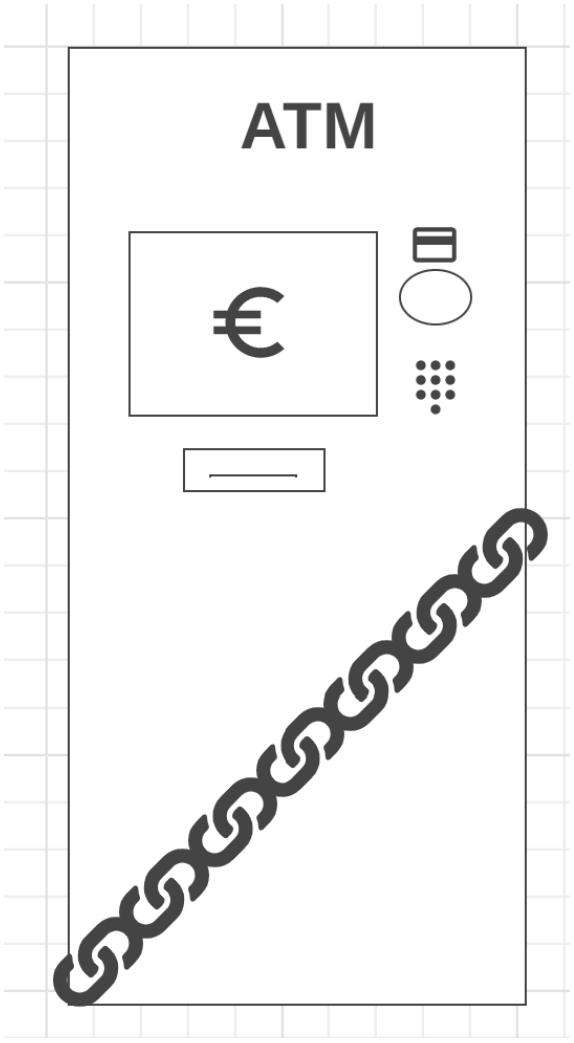
# SWD - Aflevering 1
# Chain-of-Command pattern

05-11-2011

| Name | Student nr. |
|---|---|
| Søren | 201600711 |
| Sigurd | 201804402 |
| Omar | 201804204 |



## Contents

**1 Introduction** **2**

**2 Chain-of-Responsibility pattern** **2**

2.1 Usage . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3

2.2 Pros and Cons . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4

**3 Chain of Responsibility compared to GoF Observer pattern** **4**

**4 Example of Chain of Responsibility** **4**

4.1 Diagram . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4

**5 Implementation** **6**

**6 Conclusion** **9**

**7 References** **9**

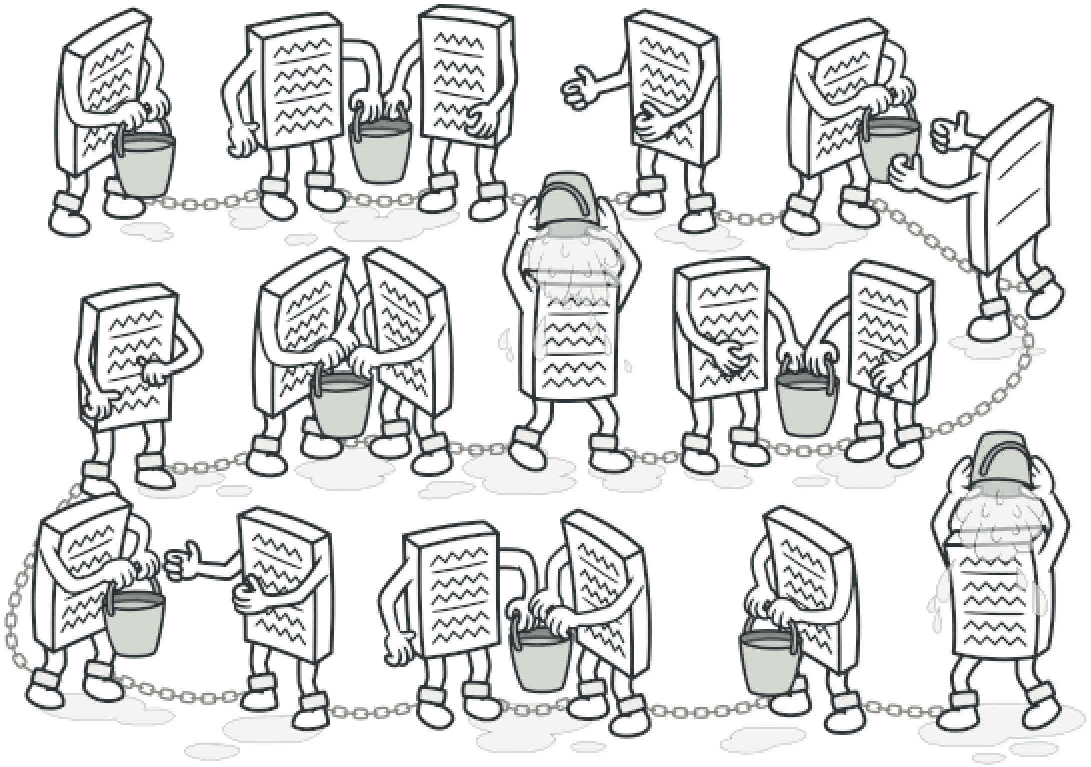


**Figure 1:** A chain of responsiblities passing a request along

## 1 Introduction

In this report, we will examine the benefits and disadvantages of the Chain-of-Responsibility pattern compared it to the GoF Observer pattern. Through this analysis, we will use an ATM machine, that pays out in danish currency as an example. This report is created in conjuction with a presentation video and a demonstration video.

## 2 Chain-of-Responsibility pattern

Chain-of-responsibility is a behavioral pattern, where a chain of handlers gets the chance to act upon a request. A handler will act upon the request, if they are suited to deal with it, otherwise they will pass it along to the next handler in the chain.

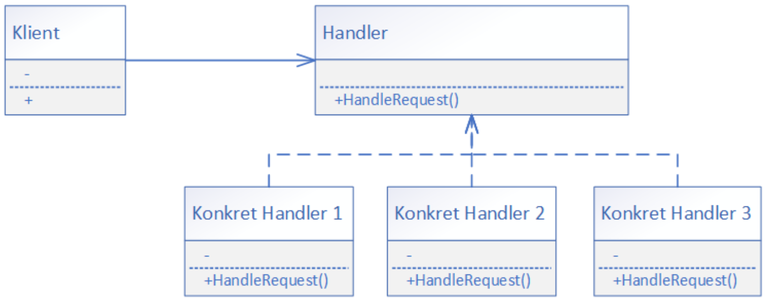


**Figure 2:** General example of an implementation of Chain-of-Responsibility shown in a class diagram. A client could use the handler to handle, as an example, a commandobject. The concrete handler, which receives the command object, also contains the next concrete handler, which this command object is also passed to, if the first object cannot or will not handle.

The chain-of-responsibility pattern consists of an interface implemented by an abstract base class, which is implemented by several concrete handlers. The interface represents the "handle" method to the client, which is method each of the concrete implementations will overwrite, and which handles the problem needed. Implementing the interface with an abstract base class is quite smart, since it implements a setNext and the property holding the next inline handler. This leaves the concrete implementations of the base class. These will just overwrite the "Handle" method from the interface, so it corresponds to the responsibility of the concrete base class.

Implementing the pattern to an application, an instance of the first objekt will be exposed to the client through the HandlerInterface. Depending on the need of the application, a setup must be created, where needed handlers are setup in a chain. When the exposed handler's "Handle" method is called, the handler will act upon the request and do one of two things. If the request is within the handlers responsibility it will deal with the request, and if the request is fully dealt with, it will return. If the request is partially/not within the responsibility of the handler, the handler will deal with it as much as it can and then pass it on to the next handler in line. The next handler will deal with the request in the same manner. The request will thus pass down the chain, untill it has been dealt with. However, it is not a garuantee, that all requests will be dealt with. If the last handler in the chain cannot deal with the request, the request will go unhandled. In this case several options are available depending on the requirements of the system. It is possible to return with the request unhandled, throw an exception or set some sort of error.

The client's responsibility is to pass the relevant input to the relevant concrete handler. This would, oftentimes, be the first link in the chain, which will be referred to as Concrete Handler 1. This would then handle the input, to the best of its ability, and the pass the input, potentially mutated, to the next concrete handler, or even return it to the client
If the input is passed on, mutated or not, Concrete Handler 2 would then be able to handle the output of Concrete Handler 2, and just as its predecessor, it must then pass on this information.
It is not necessarily three handlers, as described above. Chain-of-Responsibility pattern must, in theory, hold 1 to N concrete handlers, as many links as is necessary in the chain.
These chains may handle incredibly complex inputs, and reduce them to simpler and simpler information. This may be as complex as a battle, where the input is the battlefield, equipment, strategies and so forth. The first Concrete Handler, may represent a general, who must then provide handled information to its officers, them to theirs and so on, all throughout the chain of command.
But a Chain-of-Responsibility may also be very simple, such as a fireman's chain, where the first Concrete Handler Fireman is given a bucket of water and hand this bucket on to the next fireman, and so on, until the bucket reaches a fireman standing by the fire, who must then pour the water on the fire.
A third example is if there is a request of access to a closed off area, where the input is the clients request. But Concrete Handler 1 does not have the authority to open the gate, so it sends the request to the next concrete handler, who doesn't have authority either. And so the request will move through the chain, until it reaches concrete handler N that has the authority

As such there are many example and areas, where a Chain-of-Responsibility may be implemented. This implementation is designed with respect to the type of input given, but as such, it is also relevant to keep in mind that it is not always necessary to start at the beginning of the chain at Concrete Handler 1. Which is relevant for performance consideration

## 2.1 Usage

- Use the Chain of Responsibility pattern when your program is expected to process different kinds of requests in various ways, but the exact types of requests and their sequences are unknown beforehand.
- The pattern lets you link several handlers into one chain and, upon receiving a request, "ask" each handler whether it can process it. This way all handlers get a chance to process the request.
- Use the pattern when it's essential to execute several handlers in a particular order.
- Since you can link the handlers in the chain in any order, all requests will get through the chain exactly as you planned.
- Use the CoR pattern when the set of handlers and their order are supposed to change at runtime.
- If you provide setters for a reference field inside the handler classes, you'll be able to insert, remove or reorder handlers dynamically.

- Factory classes can be used to build a chain in a specific order, so the client doesn't have that specific responsibility.

### 2.2 Pros and Cons

Using the Chain of Responsibility pattern, you are able to control the order of which a given request is handled both statically and dynamically during runtime, if you choose to. This can prove to be very practical in cases, like request handling of order systems, security checks or other adaptive behaviors. Since a client can send a request / command to the handler, it helps the client implement the single responsibility principle. Further more, the chain itself strongly adheres to this principle, thanks to every concrete handler only having one responsibility. To handle or nah. This loose coupling also satisfies the open/closed principle and makes it super easy to introdouce new handlers, as well as changing the behavoir of a handler, without breaking the existing code. All made possible by its reliance on interface
It provides a framework for easy implementation of the SOLID principle of Single Responsibility. Every concrete handler may have their own responsibility, and if the task falls outside of their responsibility it can be passed on.
The Chain-of-Responsibility pattern also provides a general and reusable solution for a type of problem, reducing the amount of different types of solutions a programmer has to know. Each concrete handler is easier to understand than a massive loop.
The biggest consequence of the Chain of Responsibility pattern is the risk of unhandled requests / commands. If the developer is not careful, and no handler is setup to handle a given request / command it could unintentionally be lost. This is something that is important to be aware of, when implementing.
The Chain-of-Responsibility may not be the best solution for a program where performance is a high priority.
Also, debugging may prove to be a chore, and may even cause a cycle call.
The order of operations in a basic Chain-of-Responsibility is set beforehand and cannot be changed dynamically.

## 3 Chain of Responsibility compared to GoF Observer pattern

Where a Chain of Responsibility takes its input from a client. On the surface this an observer pattern does the same, with a subject serving as a client-standin. Both client and subject informs a concrete implementation, and gives some form of input.
The observer would then notify ALL listeners, who may, or may not, handle this information in a vast array of different ways. A Chain of Responsibility uses a chain, and, as per the usual intention, each concrete implementation of a chain link, must direct information, mutated or otherwise, to the next link in the chain.

## 4 Example of Chain of Responsibility

An ATM has a given amout of money it wishes to pay out. In order adhere to the ...

This ATM is designed for withdrawels of DKr. It receives an input in the form of a value to be withdrawn. When the ATM has received this input, it is passed to Payout, which uses the Chain of Responsibility pattern, to output the money in full, with the fewest amount of bills and coins. This is why it must return the highest available bills and coins first. In this example it starts with a 100 DKr bill, then a 50 DKr bill, then 20 DKr coin, and so on, until there are no more money to be withdrawn.

### 4.1 Diagram

This diagram shows the relationship between the client as an ATM, the handler as the IPayout and an abstract class, which is necessary for implementing the nextPayout function

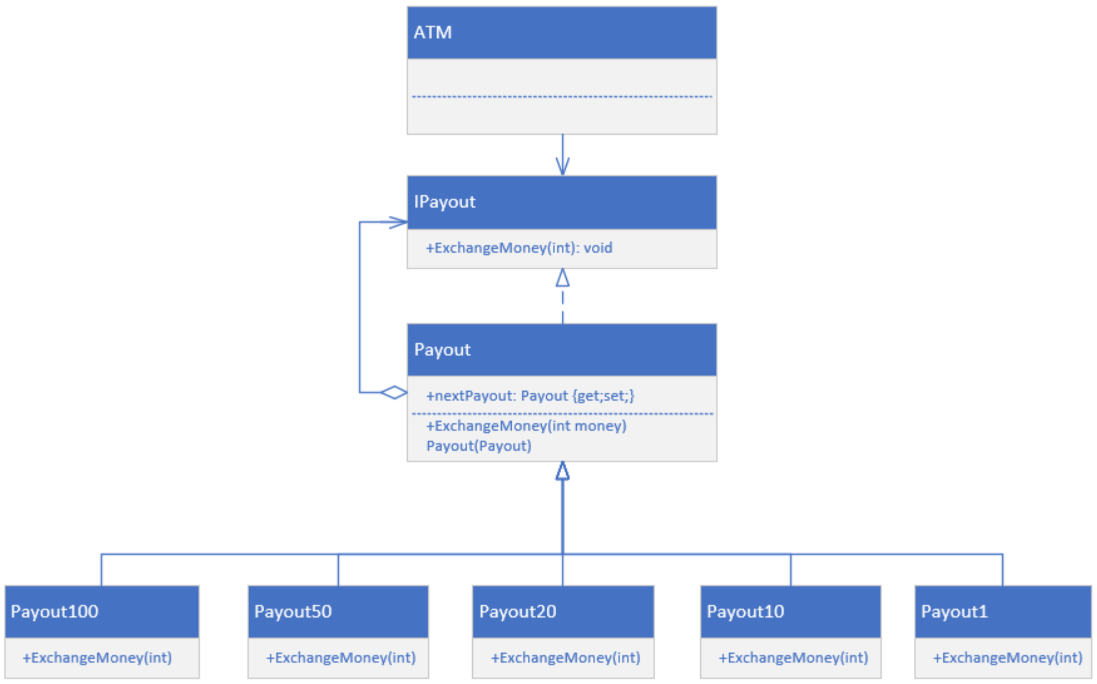


**Figure 3:** The Class Diagram of our specific implementation of an ATM

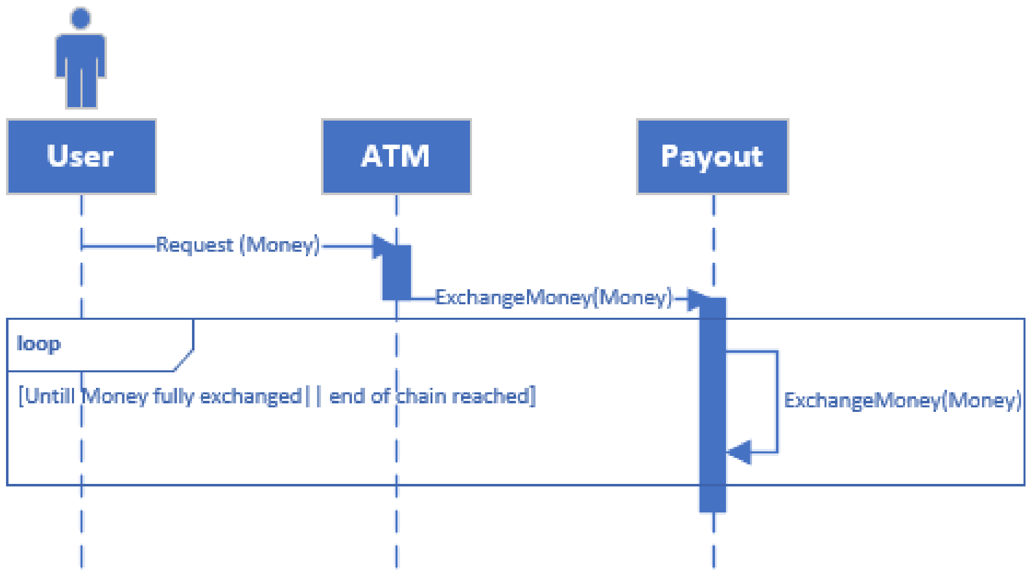
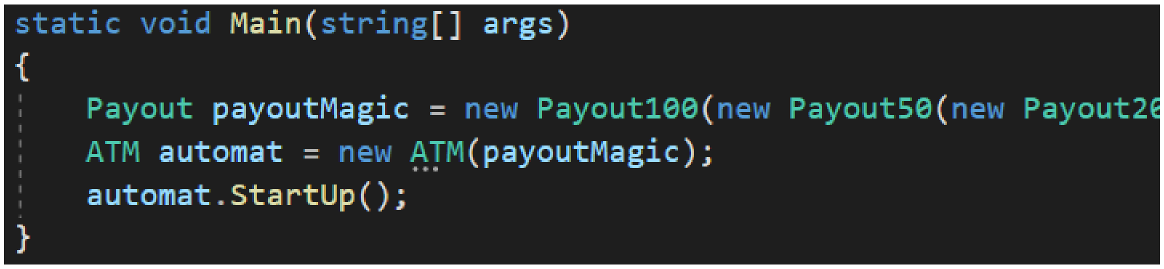


**Figure 4:** The sequence diagram of our speicific implementation of an ATM

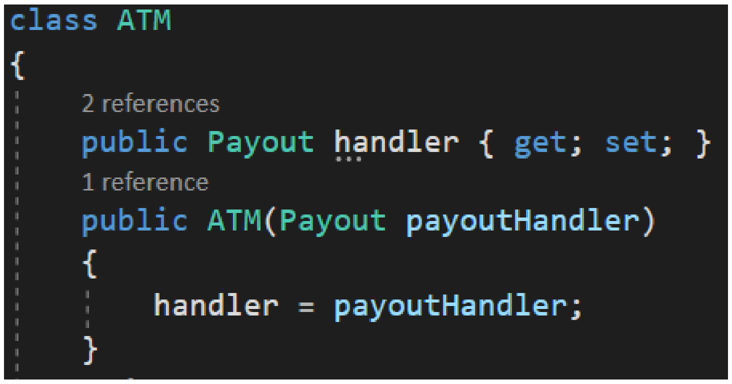
## 5 Implementation



```
static void Main(string[] args)
{
    Payout payoutMagic = new Payout100(new Payout50(new Payout20
    ATM automat = new ATM(payoutMagic);
    automat.StartUp();
}
```

**Figure 5:** The Main implementationCreates the two necessary classes and starts the ATM

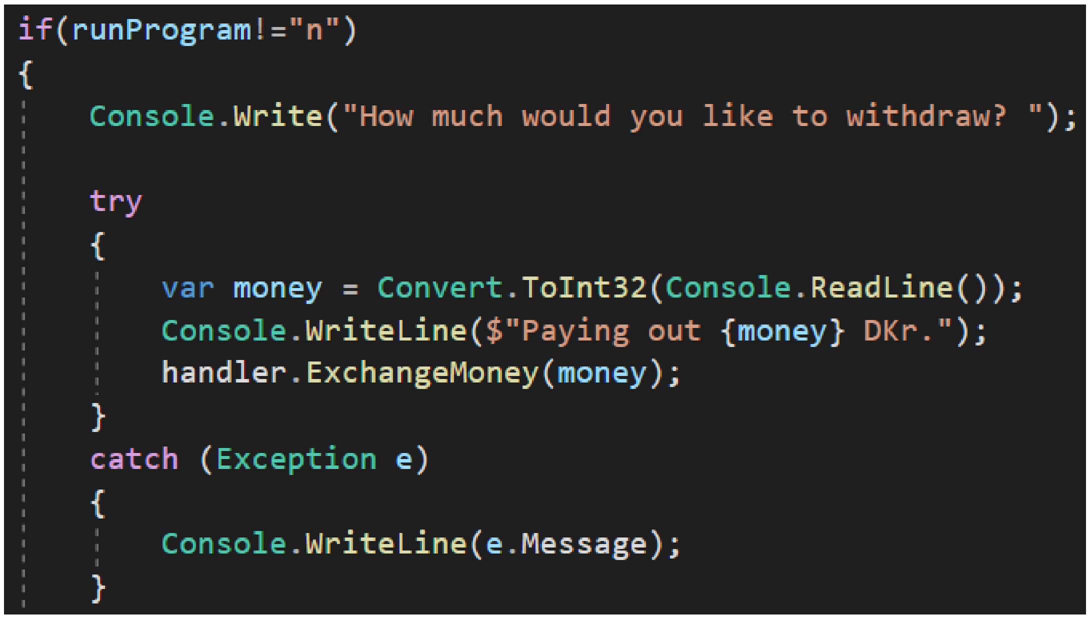
Main is only for setting up and starting the program.



```
class ATM
{
    2 references
    public Payout handler { get; set; }
    1 reference
    public ATM(Payout payoutHandler)
    {
        handler = payoutHandler;
    }
```

**Figure 6:** The constructor for the ATM defines the local payout based on the input

The ATM acts as the Client in this implementation of the Chain-of-Responsibility pattern. Within it contains its handler: Payout



```
if(runProgram!="n")
{
    Console.Write("How much would you like to withdraw? ");

    try
    {
        var money = Convert.ToInt32(Console.ReadLine());
        Console.WriteLine($"Paying out {money} DKr.");
        handler.ExchangeMoney(money);
    }
    catch (Exception e)
    {
        Console.WriteLine(e.Message);
    }
```

**Figure 7:** This is the startup function in the ATM class. It starts the payout class function exchange money

While this program may seem long, it is simply an interface for a user to write if they wish to start a withdrawal, input the amount, and then start the payout.

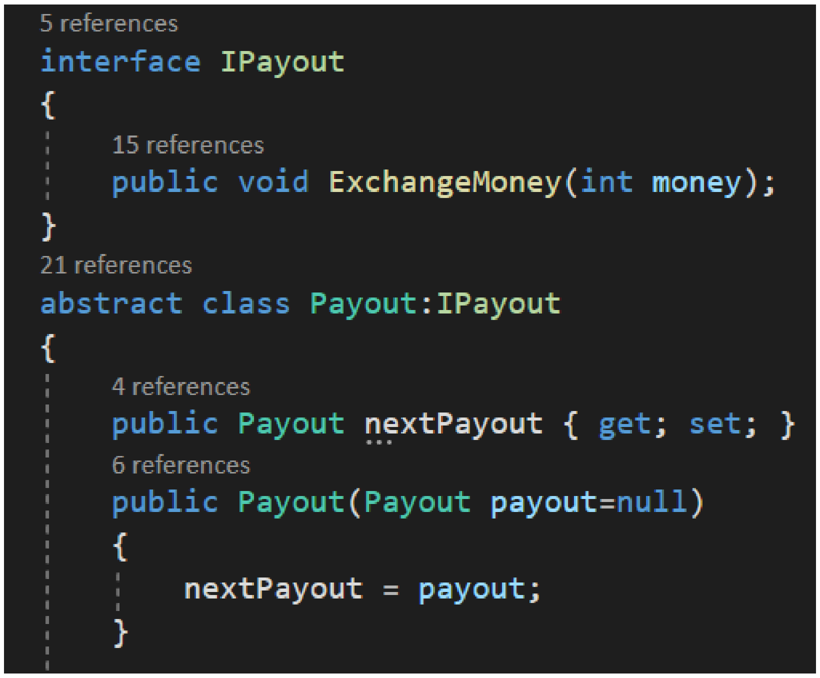
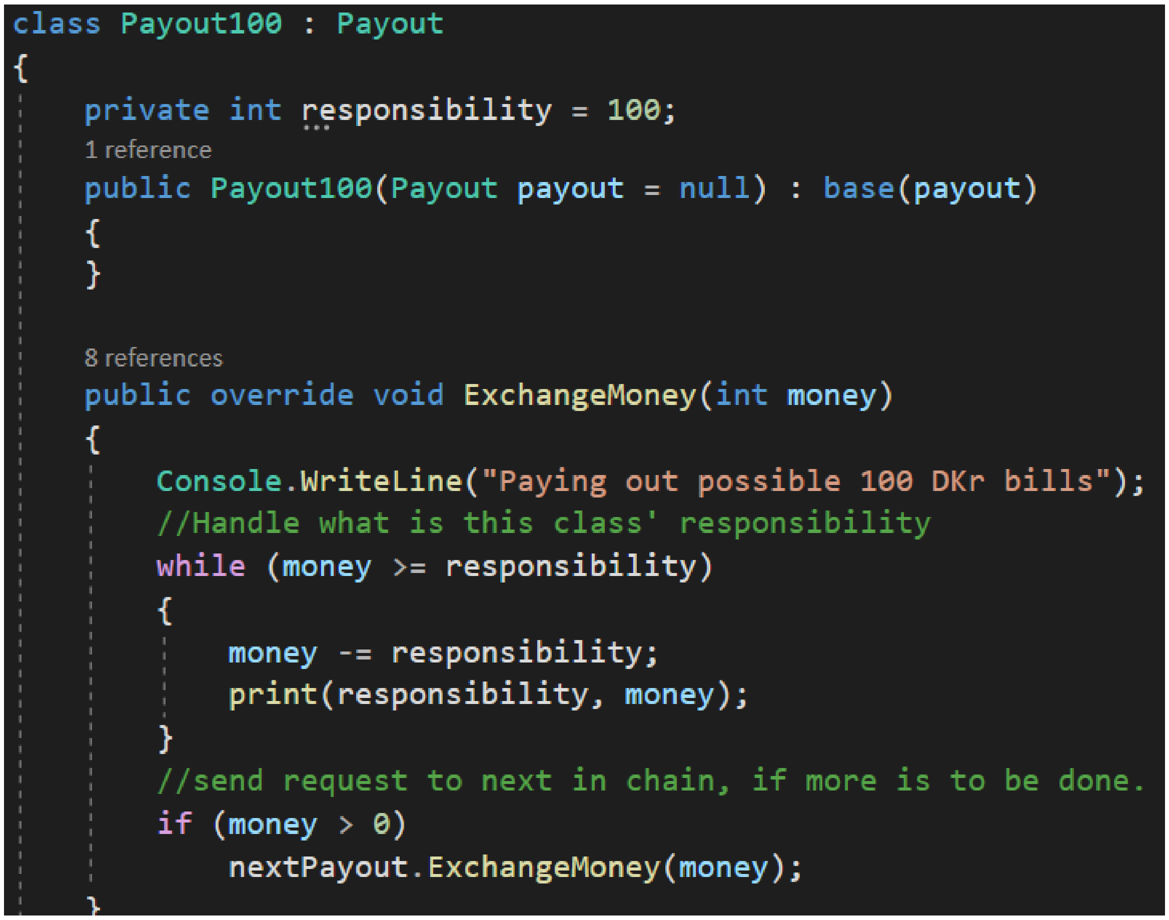

```
5 references
interface IPayout
{
    15 references
    public void ExchangeMoney(int money);
}
21 references
abstract class Payout:IPayout
{
    4 references
    public Payout nextPayout { get; set; }
    6 references
    public Payout(Payout payout=null)
    {
        nextPayout = payout;
    }
```

**Figure 8:** This is an abstract handler. This exists to create inheritance for concrete implementations

This is the handler itself. The core of Chain-of-Responsibility. From this function, all of the concrete functions are to be implemented. It is separated in interface and abstract class.



```
class Payout100 : Payout
{
    private int responsibility = 100;
    1 reference
    public Payout100(Payout payout = null) : base(payout)
    {
    }

    8 references
    public override void ExchangeMoney(int money)
    {
        Console.WriteLine("Paying out possible 100 DKr bills");
        //Handle what is this class' responsibility
        while (money >= responsibility)
        {
            money -= responsibility;
            print(responsibility, money);
        }
        //send request to next in chain, if more is to be done.
        if (money > 0)
            nextPayout.ExchangeMoney(money);
```

**Figure 9:** Implementation of a concrete payout function, to return 100

This is one of the many links in the Chain-of-Responsibility. This specifically is the first concrete handler, meaning the highest payout in this ATM is a 100 DKr bill. Every concrete handler's constructor also contains the next payout possibility as input, thereby creating the links in the chain.

## 6 Conclusion

While the Chain-of-Responsibility may seem like an unnecessarily complicated solution to problems that could be solved by using simpler solutions, such as recursion, what it provides is deceptively great. Just as when a new programmer starts their journey, classes and functions may seem unnecessary, what a Chain-of-Responsibility provides is an overview of a possibly complicated function, in a decidedly uncomplicated separation. However, it is not a very dynamic system. While it is easy to expand, it cannot change during runtime, such as the Observer Pattern. The input may change, but the chain itself must adhere to its client. While our specific implementation of an ATM may be incomplete as the danish currency holds additional bills and coins than the ones available here. It is still an astute example of a Chain-of-Responsibility. It has both mutation and a check, and fully exemplifies what a Chain of Responsibility has to give.

## 7 References

1. Dive Into Design Patterns, Shvets Alexander, excerpt from book: https://refactoring.guru/design-patterns/chain-of-responsibility