Metacommand

(c) 2000 Markus Voelter, voelter@acm.org   Version 1.5, July 13, 2000

INTENT

METACOMMAND is a compound pattern that enhances COMMAND by allowing a client application to enhance or modify the common behavior of the command classes used in the application without modifying the command classes themselves.

CONTEXT AND MOTIVATION

As described by the Gang-of-Four in [GHJV96], you can use the COMMAND pattern to

> *Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.*

However, using COMMAND requires that you know these additional features when you implement the command classes, i.e. at design/implementation time. It is not possible to dynamically add or remove features. For example, you might want to add logging facilities to all command classes in a system if debugging becomes necessary or your customer requires logging for security reasons. Or you might want to add a permissions check before executing a command, executing the command only if the check permits the execution.

The METACOMMAND pattern provides this additional flexibility by allowing you 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, usually one for each user action.

This pattern can be used in two situations: First, the pattern can be used, when a new application is built from scratch. Second, it is also possible to use the pattern to retrofit an existing application with additional functionality.

FORCES

The pattern resolves the following forces:

- The common behavior of your command classes should be flexible, enabling you to change common behavior without modifying the command classes themselves. This should be achieved without requiring big changes to the programming model of the original COMMAND pattern.
- This additional flexibility should ideally come without decreasing performance.
- Changing the common behavior should not require changes through the system. Changes should be localized.
- Retrofitting an existing system with these additional features should be simple. The necessary changes should be systematic to allow automatic conversion (e.g. by a script).

PROBLEM

How do you create a system that allows you to easily modify the common behavior of command objects, even at runtime?

SOLUTION

Wrap the created command object with a *MetaCommand* object upon creation according to the DECORATOR pattern. Use a well-known FACTORY to determine which concrete *MetaCommand* class is used to wrap the *Command* and to execute the wrapping. When a command is executed, the *MetaCommand* is executed instead. It adds its own behavior before in turn executing the contained command.

Structure

Just as in the original COMMAND pattern, the *Command* class is an abstract base class for the specific commands that you use in the application. It has an abstract *execute()* operation that gets called by the command-executing program entity, no matter what the concrete command class is.

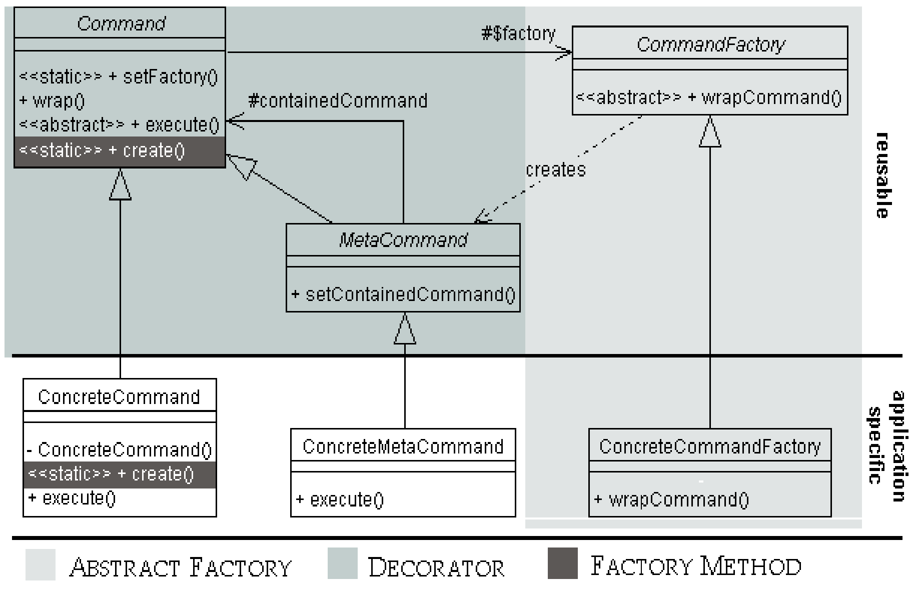

Illustration 1: Class diagram of the METACOMMAND pattern. The classes above the line form the reusable, abstract base of the pattern, whereas the classes below the line have to be created specifically for an application.

In addition, the *Command* class contains a class attribute (Java or C++ *static*) that references a well-known *CommandFactory* object (according to the ABSTRACT FACTORY pattern, [GHJV96]), which is responsible for automatically wrapping a *ConcreteMetaCommand* object around a *ConcreteCommand* upon creation. *MetaCommand* also inherits from *Command*, so that it can be used wherever a *Command* is expected. An application defines several *ConcreteMetaCommand* classes, the *ConcreteCommandFactory* determines, which of these classes will be used to wrap a specific *ConcreteCommand* object. You have to make sure that a *ConcreteMetaCommand* invokes the contained *ConcreteCommand*'s *execute()* operation during its own execution.

Creation of *ConcreteCommand* objects and the wrapping with a *ConcreteMetaCommand* is handled by one or more static *create()* operations (FACTORY METHOD, see [GHJV96]) in the *ConcreteCommand* classes. Please note that these operations formally return *Command*, not the *ConcreteCommand* in which they are declared. This is because the concrete class of the returned object depends on the factory's decision which *ConcreteMetaCommand* is wrapped around the command. (Note: You cannot use a constructor here, because a constructor always returns an instance of the class in which it is declared. A FACTORY METHOD like the *create()* operation is more flexible is this respect. That's why a factory method is also termed a virtual constructor [JC91]). The create operation 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() ....
}
```

The important part of this code snippet is the return statement: After initialization, the *wrap()* operation is called on the new object before it is returned. This creates a *ConcreteMetaCommand* and wraps the *ConcreteCommand* (*c* in the example) with it. The operation returns this newly created *ConcreteMetaCommand*:

```
public abstract class Command {
    protected static CommandFactory factory;
    public Command wrap() {
        Command result = null;
        if ( factory != null ) result = factory.wrapCommand( this );
        return result;
    }
}
```

Participants

Command	Abstract base class for application-specific concrete command classes. It contains the abstract declaration for the *execute()* operation.
MetaCommand	Abstract base class for application specific *MetaCommand* classes. Wraps around (contains) a *ConcreteCommand* object.
CommandFactory	Responsible for setting up the wrapping between the *ConcreteCommand* and the *ConcreteMetaCommand*.
ConcreteMetaCommand	An application-specific *MetaCommand* class. Executes the *ConcreteCommand* during its own execution. It contains code common to all *ConcreteCommands*.
ConcreteCommandFactory	A concrete implementation of *CommandFactory.* Responsible for wrapping an instance of a *ConcreteMetaCommnad* around a *ConcreteCommand*.
ConcreteCommand	An application- specific command class. Defines one or more *create()* operations, which work as described above.

Interactions

Initialization and command creation

The abstract class *Command* contains a static reference to an instance of a *CommandFactory* object which must be set at the beginning of a program by calling the *Command* class's static *setFactory()* operation with an instance of a *ConcreteCommandFactory* as the only parameter.

To create a *ConcreteCommand* object, you have to call the static *create()* operation on the *ConcreteCommand* class, of which you need an instance. As demonstrated above, this operation then creates an instance of its own class, processes all its augments, and then calls *wrap()* on the newly created object.

The *wrap()* operation calls the *wrapCommand()* operation on the well-known factory that you have set before, passing 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 (*mc*) is then returned to the client application. From the client's view, this looks like the following:

```
Command cmd = MyCommand.create();
```

Note that the type of the variable storing the returned object (*cmd*) is *Command*, not *MyCommand* because the concrete type of the returned object depends on the factory.

The following diagram illustrates the previously described processes:

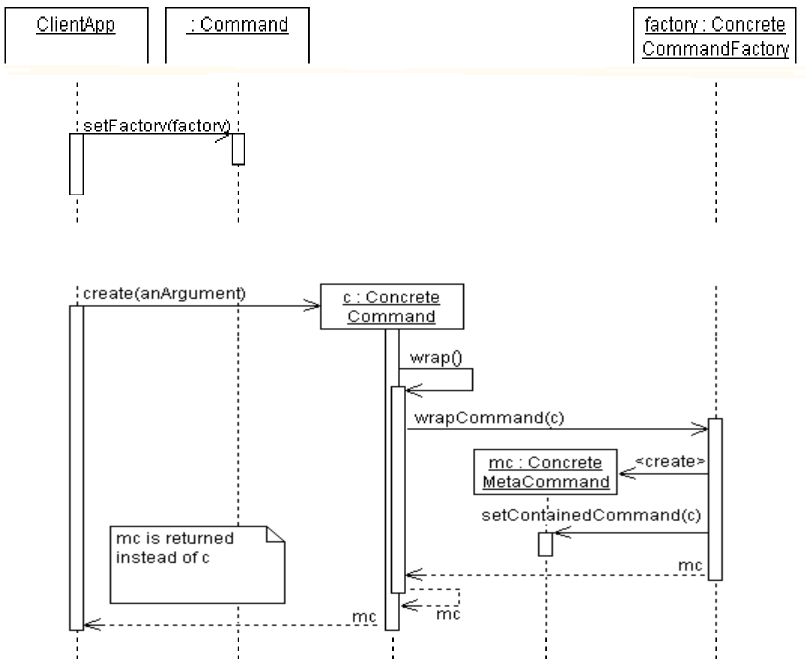

Illustration 2: Setting up the factory and creating a *ConcreteCommand* object.

Executing the command

Just as in the conventional COMMAND pattern, you execute a command by invoking its *execute()* operation. But because the *create()* operation returned the *ConcreteMetaCommand*, the invocation reaches the *ConcreteMetaCommand*'s *execute()*, which in turn calls the *ConcreteCommand*'s *execute()* after adding its own behavior. This is shown in the diagram below:

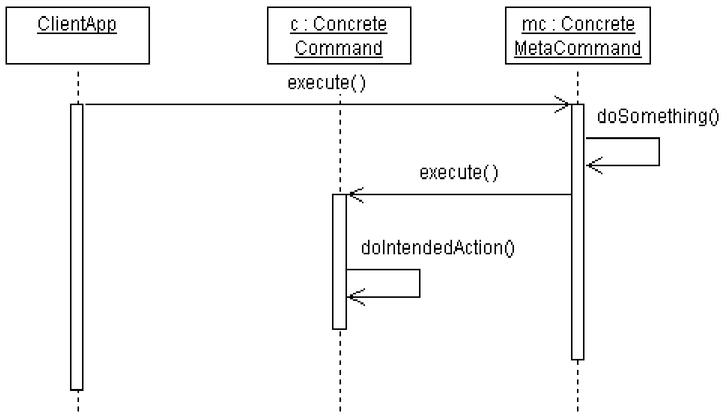

Illustration 3: Executing a command in fact executes the *ConcreteMetaCommand*, which, after executing its own code, calls the *ConcreteCommand*'s execute operation.

CONSEQUENCES

Using the pattern has the following advantages:

- **You can change the common behavior of all commands.** You only need to exchange the *ConcreteCommandFactory*. No change to the *ConcreteCommand* classes is necessary. The change is localized to the factory.
- **The programming model is not very different from the original COMMAND pattern:** You only have to use the *create()* operation instead of the constructor (and provide the *CommandFactories* and *MetaCommand*s).
- **You can easily retrofit an existing application with this increased flexibility**. The handling of created command objects and their execution remains the same, only the creation process changes (see liabilities).

However, the pattern might also has some liabilities or drawbacks:

- **An additional level of indirection is introduced.** (A *ConcreteCommand*'s *execute()* is called through *ConcreteMetaCommand*'s *execute()*). This might impose a slight performance overhead. The same is true for the creation process, when two objects must be created instead of just one.
- **Abstract *Command* class has to be modified:** If the pattern is used in a new application from the beginning, this is not an issue. However, if an existing application is retrofitted with metacommands, you will have to modify the abstract *Command* class (adding the *wrap()* operation) and the existing concrete commands need modification: the *create()* operation has to be added.
- **In C++, new link-time dependencies are introduced:** The *Command* class depends on the factory, and the depends on the *MetaCommands*.

Some consequences are neither good or bad, they are just consequences:

- **The pattern relies on the wrapping *MetaCommand* object to call the *execute()* operation of the contained *Command*.** However, this cannot easily be enforced. In some cases this is intended (e.g. not executing a *Command* in case of a failed permissions check in the *MetaCommand*). On the other hand, it is possible to actually modify the behavior of *Command* classes, and not just to add behavior. This might be a problem in certified / well tested / safety relevant systems.
- **In Java, extending commands (and metacommands) from a base class uses up their single inheritance link**. This is not a problem because usually commands are just a "glue" between two parts of the application and have no further need to extend other classes. Often, concrete commands or metacommands are even anonymous classes. In addition, it is always possible to use interfaces.
- **The client application needs to be changed to modify the behavior of the commands.** This is because another factory has to be set. If this is a problem, you need to use a configurable FACTORY (see below).

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() {
        Command result = null;
        if ( factory != null ) result = factory.wrapCommand( this );
        return result;
    }

    public abstract void execute();
}
```

The Command class above has the important property, 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 (without any performance overhead). 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, you might want to log all executed commands. You can achieve this by executing the following three-step process:

1. A suitable *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();
    }
}
```

2. 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;
    }
}
```

3. Set the factory object in the *Command* class:

```
public class Test {
    public Test() {
        Command.setFactory( new LogCommandFactory() );
        Command t = TestCommand.create();
        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 already used in the system, as well as for those, that are introduced later.

Asynchronous command execution

The pattern can be used to execute normal commands asynchronously. The following piece of code is a *MetaCommand* that creates a *Thread* before executing the contained *Command*.

```
public class AsynchMetaCmd extends MetaCommand implements Runnable {
    public void run() {
        getContainedCommand().execute();
    }
    public AsynchMetaCommand(FAFCommand cCmd) {
        setContainedCommand( cCmd );
    }
    public void execute() {
        Thread t = new Thread(this);
        t.start();
    }
}
```

Permissions

Often, you need 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 reportError(); // show message or throw exception, etc.
    }
}
```

Queuing

In some applications, you will find that it makes sense to queue some commands for asynchronous execution. You can use Metacommand to implement such a queuing facility. Two building blocks are necessary here. A *QueueingMetaCommand* and a *QueueFactory*.

Upon execution, the *QueueingMetaCommand* puts the contained *Command* 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 should be queued, it wraps the command with a *QueueingMetaCommand*.

Note: If the time between enqueuing the commands and their execution is long, and if the factory is replaced during this time span, then the already enqueued (and therefore wrapped) command objects will not change their behavior 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 common behavior in the *before()* and *after()* methods. To do this, you would have to insert an additional layer of abstract command classes which implement common behavior in the *before()* and *after()* methods. But because you usually do not know the grouping of the *ConcreteCommand*s 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.

- You cannot implement permissions and queuing cleanly this way because the result of the *before()* method does not influence the behavior of the main *execute()* method. You could create such a behavior by using exceptions somehow, but this is not a very practical solution. (Example, Queuing: The *before()* method could place 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

Create concrete Commands using the Factory

In the proposed solution, the concrete command classes provide a static *create()* operation that internally contacts the factory to wrap a *MetaCommand* around the command. There is another way how you can do that: let a factory create the concrete command and wrap the *MetaCommand* around it. Instead of calling *create()* on the required *Command* class, creating a command would then be delegated to the factory:

```
  // traditional
Command c = SomeConcreteCommand.create();
  // alternative:
Command c = CommandFactory.instance().createSomeConcreteCommand();
```

The advantage of this approach is that the *Command* class does not need to know the factory, and it does not need the *wrap()* operation. *Create()* operations on all concrete command classes are also not necessary. However, in the factory you need to implement a *create...()* operation for each new concrete command. Alternatively, you can use a pluggable factory like the one described in [OV00].

Multiple Layers

You can compose multiple layers of *MetaCommand*s. If your system contains a *LogMetaCommand* class, a *QueueingMetaCommand* class and a *PermissionMetaCommand* class, then you can easily build systems that have all features just by creating another factory:

```
public class LQPCommandFactory extends CommandFactory {
    public Command wrapCommand(Command c) {
        LogMetaCommand lc = new LogMetaCommand( c );
        QueueingMetaCommand qc = new QueueingMetaCommand( lc );
        PermissionMetaCommand pc = new PermissionMetaCommand( qc );
        return pc;
    }
}
```

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

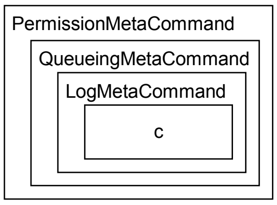

Illustration 4: Containment structure if multiple *MetaCommands* are wrapped.

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 queuing and permissions. There is no use of checking a permission after it has been inserted into the queue (because they will be executed asynchronously).

Multiple wrapped Factories

If you use many different combinations of wrapping *MetaCommand*s in an application, it might be impractical to create a corresponding factory for each combination. Instead, the *CommandFactory* class can be modified to contain a reference to another factory. The factories could recursively call *wrapCommand()* on the referenced factory, thereby adding multiple layers of *MetaCommands* around the original *ConcreteCommand*.

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 behavior 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 pass additional parameters to the *MetaCommand.*

RELATED PATTERNS

METACOMMAND of course has a close relationship to COMMAND – it is an extension. The COMMAND pattern is described in the classic Gang of Four book [GHJV96]. METACOMMAND is actually a compound pattern, i.e. it is a combination of several other patterns. These serve as building blocks for METACOMMAND. The following GoF patterns (see [GHJV96]) are used:

The *CommandFactory* is an instance of the ABSTRACT FACTORY pattern: is responsible for creating and returning instances of the correct *MetaCommand*. The *create()* operation in the command classes is implemented according to FACTORY METHOD. The containment structure of the commands (and of the factories, see VARIANTS AND IMPROVEMENTS) is an implementation of the DECORATOR pattern.

In their article *Compounding Command* [VH99], John Vlissides and Richard Helm look at how the *Command* pattern can be enhanced by combining it with other patterns. They explore the benefits of nesting commands, however, they did not include a factory.

Another pattern that extends Command and addresses quite similar concerns is Peter Sommerlad's COMMAND PROCESSOR pattern [PS96]. 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, queuing etc. So, instead of wrapping *MetaCommand* objects around each *ConcreteCommand* to implement common behavior, the COMMAND PROCESSOR pattern places 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:

- COMMAND PROCESSOR leaves the creation of commands unchanged and requires modifications 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.
- 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 refactoring projects, where it is necessary to introduce unanticipated features into a software system.

It is also interesting to see the relationship to Kevlin Henney's *Executing around Sequences* patterns [KH00]. He shows ways how to execute some code "around" other code in sequential C++ code (e.g. freeing resources at the end of a block that have been allocated at the beginning). METACOMMAND provides a way to achieve the same effect for command objects.

OTHER RELATED WORK

Metalevel programming

You could see the METACOMMAND pattern as a way to dynamically change the class of the commands, because arbitrary behavior can be added or removed at any time. Another way to achieve this is by using meta level programming, if it is available in the language in use. For example, CLOS (see [KRB91]) provides this feature as part of its meta object protocol: Whenever a class (let's say, *Command*) is declared, you can specify the metaclass (let's call it *M*), of which this class (*Command*) is an instance. The metaclass *M* is responsible for invoking operations of instances of the concrete class *Command*. By overriding the *invokeMethod()* operation in the metaclass *M*, you can add behavior to all objects of the class *Command*. In pseudo-Java, this looks like the following:

```
public metaclass M extends MetaClass {
  public void invokeMethod( Object obj, String name, Object[] args ) {
     if ( name.equals( "execute" )
       System.out.println( "executing: "+obj.getClass().getName() );
     super.invokeMethod( obj, name, args );
  }
}

public abstract class Command useMeta M {
  public void execute();
}

public class TestCommand extends Command {
  public void execute() {
         // do something
  }
}

TestCommand c = new TestCommand();
c.anOperation(); // will print log message before executing
```

To see how this is done in CLOS, have a look at the book *The Art of the Metaobject Protocol* [KRB91].

Aspect-oriented programming

It is also interesting to have a look at the relationship to aspect-oriented programming (AOP, [AOP]). 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 existing methods. Modifying existing methods this way is called "advising a method“.

You can use this to create an aspect that advises the *execute()* operations of all *ConcreteCommand* objects, thereby achieving the same effect as the code within the *MetaCommand*s. AOP requires a special tool (called an *aspect weaver* for AspectJ, an AOP implementation for Java, [AJ]). MetaCommand allows similar features to be implemented without the availability of aspects (but requiring more work!).

The example from above would look like the following:

```
aspect Log {
  advise * Command.execute() {
    static before {
      System.out.println( "executing: "+thisClassName );
    }
  }
}
```

KNOWN USES

The pattern was used in the ThoughtPad application [VS], a tool to 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 add permission checks to 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 what this integration will look like. A single metacommand will be used for all command objects to analyse the concrete command and then trigger the correct process in Lotus Workflow.

ACKNOWLEDGEMENTS

Many thanks to my shepherd Wolfgang Keller for giving many useful hints and comments on how to improve the paper. It has improved significantly! I’d also like to thank the participants of MATHEMA’s design patterns courses (Feb. 14 – Feb. 21, 2000) for discussing this paper with me. Last but not least, I'd also like to thank the participants of the EuroPLoP 2000's workshop D "Design and Programming".

REFERENCES

[AJ]	Xerox PARC, *AspectJ homepage* at http://aspectj.org
[AOP]	Xerox PARC, *AOP homepage* at http://www.parc.xerox.com/csl/projects/aop
[GHJV96]	Gamma, Helm, Johnson, Vlissides; *Design Patterns, Elements of Reusable Software Design,* Addison-Wesley 1996
[KH00]	Kevlin Henney, *Executing around Sequences*, Proceedings of EuroPLoP' 2000
[JC91]	Jim Coplien, *Advanced* C++ *Styles and Idioms*, Addison-Wesley 1991
[KRB91]	Kiczales, Rivieres, Bobrow, *The Art of the Metaobject Protocol*, MIT Press 1991
[OV00]	Oliver Vogel, *Generic Factory* in Proceedings of EuroPLoP' 2000
[PS96]	Peter Sommerlad, *Command Processor*, in *Pattern Languages of Program Design* 2, Addison-Wesley 1996
[VH96]	John Vlissides and Richard Helm, *Compounding Command*, in C++ Report April 1999
[VS]	voelterSOFTWARE, *ThoughtPad homepage* at http://www.voelter.de/thoughtpad