Command Design Pattern

Purpose and Motivation

The motivation comes from macro. Macro is defined as “an object that holds a name, symbol or key that represents a list of commands, actions or keystrokes”, which is similar to the Command design pattern. The main purpose is to encapsulate the request as an object, thereby letting developers to parameterize clients with different requests. This transformation allows developers to parameterize methods with different requests, delay or queue the request's execution and support otherwise undoable operations.

Intended Use Cases

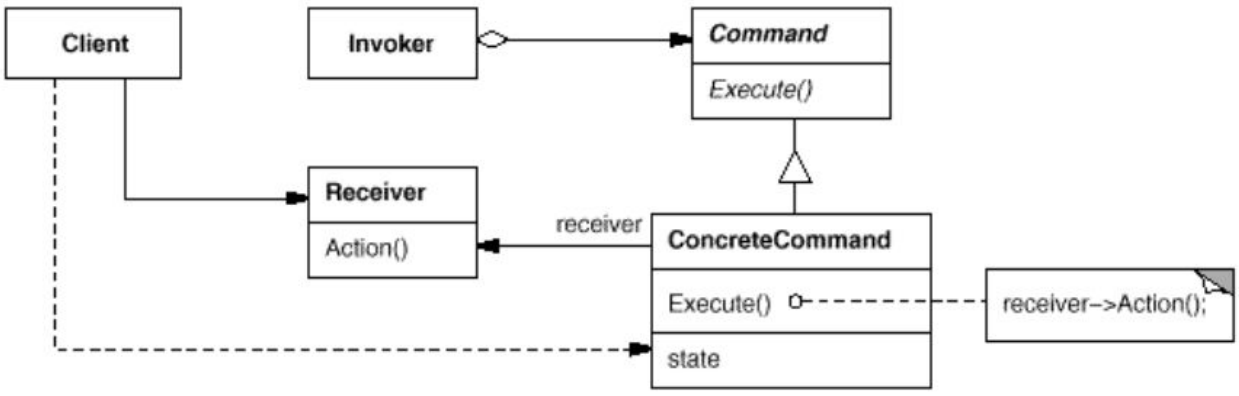
Use the Command pattern when you want to

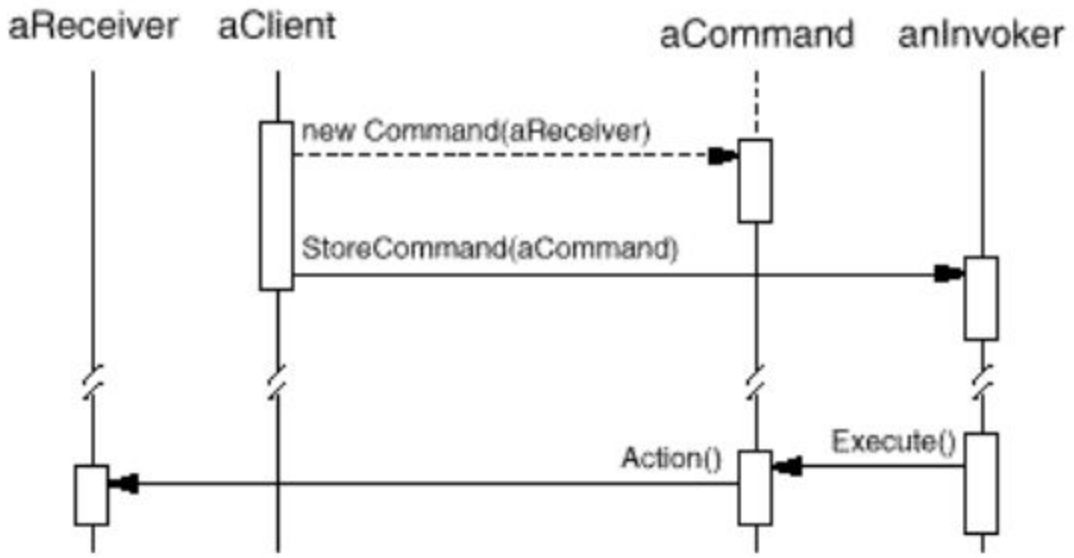
- parameterize objects by an action to perform, as MenuItem objects did above. You can express such parameterization in a procedural language with a callback function, that is, a function that's registered somewhere to be called at a later point. Commands are an object-oriented replacement for callbacks.
- specify, queue, and execute requests at different times. A Command object can have a lifetime independent of the original request. If the receiver of a request can be represented in an address space-independent way, then you can transfer a command object for the request to a different process and fulfill the request there.
- support undo. The Command's Execute operation can store state for reversing its effects in the command itself. The Command interface must have an added Unexecute operation that reverses the effects of a previous call to Execute. Executed commands are stored in a history list. Unlimited-level undo and redo is achieved by traversing this list backwards and forwards calling Unexecute and Execute, respectively.
- support logging changes so that they can be reapplied in case of a system crash. By augmenting the Command interface with load and store operations, you can keep a persistent log of changes. Recovering from a crash involves reloading logged commands from disk and reexecuting them with the Execute operation.
- structure a system around high-level operations built on primitives operations. Such a structure is common in information systems that support transactions. A 221 transaction encapsulates a set of changes to data. The Command pattern offers a way to model transactions. Commands have a common interface, letting you invoke all transactions the same way. The pattern also makes it easy to extend the system with new transactions.

Vocabulary

- **Command**: This is an object that knows about the receiver and invokes a method in the receiver. The values and parameters of the invoked method are stored in the command object. Also, the receiver object to execute these methods are also stored in the command object by aggregation. (Aggregation: Command object may only contain a reference/pointer to the receiver object and does not imply its lifetime, meaning that the command object can outlive the receiver object)
- **Receiver**: The object that actually executes the method when the execute() method in the command object is called
- **Invoker**: The object knows how to execute a command, and optionally does bookkeeping on the command execution. It does not know the concrete object of command and only knows of the command interface.
- **Client**: An object that holds the command, receiver, invoker objects and decides which receiver should be assigned to the command objects, which command it assigns to the invoker and which command to execute at which point.

Structure and Runtime Behaviour





Known Consequences

Positive Consequences

It decouples the code for actually executing the command from the code that calls the command, which provides a separation of duties and improves code maintainability

- It encapsulates the command logic into classes and has a clearly defined interface (e.g The command, the receiver object) which improves flexibility in changing which command to use under different circumstances
- Easier to construct general components that need to delegate, sequence, or execute methods at a time of their own choice without knowing the class of the method or method parameters.
- Full control over the action during runtime. This implies the command can be swapped, set, and invoked at any time when running. This is useful for remapping buttons on a remote or changing weapon actions in a game, for example.
- Better extensibility if new commands need to be added
- Able to create a sequence of commands by providing a queue system
- Able to have a rollback system by implementing rollback methods (Command can store state for reversing its execute() method effect)

Negative Consequences

- Since every command object is a concrete subclass of the command class, we end up having a high volume of classes that need to be implemented and

maintained.

NFPs

Improved NFPs

Maintainability: Good, as the invoker has no clue how the action is implemented. They only invoke the command's action when necessary. Thus, the action can be modified freely. Commands are not interdependent either, so they may be changed without affecting each other.

Inhibited NFPs

Complexity: The pattern is generally unintuitive and complex on first glance.

Credits

Students:

JADS (Shruti Basil, Ana Cheung, Daniel Whitney, Junyu Zhou)

THAT (Anand Matthew, Thomas Polkowski, Rim Romanski, Howard Tsai)

Instructor and TAs:

Mei Nagappan, Arman Naeimian, Ivens Portugal