Design Patterns

-WHY?

* Communication
* Common vocabulary

Pattern Groups

* Creational
* Structural
* Behavioral

Structural

-Adapter

-Bridge

-Composite

-Decorator

-Façade

-Flyweight

-Proxy

ADAPTER

-legacy cod (cod mostenit)

-plug adapter

-choose it when you have a client talk to an existing interface

-the case when our system is a legacy app or module that we don’t want to, or cant possibly change

-translates requests from the client to the code we are adapting to

-Exemples:

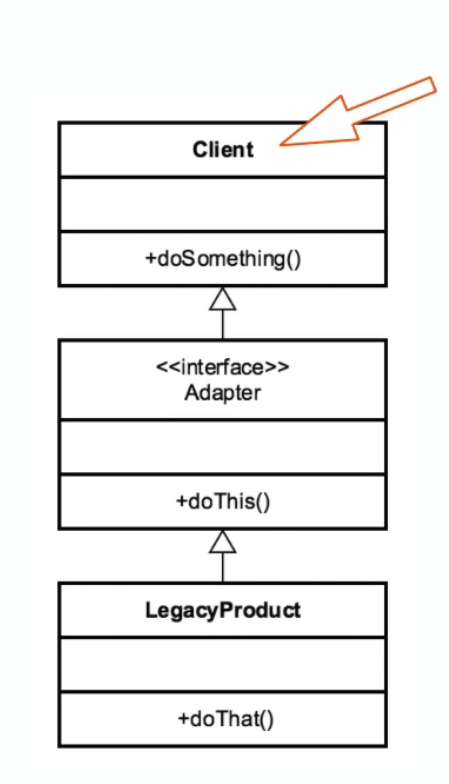
* Array -> List conversion
* Streams Classes surrounding I/O

-Client centric

Integrate new with old

-Interface, but not required

-Adaptee can be the implementation



Exemple : Arrays->List

Integer[] arrayOfInts = new Integer[] {42,43,44};

List <Integer> listOfInts = Arrays.asList(arrayOfInts); //method that is an adapter to convert an array of something into a list from the collection API

-ex: EmployeeAdapter for EmployeeDB employeeBD = new EmployeeDB( “123 ” , “name”,”email”);//nu merge, trebuie un adapter intre ele

Employees.add( new EmployeeAdapter(employeeDB));//create new instance every time because it is holding a reference to the instance that we want to adapt

|  |  |
| --- | --- |
| Adapter | Bridge |
| -works after code is design | -design upfront (proiectare în avans) |
| -legacy code | -abstraction and implementation vary independently |
| -provide different interface to our legacy code than was originally intended | -built in advance that we can provide a layer of abstraction abd let both systems be flexible while we are implementing and creating them |
| -make unrelated classes work togheter |  |
| ~both adapt multiple systems and work in concert with one another | |

Summary

-simple solution to a very descript problem

-easy to implement

-use to integrate with legacy code that we can’t or don’t want to change

- can provide multiple adapters

BRIDGE

-similar to the adapter

-main difference: BRIDGE works with new code whereas the adapter works with legacy code

-decouple abstraction and implementation with the following techniques : encapsulation, composition, inheritance

- change in abstraction wont affect client

-Examples :

-Driver (Bridge is in a lot of ways just a driver)

-JDBC Drivers -> we have an interface that we work with and a driver that works with the underlying database

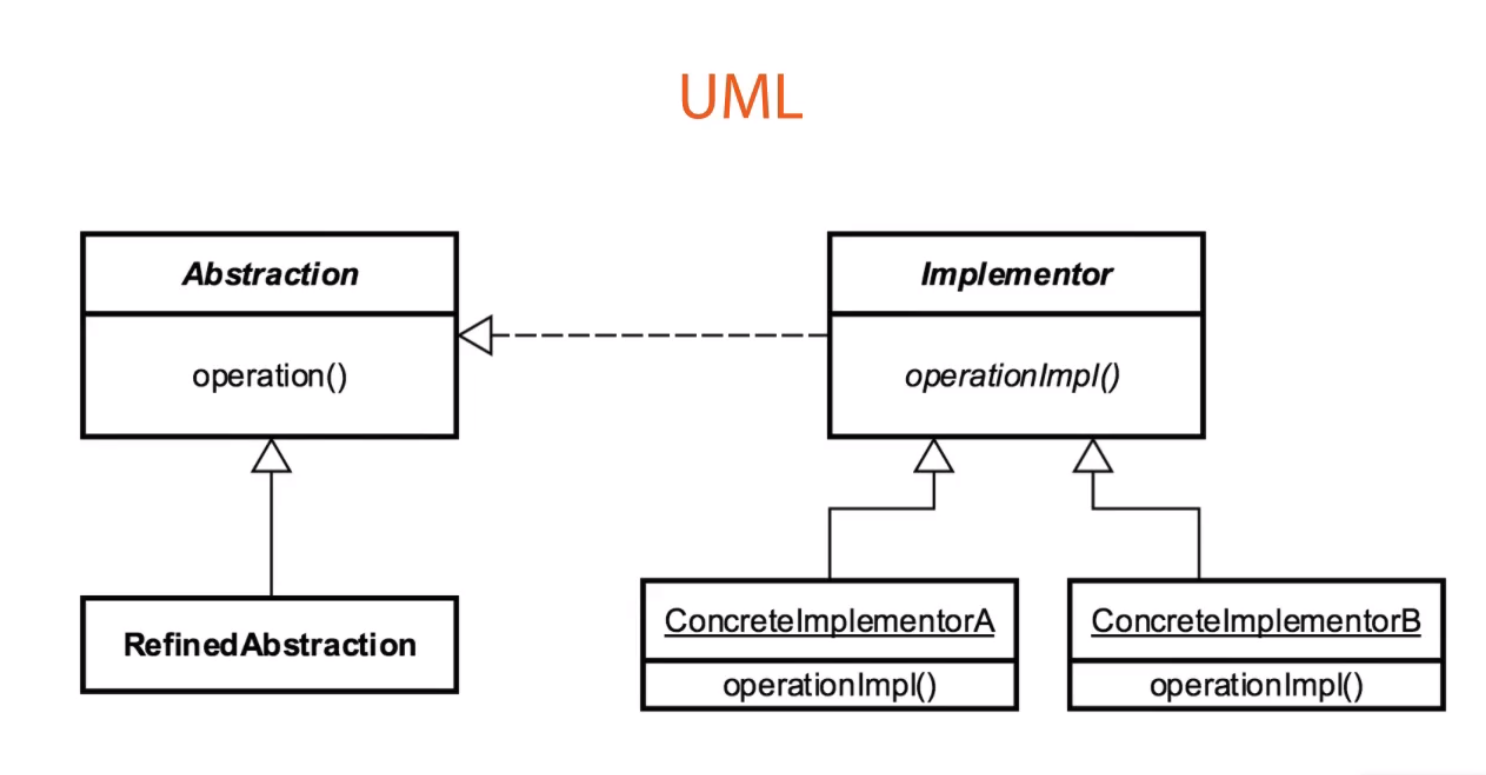
Design

-utilize interfaces and abstract classes

-composition over inheritance

-more than Composition

-expect change from both sides



-Abstaction in this case is an Interface – RefinedAbstraction

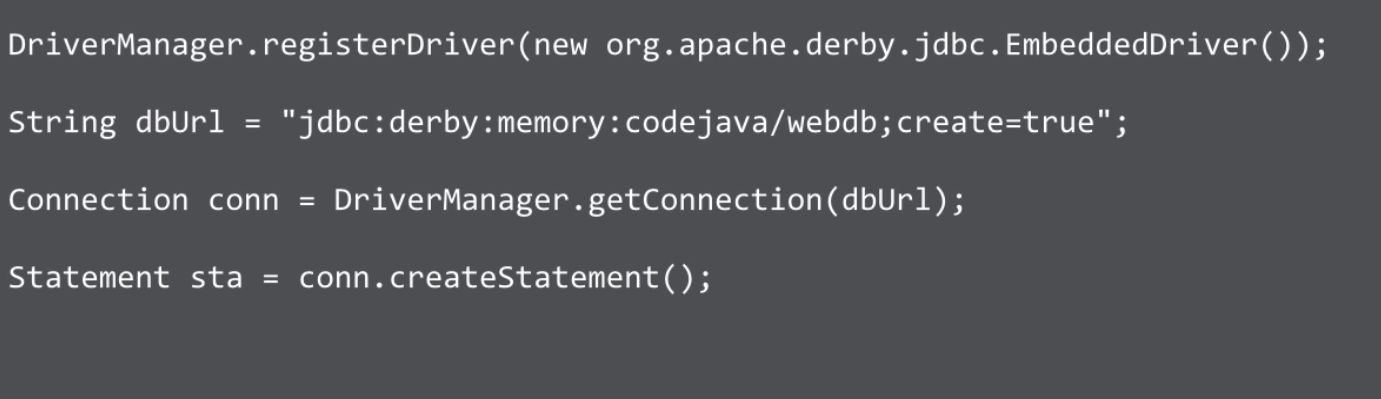
Example: JDBC

JDBC is an API for execiting SQL statements

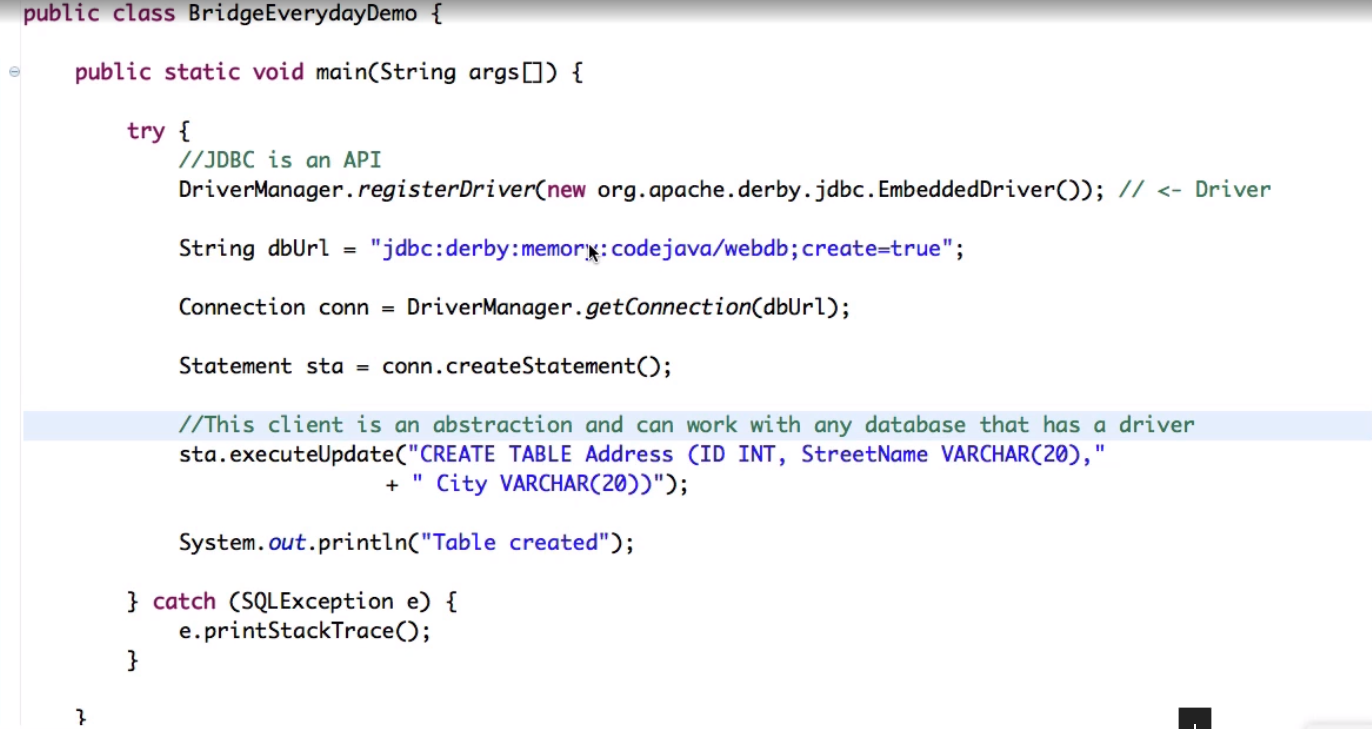
Classes that implements the interface are:

-JDBC Drivers

-applications that reky on these drivers are abstraction that can work with any database for whice a JDBC exists



DEMO:



COMPOSITE

-components represent par or whole structure

-compose objects into tree structures

- individual object treated as c composite object

-same operations applied on individual and composites and expect them to work the same way

Examples:

-java.awt.Component

-RESTful service GETs

Design

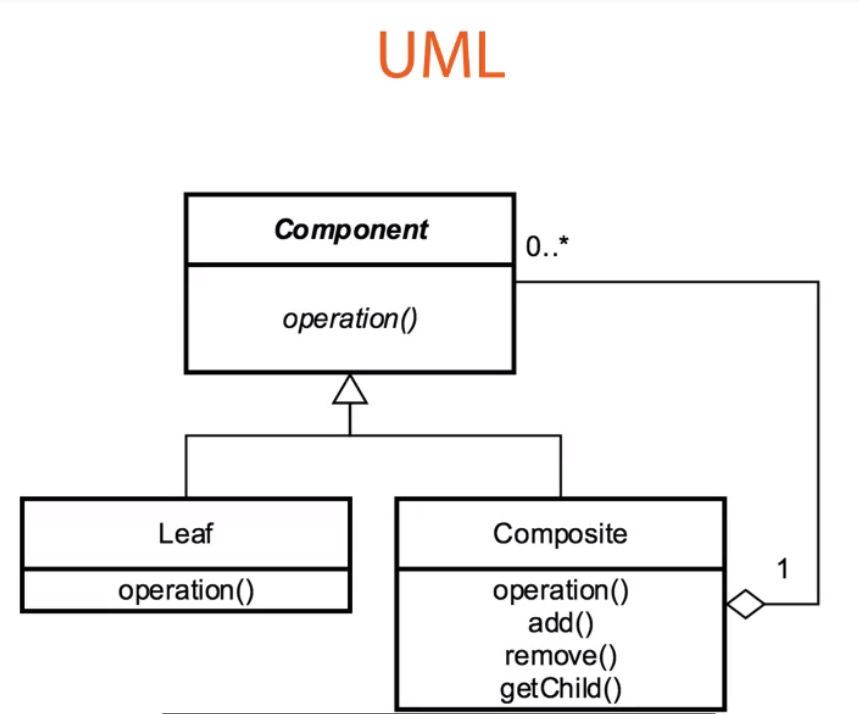
-tree-structured

- the root of the tree starts with a component

-component - leaf

-composite of objects

- composite knows about child components



The component class is the abstraction for all components, including composite ones.

It declares the interface for objects in the composition. The component can also define methods for accessing the parent, but is isn’t necessary to still use this pattern correctly.

The leaf represents the leaf objects or nodes in the composition. It should ne noted that it also implements all of the component’s methods.

The composite though represents a composite component, or a component that has children, and implements methods to manipulate those children. It also implements all the component’s methods as well, but typically delegates the functionality to its children.

So a leaf and a composite have the same functions inside of them, a composite just knows about which children it has and what it can do with those children

Example: - map

|  |  |
| --- | --- |
| COMPOSITE | DECORATOR |
| -tree structure  -leaf and the composite have the same interface to the client  -provides unity between objects | -contains another entity  -composition – object containing another one  -modifies the behavior of the contained entity  -adding functionality to an entity that it didn’t originally have  -decorates the underlying object, but doesn’t necessarily change it |

Summary

-generalizes a hierarchical structure (ex – Menu App)

-can simplify things too much

- easier for the client (Ex- collection – we can add items or an entire collection)

- Composite != Composition

DECORATOR

-hierarchical type pattern that builds functionality at each level while using composition from similar data types

When you use it ?

* When you want to wrap another object to add functionality to it
* You can add behavior to an object without effecting other parts of the hierarchy if you don’t want to
* More than just inheritance
* -You are controlling which pieces compliment your object, not necessarily trying to override it like with inheritance
* Follows Single Responsibility principle – every class shoul have responsibility over a single part of the functionality provided by the software, and that responsibility should be entirely encapsulated by the class ->> it should do one thing and do it well
* You can compose behavior dunamically by using one of the subclasses that decorate your object

Exemple: java.io.InputStream

Java.util.Collections also has a checkedList method

All UI components

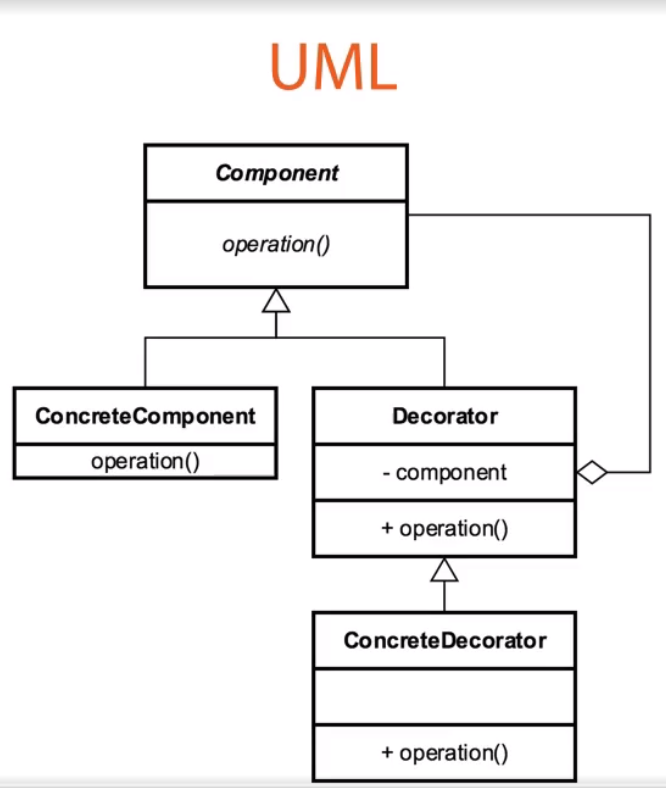
Design

-inheritance based

-utilizes composition and inheritance (is-a, has-a)

-there is a common component, but functionality is added in the subcomponents

- constructor requires instance of the component from hierarchy, which enables it to build upon that and use composition rather than inheritance to override which individual fields that is wants to



-Component class- interface / abstract class that has a concrete instance represented by the ConcreteComponent Class

-The ConcreteComponent – what we are going to eventually decorate

- the Decorator – the base decorator or wrapper that we will extend and create other decorators from

- ConcreteComponent & Decorator extend the component so that they can be treated the same

-from here you can create multiple ConcreteDecorators to decorate our object and provide functionality as we develop

Summary

-original object stays the same, so we don’t have to keep creating concrete objects to add functionality to them, we can utilize the decorator to do that

-unique way to add functionality to those concrete objects

-confused with ineritance

- can be more complex for clients

-great pattern if you don’t want to modify that base object

FAÇADE

-provides a simplified interface to a complex or difficult to use system that is often the result of a poorly designed API.

When we use it ?

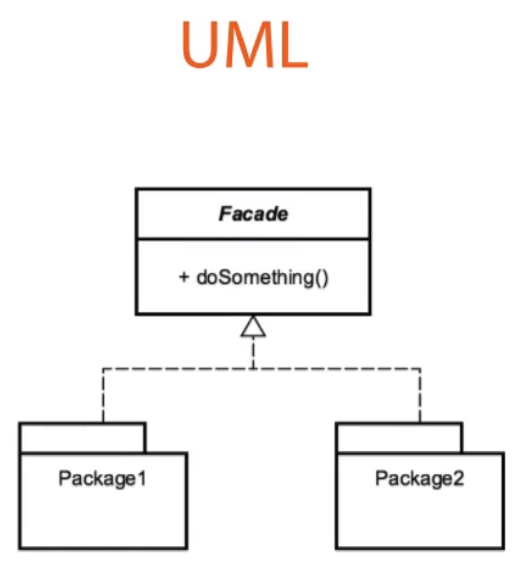
* Make an API easier to use – hide details from the client
* Reduce dependencies on outside code
* Simplify the interface or client usage ->> wrap complex code with an interface using this façade to make it simpler for the end user
* Refactoring Pattern

Examples:

-Java.net.URL

Design

* Class that utilizes just composition, not inheritance
* Encompasses the entire lifecycle of whatever object you‘re dealing with, but it doesnot necessarily have to order to considered a correct usage of this pattern



* Through composition -> multiple other packages/ APIs

|  |  |
| --- | --- |
| FAÇADE | ADAPTER |
| -simplifies Interface  -works with composites  -cleaner API | -also a refactoring pattern very similar to the façade, but it modifies behavior (adds) |

Summary

-simplify Client Interface

-easy pattern to implement

-refactoring pattern

FLYWEIGHT

-minimize memory used by sharing data with similarly types objects

When we use it ?

-make more efficient use of memory

-optimization pattern

-large number of similar objects

-immutable ->> objects that their state cant be changed after creation

Examples :

-java.lang.String ->> Strings are immutable objects

-method valueOf(int)

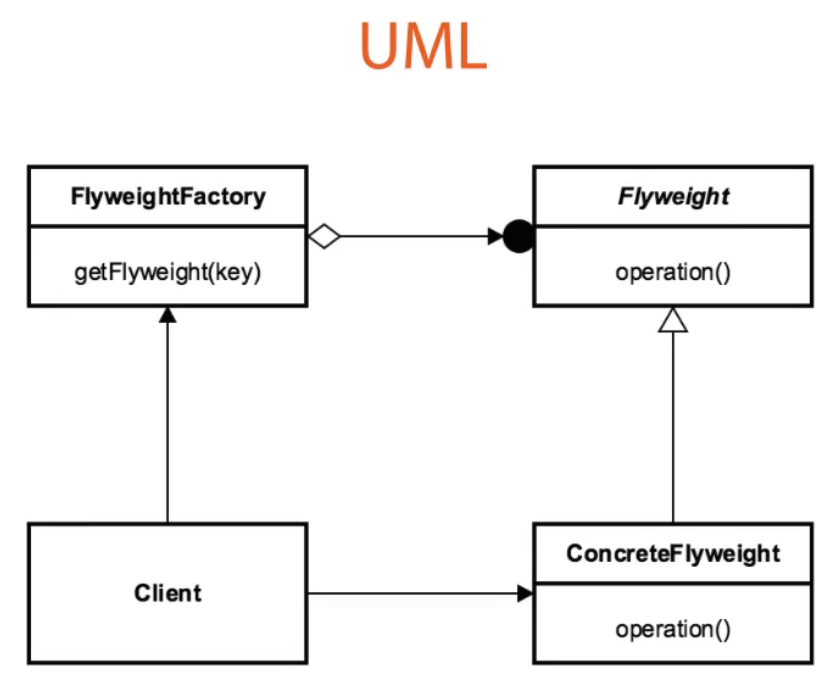
-Boolean, Byte, character, Short, Long

Design

-pattern of patterns

-utilizes a factory Pattern

-Creational pattern inside those structural patterns



* The Client is what is requesting the Flyweight object, although oftentimes it doesn’t even know that it’s a flyweight.
* It requests it from a FlyweightFactory
* The factory returns the cached object or it creates a new instance of the flyweight, eventually at the end of the process if one doesn’t already exist in our factory
* The ConcreteFlyweight is in the end whats gets returned to the client, although it thinks it’s just getting that object back
* Oftentimes a client doesn’t know the underlying structure and just has a simple interface, but regardless, this is what happens underneath it all

|  |  |
| --- | --- |
| FLYWEIGHT | FAÇADE |
| -memory optimization  -optimization pattern  -deals with immutable objects | -refactoring pattern  -simplified Client  =provides a different interface |

Summary

-great for memory optimization

-complex

-used a lot by the core API

PROXY

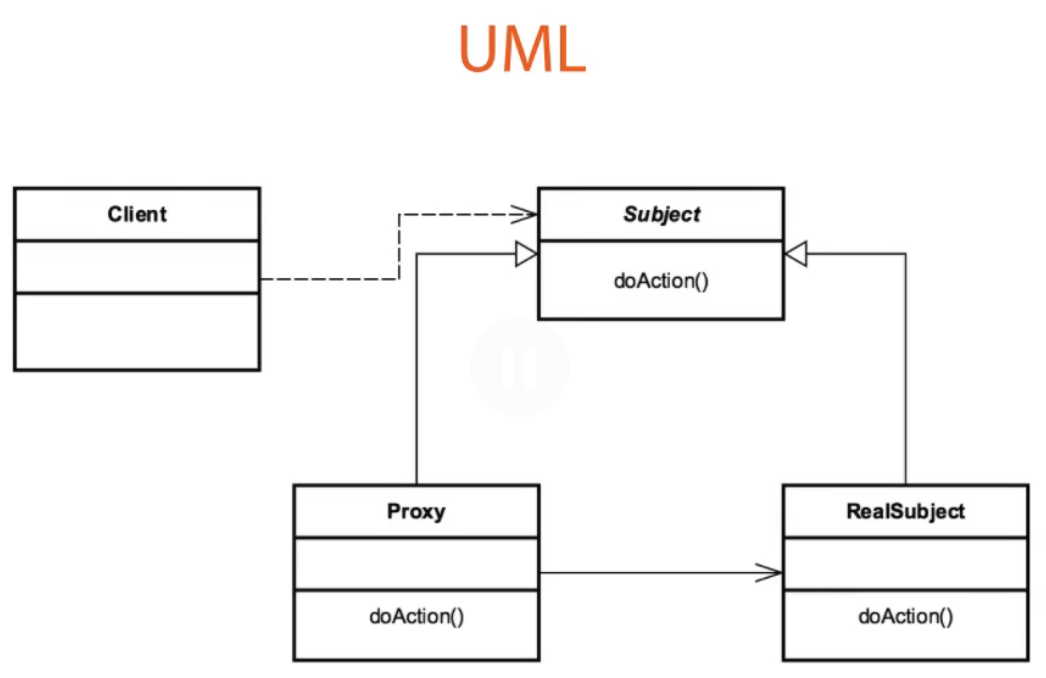
* Pattern that acts like an interface to something else
* Want to wrap a real object with a proxy for various reasons
* You create an interface to an object by wrapping it with a class to create that proxy
* Add more functionality to that wrapped object
* Security, Simplicity, Remote, Cost
* Proxy called to access real object
* We’ll have an interface, then a proxy that’s wrapping the real object, and then the underlying real object

Examples:

* Java.lang.reflect.proxy -> object is a mechanism to facilitate creating proxy pattern using Java
* Java.rmi.\* -> focused around proxy and remote method invocation, so as we looked at the concepts up above

Design

* An intermediary object that intercepts calls
* Interface based
* Interface and Implementation Class that the proxy resides in between



-The Client Class that’s going to make a reference call to some object, some Subject, and the real subject that we want it s going to be intercepted with this proxy

-The Subjet ->> interface to whatever the implementation class is that we want to retrieve

-The Proxy, using an InvocationHandler and the Proxy class in Java, intercepts that call and makes the call to the RealSubject, or if it’s a case like security would deny it or do something different, and turns around and sees if it needs to load that, if it’s going tp pull it from a cache, or whatever it’s going to do, so it then decides that,