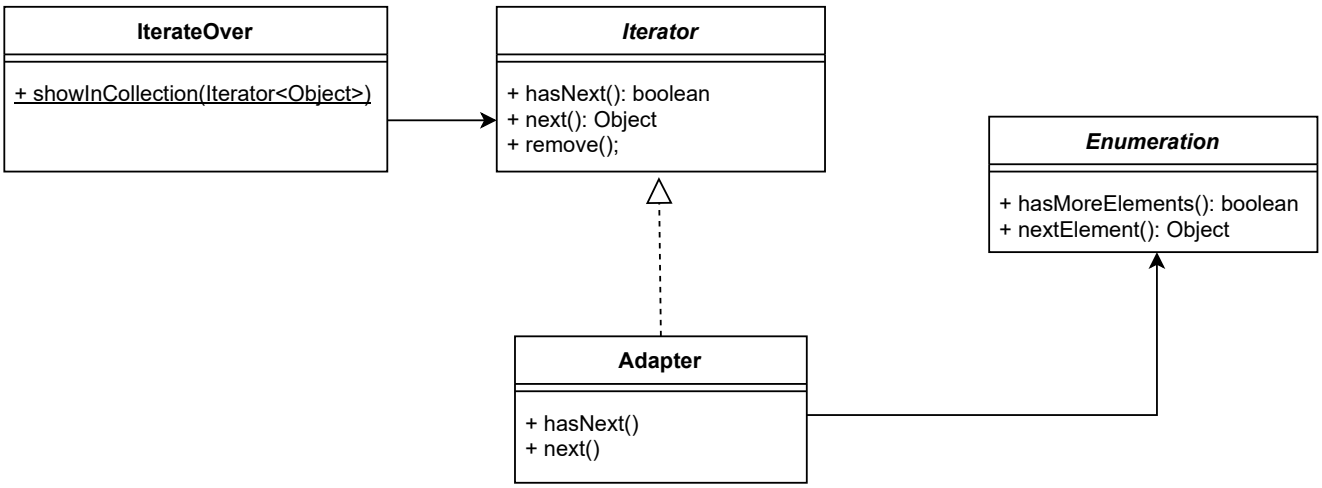
TP4 - Adaptateur / Procuration

#DesignPattern

Adaptateur

Un adaptateur permet de rendre compatible une interface sur laquelle on ne possède pas la main avec un objet existant. Dans l'exemple ci-dessous, nous souhaitons adapter la classe Iterator (java.utils) sur la classe Enumeration avec de rendre les deux intercompatibles.




```
// Notre Adapter va aller Override les méthodes de notre Iterator pour
les rendre
// compatibles avec l'Enumeration.

import java.util.Enumeration;
import java.util.Iterator;

public class Adapter implements Iterator {
    Enumeration<Object> myEnumeration;

    public Adapter(Enumeration<Object> enumeration) {
     this.myEnumeration = enumeration;
    }

    @Override
    public boolean hasNext() {
        return (myEnumeration.hasMoreElements());
    }
```

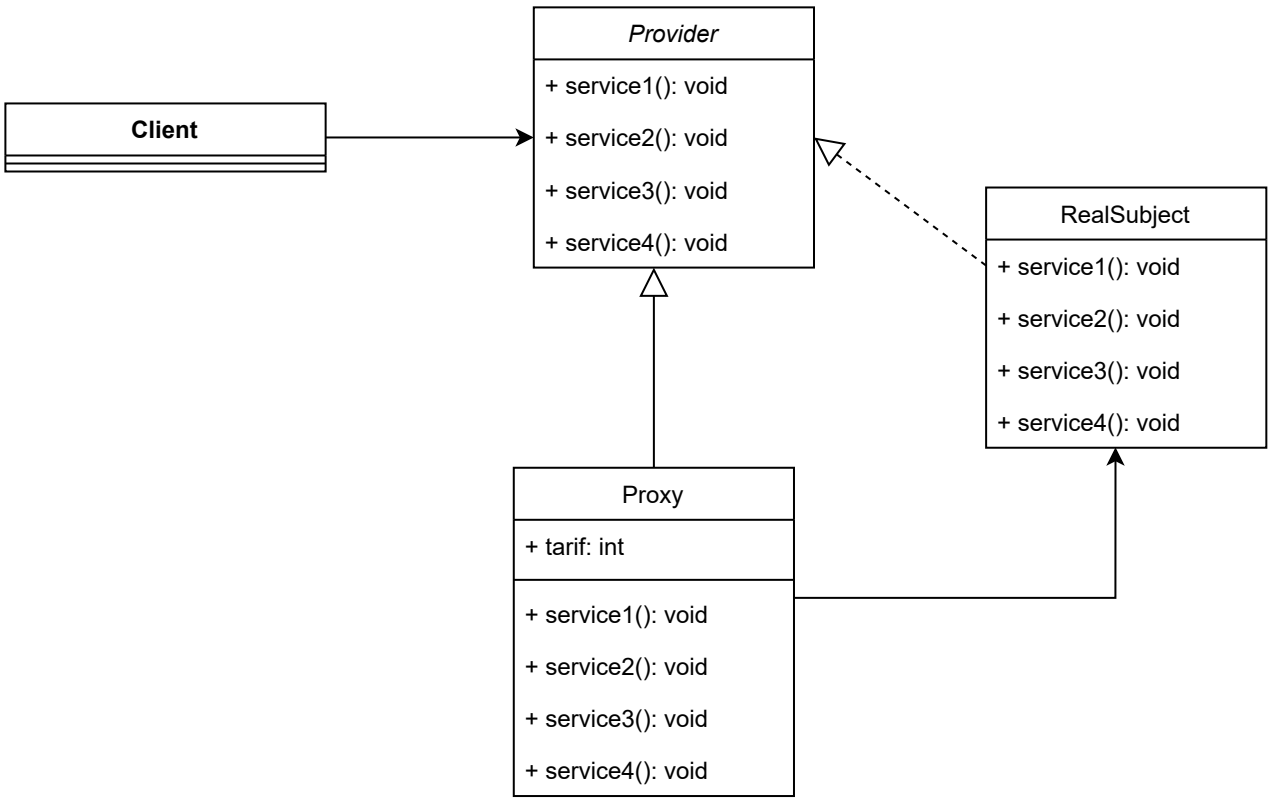
```
    @Override
    public Object next() {
        return(myEnumeration.nextElement());
    }

    @Override
    public void remove() {
    }
}
/*****************************************************************************


// nous souhaitons afficher tous les éléments d'une collection,
qu'importe si cette
// collection est une Enumerator ou une List

import java.util.ArrayList;

import java.util.Iterator;
import java.util.List;

public class IterateOver {
    public static void showInCollection(Iterator<Object> iteratorz) {
        while (iteratorz.hasNext()) {
            Object tmp = iteratorz.next();
            if (tmp instanceof String) {
                System.out.println(tmp);
            }
        }
    }
}

/*****************************************************************************


import java.util.ArrayList;
import java.util.List;
import java.util.Vector;
public class Client {
    public static void main(String[] args) {
        List<Object> myList = new ArrayList<Object>();
```

```
        myList.add(new String("Sheeesh"));
        myList.add(new Integer(42));
        myList.add(new String("Je suis trop fort au jeu"));
        myList.add(new Double(3.141592654));
        IterateOver.showInCollection(myList.iterator());
    // les deux auront le même comportement grâce à notre adapteur
        Vector<Object> myOtherList = new Vector<Object>();
        myOtherList.add(new String("Sheeesh"));
        myOtherList.add(new Integer(42));
        myOtherList.add(new String("Je suis trop fort au jeu"));
        myOtherList.add(new Double(3.141592654));
        IterateOver.showInCollection(new
Adapter(myOtherList.elements()));
    }
}
```

- Iterator définit l'interface spécifique à l'application pour un ensemble de services que le client utilise.
- Client collabore avec les objets conforme à l'interface Iterator en utilisant les services hasNext() et next().
- Enumeration possède les services recherchés mais avec la mauvaise interface.
- Adapter adapte l'interface d'Enumeration à l'interface souhaitée défini par Interator et est responsable des fonctionnalités que la classe adaptée ne fournit pas.

Procuration

La procuration (Proxy) est un subtitut à un autre objet afin d'en contrôler les accès (les opérations qui lui sont appliquées). On doit par exemple contrôler l'accès a un objet si l'on veut modifier son initialisation ou sa création avant son utilisation effective.



On souhaite implémenter un système de services qui utilisent un provider et selon le type de service, le prix est différent. Pour cela, nous allons utiliser un proxy pour faire les requêtes sur les services, ce dernier va intercepter les requêtes, calculer le cout de cette dernière et l'effectuer.

```
// Inerface du Provider, celui qui présente les services au client :
public interface Provider {
    void service1();
    void service2();

}

/*****************************************************************************



// Notre proxy qui compte le prix total.
public class ProxyCount implements Provider {

    private int couts;
    private Provider myServices;

    public int getCouts() {
        return couts;
    }
```

```
    public ProxyCount(Provider providerOfServices) {
        this.myServices = providerOfServices;
        this.couts = 0;
    }

    @Override
    public void service1() {
        this.myServices.service1();
        this.couts += 10;
    }

    @Override
    public void service2() {
        this.myServices.service2();
        this.couts += 15;
    }

}


/*************************************************************************


// création des services.

public class Service implements Provider{

    @Override
    public void service1() {
        System.out.println("Je suis le service 1");
        // do something
    }

    @Override
    public void service2() {
        System.out.println("Je suis le service 2");
        // do something
    }

}
/*************************************************************************
```

```
/**********************************************************************

// Notre client qui va instancier la classe Service qui utilise
provider et va demander
// au proxy de faire les requêtes sur le service
public class Client {

    public static void main(String[] args) {
        Provider testProvider = new Service();
        ProxyCount compteur = new ProxyCount(testProvider);
        compteur.service1();
        compteur.service2();
        System.out.println(compteur.getCouts());
    }

}
```

- Proxy à la même interface que l'objet à contrôler et gère une référence au Service, ce qui lui permet d'y accéder, d'en contrôler les accès et d'agir à sa place.
- Provider définit une interface commune pour Service et Proxy de sorte qu'un Proxy être utilisé partout ou un Service est attendu.
- Service définit l'objet réel que le proxy représente.