Command Revisited

Markus Voelter

voelter@acm.org

**Michael Kircher**

Michael.Kircher@siemens.com

This pattern revisits the Command [GoF] and Command Processor [POSA1] patterns. The reasons for this revisiting is that we think that the Command and Command Processor patterns do not really capture the essence of what the Command pattern is. We think that Command is basically a way to emulate the concept of closures in object-oriented languages that don't natively have this feature.

The Command Revisited pattern packages a piece of application functionality as well as its parameterization in an object in order to make it usable in another context, such as later in time or in a different thread.

Example

Suppose you are building a complex communication software. The individual sub-sequences of the protocol are similar, but not quite identical. The sub-sequences contain logic that operates on some form of state, also called context. For the various sub-sequences, adapted context information is necessary.

Because of maintainability and to keep the footprint small—the communication software is to be used in an embedded device—you want to reuse repeating sub-sequences.

Step A Step B Step D Step C I II
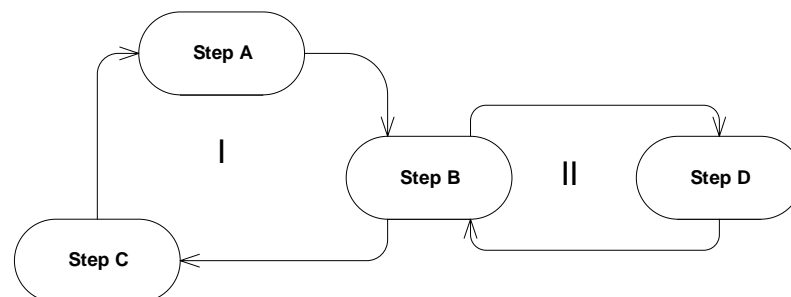
For example in the figure above you can detect process step "B" be involved in protocol I as well as in protocol II. The state on which the sub-sequence depends is similar in both cases. The isolation of reusable sub-sequences is the result of an in-depth analysis of the communication protocol. How can

you encapsulate the protocol sub-sequences as a resuable software building block?

Context

Applications that need to execute application logic in a different execution context, such as later in time or in a different thread.

Problem

When building object-oriented systems, it is frequently necessary to separate the decision of what piece of code should be executed from the decision of when this should happen.

In the example above, a button is a generic object that, when pressed, executes a piece of behavior. In order to make this possible, the button must be configured with this piece of behavior. Once the button is pressed, the behavior is executed, whatever the behavior does specifically.

Another example are internal iterators. These are functions that iterate over a collection of elements and execute a piece of functionality for each element. Again, it is necessary to configure the iterator with the functionality it should execute.

The following **forces** must be addressed:

- *Context independent execution*. The application logic should be executable independent of the context, for example independent of the thread or state.

- *Parameterization*. The application logic should be configurable via parameters.

- *Decoupling*. Executing the behavior should not required detailed knowledge about the specific behavior executed.

Solution

Encapsulate a piece of functionality in a command object. Provide a generic interface to allow the execution of the behavior independent of the behavior itself. Attributes of the object carry the parametrization needed during its execution.

A creator instantiates the command object, providing the necessary context attributes. The command is then passed to its execution context. The execution of the command is triggered via a generic interface by an external event.

Since the execution interface is generic, the execution context does not need to know about the specific behavior that is encapsulated in the command object.

Structure

The following participants form the structure of the Command Revisited pattern:

A *creator* creates command objects.
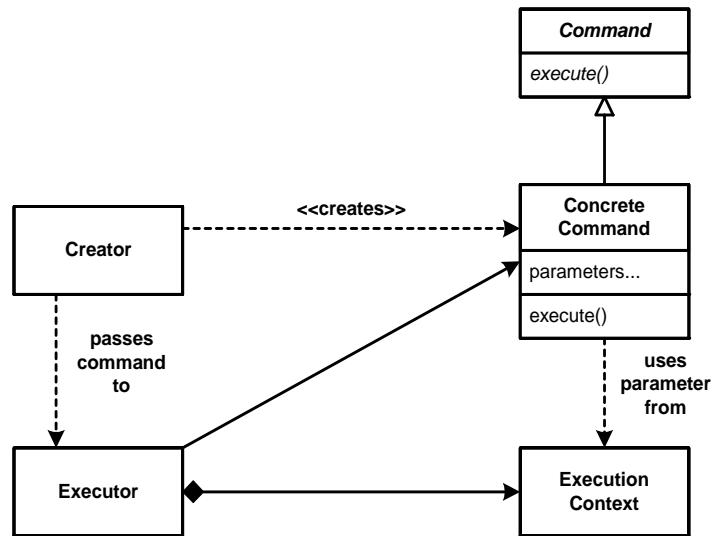
A *command object* contains the application logic.

An *execution context* provides the state and run-time environment for the command object.

An *executor* triggers the execution.

The following CRC cards describe the responsibilities and collaborations of the participants.

Class Creator	*Collaborator* • Executor
Responsibility • Creates a command object of the required type. • Parameterizes the command object with values from its own execution context. • Passes the command object to its new execution context.	

Class Command Object	*Collaborator* • Execution Context
Responsibility • Encapsulates the behavior to be executed. • Carries the parametrization required to execute the encapsulated application logic.	

Class Execution Context	*Collaborator*
Responsibility • Represents the environment in which the command object is executed.	

| *Class* Executor	*Collaborator* • Command Object • Execution Context
Responsibility • Configures the command via parameters. • Executes the command in the execution context.	

The following diagram shows a UML class diagram that illustrates the structural relationships.

```
                                          ┌─────────────────┐
                                          │     Command     │
                                          ├─────────────────┤
                                          │    execute()    │
                                          └─────────────────┘
                                                   △
                                                   │
    ┌─────────────┐      <<creates>>      ┌─────────────────┐
    │             │ - - - - - - - - - - ->│    Concrete     │
    │   Creator   │                       │    Command      │
    │             │                       ├─────────────────┤
    └─────────────┘                       │   parameters... │
          ┊                               ├─────────────────┤
          ┊ passes                        │    execute()    │
          ┊ command                       └─────────────────┘
          ┊ to                                     ┊ uses
          ▽                                        ┊ parameter
    ┌─────────────┐                                ┊ from
    │             │                       ┌─────────────────┐
    │  Executor   │◆──────────────────────>│   Execution     │
    │             │                       │    Context      │
    └─────────────┘                       └─────────────────┘
```

Dynamics

The creator decides which behavior should be executed in the execution context by instantiating a command object of a suitable type. It parametrizes the command object by setting its context attributes to the required values from its own context. It then passes the command object to the execution context. Later, the executor triggers the execution of the command. The command can access the execution context to access its information.

The following sequence diagram shows the interactions.
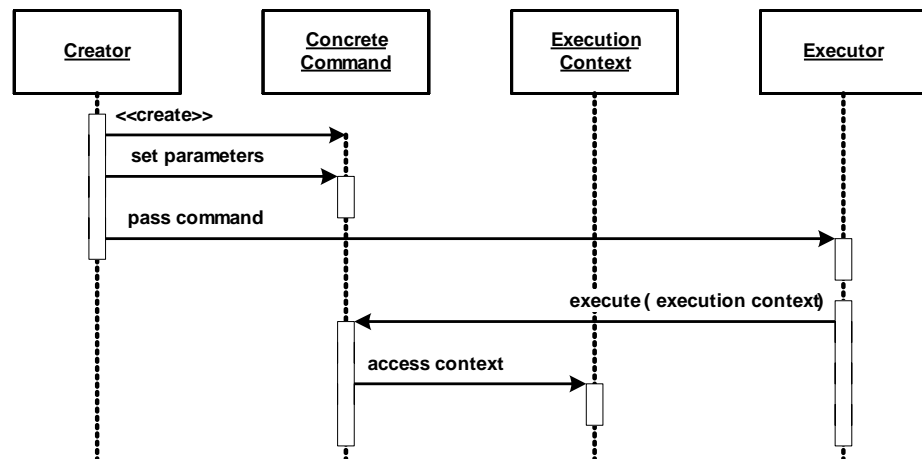
Implementation

There are several steps involved in implementing the Command Revisited pattern.

1. Define an abstract class as generic interface for command execution that will be used by the executor. You will typically define an `execute()` operation.

2. Add the state which the concrete commands need during their execution to the execution context. Make the execution context available to the concrete command.

3. Define and implement the creator, for example using patterns like Abstract Factory [GoF] or Factory Method [GoF].

4. Define the execution context. If necessary allow it to keep references to command objects, but be aware of lifecycle issues.

5. Implement your specific command functionality in subclasses of the abstract command class defined above. This includes:

   5.1 Implementing the `execute()` operation according to your specific requirements.

   5.2 If a specific command needs temporary state during execution, add the necessary attributes to the concrete command class.

Example Resolved

Imlement the common sub-sequences of the protocol logic as command objects. Factor our the state they have in common and present the encapsulated state as execution context. The state all sub-sequences have in common gets encapsulated in base classes. Sub-sequence specific state is kept in derived classes.
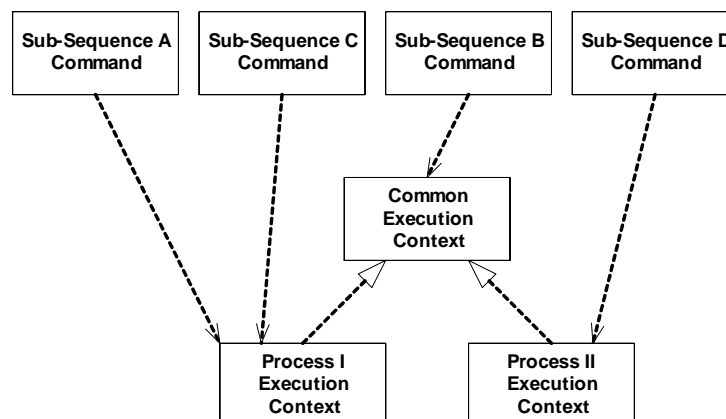
The benefits of the this design are the high degree of reusability and the improved maintainability. When parts of a protocol change, only isolated

parts of the software must be changed. Also, when the state on which the protocol operates changes, the relevant parts can be exchanged between the context base class and the a derived class.

Variants

There are two variation points in this pattern:

- A command may need to access state that is determined by the creator. In this case, the command object has to *remember* the state determined by the creator. Thus, for each parameter the `execute()` operation accesses, the command class needs to have an attribute. The creator passes the parameters to the constructor, which assigns the respective values to the attributes. They can then be accessed during execution.

- A second variation point is how the command accesses the execution context. In some cases it might be implicit (for example, since the context is global), sometimes it needs to be given access explicitly. In that case, the `execute()` operation has to have the respective parameters. The executor has to pass these values to the `execute()` operation.

Consequences

There are several **benefits** of using the Command Revisited pattern:

- *Time-independent execution of application logic*. The encapsulation of application logic allows to queue it and execute it at a different point in time.

- *Context-independent execution of the application logic*. The separation between application logic and context allows to execute the application in separate contexts, such as in a different thread or using a different state.

- *Exchangeability of application logic*. The separation between application logic and context allows to easier exchange the application logic.

There are also some **liabilities** using the Command Revisited pattern:

- *Dependency of the application logic on the state*. If the representation of the state changes slightly, all application logic has to follow that change.

- *Complexity.* Wrapping application logic and parameters in a command adds complexity to the application. This is the penalty for the benefit of context independency.

Known Uses

Commands are used in many cases. Some examples follow:

- *GUI libraries.* As outlined above, GUI commands are the most prominent example. The can be found in Java Swing, in SWT as well as almost any other GUI library.

- *Java.* Commands are often used to implement internal iterators in languages, such as Java, which do not feature them natively.

- *Transactional systems.* Finally, commands are often used in transactional systems. The infrastructure provides transaction handling, the specific behavior that should be executed within the transaction is passed in by clients via a command.

See Also

Futures [Lea99] can be used to allow the creator access to the result of executing the command.

Command objects are a way to emulate closures. Closures are available in many functional and some object-oriented languages. Examples include LISP and Smalltalk, where they are called code blocks.

References

[GoF]   E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns – Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995

[Lea99]   D. Lea, *Concurrent Programming in Java: Design Principles and Pattern*, Addison-Wesley, 1999

[POSA1]   F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture—A System of Patterns*, John Wiley and Sons, 1996