PROXY PATTERN

Group #11

Creational **Behavioral**

Structural

Concerned with how classes and objects can be **composed**, to form larger structures.

- No rewriting
- No customizing again
- Reusability
- Robust functionality

Proxy Pattern

- **Substitute** or **placeholder** for another object.

- Controls access to the original object.

- Perform something either before or after.

Motivation

Why would you want to control access to an object?

- Object consumes a vast amount of system **resources**.
- You need the object from time to time, but **not always**.

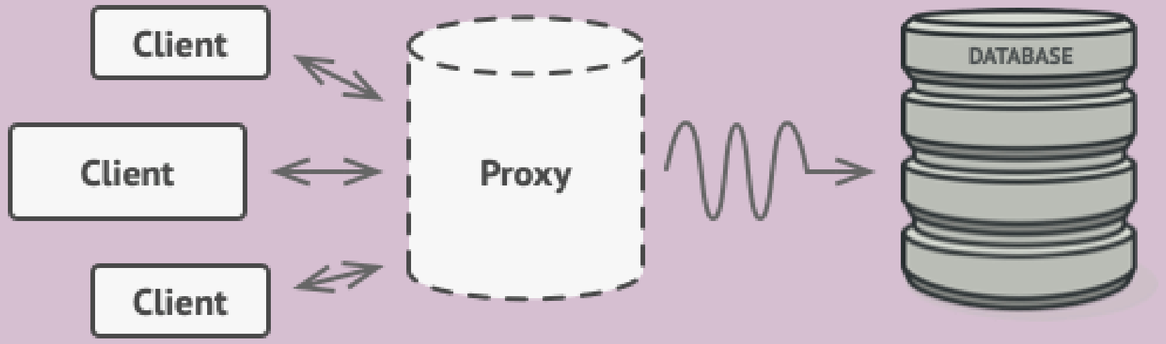
- Try lazy initialization?
- Causes code duplication.

Proxy Pattern comes to the rescue

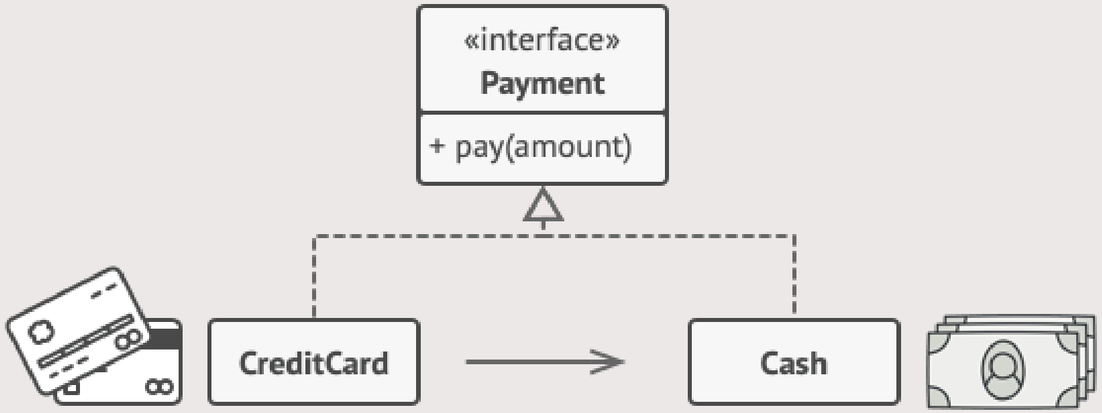
Motivation

- New proxy class with the **same interface** as an original service object.
- **Pass** the proxy object **to** all of the original object's **clients**.

- What's the **benefit**?
- If you need to execute something either **before** or **after** the primary logic of the class, the proxy lets you do this **without changing** that class.



Real-World Analogy

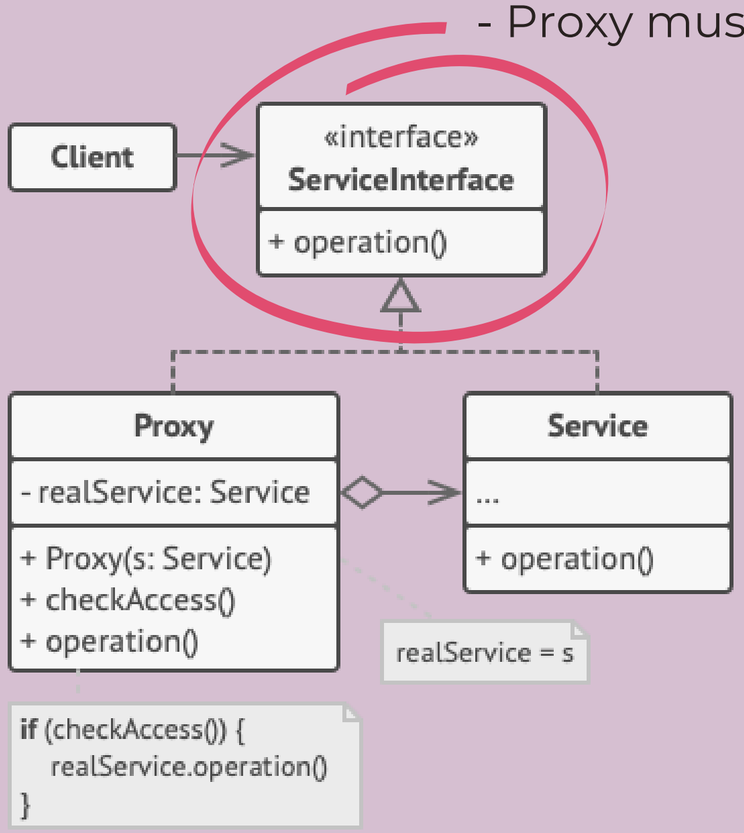


- A credit card is a proxy for a bank account,
- Bank account is a proxy for a bundle of cash.

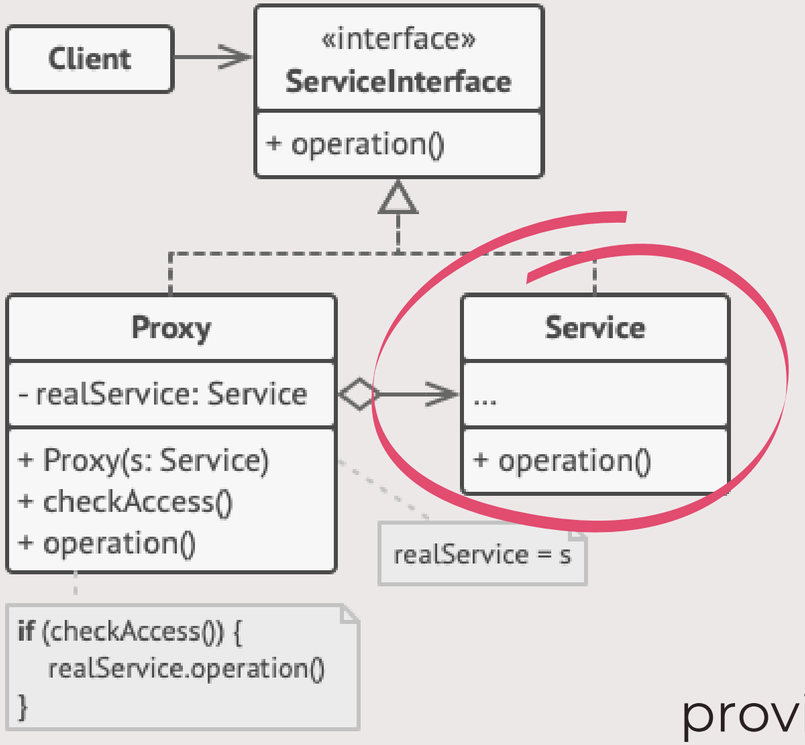
- No need to carry loads of cash around.
- Without the risk of losing the deposit or getting robbed on the way to the bank.

Structure

- Declares the interface of the Service.
- Proxy must follow this interface.

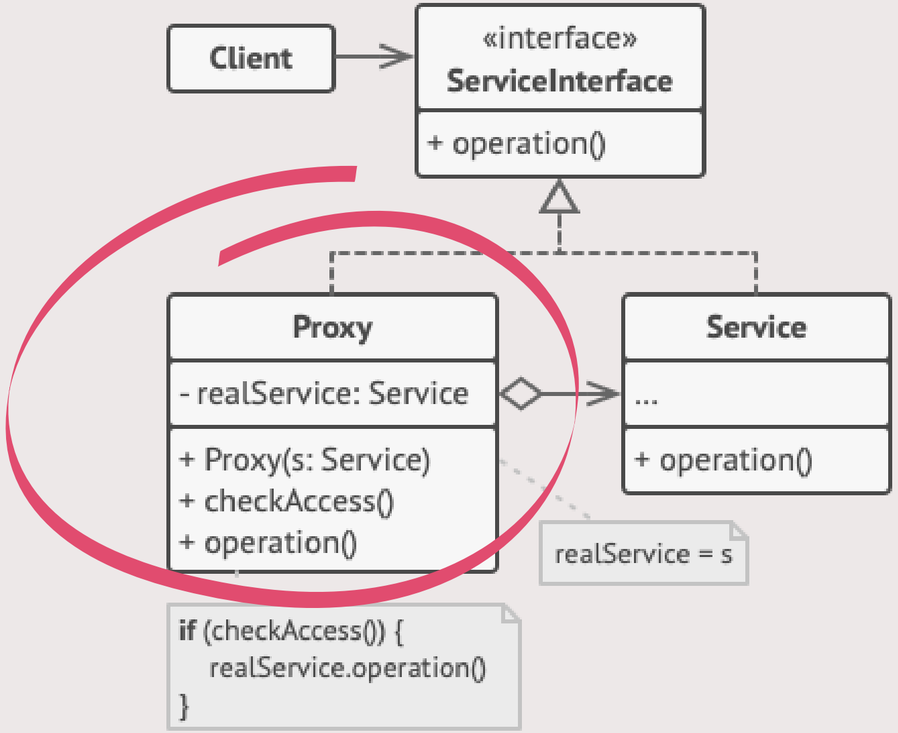


Structure



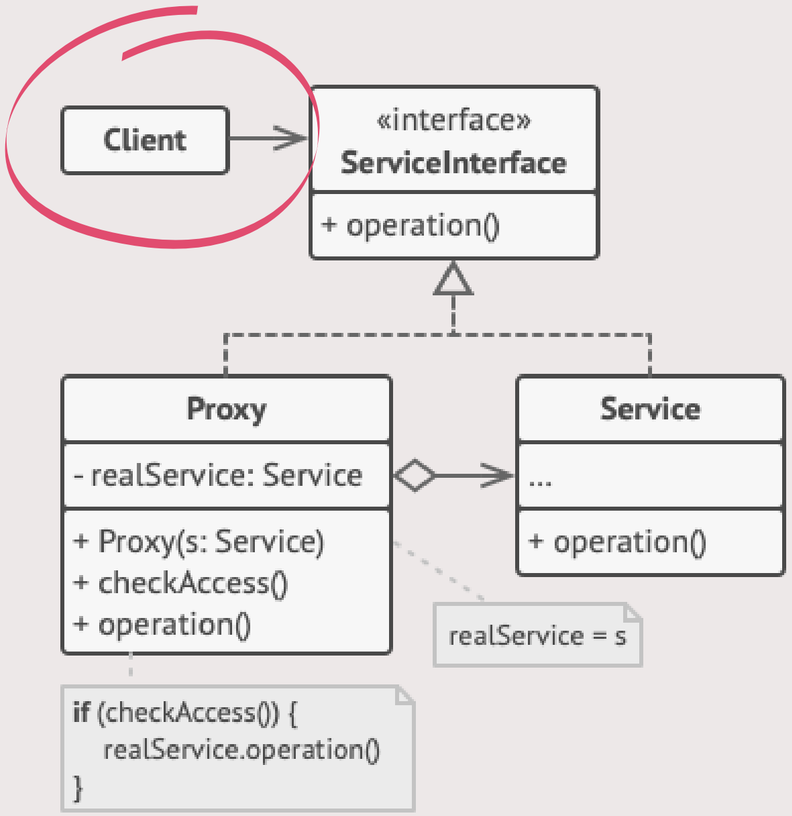
The Service is a class that provides some useful business logic.

Structure

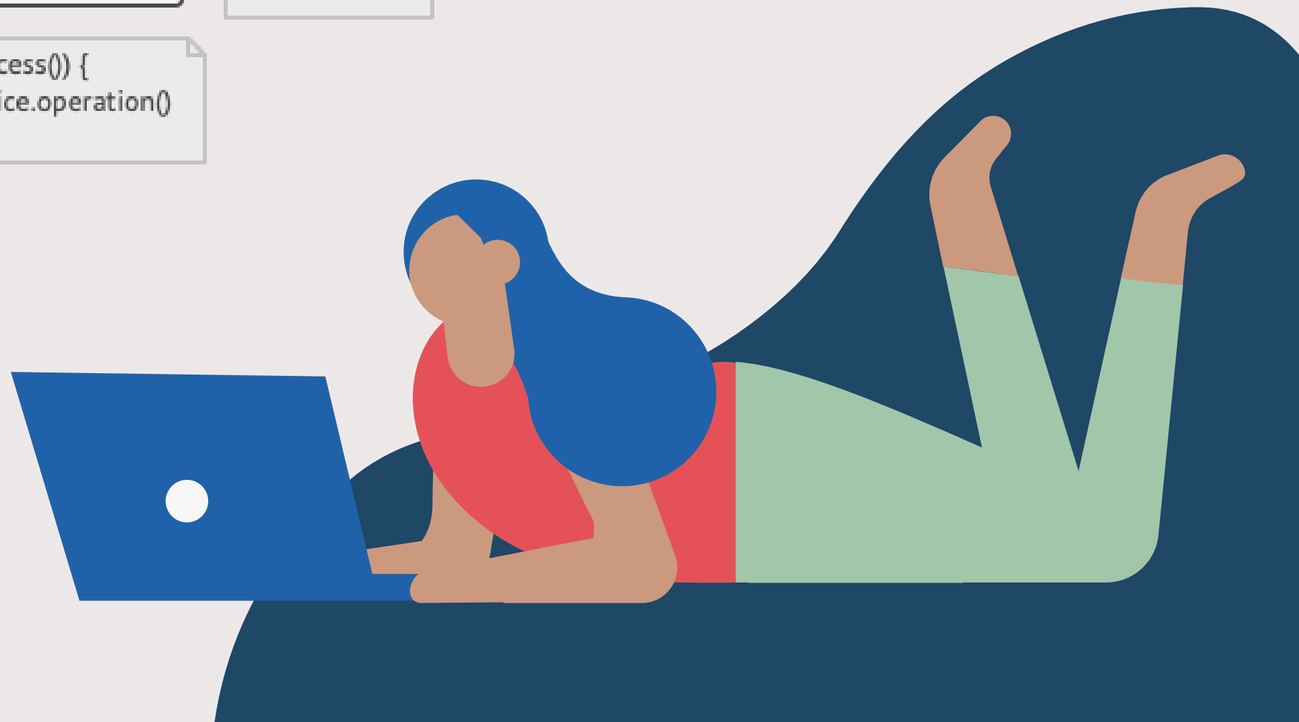


- Points to a service object.
- After processing, it passes the request to the service object.
- Usually, proxies manage the full lifecycle of their service objects.

Structure



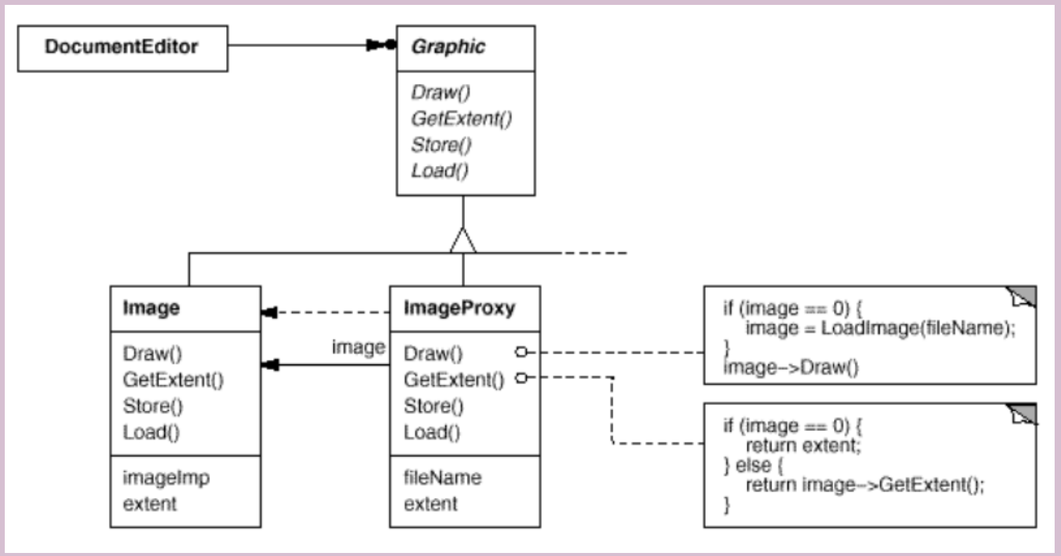
- Works with both services and proxies via the same interface.



Virtual Proxy

Lazy initialization

- **Heavyweight service** object that wastes system resources.
- You only need it from **time to time**.
- Delay the object's initialization to a time when it's really needed.



Protection Proxy

Access control

- **Only** specific clients can use the service object.

- The proxy can pass the request to the service object **only if the client's credentials match some criteria.**

Remote Proxy

Local execution of a remote service

- Service object is located on a remote server.

- Proxy handles all of the nasty details of working with the network.

Logging Proxy

Logging requests

- History of requests

- The proxy can log each request before passing it to the service.

Caching Proxy

Caching request results

- Cache results of client requests.

- Especially if results are quite large.

- Caching for recurring requests that always yield the same results.

Smart Reference

- **Dismiss** a heavyweight object if **no clients use it**.
- The proxy keep track of clients.
- From time to time, the proxy may go over the clients and check whether they are still active.
- If the client list gets empty, proxy **frees the underlying system resources**.
- The proxy tracks **modifications**.

- Controlling the service object **without client's knowledge**.
- **Managing the lifecycle of the service object** when clients don’t care about it.
- The proxy works even if the service **object isn’t ready or is not available**.
- Open/Closed Principle

ADVANTAGES

- Probably more **complicated** code.
- The **response** from the service might get **delayed**.

DISADVANTAGES

Related Patterns

Adapter

- **Different** interface to the object it adapts.
- **Protection proxy** might refuse to perform an operation that the subject will perform, so its interface may be effectively **a subset of the subject's**.

Related Patterns

Decorator

- Decorators can have **similar implementations**.
- But decorators have a **different purpose**: Adds one or more responsibilities to an object.
- A protection proxy might be implemented exactly like a decorator.
- A remote proxy will not contain a direct reference to its real subject but only an indirect reference.
- A virtual proxy will start off with an indirect reference such as a file name but will eventually obtain and use a direct reference.

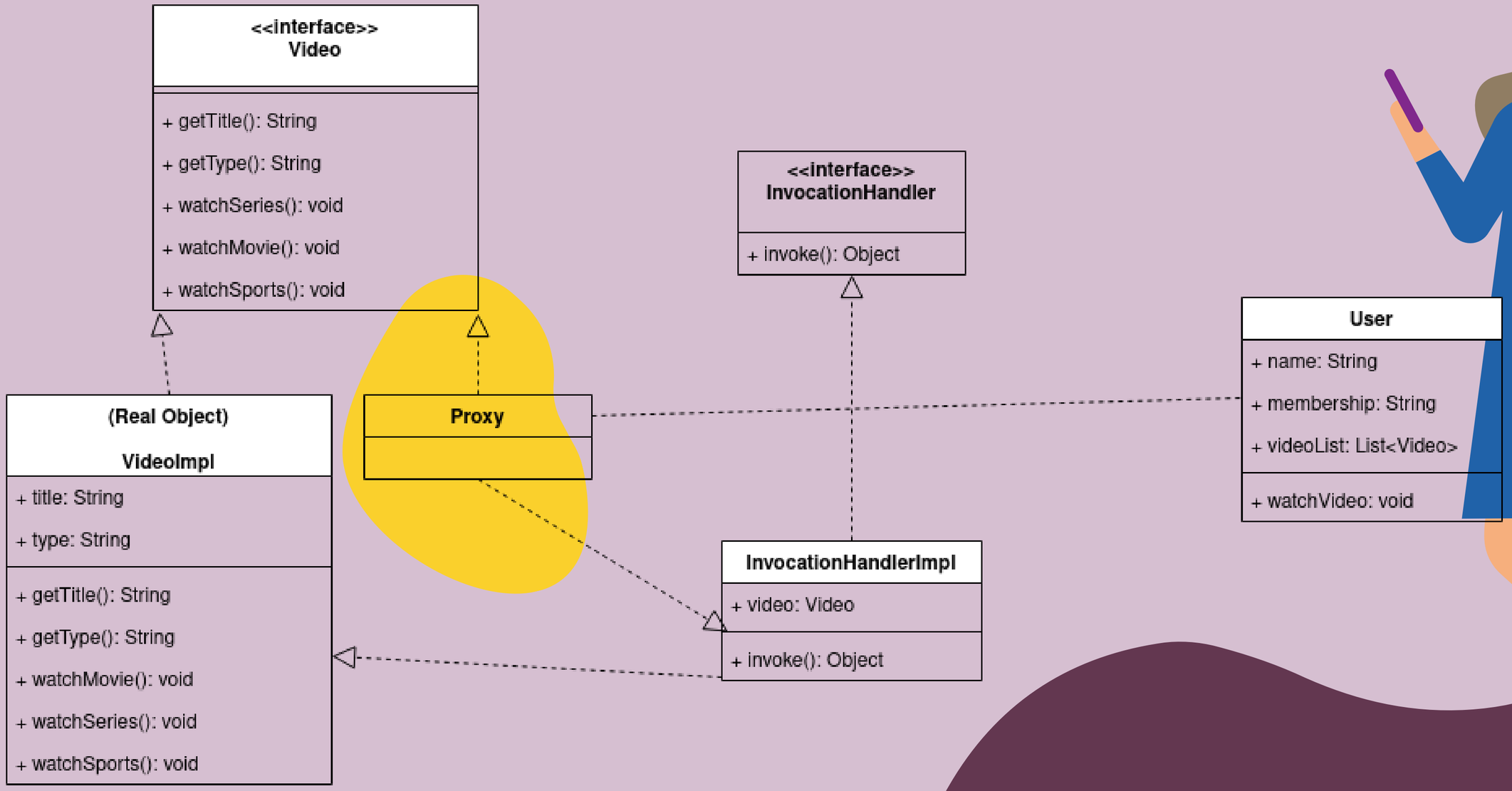
NETFILIKS

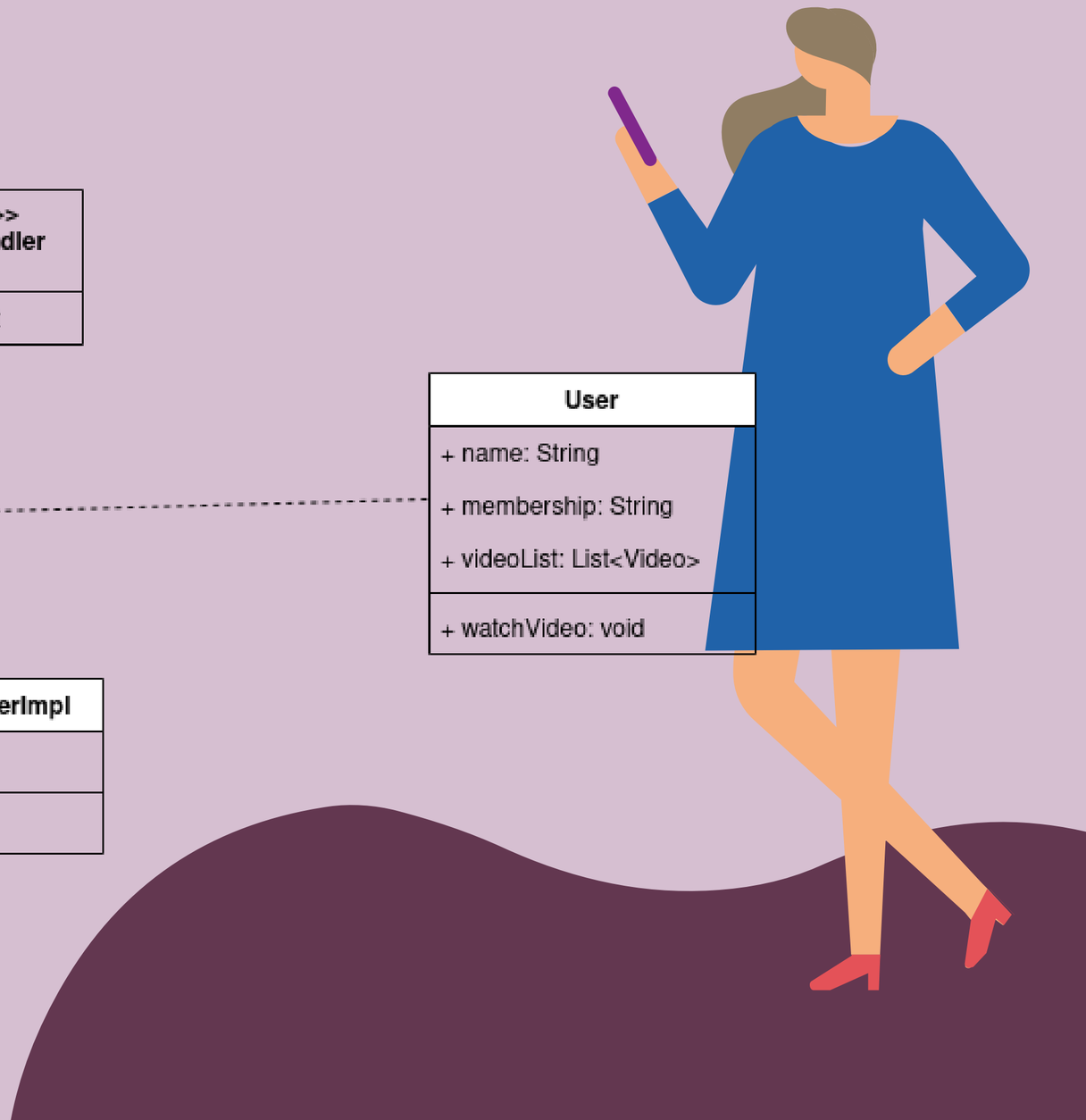
A membership-based video streaming platform.

The content user can view is limited according to the user's membership type.
There are four types of membership:

1. **Basic** : Only series.
2. **Movie** : Series + movies.
3. **Sports** : Series + sports.
4. **Premium** : Series + movies + sports.

In order to limit access, this project implements **Protection Proxy Pattern**.





```
// public class WatchVideo

public static void main(String[] args) {
    new WatchVideo().run(args[0], args[1]);
}
```

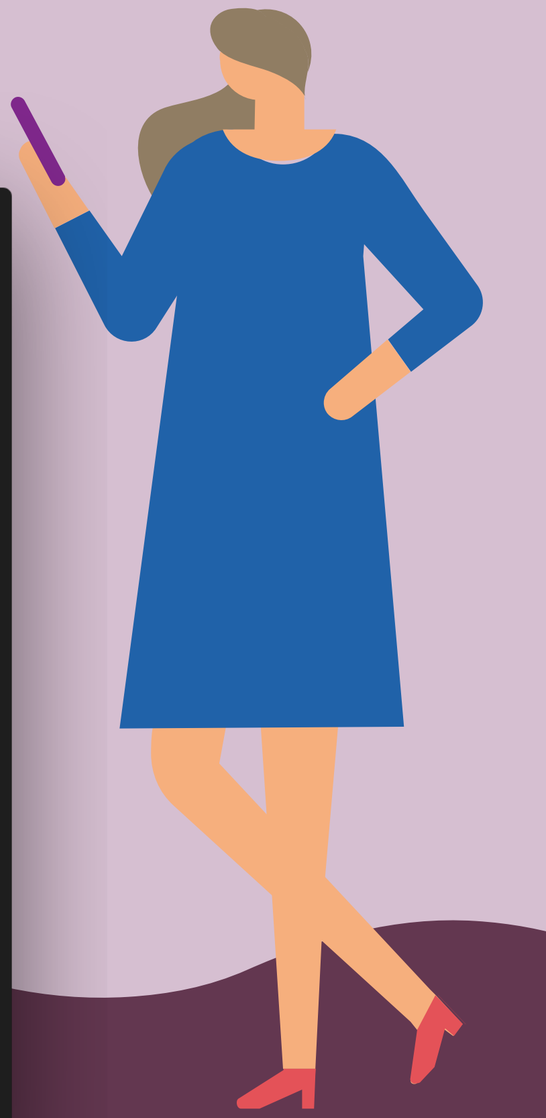
Demo

```java
// public class WatchVideo

public void run(String username, String membership) {
    String selected;
    String prompt = "";
    List<Video> videoList = buildVideoList();

    User user = new User(username, membership, videoList);

    prompt = prompt.concat("\nHi "+ user.name +". Here is our video library:\n");
    for (Video video : videoList) {
        prompt = prompt.concat("- " + video.getTitle() + " : " + video.getType()+"\n");
    }
    prompt = prompt.concat("Write <<Log out>> to log out.\n");
    prompt = prompt.concat("Write the title: ");

    Scanner scanner = new Scanner(System.in);

    System.out.print(prompt);
    while (!((selected = scanner.nextLine()).equals("Log out"))) {
        try {
            user.watchVideo(selected);
        } catch (Exception e) {
            System.out.println("\n------->Access denied for "+user.name+".
                                \n------->(Your membership is "+user.membership+".
                                           You must upgrade or change your membership.)");
        }
        System.out.print(prompt);
    }

    scanner.close();
}
```

Demo run()

Demo
buildVideoList()

```java
// public class WatchVideo

public List<Video> buildVideoList() {

    List<Video> videoList = new ArrayList<Video>();

    Video houseMD = new VideoImpl("House MD", "Series");
    InvocationHandler houseMDHandler = new InvocationHandlerImpl(houseMD);
    Video houseMDProxy = (Video) Proxy.newProxyInstance(Video.class.getClassLoader(), new Class[]{Video.class}, houseMDHandler);

    Video bladeRunner = new VideoImpl("Blade Runner", "Movie");
    InvocationHandler bladeRunnerHandler = new InvocationHandlerImpl(bladeRunner);
    Video bladeRunnerProxy = (Video) Proxy.newProxyInstance(Video.class.getClassLoader(), new Class[]{Video.class}, bladeRunnerHandler);

    Video footballGame = new VideoImpl("Football Game", "Sports");
    InvocationHandler footballGameHandler = new InvocationHandlerImpl(footballGame);
    Video footballGameProxy = (Video) Proxy.newProxyInstance(Video.class.getClassLoader(), new Class[]{Video.class}, footballGameHandler);

    videoList.add(houseMDProxy);
    videoList.add(bladeRunnerProxy);
    videoList.add(footballGameProxy)

    return videoList;
}
```



```java
public class User {

    public String name;
    public String membership;
    public List<Video> videoList;

    public User(String name, String membership, List<Video> videoList) {
        this.name = name;
        this.membership = membership;
        this.videoList = videoList;
    }

    public void watchVideo(String videoName) {

        for (Video video : this.videoList) {
            String videoTitle = video.getTitle();

            if (videoTitle.equals(videoName)) {
                String videoType = video.getType();

                switch(videoType) {
                    case "Series":
                        video.watchSeries(this.membership);
                        break;
                    case "Movie":
                        video.watchMovie(this.membership);
                        break;
                    case "Sports":
                        video.watchSports(this.membership);
                        break;
                }
            }
        }
    }

}
```

Client

```java
public class InvocationHandlerImpl implements InvocationHandler{

    private Video video;

    public InvocationHandlerImpl(Video v) {
        video = v;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        try {

            String methodName = method.getName();

            // Only Movie and Premium Members can watch movies
            if (methodName.equals("watchMovie") && (args[0].equals("Basic") || args[0].equals("Sports"))) {
                throw new IllegalAccessException();
            }
            // Only Sports and Premium members can watch sports
            else if (methodName.equals("watchSports") && (args[0].equals("Basic") || args[0].equals("Movie"))) {
                throw new IllegalAccessException();
            }
            else {
                return method.invoke(video, args);
            }

        } catch (InvocationTargetException e) {
            e.printStackTrace();
        }

        return null;
    }
}
```

Subject

```java
public interface Video {

    String getTitle();
    String getType();

    void watchSeries(String membership);
    void watchMovie(String membership);
    void watchSports(String membership);

}
```

```java
public class VideoImpl implements Video {

    public String title;
    public String type;

    public VideoImpl(String title, String type) {
        this.title = title;
        this.type = type;
    }

    public void watchSeries(String membership) {
        System.out.println("\n ------->" + this.title+" is loading...");
    }

    public void watchMovie(String membership) {
        System.out.println("\n ------->" + this.title+" is loading...");
    }

    public void watchSports(String membership) {
        System.out.println("\n ------->" + this.title+" is loading...");
    }

    public String getTitle() {
        return this.title;
    }

    public String getType() {
        return this.type;
    }

}
```

Subject Impl