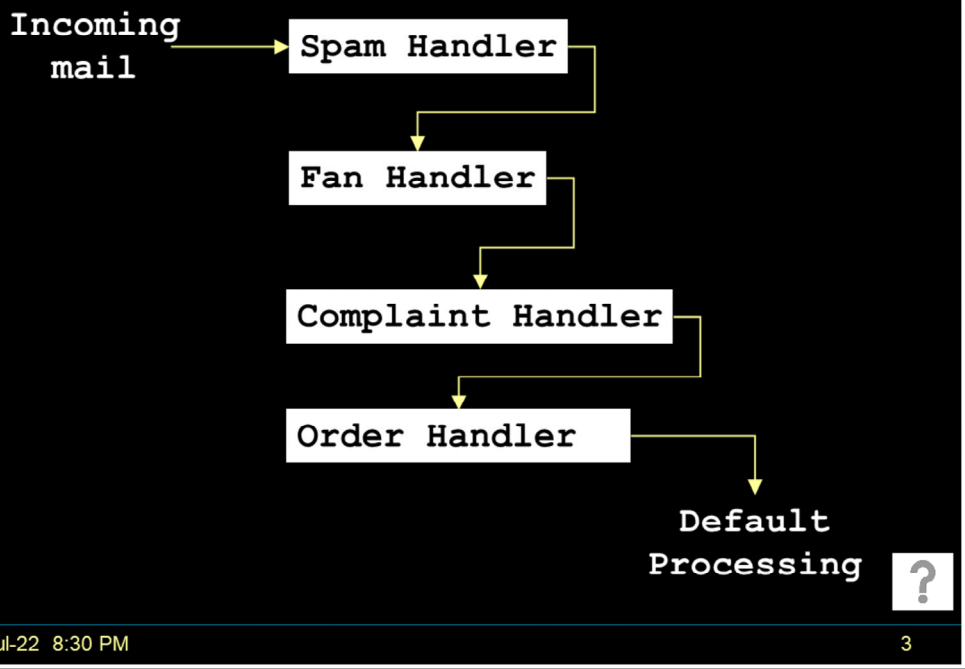
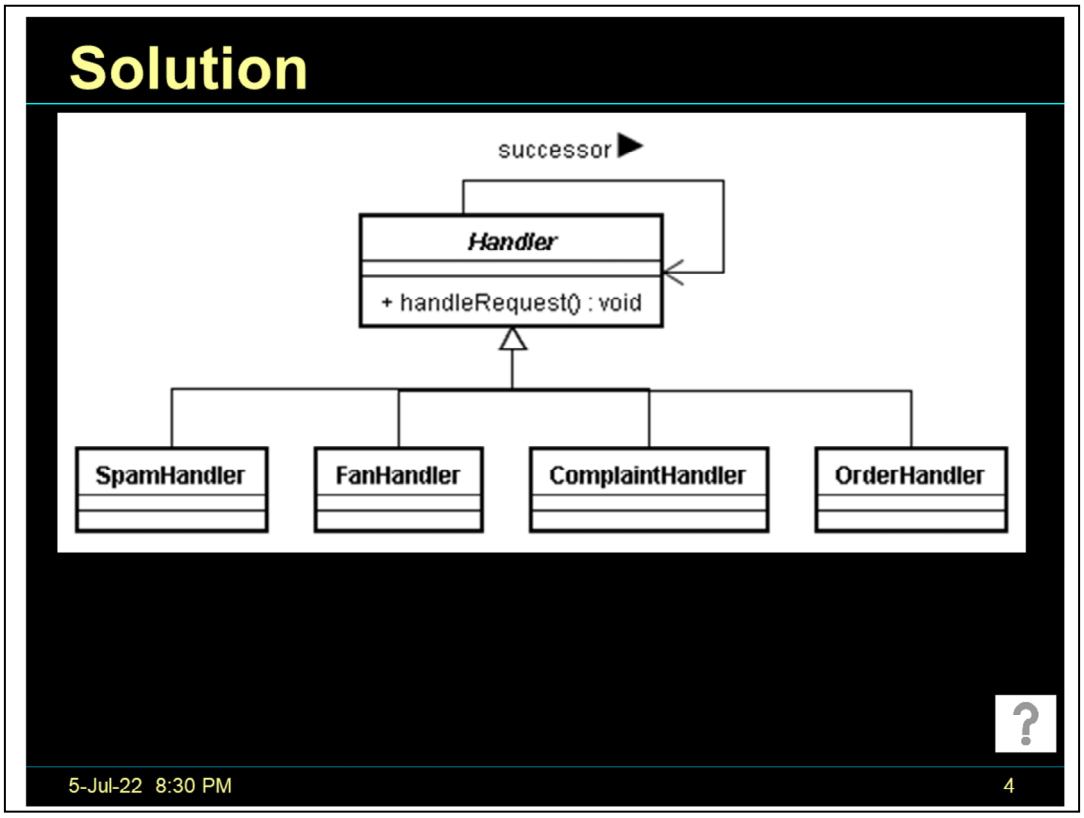
Chain of Responsibility DP

Problem

- A company receives lot of emails.
  - The spam has to be deleted.
  - The fan email should go to the design dept.
  - The complaints should go to the legal dept.
  - The orders of the customers should go to marketing dept.

Solution


Incoming mail
Spam Handler
Fan Handler
Complaint Handler
Order Handler
Default Processing
?



Code in Java and C# is simple. Code example in C++ is present.

Chain the receiving objects and pass the request along the chain until an object handles it.

Consequences:

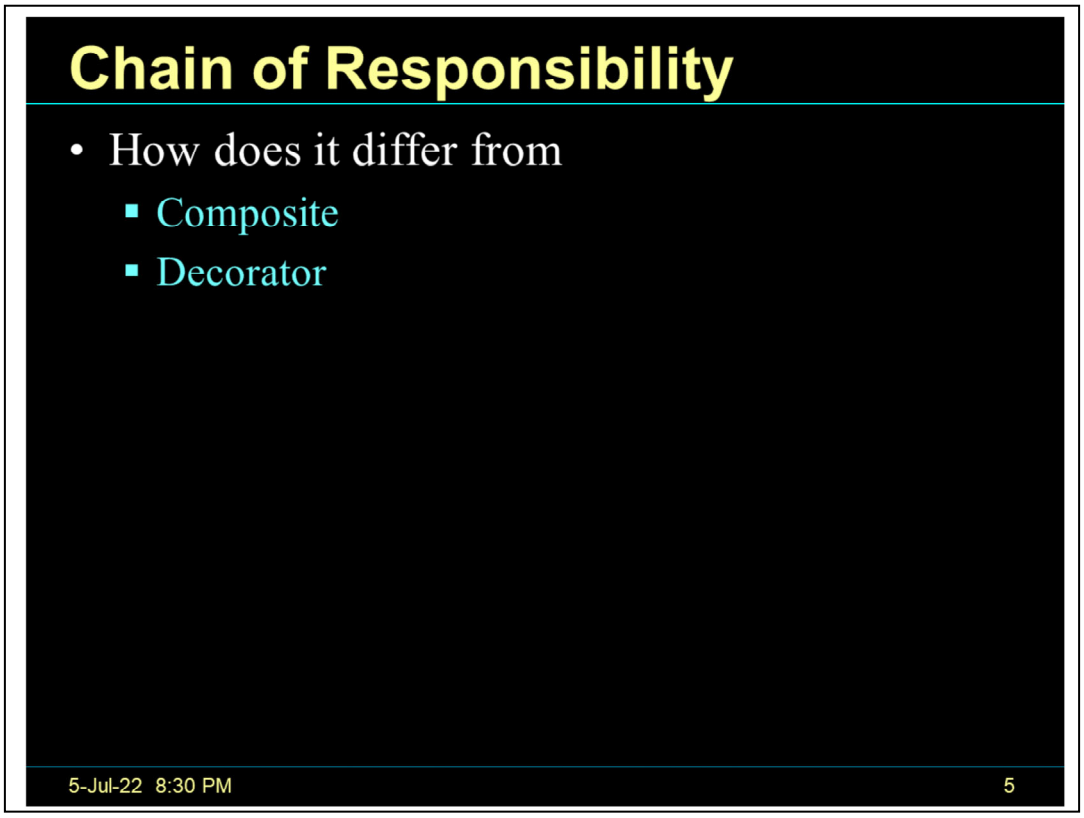
Decouples the sender of request from its receivers

Simplifies our objects because we don't have to know the chain structure and keep direct references to its members.

Allows us to add/remove responsibilities dynamically by changing the members or order of the chain.

Use Chain of Responsibility when

- More than one object may handle a request and the handler is not known in advance.
- We want to issue a request to one of several objects without specifying the receiver explicitly.
- The set of objects that can handle a request need to be specified dynamically.



Chain of Responsibility is often applied in conjunction with Composite.

There, a component's parent can act as its successor.

Comparison with Decorator:

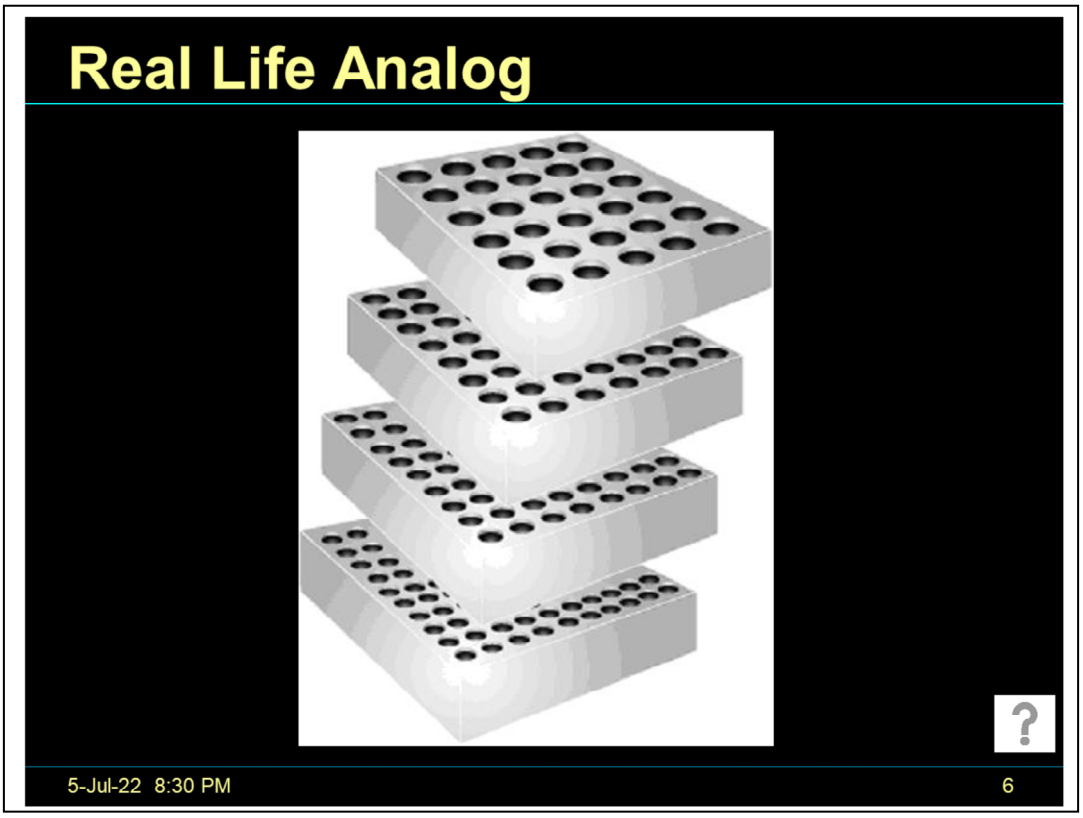
COR is like an object-oriented linked list. Decorator is like a layered architecture.

The client views the COR as “launch and leave” pipeline. The client views the decorated object as an enhanced object.

In COR, a request is forwarded until some handler takes care of it. Multiple handlers may handle one request. The Decorator object always performs pre or post processing as the request is delegated.

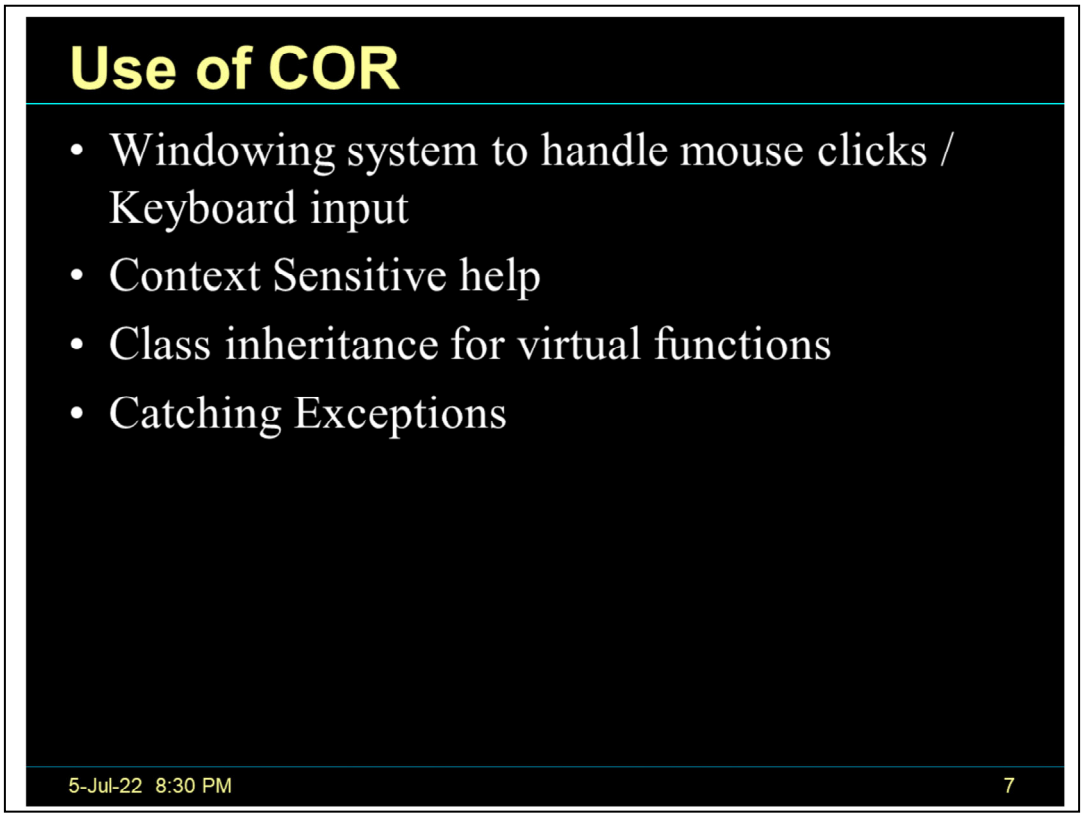
In Decorator, the core object is assumed by all layer objects. Ultimately the request is delegated to a core object. There is no special end of list processing.

In COR, an end of list default handler/processing is usually required.



Imagine you had a large number of coins that you need to sort. A simple (old-fashioned) way of doing this is to create a stack of trays, each one with holes in it that match the size of a given coin (one for half-dollars, one for quarters, one for nickels, etc…).

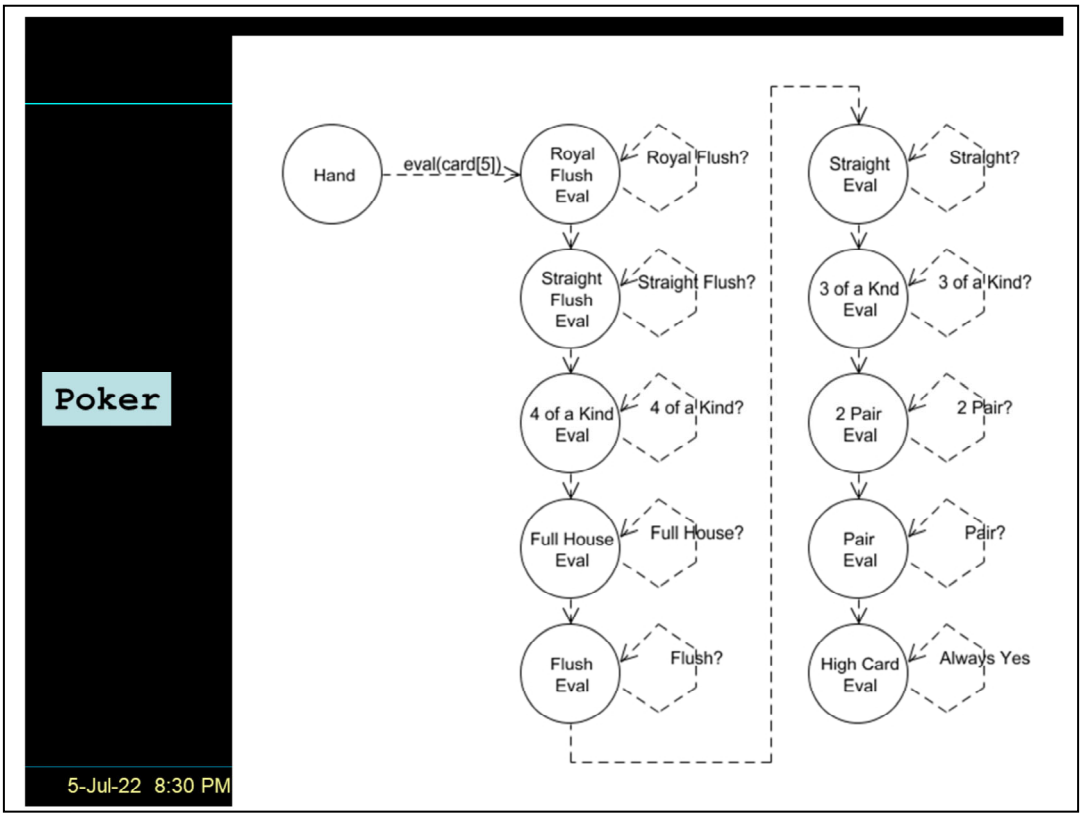
In this stack, the tray with the largest holes will be on top, the next largest will be underneath that, and so on until you have a tray with no holes at the bottom. Only the dimes (which have the smallest diameter) will fall all the way to the bottom tray. The pennies (next smallest) will be caught one tray up, and so forth.



It is commonly used in windowing systems to handle events like mouse clicks and keyboard entry.

It can be used to provide context sensitive help.

In Java, the class inheritance itself is a COR. If we call a method to be executed in a deeply derived class, the method is passed up the inheritance chain until a class containing the method is found.



The Chain of Responsibility gives us an opportunity to demonstrate how various, sometimes surprising issues can be composed into objects, rather than rendered into procedural code. This is not to say that *all* issues that were once dealt with procedurally should now be deal with in an object-oriented way, but rather that it is helpful to know that such options exist, to allow the development team to make the most advantageous decision.

Our example here is the game of poker. If we were creating a software version of this game, one issue that would have to be dealt with is the "what beats what" rules of poker.

In poker, the hand with the highest card wins, unless one hand has a pair. The highest pair wins unless one hand has two pair. The hand with the highest “higher pair”wins unless one hand has three of a kind. Three of a kind is beaten by a straight (5 cards in sequence, like 5,6,7,8,9). A straight is beaten by a flush (5 cards in the same suit). A flush is beaten by a full house (three of one kind, a pair of another). A full house is beaten by 4 of a kind. 4 of a kind is beaten by a straight-flush (5 cards in sequence and all of the same suit), and the highest hand is a royal flush (the ace-high version of the straight flush).

These are business rules, essentially. Imagine we decided to assign each of these hand types a numeric value, so that once we determined the value of two or more hands, we could easily determine the winner, second place, and so forth. The Chain of Responsibility could be used to capture the rules that bound each hand type to a specific numeric value:

The Hand would send an array of its 5 cards to the first "evaluator" in the chain. Of course, we'd likely build this chain in an object factory, and so the hand would not "see" anything more than a single service object.

The Royal Flush Eval object would look at these 5 cards, and determine, yes or no, if they qualified as a "royal flush". If the cards did qualify, the Royal Flush Eval object would return the numeric value we have assigned to Royal Flushes. If the cards did not qualify, then the Royal Flush Eval would delegate to the next object in the chain, and wait for a result, which it would then return to the Hand.

This delegation would continue until one of the Eval objects self-elected, and then the value would return up the chain and back to the Hand. Note that every possible poker hand has some card which is highest, and so we have a default condition we can use to end the chain, the High Card Eval.

The key point here is this: A Royal Flush is also a Flush. It is also a Straight. If we handed 5 cards which were in fact a Royal Flush to the Straight Eval class, it would respond in the positive: "yes, that's a Straight". 3 of a Kind is also a pair, a Full House is also 3 of a Kind, and so forth.

There is a business rule about this: if you have a Royal Flush, no one can call your hand a Straight, even though this is technically true. There is an ordering dependency about these rules… once you determine that a hand is a Full House, you do not ask if it is a Pair.

This design captures this set of dependencies, but not in conditional code. It is captured simply in the order of the evaluators in the chain, and can therefore be maintained simply by changing that order (there are many variations on Poker, as you may know).

We must emphasize here that the point is not to say that all procedural logic should be converted to object structures.

This would create overly-complex designs, would tend to degrade performance, and would challenge many developers who tried to understand the design. However, it is important to know that object structures are a possible solution to issues that we would normally think of procedurally, to enable us to make informed decisions.

The advantages to using an object structure include testability, encapsulation, and open-closed-ness.