Design patterns Behavioral

Behavioral pattern focused on how objects interact one each other

**Chain of responsibility pattern**

The chain of responsibility is a behavioral pattern that decouples request from a handling object in a chain of handlers until it has finalized recognized

Use chain of responsibility patter when you want to give more than one object a chance to handle a request

Concepts:

* Decoupling the de sender and the receiver objects
* The receiver contains reference to next receiver
* Promotes loose coupling
* No handler – OK

Examples

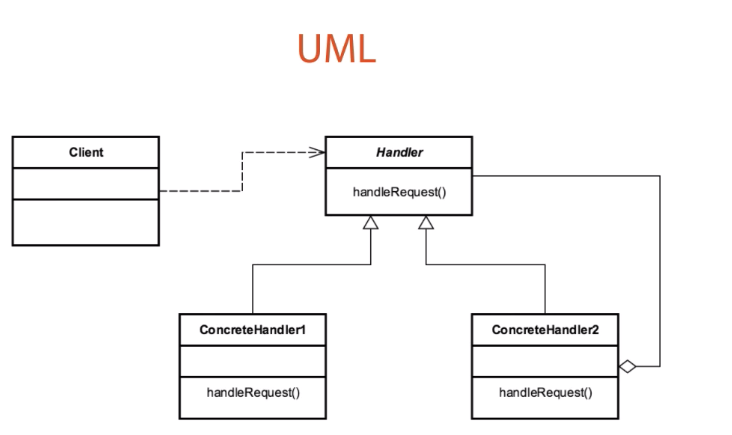
Java.util.loggin.Logger#Log()

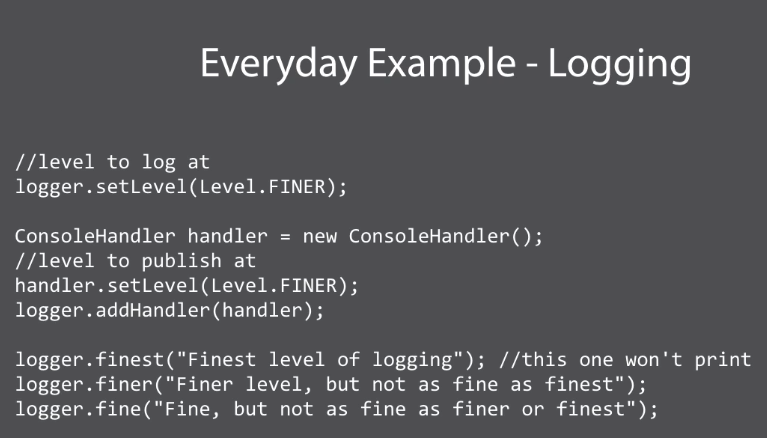
Javax.servlet.Filter#doFilter()

Spring Security Filter Chain

Design

* Chain of receiver objects
* Handlers is interface based
* Concrete Handler for each implementation
* Handler, Concrete Handler



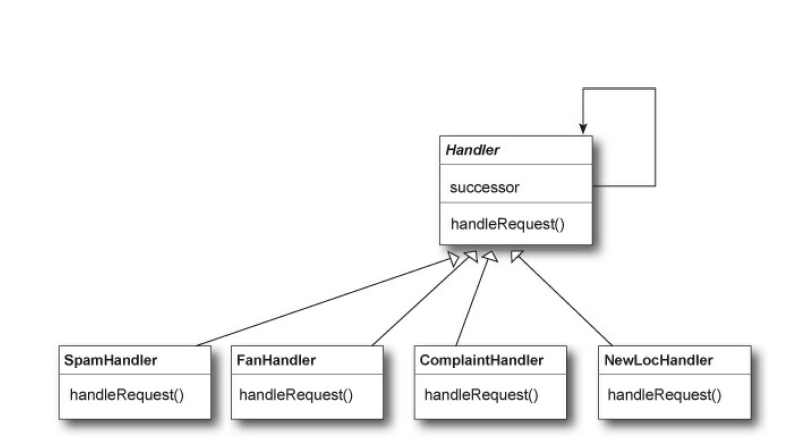


Pitfalls

* Handling /Handler guarantee we aren’t guaranteed that somene along the chain will in fact handle our requestst.
* Runtime configuration risk
* Chain length/performance issue (The performance could start to degrade in long chains)



Example:



**Command pattern**

Command pattern is a behavioral design pattern that let you encapsulate each request as an object

Concepts:

* Encapsulate each request as an object
* Object oriented callback
* Increase maintainability because decouple sender form processor
* Often used for “undo” functionality

Examples

Java.lang.Runnable

Javax.swing.Action

Design

* There is an object per command (Sometimes is called that it violates oo principles design )
* Command interface
* Execute method
* Unexecute method
* Reflection
* Command, invoker, concreteCommand

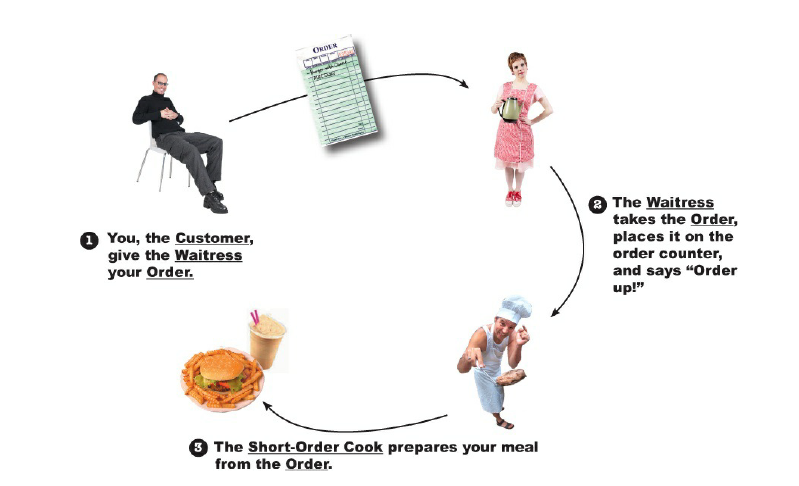
**Head First Explanation**

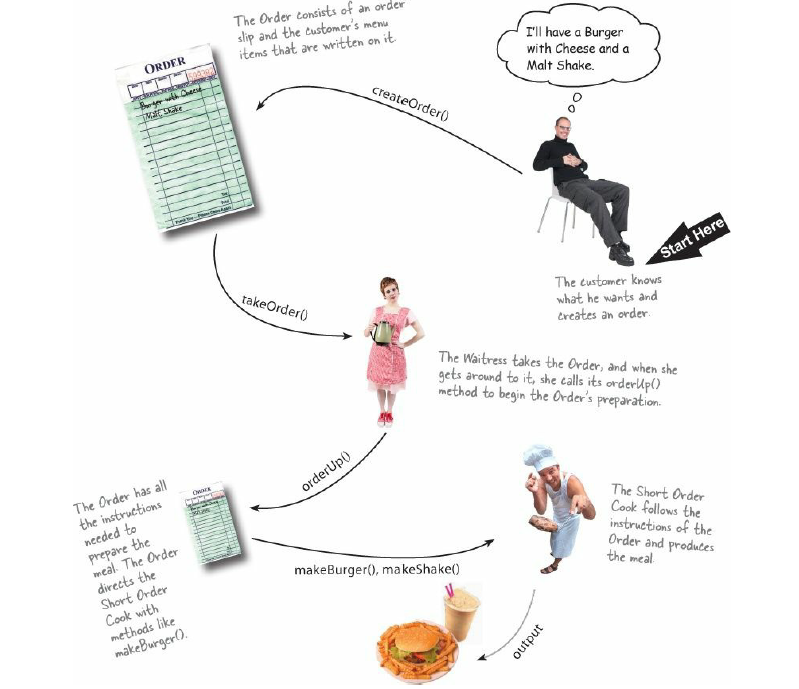
**Command pattern:**

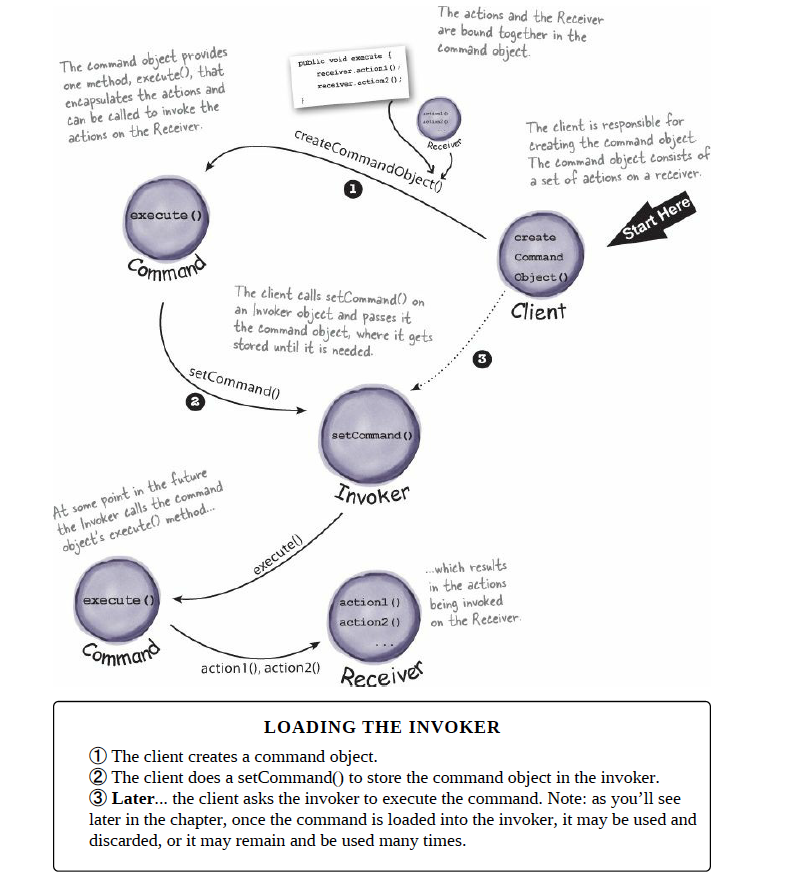
The main goal of command pattern is to encapsulate method invocation in to object, by encapsulating method invocation, we can crystalize pieces of computation so that the object invoking the computation doesn’t need to worry about how to do things, also we can do some wickedly thins whit these encapsulated method invocations, like save them away for loggin or reuse them to implement undo operations in our code.

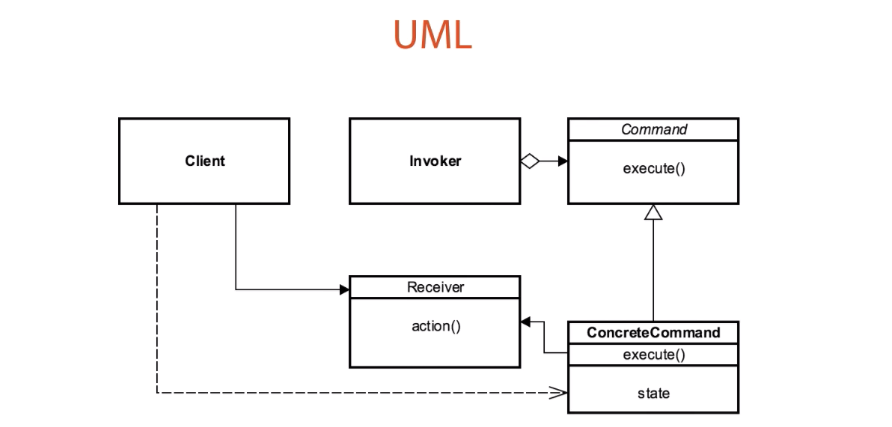
The command pattern allows you to decouple the requester of an action from the object that actually performs the action. So, here the requester would be the remote control and the object that performs the action would be an instance of one of your vendor classes.

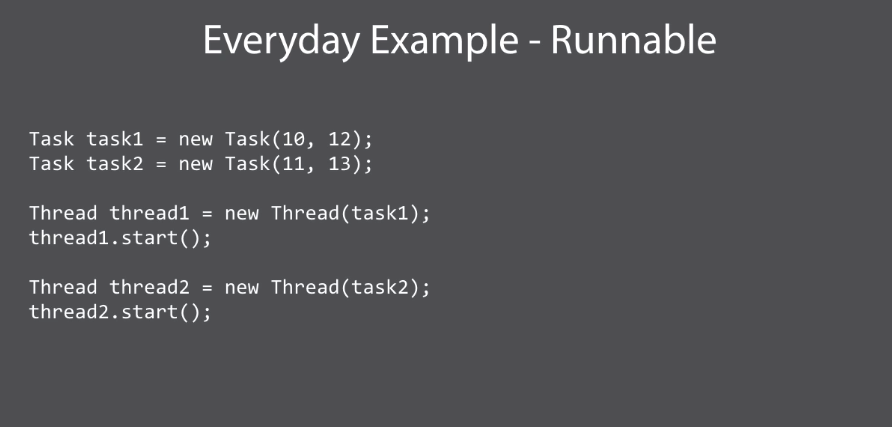
You can do that decoupling by introducing command objects in to your design. A command object encapsulates a request to do something (like turn on a light) on a specific object (Sy living room ligh object). So if we store a command object for each button, when the button is pressed we ask the command object to do some work. The remote doesn’t have any idea what the work is, it just has a command object that knows how to talk to the right object to get the work done.





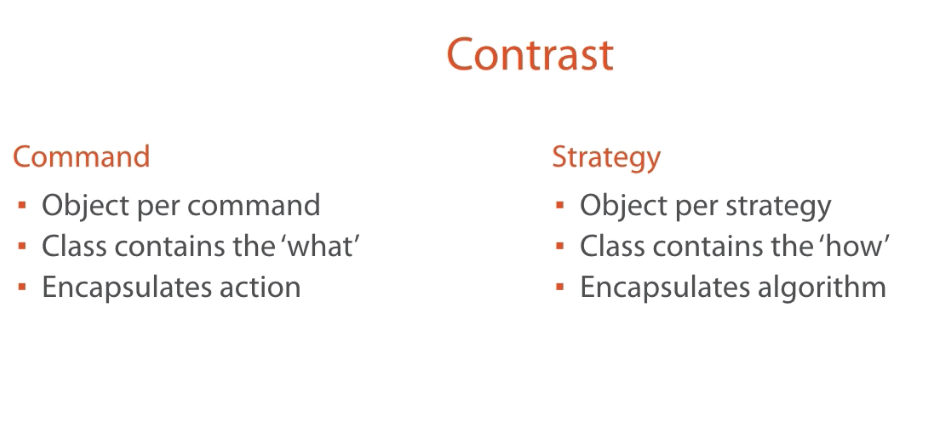






Pitfalls

* Dependence on other patterns
* Multiple commands
* Prototype for copies of commands



**Interpreter Pattern**

The interpreter pater is a behavioral pattern that you used to represent the grammar of a language, a lot of tools used this pattern to parse various aspect of the grammar.

Use interpreter pattern to build an interpreter for a language

**The interpreter Pattern requires some knowledge of formal grammars**

Concepts:

* Represent a grammar (music notation, math equation, even another language)
* Use to interpret a sentence
* Map domain specific language (Sql, XML parser)
* AST abstract syntax Tree

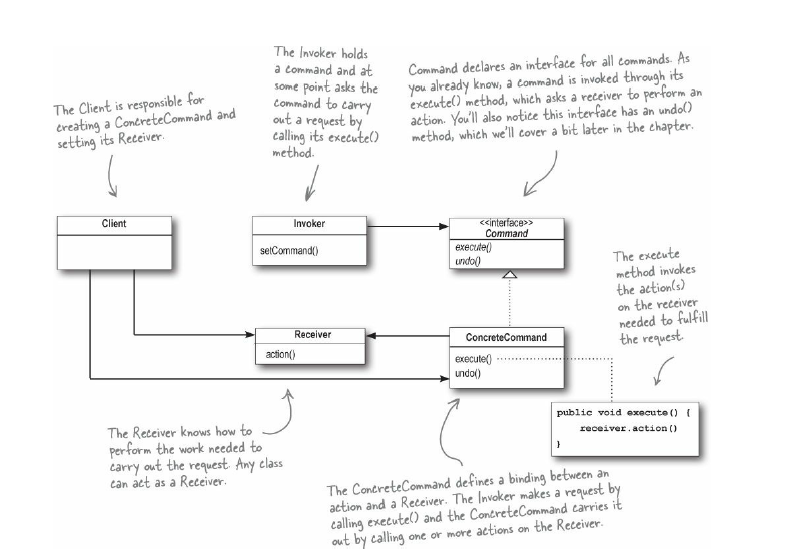
Examples:

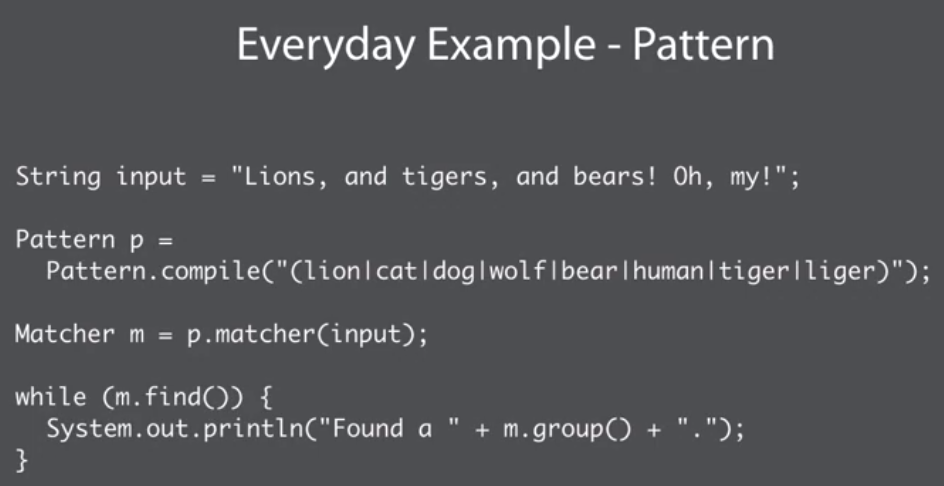
Javal.util.Pattern -> represent a compiled regular expression

Java.text.Format -> abstract based class represent Locale sentive content such as dates, numbers or strings

Design

* There’s an AbstracExpression or abstract base class that declares an interface for executing an operation
* That operation is an interpret method
* Expression are then broken into terminal expression
* NonterminalExpression (represent compound expressions)





Pitfalls

* Difficult to maintain complex grammar
* One class per rule
* Use of other patterns
* Adding new variant
* Specific case to the problem that you are trying to solve

