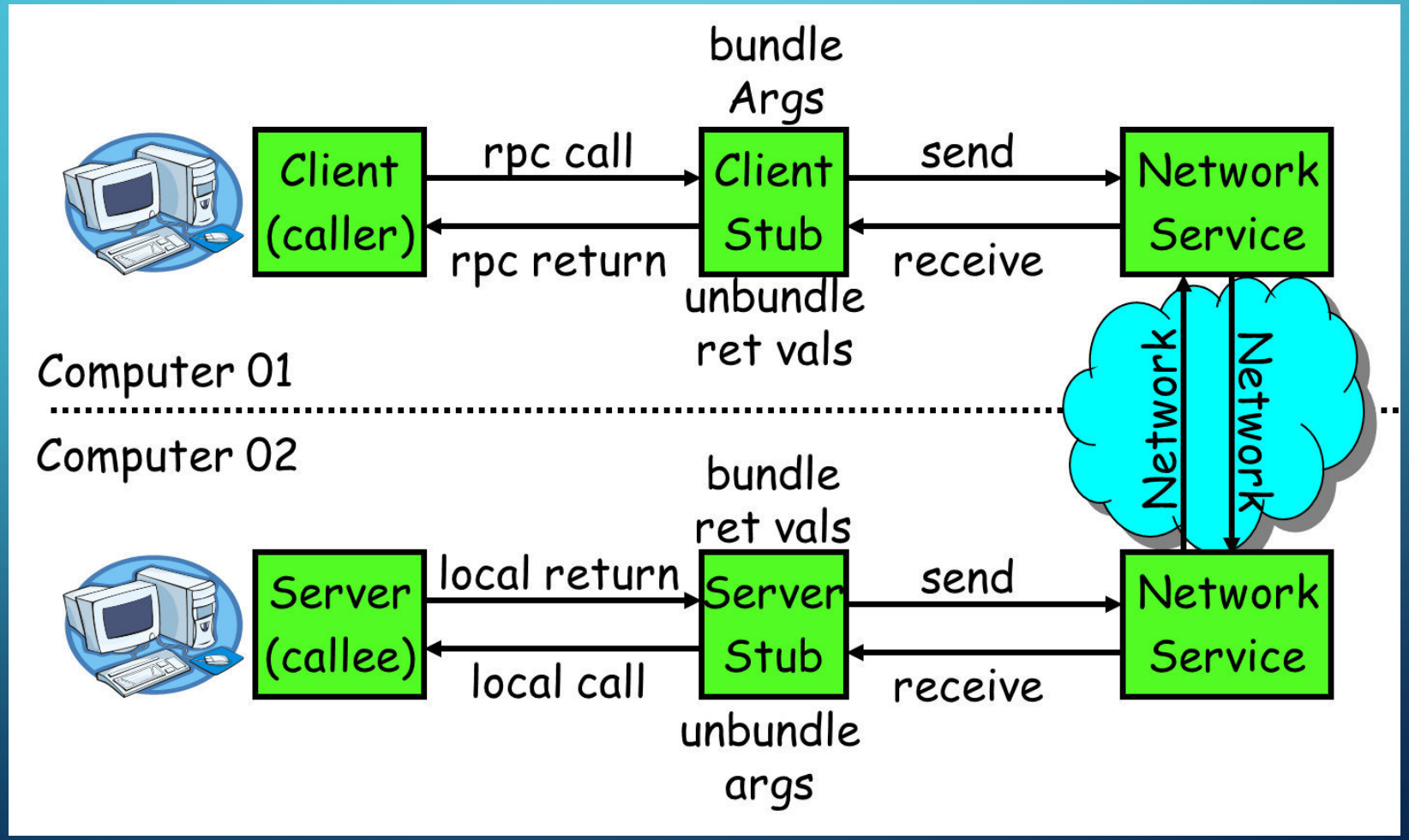
DISTRIBUTED AND CLOUD COMPUTING

LAB4 JAVA DYNAMIC PROXY IN RMI

WORKFLOW OF RPC CALLS



GOOD DESIGN VS. BAD DESIGN

- A user-friendly RPC/RMI framework should be able to hide all low-level implementations from users.

```
public static void main(String[] args) {
    RpcThingInterface stub = RpcService.lookup("service_name");
    float userParam = 123.45;
    int result = stub.userMethod(userParam);
} // good design
```

1. Get access to the remote object by name

2. Invoke the remote method

```
public static void main(String[] args) {
    String remoteIp = RpcService.findIp("service_name");
    int remotePort = RpcService.findPort("service_name");
    RpcClient client = new RpcClient(remoteIp, remotePort);
    float userParam = 123.45;
    String result = client.sendInvokeParam("userMethod", new String[]{Float.toString(userParam)});
    int realResult = Integer.parseInt(result);
} // bad design
```

1. Get the IP address

2. Get the port number

3. Create a client

4. Send the arguments via client and get the return as a string

5. Parse the returned string to get the result

WORKFLOW OF RMI

From the perspective of the client

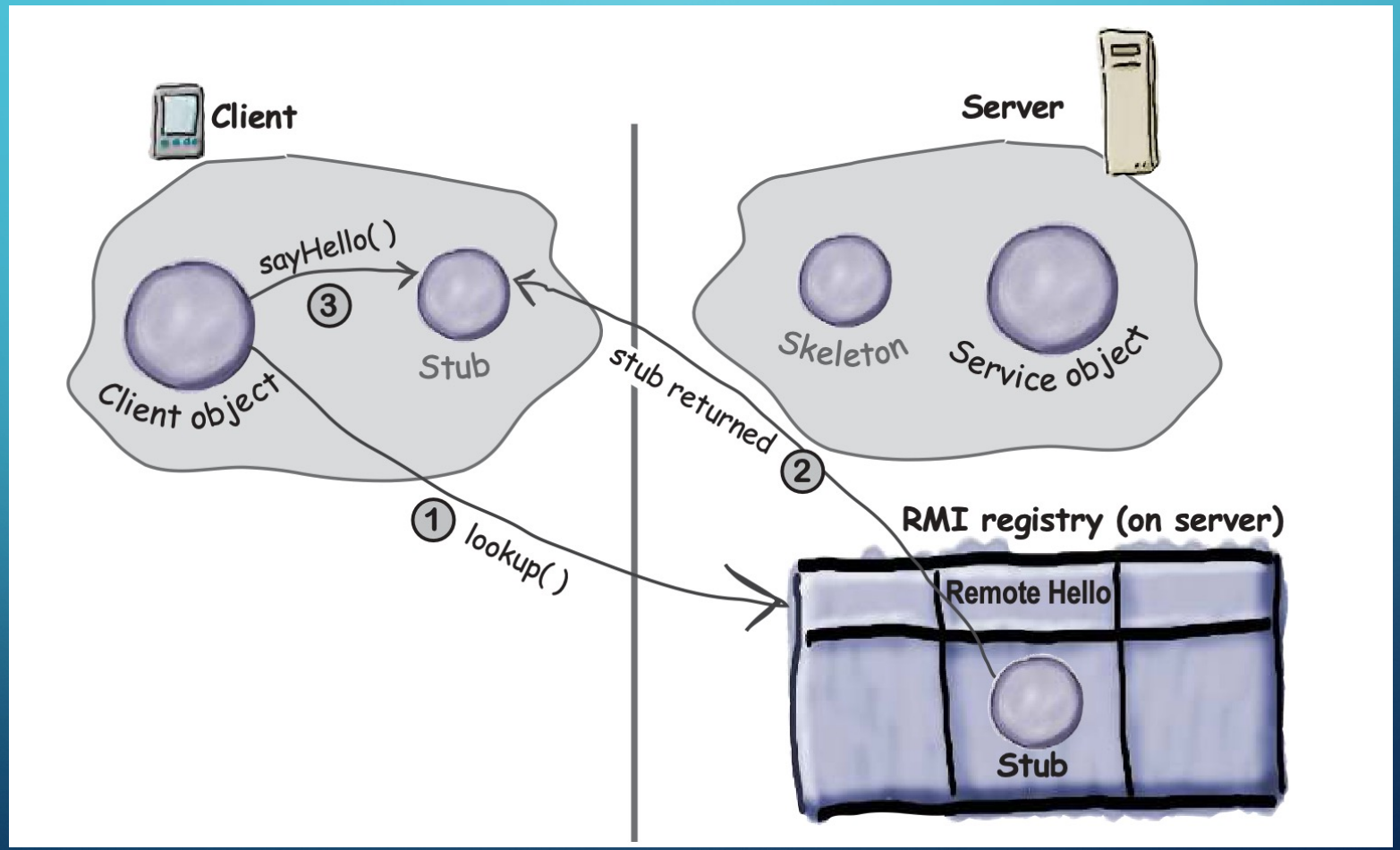
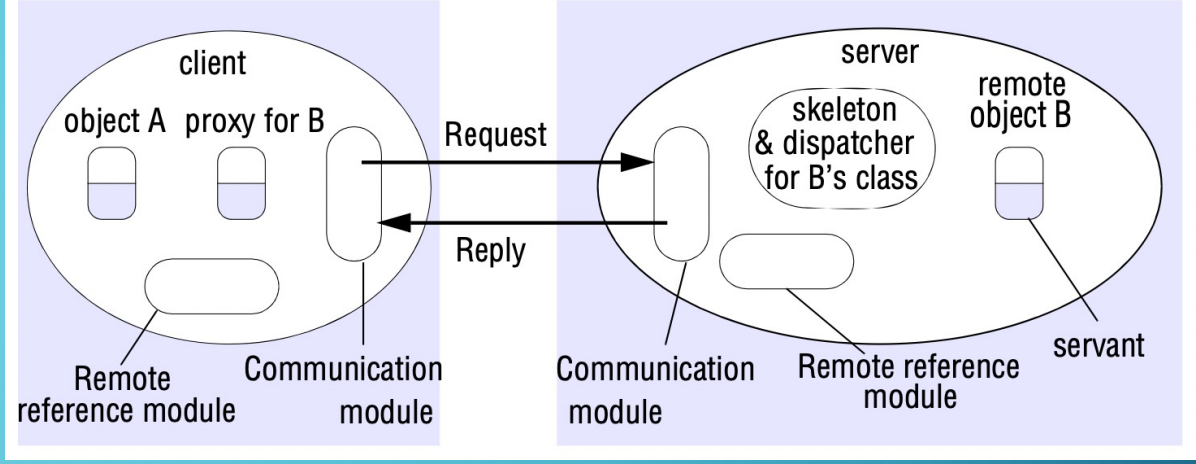


Figure from: Head First Design Patterns, 2020

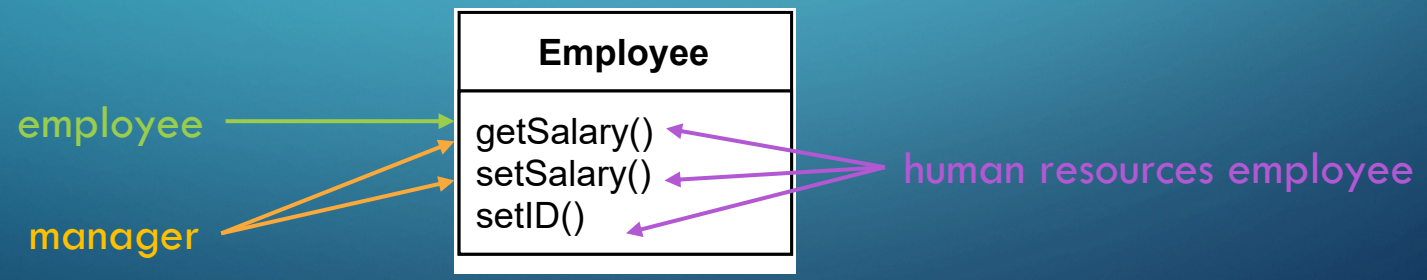
PROXY



- **The Proxy Pattern** provides a surrogate or placeholder for another object to control access to it.
  - Use the Proxy Pattern to create a representative object that controls access to another object, which may be remote, expensive to create, or in need of securing.
- For the client, usage of a proxy object is like using the real object, because they both implement the same interface.
- Extra functionality can be provided by the proxy:
  - Caching when operations on the real object are resource intensive
  - Checking preconditions before operations on the real object are invoked

PROXY

- Protection Proxy: controlling access to an object based on access rights
  - For instance, if we had an employee object, a Protection Proxy might allow the employee to call certain methods on the object, a manager to call additional methods (like setSalary()), and a human resources employee to call any method on the object.



PROXY PATTERN - EXAMPLE

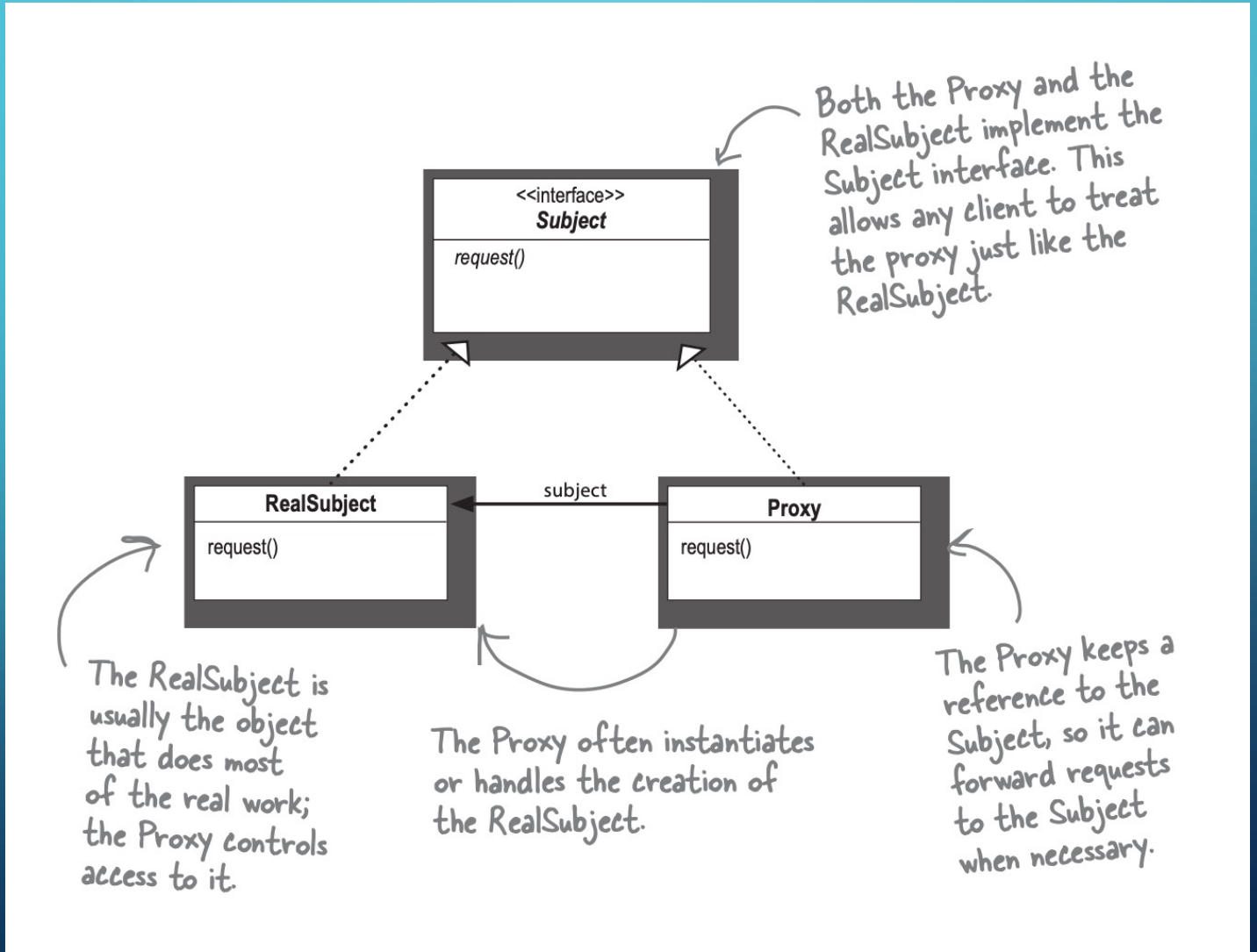


Figure from: Head First Design Patterns, 2020

PROXY PATTERN - EXAMPLE

```java
public interface Subject {
   public void request();
}
```

Subject interface


```java
public class RealSubject implements Subject{
   @Override
   public void request() {
      System.out.println(”requested");
   }
}
```

Implementation (proxy target)

```java
public class Proxy implements Subject{
   private Subject target;
   public Proxy(Subject target) {
      this.target = target;
   }

   @Override
   public void request() {
      // extensions to save()
      System.out.println("start requesting");
      target.request();
      System.out.println(”request complete");
   }
}
```

Proxy

PROXY PATTERN - EXAMPLE

```java
public interface Subject {
   public void request();
}
```

Subject interface

implements

implements

```java
public class RealSubject implements Subject{
   @Override
   public void request() {
      System.out.println("requested");
   }
}
```

Implementation (proxy target)

```java
public class Proxy implements Subject{
   private Subject target;
   public Proxy(Subject target) {
      this.target = target;
   }

   @Override
   public void request() {
      // extensions to save()
      System.out.println("start requesting");
      target.request();
      System.out.println("request complete");
   }
}
```

Additional functionality enabled by the proxy

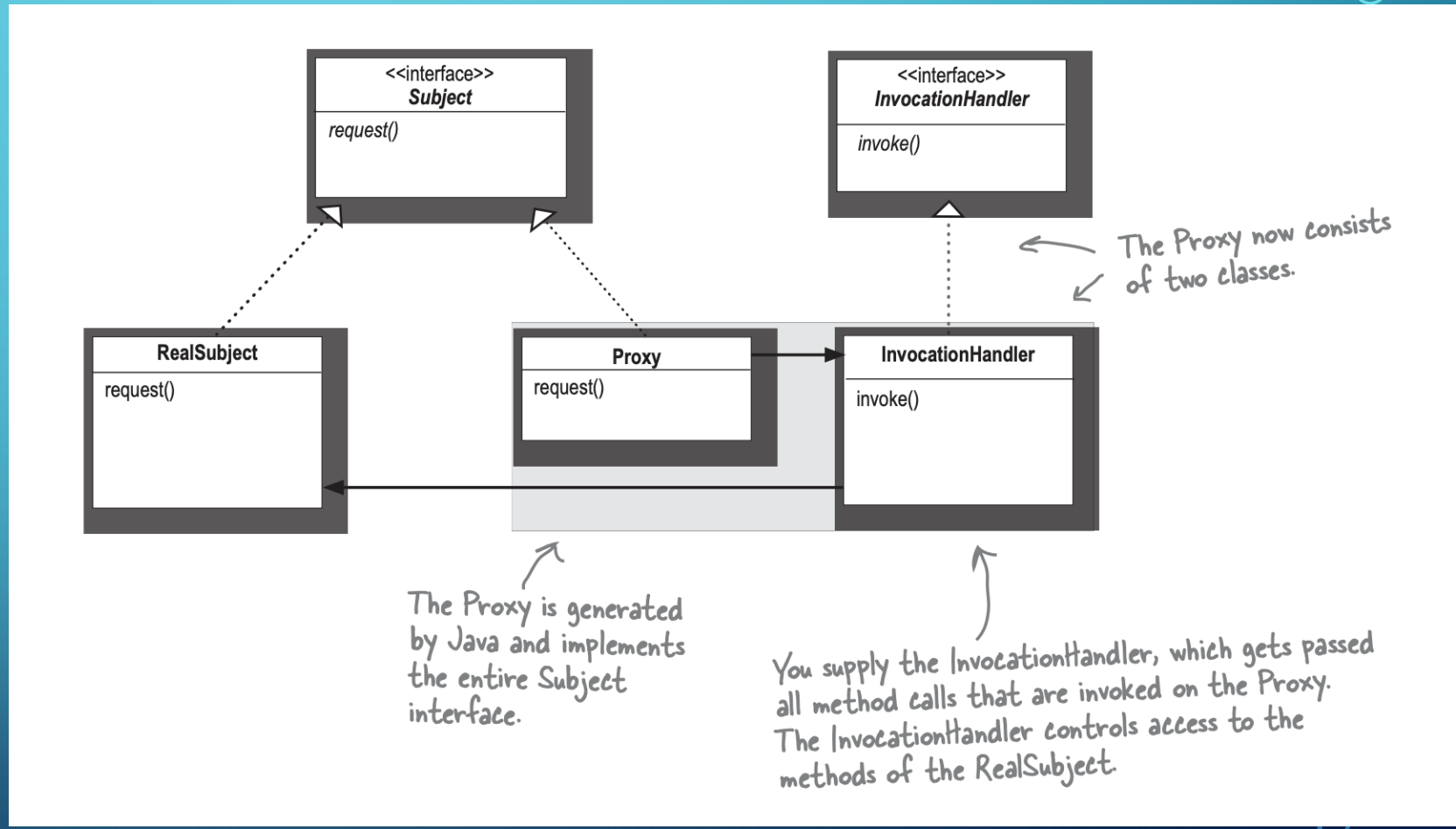
Proxy

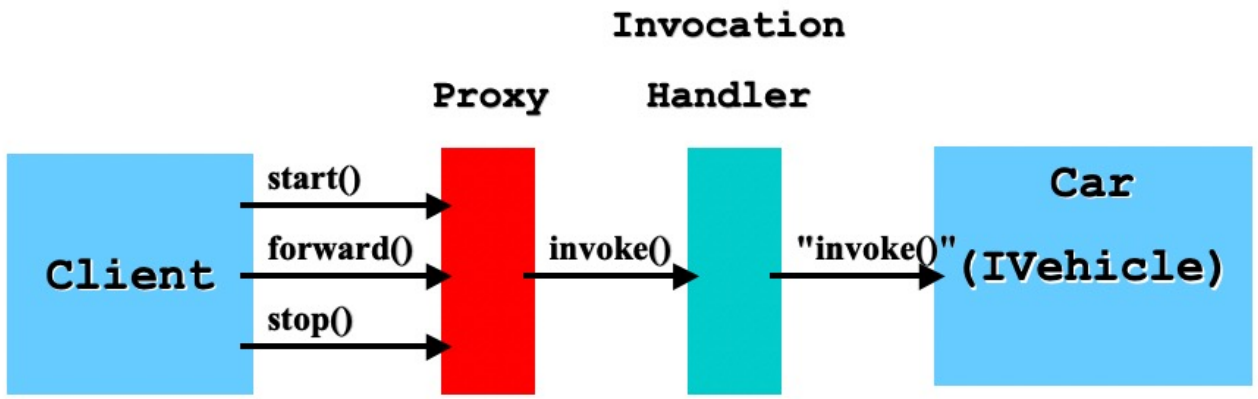
DYNAMIC PROXY

Java's got its own proxy support right in the java.lang.reflect package. With this package, Java lets you create a proxy class *on the fly* that implements one or more interfaces and forwards method invocations to a class that you specify. Because the actual proxy class is created at runtime, we refer to this Java technology as a *dynamic proxy*.

You don't write the Proxy class. The Proxy class is dynamically created on demand from the set of interfaces you pass it.

InvocationHandlers implement the behaviour of the proxy.





Invocation handler

```
import java.lang.reflect.*;
/**
 * Class VehicleHandler.
 */
public class VehicleHandler implements InvocationHandler {
  private IVehicle v;

  public VehicleHandler(IVehicle v) {this.v = v;}

  public Object invoke(Object proxy, Method m, Object[] args)
         throws Throwable {
    System.out.println("Vehicle Handler: Invoking " +
                       m.getName());
    return m.invoke(v, args);
  }
}
```

The invoke() method does the invocation on the real object

Interface

```
/**
 * Interface IVehicle.
 */
public interface IVehicle {
  public void start();
  public void stop();
  public void forward();
  public void reverse();
  public String getName();
}
```

Interface implementation

```
/**
 * Class Car
 */
public class Car implements IVehicle {
  private String name;

  public Car(String name) {this.name = name;}

  public void start() {
    System.out.println("Car " + name + " started");
  }

  // stop(), forward(), reverse() implemented similarly.
  // getName() not shown.
}
```

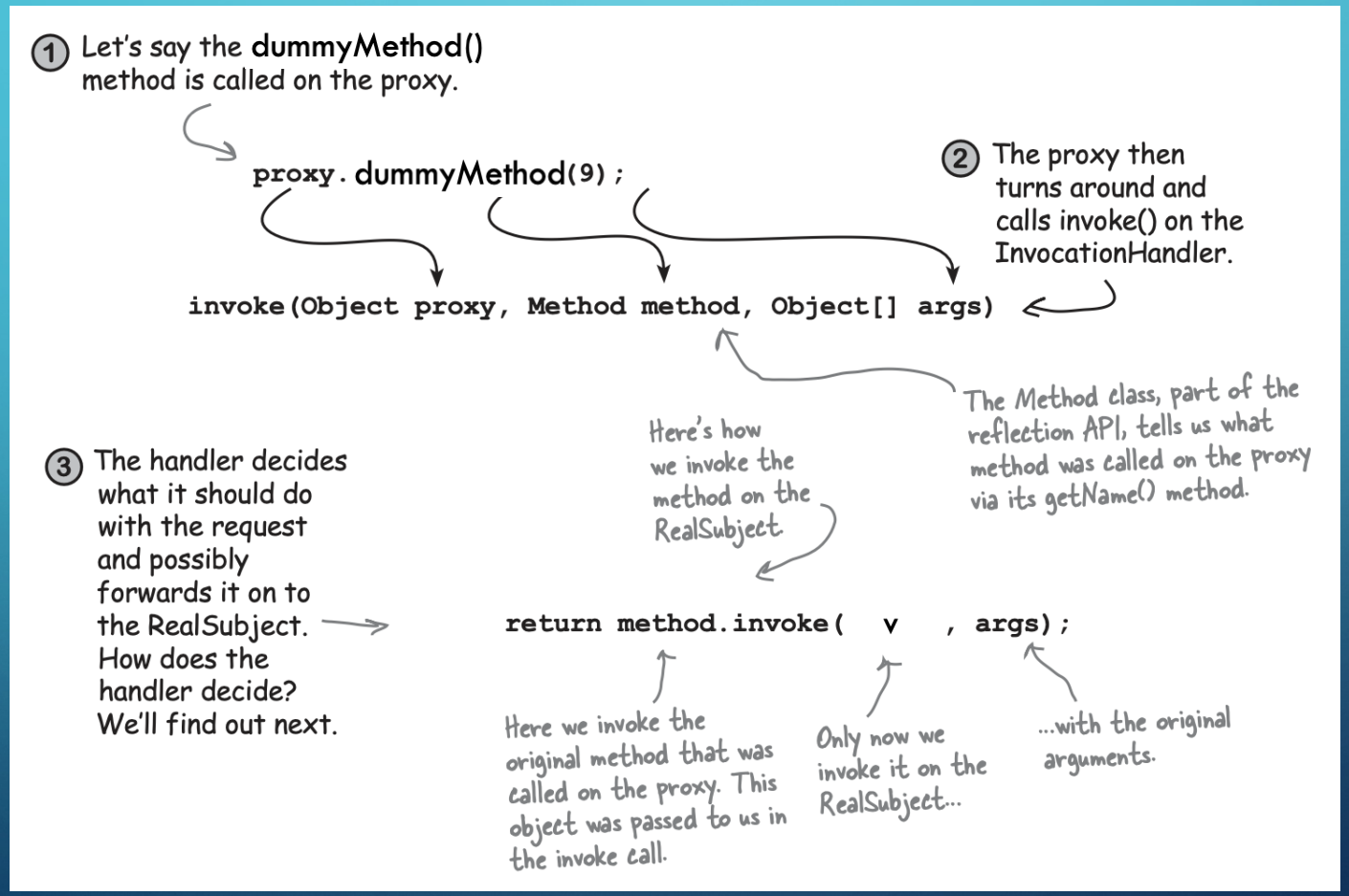
Client

```
import java.lang.reflect.*;
/**
 * Class Client3.
 * Interacts with a Car Vehicle through a dynamically
 * generated VehicleProxy.
 */
public class Client3 {
  public static void main(String[] args) {
    IVehicle c = new Car("Botar");
    ClassLoader cl = IVehicle.class.getClassLoader();
    IVehicle v = (IVehicle) Proxy.newProxyInstance(cl,
        new Class[] {IVehicle.class}, new VehicleHandler(c));
    v.start();
    v.forward();
    v.stop();
  }
}
```

Dynamic creation of an IVehicle proxy at runtime

https://web.archive.org/web/20150226062232/http://userpages.umbc.edu/~tarr/dp/lectures/DynProxies-2pp.pdf

The method *invoke()* in the InvocationHandler



This is the same as "v.method(args)"

INVOCATION HANDLER

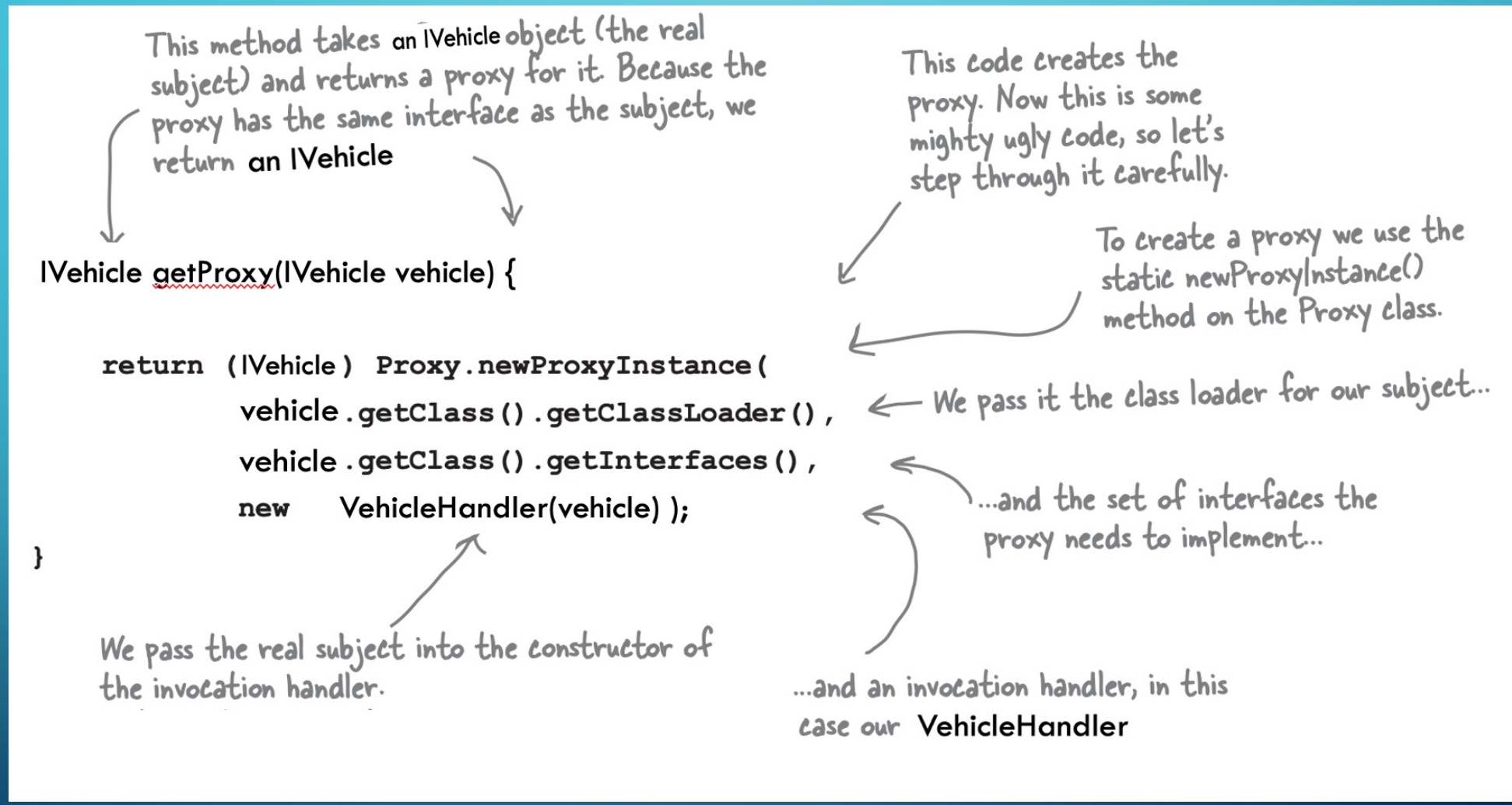
- `InvocationHandler` is the interface implemented by the invocation handler of a proxy instance.
- Each proxy instance has an associated invocation handler. When a method is invoked on a proxy instance, the method invocation is encoded and dispatched to the invoke method of its invocation handler.
- `Object invoke(Object proxy, Method method, Object[] args) throws Throwable`
  - proxy - the proxy instance that the method was invoked on
  - method - the Method instance corresponding to the interface method invoked on the proxy instance. The declaring class of the Method object will be the interface that the method was declared in, which may be a superinterface of the proxy interface that the proxy class inherits the method through.
  - args - an array of objects containing the values of the arguments passed in the method invocation on the proxy instance, or null if interface method takes no arguments. Arguments of primitive types are wrapped in instances of the appropriate primitive wrapper class, such as `java.lang.Integer` or `java.lang.Boolean`.

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

PROXY INSTANCE

- `public static Object newProxyInstance(ClassLoader loader, Class<?>[] interfaces, InvocationHandler h) throws IllegalArgumentException`
  - loader - the class loader to define the proxy class
  - interfaces - the list of interfaces for the proxy class to implement
  - h - the invocation handler to dispatch method invocations to
- Returns: a proxy instance with the specified invocation handler of a proxy class that is defined by the specified class loader and that implements the specified interfaces
- You don't write the Proxy class. The Proxy class is dynamically created on demand from the set of interfaces you pass it.

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


This method takes an IVehicle object (the real subject) and returns a proxy for it. Because the proxy has the same interface as the subject, we return an IVehicle
This code creates the proxy. Now this is some mighty ugly code, so let's step through it carefully.
To create a proxy we use the static newProxyInstance() method on the Proxy class.
IVehicle getProxy(IVehicle vehicle) {
return (IVehicle) Proxy.newProxyInstance(
vehicle.getClass().getClassLoader(),
vehicle.getClass().getInterfaces(),
new VehicleHandler(vehicle));
}
We pass it the class loader for our subject...
...and the set of interfaces the proxy needs to implement...
We pass the real subject into the constructor of the invocation handler.
...and an invocation handler, in this case our VehicleHandler

Another Example

Implementation of the invoke() method in a customised invocation handler

```
public class DebugProxy implements java.lang.reflect.InvocationHandler {
    private Object obj;

    ……

    private DebugProxy(Object obj) {
        this.obj = obj;
    }

    @Override
    public Object invoke(Object proxy, Method m, Object[] args) throws Throwable {
        Object result;
        try {
            System.out.println("before method " + m.getName());
            result = m.invoke(obj, args);
        } catch (InvocationTargetException e) {
            throw e.getTargetException();
        } catch (Exception e) {
            throw new RuntimeException("unexpected invocation exception: " + e.getMessage());
        } finally {
            System.out.println("after method " + m.getName());
        }
        return result;
    }
}
```

Equals to obj.m(args)

```
public interface Foo {
   Object bar(Object obj) throws BazException;
}

public class FooImpl implements Foo {
   Object bar(Object obj) throws BazException {
      // ...
   }
}

public class DebugProxy implements java.lang.reflect.InvocationHandler {

   private Object obj;

   public static Object newInstance(Object obj) {
      return java.lang.reflect.Proxy.newProxyInstance(
            obj.getClass().getClassLoader(),
            obj.getClass().getInterfaces(),
            new DebugProxy(obj));
   }

   private DebugProxy(Object obj) {
      this.obj = obj;
   }

   @Override
   public Object invoke(Object proxy, Method m, Object[] args) throws Throwable {
      ...
   }

   public static void main(String[] args) {
      Foo foo = (Foo) DebugProxy.newInstance(new FooImpl());
      foo.bar(null);
   }
}
```

Dynamically create a proxy of obj

We plug in our customised invocation handler

PRACTICE

- Implement a simple program with the proxy pattern.
- (Optional) Try to create a proxy instance of an **interface**.

```
public interface IHelloWorld {
   public void hello();
}
```

  - What to expect: Calling method hello() will print a line: Hello world!
  - Note: You shall not create any implementation class for the interface

Further readings (available on *Blackboard*):

- Chapter 11 the Proxy Pattern, Head First Design Patterns, 2020.