Simple Multicast LAN-Chat

The Program

We implemented a simple „Multicast LAN-Chat“, which can be used to communicate with other users in the same local area network.
It is a simple chat-program which contains basic chat commands and features. One can change the multicast group to (for example) avoid certain other users, or simply to be in a private „chat-room“ with some colleagues, change their username, their usercolor etc. The messages which can be sent are actually in HTML, so the text-formatting is done for us.

Specifications

The program runs on Java 8 and we used SWING for the GUI.
We did write the program in Java to make it platform independent.

Our Intention and Motivation

So why did we implement a chat program for the lecture?
Chat programs are things most of us use every day to communicate with others, so we thought it would be interesting and fun to implement a simple chat program ourselves.

We thought about patterns, which would fit our needs and came up with many ideas. After discussing the general build-up of our program we decided upon 4 main patterns we wanted to implement, namely: Wrapper Patterns, Command Patterns, Factory Patterns and Decorator Patterns. In the following our experiences of using these patterns are further described and discussed.

Experiences and Patterns

Wrapper Pattern

In short: We used a Wrapper Class to wrap up everything which is communication/socket relevant. Our Wrapper Class is called ChatSocket. For communication we used MulticastSockets.

Regarding MulticastSockets a simple communication build-up needs to take place. The socket needs to be bound to a certain port and a Multicast-Group via an InternedAddress and also the receiving part of a program needs to be started up. To simplify this process we wrapped it up in the constructor of our Wrapper Class ChatSockert.

Furthermore, when sending a message, the message needs to be packed in a DatagramPacket, and receiving messages can be even trickier. So we also wrapped this two aspects with two methods in our Wrapper Class, called send and receive.

Experienced Pros
Everything concerning communication/sockets is nicely wrapped up in one class and using sockets is even easier than with solely MulticastSockets.

Experienced Cons
You need to really take care to not bloat up such Wrapper Classes and end up with a „Swiss Army Knife“ like anti-pattern. Things which might seem relevant for the Wrapper Class at first glance, can disrupt the whole pattern. Also methods or members which might seem convenient to put in these classes can bloat it up unnecessarily.
We can further confirm that this aspects might be tempting, because we experienced it ourselves. When implementing the program, we put the random generation of the user name and color in this Wrapper. Hence straining its purpose.

Command Pattern

Like mentioned above earlier, we have several Chat commands, which are initiated by typing a „ / “ followed by the command name. We implemented the following commands:

name	Changes the user name. If no parameter is given a random name is generated, else the argument string is set as the user name.
color	Changes the user color. If no parameter is given a random color is chosen, else the argument (if valid) is set as the user color.
shout	The argument string is printed in upper-case.
group	The argument (if valid multicast-address) is set as the multicast-group.
me	The argument is printed italic (used to express feelings).
clear	The text field is cleared.
sound	If system sounds are available the program makes noises receiving messages. This command serves as an ON/OFF toggle for the sound.

Our intention was to make it to treat all the commands in the same way and also to make it easy to manage them. So we came up with the idea of implementing a Command Pattern.
We did this by making an interface which has only one method called execute and making a class for every command there is. All commands are treated equally by simply overriding the function execute in all of the command classes for their interface Command.

Experienced Pros
It is in fact quite easy to manage the commands and treat them equally by simply calling execute on their instance. One could also implement a list holding all the executed commands to make some sort of a undo/redo option. Another use would be the creation of macros, in the means of combining commands. We mainly used it concerning convenience of the Command implementation.

Experienced Cons
Overhead can be a huge problem when using this pattern. If the commands are rather short (like in our case) it doesn‘t seem to be really necessary to implement a class for each and every command. Every command could also simply be realized by a method. So there can be really no need for an own class for each command.
Some also say (not in our case) that having a lot of command classes is rather confusing

Factory Pattern

The Factory Pattern was used by us as a Command Factory. In our application the combination of the CommandFactory and Command Pattern build the heart and soul of managing the Commands. Our CommandFactory has only one method, which is called extractCommand. And that‘s really all it does. It matches the given command and string in a switch-case block and returns the correct Command Class instance to the calling method. The calling method then calls execute (like mentioned above) on the returned instance and the command is executed.

Experienced Pros
Having a Factory managing the correct instantiation of classes is quite nice and convenient in terms of implementation. Combined with our Command Interface it makes up the complete command control.

Experienced Cons
The cons we encountered are in fact the same as for the Command Pattern. You need to have a class for every (in our case) command there is. Which was in fact not that big of a deal for us, because these command classes where already existing due to the implementation of the command pattern.

Decorator Pattern

We used Decorators to add HTML text formatting blocks to a message. We implemented a base class called Decorator from which the specific types of Decorators are derived. The constructor of Decorator takes an Object as an argument, and the toString() function of string is overwritten by every specific Decorator class (DBold, DItalic, DColor) to add the decorations to the string. You can also (and we did) combine different decorators.

Experienced Pros
They are really handy in terms of Code Management/Code Clearity. The combinability is also a nice aspect about them.

Experienced Cons
You have to be cautious when testing Object equality.

Summary and Lessons Learned

To sum everything up, we are quite positively surprised by the level of convenience and structure certain Design Patterns (if used wisely) can add to a software project. There where many positive aspects about all of them and for some degree even made the implementation (at least partwise) easier. Even though they have several pitfalls and it is very likely that you slightly modify the patterns without you even noticing. However we would strongly advise more people to look into this kind of stuff for software projects and will very likely use them again ourselves. But one has to take in mind that every pattern has its drawbacks and pitfalls and should really think about the necessity of probable patterns.