Design Patterns in Ruby: State Pattern

Abram Hindle
abram.hindle@ualberta.ca
Department of Computing Science
University of Alberta
http://softwareprocess.es/

Design Patterns

- *Just because you have duck-typing doesn't mean you can ignore common OO idioms!*
- Design patterns communicate intent, so it is best if we have a similar understanding.
- OO is hard :(

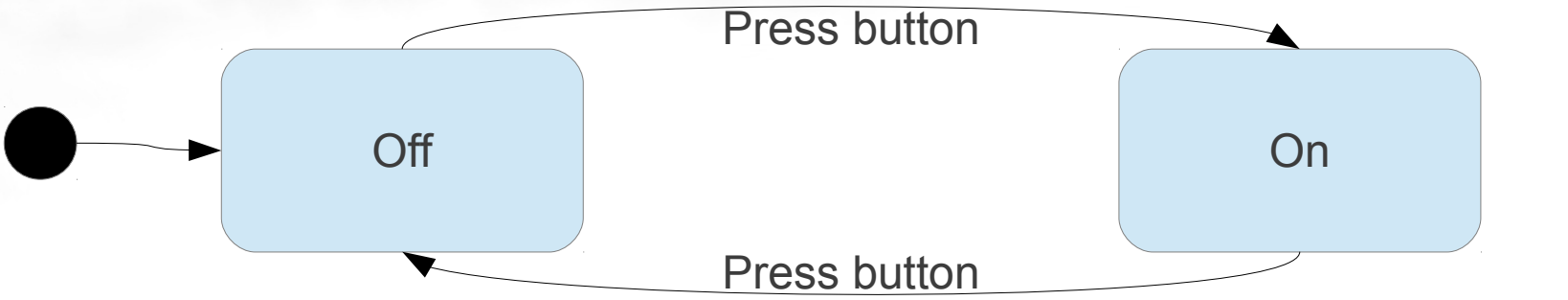
State

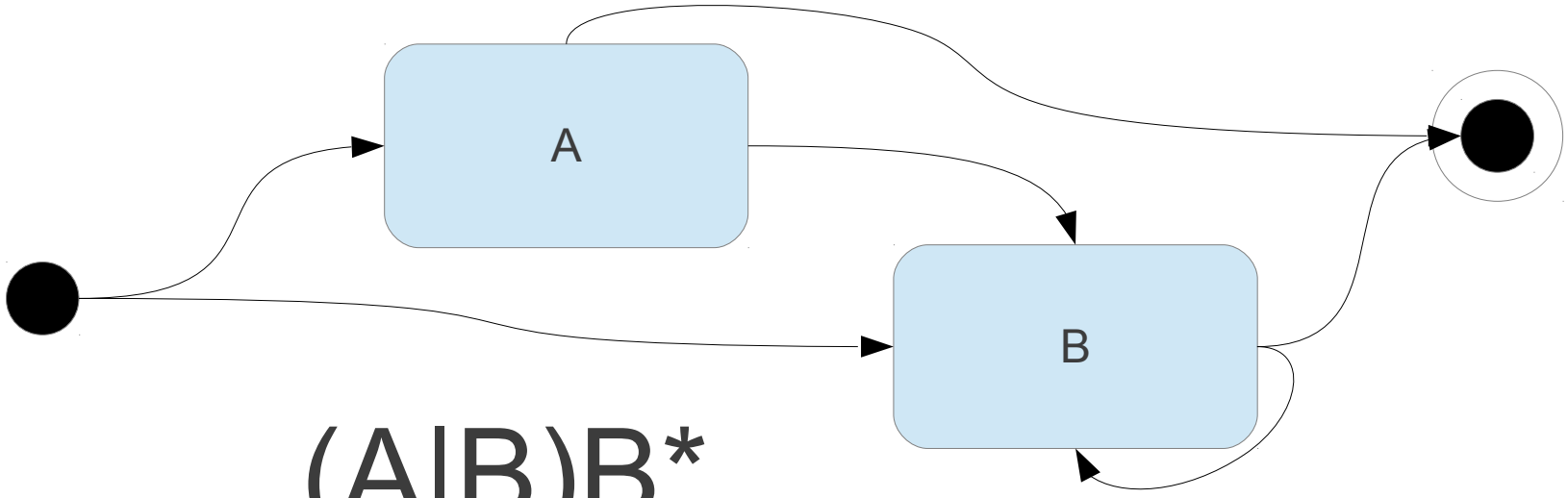
- State is often the current snapshot of values in a system.
- State machines are a modelling methodology that try to simplify complicate control flows and protocols
- State is often encoded as attributes and variables.
- State Machines are often implemented imperatively.

State Pattern

- Intent: Change an object's behaviour dynamically (runtime).
- Intent: Model a state-machine following OO rules.
- Avoid cluttering a class with all the different behaviours, avoid subclassing for changing behaviour.
- Use the type system to ensure state transition safety.

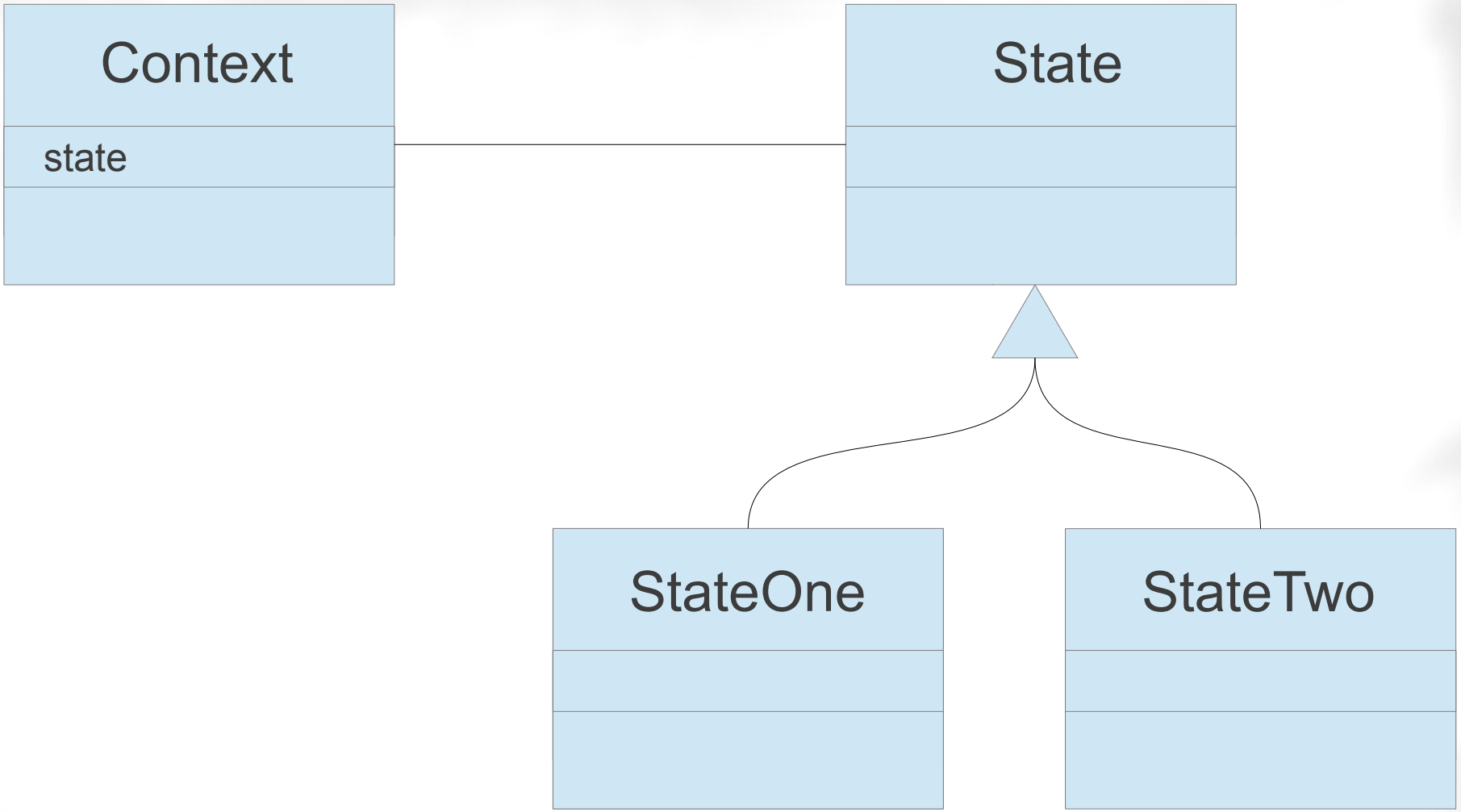
State Machines?



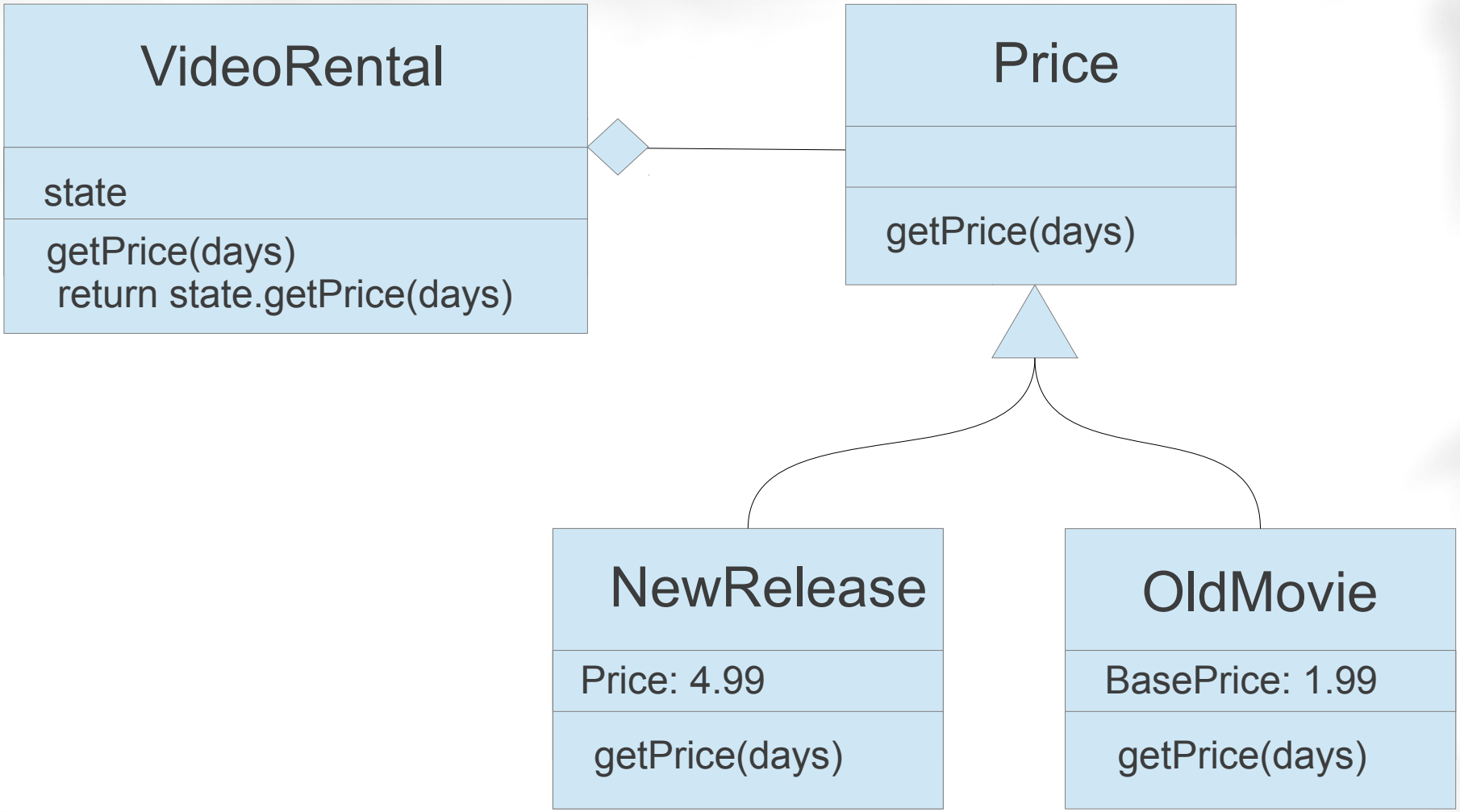


(A|B)B*

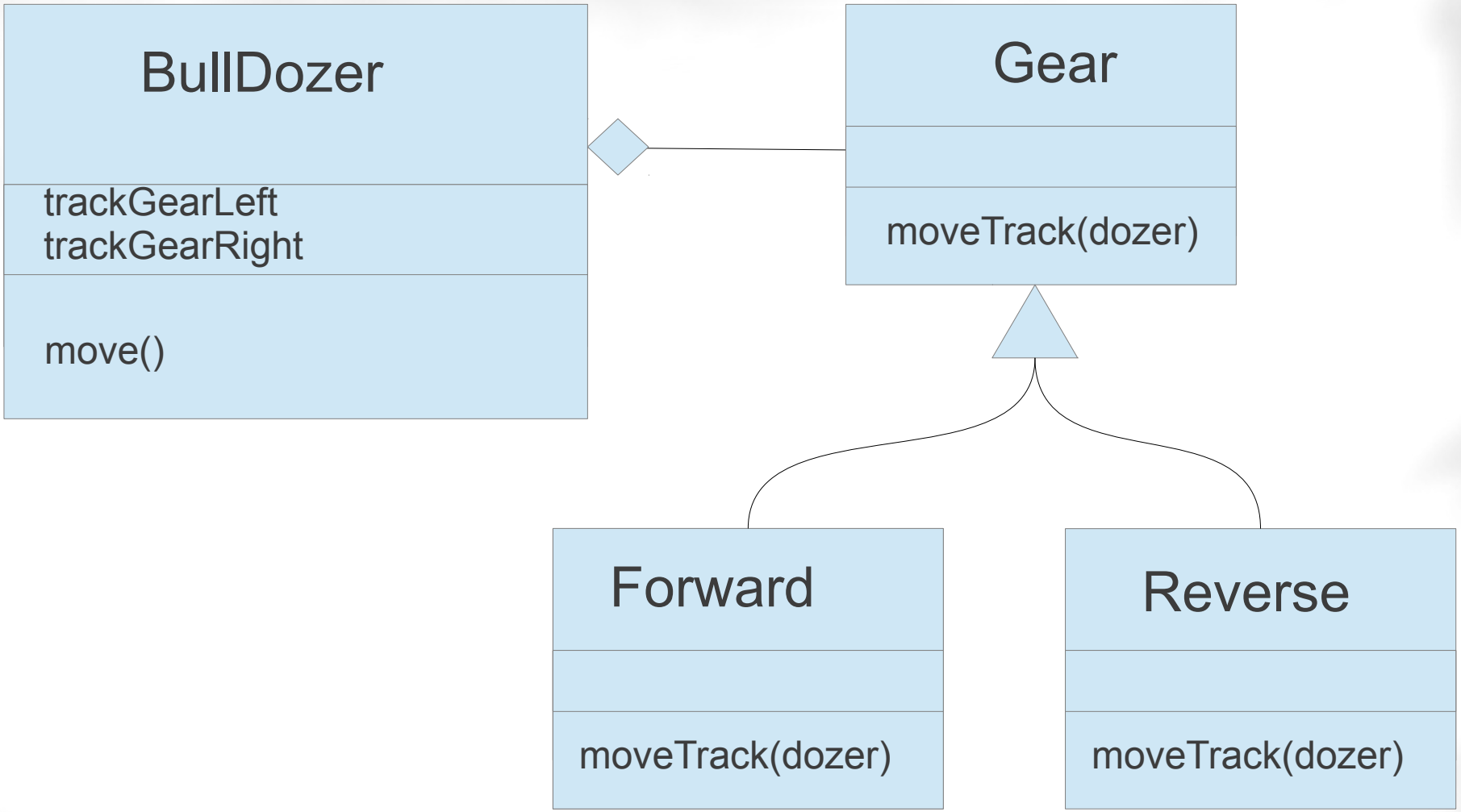
State Pattern UML (Abstract)



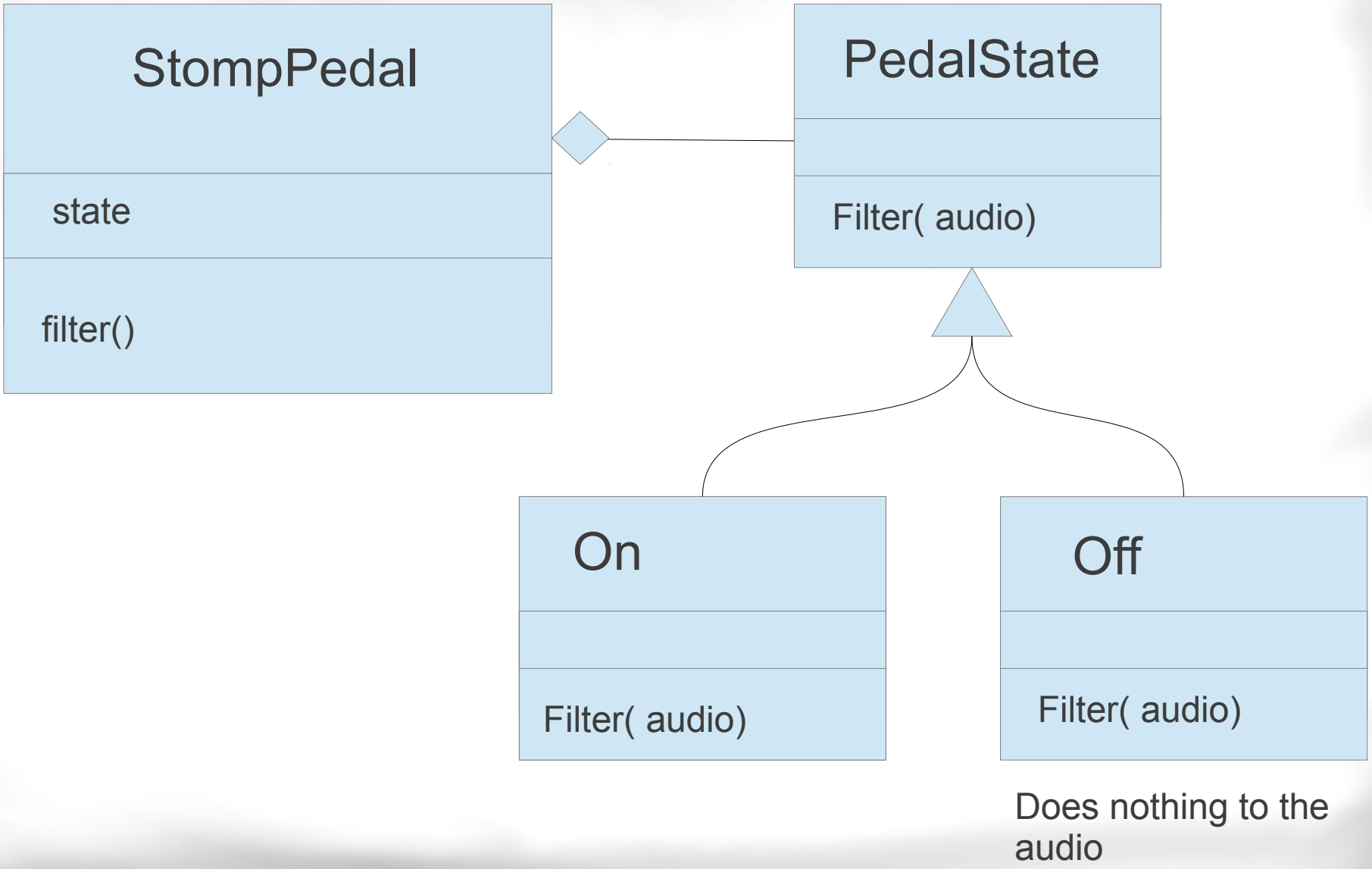
State Pattern UML (Concrete)



State Pattern UML (Concrete)



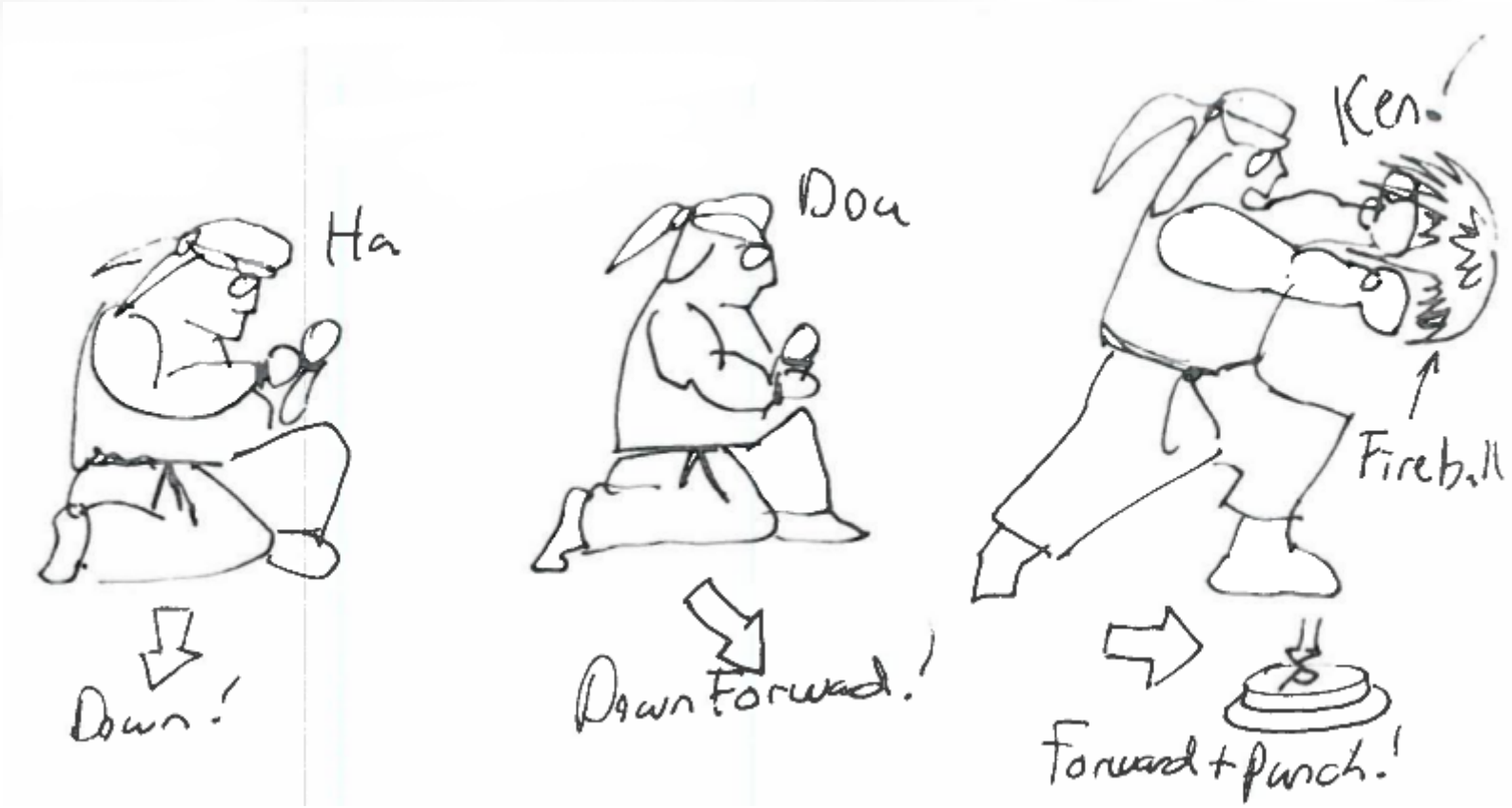
State Pattern UML (Concrete)



State Method Example

Remember Street Fighter II special moves?

SFII: HaDouKen Fireball



DOWN

DOWN-FORWARD

FORWARD + PUNCH

Our Player: Ryu

```
class RyuPlayer
 def initialize()
  @moves = [WhirlwindKickMove.new(), FireballMove.new(), DragonUppercutMove.new()]
  @todo = []
 end
 def execute( move )
  @todo = [ move ]
 end
 def move( move, action )
  @todo = []
  @moves = @moves.map {|specialMove| specialMove.nextState( self, move, action ) }
  executeMove()
 end
 def executeMove()
  if (@todo.length > 0)
   puts(@todo[0].say())
  end
 end
end
```

We run 3 seperate state machines for 3 special moves.

The state machines RETURN the next state, and thus we must replace the states when we execute a new move.

Fireball Without State Pattern

```
class FireballMoveConditional
  def initialize()
    @state = :none
  end
  def say()
    return "HaDoKen!"
  end
  # this example has a
  # state value as a guard
  # but if I mess up the
  # conditional then all
  # hell breaks loose

  def nextState( context, move, action )
    if    (@state == :none && move.down()) then
      @state = :ha
    elsif (@state == :ha && move.down()) then
      @state = :ha
    elsif (@state == :ha && move.downForward()) then
      @state = :do
    elsif (@state == :do && move.forward() && action.punch()) then
      # the body of the state machine is put into these conditional blocks
      context.execute(FireballMove.new())
      @state = :none
    else
      # the general default behaviour is here but sometimes different
      # states have default behaviour too
      @state = :none
    end
    return self
  end
end
```

FireBall with State Pattern

```
class FireballMove
 def say()
  return "HaDoKen!"
 end
 def nextState( context, move, action )
  if (move.down()) then
   return FireballHa.new()
  end
  return FireballMove.new()
 end
end
```

```
class FireballDo
 def nextState( context, move, action )
  if (move.forward() && action.punch()) then
   context.execute(FireballMove.new())
  end
  return FireballMove.new()
 end
end
```

```
class FireballHa
 def nextState( context, move, action )
  if (move.downForward()) then
   return FireballDo.new()
  elsif (move.down()) then
   return FireballHa.new()
  end
  return FireballMove.new()
 end
end
```

- State encoded as a Class
- Only relevant triggers for that state need to be handled.
- Code is better organized.
- Responsibilities seperated.
- No crazy conditionals
- Precedence or order of conditionals is explicit and clear.

Dragon Uppercut State Machine

```
class DragonUppercutMove
 def say()
  return "ShoRyuKen!"
 end
 def nextState( context, move, action )
  if (move.forward()) then
   return DragonUppercutSho.new()
  end
  return DragonUppercutMove.new()
 end
end
```

```
class DragonUppercutSho
 def nextState( context, move, action )
  if (move.downBackward()) then
   return DragonUppercutRyu.new()
  end
  return DragonUppercutMove.new()
 end
end
```

```
class DragonUppercutRyu
 def nextState( context, move, action )
  if (move.forward() && action.punch()) then
   context.execute(DragonUppercutMove.new())
  end
  return DragonUppercutMove.new()
 end
end
```

Whirlwind Kick State Machine

```
class WhirlwindKickMove
 def say()
  return "TaTsunaki!"
 end
 def nextState( context, move, action )
  if (move.up()) then
   return WhirlwindKickTa.new()
  end
  return WhirlwindKickMove.new()
 end
end
```

```
class WhirlwindKickTa
 def nextState( context, move, action )
  if (move.down()) then
   return WhirlwindKickTsun.new()
  end
  return WhirlwindKickMove.new()
 end
end
```

```
class WhirlwindKickTsun
 def nextState( context, move, action )
  if (move.downBackward()) then
   return WhirlwindKickAki.new()
  end
  return WhirlwindKickMove.new()
 end
end
```

```
class WhirlwindKickAki
 def nextState( context, move, action )
  if (move.backward() && action.kick()) then
   context.execute(WhirlwindKickMove.new())
  end
  return WhirlwindKickMove.new()
 end
end
```

Trying Some Moves Out!

```
player = RyuPlayer.new()
puts("# crouching")
player.move( Down.new(), Action.new() )
player.move( Down.new(), Action.new() )
player.move( Down.new(), Action.new() )
puts("# starting fireball")
player.move( Down.new(), Action.new() )
player.move( DownForward.new(), Action.new() )
player.move( Forward.new(), Punch.new() ) #fireball
puts("# converting to dragon uppercut")
player.move( DownBackward.new(), Action.new() )
player.move( Forward.new(), Punch.new() ) #DragonUppercut
puts("# charging whirlwind kick")
player.move( Up.new(), Action.new() )
player.move( Down.new(), Action.new() )
player.move( DownBackward.new(), Action.new() )
player.move( Backward.new(), Kick.new() ) #whirlwind
```

State Pattern Conclusions

- Use 1: Dynamic Change/Delegation of Behaviour
- Use 2: Type-Safe State Machines
- State Pattern is quite relevant to Ruby
- State can be treated as a role or responsibility and thus belongs in its own class.
- Handling State via conditionals will bite your butt very quickly
- Running multiple state machines is easier than unioning/flattening out state machines.

State Pattern Resources

- C2 on State Pattern http://c2.com/cgi/wiki?StatePattern
- Wikipedia on State Pattern http://en.wikipedia.org/wiki/State_pattern
- Source Making on State pattern http://sourcemaking.com/design_patterns/state
- State Pattern Gem http://rubygems.org/gems/state_pattern

  https://github.com/dcadenas/state_pattern
- The Delegate module can be used to implement
- State pattern http://www.ruby-doc.org/stdlib-1.9.3/libdoc/delegate/rdoc/Delegator.ht and http://www.ruby-doc.org/docs/ProgrammingRuby/html/lib_patterns.html
-