Ugly Code

# Ugly Code #3

GREGOR WEGBERG – EMBRACES LANGUAGES

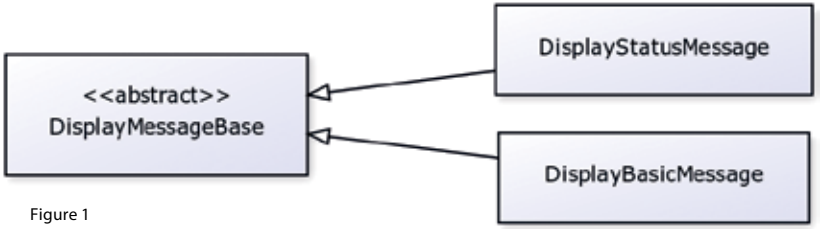


Figure 1

To be honest with you, I wasn't sure how to proceed with my article series "Ugly Code". For some reason—crazy I know—I expected people would read these articles and send me their own opinions and code examples. Well, that never happened—till now! Take a look at Listing 1, a Java class that is part of an exercise here at ETH.

My first reaction to this piece of code was in the realm of "what a piece of sh**". However, it's not necessarily ugly at all, so let's discuss it a bit. First we notice that this is apparently some class responsible for handling two kinds of messages. To make it simple for the user of this class (i.e. some developer), it provides two constructors, one for each kind of message. So, why do I not like this piece of code at all? First, it's one class representing two kinds of messages. To me this already feels somehow wrong, just think about what you'll do if you have to extend one of the messages or even add another? The other thing is the comment of the second constructor. I think the author of it already noticed that she/he is doing something terribly wrong, after all she/he fights against the language!

But this code might even be good code. First off, this code works. It does the job and this is (sadly) in most cases the first and most important goal we have as software developers. Our customers, in most cases, don't care how beautiful the code is or how easy it can be extended. Well, until they have to pay lots of money to extend it because it was never written with future modifications in mind. Further, especially as part of an exercise, it's fine to write such code, as long as it's yours and you wrote it as a student. The moment this code is provided by a teaching assistant I expect it to be well-thought-out and an example for students to learn from!

```java
// Java: Listing 1

// ...

public class DisplayMessage implements Serializable {
        // ...

        /**
         * Constructor to make a Message object
         */
        public DisplayMessage(String message, String username, boolean isMine) {
                super();
                this.message = message;
                this.username = username;
                this.isMine = isMine;
                this.isStatusMessage = false;
                this.date = Utils.getTime();
        }

        /**
         * Constructor to make a status Message object consider the parameters are
         * swaped from default Message constructor, not a good approach but have to
         * go with it.
         */
        public DisplayMessage(boolean status, String message, String username) {
                super();
                this.message = message;
                this.username = username;
                this.isMine = false;
                this.isStatusMessage = status;
                this.date = Utils.getTime();
        }

        // ...
}
```

➔


Figure 2

There are two basic ways I would consider to make it nicer; inheritance [Figure 1] or aggregation ("has a" relation, [Figure 2]). Although inheritance is a very nice feature in object oriented languages, it leads in most cases to strong coupling. Something we should try to avoid. Using aggregation with interfaces (search for "program to an interface") results in loosely coupling and more flexibility (e.g. mocking messages for testing would be much easier). It's even possible to combine both approaches [Figure 3] to minimize code duplication by implementing common class members in an abstract class. I would choose one of these three basic solutions, depending on the requirements, but with a strong inclination to the third solution.

What should we take away from this example? First off, if you have to change something because the language is in your way, think twice before you proceed. Secondly, never forget that software evolves over time, make sure you're ready for changes and can implement them as easily as possible. This means you have to invest a little more time in the first implementation, but likely will save much more time in the long-run. And as a teaching assistant, please try to be an example and show your students what well-thought-out code looks like.

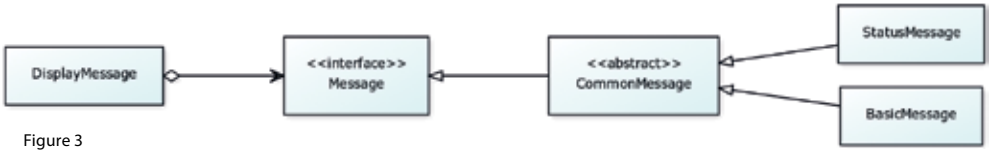


Figure 3

## Bildnachweise

**Cover:** © Manuel Braunschweiler
**S. 7; S. 9:** © VIS
**S. 13:** © VIS
**S. 20-21:** © VIS
**S. 22-33:** © VIS
**S. 34-39:** © VIS
**S. 41-42:** © VIS
**S. 45:** http://www.hdwallpapers.in/
**S. 50-53:** © VIS
**S. 57:** © Barry Barnes – Fotolia.com
**S. 58:** http://fanart.tv/