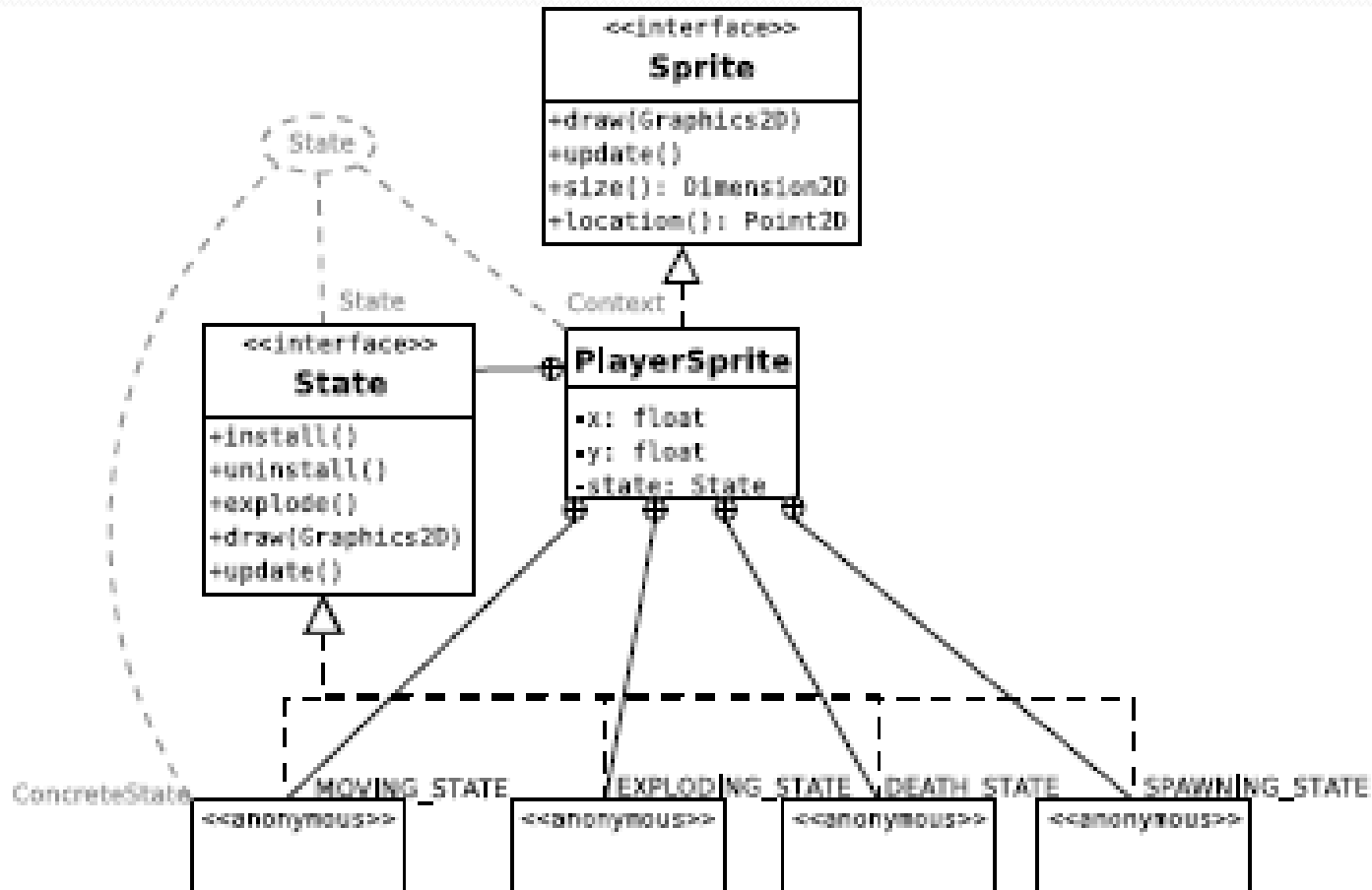
## OutPut:

```
StandbyState@7852e922  
PlayingState@4e25154f  
StandbyState@70dea4e
```

# Another Example - A game



# UML diagram of state



# Check list

- Identify an existing class, or create a new class, that will serve as the "state machine" from the client's perspective. That class is the "wrapper" class.
- Create a State base class that replicates the methods of the state machine interface. Each method takes one additional parameter: an instance of the wrapper class. The State base class specifies any useful "default" behavior.
- Create a State derived class for each domain state. These derived classes only override the methods they need to override.
- The wrapper class maintains a "current" State object.
- All client requests to the wrapper class are simply delegated to the current State object, and the wrapper object's this pointer is passed.
- The State methods change the "current" state in the wrapper object as appropriate.

# Differences with other patterns

- State objects are often Singletons.
- Flyweight explains when and how State objects can be shared.

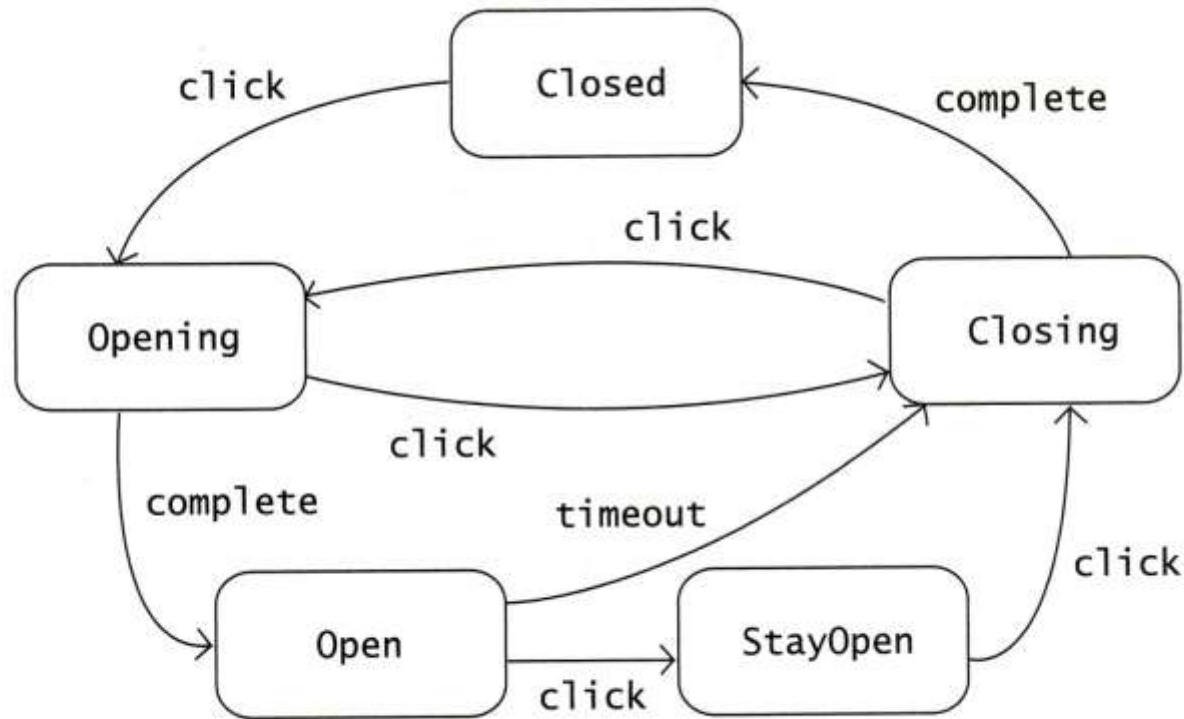
# Differences with other patterns

- Strategy has 2 different implementations, the first is similar to State. The difference is in binding times (Strategy is a bind-once pattern, whereas State is more dynamic).
- The structure of State and Bridge are identical. The two patterns use the same structure to solve different problems: State allows an object's behavior to change along with its state, while Bridge's intent is to decouple an abstraction from its implementation so that the two can vary independently.
- With Strategy, the choice of algorithm is fairly stable. With State, a change in the state of the "context" object causes it to select from its "palette" of Strategy objects.

# Another Example

- from Steve Metsker's Design Patterns Java Workbook, Addison Wesley
- Consider the state of a carousal door in a factory
  - large smart rack that accepts material through a doorway and stores material according to a bar code
  - there is a single button to operate this door
    - if closed, door begins opening
    - if opening, another click begins closing
    - once open, 2 seconds later (timeout), the door begins closing
      - can be prevented by clicking after open state and before timeout begins
- These state changes can be represented by a state machine (next slide)

# A UML State Diagram





# Things to do

- Define a “context” class to present a single interface
- Define a State abstract base class.
- Represent different “states” of the state machine as derived classes of the State base class
- Define state-specific behavior in the appropriate State derived classes (see code demo that changes state, from Opening to Closing or Closing to Opening for example)
- Maintain a reference to the current “state” in the “context” class
- To change the state of the state machine, change the current “state” reference

# Code reverse engineered (demo)

