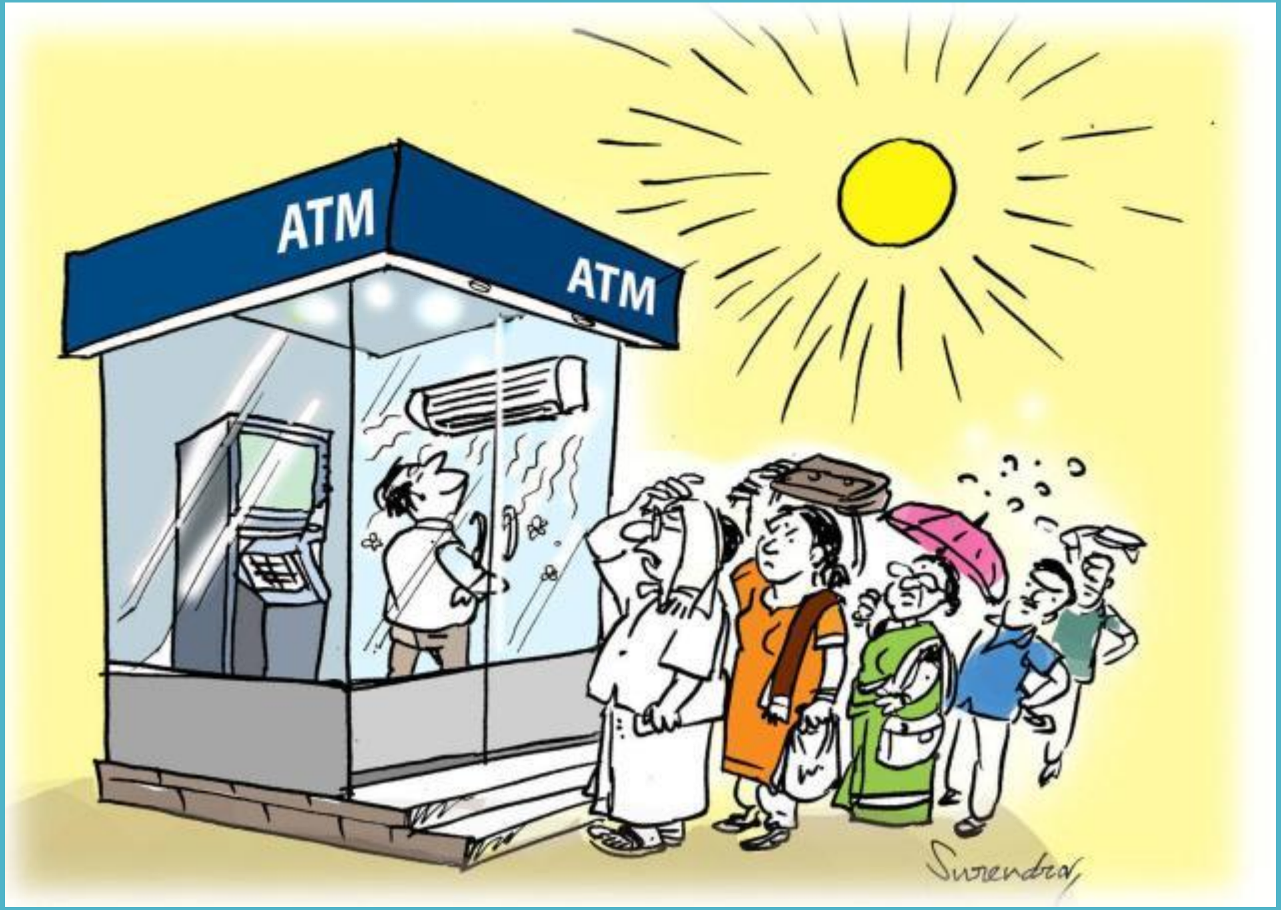
# The State Pattern

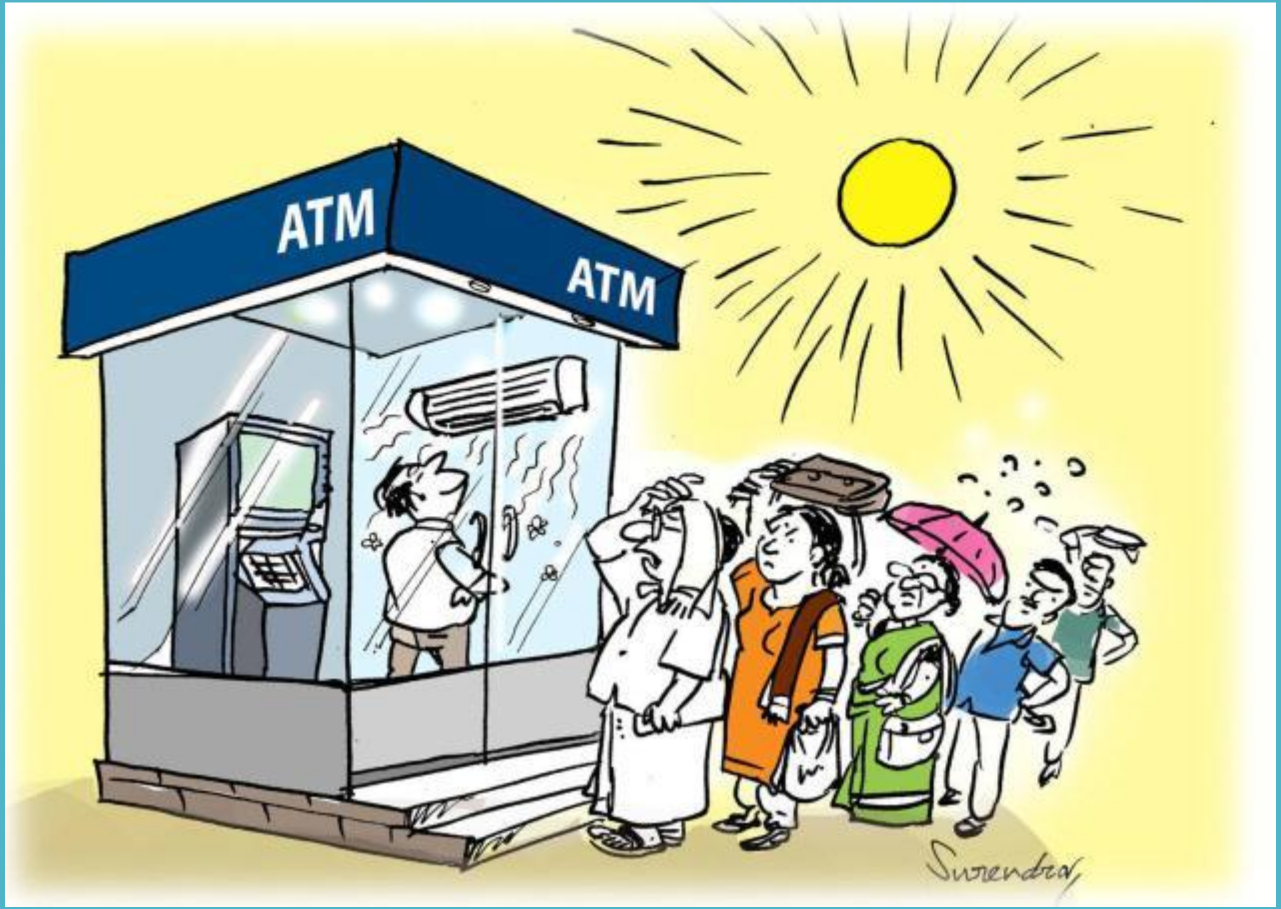
# A real-world problem



The ATM may have these four possible states:

+ Has card state
+ No card state
+ Has pin state
+ No cash state

# A real-world problem



And the different ways a user could use an ATM

+ Insert card
+ Eject card
+ Enter a pin
+ Request cash

## Let’s get started….

At first, we have to create an “ATM machine” class

```java
public class ATMMachine {
    final static int NoCard = 0;
    final static int HasCard = 1;
    final static int HasPin = 2;
    final static int NoCash = 3;

    int ATMState;

    public void insertCard() {…}

    public void ejectCard() {…}

    public void insertPin(int pinEntered) {…}

    public void requestCash(int cashWithdrawn) {…}
}
```

# Let’s get started….

The implementation of methods of ATMMachine class

```java
public void insertCard() {
    if (ATMState == NoCard) {
        ATMState = HasCard;
    }
    else if (ATMState == HasCard) {
        System.out.println("You can't insert more than one card");
    }
    else if (ATMState == HasPin) {
        System.out.println("You can't insert more than one card");
    }
    else if (ATMState == NoCash) {
        System.out.println("We don't have money. Come back later");
    }
}
```

# Let’s get started….

At first, we have to create an “ATM machine” class

```java
public void ejectCard() {
    if (ATMState == NoCard) {
        System.out.println("Insert your card first");
    }
    else if (ATMState == HasCard) {
        System.out.println("Card ejected");
        ATMState = NoCard;
    }
    else if (ATMState == HasPin) {
        System.out.println("Card ejected");
        ATMState = NoCard;
    }
    else if (ATMState == NoCash) {
        System.out.println("We don't have money");
    }
}
```

## Let’s get started….

At first, we have to create an “ATM machine” class

```
public void insertPin(int pinEntered) {…}

public void requestCash(int cashWithdrawn) {…}
```

# New method for our ATM…

The bank’s manager suddenly wants to add one new method to the ATM

When the user withdrawn the money (requestCash method), there will be a chance that the user will get some extra money
(cause now is the 30-year established anniversary of the bank…)

# Look like we're still fine...

The implementation of methods of ATMMachine class

```
public void insertCard() {
        if (ATMState == NoCard) {
                ATMState = HasCard;
        }
        else if (ATMState == HasCard) {
                System.out.println("You can't insert more than one card");
        }
        else if (ATMState == HasPin) {
                System.out.println("You can't insert more than one card");
        }
        else if (ATMState == NoCash) {
                System.out.println("We don't have money. Come back later");
        }
        else if (ATMState == LuckyState)  {...} // add one more if statement
}
void ATMMachine::insertPin(int PIN) {} // add one more if statement

void ATMMachine::rejectCard() {} // add one more if statement
```

## Look like we're still fine...

The implementation of methods of ATMMachine class

```
public void requestCash(int cashWithdrawn) {…} // add one more if statement, but
it takes some effort cause if we have the lucky state, we have to turn our state
Into HasPin state and call the method requestCash again
```

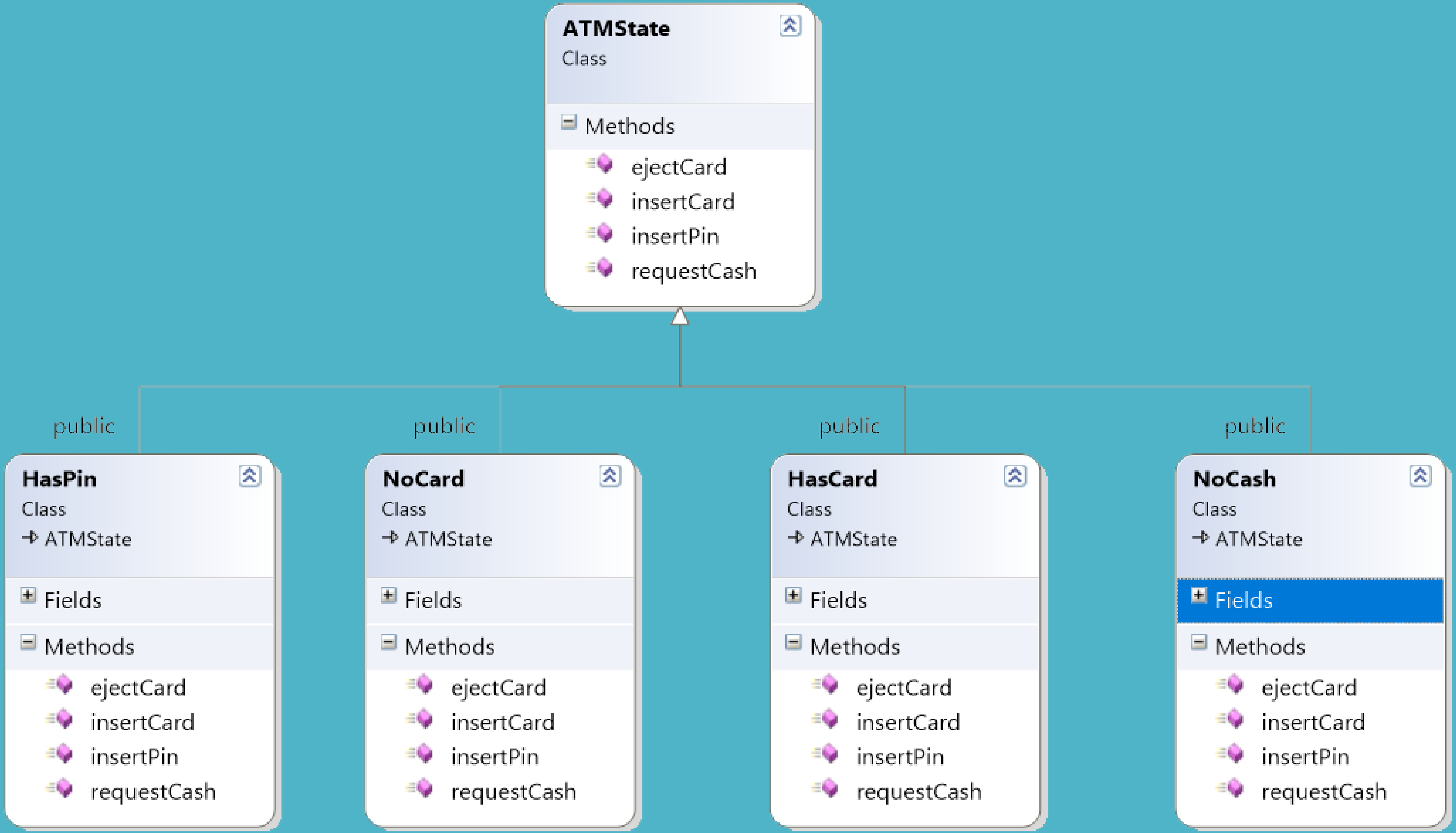
# Oops…!

The bank's manager, again, wants to add one new method to the ATM

It looks like everytime we add one new behavior to our object, we have to maintain all of the old lines of code, which will be very messy in the future work.

What if everytime we add one more, we  just have to take care that new "state" but our old code?

# New approach



## The State design pattern

The State Pattern allows an object to alter its behavior when its internal state changes. The object will appear to change its class.

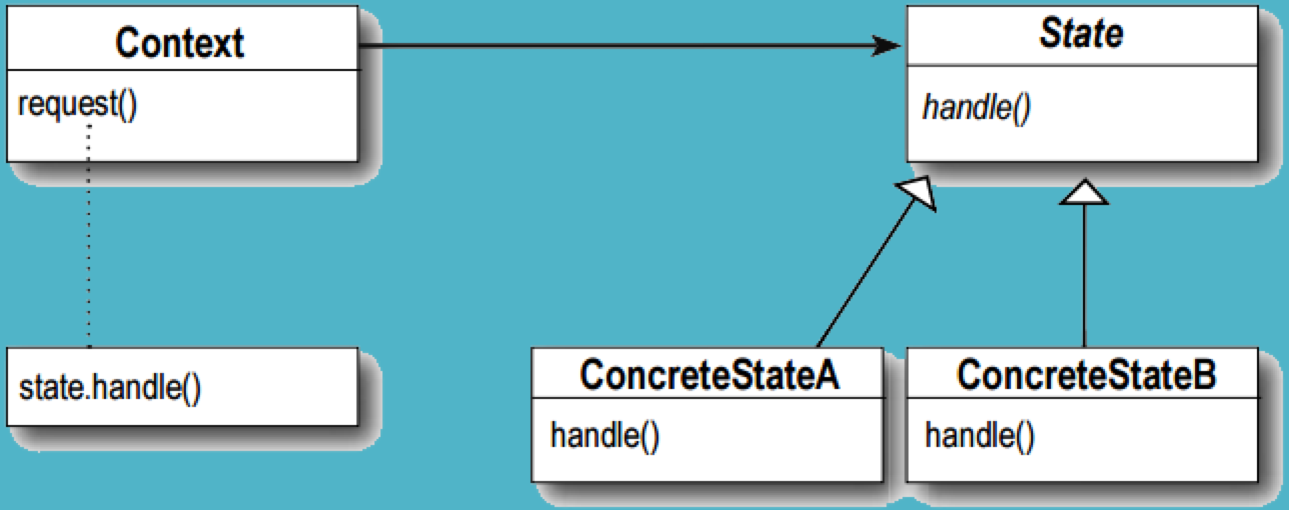
# The State design pattern

A state pattern includes:

+ Context (ATMMachine): HAS-AN instance of a ConcreteState subclass that define the current state.

+ State (ATMState): an interface to encapsulate all of the method associated with a particular state of the context.

+ Concreate State: each class implements behaviours of particular state of the context



# The State design pattern

Pros

+ Remove duplicated conditional statements

+ Easier to maintain and extend

Cons

+ Demanding more memory due to several state objects.

# The end