Iterator Design Pattern opdracht

Informatielinks

- https://en.wikipedia.org/wiki/Iterator_pattern
- https://sourcemaking.com/design_patterns/iterator
- https://dzone.com/articles/design-patterns-iterator
- https://www.youtube.com/watch?v=VKIzUuMdmag

Case

Elke hogeschool houdt een lijst bij van studenten die bij hen zijn ingeschreven. Zo houden Hogeschool Rotterdam, Inholland en NCOI dergelijke lijsten bij. Van elke student wordt de voornaam, achternaam en woonplaats bijgehouden. Elke hogeschool houdt de studenten op zijn eigen manier bij, de één gebruikt een array, de ander een standaard ingebouwde list (van Java of C#), en weer een ander gebruikt een zelf gemaakte list.

Op een dag besluit het Ministerie van Onderwijs dat ze inzage willen in de lijsten van studenten i.v.m. studiefinanciering. Voor deze inzage wilt het Ministerie één voor één door de lijsten van studenten heen kunnen gaan d.m.v. een loop, oftewel door de lijst heen kunnen itereren.

De programmeurs van het Ministerie irriteren zich aan het feit dat elke school zijn eigen type list teruggeeft aan het Ministerie:

```
class NCOI {
      public StudentNode studs()

      //...
}

class InHolland {
    public Student[] geefStudentenTerug()

      //...
}


class HogeschoolRotterdam {
    public List<Student> getSt () {

      //...
}
```

Het ministerie vraagt daarom aan elke school om één gemeenschappelijke return type te gebruiken i.p.v. StudentNode, Studenten[] en List<Student>, en om één gemeenschappelijke naam te gebruiken: getStudenten() i.p.v. namen als geefStudentenTerug(), getSt() en studs().

De scholen gaan hier echter niet mee akkoord. De returntypes en methods die studenten teruggeven worden namelijk ook door andere code die ze beheren aangesproken. Het veranderen van de returntypes en methode namen zou dan volgens de scholen betekenen dat ze de code van allerlei applicaties zouden moeten wijzigen.

Jij als Informatica student ziet echter een oplossing: de Iterator design pattern.

Gegeven code van de scholen en het ministerie

```
public class Student {
    public String voornaam;
    public String achternaam;
    public String plaats;

    public Student(String voornaam, String achternaam, String plaats) {
        this.voornaam = voornaam;
        this.achternaam = achternaam;
        this.plaats = plaats;
    }
}

public class HogeschoolRotterdam {
    private List<Student> studenten;

    public HogeschoolRotterdam() {
        // TODO Maak de studenten list, en voeg enkele studenten toe
    }

    public List<Student> getStudenten() {
        return null;
    }
}

public class InHolland {
    private Student[] studenten;

    public InHolland() {
       // TODO Maak de studenten array, en voeg enkele studenten toe
    }

    public Student[] geefStudentenTerug() {
        return null;
    }
}


public class NCOI {
    private StudentNode studentNode;

    public NCOI() {
        this.studentNode = new StudentNode(
                new Student("Jan", "Klaasen", "Rdam"),
                new StudentNode(
                        new Student("Peter", "Dirks", "Utrecht"),
                        new StudentNode(
                                new Student("Erik", "Water", "Leiden"),
                                new Empty()
                        )
                )

        );
    }

    public StudentNode studs() {
        return studentNode;
    }
}
```

```
public abstract class AbstractStudentNode {
    public boolean isEmpty;
    public Student student;
}

public class StudentNode extends AbstractStudentNode {
    public AbstractStudentNode tail;

    public StudentNode(Student student, AbstractStudentNode tail) {
        this.student = student;
        this.tail = tail;
        isEmpty = false;
    }
}

public class Empty extends AbstractStudentNode {
    public Empty() {
        isEmpty = true;
    }
}



public class Ministerie {
    InHolland inHolland;
    HogeschoolRotterdam hr;
    NCOI ncoi;

    public Ministerie(InHolland inHolland, HogeschoolRotterdam hr, NCOI
ncoi) {
        this.inHolland = inHolland;
        this.hr = hr;
        this.ncoi = ncoi;
    }

    public void printStudenten() {
        Student[] inhollandStudenten = inHolland.geefStudentenTerug();
        for (int i = 0; i < inhollandStudenten.length; i++) {
            Student student = inhollandStudenten[i];
            System.out.println(student.voornaam);
        }

        List<Student> hrStudenten = hr.getStudenten();
        for (int i = 0; i < hrStudenten.size(); i++) {
            Student student = hrStudenten.get(i);
            System.out.println(student.voornaam);
        }

        AbstractStudentNode ncoiStudentNode = ncoi.studs();
        while (!ncoiStudentNode.isEmpty) {
            Student student = ncoiStudentNode.student;
            System.out.println(student.voornaam);

            ncoiStudentNode = ncoiStudentNode.tail;
        }
    }
}

public class App {
    public static void main(String[] args) {
        InHolland inHolland = new InHolland();
        HogeschoolRotterdam hr = new HogeschoolRotterdam();
        NCOI ncoi = new NCOI();
```

```
        Ministerie ministerie = new Ministerie(inHolland, hr, ncoi);
        ministerie.printStudenten();
    }
}
```

Het probleem

De probleemcode waar de ministerie programmeurs zich aan ergeren is de printStudenten() methode. De methode werkt kwa functioneren prima, maar het moeten schrijven van aparte loops en code per school, is een vorm van duplicated code waarvoor een snelle oplossing niet direct voor de hand ligt. Het ministerie verwacht dat er honderden scholen toegevoegd gaan worden, en de programmeurs hebben geen zin om daardoor honderden aparte loops te moeten schrijven voor een op zich zelf simpel probleem: iteratie van studenten.

Het hoofdprobleem wordt veroorzaakt doordat elke school bloot legt hoe deze zijn studenten opslaat: een array, een list, een AbstractStudentNode. De code die daar gebruik van maakt (zoals printStudenten()), krijgen daardoor te concrete informatie te weten over hoe studenten opgeslagen zijn, wat zorgt een onnodig hoge coupling. Met als gevolg dat indien een school de opslagwijze van studente wilt veranderen (bijv. van een array naar een set), iedereen die daar gebruik van maakt, ook hun code moet wijzigen. (Wat onnodig veel onderhoudswerk is.)

De iterator design pattern biedt een oplossing voor dit probleem. En maakt het vervolgens tevens gemakkelijker om een algemeen interface te maken waarmee het ministerie elke school op dezelfde wijze kan benaderen. Al hoewel er een aantal classen bij zullen komen, zal de iterator design pattern de code netter maken, coupling verlagen. Zo zal er in de eindoplossing slechts één loop nodig zijn, zelfs indien er honderden scholen worden toegevoegd.

Iterator design pattern

Een aggregate is een object dat uit meerdere elementen bestaat, bijvoorbeeld de scholen, welke uit studenten bestaat.

Doel/definitie: de iterator design pattern biedt een manier om elementen van een aggregate object te benaderen zonder de onderliggende representatie (de wijze van opslag) bloot te leggen.

In de onderstaande stappen ga je de iterator design pattern toepassen op de code.

(Al hoewel deze opdracht wat achtergrond informatie bevat, is het wel de bedoeling dat de eerder vermelde links worden geraadpleegd voor informatie over het patroon.)

Stap 1 – Maak een Iterator interface

Maak de Iterator interface van de Interator design pattern, met daarin de methodes:

- boolean hasNext() // geeft true terug indien er een volgend element is
- Object next() // haalt het volgende element op. Of, indien je liever nog even met concrete studenten werkt: Student next().

HasNext() kijkt alleen of er een volgend element is, maar de iterator zelf blijft op dezelfde positie/element staan. Next() laat de iterator wel naar de volgende positie/element gaan. (Indien je next() voor de eerste keer aanroept, geeft deze het eerste element terug.)

(Eventueel kan je direct generics gebruiken i.p.v. het Object type.)

Je krijgt dan clientcode als:

```
while (collectie.hasNext()) {
      print(collectie.next());
}
```

Opdracht 2 – Maak voor elke collectie een concrete iterator

Er zijn verschillende collecties in de gegeven code: array, list en een node structuur. Maak voor elke collectie een concrete iterator implementatie.

En laat de code in printStudenten() gebruik maken van deze iterators. (De iteratie code gaat dan al wat meer op elkaar lijken.)

Stap 3 – Maak scholen abstracter

Voor het ministerie doet het er niet toe of het gaat om InHolland of HR, het ministerie wilt maar één ding: itereren over de studenten van een school. Om alle scholen op dezelfde wijze (met dezelfde code) te kunnen benaderen, moet er een type komen waar elke school van af stamt. We gebruiken hiervoor de interface School (eventueel kan je het ook “Iterable” noemen), met daarin een methode:

```
Iterator iterator();
```

Maak dit interface, en laat elke school deze implementeren.

Het doel van de iterator() methode is om een iterator te maken. Het ministerie kan deze iterator gebruiken om over studenten te iteraten. Dus kwa gebruik gaat het als volgt: de Ministerie roept van een school de iterator() method, welke ter plekke een iterator maakt en teruggeeft, waarmee het Ministerie vervolgens over alle studenten kan iteraten van die school.

Een iterator kan maar één keer gebruikt worden na elke iterator() aanroep. Oftewel één iterator object per iteratie, want je kan een iterator niet resetten naar de begintoestand.

Stap 4 – gebruik de iterator() method

Wijzig printStudenten() zodat deze de iterator () method van elke school gebruikt. Deze geeft dus een iterator terug welke vervolgens gebruikt kan worden om door de studenten te iteraten.

Stap 5 – reduceer de code tot één loop

De code kan nu gereduceerd worden tot één loop. Dit kan je bereiken door de loop (die gebruik maakt van de iterator) te plaatsen in een aparte methode waarbij je een school kan meegeven als argument/parameter om over die school te iteraten.

```
void print(School school);
```

Stap 6 – Generics voor iterator hergebruik

Al hoewel de iterator design pattern nu in principe al geïmplementeerd is, kan de code verder worden verbeterd door generics te gebruiken. Hierdoor kan je generieke iterators maken die je niet alleen voor studenten kan gebruiken maar ook voor andere type elementen. Dit kan in principe ook door Object te gebruiken i.p.v. Student:

```
interface Iterator {
      boolean hasNext();
```

```
      Object next();
}
```
i.p.v.

```
interface Iterator {
      boolean hasNext();
      Student next();
}
```
Maar met generics bereik je type safety, omdat je geen expliciete casts hoeft te gebruiken om van een object naar een student te gaan.

```
interface Iterator<T> {
      boolean hasNext();
      T next();
}
```

Opmerkingen

Iterators in de Java en C# standard library

Iterators worden dusdanig vaak gebruikt dat talen als Java en C# deze ingebouwd hebben:

https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html

https://msdn.microsoft.com/en-us/library/78dfe2yb(v=vs.110).aspx

De precieze namen kunnen echter per taal verschillen.

De for-loop in Java en de foreach-loop in C# maken ook gebruik van iterators. Indien je de officiele iterator van Java of C# gebruikt, kan je de loops als volgt schrijven:

```
// Java
for (Student s : hogeschoolRotterdam) {}

// C#
foreach (Student s in hogeschoolRotterdam) {}
```

Voltijd slides

In de voltijd slides wordt een verdere verbetering van iterators gebruikt, al hoewel dit opzich geen onderdeel is van de iterator design pattern. I.p.v.

```
interface Iterator<T> {
      boolean hasNext();
      T next();
}
```
wordt er gebruikt:

```
interface Iterator<T> {
      IOption<T> getNext();
}
```
Een voordeel is dat het gebruik van IOption oplossingen zorgt voor het minder snel voorkomen van nullPointerExceptions, en dat er maar één methode nodig is. Een nadeel zou kunnen zijn dat IOption niet in elke taal standaard een IOption class/interface bevat.

Conclusie

Het doel van de Iterator design pattern is: het aanbieden van een manier om elementen van een aggregate object te benaderen zonder de onderliggende representatie (de wijze van opslag) bloot te leggen.

Het enige dat de scholen (aggregates) nu bootleggen is de School en Iterator interface. De School interface (ook wel Iterable interface genoemd), maakt een iterator, en de iterator wordt gebruikt om over studenten te iteraten met slechts alleen maar abstracte methodes als next() en hasNext(). Het ministerie kan niet meer zien of het gaat om een array, list, of set, waardoor de school gemakkelijk zijn interne onderliggende opslagrepresentatie kan wijzigen, zonder dat het ministerie redenen heeft om dan te gaan klagen.

Een ander effect is dat responsibility/verantwoordelijkheid van iteratie wordt verplaatst van de aggregate naar een iterator, zodat de aggregate/school zich meer kan focussen op het beheren van studenten i.p.v. ook het itereren van studenten. (Single Responsibility, de S van het SOLID principe. Wat zorgt voor een hogere cohesie.)