Rendering Branching Over Boolean Tests Obsolete

Zoran Horvat

OWNER AT CODING HELMET CONSULTANCY

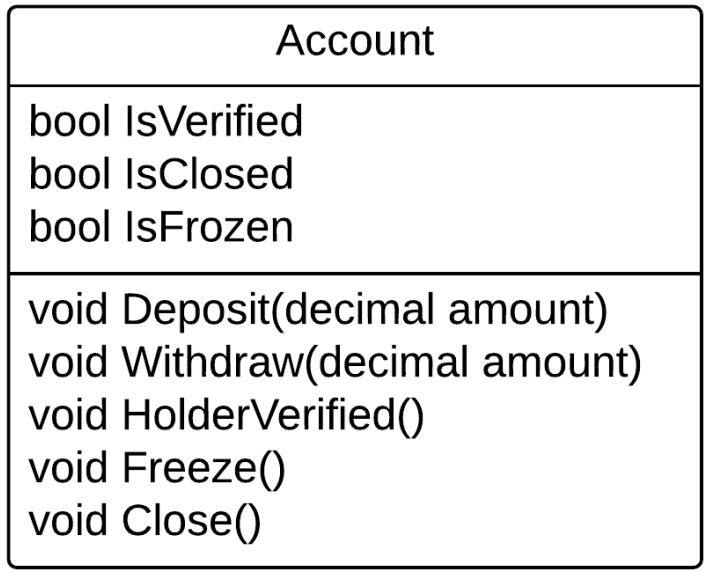
@zoranh75 codinghelmet.com

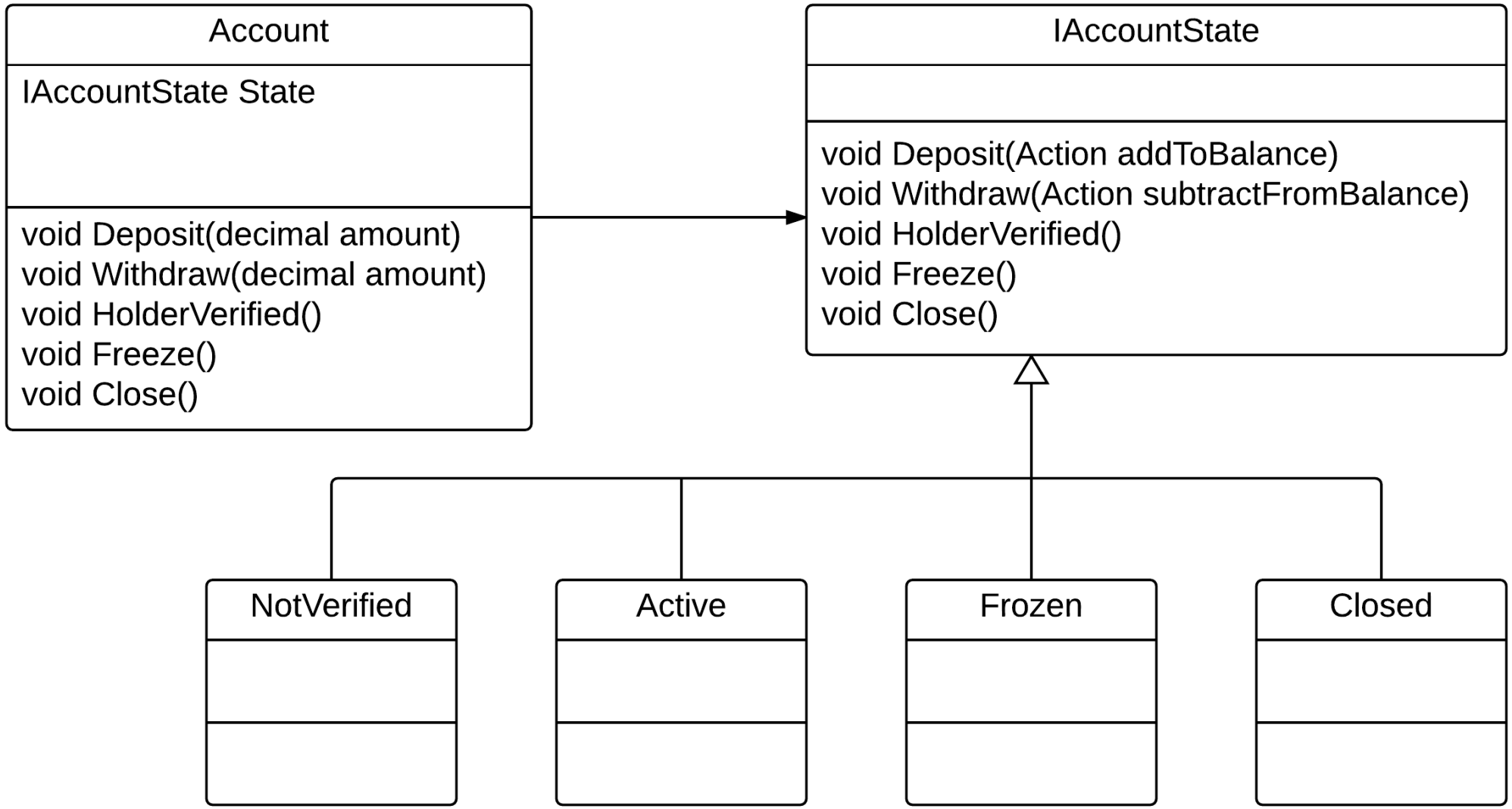
```
operationA();

operationB();
```

- **Two operations are coupled**
  Call to op.A must be followed by a call to op.B

```
operationA(operationB);


operationA(f)
{
    // ...
    f();
}
```

- **The callback principle**
  Pass op.B as an argument to op.A
- **Let op.A call op.B at its end**
- **This implementation guarantees that op.B will always follow op.A**


Account
bool IsVerified
bool IsClosed
bool IsFrozen
void Deposit(decimal amount)
void Withdraw(decimal amount)
void HolderVerified()
void Freeze()
void Close()


Account
IAccountState State
void Deposit(decimal amount)
void Withdraw(decimal amount)
void HolderVerified()
void Freeze()
void Close()
IAccountState
void Deposit(Action addToBalance)
void Withdraw(Action subtractFromBalance)
void HolderVerified()
void Freeze()
void Close()
NotVerified
Active
Frozen
Closed

Summary

Bad design

- A class which is doing everything itself
- This typically leads to if-then-else instructions everywhere

Better design

- Move separate implementations to separate *state* classes
- Substitute the *state* object to substitute implementation

Summary

Benefits of the State pattern

- Class that uses states becomes simple
- It can focus on its primary role
- Other roles are delegated to concrete state classes
- Each concrete class is simple

Refer to *Tactical Design Patterns in .NET: Control Flow* course

Next module –
Dealing with loops and sequences