HowToDoInJava

Proxy Design Pattern

Last Updated: August 30, 2021 By: Lokesh Gupta Structural Patterns Design Patterns

According to GoF definition of **proxy design pattern**, a proxy object provide **a surrogate or placeholder** for another object to control access to it. A proxy is basically a substitute for an intended object which we create due to many reasons e.g. security reasons or cost associated with creating fully initialized original object.

1. When to use proxy design pattern

A proxy object hides the original object and control access to it. We can use proxy when we may want to use a class that can perform as an interface to something else.

Proxy is heavily used to implement lazy loading related usecases where we do not want to create full object until it is actually needed.

A proxy can be used to add an additional security layer around the original object as well.

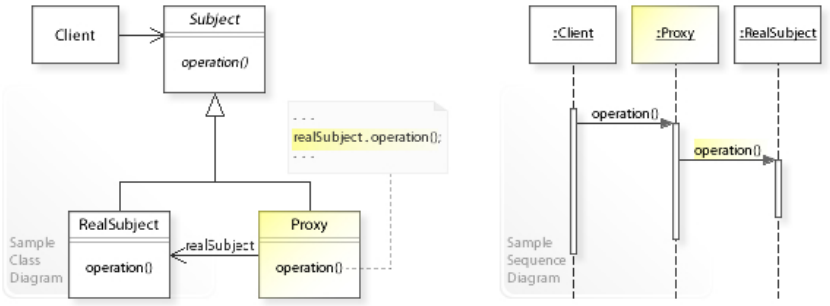
2. Real world example of proxy pattern

- In hibernate, we write the code to fetch entities from the database. Hibernate returns an object which a proxy (by dynamically constructed by Hibernate by extending the domain class) to the underlying entity class. The client code is able to read the data whatever it needs to read with the proxy.
  These proxy entity classes help in implementing lazy loading scenarios where associated entities are fetched only when they are requested explicitly. It helps in improving performance of DAO operations.

- In corporate networks, internet access is guarded behind a network proxy. All network requests goes through proxy which first check the requests for allowed websites and posted data to network. If request looks suspicious, proxy block the request – else request pass through.
- In aspect oriented programming (AOP), an object created by the AOP framework in order to implement the aspect contracts (advise method executions and so on). For example, in the Spring AOP, an AOP proxy will be a JDK dynamic proxy or a CGLIB proxy.

3. Proxy design pattern

3.1. Architecture



Proxy design pattern

3.2. Design participants

- **Subject** – is an interface which expose the functionality available to be used by the clients.
- **Real Subject** – is a class implementing `Subject` and it is concrete implementation which needs to be hidden behind a proxy.
- **Proxy** – hides the real object by extending it and clients communicate to real object via this proxy object. Usually frameworks create this proxy object when client request for real object.

4. Proxy design pattern example

In given example, we have a `RealObject` which client need to access to do something. It will ask the framework to provide an instance of `RealObject`. But as the access to this object needs to be guarded, framework returns the reference to `RealObjectProxy`.

Any call to proxy object is used for additional requirements and the call is passed to real object.

```
RealObject.java

public interface RealObject
{
  public void doSomething();
}
```

```
RealObjectImpl.java

public class RealObjectImpl implements RealObject {

  @Override
  public void doSomething() {
    System.out.println("Performing work in real object");
  }

}
```

```
RealObjectProxy.java

public class RealObjectProxy extends RealObjectImpl
{
  @Override
  public void doSomething()
  {
    //Perform additional logic and security
```

```
    //Even we can block the operation execution
    System.out.println("Delegating work on real object");
    super.doSomething();
  }
}
```

```
Client.java

public class Client
{
  public static void main(String[] args)
  {
    RealObject proxy = new RealObjectProxy();
    proxy.doSomething();
  }
}
```

Program output.

```
Console

Delegating work on real object
Performing work in real object
```

5. FAQs

5.1. what are different types of proxies

Proxies are generally divided into four types –

1. **Remote proxy** – represent a remotely lactated object. To talk with remote objects, the client need to do additional work on communication over network. A proxy object does this communication on behalf of original object and client focuses on real talk to do.
2. **Virtual proxy** – delay the creation and initialization of expensive objects until needed, where the objects are created on demand. Hibernate created proxy entities are example of virtual proxies.
3. **Protection proxy** – help to implement security over original object. They may check for access rights before method invocations and allow or deny access based on the conclusion.
4. **Smart Proxy** – performs additional housekeeping work when an object is accessed by a client. An example can be to check if the real object is locked before it is accessed to ensure that no other object can change it.

5.2. Proxy pattern vs decorator pattern

The primary difference between both patterns are responsibilities they bear. Decorators focus on adding responsibilities, but proxies focus on controlling the access to an object.

Drop me your questions related to **proxy pattern** in comments.

Happy Learning !!

References:

Image Credit – Wikipedia

Recommended Reading:

1. Decorator Design Pattern in Java
2. Adapter Design Pattern in Java
3. Bridge Design Pattern
4. Composite Design Pattern
5. Facade Design Pattern
6. Flyweight Design Pattern
7. Prototype design pattern in Java
8. Strategy Design Pattern
9. Observer Design Pattern
0. Mediator Design Pattern

Subscribe

* We do not spam !!

4 thoughts on “Proxy Design Pattern”

Goutam

February 14, 2020 at 12:50 am

Hi there, i wanted your help to understand how can i use these GOF patterns to solve this problem

Suppose i have an Interface called “Adapter” which has a method Execute that takes an input and executes to return an output. I can have various implementations of this interface to simulate a Database Adapter, Webservice Adapter, REST Adapter, Messaging Adapter etc. All these implementations will have certain common independent characteristics – like Logging the input and output (i don’t want to have them implemented in each implementation).

Also i have another Interface called “Controller” – this one has a method Execute . The responsibility of this component is to call 1..N Adapters defined above. Something like a orchestrator based on individual scenarios. But also need to have common functionality like security/access checks that must be executed automatically in this method.

Such a way, i can have a plugin Receiver (REST/WS/MQ) that receives the input and dynamically calls my Controller implementation which checks cross cutting concerns (security/access) and invokes my implementation of “Controller.Execute”. And in my implementation of Controller.execute i call the dynamic references of the Adapter implementations to invoke the Database/Webservice/REST etc to perform the transaction/operation.

Basically Receiver->Calling something 1-> My Controller-> Calling something 2-> 1..N Adapter implementations.

What sort of mix of patterns do you think i should use here? Any suggestions?

Reply

Lokesh Gupta

February 15, 2020 at 11:54 pm

You may use a customized implementation of chain of responsibility.

Reply

Andy

May 15, 2019 at 12:30 pm

@Sanjay Tiwari
I think maybe I can answer you question. Correct me if I'm wrong.
If you have a object need to delay its creation until real demand. You can init it in the proxy when it need to be used.

```
public class DelayInitObject {

    public DelayInitObject() {
        System.out.println("init the delay init object");
    }
}

public class RealObjectImpl implements RealObject {

    DelayInitObject delayInitObject;

    @Override
    public void doSomething() {
        System.out.println("do something in real object impl");
    }

}

public class RealObjectProxy extends RealObjectImpl {

    @Override
    public void doSomething() {
        System.out.println("delegate in the proxy and call the real method");
        if (super.delayInitObject==null){
            System.out.println("delayInitObject in real object is null, init one");
            super.delayInitObject = new DelayInitObject();
        }
        super.doSomething();
    }

    public static void main(String[] args) {
        RealObjectProxy realObjectProxy = new RealObjectProxy();
        realObjectProxy.doSomething();
        realObjectProxy.doSomething();
    }
}
```

Reply

Sanjay Tiwari

March 30, 2019 at 1:59 pm

How this pattern delaying object creation. As you have extended the real object class from proxy. Creation of proxy object will call super class constructor as well..

Reply

Leave a Comment

Name *

Email *

Website

Add me to your newsletter and keep me updated whenever you publish new blog posts

Post Comment

Search ...

- Contact Us
- Privacy policy
- Advertise
- Guest Posts

Blogs

REST API Tutorial