

xcommand

extended command pattern
(for Java)

Sven Ehrke
(sven.ehrke@web.de)

10.08.2006

Table of contents

Introduction.....	I
How others do it.....	I
Struts.....	2
Webwork.....	3
Restlet.....	3
Spring MVC.....	4
Servlet API.....	4
Business Service.....	5
Solutions.....	6
Commons-Chain.....	6
xcommand.....	6
Use it.....	7
Context-views.....	8
Contracts.....	12
Command chaining.....	15
Summary.....	15
Ressources.....	15

Introduction

One of the most popular software design patterns is the command pattern. Many web-application frameworks and among those especially the MVC based ones make heavy use of it to implement their action concepts. But web-application frameworks are just one of many areas in which the command pattern is used. Any service oriented software component, library or framework make intensive use of it in one or the other way. Unfortunately each of them has it's own idea on how to implement the command pattern which makes their integration difficult. In this article I will demonstrate how an enhanced variant of the command pattern may alleviate the situation.

First we will briefly look at how some of the popular Java frameworks and libraries implement the command pattern. Having explored the advantages and disadvantage of each technique I'll explain why I came to the conclusion that a new extended command pattern, called xcommand in the rest of this article, has many advantages over existing techniques:

- **Universality:**

the extended command interface can be used in any kind of software the command pattern is needed

- **Independence:**

software products can all use xcommand and still stay independent from each other.

- **Integratability:**

It is much easier for an application to integrate different libraries if they all use the same command pattern.

You will notice that most of the software we look at in this article are web-application based frameworks or libraries. The reason is that I have been working mostly on web-application frameworks during the last couple of years. The extended command pattern however is not restricted to web-applications. In fact one of it's goals was seamless interoperability of web-applications and other libraries and frameworks.

This document describes the main concept behind the extended command pattern and is supposed to stay stable and change not frequently. It is accompanied by second document describing best practices on how to use the extended command pattern and contains frequently asked questions and answers to them. That document will be updated more frequently.

How others do it

In our search for the universal command pattern let us take a look on how popular open source projects implement the command pattern. From doing this we expect the following advantages:

1. **Learning from other's implementation:** we can assume that a lot of technical expertise has been put into these products and thus can be sure that a very good command implementation has been established for the individual product.

2. **Learn real world requirements:** we can also assume that quite some knowledge and experience has been put into these frameworks. Thus if we prove that our universal command pattern is able to fulfill the current requirements of the product we know that it could be used as a replacement of their current command implementation.
3. **Stand the test:** When we think that we found the universal command pattern and want to prove that it really holds what it promises we should strive to replace or integrate the current command pattern implementation of the individual products with ours. While doing this we learn if we still need to improve it and what it means in terms of return on investment: Is it worth it or is it just a nicer design which does not bring any real added value?
4. **Usability (eat our own food):** by trying to use the new command pattern implementation we will learn how usable it is and if we are heading in the right direction.¹

Struts

In Struts' Action class we find the method execute:

```
ActionForward execute(ActionMapping mapping,  
    ActionForm form, HttpServletRequest request,  
    HttpServletResponse response);
```

With this method Struts' Action class implements the command pattern. Since Struts' actions are stateless all data is passed in via method arguments. This has the advantage that not a new Action object has to be created for each request. On the other hand most of the arguments and the return type are Struts specific. To use the mechanism one has to inherit from Struts' Action class which is problematic if your class has the need to inherit from some other, probably more business domain specific class but cannot².

Pro

- stateless

Con

- Struts specific arguments
- no general argument mechanism
- use of inheritance

¹ It took me several redesign iterations until I had satisfying version.

² this has nothing to do with the command pattern itself but with the fact that Java supports only single inheritance. In a language supporting multiple inheritance this would not be a (technical) problem.

Webwork

Webwork's Actions implement either Webwork's (actually xwork's upon which Webwork is built on) interface `Action` or extend the class `ActionSupport`. The interface also defines a method `execute`:

```
String execute() throws Exception
```

`ActionSupport` simply implements `Action` and does not provide anything new in regards to the command pattern. In contrast to Struts Webwork creates a new `Action` object for each request which explains why the method does not take any arguments; the action objects are stateful. There are no Webwork specific parts involved in this interface which is much closer to what we want. Just the `String` return type is a bit questionable (what do we do if we want to return something different?). And for some of us the throws clause might be questionable as well.

Pro

- use of interface instead of inheritance
- no Webwork specific dependencies in command interface

Con

- `String` return type
- Use of throws clause³
- stateful

Restlet

Our next candidate is Restlet, a Java API plus an implementation of REST. Its tutorial shows how we use Restlet's implementation of the command pattern:

```
UniformInterface handler = new
    AbstractHandler()
    {
        public void handle(UniformCall call)
        {
            ...
        }
    };
```

No return type and no throws clause here. But we need to pass in a restlet specific argument and we have to implement `UniformInterface`⁴.

³ Regarding this point contradictory opinions exist as you certainly know. It is the topic of error-handling and whether it is better to use Runtime or checked Exceptions. Since this is not our core topic here I will not discuss it in more detail here. Personally I prefer the usage of Runtime exceptions over checked exceptions.

⁴ `AbstractHandler` is just a convenience class helping to create implementations

UniformInterface, implements the command pattern with the following routine:

```
public void handle(UniformCall call)
```

If we further look at the javadoc of UniformCall we soon realize that it is tied very closely to web-applications. So apart from its promising name it is not really universal at all. On the other hand it seems to be able to transport everything in a single container: an instance of UniformCall.

Pro

- use of interface instead of inheritance
- ability to transport everything in a single container

Con

- Restlet specific dependency in command interface

Spring MVC

Spring MVC's command pattern implementation has the following interface:

```
public interface Controller
{
    ModelAndView handleRequest(
        HttpServletRequest request,
        HttpServletResponse response) throws Exception;
}
```

Pro

- use of interface instead of inheritance
- stateless

Con

- Spring MVC specific dependency in command interface (ModelAndView)

Servlet API

The command pattern implemented by servlets has the following interface (Servlet):

```
public void service(ServletRequest req,
    ServletResponse res)
    throws ServletException, java.io.IOException
```

respectively for HttpServlet:

```
protected void service(HttpServletRequest req,
    HttpServletResponse resp)
    throws ServletException, java.io.IOException
```

Pro

- use of interface instead of inheritance
- stateless

Con

- Servlet specific dependencies

Business Service

The project I currently work on is based on a service oriented architecture (SOA). One main pattern used in the past to access business services is to have a very general interface implementing the following command pattern:

```
public String execute(String input);
```

The input and the return Strings always contain XML according to the Schema the individual services define. The advantage is that with this general interface it is easy to build frameworks around it, which map the web-application world to the business service world. Of course there is also the usual business with marshalling and unmarshalling XML. Since the business services not only are called from web-applications but also from standalone java applications the business service command pattern does not have to know in which context it is running: web-app or standalone app. Thus it is relatively easy to integrate this command pattern implementation with other command pattern implementations.

Pro

- use of interface instead of inheritance
- no specific dependencies in command interface
- relatively general usable interface by using XML Strings as input and output

Con

- difficult to transport data other than XML Strings
- XML needs to be parsed which is overhead in simple cases

Instead of XML another String format could be used. But this would involve parsing as well.

We could continue and investigate other libraries but we will stop with our analysis here since we will find similar implementations of the command pattern in all of them. Let us rather use our time to see if we find a better approach.

Solutions

Commons-Chain

It seems like other people have thought as well about the command pattern and how a common API/implementation can be created and thus came up with Commons-chain. Commons-chain implements the patterns *Command* and *Chain of Responsibility*. If we look at the interface for the command pattern `Command` it looks very familiar:

```
boolean execute(Context context);
```

Unlike a `String` as in `Webwork`, Commons-chain's commands return a `boolean`. This means, when the command is used within a chain it can tell the chain whether execution of the chain should continue or not⁵. And then there is the type of the argument: `Context`. If you look at the documentation of this interface it seems like we came quite close to what we want: it inherits from `java.util.Map`. At the beginning I have been quite enthusiastic about Commons-chain and started to try it out in our own project (some other people got even more enthusiastic about it and migrated Struts to use this approach which will replace the old one I already described). After some experiments with custom context classes for my specific project needs I ended up with complex hierarchies of context classes until I understood that that was the wrong way. Very often I needed the argument type not to be a specific `Context` class but simply a `java.util.Map` since my data was in that format anyway and thus had to create new `Context` classes over and over again what ended up with routines copying over the data forth and back again as adapters to existing code.

xcommand

This was the time when I started to work again on my own design for a command interface. It looked very much like the one from Commons-chain but with a `java.util.Map` as argument and a return type of `Object`. After trying this out in my project for a while it turned out that the return type is not really helpful because the code that called one command very soon called another command which needed the result of the first one as an input argument. So my code very often looked like this:

```
Object r = command1.execute(map);
map.put("inputForCommand2", r);
r = command2.execute(map);
...
```

Therefore I decided that possible return values are put in the `Map` as well. My interface now looked like this:

⁵ Since we are not talking about the chain of responsibility pattern at the moment we will not discuss if this is a good or bad idea. Let us just note that the command API is contaminated with the needs of the chain API even in those situations where a chain is not used at all and readers of the code (maybe unaware of chains since they do not need it) might wonder how they should handle the return value.

```
public interface IXCommand
{
    public void execute(java.util.Map aContext);
}
```

and my usage code became much cleaner:

```
command1.execute(map);
command2.execute(map);
```

In my current project we use this approach for quite a while now and it works very well. The web-app world (Struts in our project) and the business service world are able to use exactly the same `java.util.Map` as an execution context. So apart from `IXCommand` there is neither a coupling between the different worlds nor command specific classes or interfaces.

From the command pattern point of view that's it. Our final implementation of the command pattern:

```
public interface IXCommand
{
    public void execute(java.util.Map aContext);
}
```

fulfills all the requirements we had for a universal command pattern.

To use it in practice however some helping constructs around it and best practices are helpful. These will be explored in the next sections.

Use it

To use `IXCommand` there are some points we should think about:

1. The general put commands and the key Strings like "inputForCommand2" are not really good, even if we would use constants.
2. With this highly dynamic interface we lose the advantages of static typing. Therefore we need to find a way get those advantages back.

Addressing the second point is not easy. It is completely dependent on how far one wants to go. We will come back to this point later in the section *Contracts*.

Regarding the first point the temptation is high to simply create a class `StrutsContext` implementing `java.util.Map` which provides getters and setters for the Struts specific properties. `StrutsContext` then could be passed into `execute()`. Next we would probably do the same for our businessservices; let's call it `BusinessServiceContext`. The problem emerges at the borderline: Somehow part of the content of `StrutsContext` like the data of the inputform needs to be transferred to our `BusinessServiceContext` to be able to execute the business service. This means that a lot of ugly copy code needs to be written which clutters up our

real code. The same approach also leads very soon to complex hierarchies and weird situations which again need to be addressed with more weird code.

After having made this experience for a while it became clear that there is a need to find a better way.

Context-views

The idea of context-views is that we always work with exactly the same context object but look at it from different angles (views). It is based on the pattern *Adapter*, also known as *Wrapper*. How does this look like in practice?

Say we have an object of type `Person` which we would like to put into our context. So far we probably would have done it like this:

```
ctx.put("person", person);
cmd.execute(ctx);
...
```

And then later when we need the person object again we use the following code construct:

```
Person person = (Person) ctx.get("person");
```

The first ugly thing is the use of the keys. If we do not want to have them as String literals in the code we would create constants for them and thus would need a place to put them. A new class called `DomainContextConstants` could be a good place and then our code would look as follows:

```
ctx.put(DomainContextConstants.PERSON, person);
...
Person person = (Person)
    ctx.get(DomainContextConstants.PERSON);
...
```

which is better but still leaves us with the ugly typecasts. Views will help us. The following context-view class is able to handle `Person` and `Account` objects:

```
public class DomainContextView
{
    public DomainContextView(Map aContext)
    {
        context = aContext;
    }
    public Person getPerson()
    {
        return (Person)context.get(
            DomainContextConstants.KEY_PERSON);
    }
    public void setPerson(Person aPerson)
    {
        context.put(DomainContextConstants.KEY_PERSON,
            aPerson);
    }
    public Account getAccount()
    {
        return (Account)context.get(
```

```

        DomainContextConstants.KEY_ACCOUNT);
    }
    public void setAccount(Account aAccount)
    {
        context.put(DomainContextConstants.KEY_ACCOUNT,
            aAccount);
    }
    ...
    private Map context;
}

```

Now getting and setting context properties is the responsibility of the view and client code becomes much cleaner:

```

DomainContextView dcv = new DomainContextView(ctx);
dcv.setPerson(person);
dcv.setAccount(account);
...
Person p = dcv.getPerson();
...

```

In performance critical situations it might be better to use a static version of the context view because then it is not necessary to create a new context view object in each situation one is needed.

```

public class StaticDomainContextView
{
    public static Person getPerson(Map aContext)
    {
        return (Person)aContext.get(
            DomainContextConstants.PERSON);
    }
    public static void setPerson(Map aContext,
        Person aPerson)
    {
        aContext.put(DomainContextConstants.PERSON,
            aPerson);
    }
    public static Account getAccount(Map aContext)
    {
        return (Account)
            aContext.get(DomainContextConstants.ACCOUNT);
    }

    public static void setAccount(Map aContext,
        Account aAccount)
    {
        aContext.put(DomainContextConstants.ACCOUNT,
            aAccount);
    }
}

```

The client code then looks as follows:

```

StaticDomainContextView.setPerson(ctx, person);
StaticDomainContextView.setAccount(ctx, account);
...
Person p = StaticDomainContextView.getPerson(ctx);
...

```

When a context view is likely to be used more than once in a request-response cycle it is worth to consider using the static version. Although it is not a problem in most cases you should be aware that it is not possible to redefine the static methods in subclasses.

For non performance critical situations the non static version certainly is more convenient to use. But even then you have several possibilities to prevent the creation of thousands of new Context-View objects over and over again. You have to distinguish between stateless Context-Views and stateful Context-Views. The stateless ones always take the context as argument in their routine signatures. Stateful ones don't since they contain the context as state.

Using stateless Context-Views one possibility is to use them as singletons in the application and e.g. injected into the required places by dependency injection. Using stateful Context-Views you can create such objects once at the place where they are first used and then set on the context and thus can be retrieved from it when needed.

For the rest of this article the following context views use the non-static approach.

Servlet context view

In the same way `DomainContextView` was done we can build a `ServletContextView`:

```
public class ServletContextView
{
    public ServletContextView(Map aContext)
    {
        context = aContext;
    }
    public HttpServletRequest getHttpRequest()
    {
        return (HttpServletRequest) context.get(
            ServletContextConstants.
                KEY_HTTP_SERVLET_REQUEST);
    }
    public void setHttpRequest(
        HttpServletRequest aRequest)
    {
        context.put(ServletContextConstants.
            KEY_HTTP_SERVLET_REQUEST, aRequest);
    }
    public HttpServletResponse getHttpResponse()
    {
        return (HttpServletResponse) context.get(
            ServletContextConstants.
                KEY_HTTP_SERVLET_RESPONSE);
    }
    public void setHttpResponse(
        HttpServletResponse aResponse)
    {
        context.put(StrutsContextConstants.
            KEY_HTTP_SERVLET_RESPONSE, aResponse);
    }
    private Map context;
}
```

Struts context view

And here is our context view for Struts:

```
public class StrutsContextView
{
    public StrutsContextView(Map aContext)
    {
        context = aContext;
    }
    public Action getAction()
    {
        return (Action) context.get(
            StrutsContextConstants.KEY_ACTION);
    }
    public void setAction(Action aAction)
    {
        context.put(
            StrutsContextConstants.KEY_ACTION, aAction);
    }
    public ActionMapping getActionMapping()
    {
        return (ActionMapping) context.get(
```

```

        StrutsContextConstants.KEY_ACTION_MAPPING);
    }
    public void setActionMapping(
        ActionMapping aActionMapping)
    {
        context.put(
            StrutsContextConstants.KEY_ACTION_MAPPING,
            aActionMapping);
    }
    public ActionForm getActionForm()
    {
        return (ActionForm)context.get(
            StrutsContextConstants.KEY_ACTION_FORM);
    }
    public void setActionForm(ActionForm aActionForm)
    {
        context.put(
            StrutsContextConstants.KEY_ACTION_FORM,
            aActionForm);
    }
    public ActionErrors getActionErrors()
    {
        return (ActionErrors)context.get(
            StrutsContextConstants.KEY_ACTION_ERRORS);
    }

    public void setActionErrors(
        ActionErrors aActionErrors)
    {
        context.put(
            StrutsContextConstants.KEY_ACTION_ERRORS,
            aActionErrors);
    }
}
...

```

Thus the very same context which was initialized before with the `ServletContextView` can now be initialized by the `StrutsContextView` and the both of them do not get into each other's way. At some later point in the request-response life-cycle a `BusinessService-ContextView` may initialize specific properties of the business service call. And while using that view on the same context object the business-service library can be kept completely decoupled from the Servlet API and Struts objects living in that very context.

A big advantage of the context view concept is that whenever one specific implementation does not fit your requirements (like shown with the static version) it is always possible to create your own. As long as you use the same constants it is even guaranteed that the different context views work together seamlessly.

Contracts

By using a `java.util.Map` as `executionContext` we have gained a lot of flexibility. On the other side we also lost the advantages of static typing. What can we do to gain back these advantages but keep the flexibility? The answer is contracts.

Based on the ideas of Design by Contract each command should be able to require that certain preconditions are met on entry and it also might want to declare certain guarantees which are met after execution has been performed. Let

us look at an example. Say we have a command that in order to operate successfully needs to read a parameter from a HTTP request:

```
public class HelloWorldCommand implements IXCommand
{
    public void execute(Map aContext)
    {
        ServletContextView scv =
            new ServletContextView(aContext);
        HttpServletRequest req =
            scv.getHttpServletRequest();
        String firstName = req.getParameter("firstName");
        String greeting = "Hello " + firstName;
        ResultContextView rcv =
            new ResultContextView(aContext);
        rcv.setResult(greeting);
    }
}
```

As you can see it takes the request parameter named *firstName* and „returns“ a greeting message containing this *firstName*. Most likely another command will take this message to put it into an HTML page using some web view technology like Freemarker, Velocity, JSPs or whatever.

So in order to operate successfully the command must insist on being passed a `HttpServletRequest` object. To check this precondition a precondition command can be used:

```
public class HelloWorldPreconditionCommand
    implements ICommand
{
    public void execute(Map aContext)
    {
        try
        {
            ServletContextView scv =
                new ServletContextView(aContext);
            HttpServletRequest req =
                scv.getHttpServletRequest();
        }
        catch (Exception e)
        {
            throw RuntimeException(
                "wrong type passed", e);
        }
        if (req == null)
        {
            throw RuntimeException(
                "HttpServletRequest missing");
        }
        String firstName =
            req.getParameter("firstname");
        if (firstName == null)
        {
            throw RuntimeException(
                "request parameter 'firstname' missing");
        }
        if (firstName.length == 0)
        {
            throw RuntimeException(
                "request parameter 'firstname'" +
                "may not be empty");
        }
    }
}
```

The first thing which is checked is that the request object, if provided is of the appropriate type. If not an Exception is thrown because of the failing typecast used in the context-view. The next things checked are that the request parameter is provided and that it is not empty. Otherwise an exception is thrown. Throwing an exception is just one possible way to react. Setting some error flag or object in the context would be another possibility. In the scope of this article we will not explore further which method is best and how the precondition command should be invoked before the real command is invoked. But it is important to show that with these contract commands type safety can be gained back again.

In the same manner as precondition commands validation commands can be applied to perform input validation. Just remember not to confuse or mix the both of them. Precondition commands are used to detect bugs in the software. Validation command are used to detect normal validation errors in the usage of the software like input errors.

Command chaining

Commons-chain not only provides commands. As it's name already implies it also supports the pattern *Chain of Responsibility*. The reason for this is that when you work with commands you very soon realize that one command is not sufficient. You need to invoke several of them. In commons-chain a chain executes in turn a list of commands until all are executed or until one of the commands reports a problem.

The reason why this article does not show how to do this with the extended command pattern is because I think such a chain is very limited. It is very important to be able to execute more than one command, but using a simple list and a rather simple control mechanism is probably not sufficient. Very often you have the case that you need to invoke command only if a certain condition holds and otherwise call a different command. Something like this is done in Spring Webflow. But using something like Spring Webflow for the extended command pattern is enough stuff worth to be covered in it's own article.

Very often such a flexible support as it is offered by Spring Webflow is not necessary. For those cases it is much easier to hard code the flow control in a special flow controller class.

Summary

This article introduced the concept of xcommand as a way to extend the command pattern in order to create an implementation which is usable in a very general way and in addition supports the integration of individual software libraries in an independent way. xcommand could be used as a replacement for all the cases the command pattern is used today. Even without a support for xcommand by the individual products themselves it is not difficult to apply it to them.

Ressources

1. xcommand: <http://xcommand.sourceforge.net>
2. Struts: <http://struts.apache.org>
3. Webwork: <http://www.opensymphony.com/webwork>
4. Restlet: <http://www.restlet.org>
5. Spring: <http://www.springframework.org>
6. Commons Chain: <http://jakarta.apache.org/commons/chain>