# 1. Which of the following types of extensions did your solutions contain? Give concrete examples for each!

## Homogeneous and heterogeneous extensions

Our chat product line from assignment 6 contains examples of heterogeneous extensions. As an example, we can have a look at the `ColoredMessages` feature. This feature lives in the `features/ColoredMessages` directory. In this directory, there are two subdirectories, `client` and `messages`. Each of these refers to the respective directory of our base implementation (found in `src`). This means that the code for the `ColoredMessages` feature is separated over multiple files (ie. an heterogeneous extension).

Our chat product line from assignment 7 does not contain any examples of homogeneous extensions. The only extension that we implemented uses pointcuts in only one class. If we would have continued developing the chat-application in an AOP fashion, we would have created homogenous extensions.

## Static and dynamic extensions

Our chat product line from assignment 6 contains examples of static extensions. As an example, we can have a look at the `ColoredMessages` feature, specifically, have a look at `feature/ColoredMessages/messages/Message.java`. This class changes the constructor of the base Message class and overwrites some methods (`getColor()` and `getPlainMessage()`). This is an example of a static extension.

Our chat product line from assignment 7 contains an example of a dynamic extension. Have a look at the `ColoredMessages.aj` file. In this file, we use the `around` operator to dynamically extend the functionality of an existing method. In this case, we extend the `Message.getMessageBody()` method. In the extended method, we modify the return value of the function so that it includes the color of the message and ends with the color for resets.

## Simple and advanced dynamic extensions

The dynamic extension we used in AspectJ as said in the previous section -- dynamically extending the method to get the message body in order to add color codes around the existing message contents -- can be counted as a simple dynamic extension. In FeatureHouse, which we used for our feature-oriented solution, simple dynamic extensions can be mostly found in the use of original(). We have used this in the client to allow different features to add preparation code to be executed after the client is created. The features implement the same method, and the ordering of the preparation code is not very important. This way our authentication feature could ask for the password, and our colored messages feature could ask for the color to use.

We have not used concepts from advanced dynamic extensions in our program, but we think that if we had gone further with AspectJ and reimplemented all our features using it, we would have used advanced dynamic extensions to our advantage in ways we could not do using feature-oriented programming.

## 2. In which situation did you find either AOP or FOP not expressive enough? Which problems arose from these expressiveness limitations? Would it have been possible to avoid these problems by using a combination of AOP and FOP?

When using FOP in our small chat program we did not come across any expressive limitations of FeatureHouse. Refining classes was intuitive and clear. Using AOP however was more complex, but also more expressive. With wildcards general cases could be constructed and easily be reused and modified. Our chat program is however relatively small, therefore the full potential of wildcards could not be experienced. Using wildcards can also cause unexpected behaviour, as any added or modified method can unintentionally fire a wildcard. With refinements this unintentional behaviour is not an issue, as refined classes only apply to a specific class. However for big software projects code cannot be reused easily, as refinements are limited to a single class. When comparing FOP to AOP we would prefer FOP for smaller projects as it is clearer and less sensitive to errors, and AOP for bigger software projects as code can be reused easily. However, when using AOP for bigger software projects wildcards should be applied with caution, as wildcards can be fired unintentionally. A combination of FOP and AOP could combine the best of both. For concentrated features FOP could be used, refining classes. For features which affect the whole product AOP could be used to implement dynamic extensions using wildcards. Aspects can be kept specific by using FOP to refine several specific cases, this would keep the wildcards specific enough, reducing error sensitivity. For concentrated features static extensions could be used to keep the code clear, dynamic extensions can be reserved for diffuse features. Combining FOP and AOP would balance code duplication and clarity, as FOP could be used for heterogeneous extensions and AOP for homogenous extensions, which reduce code duplication, but are more complex.