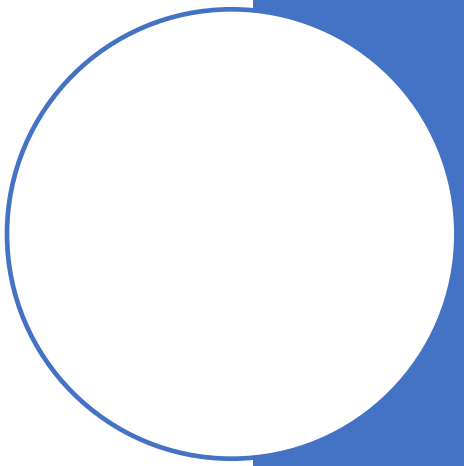
Virtualization & Containerization



Virtualization

In 1960s, the problem of sharing the computer resources among multiple applications is faced. Only one application could run at a time.

Later on, several mechanisms were proposed to share the resources among applications, but a **new problem** was faced. What, if one application consumes all the resources?

There are two ways to achieve virtualization, i) Virtual Machines (VMs), and ii) Containers

Virtual Machines (VMs)

Virtualization (using VMs) solve the problem using **isolation** and **sharing** mechanisms among resources, i.e., process scheduling (schedular assign threads to CPUs), using virtual memory to complement physical memory, disk controller to access the disk and allowing only the valid threads to access the disk content, and a network isolation is achieved through the identification of messages.

Every VM has an address used to identify messages to or from that VM. The **hypervisor** implements network infrastructure within the physical machine that allows VMs to share and isolate use of physical network interface using the same approaches and protocols used for networking between physical machines.

VMs allow the execution of multiple simulated, or virtual, computers in a single physical computer.

Virtualization is achieved at hardware level.

Example: Creating multiple servers (using VMs) on a single machine, each one is independent of other.

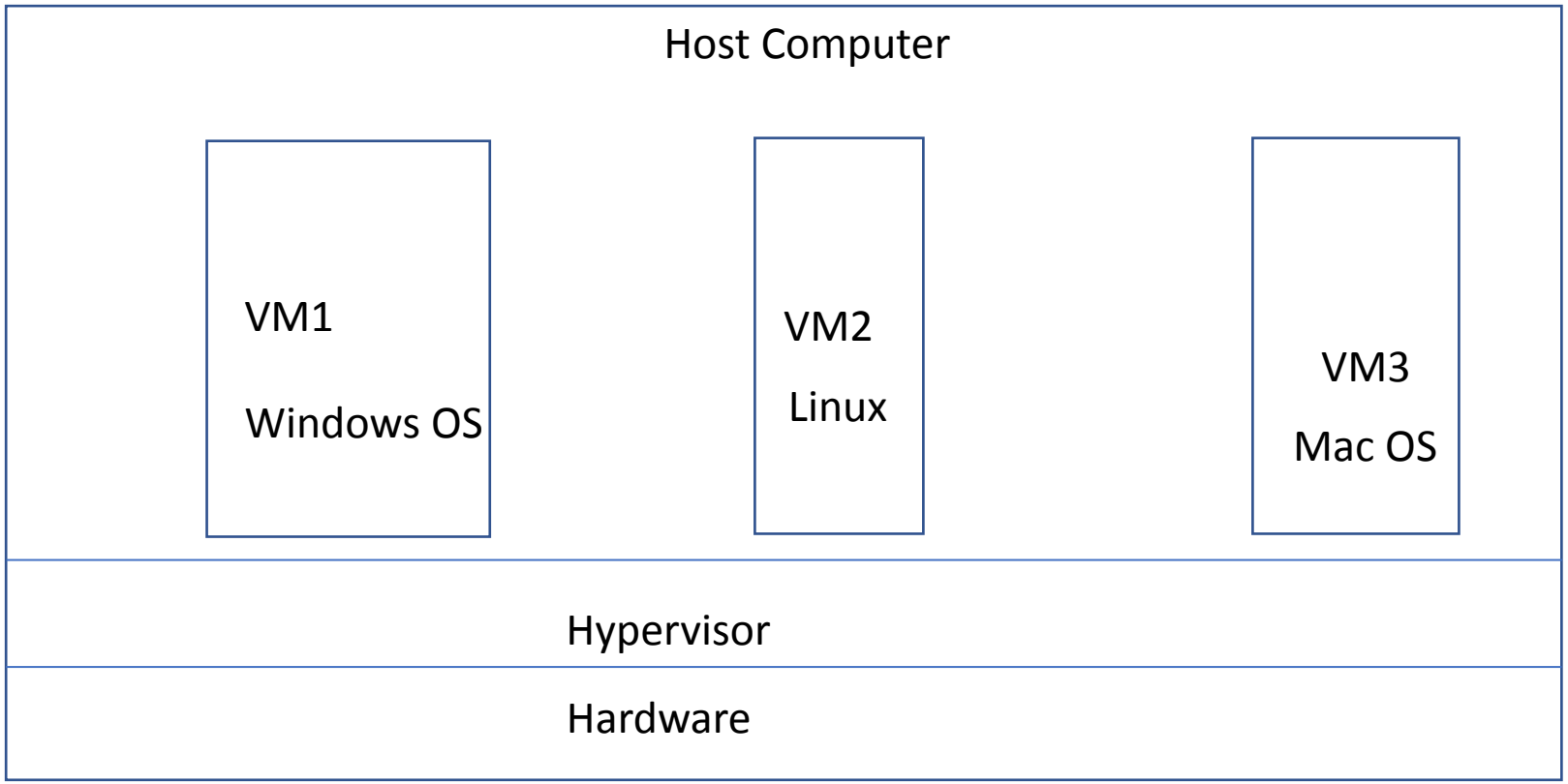
Virtual Machines (VMs)
Hypervisor

VMs must be managed, i.e, they must be created and destroyed. Managing VMs is a function of the hypervisor by taking instructions from the user or cloud infrastructure. It also monitors VMs, health checks, security etc. It also ensures that VM does not exceed its resource utilization limits.

Imagine you have 100 servers, and you need to configure different OS, Software, and libraries on these systems. How can you do that, and the problems/challenges associated?

VM images.

Virtual Machines (VM)



Containers

VMs solve the problem of sharing resources and maintaining isolation but VM images can be large and is time consuming to transfer VM images around the network i.e., transferring 8GB VM image to 2000 machines.

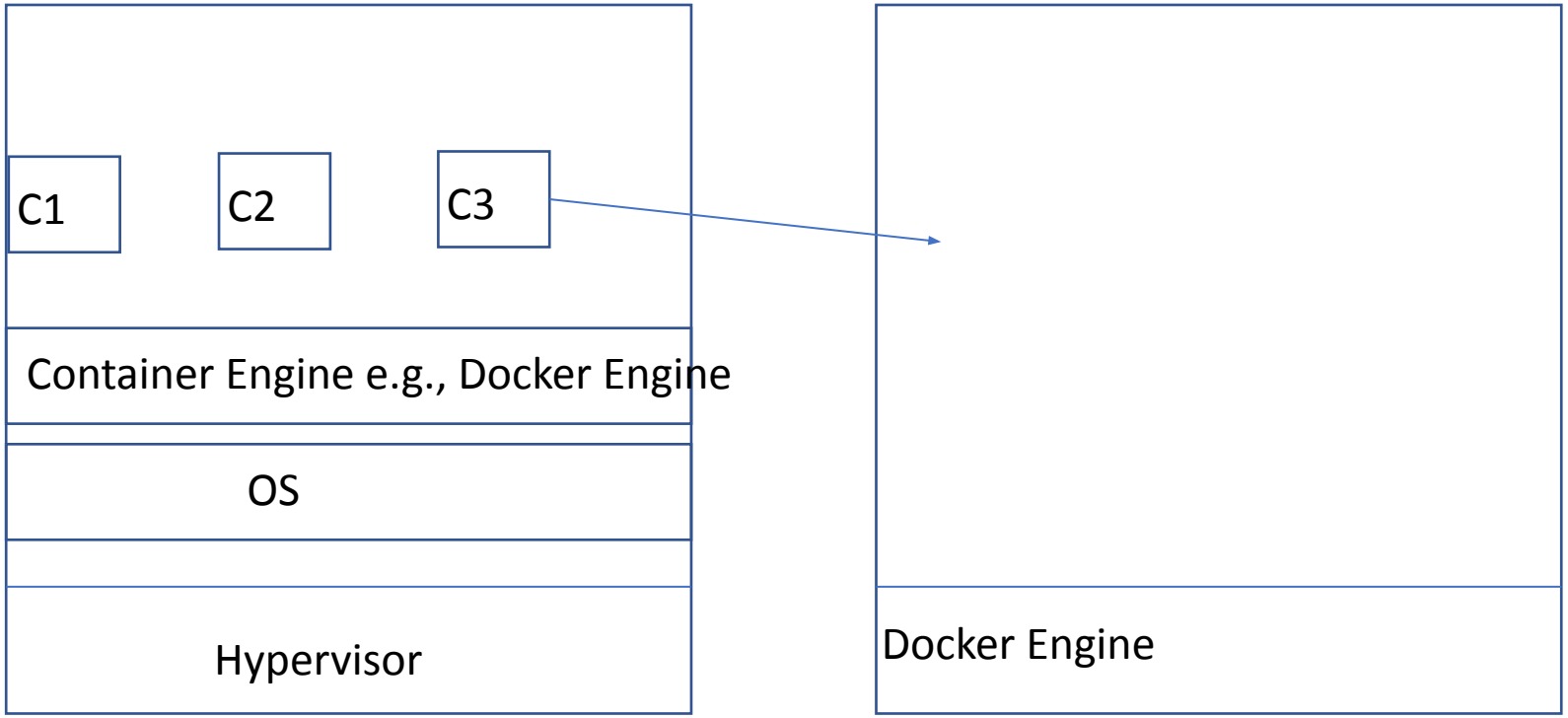
Once the image is transferred, it needs to boot the OS and start your services, which still takes more time.

Containers are a mechanism to maintain the advantages of virtualization while reducing the image transfer time and start-up time.

Containers creates illusion of the isolation of processes (i.e., one container containing an application is isolated from another container), hence a security is maintained because one container can not mistakenly access the process of other container.

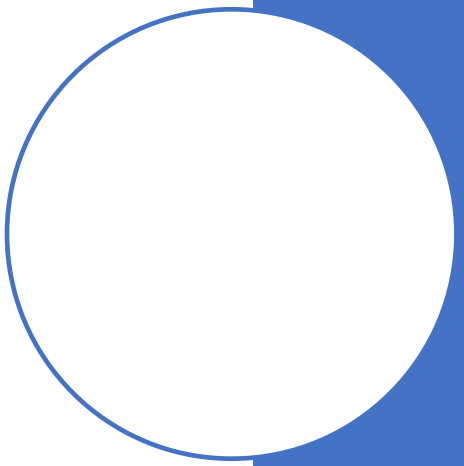
If we talk about cloud, a container is a package of software e.g., docker image.

Container



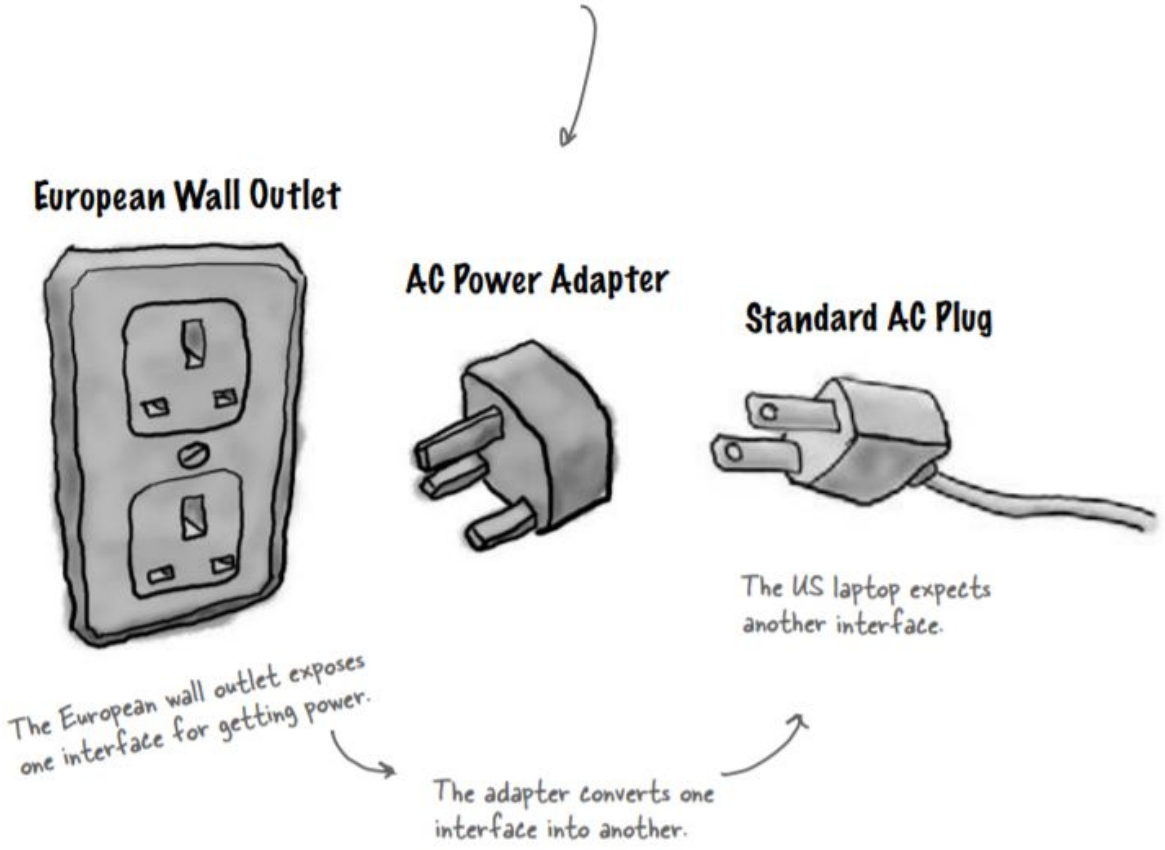
Example in DevOps Context.

Design Patterns



Adapter Pattern

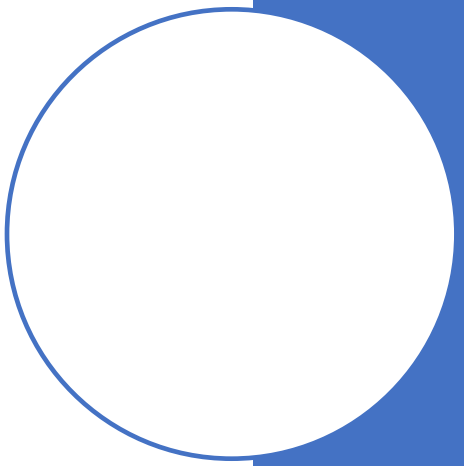
You'll have no trouble understanding what an OO adapter is because the real world is full of them. How's this for an example: Have you ever needed to use a US-made laptop in a European country? Then you've probably needed an AC power adapter...



Adapter Pattern

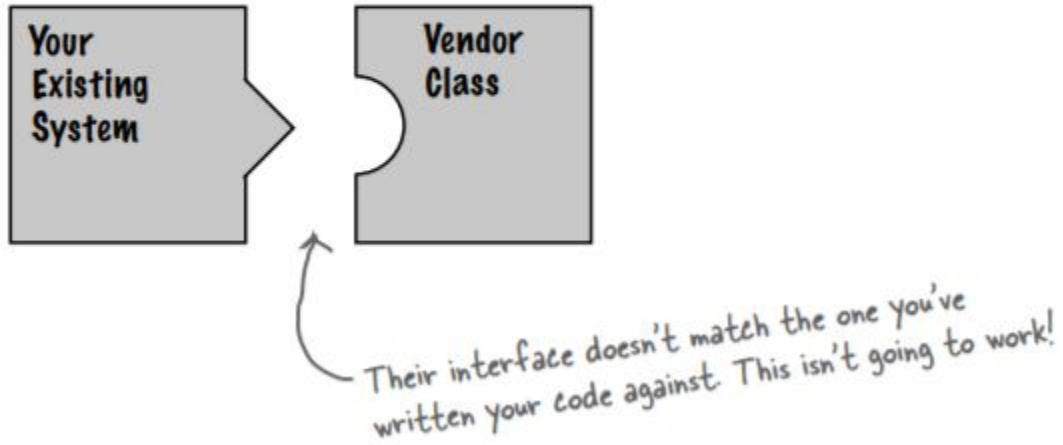
Now, what about Object-Oriented (OO) adapters?

Well, our OO adapters play the same role as their real-world counterparts: they take an interface and adapt it to one that a client is expecting.



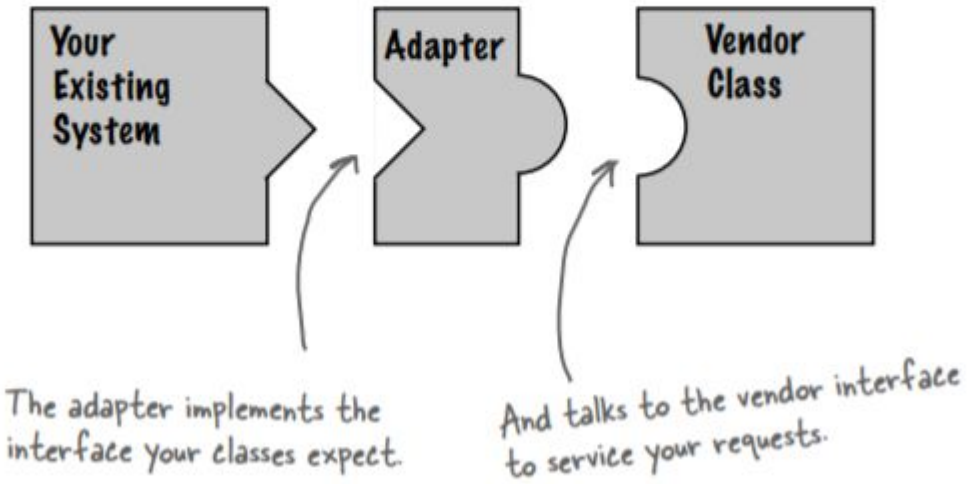
Adapter Pattern

Say you've got an existing software system that you need to work a new vendor class library into, but the new vendor designed their interfaces differently than the last vendor:



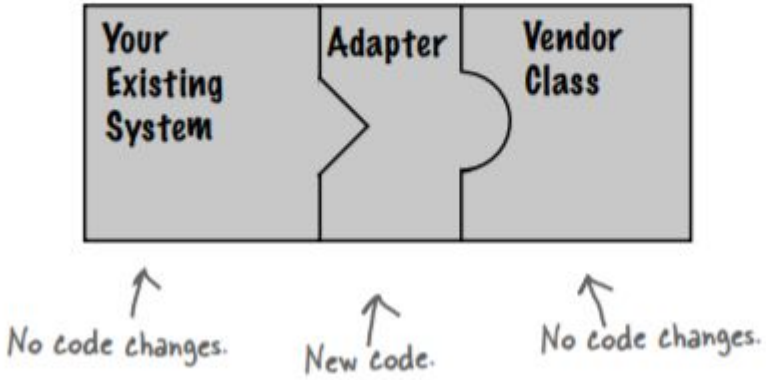
Adapter Pattern

Okay, you don't want to solve the problem by changing your existing code (and you can't change the vendor's code). So what do you do? Well, you can write a class that adapts the new vendor interface into the one you're expecting.



Adapter Pattern

The adapter acts as the middleman by receiving requests from the client and converting them into requests that make sense on the vendor classes.



Can you think of a solution that doesn't require YOU to write ANY additional code to integrate the new vendor classes? How about making the vendor supply the adapter class.

Let's see the Implementation with an Example

If it walks like a duck and quacks like a duck, then it ~~must~~ might be a ~~duck~~ turkey wrapped with a duck adapter...

It's time to see an adapter in action. Remember our ducks from Chapter 1? Let's review a slightly simplified version of the Duck interfaces and classes:

```
public interface Duck {
    public void quack();
    public void fly();
}
```

This time around, our ducks implement a Duck interface that allows Ducks to quack and fly.

Let's see the Implementation with an Example

Here's a subclass of Duck, the MallardDuck.

```
public class MallardDuck implements Duck {
    public void quack() {
        System.out.println("Quack");
    }

    public void fly() {
        System.out.println("I'm flying");
    }
}
```

Simple implementations: the duck just prints out what it is doing.

Let's see the Implementation with an Example

Now it's time to meet the newest fowl on the block:

```
public interface Turkey {
    public void gobble();
    public void fly();
}
```

Turkeys don't quack, they gobble.

Turkeys can fly, although they can only fly short distances.

Let's see the Implementation with an Example

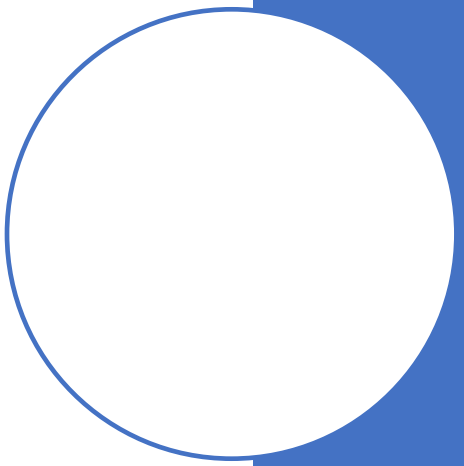
```
public class WildTurkey implements Turkey {
    public void gobble() {
        System.out.println("Gobble gobble");
    }

    public void fly() {
        System.out.println("I'm flying a short distance");
    }
}
```

Here's a concrete implementation of Turkey; like Duck, it just prints out its actions.

Let’s see the Implementation with an Example

Now, let’s say you’re short on Duck objects and you’d like to use some Turkey objects in their place. Obviously, we can’t use the turkeys outright because they have a different interface.

So, let’s write an Adapter:



Let's see the Implementation with an Example

First, you need to implement the interface of the type you're adapting to. This is the interface your client expects to see.

```
public class TurkeyAdapter implements Duck {
    Turkey turkey;

    public TurkeyAdapter(Turkey turkey) {
        this.turkey = turkey;
    }

    public void quack() {
        turkey.gobble();
    }

    public void fly() {
        for(int i=0; i < 5; i++) {
            turkey.fly();
        }
    }
}
```

Next, we need to get a reference to the object that we are adapting; here we do that through the constructor.

Now we need to implement all the methods in the interface; the quack() translation between classes is easy: just call the gobble() method.

Even though both interfaces have a fly() method, Turkeys fly in short spurts – they can't do long-distance flying like ducks. To map between a Duck's fly() method and a Turkey's, we need to call the Turkey's fly() method five times to make up for it.

Let's see the Implementation with an Example

Let see some code to test drive our adapter:

```
public class DuckTestDrive {
    public static void main(String[] args) {
        MallardDuck duck = new MallardDuck();

        WildTurkey turkey = new WildTurkey();
        Duck turkeyAdapter = new TurkeyAdapter(turkey);

        System.out.println("The Turkey says...");
        turkey.gobble();
        turkey.fly();

        System.out.println("\nThe Duck says...");
        testDuck(duck);

        System.out.println("\nThe TurkeyAdapter says...");
        testDuck(turkeyAdapter);
    }

    static void testDuck(Duck duck) {
        duck.quack();
        duck.fly();
    }
}
```

Let's create a Duck...

and a Turkey.

And then wrap the turkey in a TurkeyAdapter, which makes it look like a Duck.

Then, let's test the Turkey: make it gobble, make it fly.

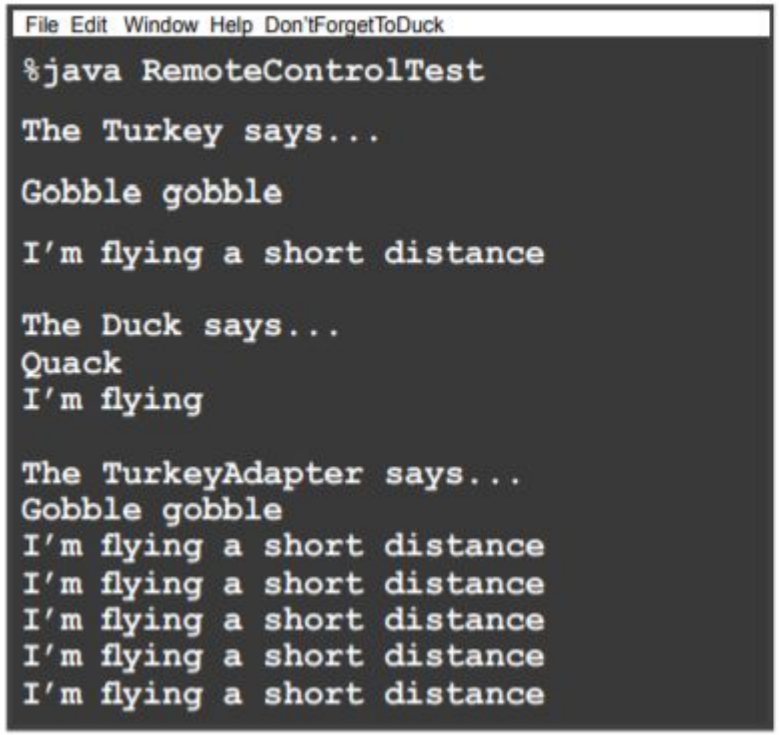
Now let's test the duck by calling the testDuck() method, which expects a Duck object.

Now the big test: we try to pass off the turkey as a duck...

Here's our testDuck() method; it gets a duck and calls its quack() and fly() methods.

Let’s see the Implementation with an Example

Output

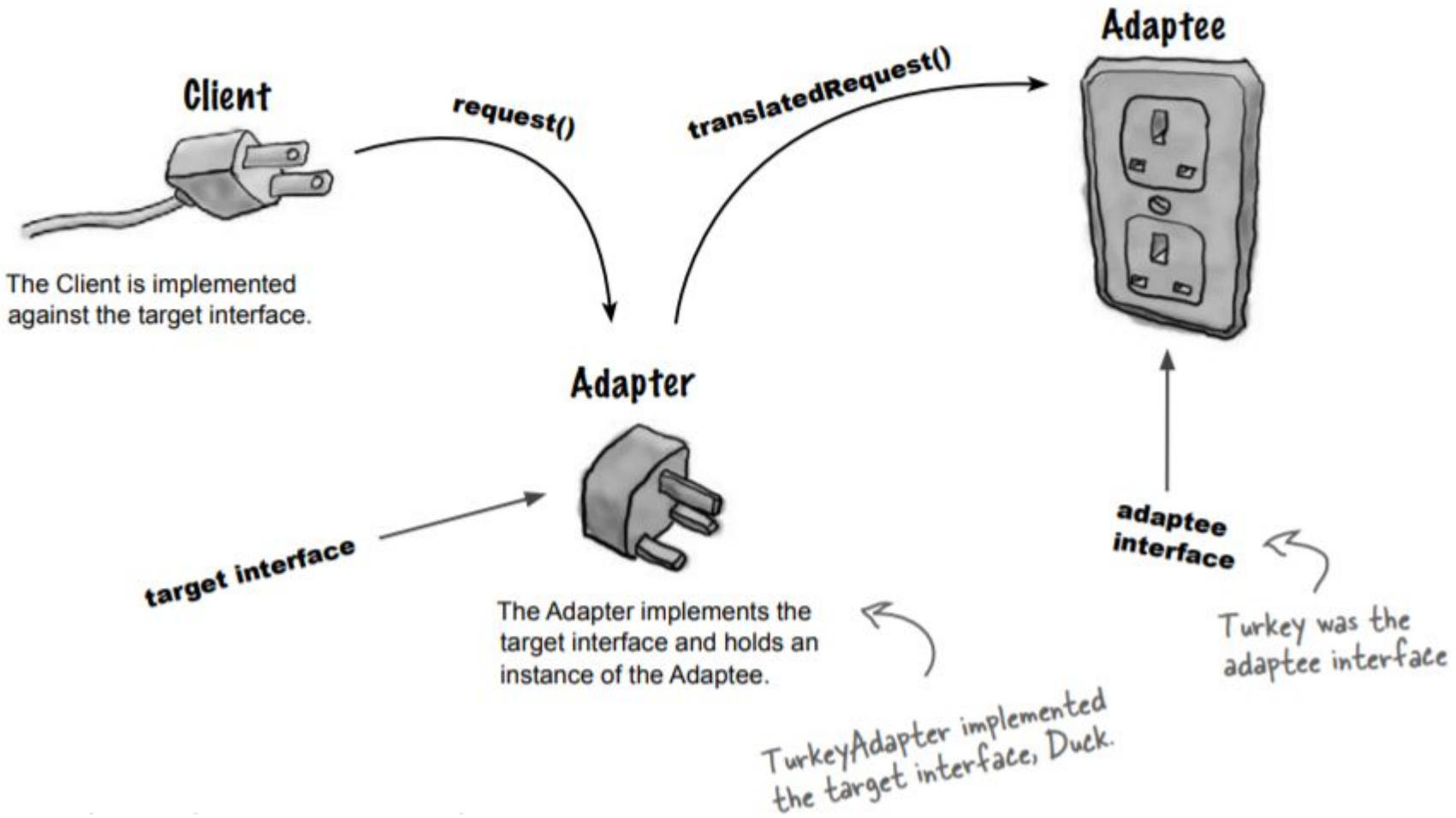


The Turkey gobbles and flies a short distance.

The Duck quacks and flies just like you'd expect.

And the adapter gobbles when quack() is called and flies a few times when fly() is called. The testDuck() method never knows it has a turkey disguised as a duck!

Explanation



Explanation

Here's how the Client uses the Adapter

1. **The client makes a request to the adapter by calling a method on it using the target interface.**

2. **The adapter translates the request into one or more calls on the adaptee using the adaptee interface.**

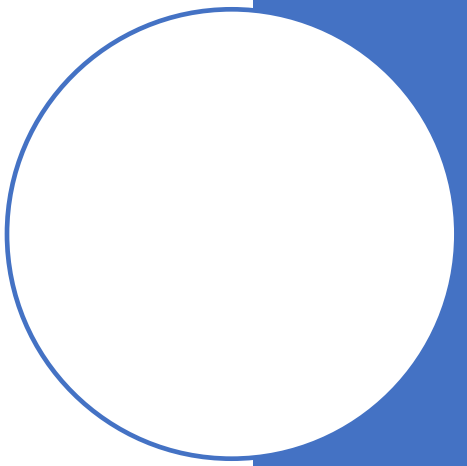
3. **The client receives the results of the call and never knows there is an adapter doing the translation.**

Note that the Client and Adaptee are decoupled – neither knows about the other.

Adapter Pattern

The **Adapter Pattern** converts the interface of a class into another interface the clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

Now, we know this pattern allows us **to use a client with an incompatible interface** by creating an Adapter that does the conversion. This acts to **decouple the client** from the implemented interface, and if we expect the interface to change over time, the adapter encapsulates that change so that the client doesn't have to be modified each time it needs to operate against a different interface.



Object Diagram