Behavioral Pattern: State

Kevin Dockx
Architect

@KevinDockx https://www.kevindockx.com

Coming Up

Describing the state pattern

- Withdrawing money from a bank account

Structure of the state pattern

Coming Up

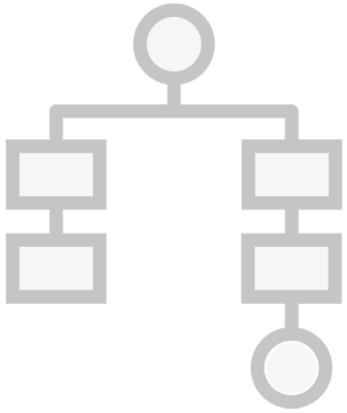
Use cases for this pattern

Pattern consequences

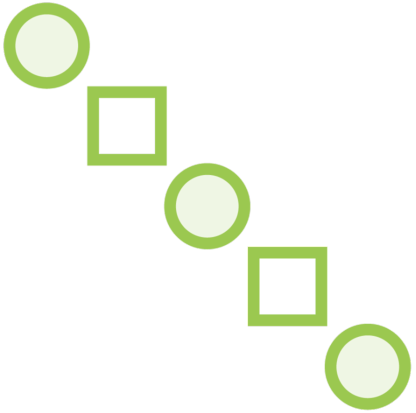
Related patterns

Describing the State Pattern

Creational



Structural



Behavioral

State

The intent of this pattern is to allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

Describing the State Pattern

Withdrawing or depositing money in a bank account

- BankAccount class has a BankAccountState field
- State can change when one of these operations is executed

```
public void Deposit(decimal amount) {
     Balance += amount;
     if (Balance >= 0)
     {
          BankAccountState = BankAccountState.Regular;
     }}
```

Describing the State Pattern

Part of a BankAccount class: withdrawing money can change its state, signified by a BankAccountState property

```
public void Deposit(decimal amount) {
     Balance += amount;
     if (Balance >= 0)
     {
          BankAccountState = BankAccountState.Regular;
     }}
```

Describing the State Pattern

Part of a BankAccount class: withdrawing money can change its state, signified by a BankAccountState property

```
public void Withdraw(decimal amount) {
     switch (BankAccountState) {
          case BankAccountState.Regular:
               Balance -= amount;
               if (Balance < 0) {
                    BankAccountState = BankAccountState.Overdrawn;
               }
               break;
          case BankAccountState.Overdrawn:
               // cannot withdraw
               break;
          default:
               break; }}}
```

Describing the State Pattern

Part of a BankAccount class: withdrawing money can change its state, signified by a BankAccountState property

```
public void Withdraw(decimal amount) {
     switch (BankAccountState) {
          case BankAccountState.Regular:
               Balance -= amount;
               if (Balance < 0) {
                    BankAccountState = BankAccountState.Overdrawn;
               }
               break;
          case BankAccountState.Overdrawn:
               // cannot withdraw
               break;
          default:
               break; }}}
```

Describing the State Pattern

Part of a BankAccount class: withdrawing money can change its state, signified by a BankAccountState property

```
public void Withdraw(decimal amount) {
     switch (BankAccountState) {
          case BankAccountState.Regular:
               Balance -= amount;
               if (Balance < 0) {
                    BankAccountState = BankAccountState.Overdrawn;
               }
               break;
          case BankAccountState.Overdrawn:
               // cannot withdraw
               break;
          default:
               break; }}}
```

Describing the State Pattern

Part of a BankAccount class: withdrawing money can change its state, signified by a BankAccountState property

Describing the State Pattern

Adding additional state

- Logic for transitioning can become more complex
- Conditional statement can become more complex

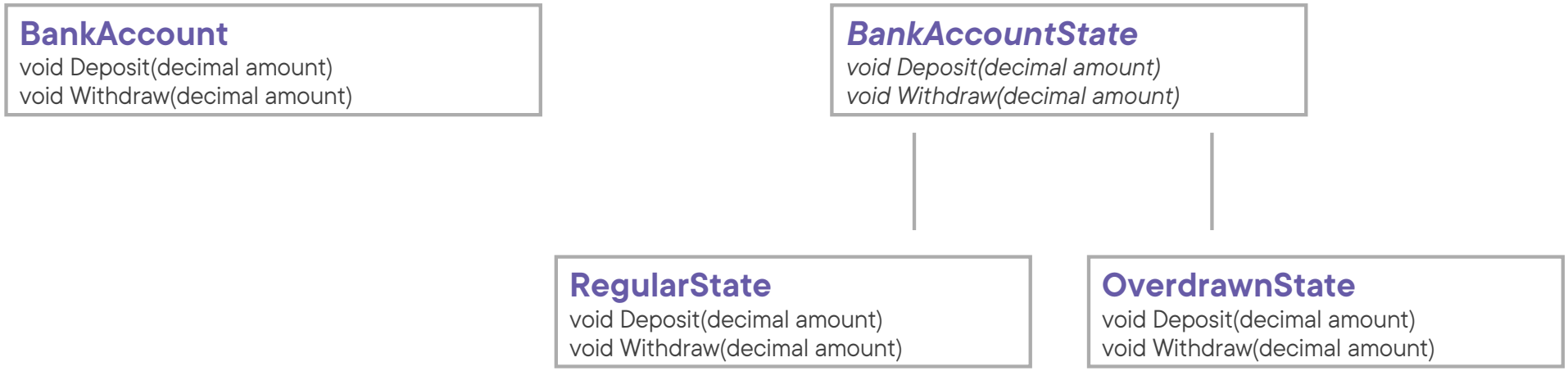
Adding additional business rules also increases complexity

Describing the State Pattern

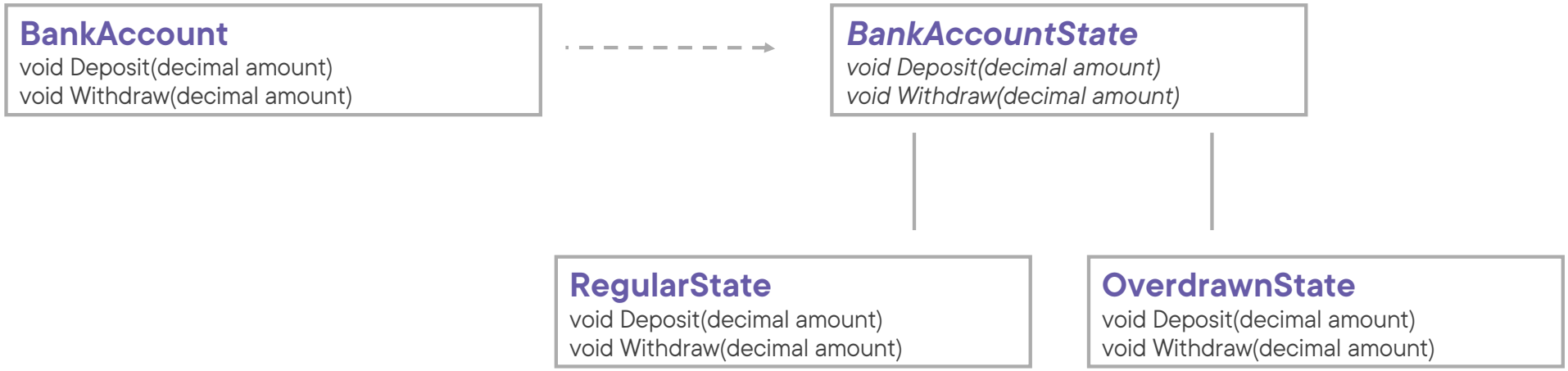
Describing the State Pattern

Describing the State Pattern

Describing the State Pattern



Describing the State Pattern



Describing the State Pattern

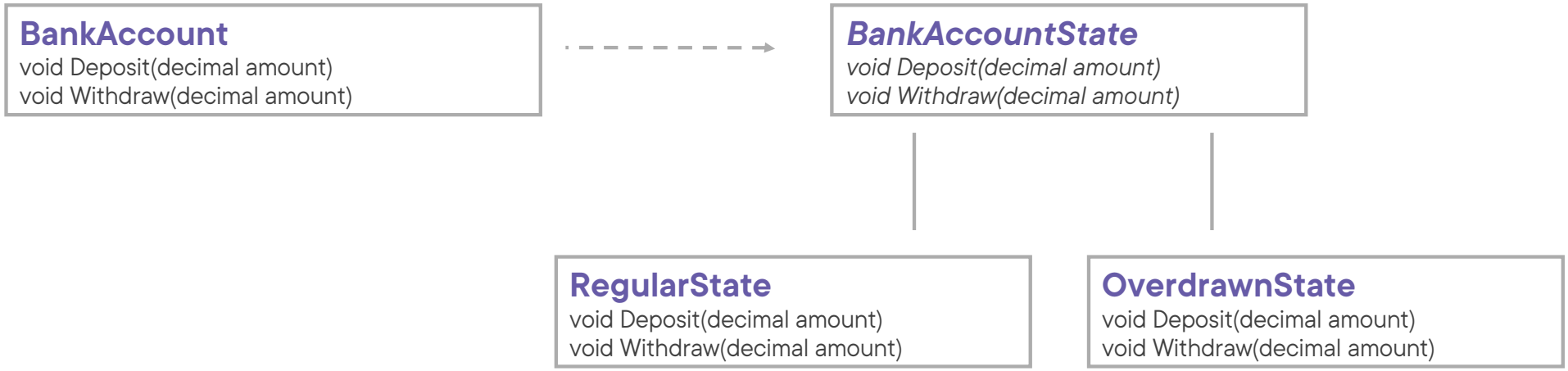
The more states you have, the more improvements in regards to complexity you'll notice

- Typically not every state can transition to all others

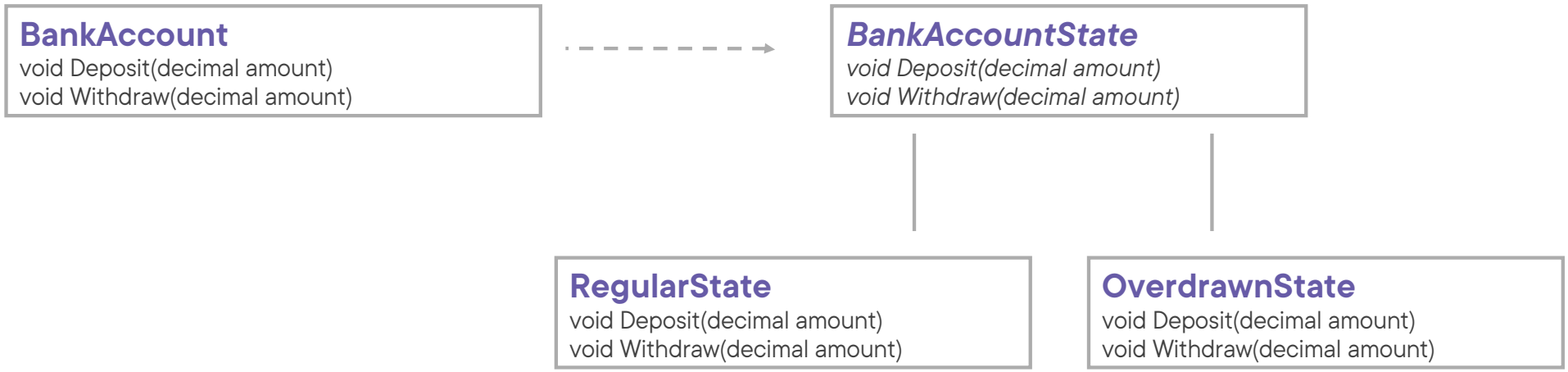
Conditional statements become easier

- You don't have to check for the state you're in when you're in a state object

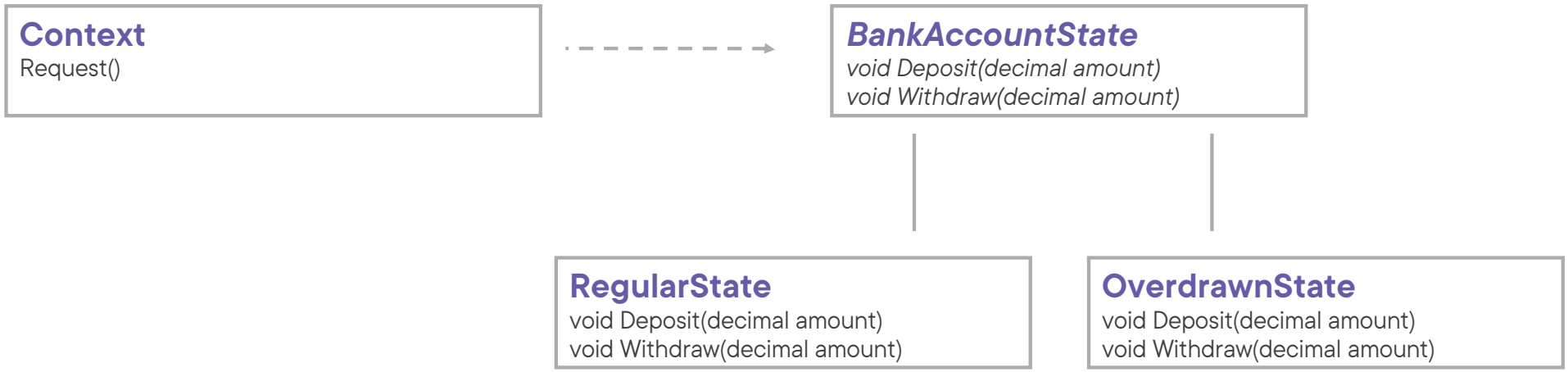
Describing the State Pattern



Structure of the State Pattern

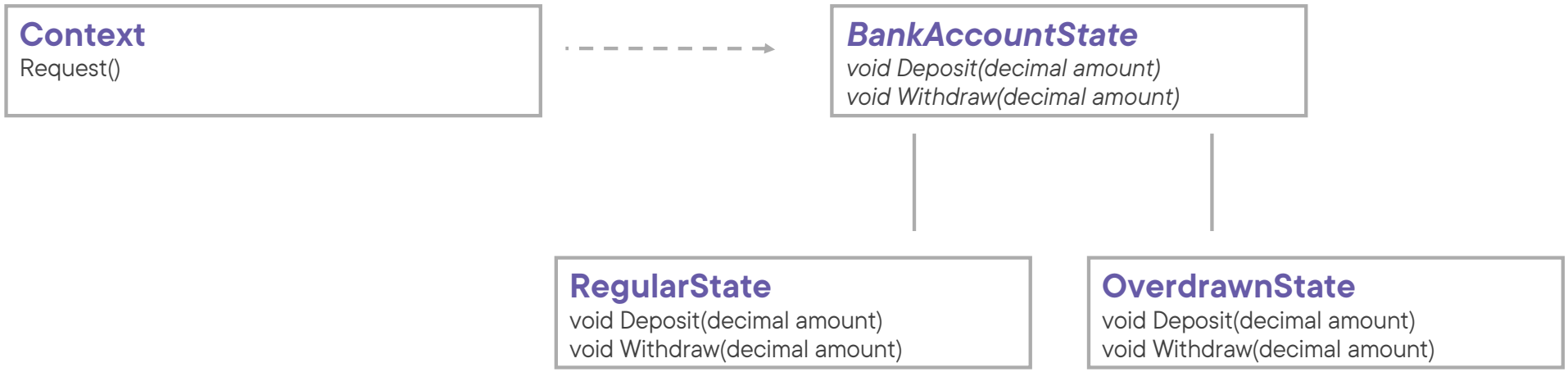


Structure of the State Pattern

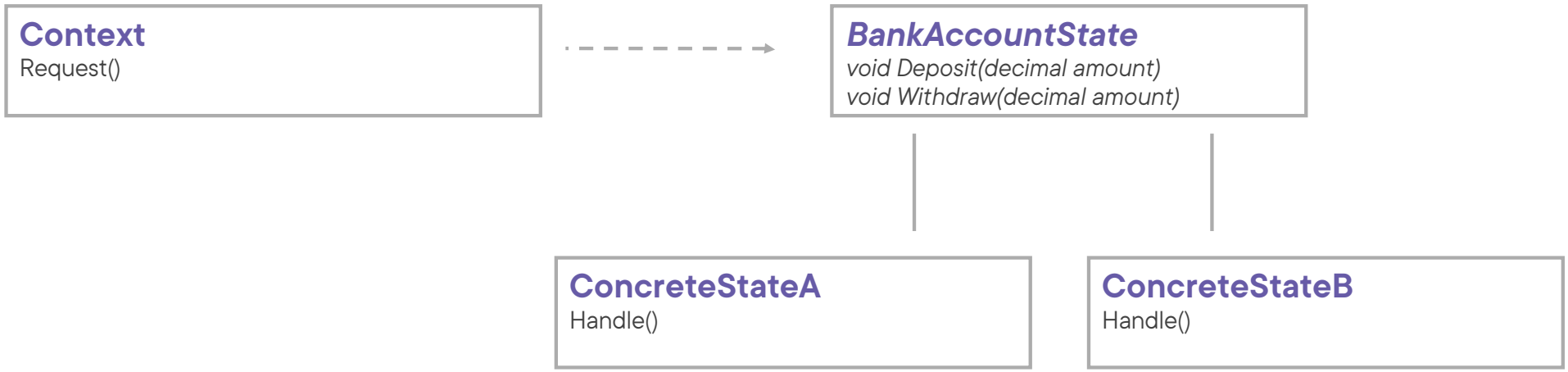


Context defines the interface that's of interest to clients. It maintains an instance of a **ConcreteState** subclass that defines the current state.

Structure of the State Pattern



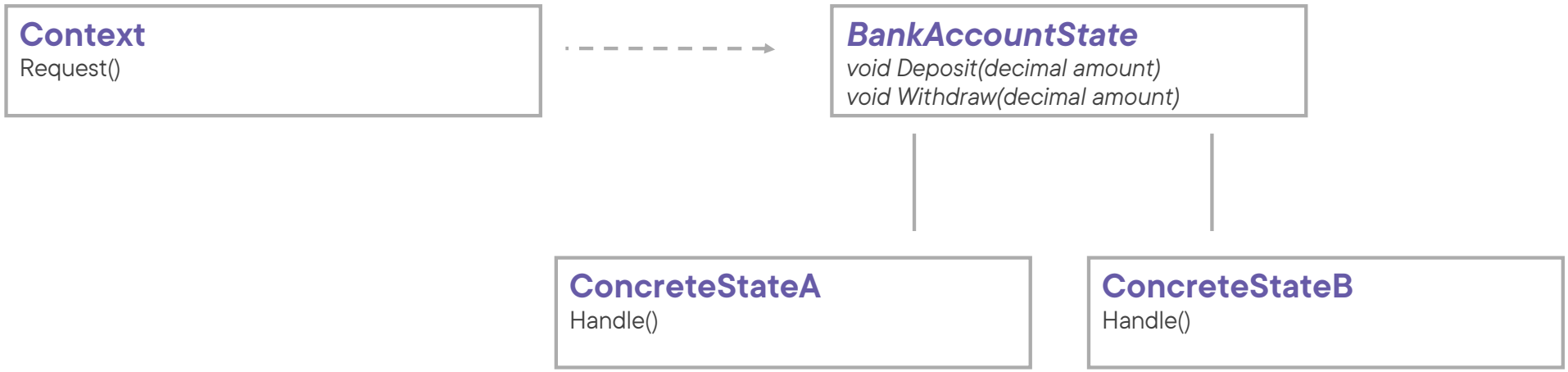
Structure of the State Pattern



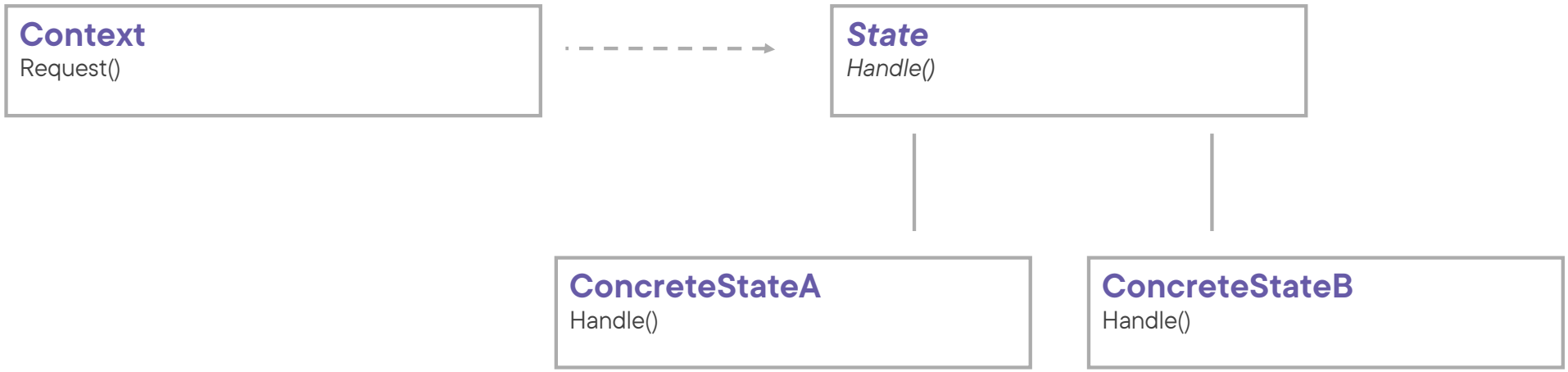
ConcreteState implements behavior associated with a state of the **Context**

State defines an interface for encapsulating the behavior associated with a particular state of the **Context**

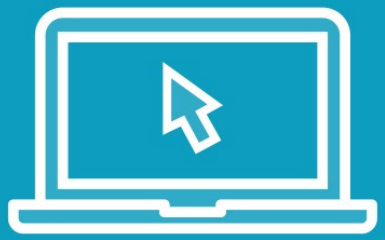
Structure of the State Pattern



Structure of the State Pattern

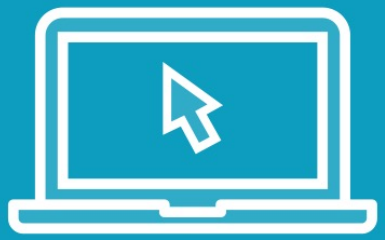


Demo



Implementing the state pattern

Demo



Extending the bank account sample with an additional state

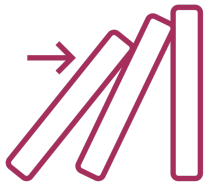
Use Cases for the State Pattern

When an object's behavior depends on its state and it must change it at runtime (depending on that state)

When your objects are dealing with large conditional statements that depend on the object's state

Pattern Consequences

It localizes state-specific behavior and partitions behavior for different states: single responsibility principle



New states and transitions can easily be added by defining new subclasses: open/closed principle

The number of classes is increased, which adds additional complexity

Related Patterns

Flyweight
Without instance variables in the state objects, they become flyweights

Singleton
State objects are often singleton

Strategy
Also based on composition, but solves a different problem

Bridge
Also based on composition, but solves a different problem

Summary

Intent of the state pattern:

- To allow an object to alter its behavior when its internal state changes

Summary

Implementation:

- Transitions are handled in the state objects themselves
- Context needs to pass requests through to the underlying state objects to handle them

Up Next:
Behavioral Pattern: Iterator