Command Design Pattern

Intent

- Encapsulate a request as an object, thereby letting you parametrize clients with different requests, queue or log requests, and support undoable operations.
- Promote "invocation of a method on an object" to full object status
- An object-oriented callback

Problem

Need to issue requests to objects without knowing anything about the operation being requested or the receiver of the request.

Discussion

Command decouples the object that invokes the operation from the one that knows how to perform it. To achieve this separation, the designer creates an abstract base class that maps a receiver (an object) with an action (a pointer to a member function). The base class contains an `execute()` method that simply calls the action on the receiver.

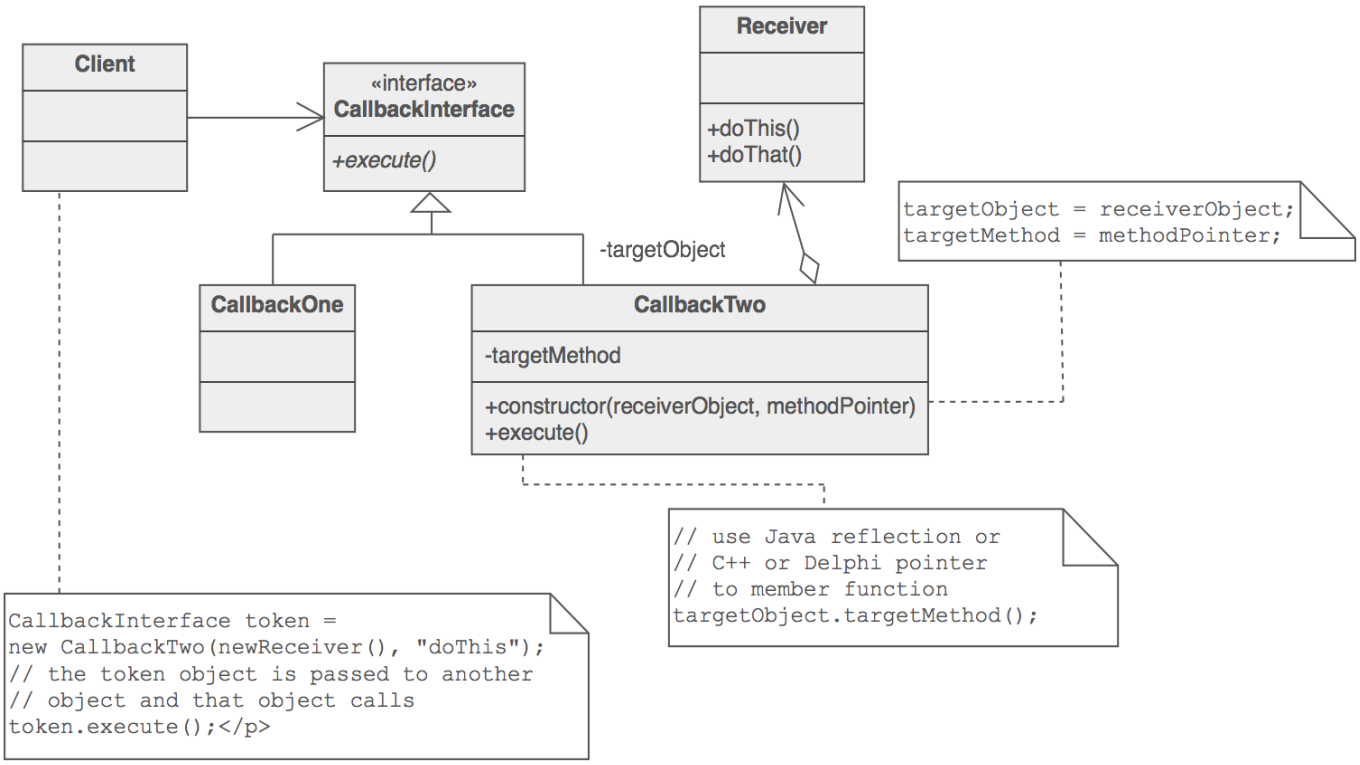
All clients of Command objects treat each object as a "black box" by simply invoking the object's virtual `execute()` method whenever the client requires the object's "service".

A Command class holds some subset of the following: an object, a method to be applied to the object, and the arguments to be passed when the method is applied. The Command's "execute" method then causes the pieces to come together.

Sequences of Command objects can be assembled into composite (or macro) commands.

Structure

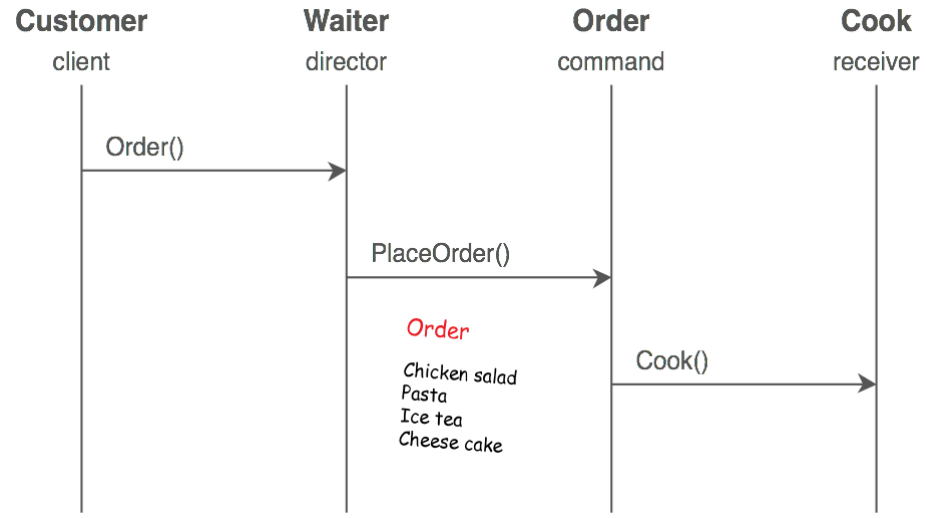
The client that creates a command is not the same client that executes it. This separation provides flexibility in the timing and sequencing of commands. Materializing commands as objects means they can be passed, staged, shared, loaded in a table, and otherwise instrumented or manipulated like any other object.



Command objects can be thought of as "tokens" that are created by one client that knows what need to be done, and passed to another client that has the resources for doing it.

Example

The Command pattern allows requests to be encapsulated as objects, thereby allowing clients to be parametrized with different requests. The "check" at a diner is an example of a Command pattern. The waiter or waitress takes an order or command from a customer and encapsulates that order by writing it on the check. The order is then queued for a short order cook. Note that the pad of "checks" used by each waiter is not dependent on the menu, and therefore they can support commands to cook many different items.



Check list

1. Define a Command interface with a method signature like `execute()`.
2. Create one or more derived classes that encapsulate some subset of the following: a "receiver" object, the method to invoke, the arguments to pass.
3. Instantiate a Command object for each deferred execution request.
4. Pass the Command object from the creator (aka sender) to the invoker (aka receiver).
5. The invoker decides when to `execute()`.

Rules of thumb

- Chain of Responsibility, Command, Mediator, and Observer, address how you can decouple senders and receivers, but with different trade-offs. Command normally specifies a sender-receiver connection with a subclass.
- Chain of Responsibility can use Command to represent requests as objects.
- Command and Memento act as magic tokens to be passed around and invoked at a later time. In Command, the token represents a request; in Memento, it represents the internal state of an object at a particular time. Polymorphism is important to Command, but not to Memento because its interface is so narrow that a memento can only be passed as a value.
- Command can use Memento to maintain the state required for an undo operation.
- MacroCommands can be implemented with Composite.
- A Command that must be copied before being placed on a history list acts as a Prototype.

- Two important aspects of the Command pattern: interface separation (the invoker is isolated from the receiver), time separation (stores a ready-to-go processing request that's to be stated later).

Read next

This article is taken from our book **Design Patterns Explained Simply**.

All of the design patterns are compiled there. The book is written in clear, simple language that makes it easy to read and understand (just like this article).

We distribute it in PDF & EPUB formats so you can get it onto your iPad, Kindle, or other portable device immediately after your purchase.

Learn more

Code examples

Java	**Command in Java: Decoupling producer from consumer**	**Command in Java**	
C++	**Command in C++: Before and after**	**Command in C++**	**Command in C++: Simple and 'macro' commands**
PHP	**Command in PHP**		
Delphi	**Command in Delphi**		
Python	**Command in Python**		

READ NEXT

Interpreter

RETURN

Design Patterns
AntiPatterns
Refactoring
UML

My account
Forum
Contact us
About us