Metacommand

Markus Völter, voelter@acm.org

Version 0.2, 15.02.2000

INTENT

Metacommand enhances the Command pattern by providing a facility to enhance or modify the common behaviour of the command classes used in a system at any time without modifying the command classes themselves.

PROBLEM AND MOTIVATION

In this paper I assume that the reader is familiar with the Command pattern as it is described by the Gang of Four [1]. In short, Command describes a means to encapsulate requests in objects, thereby allowing well structured integration into user interfaces, providing logging or queueing facilities as well as enabling undo / redo.

However, the pattern requires that these additional features are known when the command classes are programmed. It is not possible to dynamically add or remove features. For example, it would be interesting to add logging facilities to all command classes in a system, if debugging becomes necessary or the customer requires logging of who did what. Or it might be necessary to add or change a permissions check before the execution of every command, and execute the command only if the check allows it. It should be possible to add these features dynamically, at runtime, without modifying the command classes themselves. The last point is especially important, because in general, there are quite many command classes in a system.

The pattern resolves the following forces:

- The common behaviour of your command classes should be kept flexible.
- No additional runtime overhead should be imposed by providing this flexibility
- The solution should be as transparent as possible, i.e. it should be used more or less like the ordinary Command.
- The modification of the common behaviour of the commands should not require changes in the command classes themselves.

APPLICABILITY

This pattern can be used whenever Command is applicable, and when the following additional features are required:

- The common behaviour of the command objects should be kept flexible.
- The modification of the common behaviour of the command classes should be possible without modification of the command's classes themselves.

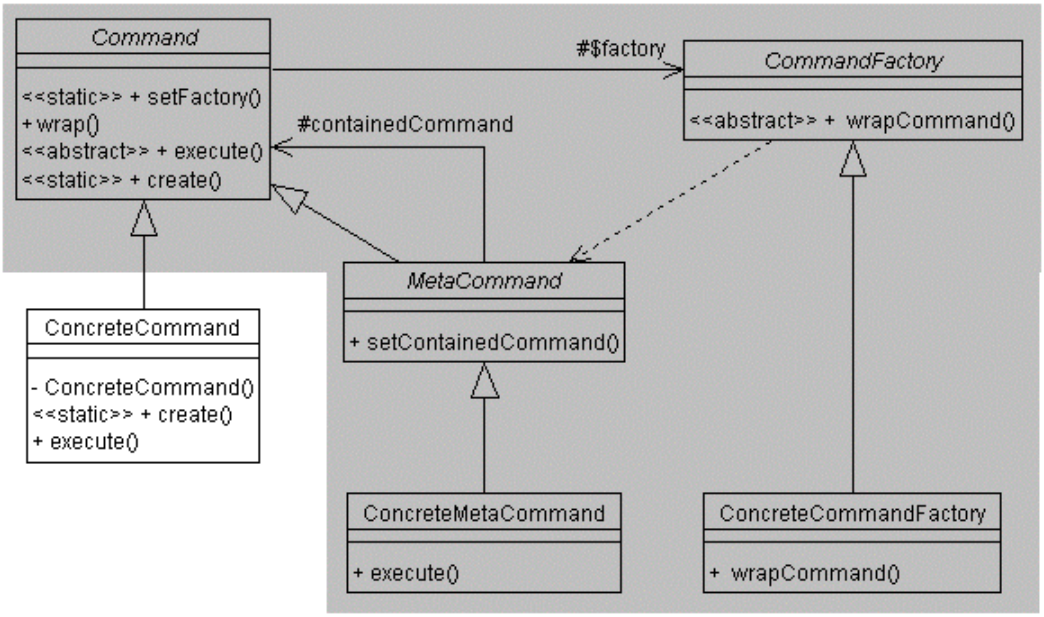
SOLUTION

Outline

Whenever a *ConcreteCommand* object is created, the system creates a *MetaCommand* and wraps the *ConcreteCommand* with it. When the *ConcreteCommand* is executed, the *MetaCommand* is executed instead. The *MetaCommand* is responsible for in turn executing the *ConcreteCommand* during its own execution. The Abstract Factory [1] pattern is used to implement the creation policy.

Structure

The *Command* class is an abstract base class for user-defined commands that are used in the application. Like in the Command pattern, there is an abstract *execute()*method that gets called by the command-executing program entity, independent of the concrete command class.



In addition, the *Command* class contains a static reference to a *CommandFactory* object, which is responsible for wrapping a *ConcreteMetaCommand* object around a *ConcreteCommand* object.

Participants

Command	Abstract base class for application specific concrete command classes.
MetaCommand	Abstract base class for application specific meta command classes. Wraps around (contains) a concrete command object.
CommandFactory	Responsible for setting up the wrapping between the *ConcreteCommand* and the *ConcreteMetaCommand*.
ConcreteMetaCommand	An application specific *MetaCommand* class, that executes the *ConcreteCommand* during its own execution. It contains the code common to *ConcreteCommands*.
ConcreteCommandFactory	A concrete implementation of *CommandFactory* that wraps

	an instance of a *ConcreteMetaCommnad* around a *ConcreteCommand*.
ConcreteCommand	An application specific command class.

The pattern works by automatically wrapping a *ConcreteCommand* created by the application with a *ConcreteMetaCommand*. The programmer has to follow some programming guidelines to make this process work.

First, the *ConcreteMetaCommand*'s *execute()* method does everything that is common to all commands. During its own execution, it has to call the wrapped *ConcreteCommand*'s *execute()* operation. For the application programmer, it looks like the *ConcreteCommand*'s *execute()* operation would be executed.

Second, if a programmer creates an instance of a *ConcreteCommand*, the system will return an instance of the *ConcreteMetaCommand* (as it is determined by the *ConcreteCommandFactory* in use) that contains the required *ConcreteCommand*. Because such semantics are not possible with constructors, a special static *create()* operation is used. Every *ConcreteCommand* class can have one or more *create()* operations with a distinct signature. It is important to realize that the *ConcreteCommand.create()* operations do not return an instance of *ConcreteCommand*, but an instance of *Command*. This is because the *create()* operations wrap the required *ConcreteCommand* with the appropriate *ConcreteMetaCommand*. To achieve this, it calls the *wrap()* operation on the created *ConcreteCommand*. *wrap()* calls the factory's *wrapCommand()* method, which in turn creates a *ConcreteMetaCommand* and wraps it around the *ConcreteCommand* (details follow below).

In general, the *create()* operation of a *ConcreteCommand* will usually look something like this (Java example):

```
class ConcreteCommand extends Command {
    public static Command create( String anArgument ) {
      ConcreteCommand c = new ConcreteCommand();
        // process arguments with c
       c.anArgument = anArgument;
        // call wrap to wrap the created ConcreteCommand
       return c.wrap();
    }
    public void execute() ....
}
```

INTERACTIONS

Setting up the factory

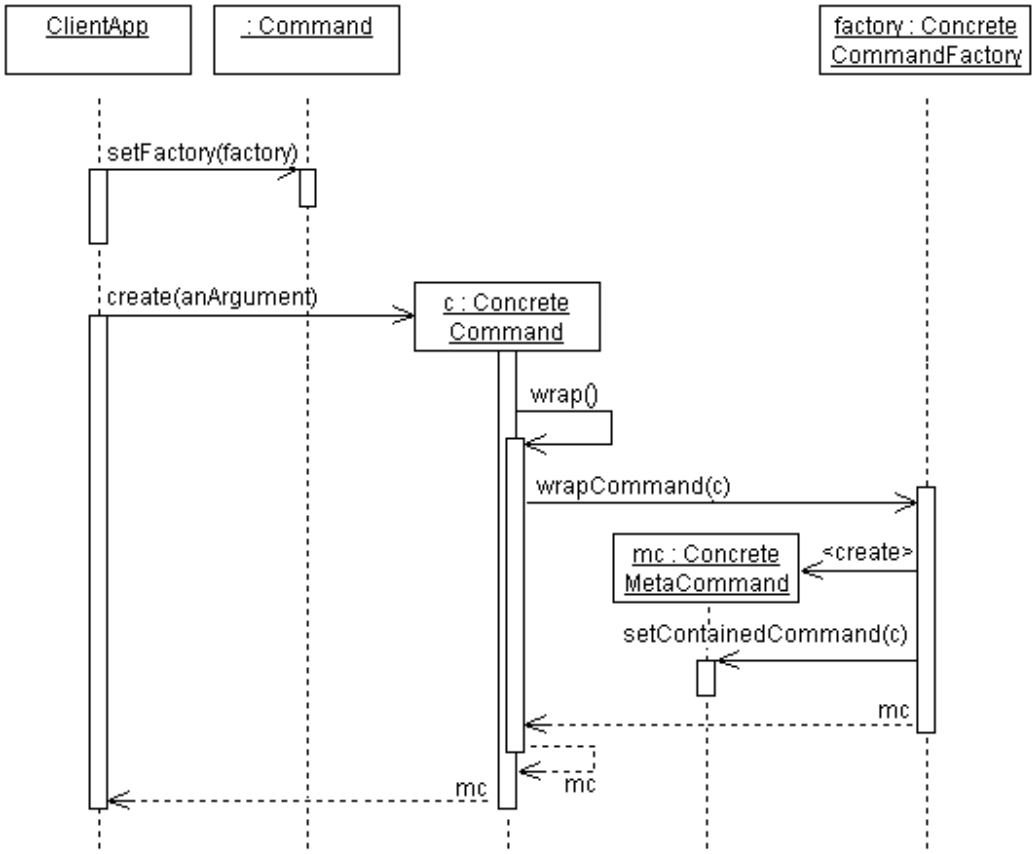
At the beginning of a program, the factory that should be used by the application must be set by the application programmer. The abstract class *Command* contains a static reference to an instance of a *CommandFactory* object. To set this reference, the programmer calls the static operation *setFactory()* with an instance of a *ConcreteCommandFactory* as the only parameter.

Creating a ConcreteCommand object

The sequence diagram below shows what's going on when a *ConcreteCommand* object is created.

The client application calls a static *create()* operation on the *ConcreteCommand* class, of which an object is required. This operation then creates an instance of its own class, processes all its aguments, and then calls *wrap()* on the newly created object.

The *wrap()* operation calls the *wrapCommand()* operation on the factory that has been set before with the *ConcreteCommand* as a parameter. The factory determines which *ConcreteMetaCommand* should be used, creates an instance of it, and then sets the passed *ConcreteCommand* to be the contained command of the newly created *ConcreteMetaCommand*. This metacommand is then returned to the client application.



Executing the command

Just as in the conventional Command pattern, a command is executed by invoking its *execute()* operation. The difference here is, that the invocation reaches the *ConcreteMetaCommand*'s *execute()*, which in turn calls the *ConcreteCommand*'s *execute()*. The *ConcreteMetaCommand* is free to add whatever code it wants, examples follow below.

CONSEQUENCES

Using this pattern has the following advantages:

- The policy of which *ConcreteMetaCommand* should be used for a certain *ConcreteCommand* object is encapsulated in the *ConcreteCommandFactory*. By setting another factory, the policy can be changed, even at runtime.
- The pattern is totally transparent to applications that use the ordinary Command pattern. After all, the most important advantage of the Command pattern is that the

different kinds of *ConcreteCommand* objects are treated as if they were just *Command*s, except upon instance creation. This is still true if Metacommand is used instead.

One the other hand, the pattern also has the following potential drawbacks:

- The *ConcreteCommand* objects cannot be created with the help of a constructor. Instead, a static *create()* operation must be used.
- The pattern might impose some slight overhead in the execution of a *ConcreteCommand*, because an additional layer of indirection is added (*ConcreteCommand*'s *execute()* is called through *ConcreteMetaCommand*'s *execute()*).However, if no *ConcreteCommandFactory* is set in the *Command* class, the *wrap()* operation can directly return itself (*this*). Then, the *ConcreteCommand* eventually gets executed directly, and the pattern adds absolutely no overhead to command execution.
- The pattern relies on the wrapping *MetaCommand* object to call the *execute()* operation of the contained *Command*. However, it cannot be easily enforced that this is actually done by the *MetaCommand*. In some cases this is intended (e.g. not executing a *Command* if when the *MetaCommand* determines that the permissions required to execute the *Command* are not available). On the other hand, it is possible to actually modify the behaviour of *Command* classes, and not just to add behaviour. This might be a problem in certified / well tested / safety relevant systems.

EXAMPLES AND SAMPLE CODE

The simplest case

Let's begin with a very simple implementation of the *Command* class. Java is used for the examples.

```
public abstract class Command {
    protected static CommandFactory factory;

    public static void setFactory(CommandFactory cf) {
        factory = cf;
    }

    public Command wrap() {
        if ( factory == null ) return this;
        return factory.wrapCommand( this );
    }

    public abstract void execute();
}
```

The Command class above has the important feature, that whenever no factory is set, the *wrap()* operation returns *this*, i.e. it returns the object on which it was called. So whenever no factory is set, the system behaves just like the standard Command pattern. The check whether the *factory* is *null* is implemented very efficiently in Java, and imposes nearly no performance overhead.

Logging

The above example might be the initial implementation for a software system. Later, when the system is deployed, it might become necessary to log all executed commands. This can be achieved by the following three step process:

First, a corresponding *MetaCommand* class is created. It is called *LogMetaCommand* in the example and could look something like the following:

```
public class LogMetaCommand extends MetaCommand {
    public void execute() {
        System.out.println( (new Date()).toString() +
             " executing : "+containedCommand );
        containedCommand.execute();
    }
}
```

The second step is to create a factory that wraps the *ConcreteCommand*s with a *LogMetaCommand*. This is also very simple and straightforward:

```
public class LogCommandFactory extends CommandFactory {
    public Command wrapCommand(Command c) {
        LogMetaCommand lcmd = new LogMetaCommand();
        lcmd.setContainedCommand( c );
        return lcmd;
    }
}
```

The third step is the simplest of the three: Setting the factory object in the *Command* class. The following program does just that:

```
public class Test {
    public Test() {
        Command.setFactory( new LogCommandFactory() );
        Command t = TestCommand.create( "Hallo" );
        t.execute();
    }
    public static void main(String[] args) {
        Test test = new Test();
    }
}
```

The above piece of code has the consequence, that every command execution in the system gets logged. This is true for command classes are already used in the system, as well as for those, that are introduced later.

Permissions

Often, it becomes necessary to retrofit a system with security features, i.e. that certain commands may only be executed by certain users (or groups of users). Once again, Metacommand provides a simple solution for this problem by using *MetaCommand*s that check the permissions before executing the contained command. The check itself could be delegated to another class:

```
public class PermissionMetaCommand extends MetaCommand {
    public void execute() {
        if ( PermissionManager.instance().allows( containedCommand )
            containedCommand.execute();
        else showMessage( „Required permissions not available!“ );
    }
}
```

Queueing

In some applications, it might become necessary to queue some command for asynchronoues execution. The Metacommand pattern can be used to implement such a queueing facility. Two building blocks are necessary here. A *QueueingMetaCommand* and a *QueueFactory*.

Upon execution, the *QueueingMetaCommand* puts the *containedCommand* into a queue which is then processed asynchronously by another thread. The *QueueFactory* determines whether the command should be queued or not. If it should not be queued, then the factory does nothing and returns the command itself. If, on the other hand, the command shoud be queued, it wraps the command with a *QueueingMetaCommand*.

Note: If the time between enqueueing the commands and their execution is long, and if the factory is replaced during this timespan, then the already enqueued (and therefore wrapped) command objects will not change their behaviour to reflect the chances imposed by the new factory.

ANOTHER SOLUTION – AND WHY IT DOES NOT WORK

Another solution that comes to mind immediately is simply to provide the *Command* classes with a *before()* and *after()* method. It would be the responsibility of the *ConcreteCommand* classes *execute()* methods to call them. This does not work because:

- You cannot give different groups of *ConcreteCommand* classes different behaviour – to do this, wou yould have to insert an additional layer of abstract command classes. But because you do not know the grouping of the *ConcreteCommand*s in from the beginning, you would have to change the class hierarchy constantly, every time, you want to change the grouping.
- You cannot change anything at runtime, except by using state dependent case statements in the command classes *before()* and *after()* methods.
- Permissions and queueing cannot be implemented cleanly this way because the result of the *before()* method does not influence the behaviour of the main *execute()* method. This could be achieved by using exceptions somehow, but this is not a very practical solution. (Example, Queueing: The before method could place the itself (*this*) into a queue, then set a flag, which is tested by the *execute()* method after its *before()* call. If set, the method returns. When *execute()* is once again called, this time by the thread that processes the queue, *before()* must not be called again, instead the core of the *execute()* operation must be run. This can also be achieved by using a couple of flags. But all in all, this is not a very elegant solution).
- Composability (as described in the next section) is not possible.

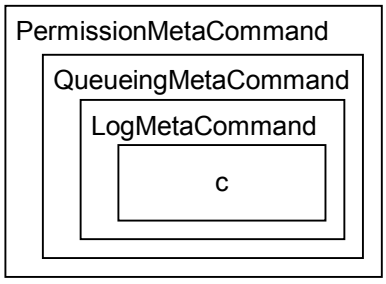
VARIANTS AND IMPROVEMENTS

Multiple Layers

It is possible to compose multiple layers of *MetaCommand*s. If a system contains a *LogMetaCommand* class, a *QueueingMetaCommand* class and a *PermissionMetaCommand* class, then it is easy to build systems that have all features just by creating another factory:

```
public class LQPCommandFactory extends CommandFactory {
    public Command wrapCommand(Command c) {
        LogMetaCommand lcmd = new LogMetaCommand();
        lcmd.setContainedCommand( c );
        QueueingMetaCommand qcmd = new QueueingMetaCommand();
        qcmd.setContainedCommand( lcmd );
        PermissionMetaCommand pcmd = new PermissionMetaCommand();
        pcmd.setContainedCommand( qcmd );
        return pcmd;
    }
}
```

The result of the work of this factory is the following containment structure:



Upon execution, the commands are executed "outside-in". First, the PermissionMetaCommand checks whether the command is allowed to be executed. If so, the command is put in a queue. Upon execution, a log message is created and then, finally, the command itself gets executed. Note, that the order wrap order of the commands is significant: If, for example, permission checking is contained inside of logging, the command will be logged although perhaps it will not be executed. The same is true for queueing and permissions. There is no use of checking a permission after it has been inserted into the queue (because they will be executed asychronously).

Factory as Decorator

If many different combinations of *MetaCommand*s are necessary in an application, it is possible to design the factories according to the Decorator (aka Wrapper) pattern. This makes it possible to wrap several factories around each other, thereby achieving composability in the area of command creation.

Controlling the Factory

If there is no general common policy about which *MetaCommand* the factory should use, then it might make sense to add additional *create...(...)* operations on the *Command* classes that have other names or additional parameters. This makes it possible for the creator of a command to determine the behaviour of the factory at runtime, because the

factory can use the supplied parameters to determine whether a *MetaCommand* will be set or not, and to give the MetaCommand additional parameters.

RELATED PATTERNS

The Metacommand pattern of course has a close relationship to the Command pattern – it is an extension. The Command pattern is described in the classic Gang of Four book [1]. In addition, Metacommand makes use of the Abstract Factory pattern, which is also described in [1]. The containment structure of the commands (and of the factories, see VARIANTS AND IMPROVEMENTS) can be seen as an implementation of the Decorator pattern [1]. Its behaviour (recursively calling *execute()*) can be explained by with the help of the Chain of Responsibility pattern also described in [1].

Another pattern that extends Command and adresses quite similar concerns is Peter Sommerlad's Command Processor pattern [2]. In the Command Processor pattern, commands are not executed directly. Instead, they are executed by a command processor. This component can be used to implement additional code, for logging, queueing etc. So, instead of wrapping *MetaCommand* objects around each *ConcreteCommand* to implement common behaviour, the Command Processor pattern locates this common code into the command processor component. The decision which of the two patterns should be used, could be guided by the following observation:

- The Command Processor leaves the creation of commands unchanged and requires modification at all locations where commands are executed. So whenever it is not possible (or feasible) to modify the creation of command objects, the Command Processor pattern should be used.
- The MetaCommand leaves the execution of commands unchanged but requires changes in the code at all locations, where commands are created. So whenever it is not possible (or feasible) to modify the execution of commands, the MetaCommand pattern should be used.

Because of the above, the Metacommand pattern as well as the Command Processor pattern can be used in refactorization projects, where it is necessary to introduce unanticipatet features into a software system.

OTHER RELATED WORK

It is also interesting to look at the relationship to aspect-oriented programming (AOP, [3]). AOP supports the definition of cross-cutting concerns of a software system in a single entity, called an *aspect*. Aspects can introduce new methods into (one or more) classes, and can add *before* and *after* code to methods. The following piece of *AspectJ* [4] code adds logging to the command classes:

```
aspect Log {
    crosscut commands():
        public void execute() & com.ourcompany.theproduct.commands.*;
    static advice commands() {
        before {
            System.out.println( (new Date()).toString() +
                                " executing : "+containedCommand );
        }
    }
}
```

The above piece of code defines an aspect called *Log*. It applies to all classes in the *com.outcompany.theproduct.commands* package. In these classes, it advices (i.e. annotates, modifies) the *public void execute()* method. It does so by introducing *before* code, that simply puts out the log message. The aspect is "woven" into the ordinary Java code by an AspectJ-supplied utility called the aspect weaver to produce ordinary Java code.

One could also see the Metacommand pattern as a way to dynamically change the class of the commands, because arbitrary behaviour can be added or removed at any time. This can also be achieved by using meta level programming, if it is available in the language in use. This is where the pattern borrows its name.

KNOWS USES

The pattern was used in the ThoughtPad application [5], a tool create topic maps. Here, the pattern has been applied after the program has been finished (it already used the conventional Command pattern for all user interface actions). As described above, only the creation of the commands had to be changed. The Metacommands have been used to implement permissions into the program.

IBM's PRODIKOS project uses the Metacommand pattern in its user interface architecture. The system will later be integrated with Lotus Workflow, although it is not yet clear at the beginning of the project how this integration will look like. A single metacommand will be used for all command objects, that analyses the concrete command and then triggers the correct process in Lotus Workflow.

ACKNOWLEDGEMENTS

I'd like to thank the participants of MATHEMA's design patterns courses (Feb. 14 – Feb. 21, 2000). They provided many useful comments and improvements to this paper.

REFERENCES

[1] Gamma, Helm, Johnson, Vlissides; *Design Patterns, elements of reusable software design,* Addison-Wesley 1996

[2] Peter Sommerlad, *Command Processor*, in *Pattern Languages of Program Design 2*, Addison-Wesley 1996

[3] Xerox PARC, *AOP homepage* at http://www.parc.xerox.com/csl/projects/aop

[4] Xerox PARC, *AspectJ homepage* at http://aspectj.org

[5] voelterSOFTWARE, *ThoughtPad homepage* at http://www.voelter.de/thoughtpad