State Design Pattern

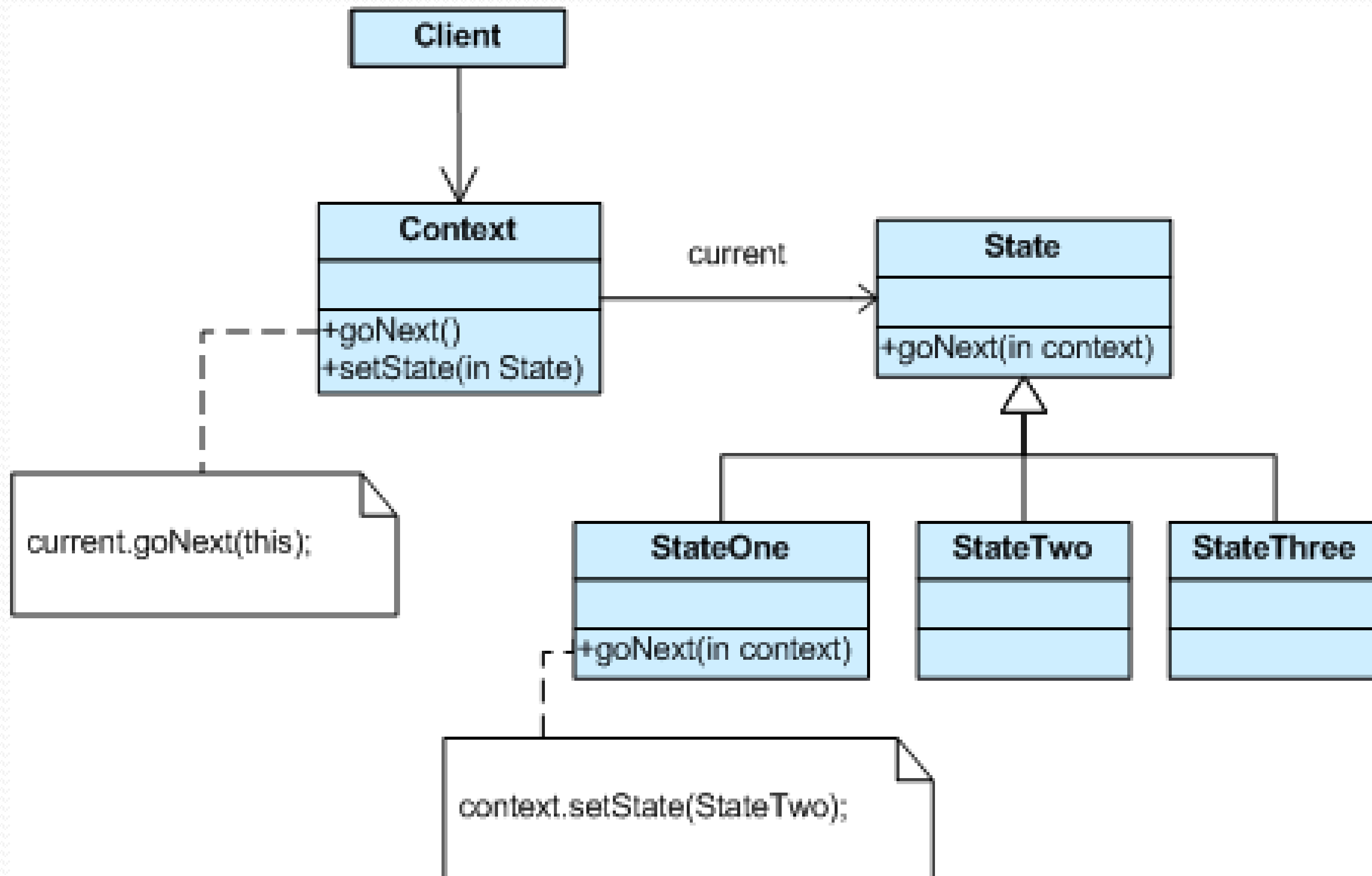
Pattern Description

- Behavioral Design pattern, aka Objects for States, used to represent object state and its behavior.
- Used to model the dynamic nature of a system.
- It is the sequences of states an object goes through in response to events during its lifetime.
- States of objects are a period of time which the object satisfies a condition, performs an activity or waits for an event.
- Allows objects to alter behavior at runtime when their internal state changes.
- Provides an efficient and clean way to modify an object’s State and its behavior.
- State machines are used to model behavior of any class, use case or an entire system using a state diagram.

Related Patterns

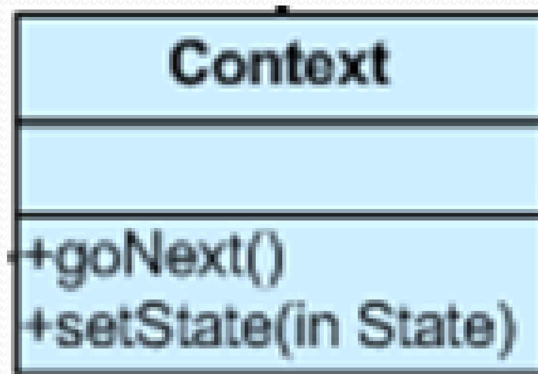
- Often is a Singleton
- Similar to the Strategy implementation,
  - Builds upon Strategy but differs in intent
  - Strategy is a bind-once pattern, State is dynamic
- Almost identical in structure to the Bridge

Generic Example



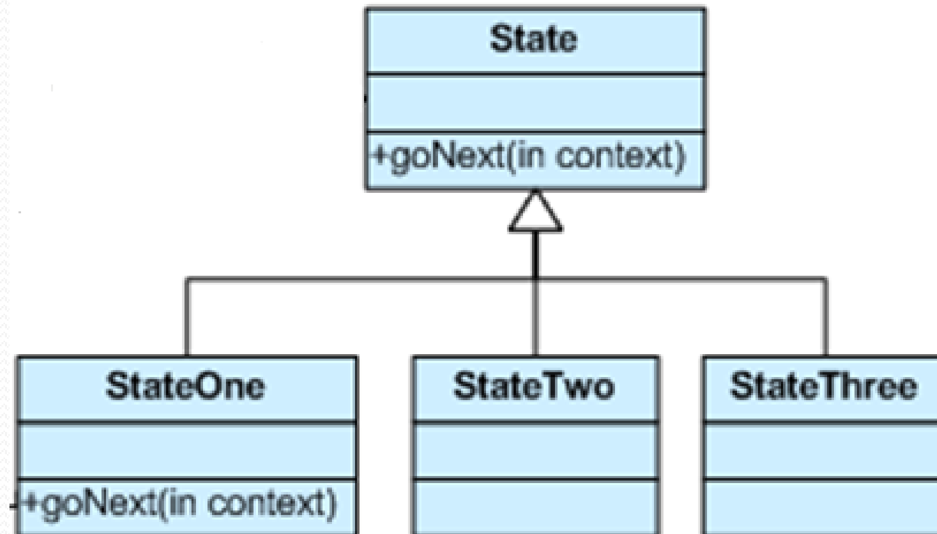
Define Context Class

- Define the context class that is used to interface with the State class.
- Context class points to the current state
- The state is changed by changing what state the Context state points to.



Define Base and Concrete Classes

- Next define the State abstract base class and the derived classes.
- Each derived class represents the “states” and contains the behavior that pertains to that “state”.

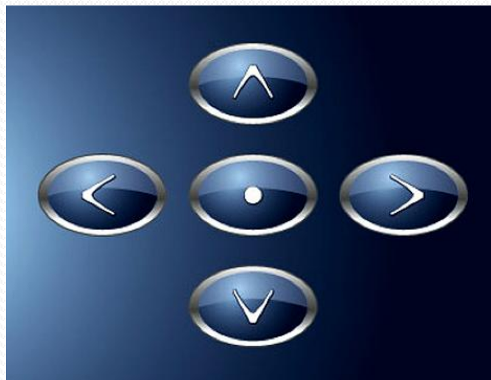


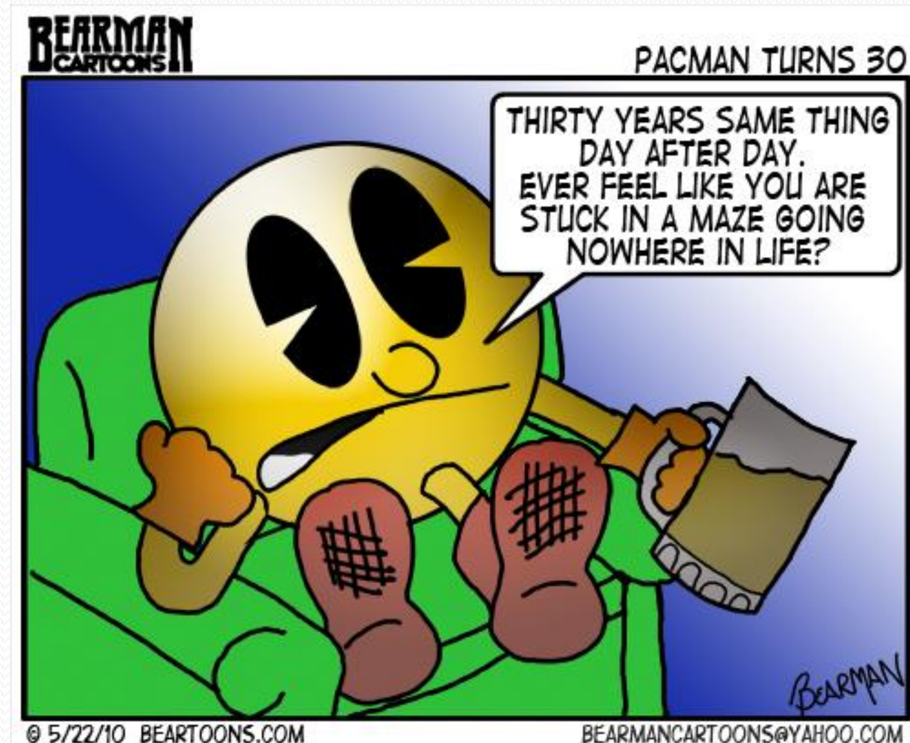
My Implementation – Control a Character

- Implement a Game Control pad.
- Client has not provided input yet, the Context is "Waiting for Input".
- Client presses a button and interacts with the Context class instructing it to change states.
- Context Class changes the state of our Character to reflect the state we would like to change to.
  - Up
  - Down
  - Left
  - Right
  - Power Up - Center button is a "Power Up" state.
- Each state performs its behavior and will return the Character back to its default state of "Waiting for Input".
- What "Character" should we use.

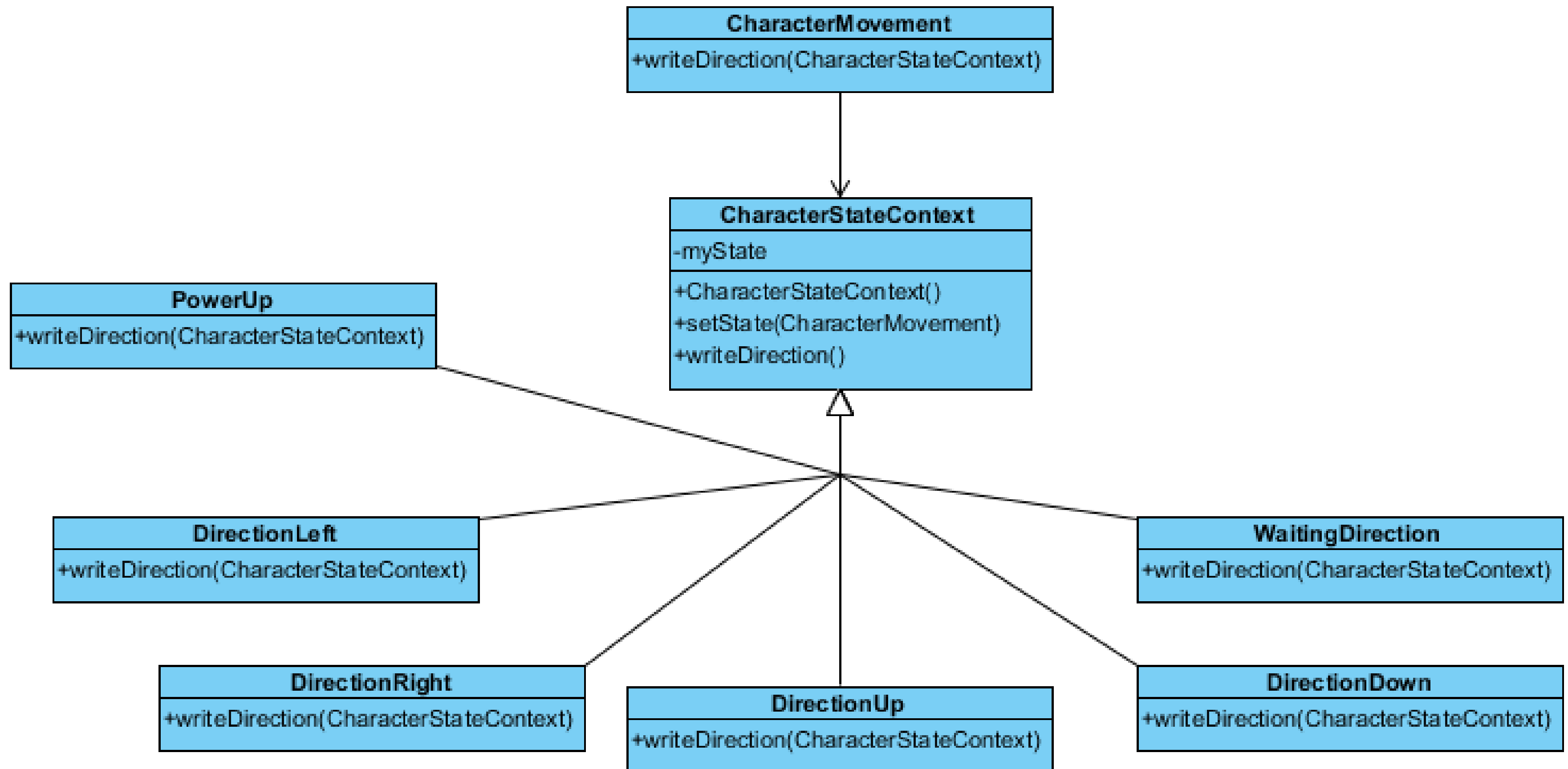
Implementation cont'd

- Pacman is in a sad "*state*" of affairs
- Lets use a State Machine to add direction to his life.

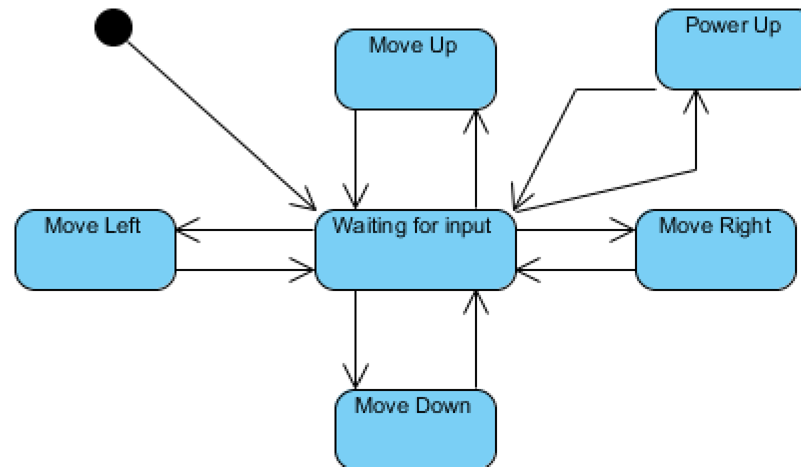




UML Class Diagram



State Diagram



Class: ControlCharacter

```
public class ControlCharacter {
  public static void main(String[] args)  {
    CharacterStateContext myPacMan = new CharacterStateContext();

    System.out.println("\nWelcome to Pacman's life.");
    myPacMan.writeDirection();

    while(true) {
      String inputMovement = System.console().readLine("Please input direction for your life: ");

      if(inputMovement.equals("l")){
        myPacMan.setState(new DirectionLeft());
      }
      else if(inputMovement.equals("r")) {
        myPacMan.setState(new DirectionRight());
      }
      else if(inputMovement.equals("u")){
        myPacMan.setState(new DirectionUp());
      }
      else if(inputMovement.equals("d")){
        myPacMan.setState(new DirectionDown());
      }
      else if(inputMovement.equals("p")){
        myPacMan.setState(new PowerUp());
      }
      else if(inputMovement.equals("e")){
        break;
      }
      else {
        System.out.println("\nUnkwon Command, stop drinking and clear your head.\n");
      }

      myPacMan.writeDirection();
    }
  }
}
```

Class: CharacterStateContext

```
public class CharacterStateContext {
  private CharacterMovement myState;

  public CharacterStateContext()  {
    setState(new WaitingDirection());
  }

  public void setState(CharacterMovement newState)  {
    this.myState = newState;
  }

  public void writeDirection()  {
    this.myState.writeDirection(this);
  }
}
```

Interface and State classes

- Note: More subclasses defined in the code files

```
interface CharacterMovement {
  public void writeDirection(CharacterStateContext stateContext);
}

class WaitingDirection implements CharacterMovement {
  public void writeDirection(CharacterStateContext stateContext)  {
    System.out.println("\nHelp PacMan find direction in life.\n l-Left, r-Right,
  }
}

class DirectionLeft implements CharacterMovement {
  public void writeDirection(CharacterStateContext stateContext)  {
    System.out.println("\nMoved one space LEFT, make another Season of TV.\n");
    stateContext.setState(new WaitingDirection());
  }
}

class DirectionRight implements CharacterMovement {
  public void writeDirection(CharacterStateContext stateContext)  {
    System.out.println("\nMoved one space RIGHT, do movies for spare $$$.\n");
    stateContext.setState(new WaitingDirection());
  }
}
```

Output from the Code

```
Welcome to Pacman's life.

Help PacMan find direction in life.
 l-Left, r-Right, u-Up, d-Down, p-PowerUp!

Please input direction for your life: l

Moved one space LEFT, make another Season of TV.

Please input direction for your life: r

Moved one space RIGHT, do movies for spare $$$.

Please input direction for your life: u

Moved UP, go to Vegas and trash hotel rooms!.

Please input direction for your life: d

Moved DOWN, lose hit TV show and goddesses.

Please input direction for your life: p

Charlie Sheen POWER UP! Winning!

Please input direction for your life: g

Unkwon Command, stop drinking and clear your head.


Help PacMan find direction in life.
 l-Left, r-Right, u-Up, d-Down, p-PowerUp!

Please input direction for your life: e
```

State Machines Give our Objects Direction in Life

- Winning!

Sources

- Booch, Grady, James Rumbaugh, and Ivar Jacobson. The Unified Modeling Language User Guide. Upper Saddle River, NJ: Addison-Wesley, 2005. Print.
- "State Design Pattern." Design Patterns and Refactoring. Web. 05 Mar. 2011. <http://sourcemaking.com/design_patterns/state>.
- "State Pattern." Wikipedia, the Free Encyclopedia. Web. 05 Mar. 2011. <http://en.wikipedia.org/wiki/State_pattern>.