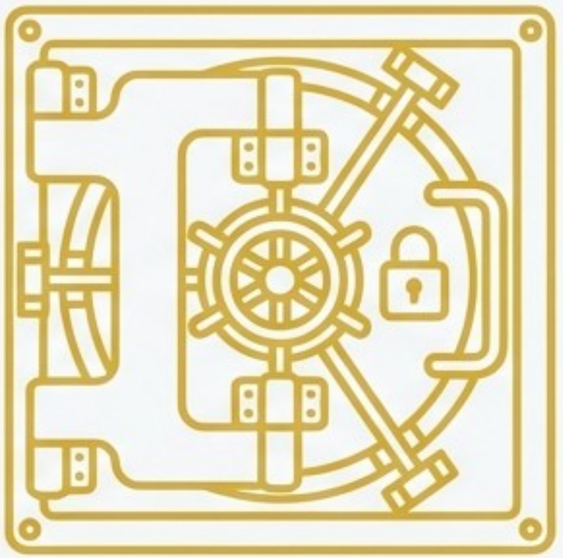
The Proxy Pattern

Gaining Controlled Access to Complex Objects

Let’s start with something familiar: **your credit card.**

You don’t always carry stacks of cash. It’s heavy, risky, and inefficient.

Instead, you use a card. It’s a lightweight stand-in that provides access to your money.



The bank holds the real money, but the card handles the transaction, checks your balance, and logs the payment.

Key Takeaway: You still pay, but you do it indirectly through a secure and convenient placeholder.

In software, the Proxy works just like a credit card.

The Card → The Proxy

A stand-in object that the client interacts with. It looks and feels like the real thing.

The Bank → The Real Object

The actual object that does the work. It might be resource-heavy, remote, or sensitive.

The Transaction → The Method Call

The client calls a method on the Proxy, which then decides how and when to involve the Real Object.

The Problem: Direct access can be slow and inefficient.

Imagine loading a large video file directly.

The Old Way

```
// Client has to handle the complexity directly.
var video = new RealVideo("big_movie.mp4");

video.Play();
```

❌ **No Lazy Loading:** The entire heavy video is loaded into memory immediately, even if the user never clicks play.

❌ **No Access Control:** Any part of the system can load the video, with no restrictions or permissions checks.

❌ **No Logging:** There’s no easy way to track when or how often the video is accessed.

❌ **Tight Coupling:** The client is directly tied to the concrete, heavy `RealVideo` class.

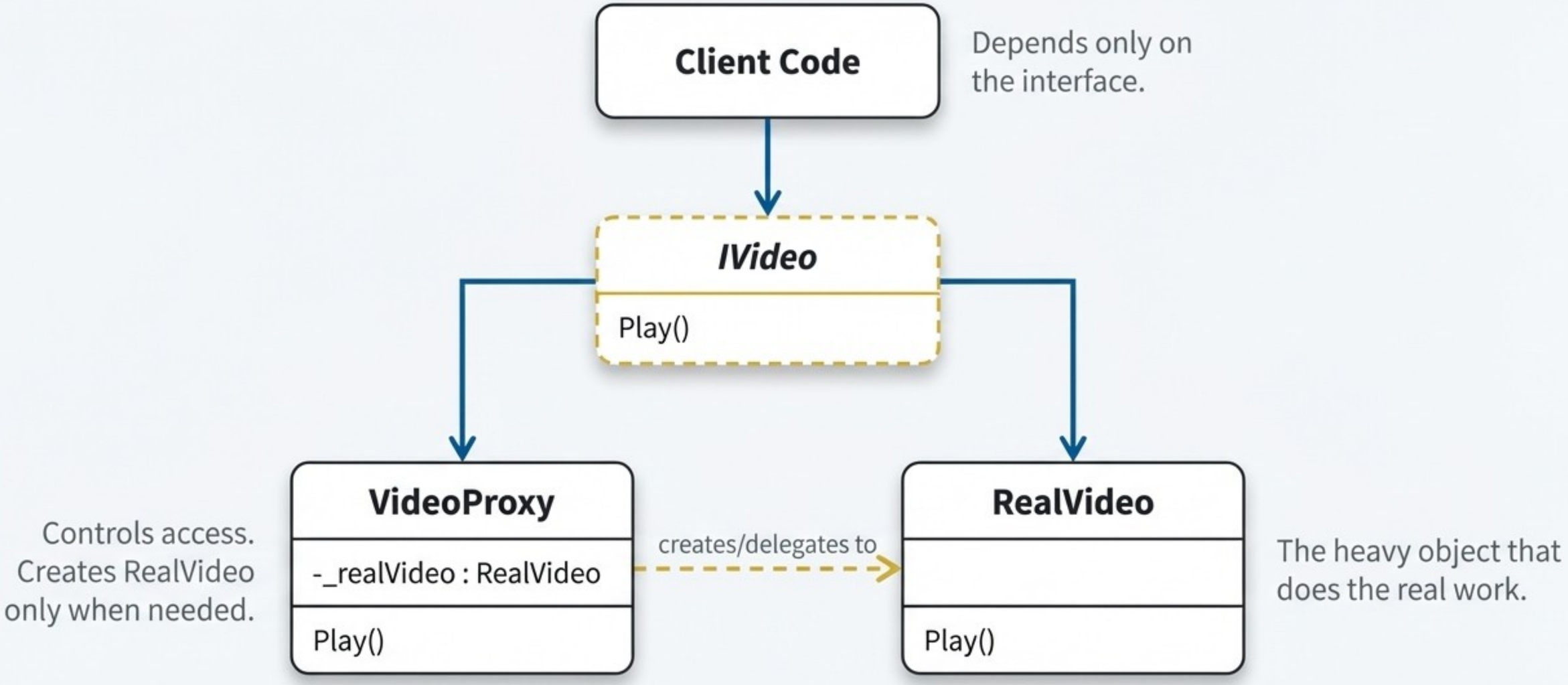
The Solution: The Proxy acts as a smart stand-in.

Create an object that *looks like the real one* but adds a layer of control before delegating the call.

- ✅ **Lazy Load**: Postpone creation of the real object until it's absolutely necessary.
- ✅ **Secure**: Check permissions and validate access before proceeding.
- ✅ **Cache:** Store results of expensive operations and return them quickly.
- ✅ **Log**: Transparently record all requests made to the object.
- ✅ **Remote**: Hide the complexity of communicating with an object on another server.

Key Principle: The client continues to use the same interface, completely unaware of the complex work happening behind the scenes.

How It's Structured: The Client, The Interface, and The Two Objects



Step-by-Step Code: The Shared Interface and The Real Object

Step 1 — The Common Interface

Both the real object and the proxy will share a common interface, ensuring the client doesn't need to know which one it's using.

```
public interface IVideo
{

    void Play();
}
```

Step 2 — The Real (Heavy) Object

This class performs the resource-intensive work. Notice that the expensive operation happens immediately upon creation.

```
public class RealVideo : IVideo
{
  private readonly string _fileName;
  public RealVideo(string fileName)
  {
    _fileName = fileName;
    // This is the expensive operation!
    Console.WriteLine("Loading video from disk...");
  }
  public void Play()
    => Console.WriteLine($"Playing {_fileName}");
}
```

Heavy work happens here, in the constructor.

Step-by-Step Code: The Proxy Delays Creation

Step 3 — The Proxy

The Proxy implements the same interface but holds a reference to the real object. It only creates the `RealVideo` instance the first time `Play()` is called.

```
public class VideoProxy : IVideo
{
    private readonly string _fileName;
    private RealVideo _video; // Initially null!

    public VideoProxy(string fileName)
    {
        _fileName = fileName;
    }

    public void Play()
    {
        // The magic happens here!
        if (_video == null)
        {
            _video = new RealVideo(_fileName);
        }
        _video.Play();
    }
}
```

Reference to the real object starts as `null`.

Lazy Instantiation: The `RealVideo` object is only created *inside* the method call, and only if it doesn't already exist.

The Result: Better Performance Through Lazy Loading

Step 4 — The Client Code

The client code is simple. It creates the proxy and calls `Play()`, unaware of the lazy loading.

```
// Create the proxy. The real video is NOT loaded yet.
IVideo video = new VideoProxy("movie.mp4");
Console.WriteLine("Proxy has been created.");

// Now, the user clicks play.
video.Play(); // The RealVideo object is created now!
```

Console Output

Observe the order of operations. The expensive 'Loading' step only occurs when `Play()` is called.

```
Proxy has been created.
Loading video from disk...
Playing movie.mp4
```

✓ Same interface

✓ Lazy load

✓ Better performance

The Proxy is more than just lazy loading.

It's a versatile pattern with many enterprise use cases.

🛡️ Security Proxy

Checks a client's permissions before allowing access to the real object.

Use Case: Admin-only dashboards, secure APIs.

☁️ Remote Proxy

Hides the network communication needed to interact with an object on a different server.

Use Case: Microservices, gRPC, REST API clients.

💾 Caching Proxy

Stores the results of expensive calls. If the same request comes in again, it returns the cached result instead of calling the real object.

Use Case: Product catalogue APIs, weather services.

✍️ Logging Proxy

Intercepts calls to the real object to log them for auditing or monitoring purposes.

Use Case: Transparently tracking API requests.

⏳ Virtual Proxy

The pattern we just saw. Manages the lifecycle of a resource-heavy object.

Use Case: Loading large images, database connections.

When should you use the Proxy pattern?

If you need to introduce an intermediate layer to manage *how* or *when* a real object is accessed, the Proxy pattern is an excellent fit.

- ✅ **Access Control**: You need to enforce permissions. Only certain users or functions should be allowed to call the object's methods.
- ✅ **Lazy Loading:** The object is expensive to create (memory, I/O, network) and should only be instantiated when truly needed.
- ✅ **Logging / Monitoring**: You want to track all calls to an object without modifying its source code.
- ✅ **Caching:** You want to reuse the results of expensive operations to improve performance.
- ✅ **Remote Communication**: Your client code needs to interact with an object that lives on another server or in a different process.

When should you AVOID the Proxy pattern?

Don’t add layers of abstraction ‘just because’. A Proxy adds complexity, so ensure the benefits outweigh the cost.

- ❌ **Simple & Cheap Access**: If the real object is lightweight and direct access is perfectly fine, a proxy is unnecessary overhead.
- ❌ **Unnecessary Complexity**: For simple applications, adding interfaces and proxy classes can make the code harder to understand and maintain.
- ❌ **Performance Overhead**: In high-performance, low-latency systems, the extra layer of indirection from a proxy, however small, might be unacceptable.
- ❌ **Transparency Isn’t Needed**: If the client *should* be aware of complexities like network state or caching logic, a proxy might hide too much.

The Proxy reinforces core SOLID principles.

S - Single Responsibility Principle (SRP)

The Proxy handles the access logic (caching, security, logging). The Real Object focuses purely on its core business logic.

O - Open/Closed Principle (OCP)

You can introduce new proxy types (e.g., adding a Caching Proxy on top of a Security Proxy) without ever modifying the original `RealVideo` class.

D - Dependency Inversion Principle (DIP)

The client depends on the `IVideo` interface, not on the concrete `RealVideo` or `VideoProxy` classes. This decouples the client from the implementation details.

The Proxy Pattern: At a Glance

Item	Meaning
Problem	We need to control access to an object for reasons like performance, security, or location.
Solution	A Proxy object stands in front of the Real Object, intercepting calls.
Key Idea	The Proxy and Real Object share the same interface, making the substitution transparent to the client.
Core Benefits	Lazy loading, caching, security, logging, and simplifying remote communication.
Common Use Cases	Secure APIs, database connection pools, microservice clients, image loading.
Principles	Adheres to SRP, OCP, and DIP.