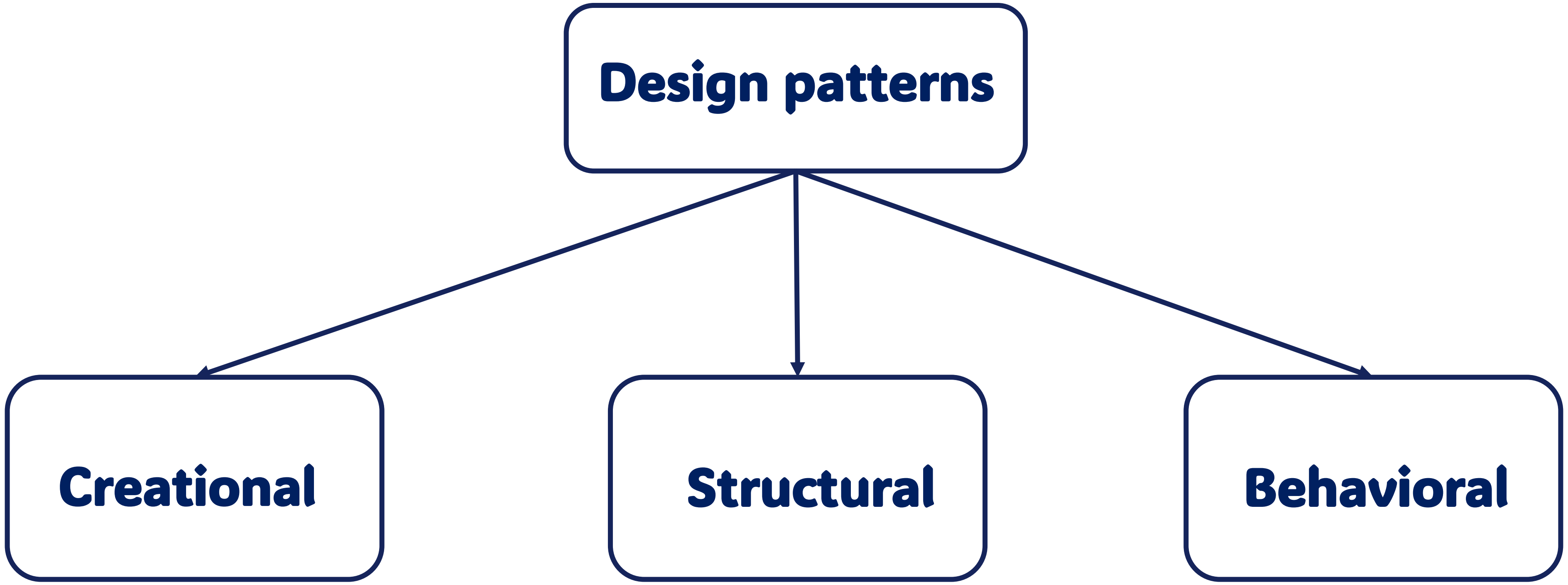
الجامعة السورية الخاصة
SYRIAN PRIVATE UNIVERSITY

Software system design – practical

Lecture 05 – state design patterns

Eng. Raghad al-hossny

Design patterns:

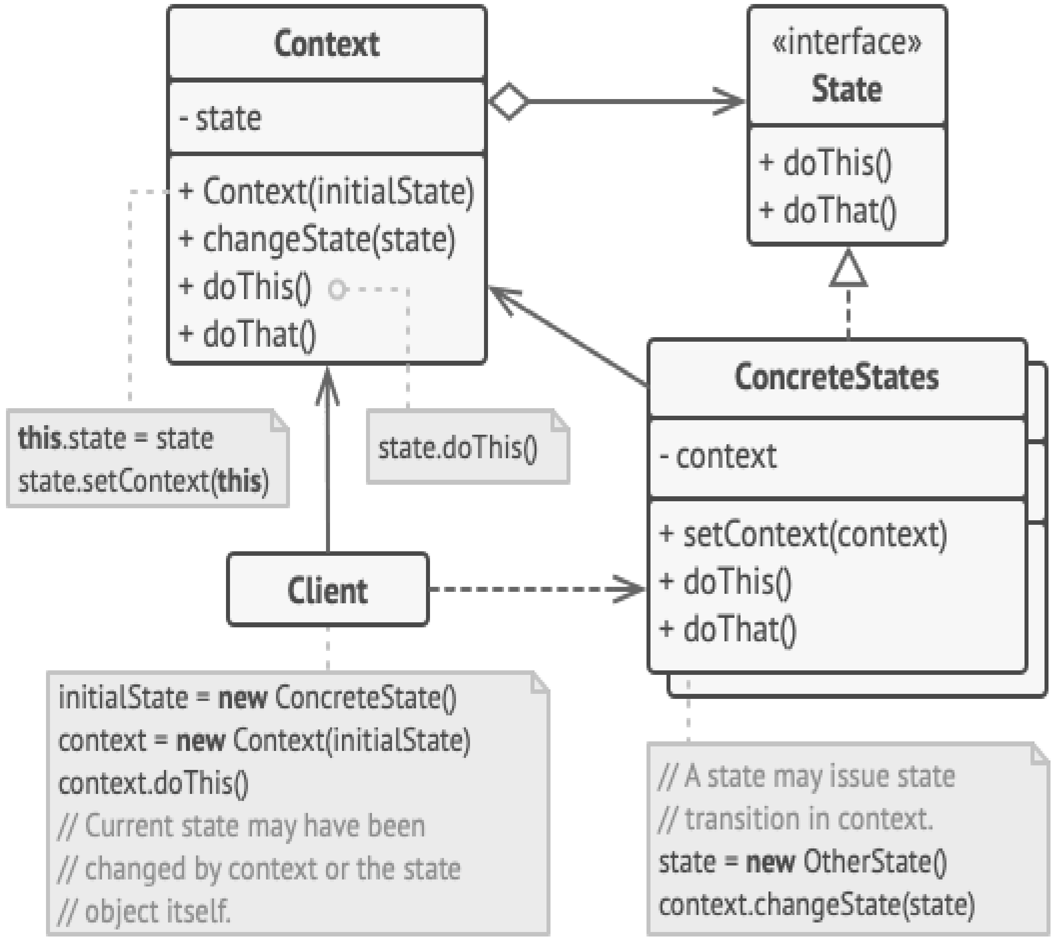


- *Factory method*
- *Abstract factory*
- *Singelton*

- *Strategy.*
- *State.*
- *Template method.*

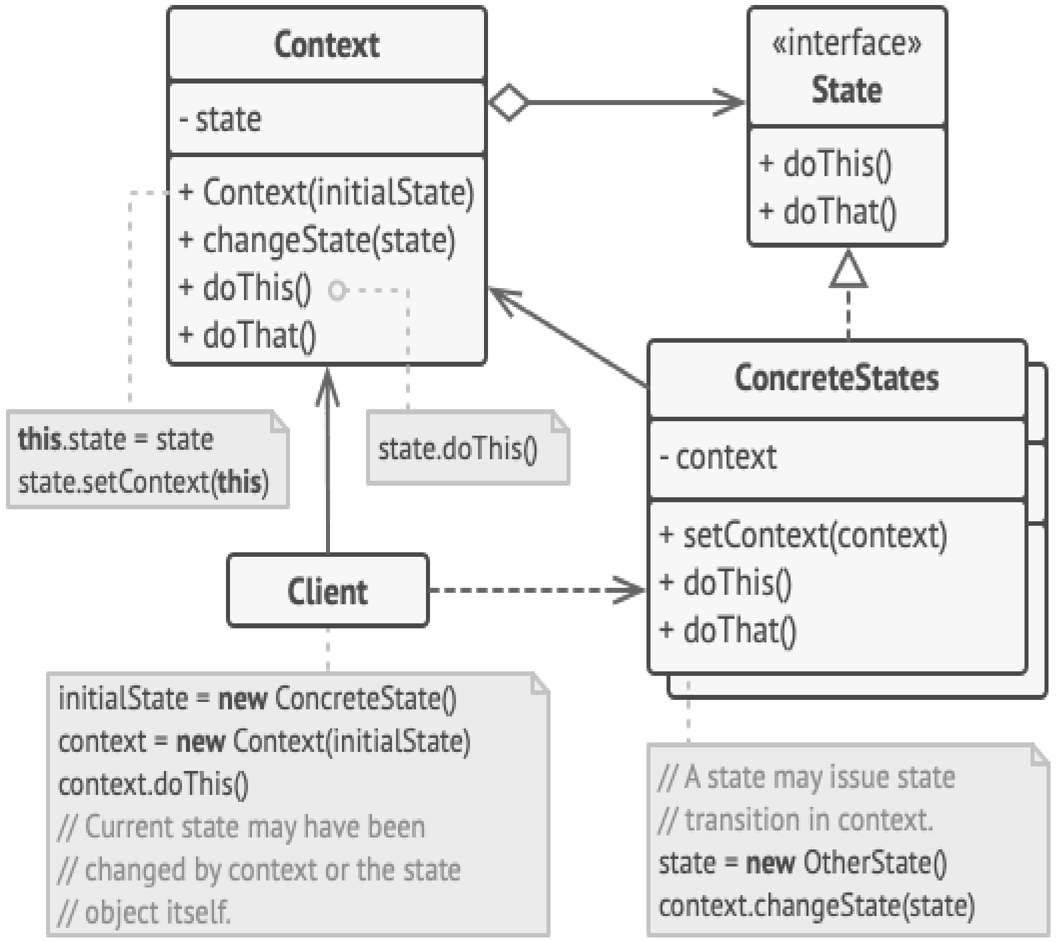
State design pattern:

State is a behavioral design pattern that lets an object alter its behavior when its internal state changes. It appears as if the object changed its class.



State design pattern:

State is a behavioral design pattern that lets an object alter its behavior when its internal state changes. It appears as if the object changed its class.

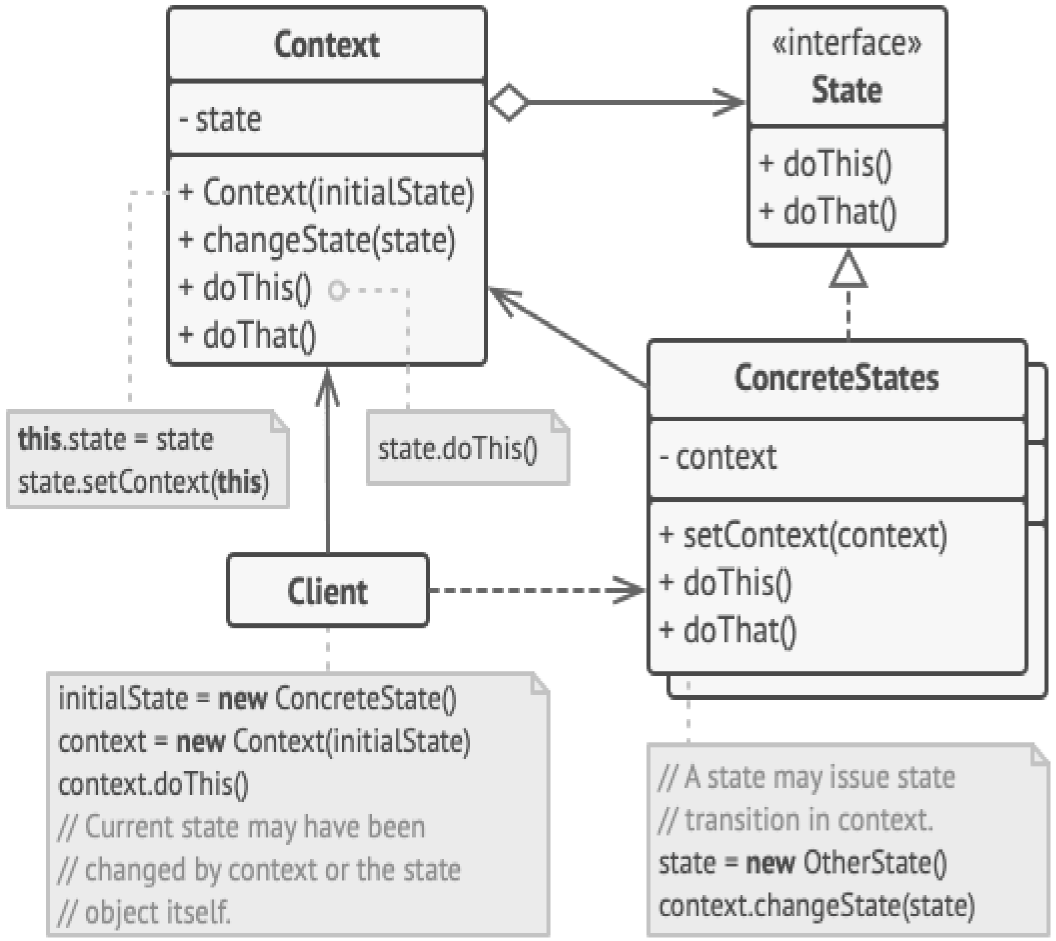


State gives a good solution to the problem where an object must change its behavior based on its internal state, and these changes should be clean, modular, and easily extendable.
Example: phone buttons.

State design pattern:

State is a behavioral design pattern that lets an object alter its behavior when its internal state changes. It appears as if the object changed its class.

State can be considered as an extension of Strategy. Both patterns are based on composition: they change the behavior of the context by delegating some work to helper objects.
***Strategy* makes these objects completely independent and unaware of each other. However, *State* doesn't restrict dependencies between concrete states, letting them alter the state of the context at will.**



```
this.state = state
state.setContext(this)
```

```
state.doThis()
```

```
initialState = new ConcreteState()
context = new Context(initialState)
context.doThis()
// Current state may have been
// changed by context or the state
// object itself.
```

```
// A state may issue state
// transition in context.
state = new OtherState()
context.changeState(state)
```

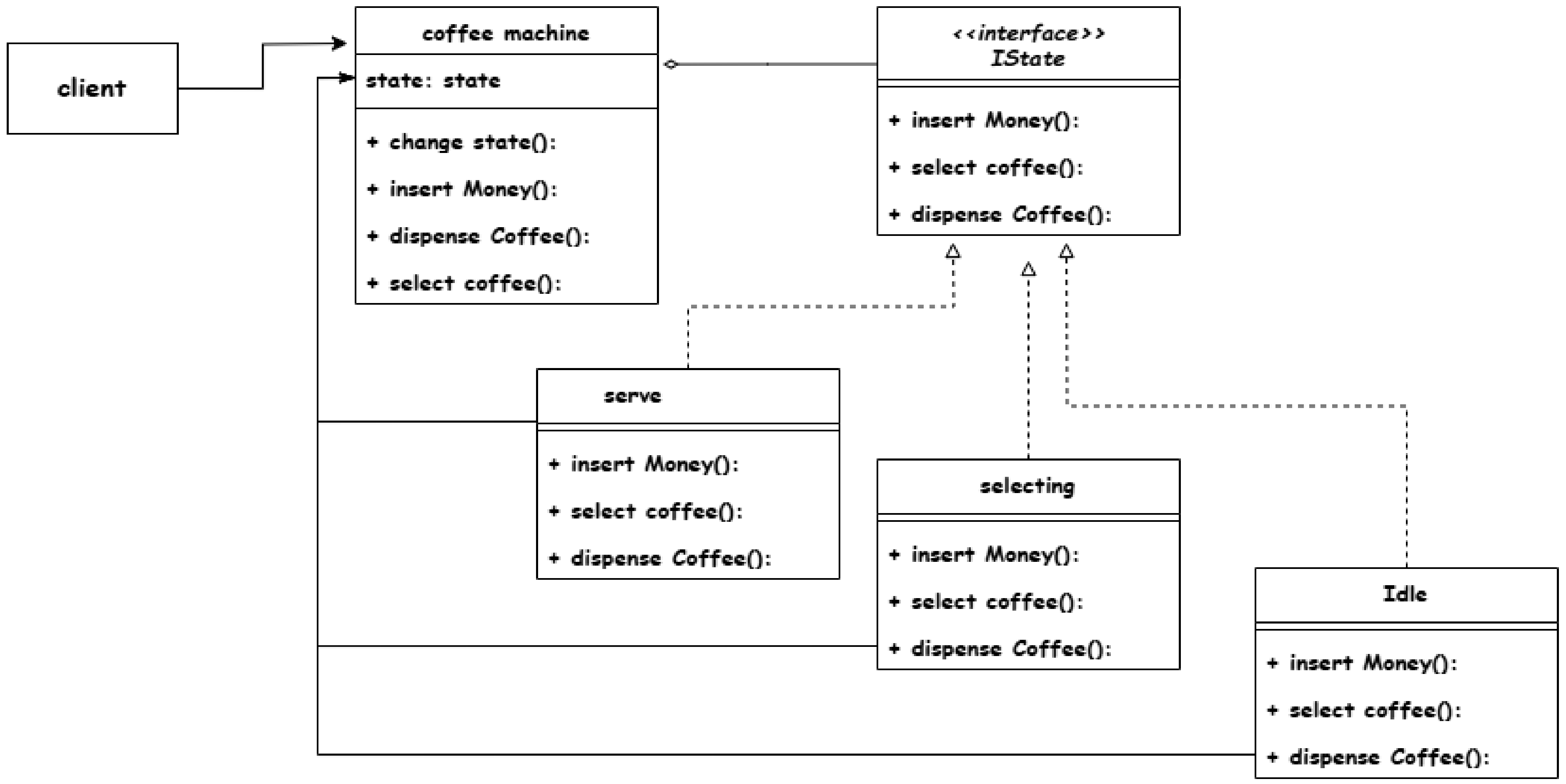
Example ...

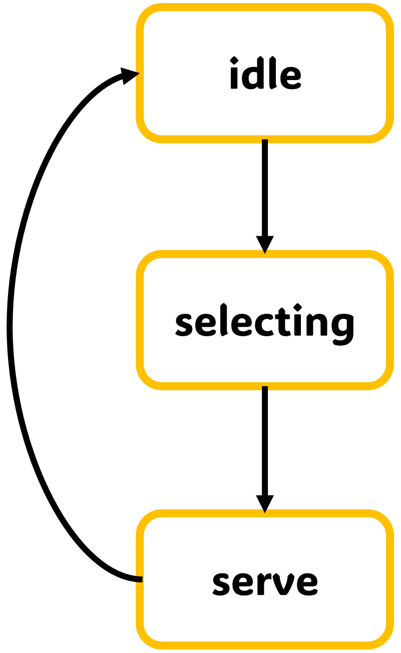
Problem1 – coffee machine:

A company has hired your team to develop a smart coffee machine.
The machine supports three user actions:
insertMoney() when it is idle, selectCoffee() after money has been inserted, and serveCoffee() once the coffee is ready.

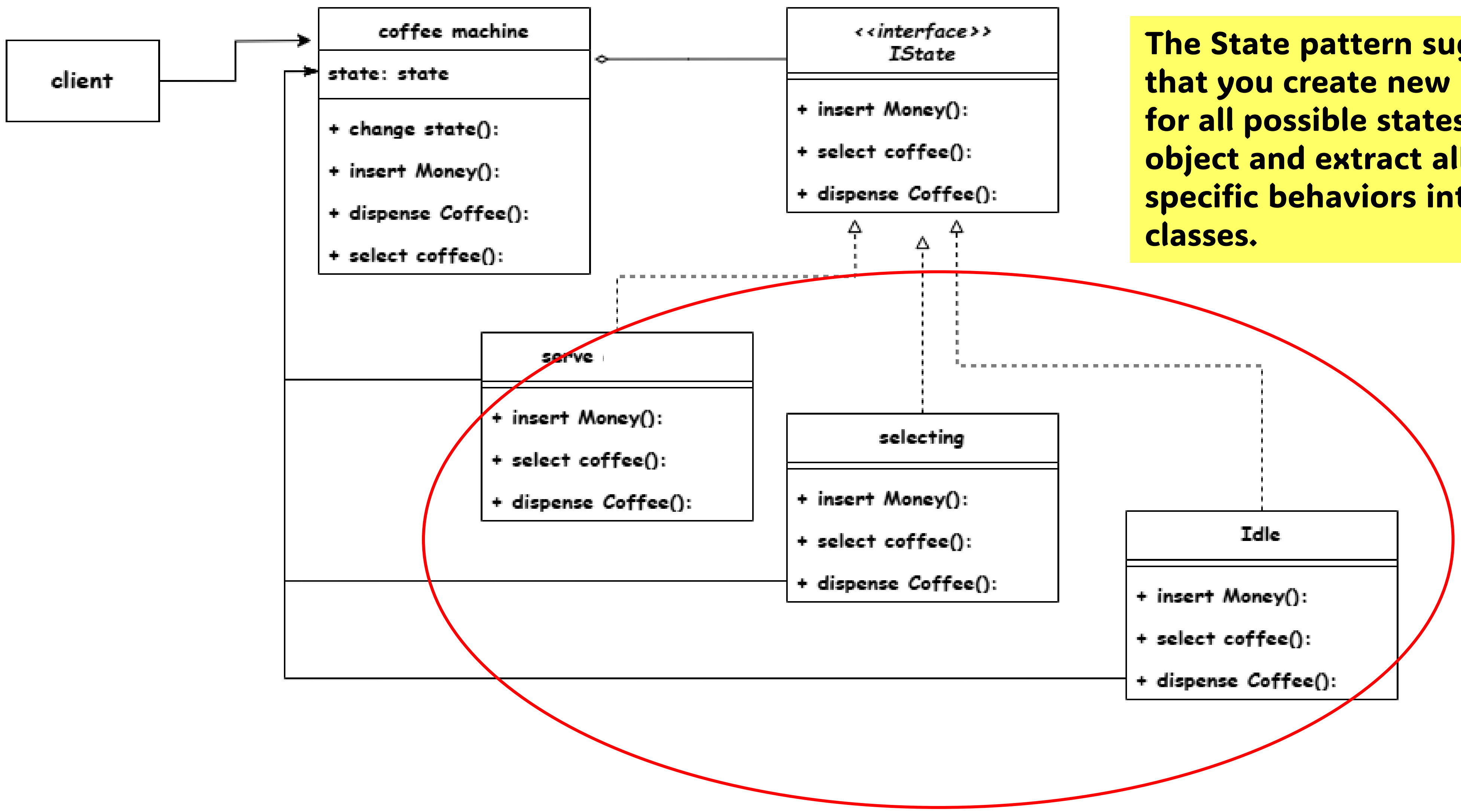
1. *Find the needed design pattern for this problem*
2. *Design a solution.*

Problem 1 – design with state design pattern:



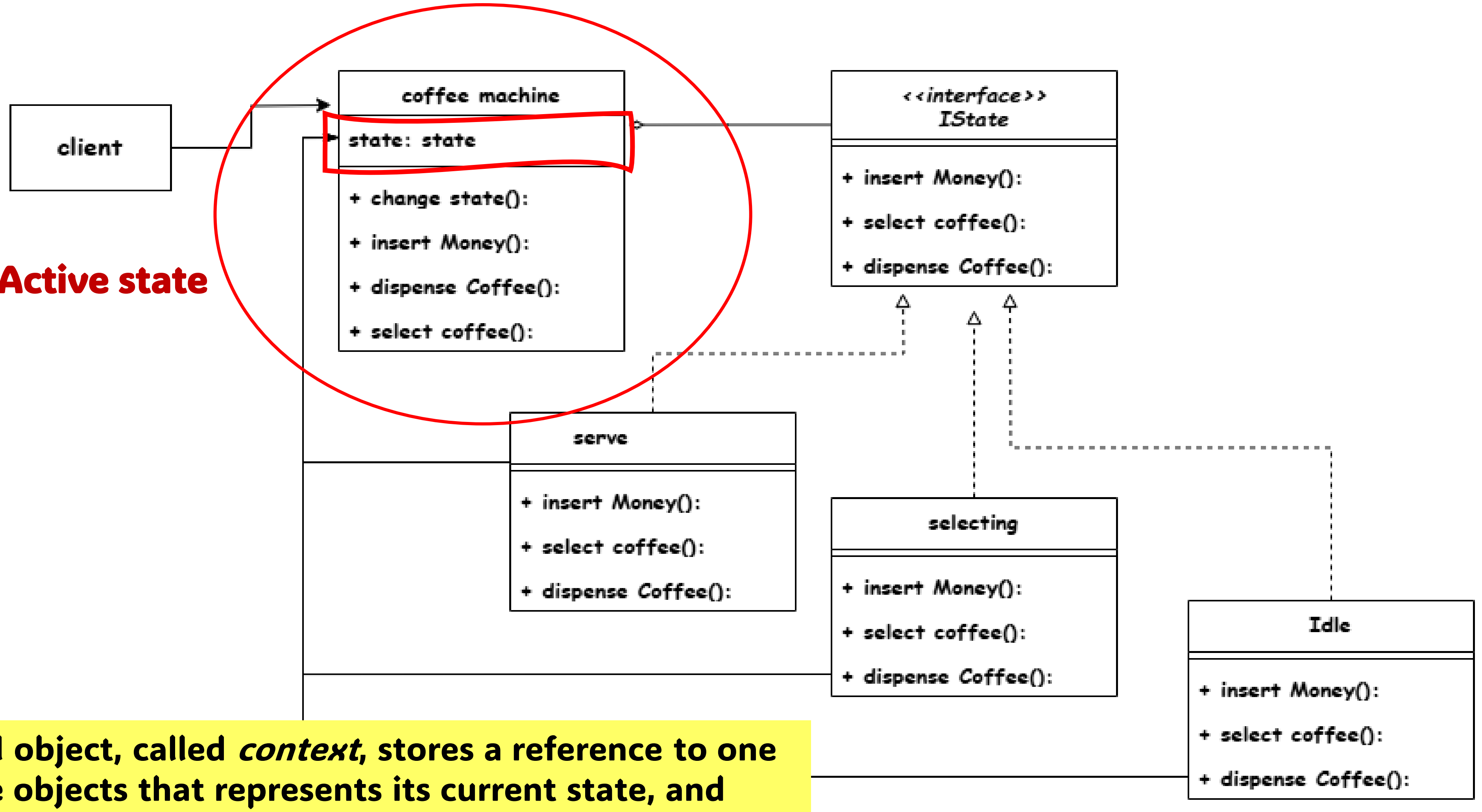


Problem 1 – design with state design pattern:



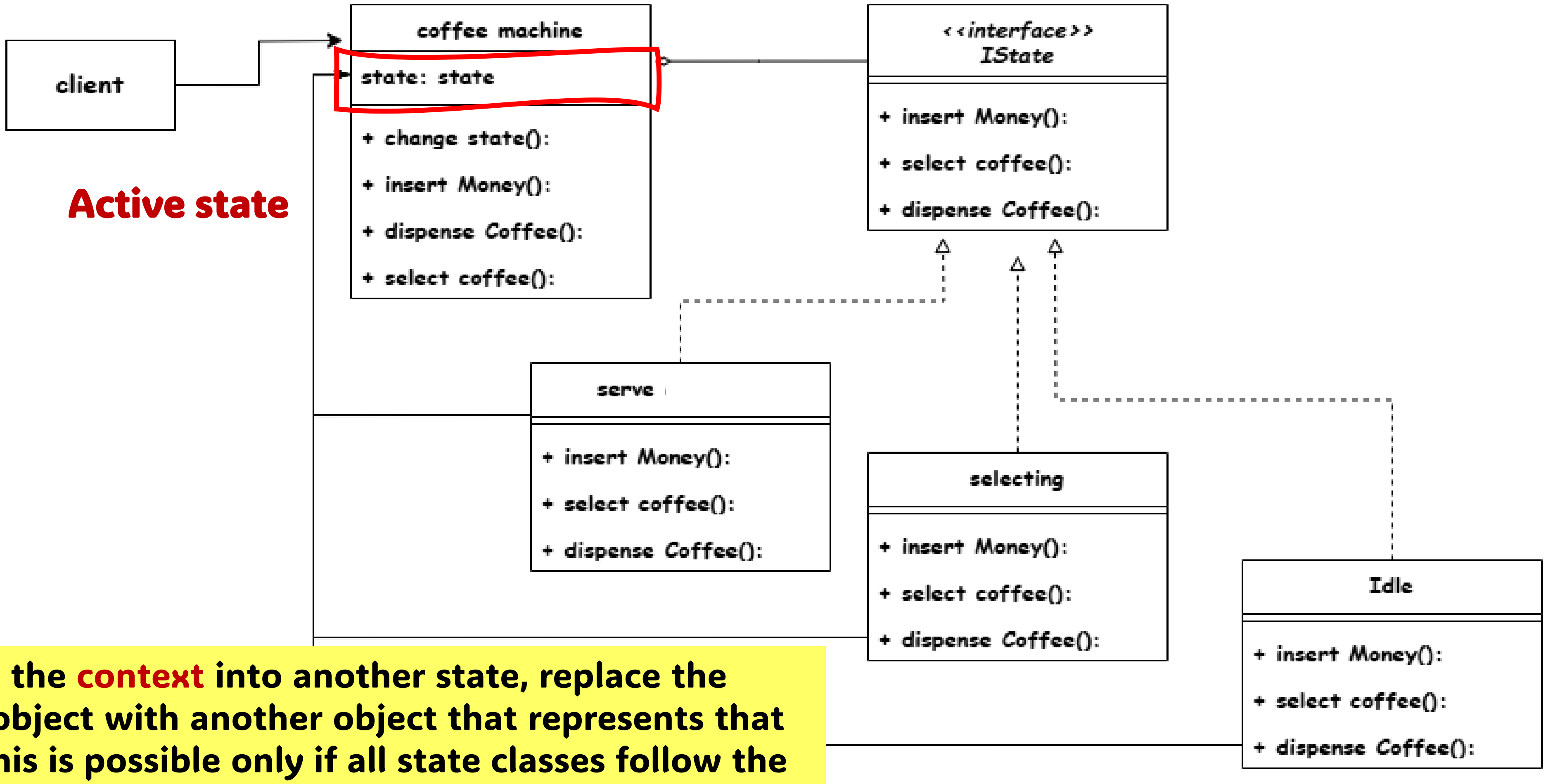
The State pattern suggests that you create new classes for all possible states of an object and extract all state-specific behaviors into these classes.

Problem 1 – design with state design pattern:



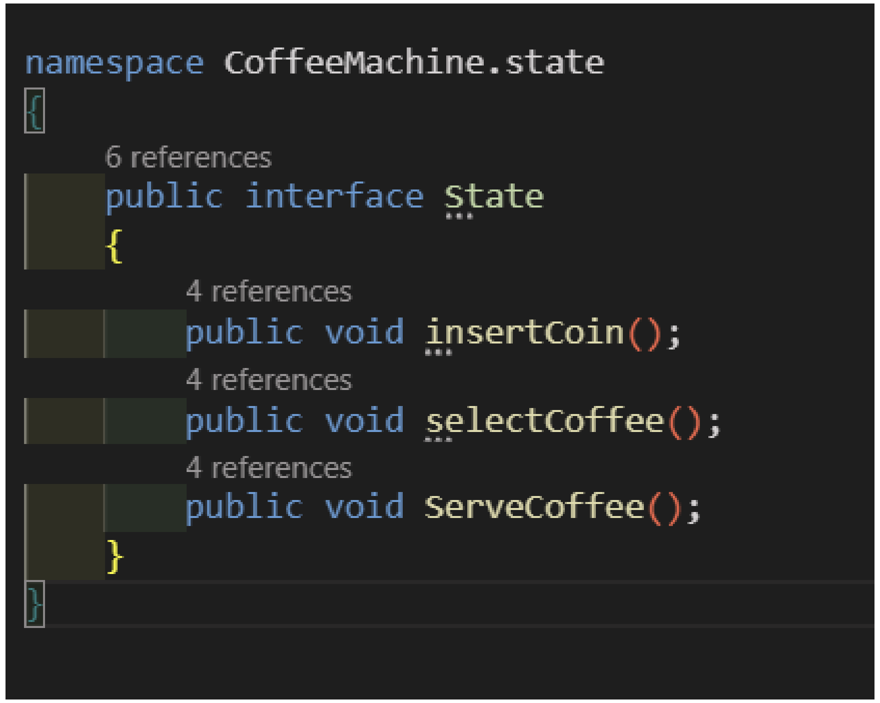
the original object, called *context*, stores a reference to one of the state objects that represents its current state, and delegates all the state-related work to that object.

Problem 1 – design with state design pattern:



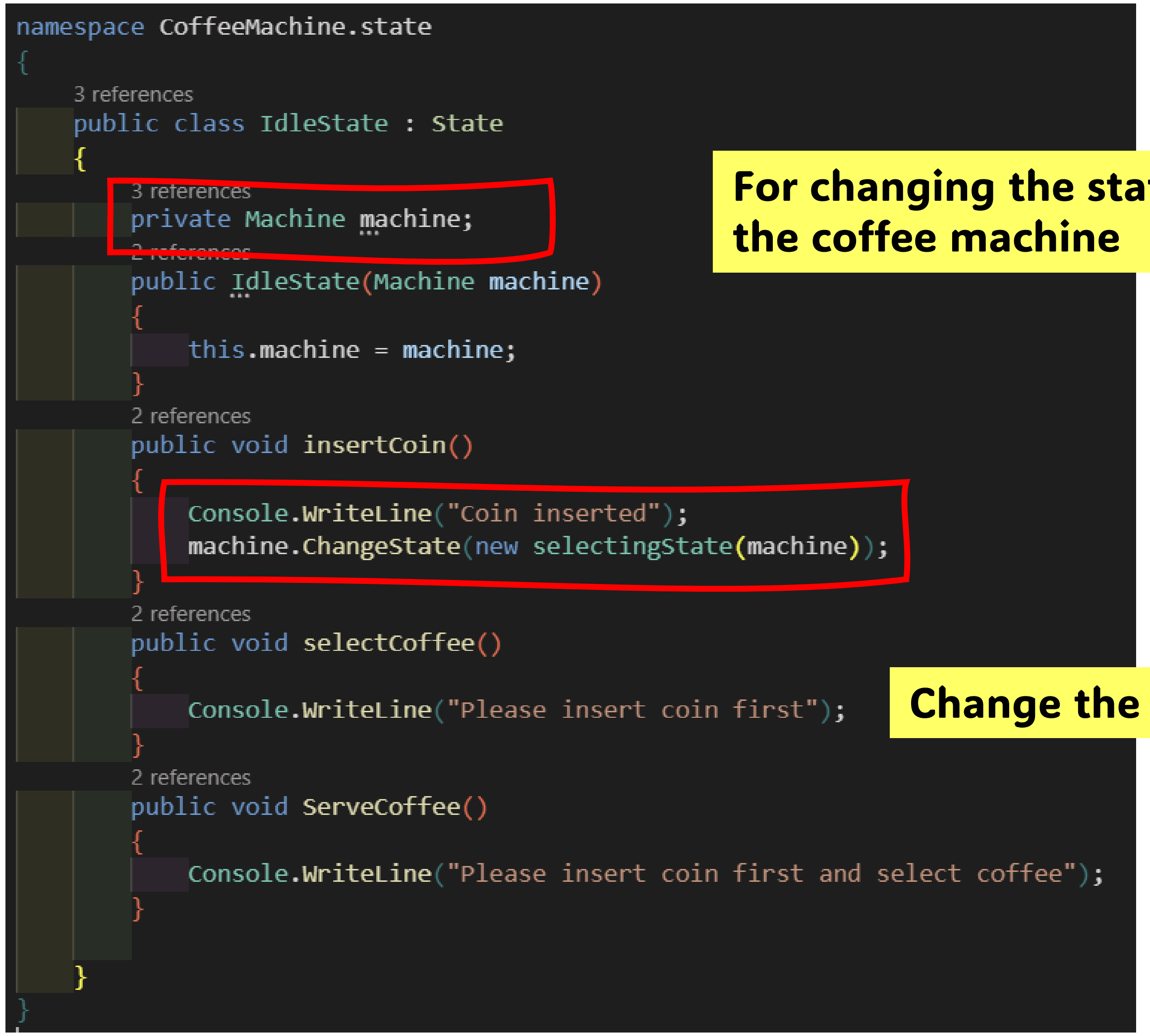
To transition the context into another state, replace the active state object with another object that represents that new state. This is possible only if all state classes follow the same interface and the context itself works with these objects through that interface.

Problem 1 – Implementation state design pattern:



```
namespace CoffeeMachine.state
{
    6 references
    public interface State
    {
        4 references
        public void insertCoin();
        4 references
        public void selectCoffee();
        4 references
        public void ServeCoffee();
    }
}
```

State interface



```
namespace CoffeeMachine.state
{
    3 references
    public class IdleState : State
    {
        3 references
        private Machine machine;
        2 references
        public IdleState(Machine machine)
        {
            this.machine = machine;
        }
        2 references
        public void insertCoin()
        {
            Console.WriteLine("Coin inserted");
            machine.ChangeState(new selectingState(machine));
        }
        2 references
        public void selectCoffee()
        {
            Console.WriteLine("Please insert coin first");
        }
        2 references
        public void ServeCoffee()
        {
            Console.WriteLine("Please insert coin first and select coffee");
        }

    }
}
```

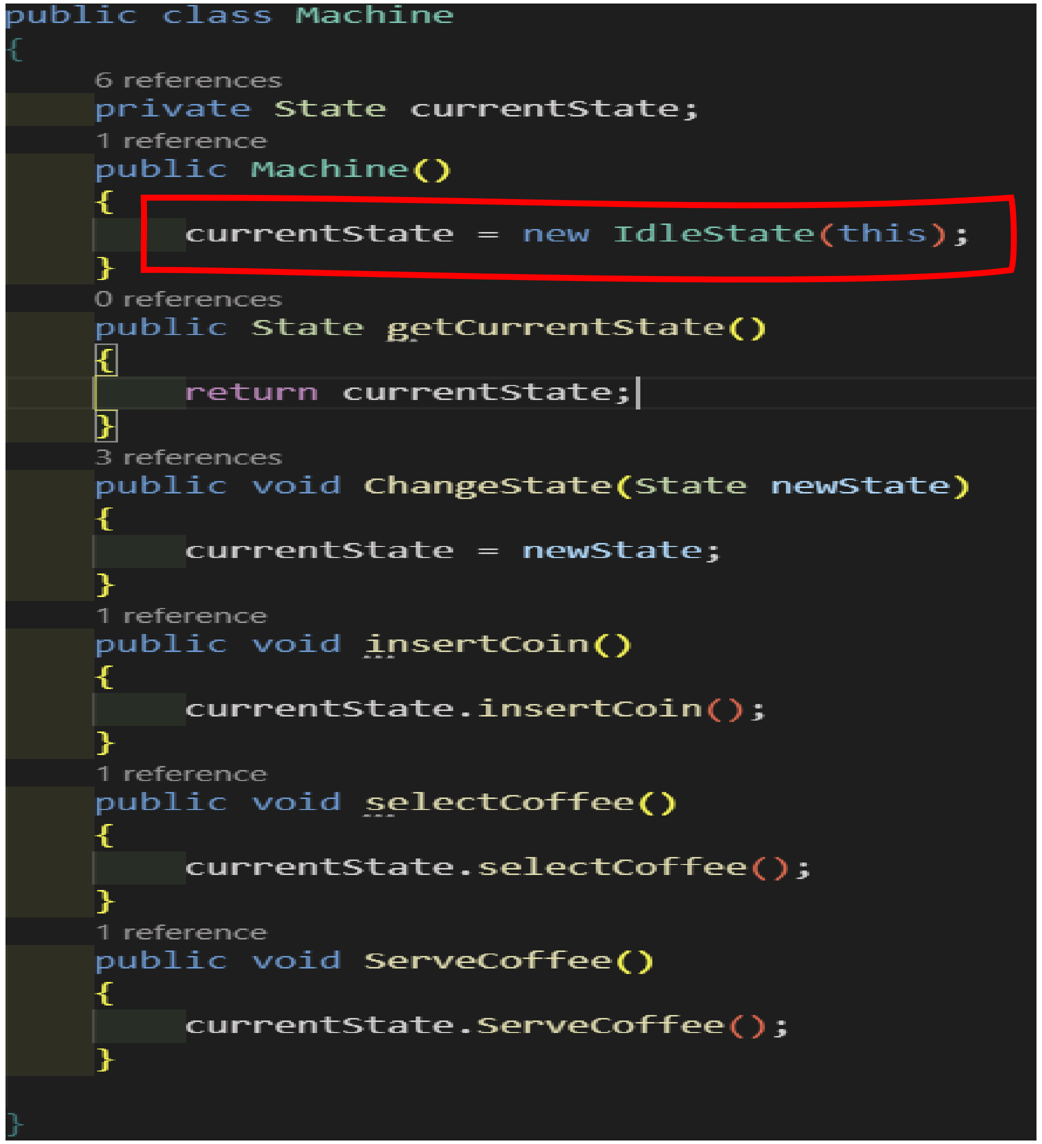
For changing the state of the coffee machine

Change the state

Concrete class – idle state

concrete classes provide their own implementation of the three functions.

Problem 1 – Implementation state design pattern:

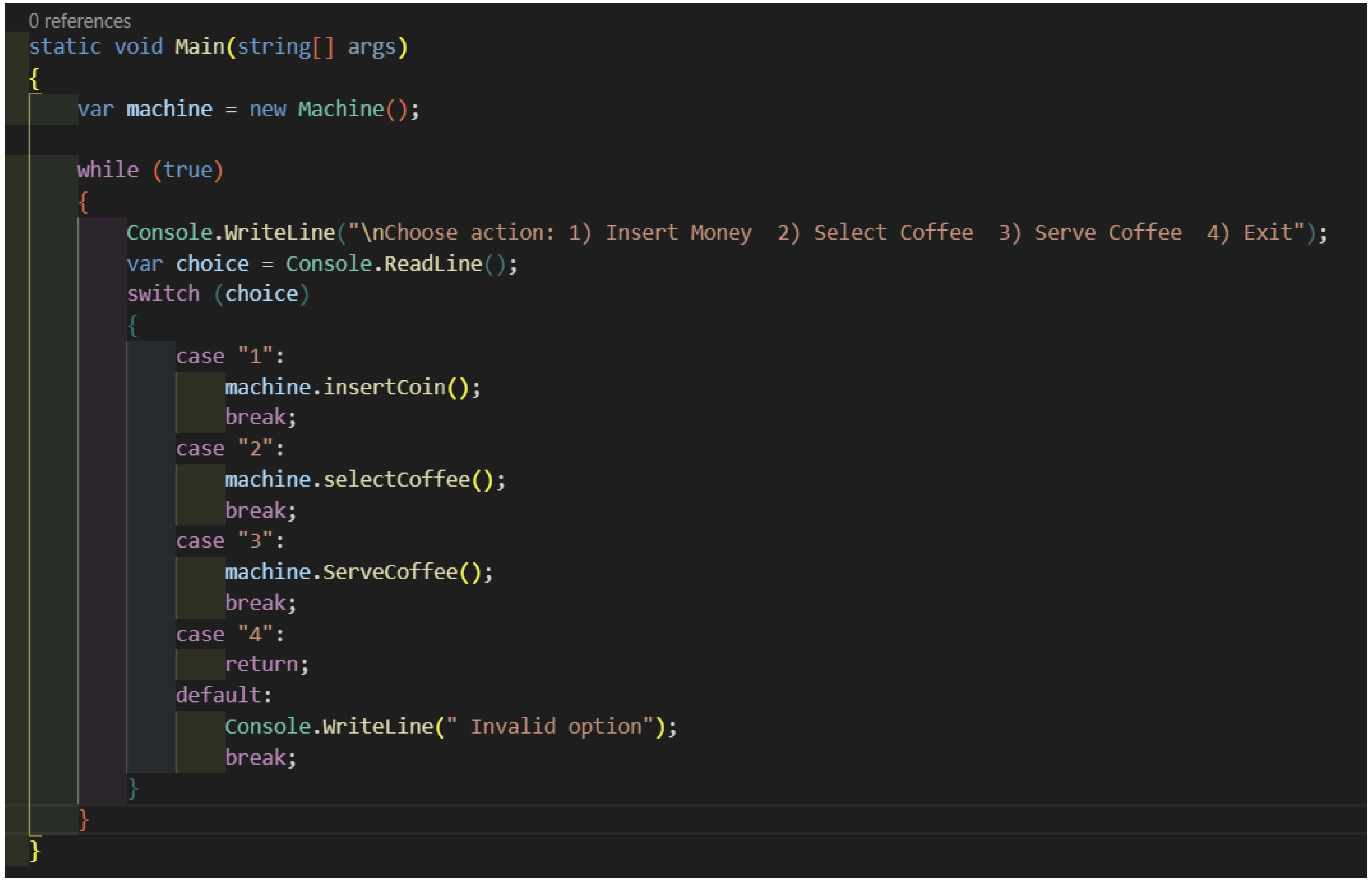


```
public class Machine
{
        6 references
        private State currentState;
        1 reference
        public Machine()
        {
            currentState = new IdleState(this);
        }
        0 references
        public State getCurrentState()
        {
            return currentState;
        }
        3 references
        public void ChangeState(State newState)
        {
            currentState = newState;
        }
        1 reference
        public void insertCoin()
        {
            currentState.insertCoin();
        }
        1 reference
        public void selectCoffee()
        {
            currentState.selectCoffee();
        }
        1 reference
        public void ServeCoffee()
        {
            currentState.ServeCoffee();
        }

}
```

Coffee machine class (Context) stores a reference to one of the concrete state objects and delegates to it all state-specific work. The context communicates with the state object via the state interface. The context exposes a setter for passing it a new state object.

Problem 1 – Implementation state design pattern:



```
0 references
static void Main(string[] args)
{
    var machine = new Machine();

    while (true)
    {
        Console.WriteLine("\nChoose action: 1) Insert Money  2) Select Coffee  3) Serve Coffee  4) Exit");
        var choice = Console.ReadLine();
        switch (choice)
        {
            case "1":
                machine.insertCoin();
                break;
            case "2":
                machine.selectCoffee();
                break;
            case "3":
                machine.ServeCoffee();
                break;
            case "4":
                return;
            default:
                Console.WriteLine(" Invalid option");
                break;
        }
    }
}
```

Client class

Good luck :)