

Behavioral Pattern: Command



Kevin Dockx

Architect

@Kevindockx | www.kevindockx.com



Coming Up



Describing the command pattern

- Clicking a button to add an employee to a list

Structure of the command pattern

Variation: supporting undo with a command manager



Coming Up



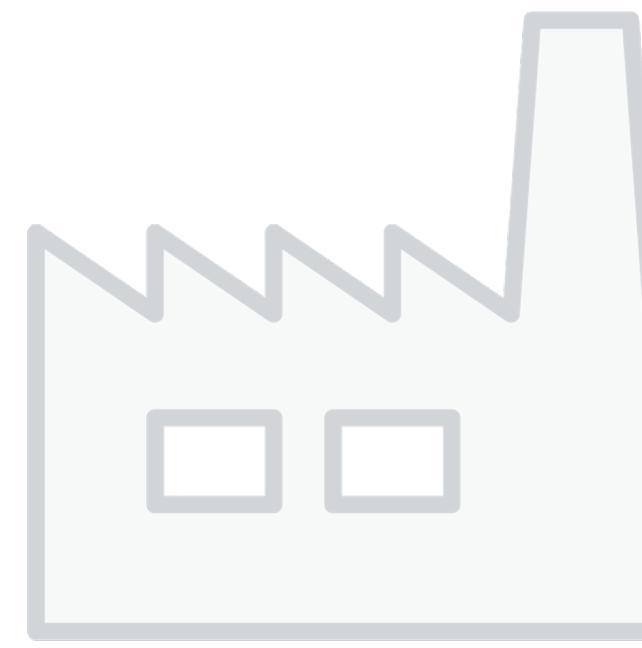
Use cases for this pattern

Pattern consequences

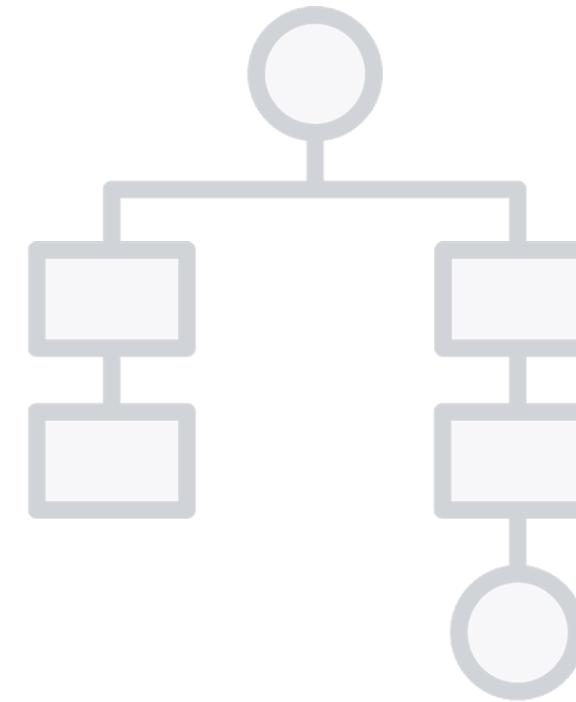
Related patterns



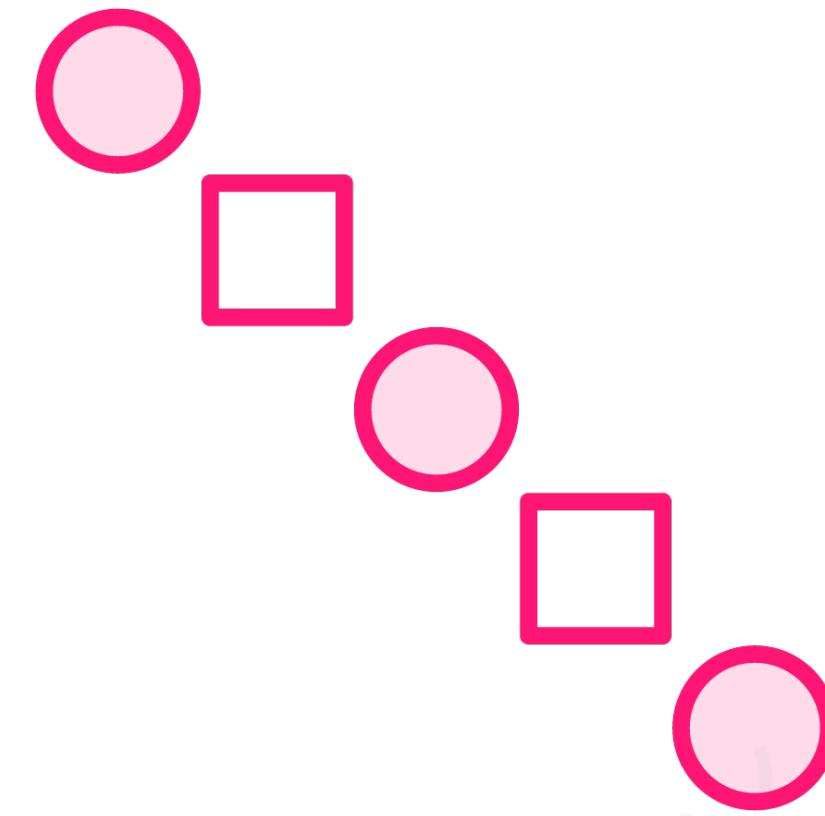
Describing the Command Pattern



Creational



Structural



Behavioral



Command

The intent of this pattern is to encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.



```
void SomeButton_Click(Object sender, EventArgs e)
{
    // open file...
    // add product...
    // add employee to list...
}
```

Describing the Command Pattern

No separation of concerns

Not a good approach for code reuse

Not technically feasible sometimes



Describing the Command Pattern

Command pattern allows decoupling the requester of an action from the receiver

- Very common in mobile or rich UI development



```
// execute a command on click via binding  
<Button Command="{Binding SomeCommand}" Content="A button"/>  
  
// manually execute a command on click  
void SomeButton_Click(Object sender, EventArgs e)  
{  
    var viewModel = (AViewModelClass)DataContext;  
    if (viewModel.SomeCommand.CanExecute())  
    {  
        viewModel.SomeCommand.Execute();  
    }  
}
```

Describing the Command Pattern



```
// execute a command on click via binding  
<Button Command="{Binding SomeCommand}" Content="A button"/>  
  
// manually execute a command on click  
void SomeButton_Click(Object sender, EventArgs e)  
{  
    var viewModel = (AViewModelClass)DataContext;  
    if (viewModel.SomeCommand.CanExecute())  
    {  
        viewModel.SomeCommand.Execute();  
    }  
}
```

Describing the Command Pattern



```
// execute a command on click via binding  
<Button Command="{Binding SomeCommand}" Content="A button"/>  
  
// manually execute a command on click  
void SomeButton_Click(Object sender, EventArgs e)  
{  
    var viewModel = (AViewModelClass)DataContext;  
    if (viewModel.SomeCommand.CanExecute())  
    {  
        viewModel.SomeCommand.Execute();  
    }  
}
```

Describing the Command Pattern



Describing the Command Pattern

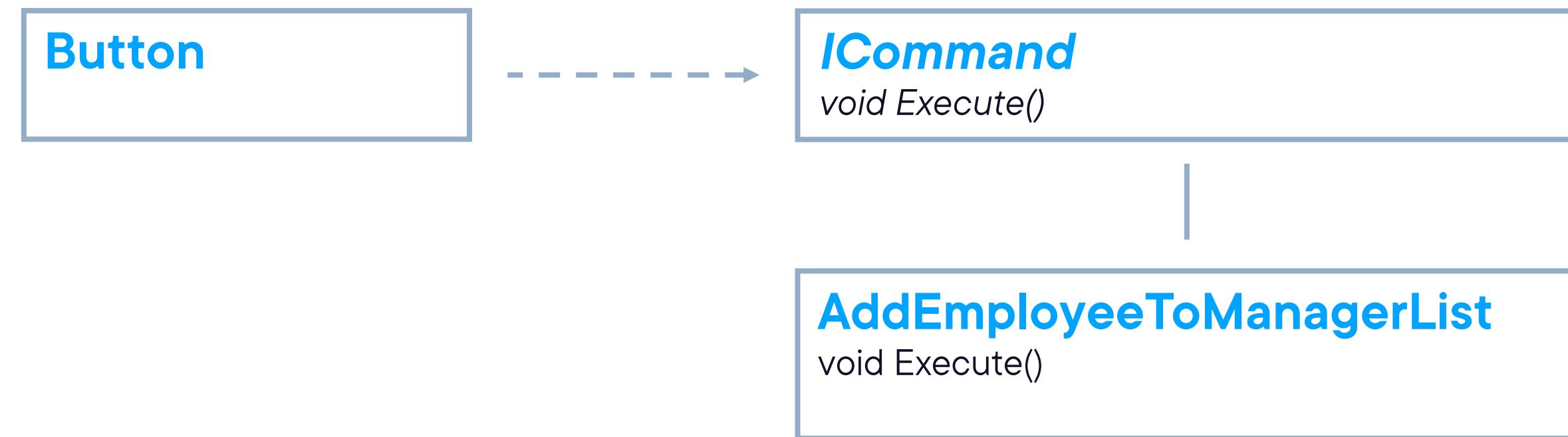
Button



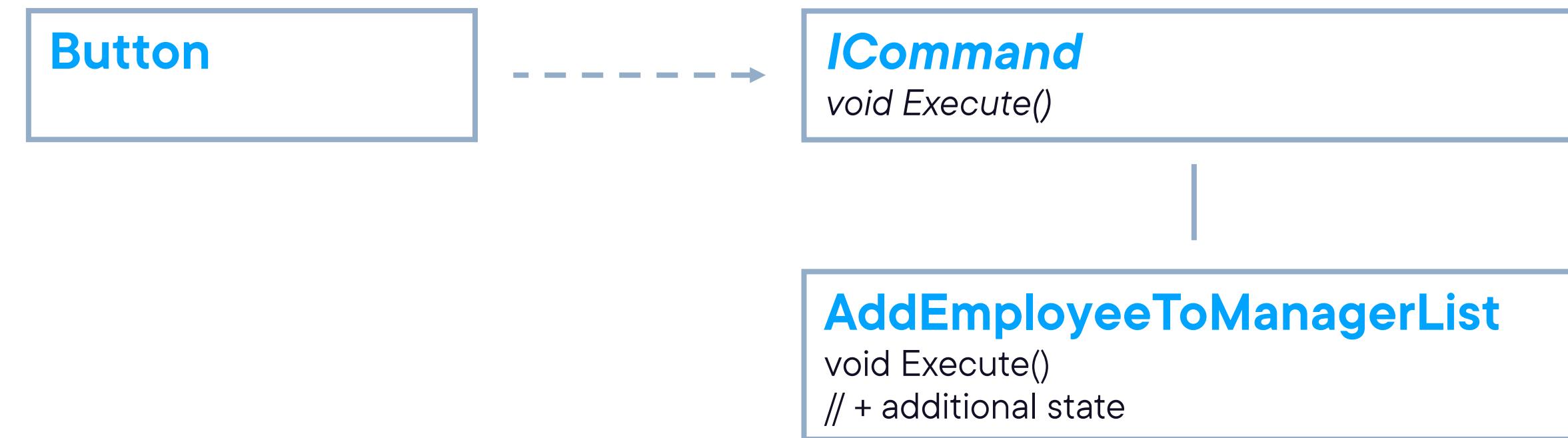
Describing the Command Pattern



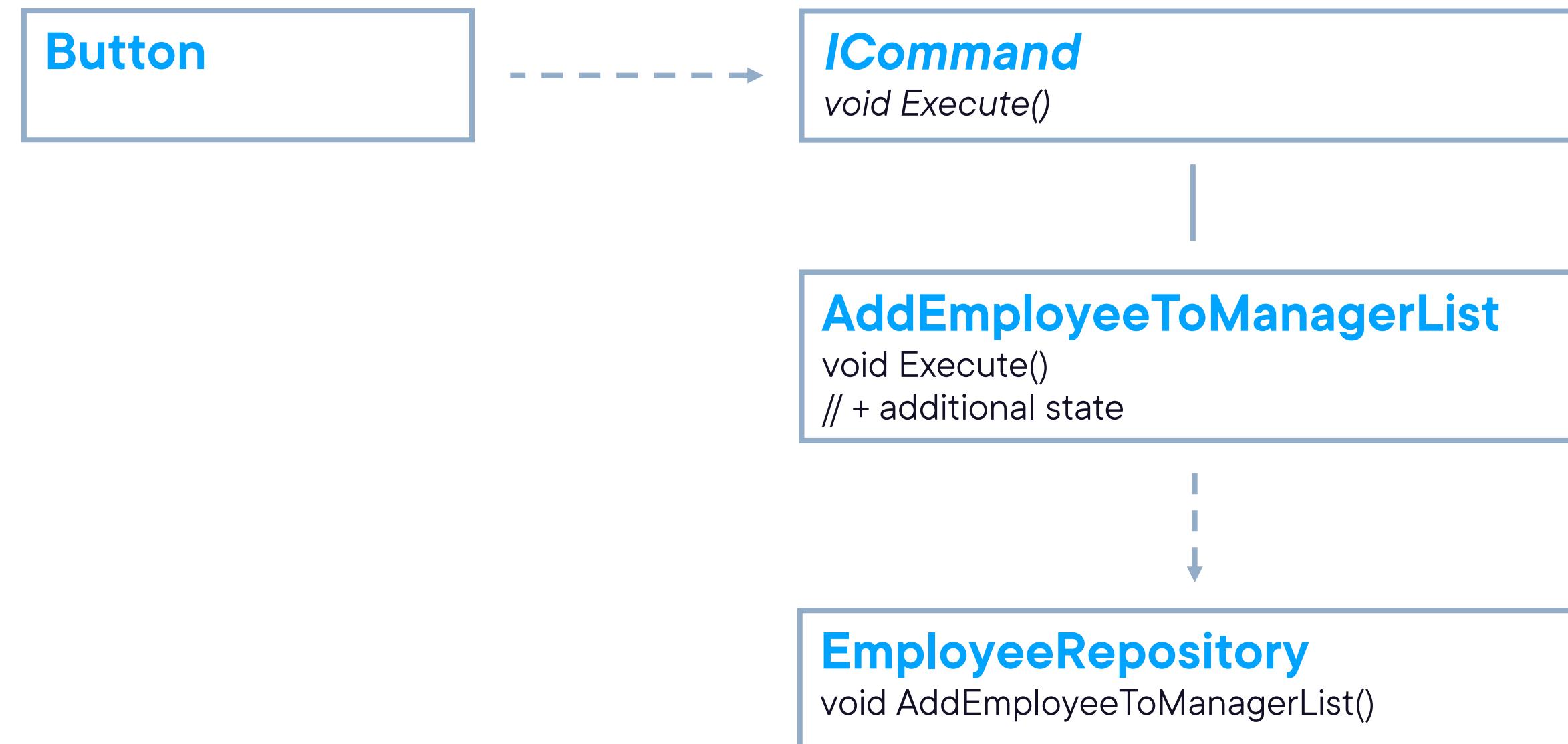
Describing the Command Pattern



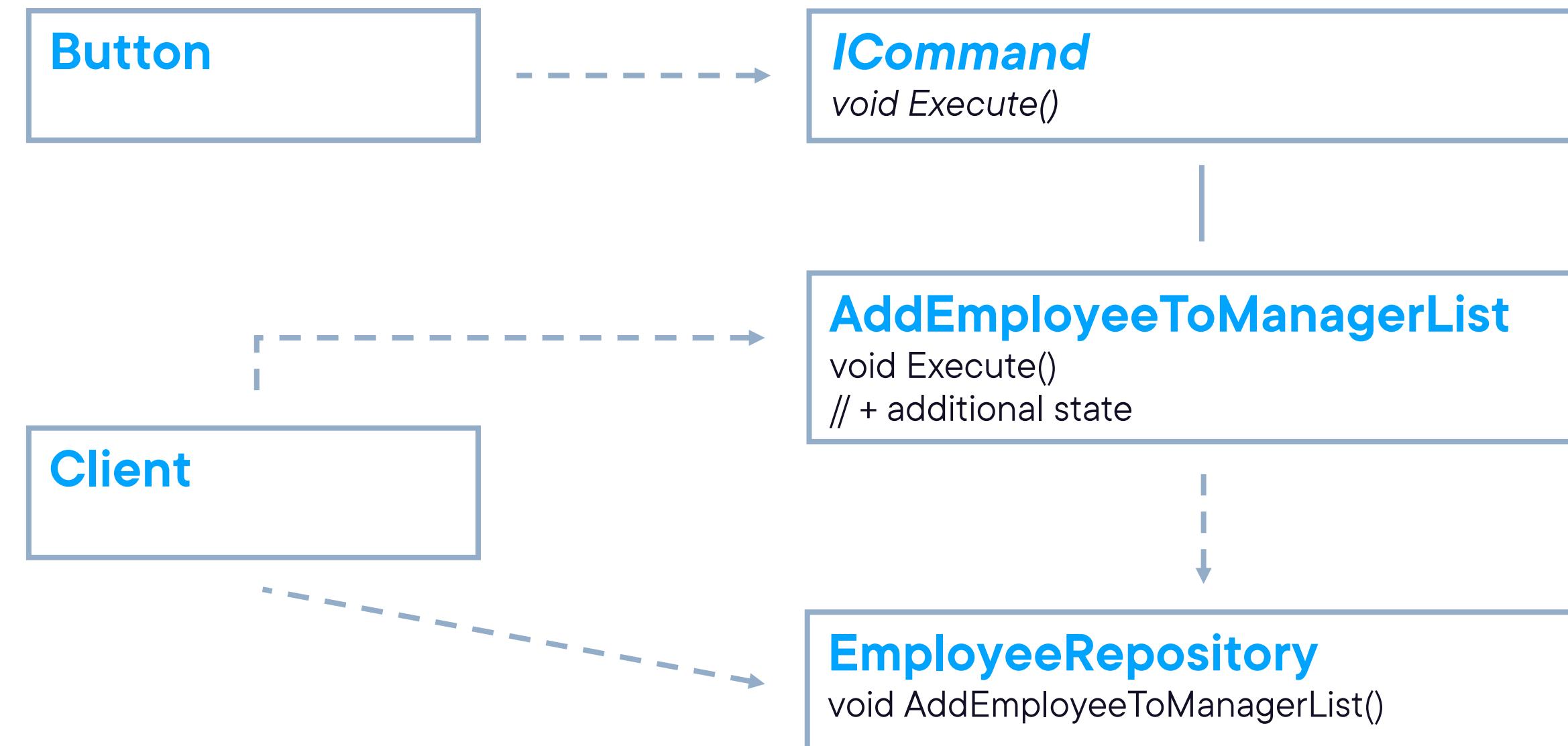
Describing the Command Pattern



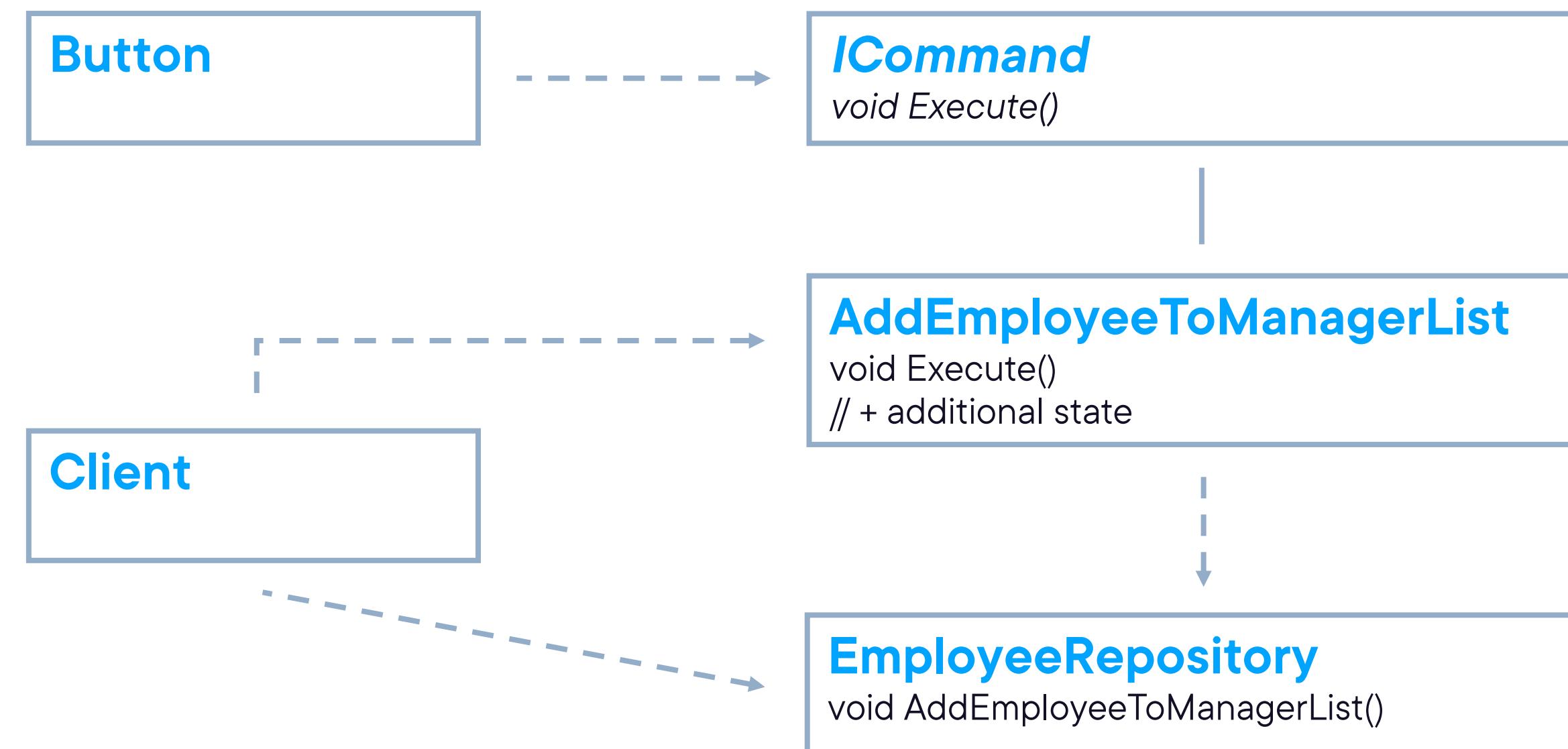
Describing the Command Pattern



Describing the Command Pattern



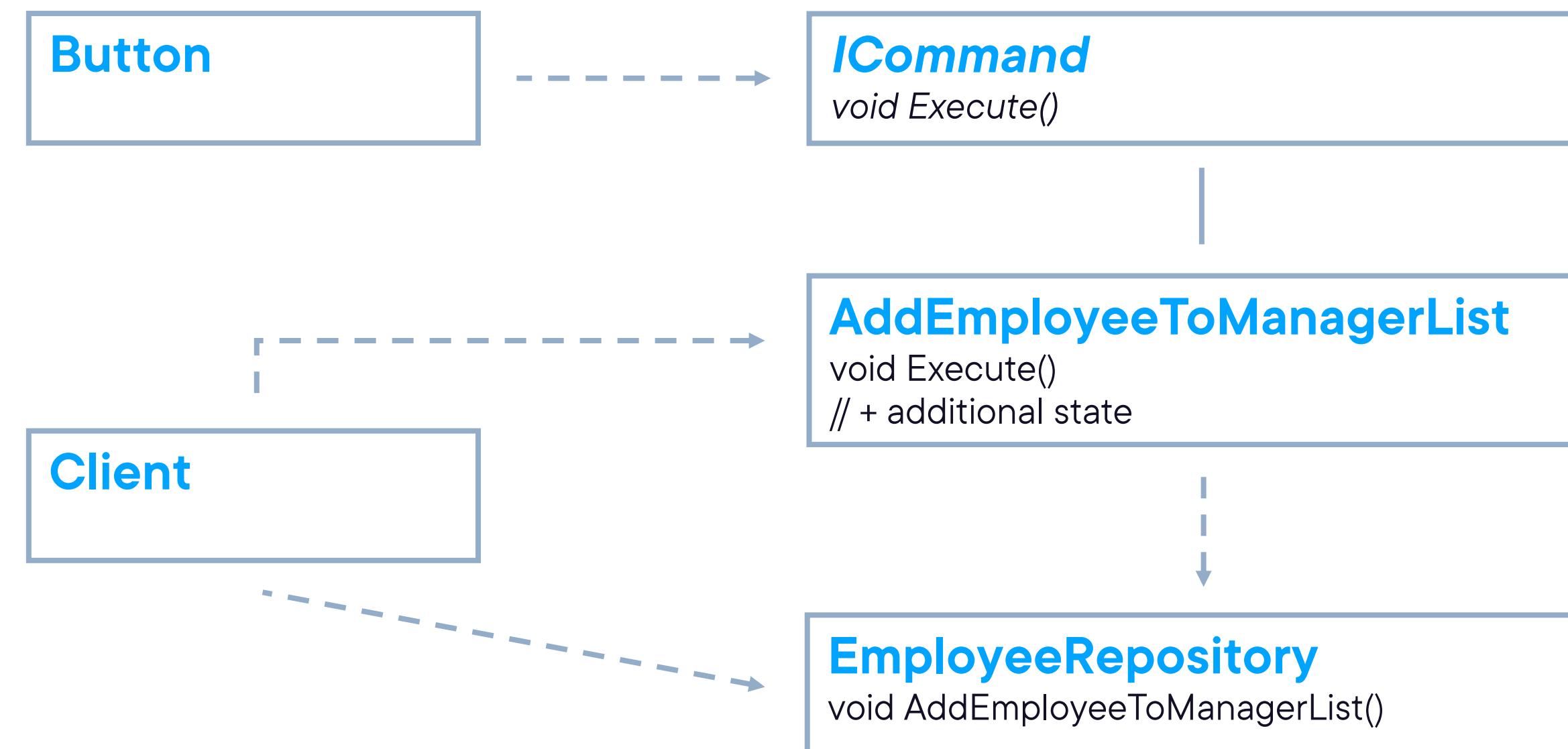
Structure of the Command Pattern



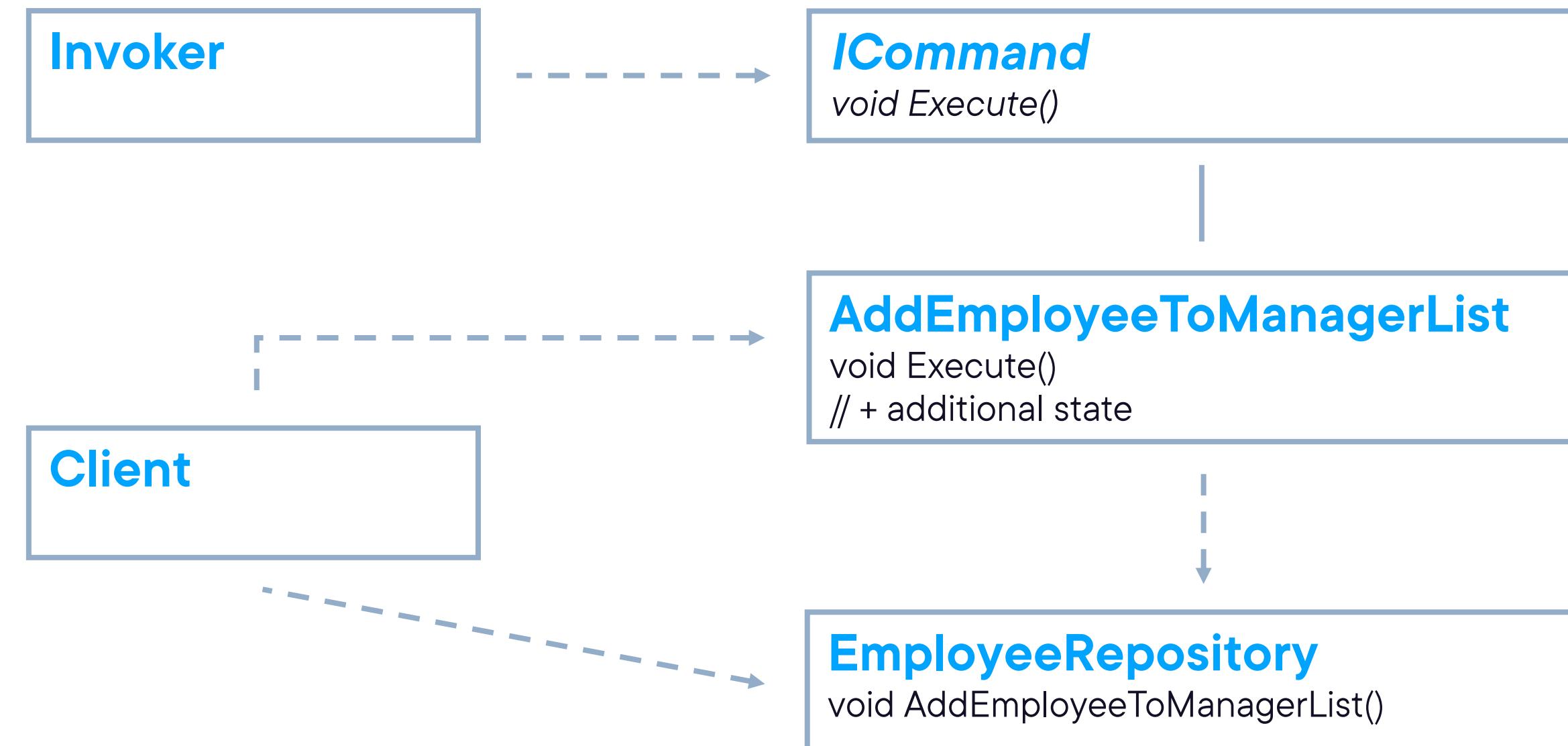
Invoker asks Command to carry out a request



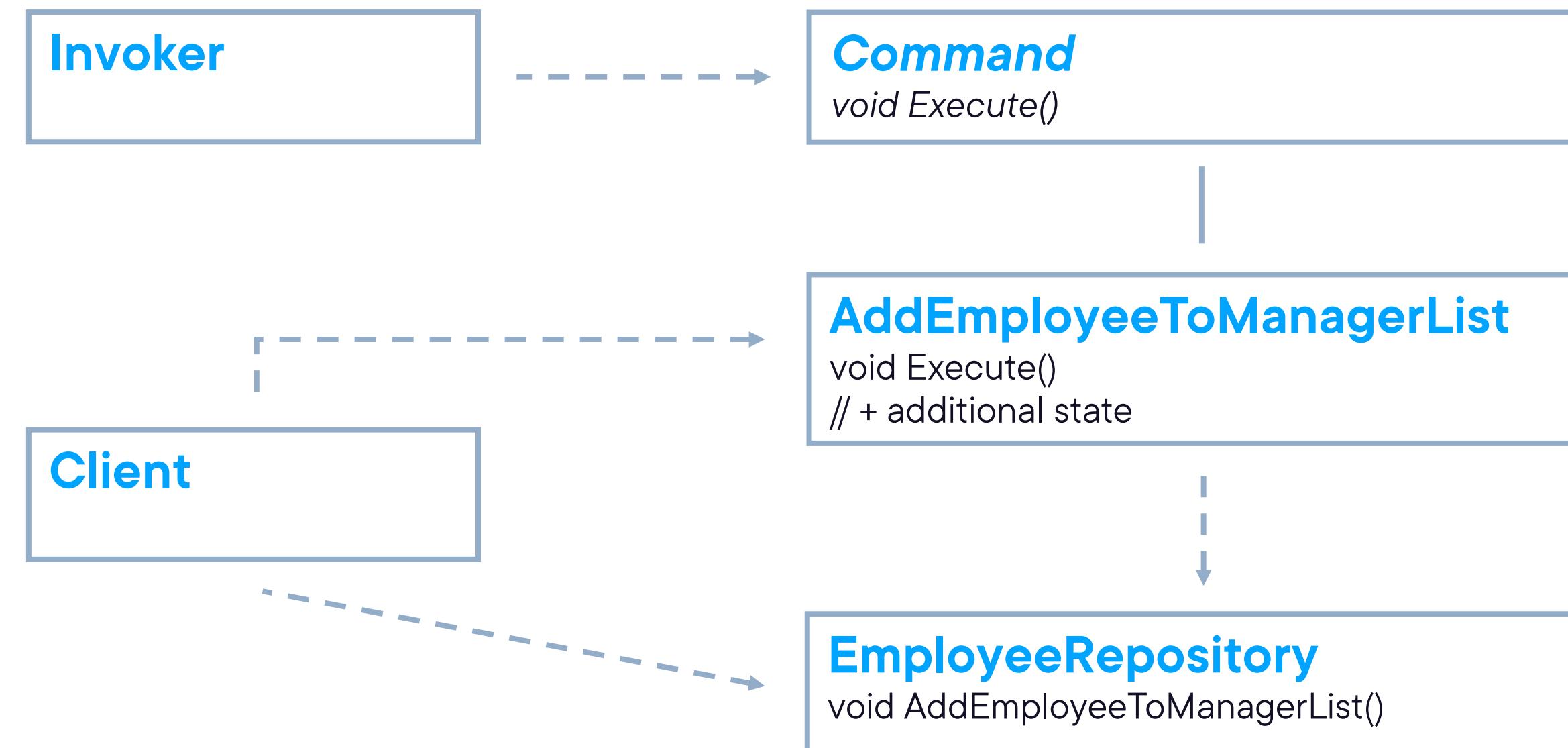
Structure of the Command Pattern



Structure of the Command Pattern



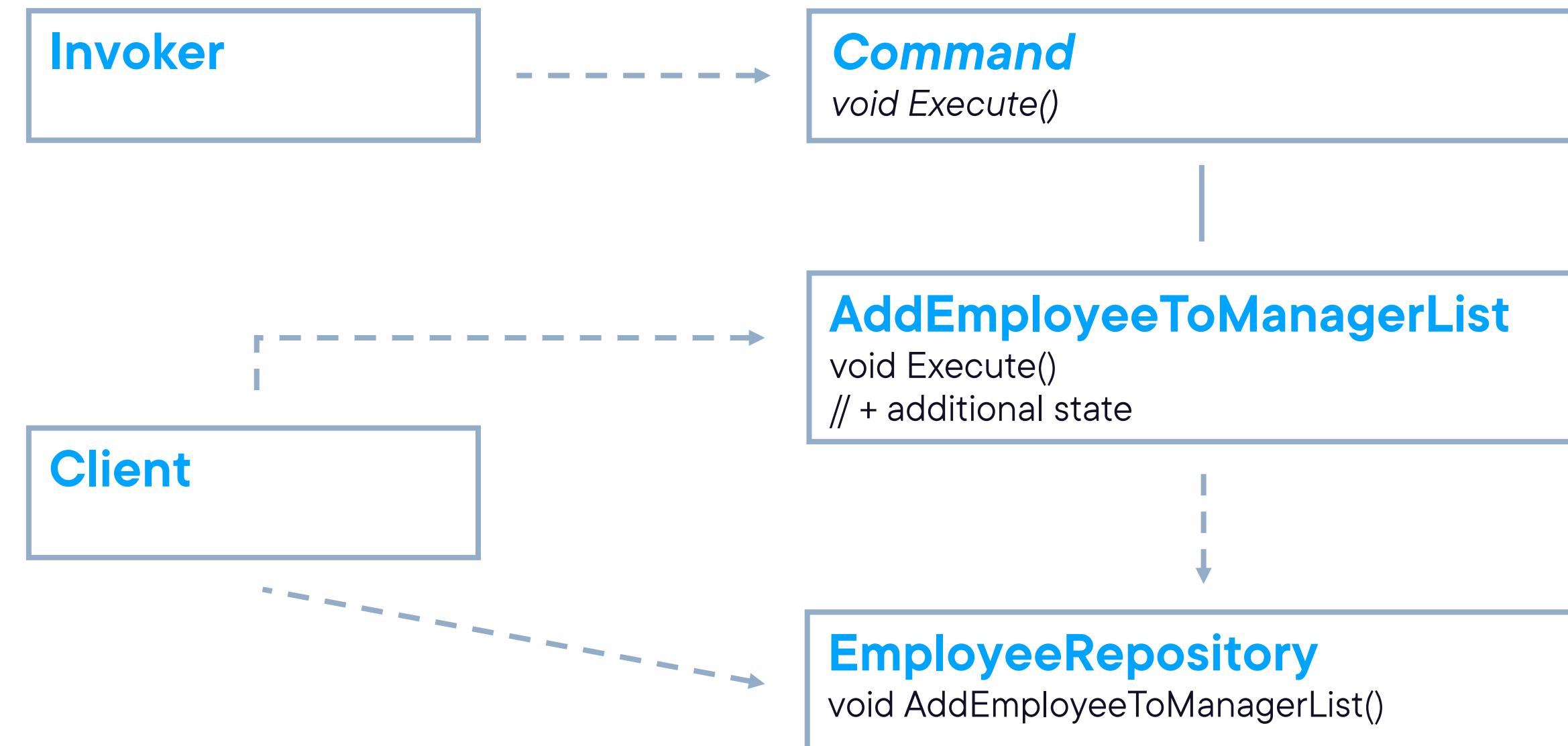
Structure of the Command Pattern



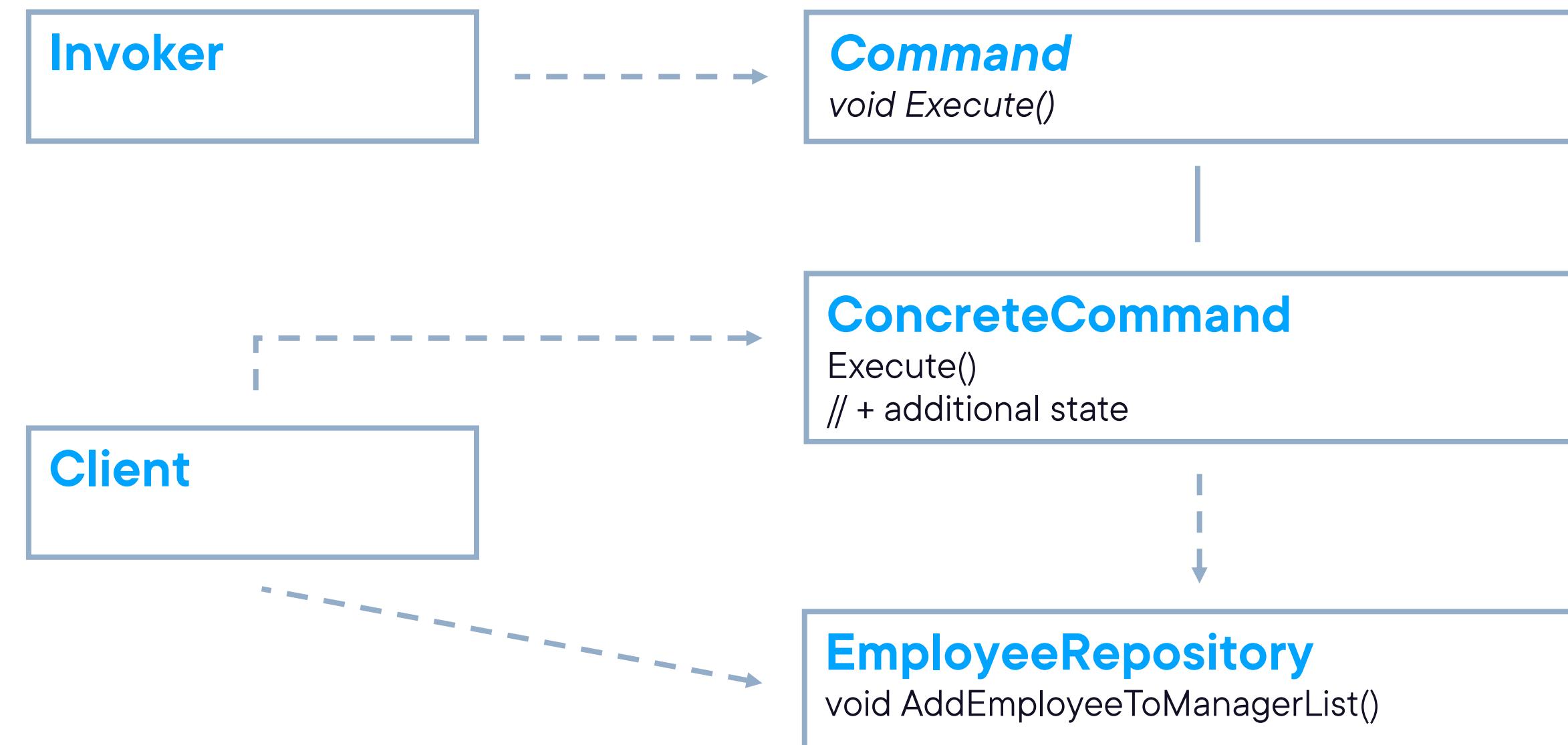
**Command declares an
interface for executing an
operation**



Structure of the Command Pattern



Structure of the Command Pattern



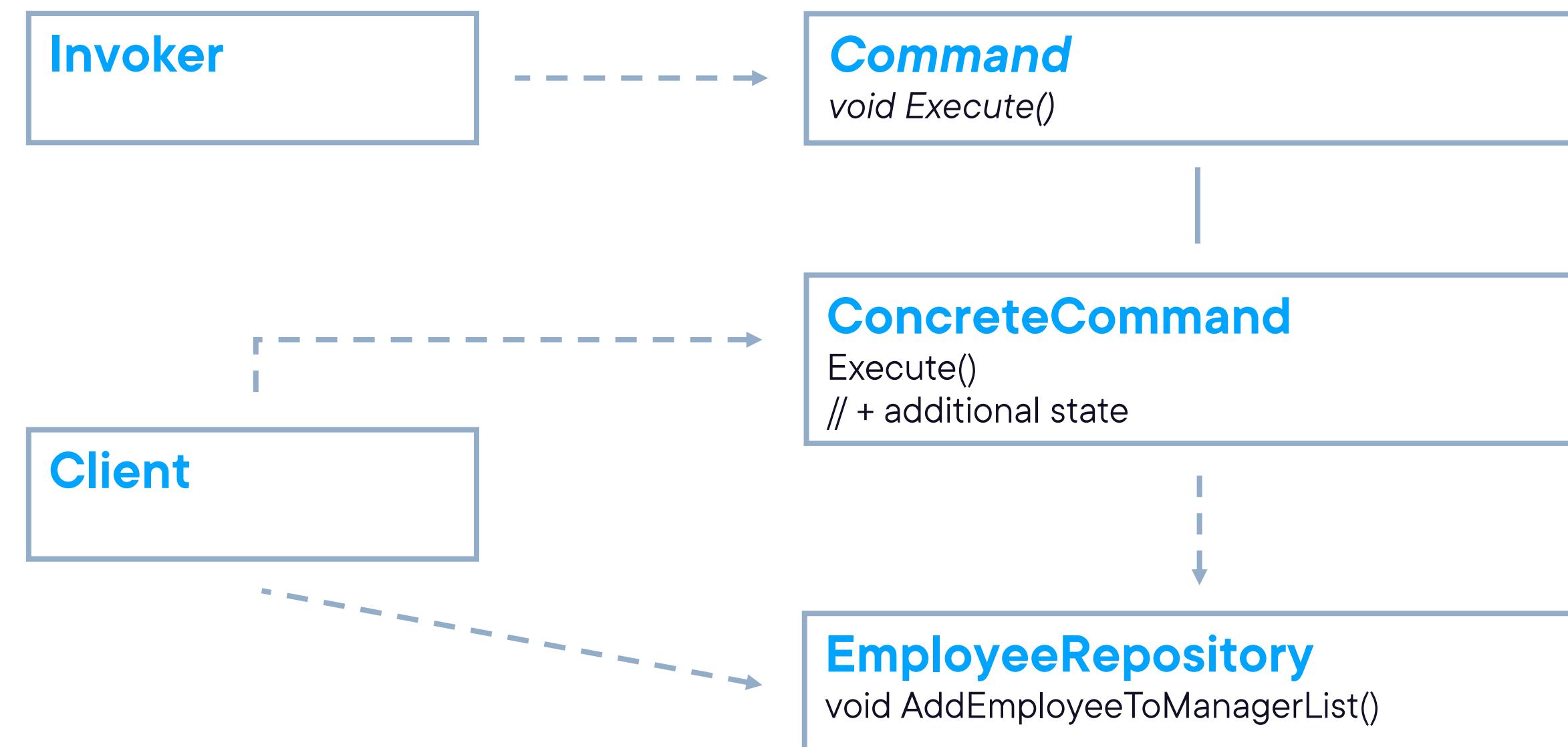
ConcreteCommand defines a binding between a Receiver and an action. It implements Execute by invoking the corresponding operation(s) on Receiver.



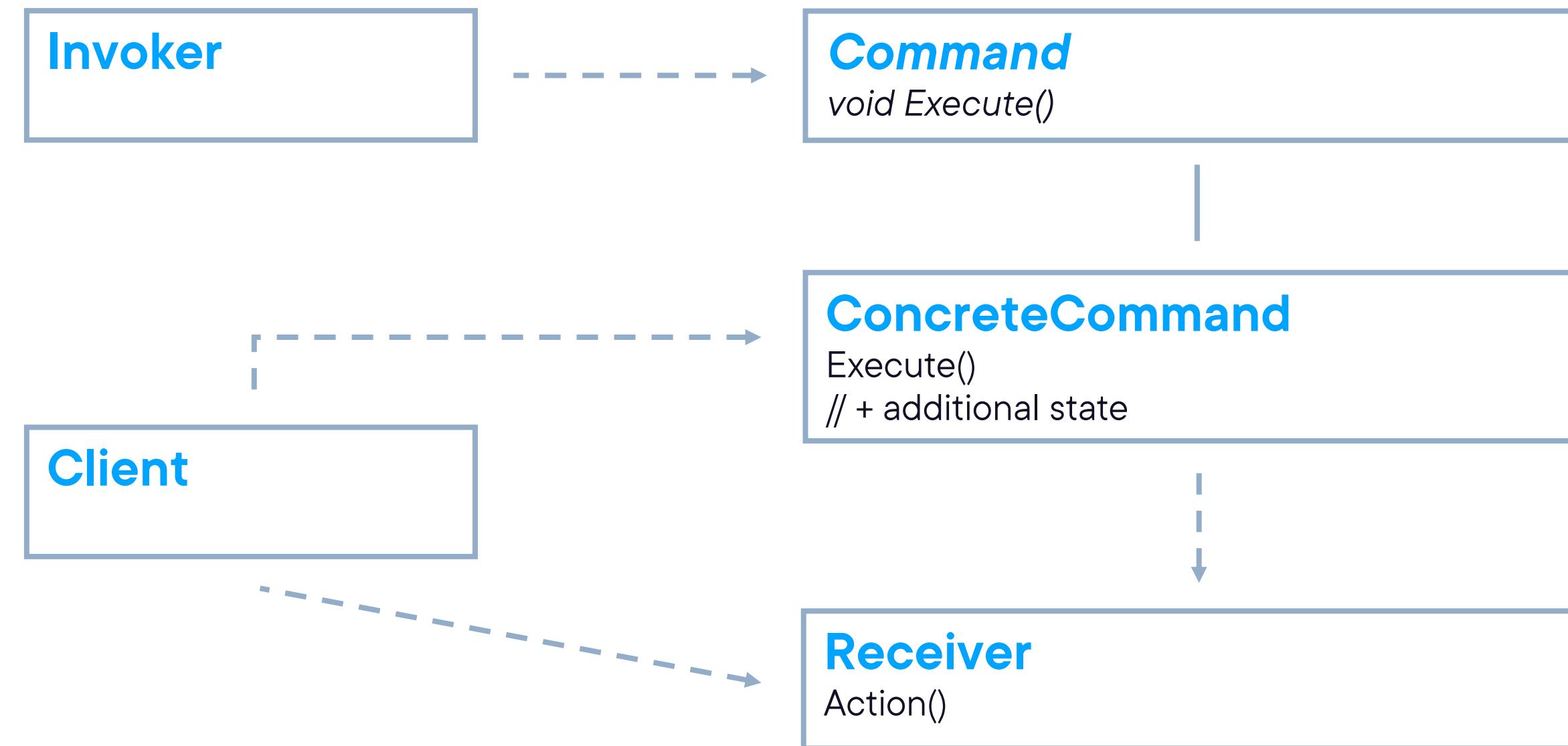
**Receiver knows how to
perform the operations
associated with carrying
out a request**



Structure of the Command Pattern



Structure of the Command Pattern



**Client creates the
ConcreteCommand and sets
its Receiver**



Demo



Implementing the command pattern



Demo



Supporting undo with a command manager



Use Cases for the Command Pattern



When you want to parameterize objects with an action to perform



When you want to support undo



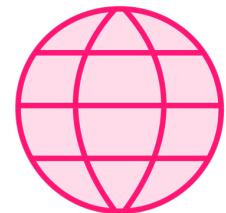
When you want to specify, queue and execute requests at different times



When you need to store a list of changes to potentially reapply later on



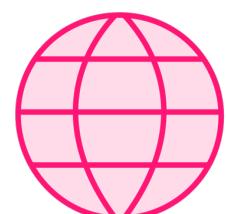
Use Cases for the Command Pattern



Financial transactions



Workflow management



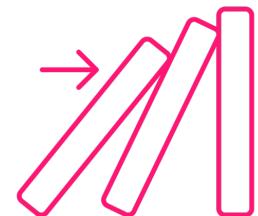
Macro recording and playback



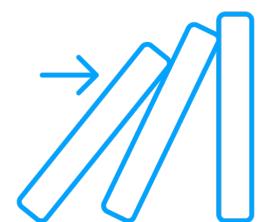
Game engines



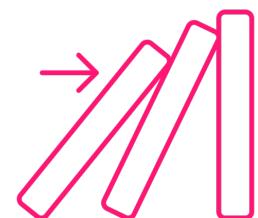
Pattern Consequences



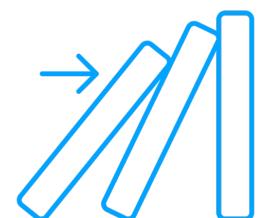
It decouples the class that invokes the operation from the one that knows how to perform it: **single responsibility principle**



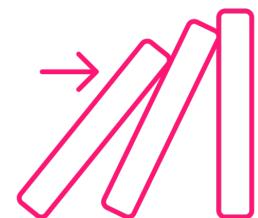
Commands can be manipulated and extended



Commands can be assembled into a composite command



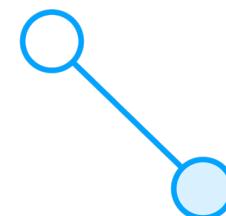
Existing implementations don't have to be changed to add new commands: **open/closed principle**



Because an additional layer is added, complexity increases

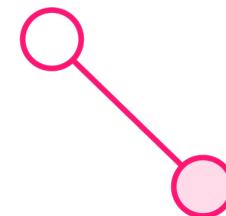


Related Patterns



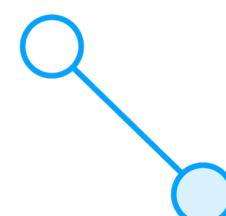
Composite

Can be used to implement commands composed of other commands



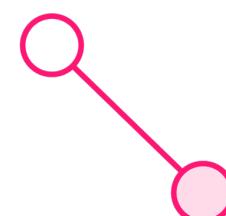
Memento

Can be used to store the state a command requires to undo its effect



Prototype

In case of supporting undo, a command that must be copied acts as a prototype

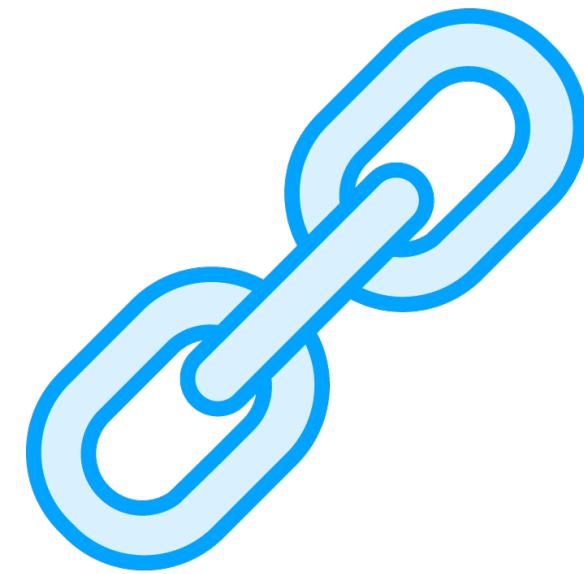


Chain of Responsibility

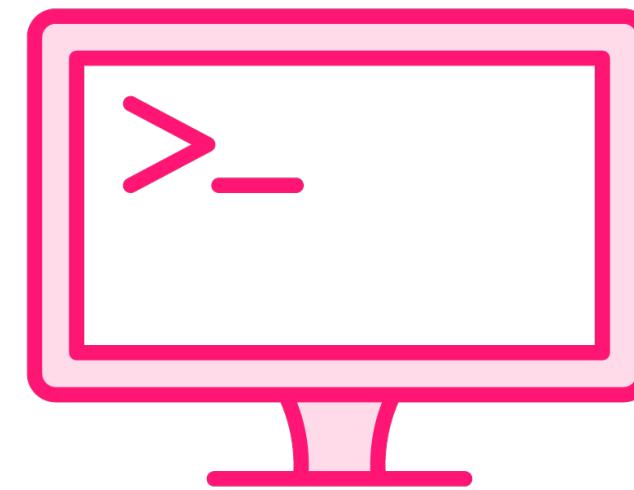
Handlers can be implemented as commands



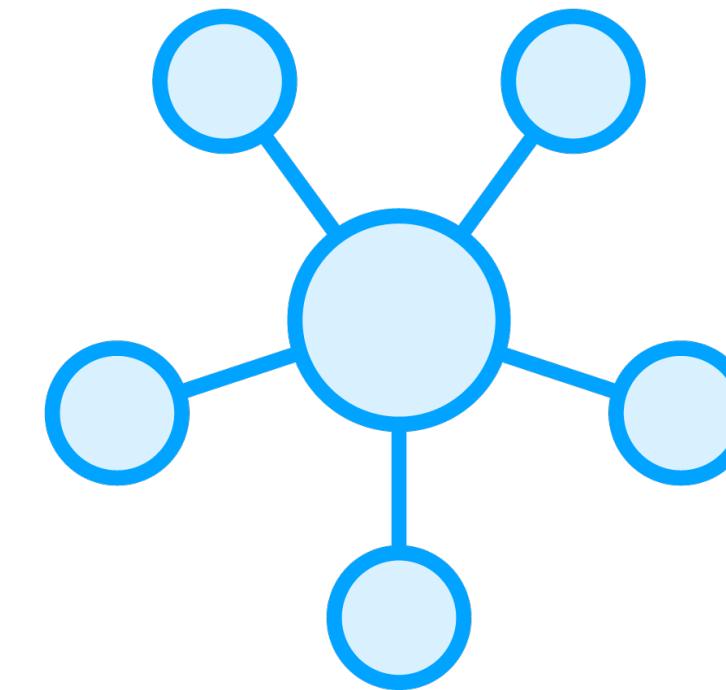
Patterns that Connect Senders and Receivers



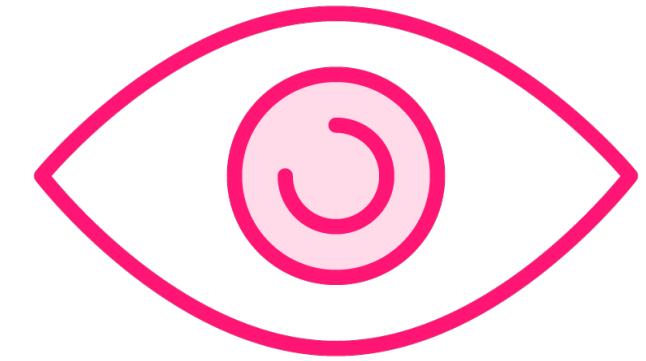
Chain of Responsibility
Passes a request along a chain of receivers



Command
Connects senders with receivers unidirectionally



Mediator
Eliminates direct connections altogether



Observer
Allows receivers of requests to (un)subscribe at runtime



Summary



Intent of the command pattern:

- To encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations



Summary



Implementation:

- Define methods on Command
- Implement on ConcreteCommand
- Invoker is often a UI element
- Receiver can be any object

Consider using a command manager



Up Next:

Behavioral Pattern: Memento

