Design Patterns (II)
Adapter and Proxy Patterns
أنماط التصميم (II)
المحولات وأنماط البروكسي

Lecture 3

Teacher: Baida'a Lala'a

Adapter Pattern

محول

- The Adapter pattern is a versatile pattern that joins together types that were not designed to work with each other. It is one of those patterns that comes in useful when dealing with **legacy code**—i.e., code that was written a while ago and to which one might not have access. In such cases, Adapters make legacy code work with modern classes.
- There are different kinds of adapters, including
  - **Class, Object, Two-way, and Pluggable.**

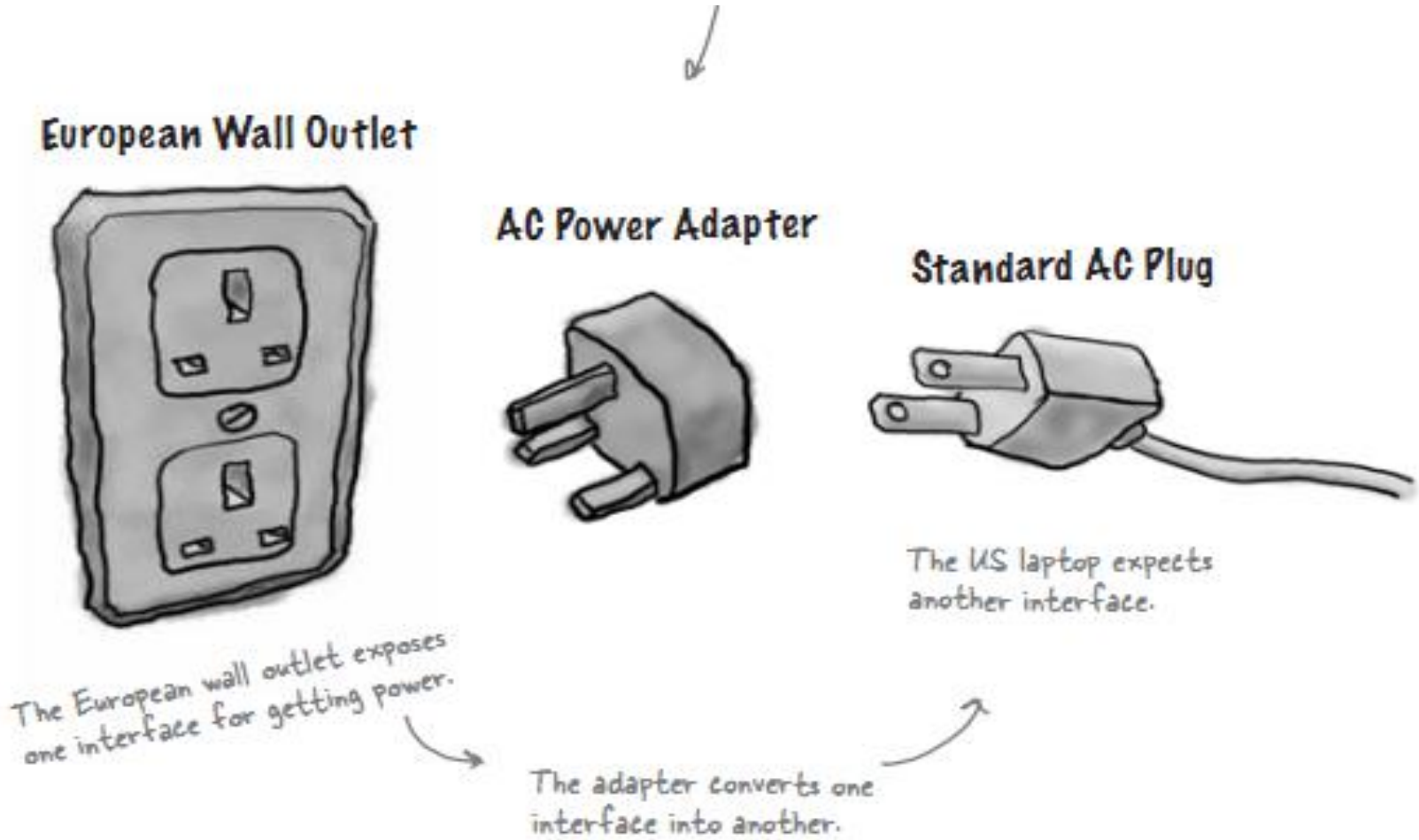
نمط المحول هو نمط متعدد الاستخدامات يجمع بين الأنواع التي لم تكن مصممة للعمل مع الآخرين .انها واحدة من هؤلاء الأنماط التي تكون مفيدة عند التعامل
مع رمز legacycode —ie،
الكود الذي تم كتابته أثناء الانتقال إلى ما لا يملكه شخص واحد الوصول.في مثل هذه الحالات ، المحولات تجعل العمل مع الحديث
الطبقات.

- أنواع مختلفة من المحولات ، بما في ذلك
- فئة ، كائن ، ثنائي الاتجاه ، وقابل للتوصيل.

- The Adapter pattern enables a system to use classes whose interfaces **don’t quite** match **its requirements**. Many examples of the Adapter pattern involve input/output because that is one domain that is constantly changing.
  - For example, programs written in the 1980s will have very different user interfaces from those written in the 2000s. Being able to adapt those parts of the system to new hardware facilities would be much more cost effective than rewriting them.

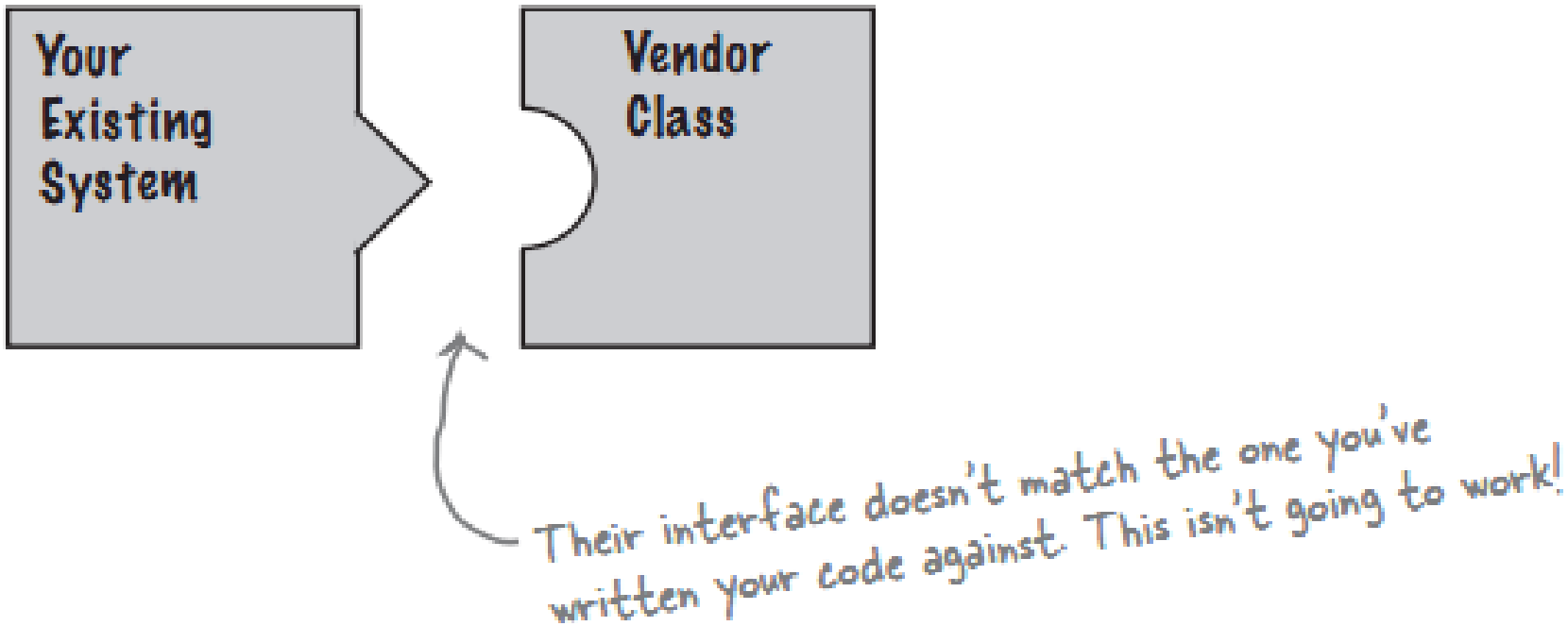
يتيح نمط المحول نظامًا لاستخدام الفئات ذات الواجهات
دون 'ر تماما تتطابق متطلباته .العديد من الأمثلة على
المحول
يشتمل النمط على إدخال / إخراج لأن هذا هو أحد المجالات
يتغير باستمرار.
•على سبيل المثال ، سيكون للبرامج المكتوبة في
الثمانينيات مستخدم مختلف تمامًا
واجهات من تلك المكتوبة في ٢٠٠٠ .sوالقدرة على التكيف
مع هذه الأجزاء
سيكون النظام إلى مرافق الأجهزة الجديدة أكثر فعالية من
حيث التكلفة
من إعادة كتابتها

Adapter Pattern نمط المحول

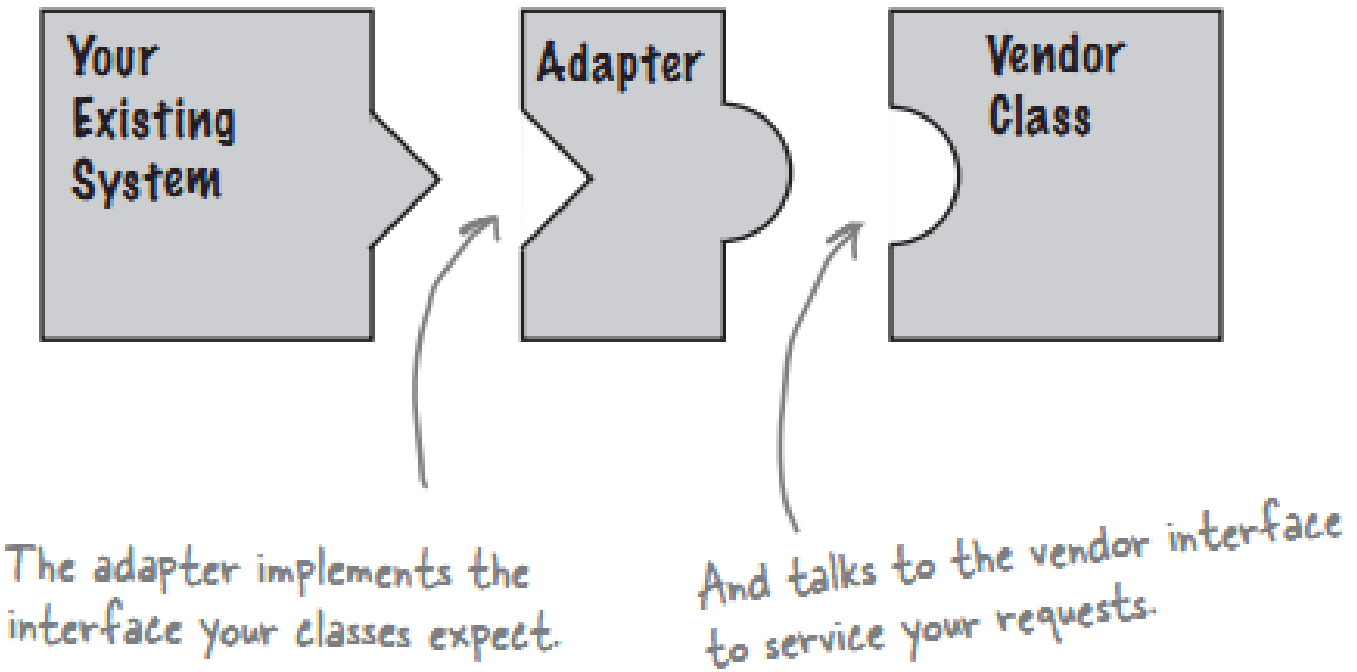


Object oriented adapters

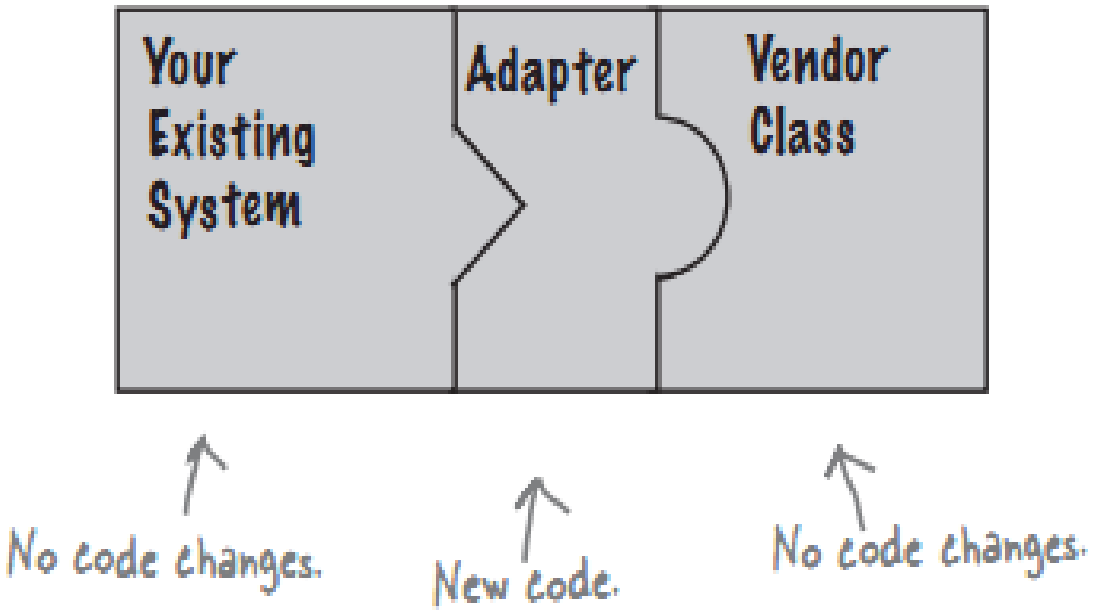
Say you've got an existing software system that you need to work a new vendor class library into, but the new vendor designed their interfaces differently than the last vendor:



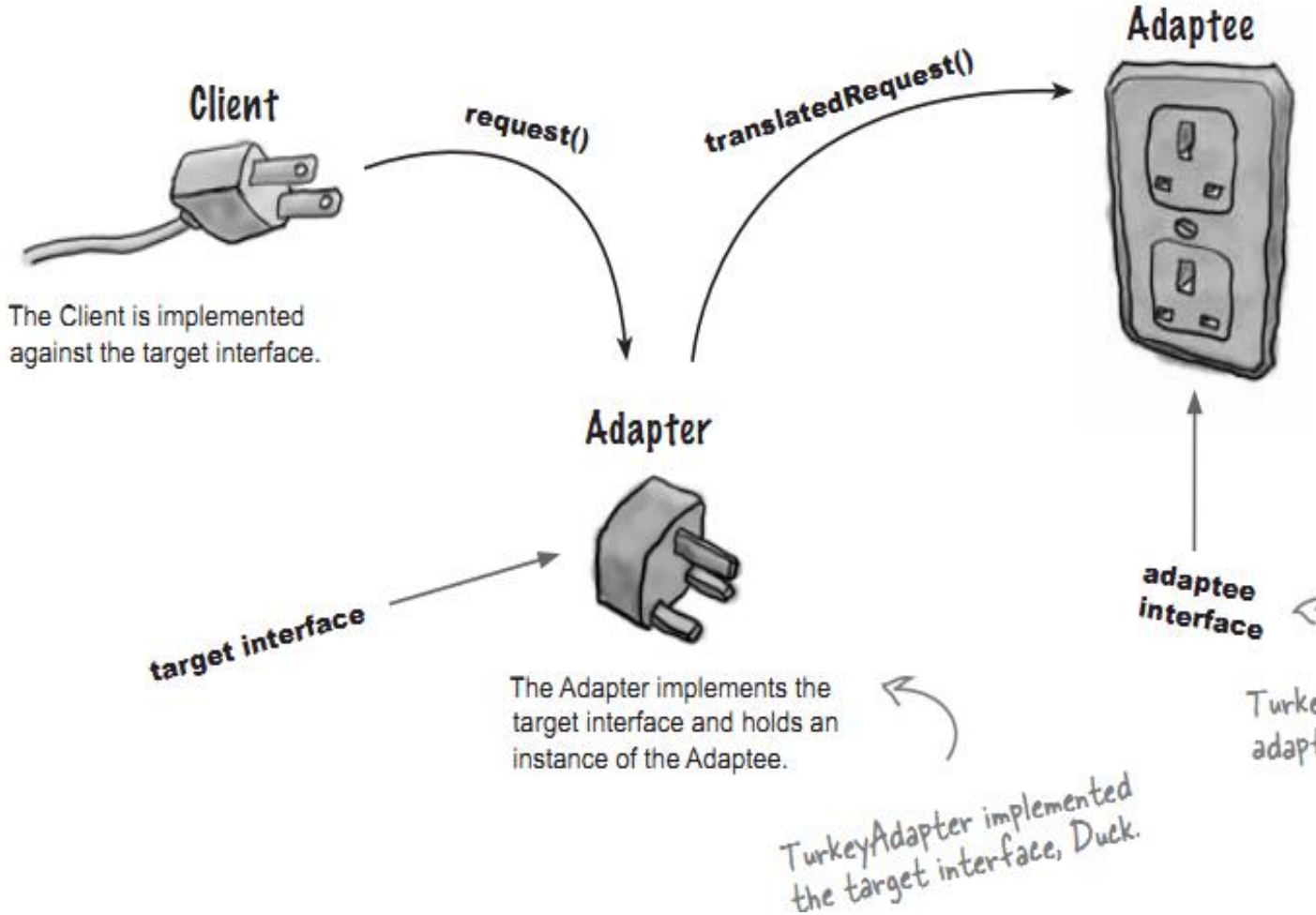
Okay, you don't want to solve the problem by changing your existing code (and you can't change the vendor's code). So what do you do? Well, you can write a class that adapts the new vendor interface into the one you're expecting.



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



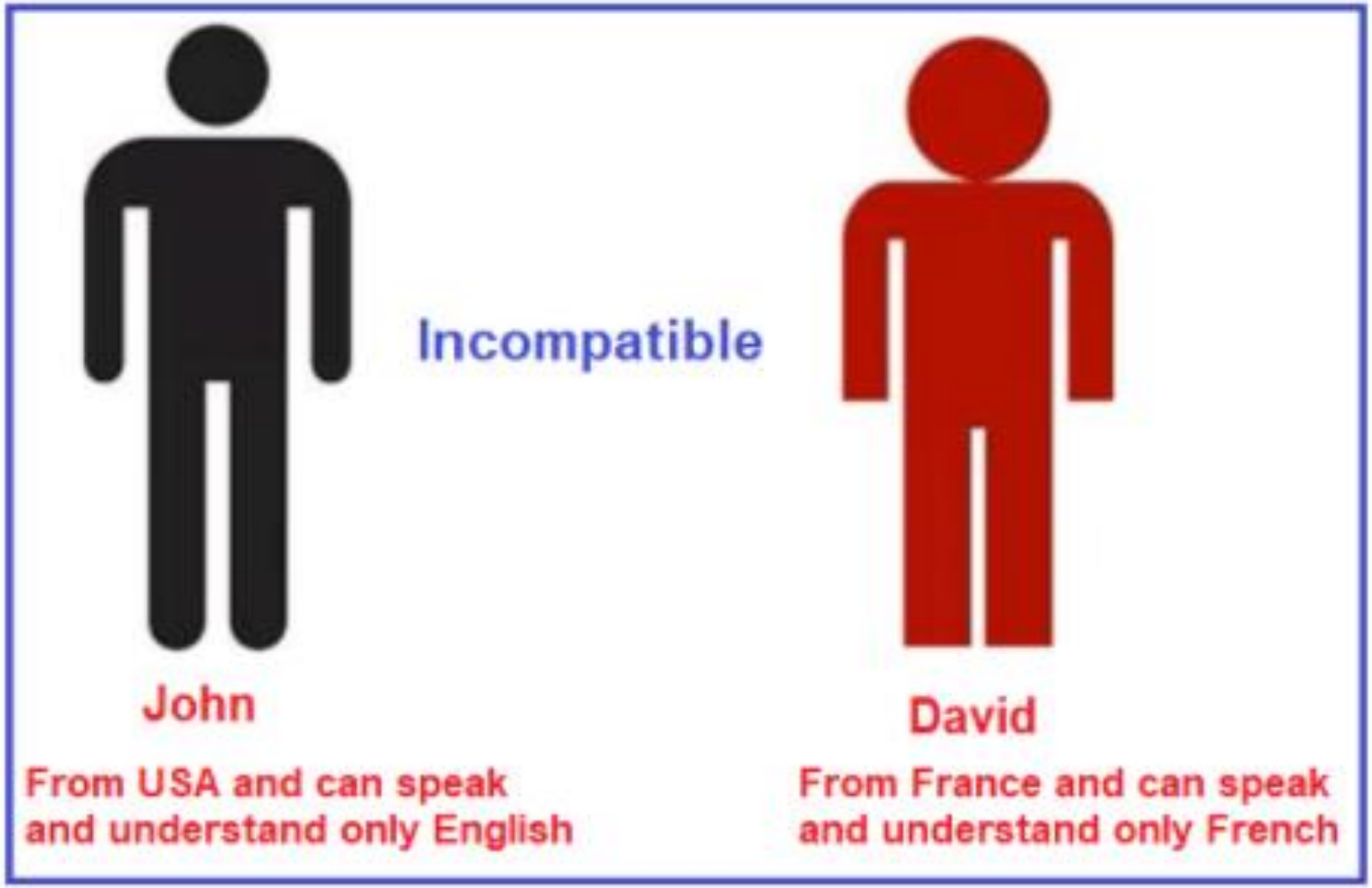
Can you think of a solution that doesn't require YOU to write ANY additional code to integrate the new vendor classes? How about making the vendor supply the adapter class.

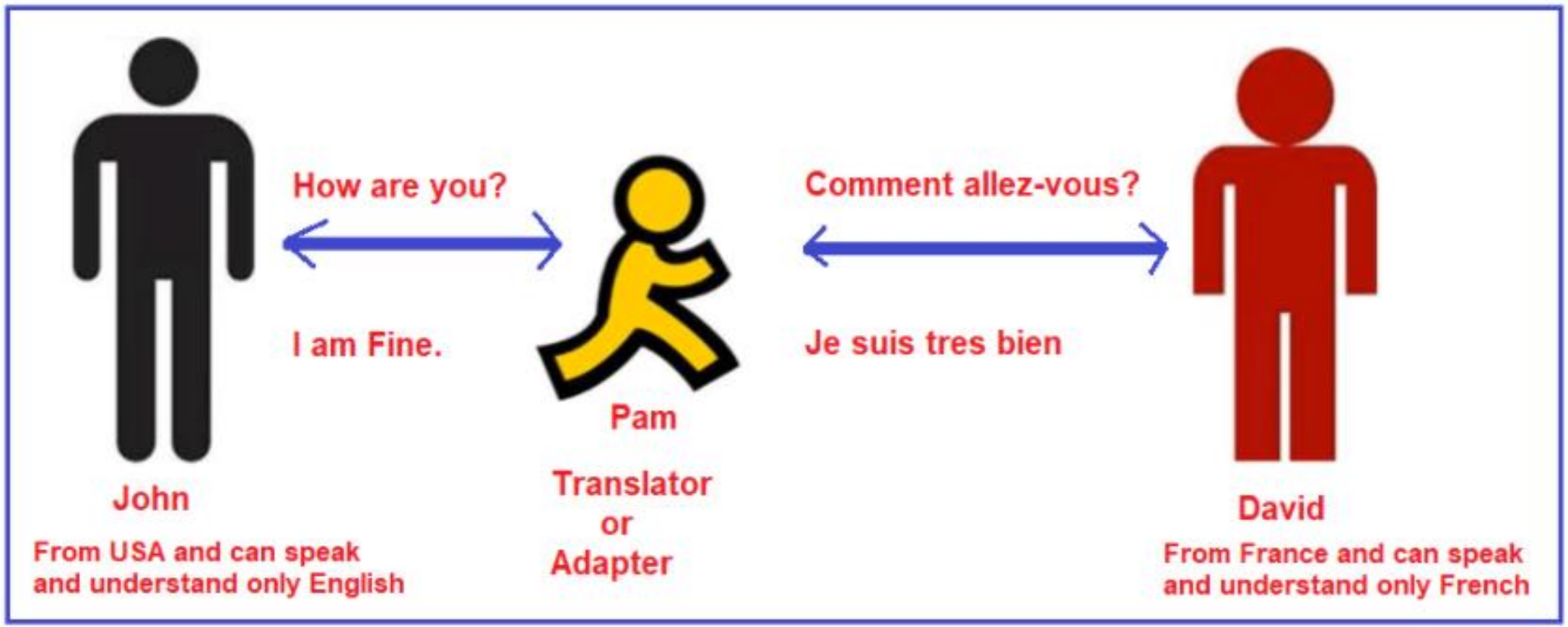


Here's how the Client uses the Adapter

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

Language Translator example مثال مترجم اللغة



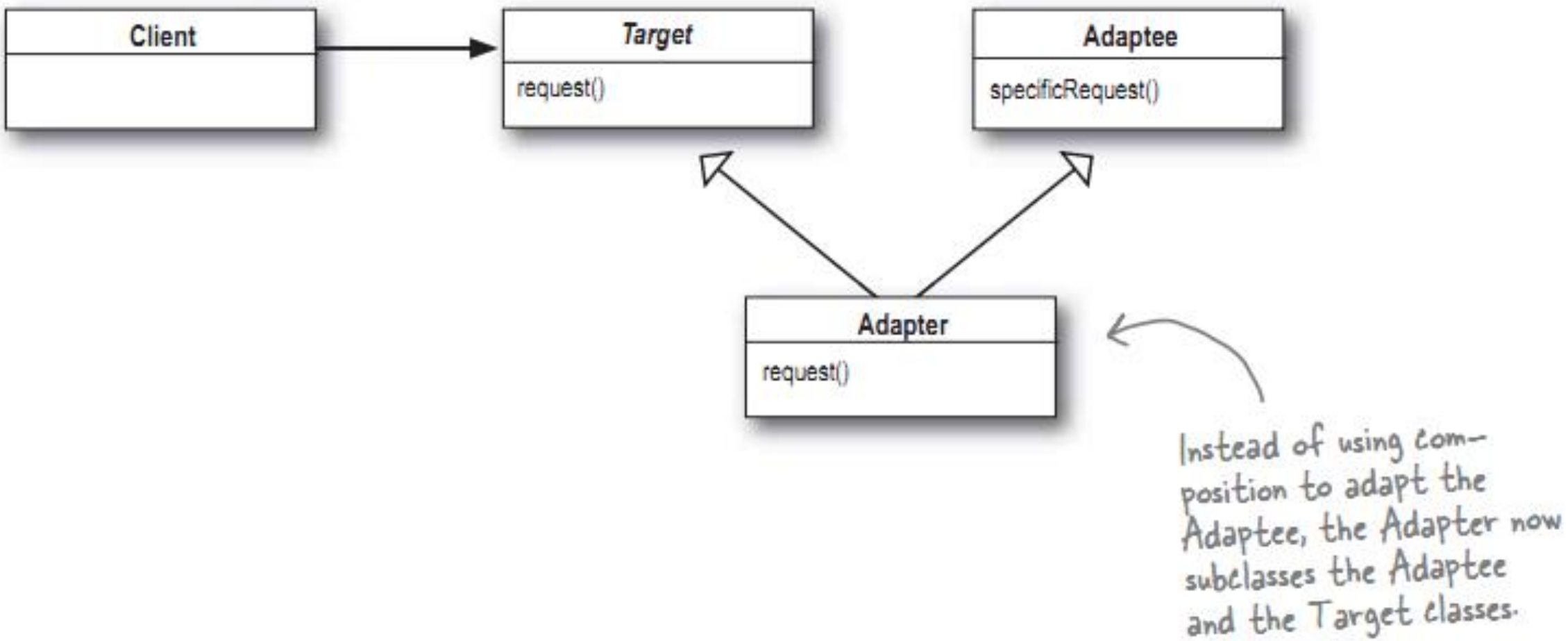

How are you?
I am Fine.
Comment allez-vous?
Je suis tres bien
John
From USA and can speak
and understand only English
Pam
Translator
or
Adapter
David
From France and can speak
and understand only French

The Adapter Pattern نمط المحول

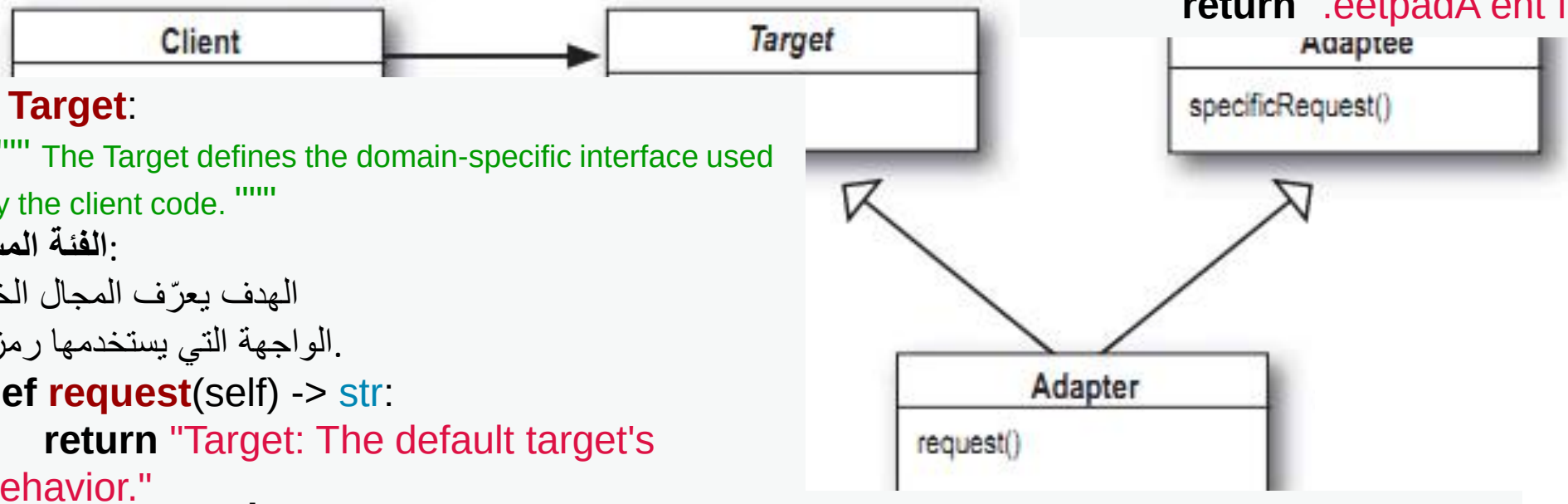
- Converts the interface of a class into another interface the client expect. Adapter Design Pattern works as a bridge between two incompatible interfaces. This design pattern involves a single class called **adapter** which is responsible for **communication** between two independent or incompatible interfaces.

يحول واجهة فئة إلى واجهة أخرى العميل
توقع .يعمل نمط تصميم
المحول كجسر بين اثنين
واجهات غير متوافقة .يتضمن نمط
التصميم هذا فئة واحدة
يسمى محول وهو المسؤول
عن الاتصال بين اثنين
واجهات مستقلة أو غير متوافقة.

1- Class Adapter فئة المحول:



1- Class Adapter فئة المحول:




```
class Adaptee:
    """ The Adaptee contains some useful behavior, but its interface is
    incompatible with the existing client code. The Adaptee needs some
    adaptation before the client code can use it. """
```

يحتوي Adaptee ، على بعض السلوكيات المفيدة
لكن واجهته غير متوافقة مع ملفات
إلى بعض التكيف Adaptee يحتاج .رمز العميل
قبل أن يتمكن رمز العميل من استخدامه. """

```
    def specific_request(self) -> str:
        return ".eetpadA eht fo roivaheb laicepS"
```

```
class Target:
    """ The Target defines the domain-specific interface used
    by the client code. """
```

الفئة المستهدفة:

""" الهدف يعرّف المجال الخاص
الواجهة التي يستخدمها رمز العميل.

```
    def request(self) -> str:
        return "Target: The default target's
        behavior."
```

```
class Adapter(Target, Adaptee):
    """ The Adapter makes the Adaptee's interface compatible with the Target's interface via multiple
    inheritance. """
```

إرجاع "الهدف: سلوك الهدف الافتراضي".
Target متوافقة مع Adaptee المحول يجعل واجهة "":(الهدف ، المحول)**محول الفئة**
واجهة عبر الوراثة المتعددة

```
    def request(self) -> str:
        return f"Adapter: (TRANSLATED) {self.specific_request()[::-
```

```
def client_code(target: "Target") -> None:
    """ The client code supports all classes that follow the Target interface.
    """
    print(target.request(), end="")
if __name__ == "__main__":
     print("Client: I can work just fine with the Target objects:")
     target = Target()
    client_code(target)
    print("\n")
     adaptee = Adaptee()
     print("Client: The Adaptee class has a weird interface. " "See, I don't
    understand it:")
    print(f"Adaptee: {adaptee.specific_request()}", end="\n\n")
    print("Client: But I can work with it via the Adapter:")
    adapter = Adapter()
    client_code(adapter)
```

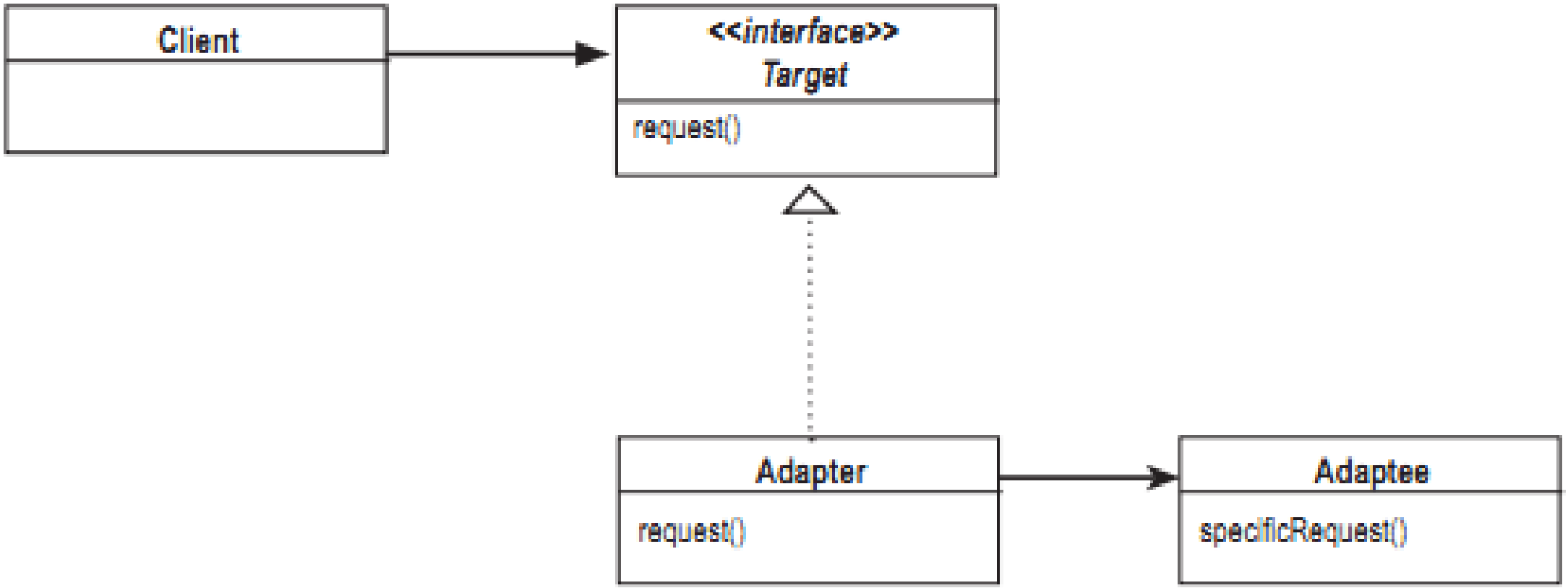
Output

```
Client: I can work just fine with the
Target objects:
Target: The default target's behavior.

Client: The Adaptee class has a weird
interface. See, I don't understand it:
 Adaptee: .eetpadA eht fo roivaheb
laicepS

 Client: But I can work with it via the
Adapter:
Adapter: (TRANSLATED) Special
behavior of the Adaptee.
```

اخراج|

```
العميل: يمكنني العمل بشكل جيد مع
الكائنات المستهدفة:
الهدف: سلوك الهدف الافتراضي.
العميل: فئة Adaptee لها واجهة
غريبة .انظر ، لا أفعل
فهمته:
التكيف eetpadAeht fo. :
roivaheb laicepS
العميل: لكن يمكنني العمل معه عبر
المحول:
المحول: (مترجم) سلوك خاص
للمكيف.
```

2- Object Adapter

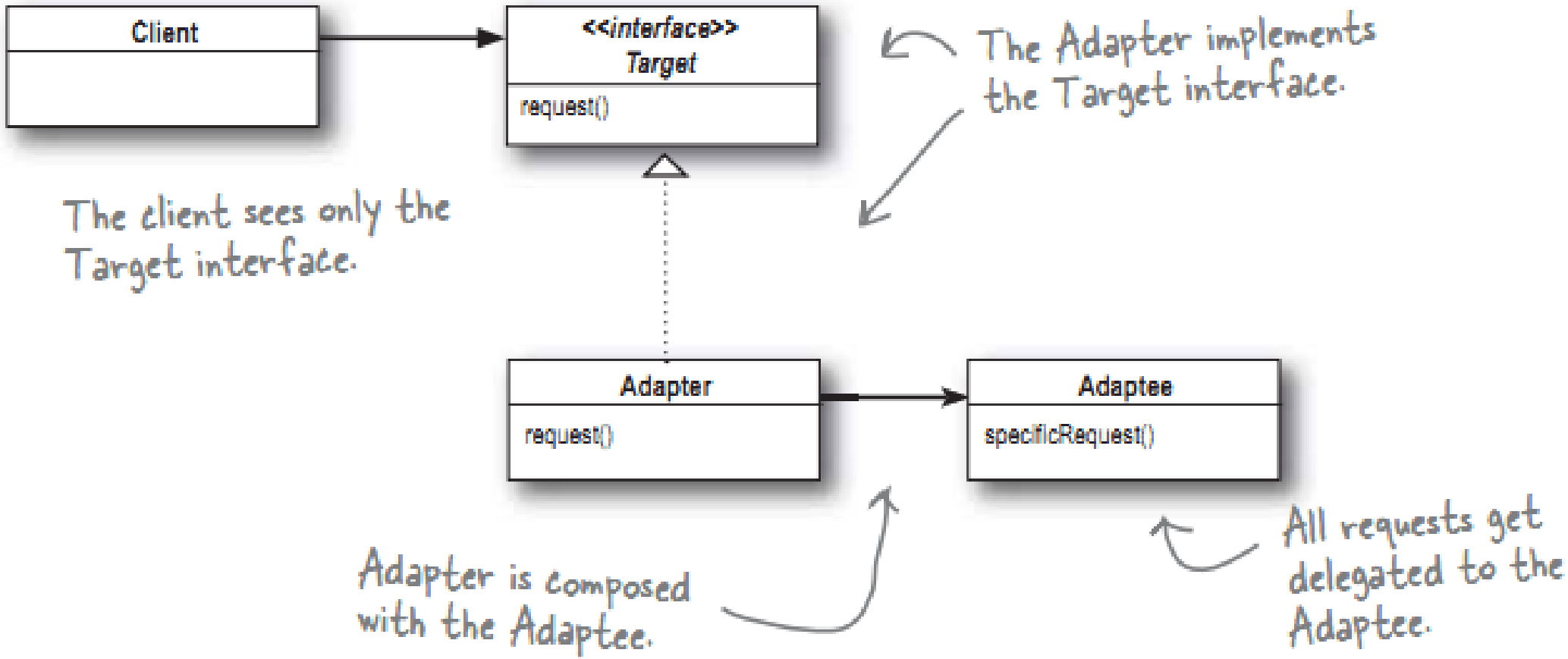


2- Object Adapter

– A class implements an interface

– Inheritance -> IS relationship

– Composition -> HAS relationship



- """ Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces. """ قم بتحويل واجهة الفصل إلى واجهة أخرى يتوقعها العملاء. يتيح المحول للفئات العمل معًا ولكن لا يمكن ذلك خلاف ذلك بسبب واجهات غير متوافقة.

```
import abc
class Target(metaclass=abc.ABCMeta):
    """ Define the domain-specific interface that Client uses. تحديد الواجهة الخاصة بالمجال التي يستخدمها العميل
    def __init__(self):
        self._adaptee = Adaptee()
        @abc.abstractmethod
    def request(self):
        pass
class Adapter(Target):
    """ Adapt the interface of Adaptee to the Target interface. تكييف واجهة Adaptee مع واجهة الهدف """
    def request(self):
        self._adaptee.specific_request()
class Adaptee:
    """ Define an existing interface that needs adapting .تحديد واجهة موجودة تحتاج للتكيف """
    def specific_request(self):
        pass
def main():
    adapter = Adapter()
    adapter.request()
if __name__ == "__main__":
        main()
```

- The purpose of the ***ITarget*** interface is to enable objects of ***adaptee*** types to be interchangeable with any other objects that might implement the same interface. However, the ***adaptees*** might not conform to the operation names and signatures of ***ITarget***, so an interface alone is not a sufficiently powerful mechanism. That is why we need the Adapter pattern. An ***Adaptee*** offers similar functionality to Request, but under a different name and with possibly different parameters. The ***Adaptee*** is completely independent of the other classes and is oblivious to any naming conventions or signatures that they have. Now, let’s consider the roles in the pattern:

الغرض من واجهة واجهة الهدف لتمكين الكائنات من التهيئة التبادل مع أي كائن آخر قد يقوم بتطبيق نفس الواجهة.
ومع ذلك ، قد لا يتوافق المقبوض عليهم مع أسماء العملية و توقيعات الهدف ، لذا فإن الواجهة البينية ليست قوية بما فيه الكفاية
آلية .هذا هو السبب في حاجة إلى نمط المحول
وظائف للطلب ، ولكن تحت اسم مختلف وربما مختلف العوامل .يكون المحول مستقلاً تمامًا عن الفئات الأخرى وهو كذلك
عدم الموافقة على أي اشتراكات أو توقيعات لها .دعنا الآن ضع في اعتبارك الثيرولين في النمط

ITarget

The interface that the Client wants to use

Adaptee

An implementation that needs adapting

Adapter

The class that implements the ITarget interface in terms of the Adaptee

Request

An operation that the Client wants

SpecificRequest

The implementation of Request’s functionality in the Adaptee

IT الهدف

الواجهة التي يريد العميل استخدامها

التكيف

تنفيذ يحتاج إلى التكيف

Adapter

الفئة التي تنفذ واجهة IT المستهدفة من حيث Adaptee

طلب

عملية يريدها العميل

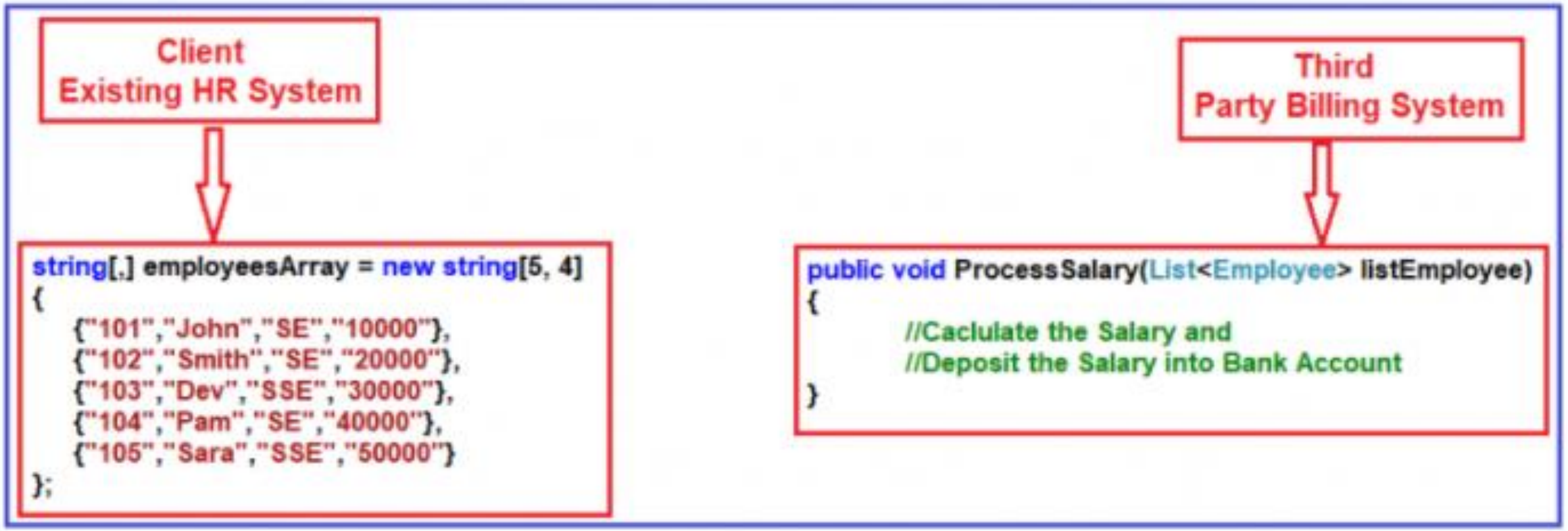
طلب محدد

تنفيذ وظيفة الطلب في التكيف

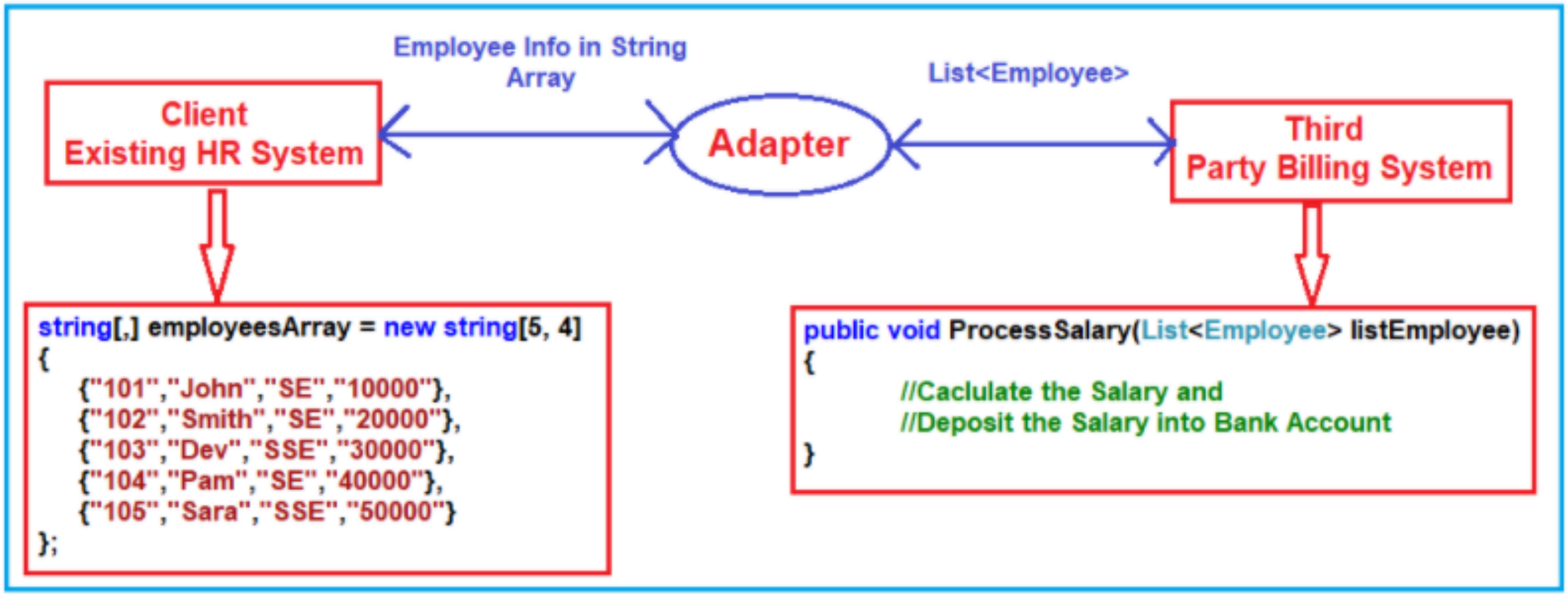
Adapter Examples

أمثلة المحول

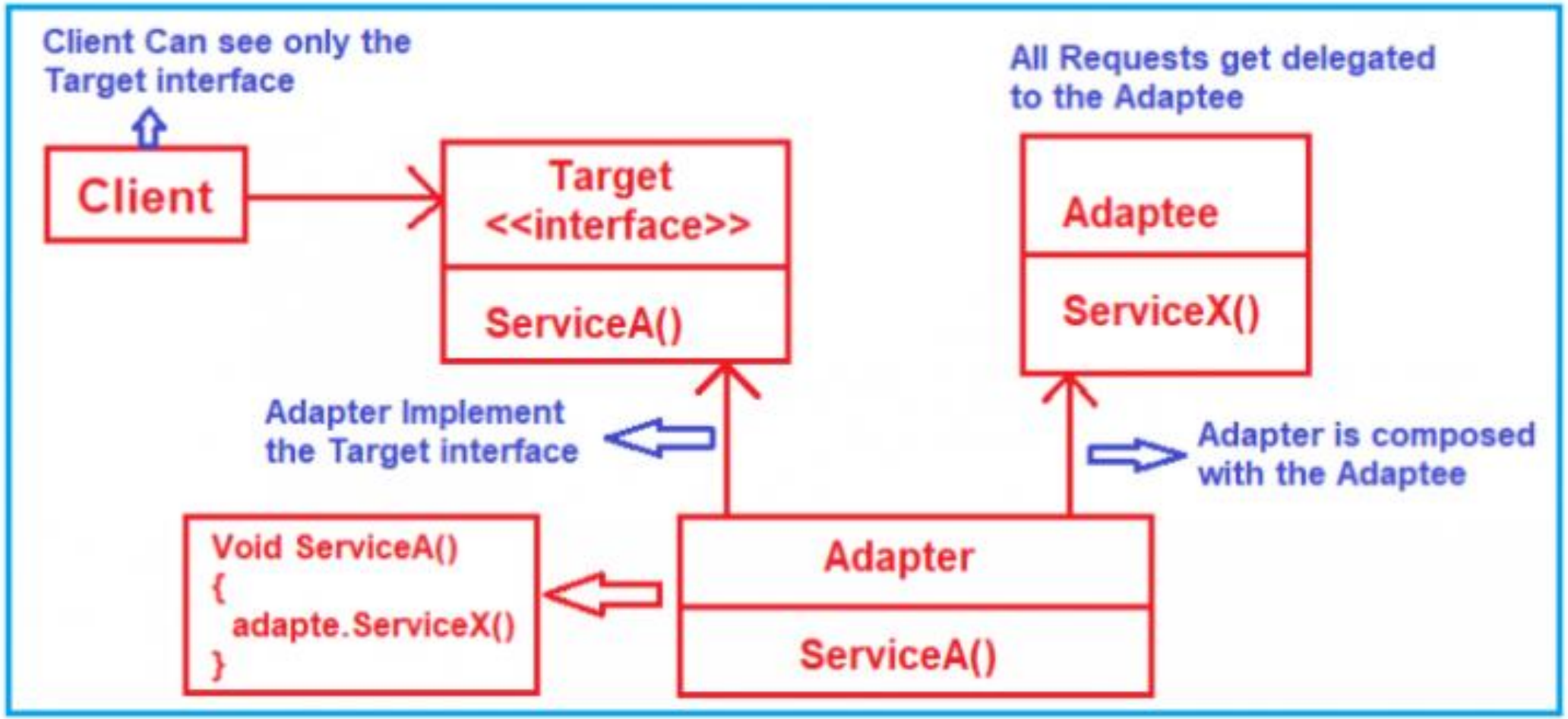
Example : HR System- Party Billing System



Example : HR System- Party Billing System



Example : HR System- Party Billing System



Step1: Creating Employee class

Note: Code in C# Language
Try to change it to Python language

```
namespace AdapterDesignPattern
{
    public class Employee
    {
        public int ID { get; set; }
        public string Name { get; set; }
        public string Designation { get; set; }
        public decimal Salary { get; set; }

        public Employee(int id, string name, string designation, decimal salary)
        {
            ID = id;
            Name = name;
            Designation = designation;
            Salary = salary;
        }
    }
}
```

Step2: Creating Adaptee

```
using System;
using System.Collections.Generic;
namespace AdapterDesignPattern
{
    public class ThirdPartyBillingSystem
    {
        //ThirdPartyBillingSystem accepts employees information as a List to process
each employee salary
        public void ProcessSalary(List<Employee> listEmployee)
        {
            foreach (Employee employee in listEmployee)
            {
                Console.WriteLine("Rs." +employee.Salary + " Salary Credited to " +
employee.Name + " Account");
            }
        }
    }
}
```

Step3: Creating Target interface

```
namespace AdapterDesignPattern
{
    public interface ITarget
    {
        void ProcessCompanySalary(string[,] employeesArray);
    }
}
```

Step4: Creating Adapter

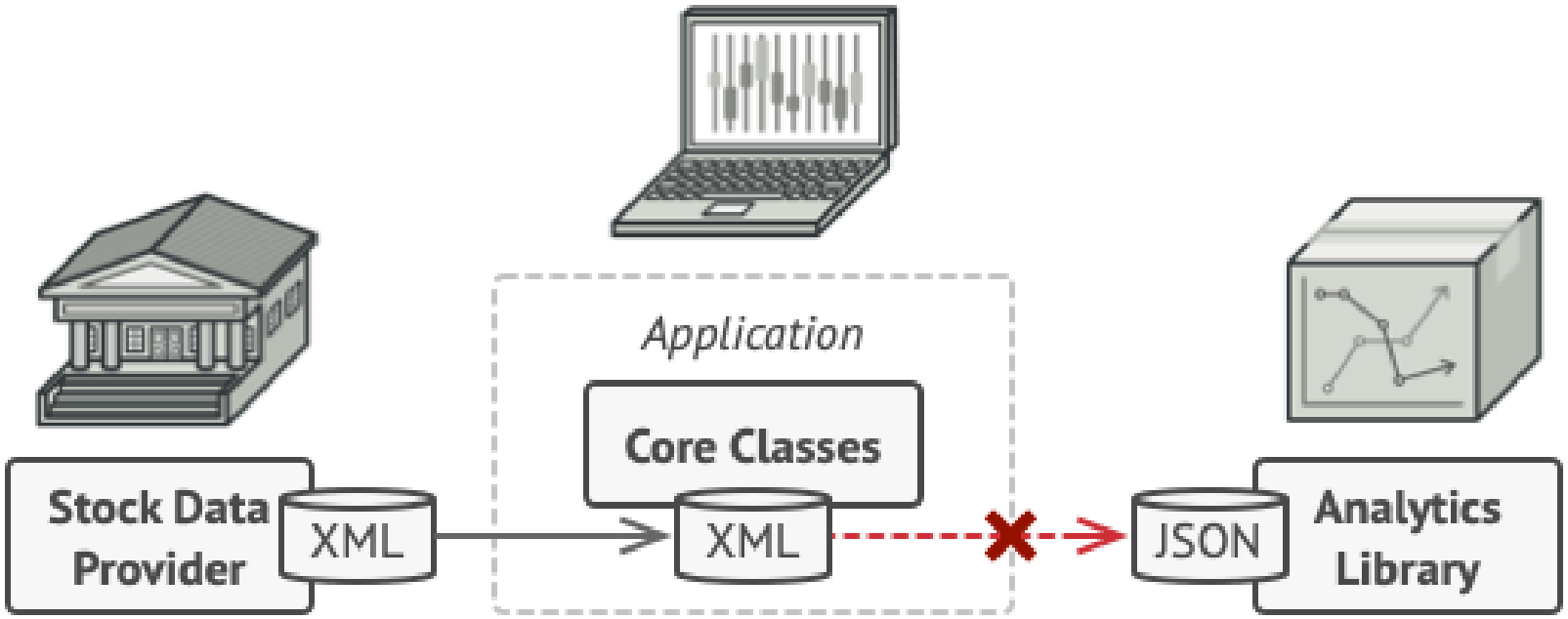
```
{
    public class EmployeeAdapter : ITarget
    {
        ThirdPartyBillingSystem thirdPartyBillingSystem = new ThirdPartyBillingSystem();

        public void ProcessCompanySalary(string[,] employeesArray)
        {
            string Id = null;
            string Name = null;
            string Designation = null;
            string Salary = null;

            List<Employee> listEmployee = new List<Employee>();

            for (int i = 0; i < employeesArray.GetLength(0); i++)
            {
                for (int j = 0; j < employeesArray.GetLength(1); j++)
                {
                    if (j == 0)
                    {
                        Id = employeesArray[i, j];
                    }
                    else if (j == 1)
                    {
                        Name = employeesArray[i, j];
                    }
                    else if (j == 1)
                    {
                        Designation = employeesArray[i, j];
                    }
                    else
                    {
                        Salary = employeesArray[i, j];
                    }
                }
```

```
                listEmployee.Add(new Employee(Convert.ToInt32(Id), Name, Designation,
Convert.ToDecimal(Salary)));
            }

            Console.WriteLine("Adapter converted Array of Employee to List of
Employee");
            Console.WriteLine("Then delegate to the ThirdPartyBillingSystem for
processing the employee salary\n");
            thirdPartyBillingSystem.ProcessSalary(listEmployee);
        }
    }
}
```

Step 5: Client

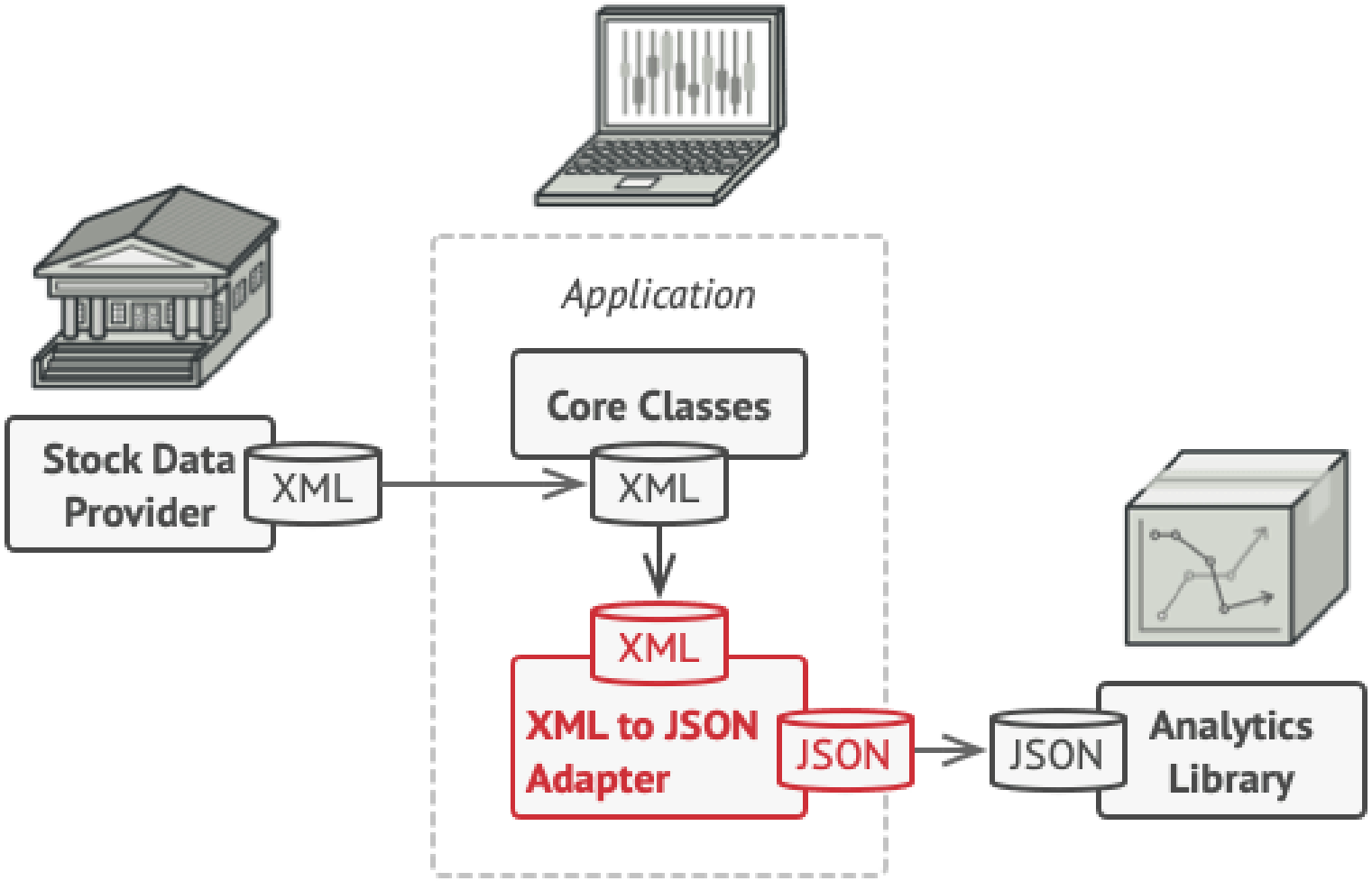
```
namespace AdapterDesignPattern
{
    class Program
    {
        static void Main(string[] args)
        {
            string[,] employeesArray = new string[5, 4]
            {
                {"101","John","SE","10000"},
                {"102","Smith","SE","20000"},
                {"103","Dev","SSE","30000"},
                {"104","Pam","SE","40000"},
                {"105","Sara","SSE","50000"}
            };

            ITarget target = new EmployeeAdapter();
            Console.WriteLine("HR system passes employee string array to Adapter\n");
            target.ProcessCompanySalary(employeesArray);

            Console.Read();
        }
    }
}
```

Example: XML-JSON



You can't use the analytics library "as is" because it expects the data in a format that's incompatible with your app.

Example : XML-JSON



Proxy Pattern

- **Proxy** is a structural design pattern that lets you provide 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.

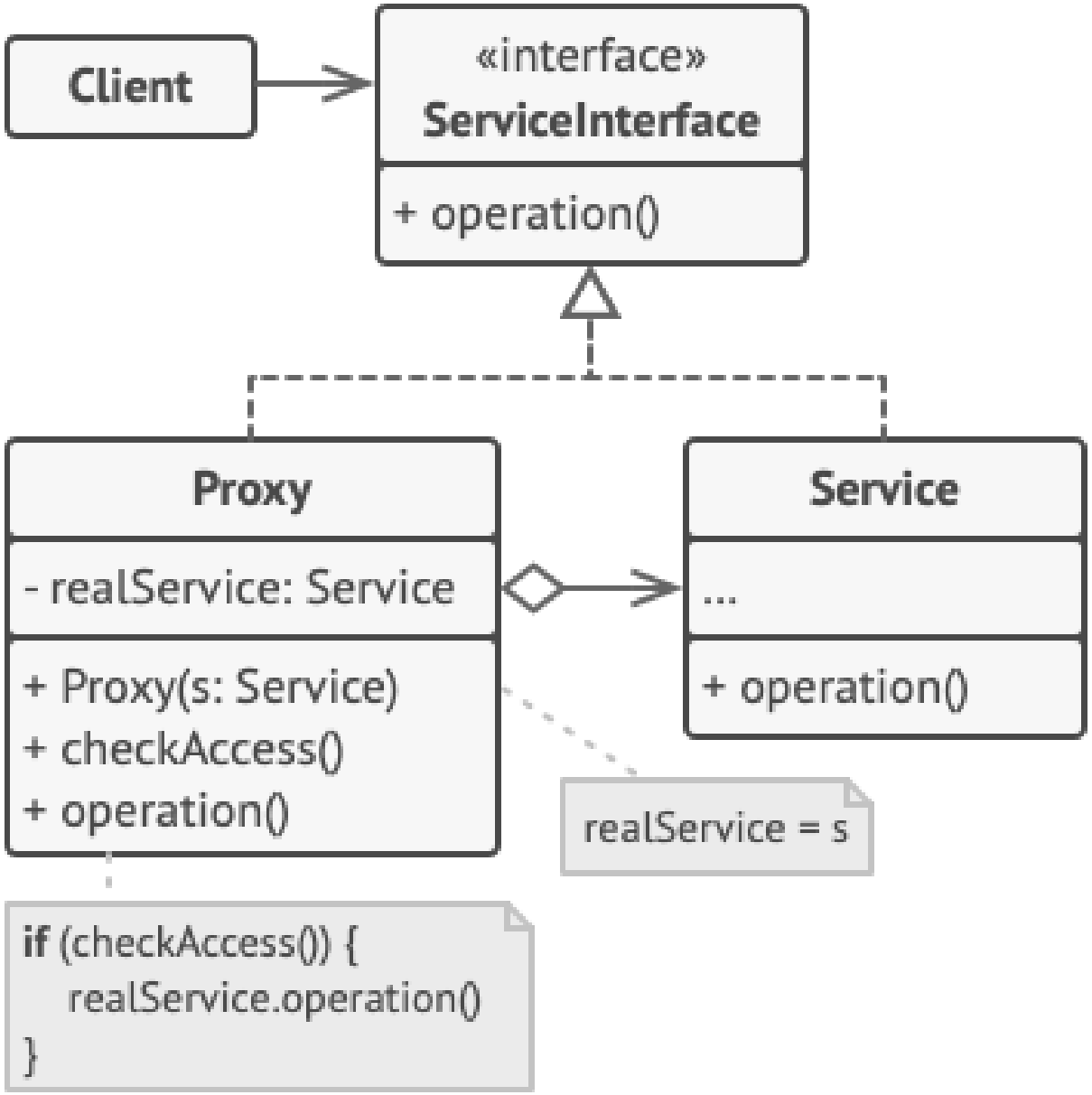
الوكيل هو نمط تصميم هيكلي يتيح لك تقديم بديل
أو عنصر نائب لكائن
آخر .يتحكم الوكيل في الوصول إلى
الكائن الأصلي ، مما يسمح لك بأداء
شيء ما من قبل أو
بعد ذلك ، يمكنك الوصول إلى الكائن
الأصلي.

You can use the proxy design pattern for.
يمكنك استخدام نمط تصميم ثيبر وكسيدي

- Adding security access to an existing object. The proxy will determine if the client can access the object of interest.
- Simplifying the API of complex object. The proxy can provide a simple API so that the client code does not have to deal with the complexity of the object of interest.
- Providing interfaces for remote resources such as web service or REST resources.

.
•إضافة الأمان إلى مشروع موجود. سيتم تحديد المشروع
إذا استطاع العميل التعرف على موضوع الاهتمام.
•تبسيط .APIof complexobject يمكن
أن يقدم proxycan بسيط
APIبحيث لا يتعين على العميل المشفر
التعامل مع التعقيد
من موضوع المصلحة.
•توفير واجهات للمنازل البعيدة مثل خدمة الويب أوREST
مصادر.

Proxy pattern



Types of proxies

- **Virtual Proxy:** A virtual proxy is a place holder for “expensive to create” objects. The real object is only created when a client first requests or accesses the object.
- **Remote Proxy:** A remote proxy provides local representation for an object that resides in a different address space.
- **Protection Proxy:** A protection proxy control access to a sensitive master object. The surrogate object checks that the caller has the access permissions required prior to forwarding the request.

• Virtual Proxy: عنصر نائب ظاهري لـ "proxyisa–لـexpensiveto"
خلق "كائنات .يتم إنشاء الكائن Thereal فقط عندما يكون العميل أولاً
الطلبات أو الوصول إلى الكائن.
• RemoteProxy: وكيل عن بُعد يوفر تمثيلًا محليًا لملف
الكائن الموجود في مساحة عنوان مختلفة.
•وكيل الحماية: وهو وكيل حماية للتحكم في الوصول إلى حساس
الكائن الرئيسي .هذا الكائن البديل يتحقق من استعجال المتصل
accesspermissionsالمطلوبة قبل إعادة توجيه الطلب.

Remote Proxy

- A *remote proxy* is a local object, a copy, that looks and acts like the "real" object which is situated at a remote site.
- The remote proxy is used to minimize slow, long-distance communications. It is also used to hide the technical details of making the long-distance communications.
- Consider a banking example, where an ATM and its controller must communicate with the bank's database to login and do transactions on an account.

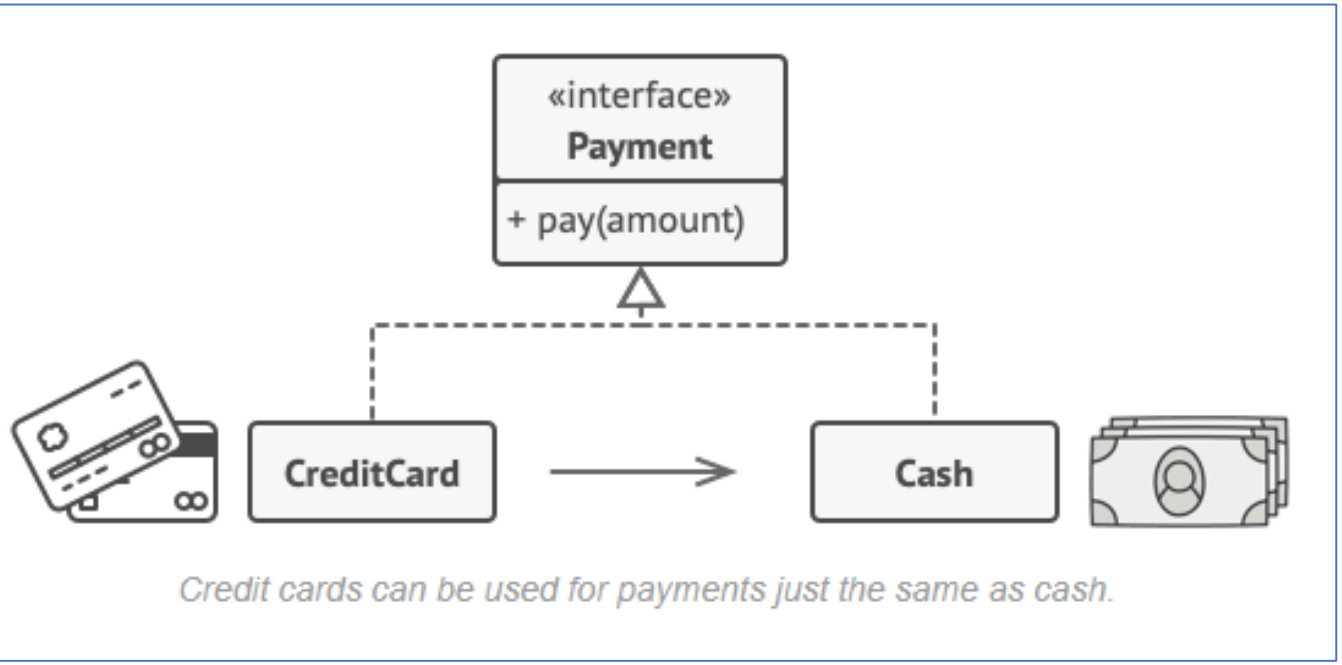
الوكيل البعيد هو كائن محلي ، نسخة ، تبدو وكأنها حقائق
كائن "حقيقي" تم طرحه في موقع بعيد.
•يتم استخدام وكيل Theremote لتقليل المسافات الطويلة والبعيدة
الاتصالات.كما تستخدم لإخفاء التفاصيل الفنية للعملية
الاتصالات بعيدة المدى.
•النظر في مثال مصرفي ، حيث يجب أن تكون أجهزة الصراف الآلي ومراقبها
التواصل مع قاعدة بيانات البنك لتسجيل الدخول والمعاملات الرقمية
حساب.onan

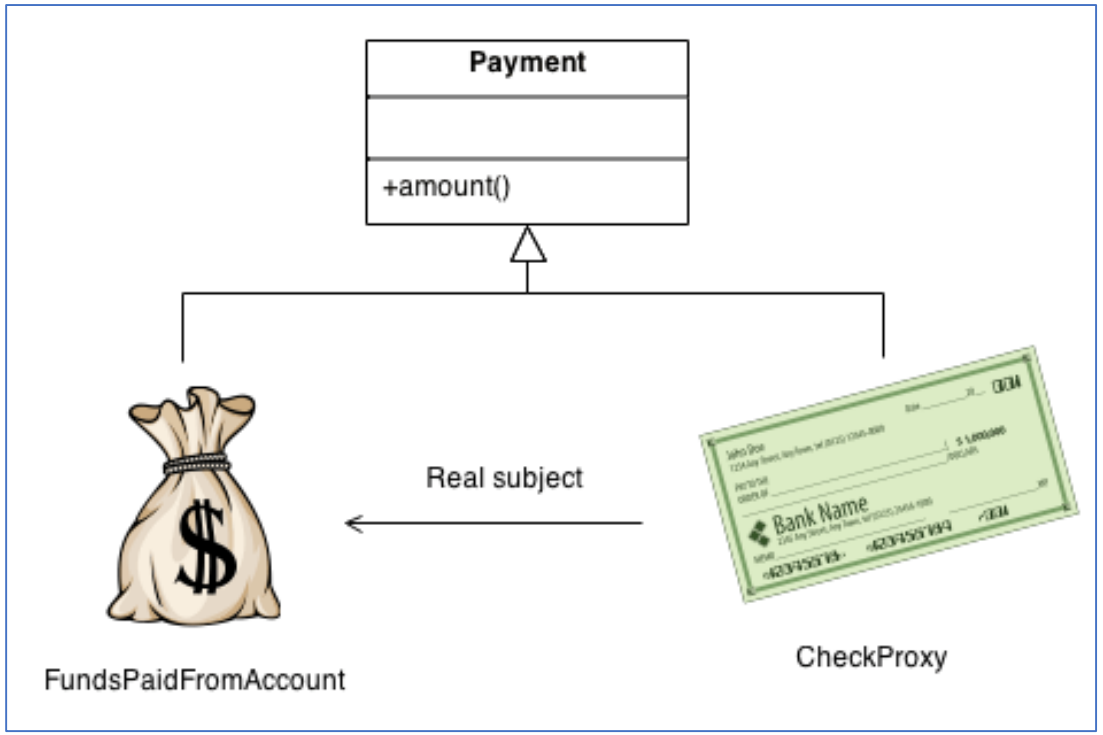
Remote Proxy

- A credit card is a proxy for a bank account, which is a proxy for a bundle of cash. Both implement the same interface: they can be used for making a payment. A consumer feels great because there’s no need to carry loads of cash around. A shop owner is also happy since the income from a transaction gets added electronically to the shop’s bank account without the risk of losing the deposit or getting robbed on the way to the bank.

وكيل بطاقة ائتمان لحساب مصرفي ، وهو وكيل لـ
حزمة نقدية .كلاهما ينفذ نفس الواجهة: يمكن استخدامها
لدفع المبلغ .المستهلك يشعر بالارتياح لأنه لا يوجد
يجب أن تحمل حمولات نقدية حولها
يتم إضافتها إلكترونيًا إلى المتجر
حساب مصرفي دون مخاطر فقدان الوديعة أو التعرض للسرقة
على البنك.

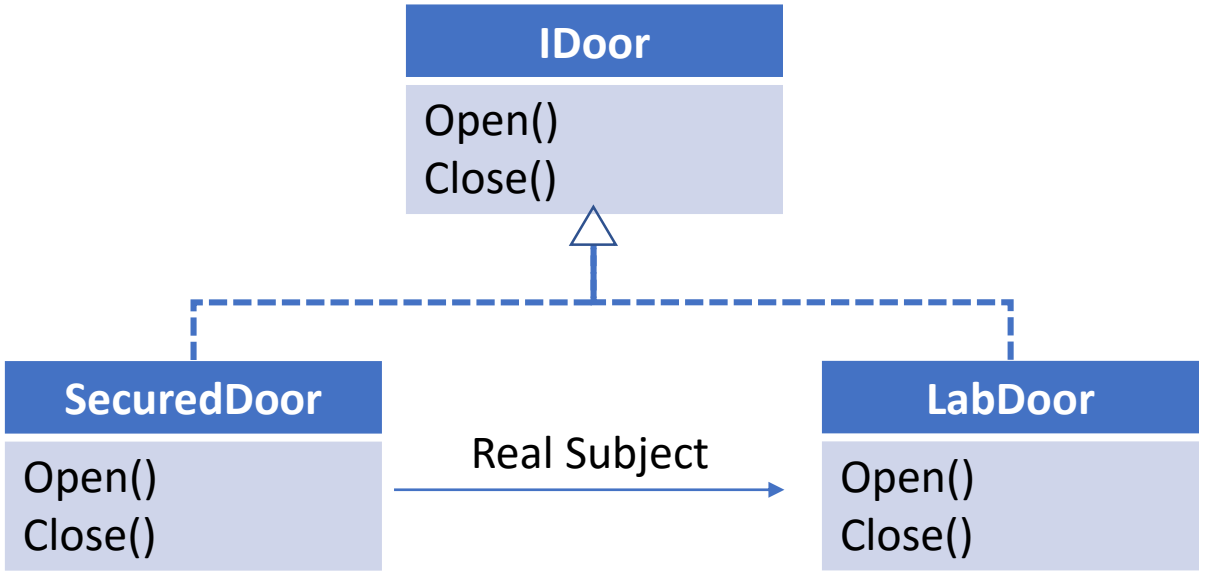
Remote Proxy



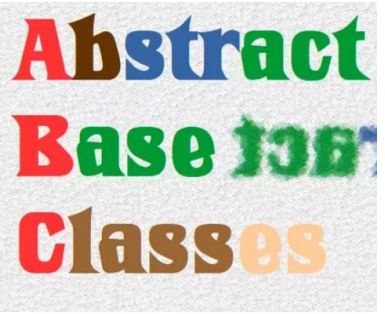


Protection Proxy: Door Example

```
import abc
#Interface
class Idoor(metaclass=abc.ABCMeta):
@abc.abstractmethod
      def open(self):
            pass
@abc.abstractmethod
      def close(self):
            pass
#Real Class
class Labdoor ( Idoor):
     def close(self):
          print ”Closing lab door”

     def open(self):
          print "Opening lab door"
```



Note: import abc



Proxy: Door Example

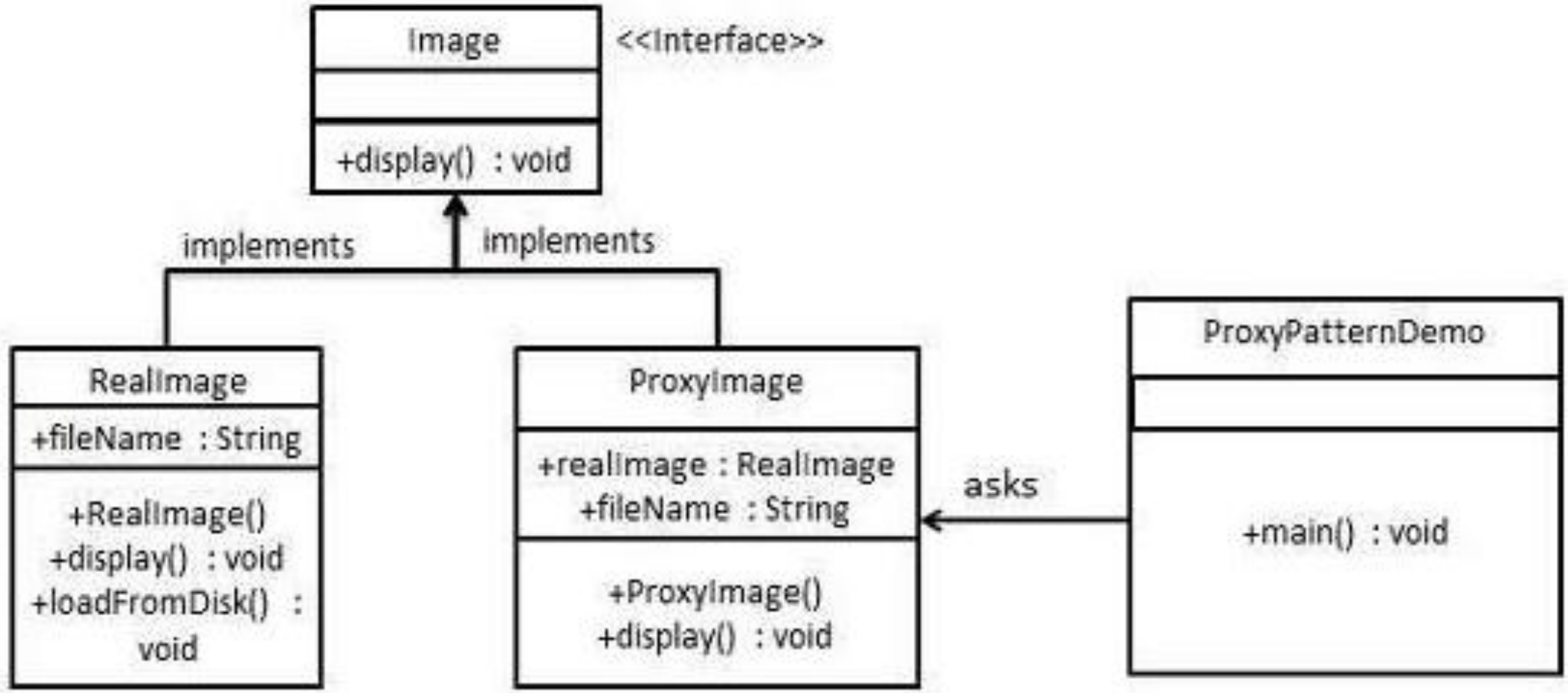
```
#Proxy Class
class Secured_door ( Idoor):
    def __init__(self, door):
        self. __door = door;
```

```
    def open(self, password):
        if  self.authenticate(password):
                self. __door.open()
        else:
                print”Big no! It ain't possible."

    def authenticate(self,password):
            return password == "$ecr@t“
    def close(self):
            self. __door.close()
```

Proxy: Door Example

```
#client
labdoord= Labdoor()
door = SecuredDoor(labdoor)
door.open("invalid") // Big no! It ain't possible.

door.open("$ecr@t") // Opening lab door
door.close() // Closing lab door
```

Virtual Proxy pattern: Image Example



```
#Interface
Class Image:
    def display(self):
        pass
```

```
#Real Class
Class Realimage ( Image ):

    def __init__(self, filename):
            self.filename = filename
            self.loadfromdisk(self.filename)

    def display(self) :
        print"Displaying "
        print( self.filename)

    def loadfromdisk(filename):
        print"Loading "
         print( filename)
```

```
#proxy
Class Proxyimage (Image):
    def __init__(self, realimage):
        self.__realimage=realimage
    def display(self) :
        if not exist self.__realimage :
            self.__realImage =  Realimage(self.__realimage .filename)
        self.__realimage.display();
```

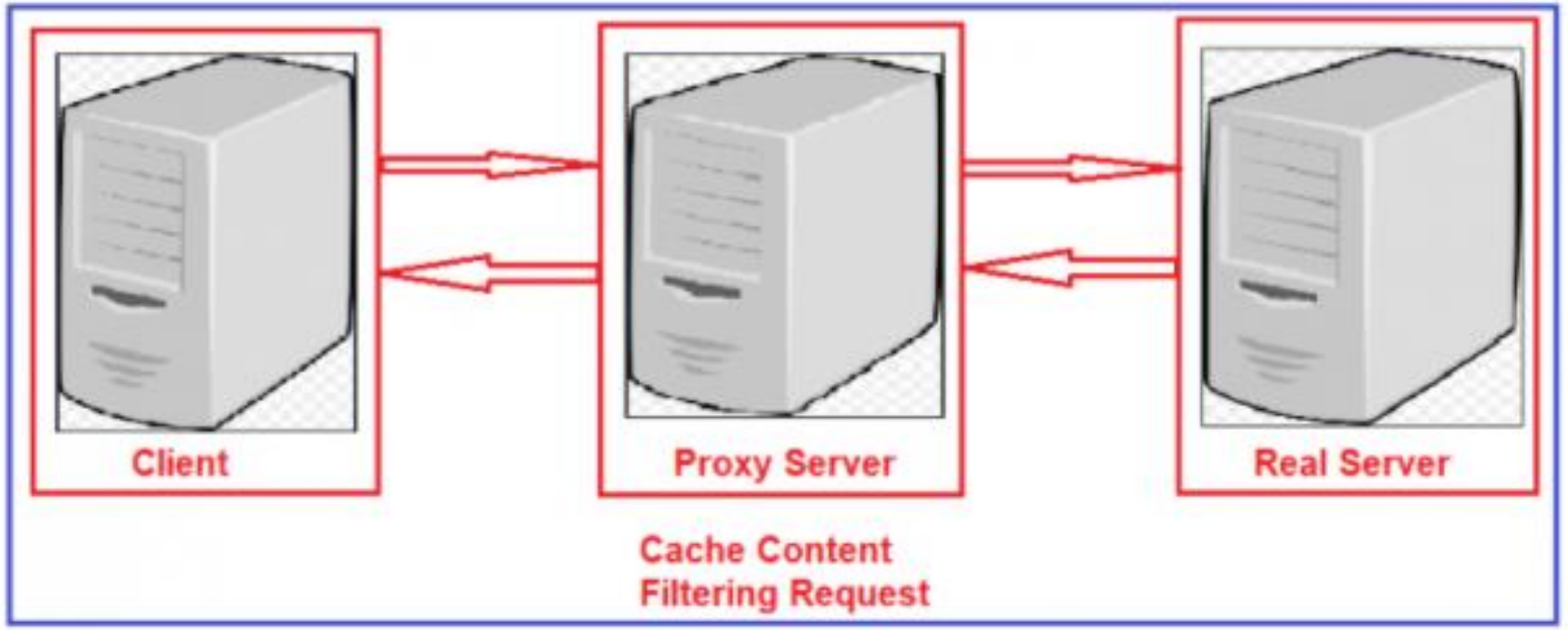
```
#client
Class Proxypatterndemo :
        realimage=Realimage ("test_10mb.jpg")
       image = Proxyimage(realimage)

        #image will be loaded from disk
        image.display()

        #image will not be loaded from disk
        image.display()
```

Examples Of Proxy

Protection Proxy pattern: Block Sites Example

Filter Requests:

The Proxy servers can also be used to filter the incoming requests. For example, a company might use the proxy server to prevent its employees from accessing a specific set of websites such as Facebook, Twitter, etc.



```
Class Internet:
        def grant_internet_access(site):
            pass

  Class Real_internet (Internet):

       def grant_internet_access(site):

           print"Internet allowed "
```

```
Class Proxy_internet (Internet):

    blocksites = [www.facebook.com, "www.gmail.com"]
    def __init__(self):
        pass
    def grant_internet_access( site):

      if site  in blocksites:
                print "Internet is not allowed”

      else:
                r = Real_internet()
                r.grant_internet_access(site)
```

```
Internet internet =  Proxy_internet()
internet.grant_internet_access("www.dotnetglance.com")
internet.grant_internet_access("www.facebook.com")
```

Business Requirement:

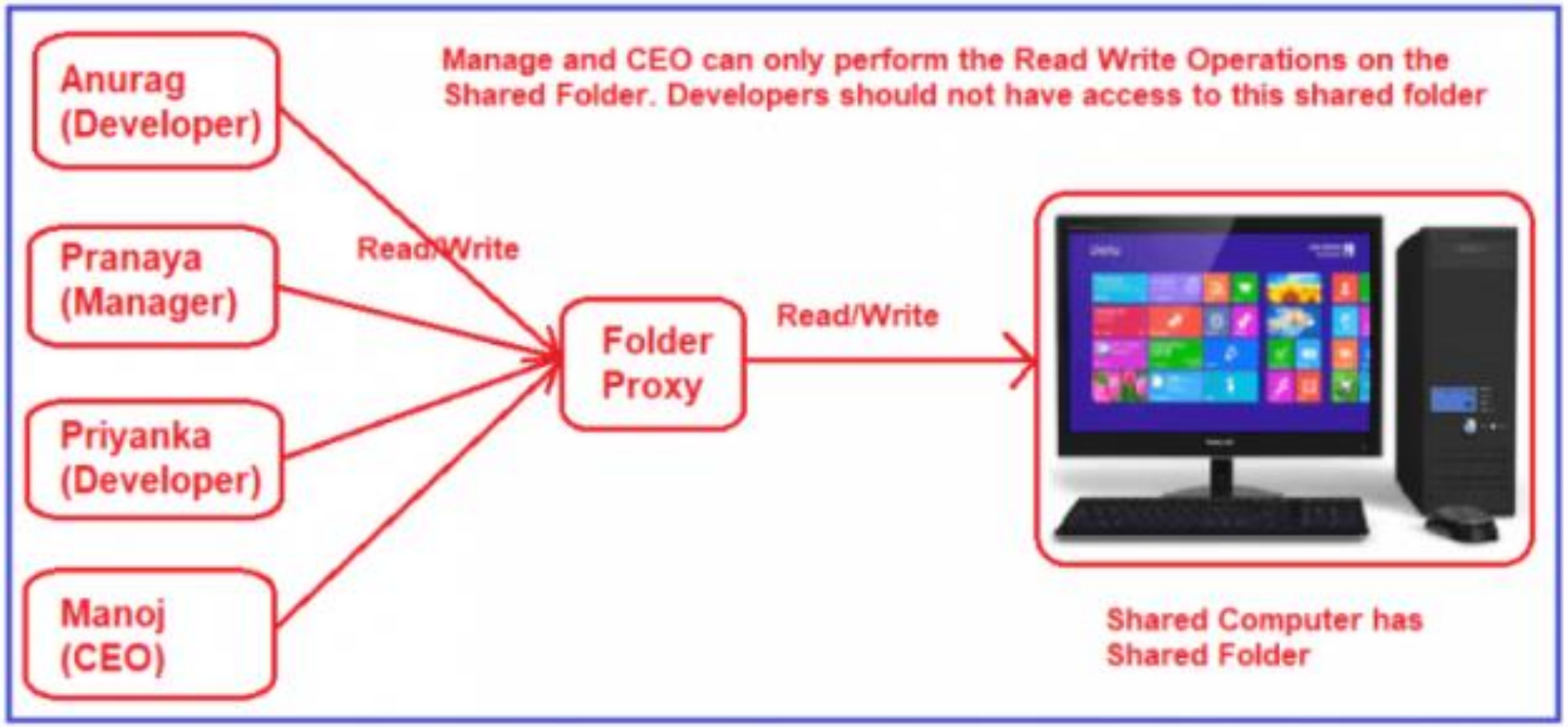
Please have a look at the following diagram. As you can see in the following image, on the right side we have a shared computer that has a shared folder. On the left side, we have employees who are working on a software farm. The shared computer contains a shared folder that contains confidential information and only the employee having role Manager and CEO can access this shared folder and perform the read-write operations. On the other hand, if the employee is a developer, then it should not allow access to the shared folder. That is we need to do some kind of protection. In scenarios like this, the Protection Proxy can be handy.

What we can do here is, in between the employees and the shared computer we need to introduce the Folder Proxy. What this Folder proxy can do is, it will check if the employee role is Manager or CEO, then it allows the employee to access the shared folder and perform the read-write operation. On the other hand, if the employee role is Developer then it will say you don't have permission to access this folder. That kind of protection logic we can write in the folder proxy.

متطلبات العمل:

الرجاء إلقاء نظرة على الرسم البياني التالي .كما ترى في ما يلي الصورة ،على الجانب الأيمن لدينا جهاز كمبيوتر مشترك به ملف مجلد .على الجانب الأيسر ، لدينا موظفون يعملون على برنامج مزرعة .يحتوي الكمبيوتر المشترك على مجلد مشترك يحتوي على ملفات
المعلومات السرية والموظف الوحيد الذي له دور مدير و يمكن لـ CEO الوصول إلى هذا المجلد المشترك وإجراء عمليات القراءة والكتابة.
من ناحية أخرى ، إذا كان الموظف مطورًا ، فلا ينبغي أن يسمح بذلك
الوصول إلى المجلد المشترك.هذا هو أننا بحاجة إلى القيام بنوع من الحماية.
في مثل هذه السيناريوهات ، يمكن أن يكون وكيل الحماية مفيدًا.
ما يمكننا القيام به هنا هو ، بين الموظفين والمشتركين
كمبيوتر نحتاج إلى إدخال وكيل المجلد .ما هذا الوكيل المجلد يمكن أن يفعل هو ، سيتحقق مما إذا كان دور الموظف هو المدير أو الرئيس التنفيذي ، ثم هو يسمح للموظف بالوصول
إلى المجلد المشترك وإجراء القراءة عملية الكتابة .من ناحية أخرى ، إذا كان دور -الموظف هو المطور
ثم سيقول أنه ليس لديك إذن للوصول إلى هذا المجلد .هذا النوع من منطق الحماية يمكننا الكتابة في وكيل المجلد.

Protection Proxy pattern: File Sharing Example



1.Subject (ISharedFolder): This is an interface that defines members that are going to be implemented by the RealSubject and Proxy class so that the Proxy can be used anywhere the RealSubject is expected. In our example, it is the ISharedFolder interface.
2.RealSubject (SharedFolder): This is a class that we want to use more efficiently by using the proxy class. In our example, it is the SharedFolder class.
3.Proxy (SharedFolderProxy): This is a class that holds a reference of the RealSubject class and can access RealSubjecr class members as required. It must implement the same interface as the RealSubject so that the two can be used interchangeably. In our example, it is the SharedFolderProxy class.

١الموضوع :(ISharedFolder) هذه واجهة تحدد الأعضاء
سيتم تنفيذها بواسطة فئة RealSubject و Proxy بحيث يتم تنفيذ الامتداد
يمكن استخدام الوكيل في أي مكان يتوقع فيه RealSubject.في مثالنا ، هو هي واجهةISharedFolder.
2. RealSubject (SharedFolder): هذه فئة نريد استخدامها أكثر بكفاءة باستخدام فئة الوكيل .في مثالنا ، هو SharedFolder صف دراسي.
3.الوكيل :(SharedFolderProxy) هذه فئة تحتوي على مرجع لملف فئة RealSubject ويمكن الوصول إلى أعضاء فئة RealSubjecrكما هو مطلوب .هو - هي يجب أن تنفذ نفس الواجهة مثل RealSubject بحيث يمكن أن يكونا استعمل بشكل تبادلي .في مثالنا ، إنها فئة

Step1: Creating the Employee class الخطوة ١: إنشاء فئة الموظف

```
namespace ProxyDesignPattern
{
    public class Employee
     {
         public string Username { get; set; }
         public string Password { get; set; }
         public string Role { get; set; }

         public Employee(string username, string password, string role)
         {
             Username = username;
             Password = password;
             Role = role;
         }
     }
}
```

Step2: Creating Subject الخطوة ٢: إنشاء الموضوع

```
using System;
namespace ProxyDesignPattern
{
    public interface ISharedFolder
    {
        void PerformRWOperations();
    }
}
```

Step3: Creating Real Object Step3: إنشاء كائن حقيقي

```
using System;
namespace ProxyDesignPattern
{
    public class SharedFolder : ISharedFolder
    {
        public void PerformRWOperations()
        {
            Console.WriteLine("Performing Read Write operation on the Shared Folder");
        }
    }
}
```

Step4: Creating the Proxy Object الخطوة ٤: إنشاء كائن الوكيل

```
using System;
namespace ProxyDesignPattern
{
    class SharedFolderProxy : ISharedFolder
    {
        private ISharedFolder folder;
        private Employee employee;

        public SharedFolderProxy(Employee emp)
        {
            employee = emp;
        }

        public void PerformRWOperations()
        {
            if (employee.Role.ToUpper() == "CEO" || employee.Role.ToUpper() =="MANAGER")
            {
                folder = new SharedFolder();
                Console.WriteLine("Shared Folder Proxy makes call to the RealFolder
'PerformRWOperations method'");
                folder.PerformRWOperations();
            }
            else
            {
                Console.WriteLine("Shared Folder proxy says 'You don't have permission
to access this folder'");
            }
        }
    }
}
```

Step5: Client code الخطوة ٥: كود العميل

```
using System;
namespace ProxyDesignPattern
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Client passing employee with Role Developer to
folderproxy");
            Employee emp1 = new Employee("Anurag", "Anurag123", "Developer");
            SharedFolderProxy folderProxy1 = new SharedFolderProxy(emp1);
            folderProxy1.PerformRWOperations();

            Console.WriteLine();

            Console.WriteLine("Client passing employee with Role Manager to
folderproxy");
            Employee emp2 = new Employee("Pranaya", "Pranaya123", "Manager");
            SharedFolderProxy folderProxy2 = new SharedFolderProxy(emp2);
            folderProxy2.PerformRWOperations();

            Console.Read();
        }
    }
}
```

Key Points of Proxy vs. Facade

- The purpose of the Proxy is to add behavior while The purpose of the Facade is to simplify, which may actually involve removing behavior.
- Proxy object represents a singly object while Facade object represents a subsystem of object.

النقاط الرئيسية للوكيل مقابل الواجهة

- الغرض من البروكسي هو إضافة سلوك أثناء تشغيل ملف
الغرض من الواجهة هو التبسيط ، وهو ما قد يحدث بالفعل
تنطوي على إزالة السلوك.
- يمثل الكائن الوكيل كائنًا منفردًا أثناء الواجهة يمثل الكائن نظامًا فرعيًا للكائن .