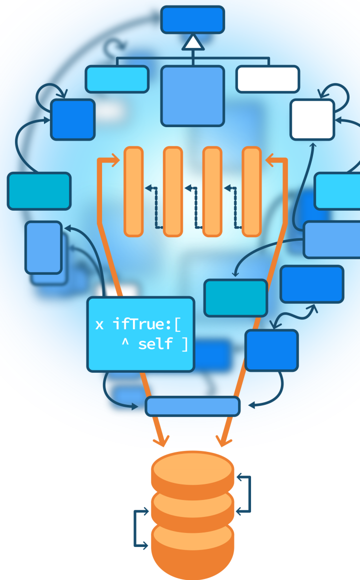
About state Design Pattern

S.Ducasse, L. Fabresse, G. Polito, and P. Tesone

http://www.pharo.org



Goals

- Motivating Example
- Representing Different States as Objects
- Operations and State Transitions are Encapsulated by each state
- Handling Instance State

Analysing a case

- Imagine an automatic coffee machine.
- It has different states:
  - Waiting for Order
  - Waiting for Payment
  - Making Coffee
  - Coffee Ready

Our States

start → Waiting for Order (Idle State)

Waiting for Order (Idle State) → acceptOrder → Waiting for Payment (ToPay State)

Waiting for Payment (ToPay State) → Pay → Making Coffee (Making State)

Making Coffee (Making State) → coffeeDone → Coffee Ready (DrinkReady State)

Coffee Ready (DrinkReady State) → takeCoffee → Waiting for Order (Idle State)

Our Operations

```
CoffeeMachine >> acceptOrder: anOrder
CoffeeMachine >> howMuchIsIt
CoffeeMachine >> pay: someMoney
CoffeeMachine >> coffeeDone
CoffeeMachine >> takeCoffee
```

Our Operations

- The available operations depend on the current state
- We need to add a lot of conditional code

```
CoffeeMachine >> acceptOrder: anOrder
  "Checking state every time..."
  machineState = #idle ifFalse: [ self error: 'Machine working...'].
  "Changing state in each operation"
  machineState := #toPay.
  "... Do the magic to order a coffee..."
```

Repeated Pattern

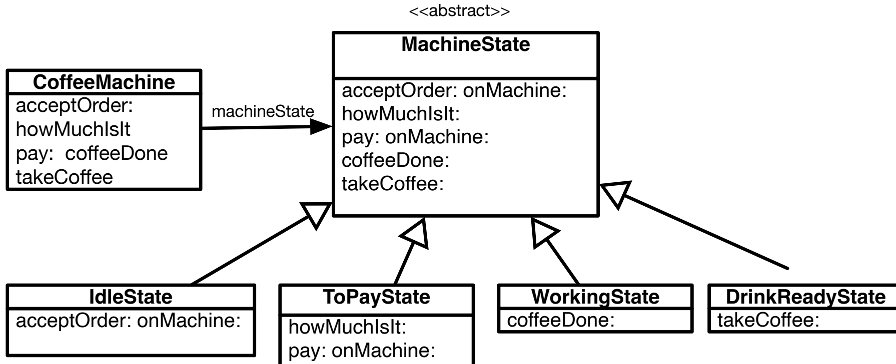
```
CoffeeMachine >> howMuchIsIt
  "Checking state every time..."
  machineState = #toPay ifFalse: [ self error: 'Invalid State'].

  "Some operations do not change state"

  ^ ourPrice
```

Proposed Idea

- Represent each state as an object
- We delegate the operations to the state



Our new Operations (1/2)

```
CoffeeMachine >> acceptOrder: anOrder
  ^ machineState acceptOrder: anOrder onMachine: self
```

```
MachineState >> acceptOrder: anOrder onMachine: aMachine
  ^ self error: 'Invalid State'
```

```
IdleState >> acceptOrder: anOrder onMachine: aMachine
  "The operation code"
  aMachine doTheMagicToOrder: anOrder
  "To the new State"
  aMachine machineState: ToPayState new.
```

Our new Operations (2/2)

```
CoffeeMachine >> howMuchIsIt
  ^ machineState howMuchIsIt: self
```

```
MachineState >> howMuchIsIt: aMachine
  ^ self error: 'Invalid State'
```

```
IdleState >> howMuchIsIt: aMachine
  ^ aMachine ourPrice
```

Advantages

- Each state just implements its operations
- State transitions are implemented in each state
- Less conditional code
- Elegant solution when having many states

Where to Put the instance state? (1/3)

- Instance State as:
  - Selected coffee
  - Price
- We can put the machine instance state in:
  - The Machine Object
  - In the State object

Where to Put the instance state? (2/3)

- In the Machine Object:
  - Useful if the internal state is the same for all the machine states
  - We don’t need to copy on every state change
  - Bad if each state has different instance variables

Where to Put the instance state? (3/3)

- In the State Object:
  - Useful if the internal state is different for all the machine states
  - Each state object has direct access to the instance state, we don't need accessors
  - Creating a state requires passing all instance variables that it stores

Conclusion

State pattern:

- Is useful when we have an object with many states
- Encapsulates the operations and the state transitions
- Uses delegation instead of conditional code
- It is easy to add new states and operations
- It is a more complex solution, we need to trade off the new complexity vs clarity/flexibility

Produced as part of the course on http://www.fun-mooc.fr

Advanced Object-Oriented Design and Development with Pharo

A course by
S.Ducasse, L. Fabresse, G. Polito, and P. Tesone

Inria
LearningLab

Except where otherwise noted, this work is licensed under CC BY-NC-ND 3.0 France
https://creativecommons.org/licenses/by-nc-nd/3.0/fr/

2023