# Solving the problem of mixing objection with business logic

As we have shown there are many problems involved when business logic contains object creation logic, in order to solve this problem it is necessary to separate the two. This can be done in multiple ways, for instance when going back to our printer example it could be that instead of having the method ##PrintPaper take only a message instead make it take both a message and the paper as part of its arguments. This way we would avoid having to create the Paper as part of printing the paper. The only problem is that this would only remove the problem from that part of the business logic, and move it to the code that uses the method ##PrintPaper. It also changes the overall functionality of the method which was to produce paper with messages on them and instead makes it so it doesn’t produce the paper and only prints on paper given to it. Using the factory design pattern we can avoid both problems while at the same time separate the object creation away from the ##PrintPaper method. We do this by giving the ##Printer class a factory which we will name ##PaperFactory.

The ##PaperFactory only has one task and that is to create Paper objects. That means we have removed the object creation code away from the ##PrintPaper method, all the ##PrintPaper method has to do is simply request a new piece of paper from the ##PaperFactory.

The pseudo code for the ##PaperFactory could look something like this:

Class PaperFactory

Method CreatePaper takes nothing returns Paper

Return new A4()

Endmethod

Endclass

And the pseudo code for the new PrintPaper method of the Printer class would be this

Method PrintPaper takes paperFactory, message returns paper

Paper = paperFactory.CreatePaper()

Paper.Print(message)

Return Paper

endmethod

Thus as we can see the responsibility of creating paper has been removed from the ##Printer class and instead moved to the ##PaperFactory class.

# Abstract Factory design pattern

The abstract factory design pattern is closely tied to Factory design pattern in that an abstract Factory like an abstract class only defines specification of the implementing class, and doesn’t contain any logic at all.

An Abstract Factory is made by making an Abstract class of the factory, going back to Printer example, imagine that the ##PaperFactory instead was an abstract class.

Abstract class PaperFactory

Method CreatePaper takes nothing return Paper

Endclass

Now the Printer class only has to know about a Factory capable of producing Paper, but it will have no information on what kind of paper is produced, an implementation of the ##PaperFactory could now be made for each type of paper that one wishes to produce.

For instance an implementation of the PaperFactory for creating A4 papers could be designed like this:

Class A4PaperFactory implements PaperFactory

Method CreatePaper takes nothing return paper

Return new A4Paper()

Endmetod

Endclass

As we can see the ##A4PaperFactory class that is an implementation of the ##PaperFactory class is capable of producing a special type of paper, as such if this factory was used by the ##Printer class we saw in an earlier example, the ##Printer would be capable of producing A4 papers with messages on them. However were we to want another type of paper, the code for the printer would not need to be changed or copied since we simply implement a new version of the ##PaperFactory class and give that to the ##Paper class.

# Summary

A lot of problems arise in code when mixing business logic with object creation logic, this section has explained why the problems occur and what the reasons behind them are. Furthermore it has shown how to solve these problems by using the factory design pattern.

To summarize some of the problems when mixing business logic with object creation logic

* Increases the complexity in business logic, because of added creation logic
* Removes flexibility of the business logic, forcing it to work only with special classes
* Increase the need for redundancy of business logic to accommodate for new object types
* Makes making changes to legacy code difficult as the code is tied to specific object types