Command Revisited

Markus Voelter
voelter@acm.org

Michael Kircher
Michael.Kircher@siemens.com

This pattern revisits the Command [GoF] and Command Processor [POSA1] patterns.

Command Revisited

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 GUI libraries provide reusable widgets for building graphical user interfaces. These libraries have to be independent of specific application code. Nonetheless, many user interface elements have to interact with application code. For example, when a button is pressed, application code has to be executed. In addition, it is often required to trigger the same functionality from *several* GUI widgets. In many cases, a piece of functionality must be triggered from the main menu, a toolbar as well as a context menu in a tree view.

Using the template method pattern [GoF] is not practical, since it would require subclassing the Button widget for each usage, and it would also bind the implemented functionality to its use in a button.

In essence, the application has to package up some piece of application logic and pass it to a button (or menu item) instance. If the GUI widget is triggered, the application logic has to be executed.

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 need not 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***
Responsibility • Creates a command object of the specific type it requires. • Parametrizes the command object with values from its own execution context. • Passes the command object to its new execution context.	• Requestor

Class Command Object	***Collaborator***
Responsibility • Implements the behavior to be executed. • Carries the parametrization required to execute the encapsulated application logic. • Provides a generic way to the encapsulated behavior independent of the concrete behavior	• Execution Context

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

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

Dynamics The creator decides which behavior should be executed in the other 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 context. The execution context executes the command object in its new context.

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

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

1.1 Decide on whether concrete commands need access to the some parameters in the execution context. If so, add these parameters to the `execute()` operation.

2 Implement the Execution Context. Allow it to keep references to command objects.

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

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

3.2 If a specific command needs to access state from the creation context during execution, add the necessary attributes to the concrete command class. Also make them constructor parameters; the constructor needs to set the values for the attributes.

Example Resolved Application functionality is packaged into the `execute()`operation a specific command class. The application instantiates a command, passing its configuration data to the instance (such as a selected tree item). The application then passes the command object to a button instance and/or a menu item. Later, when the button is pressed, the button implementation simply calls the command's execute operation, triggering the encapsulated application logic.

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 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.

Known Uses. Commands are used in many cases. Some examples follow:

- 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.
- Commands are often used to implement internal iterators in languages which do not feature them natively.
- Finally, commands are often used in transactional systems. The infrastructure provides transaction handling, the specific behaviour that should be executed within the transaction is passed in by clients via a command.

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

- *Time-independent execution of application logic*
- *Thread-independent execution of the application logic*
- *Exchangeability of 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.

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