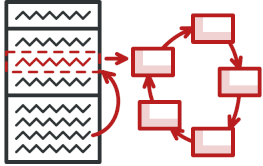





[Home](#) / [Design Patterns](#) / [State](#) / [Go](#)



State in Go

State is a behavioral design pattern that allows an object to change the behavior when its internal state changes.

The pattern extracts state-related behaviors into separate state classes and forces the original object to delegate the work to an instance of these classes, instead of acting on its own.

 [Learn more about State →](#)

Navigation

 [Intro](#)

 [Conceptual Example](#)

 [vendingMachine](#)

 [state](#)

 [noItemState](#)

 [hasItemState](#)

[Home](#) [Refactoring](#) [Design Patterns](#) [Premium Content](#)
[Forum](#) [Contact us](#)



Conceptual Example

Let's apply the State Design Pattern in the context of vending machines. For simplicity, let's assume that vending machine only has one type of item or product. Also for simplicity let's assume that a vending machine can be in 4 different states:

- hasItem
- noItem
- itemRequested
- hasMoney

A vending machine will also have different actions. Again for simplicity let's assume that there are only four actions:

- Select the item
- Add the item
- Insert money
- Dispense item

The State design pattern should be used when the object can be in many different states and depending upon incoming request the object needs to change its current state.

In our example, a vending machine can be in many different states and these states will constantly switch from one to another. Let's say vending machine is in `itemRequested`. Once the action "Insert Money" occurs, the machine moves to `hasMoney` state.

Depending on its current state, the machine can behave differently to the same requests. For example, if user wants to purchase an item then the machine will proceed if it's in `hasItemState` or it will reject if it's in `noItemState`.

The code of the vending machine is not polluted with this logics, all the state-dependent code lives in respective state implementations.

vendingMachine.go: Context

```
package main
```

```

import "fmt"

type VendingMachine struct {
    hasItem      State
    itemRequested State
    hasMoney     State
    noItem       State

    currentState State

    itemCount int
    itemPrice int
}

func newVendingMachine(itemCount, itemPrice int) *VendingMachine {
    v := &VendingMachine{
        itemCount: itemCount,
        itemPrice: itemPrice,
    }
    hasItemState := &HasItemState{
        vendingMachine: v,
    }
    itemRequestedState := &ItemRequestedState{
        vendingMachine: v,
    }
    hasMoneyState := &HasMoneyState{
        vendingMachine: v,
    }
    noItemState := &NoItemState{
        vendingMachine: v,
    }

    v.setState(hasItemState)
    v.hasItem = hasItemState
    v.itemRequested = itemRequestedState
    v.hasMoney = hasMoneyState
    v.noItem = noItemState
    return v
}

func (v *VendingMachine) requestItem() error {
    return v.currentState.requestItem()
}

func (v *VendingMachine) addItem(count int) error {
    return v.currentState.addItem(count)
}

```

```

func (v *VendingMachine) insertMoney(money int) error {
    return v.currentState.insertMoney(money)
}

func (v *VendingMachine) dispenseItem() error {
    return v.currentState.dispenseItem()
}

func (v *VendingMachine) setState(s State) {
    v.currentState = s
}

func (v *VendingMachine) incrementItemCount(count int) {
    fmt.Printf("Adding %d items\n", count)
    v.itemCount = v.itemCount + count
}

```

state.go: State interface

```

package main

type State interface {
    addItem(int) error
    requestItem() error
    insertMoney(money int) error
    dispenseItem() error
}

```

noltemState.go: Concrete state

```

package main

import "fmt"

type NoItemState struct {
    vendingMachine *VendingMachine
}

func (i *NoItemState) requestItem() error {
    return fmt.Errorf("Item out of stock")
}

```

```

}

func (i *NoItemState) addItem(count int) error {
    i.vendingMachine.incrementItemCount(count)
    i.vendingMachine.setState(i.vendingMachine.hasItem)
    return nil
}

func (i *NoItemState) insertMoney(money int) error {
    return fmt.Errorf("Item out of stock")
}

func (i *NoItemState) dispenseItem() error {
    return fmt.Errorf("Item out of stock")
}

```

hasItemState.go: Concrete state

```

package main

import "fmt"

type HasItemState struct {
    vendingMachine *VendingMachine
}

func (i *HasItemState) requestItem() error {
    if i.vendingMachine.itemCount == 0 {
        i.vendingMachine.setState(i.vendingMachine.noItem)
        return fmt.Errorf("No item present")
    }
    fmt.Printf("Item requestd\n")
    i.vendingMachine.setState(i.vendingMachine.itemRequested)
    return nil
}

func (i *HasItemState) addItem(count int) error {
    fmt.Printf("%d items added\n", count)
    i.vendingMachine.incrementItemCount(count)
    return nil
}

func (i *HasItemState) insertMoney(money int) error {
    return fmt.Errorf("Please select item first")
}

```

```
func (i *HasItemState) dispenseItem() error {
    return fmt.Errorf("Please select item first")
}
```

itemRequestedState.go: Concrete state

```
package main

import "fmt"

type ItemRequestedState struct {
    vendingMachine *VendingMachine
}

func (i *ItemRequestedState) requestItem() error {
    return fmt.Errorf("Item already requested")
}

func (i *ItemRequestedState) addItem(count int) error {
    return fmt.Errorf("Item Dispense in progress")
}

func (i *ItemRequestedState) insertMoney(money int) error {
    if money < i.vendingMachine.itemPrice {
        return fmt.Errorf("Inserted money is less. Please insert %d", i.vendingMachine.itemPrice)
    }
    fmt.Println("Money entered is ok")
    i.vendingMachine.setState(i.vendingMachine.hasMoney)
    return nil
}

func (i *ItemRequestedState) dispenseItem() error {
    return fmt.Errorf("Please insert money first")
}
```

hasMoneyState.go: Concrete state

```
package main

import "fmt"

type HasMoneyState struct {
```

```

    vendingMachine *VendingMachine
}

func (i *HasMoneyState) requestItem() error {
    return fmt.Errorf("Item dispense in progress")
}

func (i *HasMoneyState) addItem(count int) error {
    return fmt.Errorf("Item dispense in progress")
}

func (i *HasMoneyState) insertMoney(money int) error {
    return fmt.Errorf("Item out of stock")
}

func (i *HasMoneyState) dispenseItem() error {
    fmt.Println("Dispensing Item")
    i.vendingMachine.itemCount = i.vendingMachine.itemCount - 1
    if i.vendingMachine.itemCount == 0 {
        i.vendingMachine.setState(i.vendingMachine.noItem)
    } else {
        i.vendingMachine.setState(i.vendingMachine.hasItem)
    }
    return nil
}

```

main.go: Client code

```

package main

import (
    "fmt"
    "log"
)

func main() {
    vendingMachine := newVendingMachine(1, 10)

    err := vendingMachine.requestItem()
    if err != nil {
        log.Fatalf(err.Error())
    }

    err = vendingMachine.insertMoney(10)
    if err != nil {

```

```

        log.Fatalf(err.Error())
    }

    err = vendingMachine.dispenseItem()
    if err != nil {
        log.Fatalf(err.Error())
    }

    fmt.Println()

    err = vendingMachine.addItem(2)
    if err != nil {
        log.Fatalf(err.Error())
    }

    fmt.Println()

    err = vendingMachine.requestItem()
    if err != nil {
        log.Fatalf(err.Error())
    }

    err = vendingMachine.insertMoney(10)
    if err != nil {
        log.Fatalf(err.Error())
    }

    err = vendingMachine.dispenseItem()
    if err != nil {
        log.Fatalf(err.Error())
    }
}

```

 **output.txt: Execution result**

Item requestd
Money entered is ok
Dispensing Item

Adding 2 items

Item requestd
Money entered is ok
Dispensing Item

*Based on: **Golang By Example***

RETURN

READ NEXT

← Observer in Go

Strategy in Go →

State in Other Languages

