The Proxy Pattern

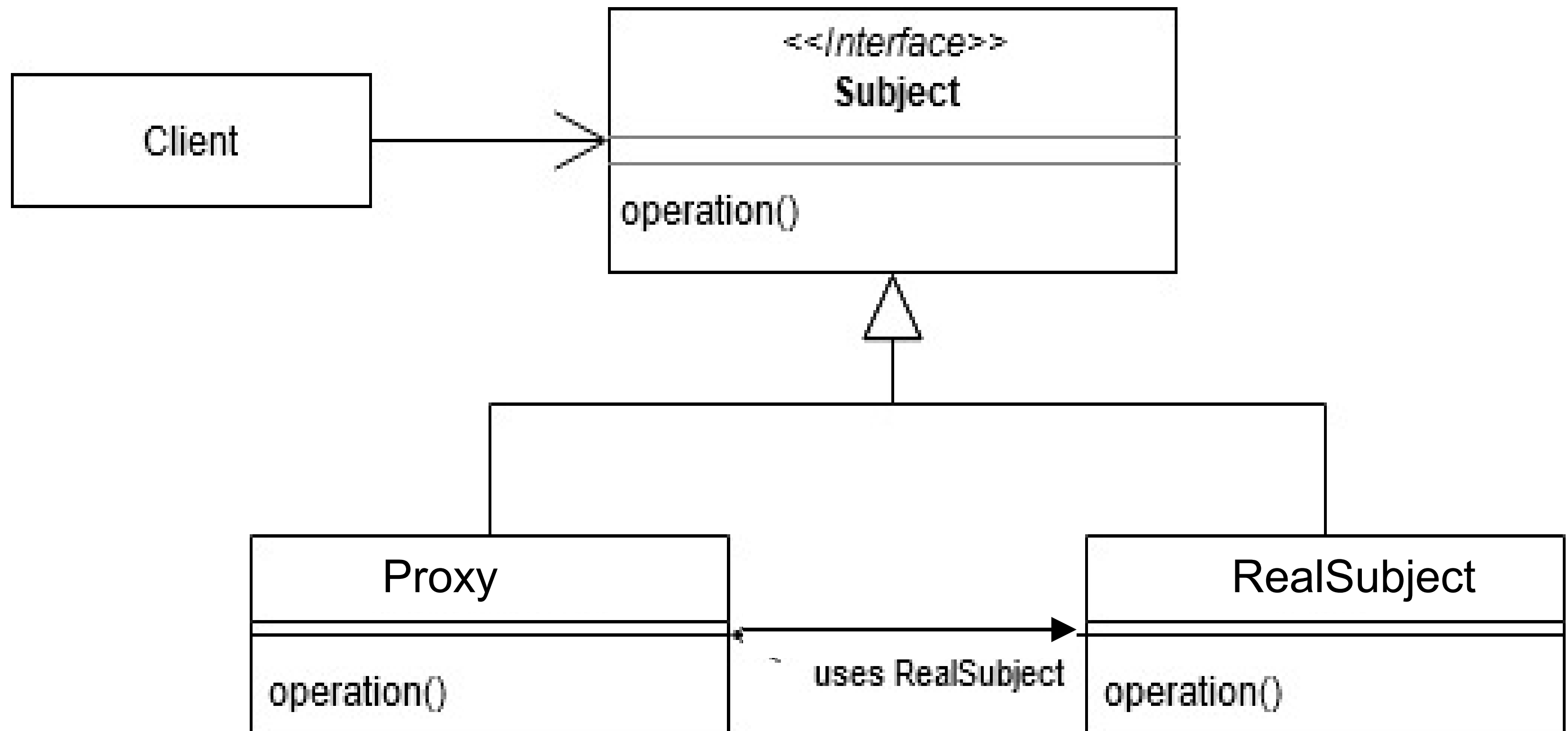
The Java Dynamic Proxy

The Proxy Pattern

- Proxy provides a substitute or placeholder for another object.
- A proxy controls access to the original object, allowing you to perform something either before or after the request gets through to the original object.

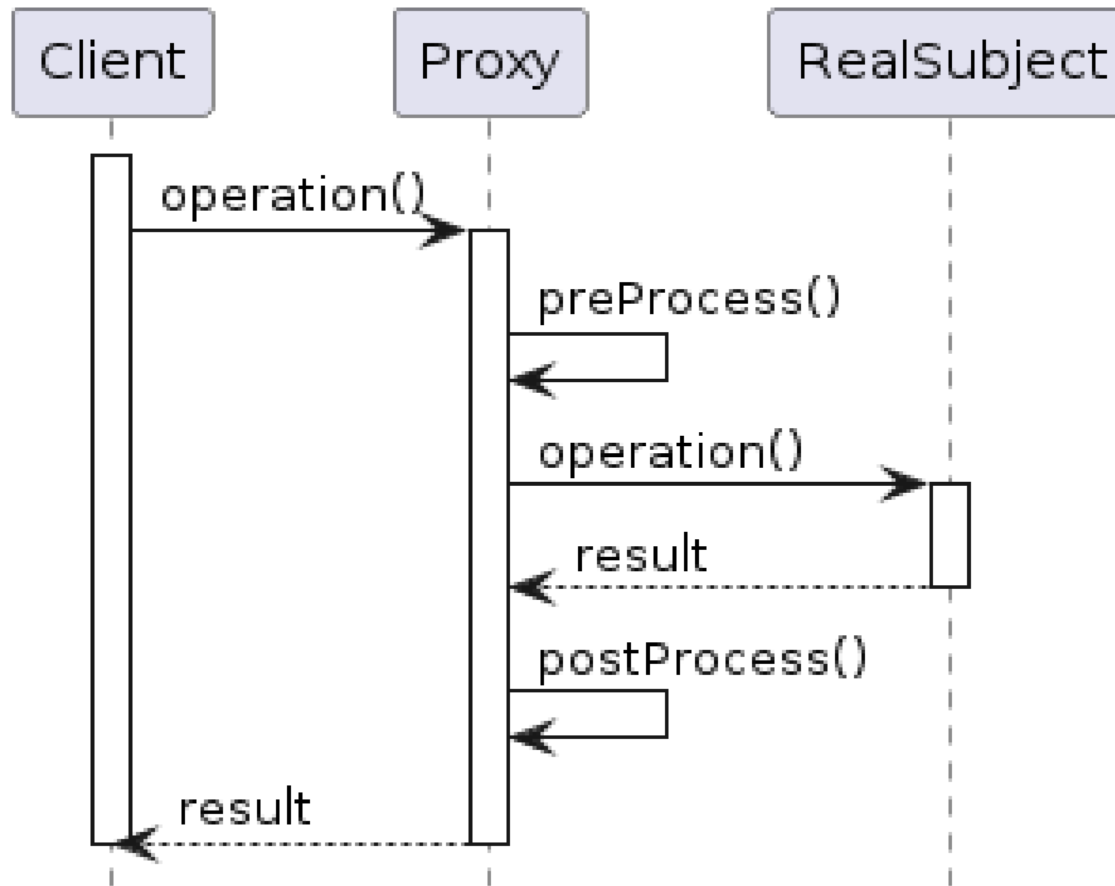
- Use Cases:
  - Controlling access to sensitive objects (security proxy)
  - Lazy initialization (virtual proxy)
  - Acces to remote object (remote proxy)
  - Adding additional behavior such as logging, timing, or caching (decorator-like proxy)

The Proxy Pattern - Structure



The Proxy class must implement the same interface as the RealSubject.
The Client does not know that it interacts with a Proxy instead of RealSubject

The Proxy Pattern - Behaviour



The Client interacts with a Proxy. The Proxy forwards the call to the RealSubject but can add its own pre and post-processings.

Example: A Timing Proxy for IMath

```
public interface IMath {
    int add(int a, int b);
    int mult(int a,int b);
}
```

Abstract Subject

```
public class Math implements IMath{
    public int add(int a, int b) {
        return a+b;
    }
    public int mult(int a, int b){
        return a*b;
    }
}
```

Real Subject

```
public class MathTimingProxy implements IMath{

    private IMath target;

    public MathTimingProxy(IMath target) {
        this.target=target;
    }

    public int add(int a,int b){

        long start = System.nanoTime();
        int result = target.add(a,b);
        long elapsed = System.nanoTime() - start;
        System.out.println("Method add in " + elapsed + " ns");
        return result;
    }

    public int mult(int a,int b){

        long start = System.nanoTime();
        int result = target.mult(a,b);
        long elapsed = System.nanoTime() - start;
        System.out.println("Method mult in " + elapsed + " ns");
        return result;
    }
}
```

```java
public class Main {
    public static void main(String[] args) {

        IMath target = new Math();

        IMath mathTimingProxyInstance = new MathTimingProxy(target);

        mathTimingProxyInstance.add(3,4);
        mathTimingProxyInstance.mult(5,7);

    }
}
```

Limitations of Classical Proxy

- What if we want to apply a Timing Proxy to another interface?
- If you want to apply the same proxy to another interface, you must create a new proxy class for each interface
  - This is tedious and requires recompilation.
- Dynamic Proxy solves this by generating proxy classes at runtime that implement any given interface

Java Dynamic Proxy

Java Reflection

- Java Reflection API gives you introspection and some **limited intercession**, but in general **it does not support creating new types via reflection, cannot modify existing types via reflection**
- **One particular case that is an exception**: Java Reflection API offers support for **creating a new type at runtime via Reflection** in a well-defined **particular context** = **The Dynamic Proxy**
- **The Java Dynamic proxy:**
  - allows to create at runtime a new class that implements a set of interfaces that are given at runtime
  - uses a single InvocationHandler to intercept all method calls
  - eliminates the need for multiple proxy classes in a situation such as the TimingProxy example
  - has been introduced in order to handle java RMI proxies
  - it is used in frameworks such as Spring Data

Java Dynamic Proxy

- Implements a list of interfaces given at runtime.
- Uses a single InvocationHandler to intercept all method calls.
- Key Classes/Interfaces:
  - java.lang.reflect.Proxy
    - Class java.lang.reflect.Proxy acts as a factory to create new classes that implement some given interfaces (dynamic proxy classes), and also their instances (proxy instances)
  - java.lang.reflect.InvocationHandler
    - Each proxy instance has an associated invocation handler object, which implements the interface java.lang.reflect.InvocationHandler.
    - The invocation handler "intercepts" method calls and reroute them or add functionality dynamically.

The InvocationHandler Interface

```
public interface InvocationHandler {
    Object invoke(Object proxy, Method method, Object[] args) throws Throwable;
}
```

The job of an invocation handler is to actually perform the requested method invocation on behalf of a dynamic proxy. He gets a Method object (from the Reflection API) and the objects that are the arguments for the method call. In the simplest case, he can just call Mehod.invoke() or add pre or post processings.

Proxy object – an instance
of the dynamic proxy class
created automatically at runtime

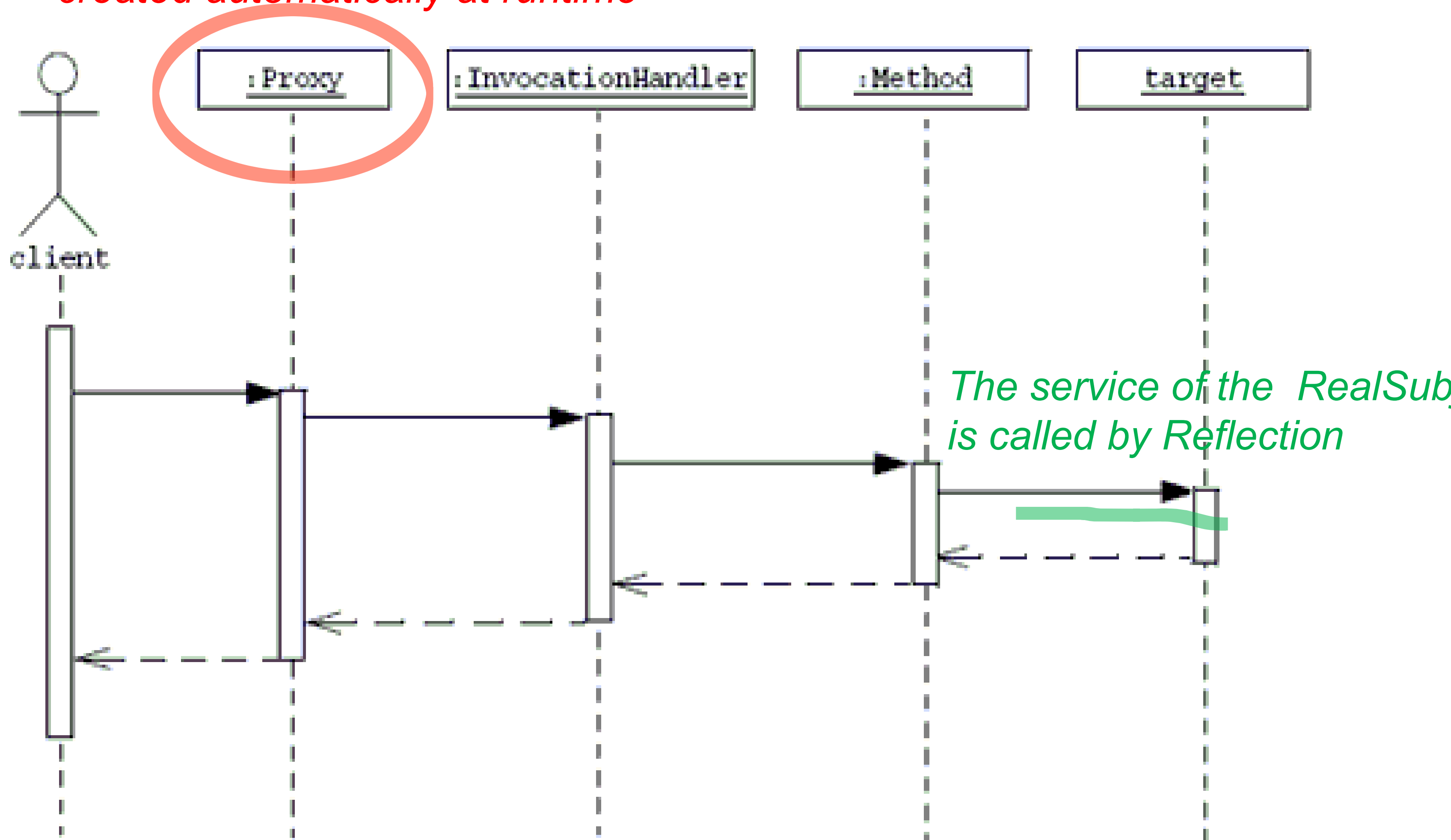


Figure 4.2 Sequence diagram illustrating the actual objects involved in forwarding a method when the invocation handler of the proxy uses the `invoke` method of `Method`.

Example: Create a Dynamic Proxy

- Example: Create Proxy instance for an interface Foo, using MyInvocationHandler as an interceptor

```
InvocationHandler handler = new MyInvocationHandler(...);

Foo f = (Foo) Proxy.newProxyInstance(
                    Foo.class.getClassLoader(), // ClassLoader
                    new Class[] { Foo.class },  // implemented interfaces
                    handler);                   // invocationhandler
```

Example: The Timing Proxy as a Dynamic Proxy

```
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;

public class TimingHandler implements InvocationHandler {

    private Object target;

    public TimingHandler(Object target) {
        this.target = target;
    }

    public Object invoke(Object proxy, Method method, Object[] args)
            throws Throwable {
        long start = System.nanoTime();
        Object result = method.invoke(target, args);
        long elapsed = System.nanoTime() - start;

        System.out.println("Execution of method "+method.getName()+" finished
in "+elapsed +" ns");

        return result;
    }
}
```

Using the Dynamic Timing Proxy

```
import java.lang.reflect.Proxy;

public class Main {
    public static void main(String[] args) {

        IMath target = new Math();

        IMath mathTimingProxyInstance = (IMath) Proxy.newProxyInstance(
                Main.class.getClassLoader(),
                new Class[] { IMath.class },
                new TimingHandler(target));

        System.out.println("type of proxy is " +
                          mathTimingProxyInstance.getClass().getName());

        mathTimingProxyInstance.add(3,4);
        mathTimingProxyInstance.mult(5,7);

    }
}
```

Example Source Code

- MathTimingProxy with Dynamic Proxy

More on Dynamic Proxy

- https://docs.oracle.com/javase/8/docs/technotes/guides/reflection/proxy.html

- Similar class in .NET Reflection: System.Reflection.DispatchProxy
  https://learn.microsoft.com/en-us/dotnet/api/system.reflection.dispatchproxy?view=net-9.0