Het Iterator Pattern

Objectgeoriënteerde analyse en ontwerp

Iteratie
De context

- Je hebt een **verzameling** dingen (synoniemen: *collectie*, *aggregaat*)
- Je wil iets doen met elk element van de aggregatie, een na een.
- Een **iterator** is een object dat de verantwoordelijkheid heeft om de verzameling element per element te overlopen. Omdat de iterator er is, moet je niet meer weten hoe de verzameling concreet geïmplementeerd is.

Welke garanties geeft een iterator?

- Elk element komt minstens 1 keer aan bod

  JA, ALTIJD

- Elk element komt juist 1 keer aan bod

  ALTIJD

- De elementen komen in een vaste volgorde aan bod

  NIET NOODZAKELIJK

Welke garanties geeft een iterator? (2)

- Je mag elementen toevoegen of verwijderen tijdens de iteratie **(fail safe)**

  SOMS

- Je mag elementen veranderen tijdens de iteratie

  SOMS

- Als je toch elementen toevoegt of verwijdert tijdens de iteratie, dan krijg je een error **(fail fast)**

  NIET NOODZAKELIJK

- Meer dan 1 iterator kan tegelijk dezelfde collectie overlopen (thread safety)

  ja, meestal (maar let op als je iets wijzigt…)

Waarom een Iterator Pattern

Om de Client, die de collectie overloopt, **los** te **koppelen** van de concrete implementatie van de collectie.

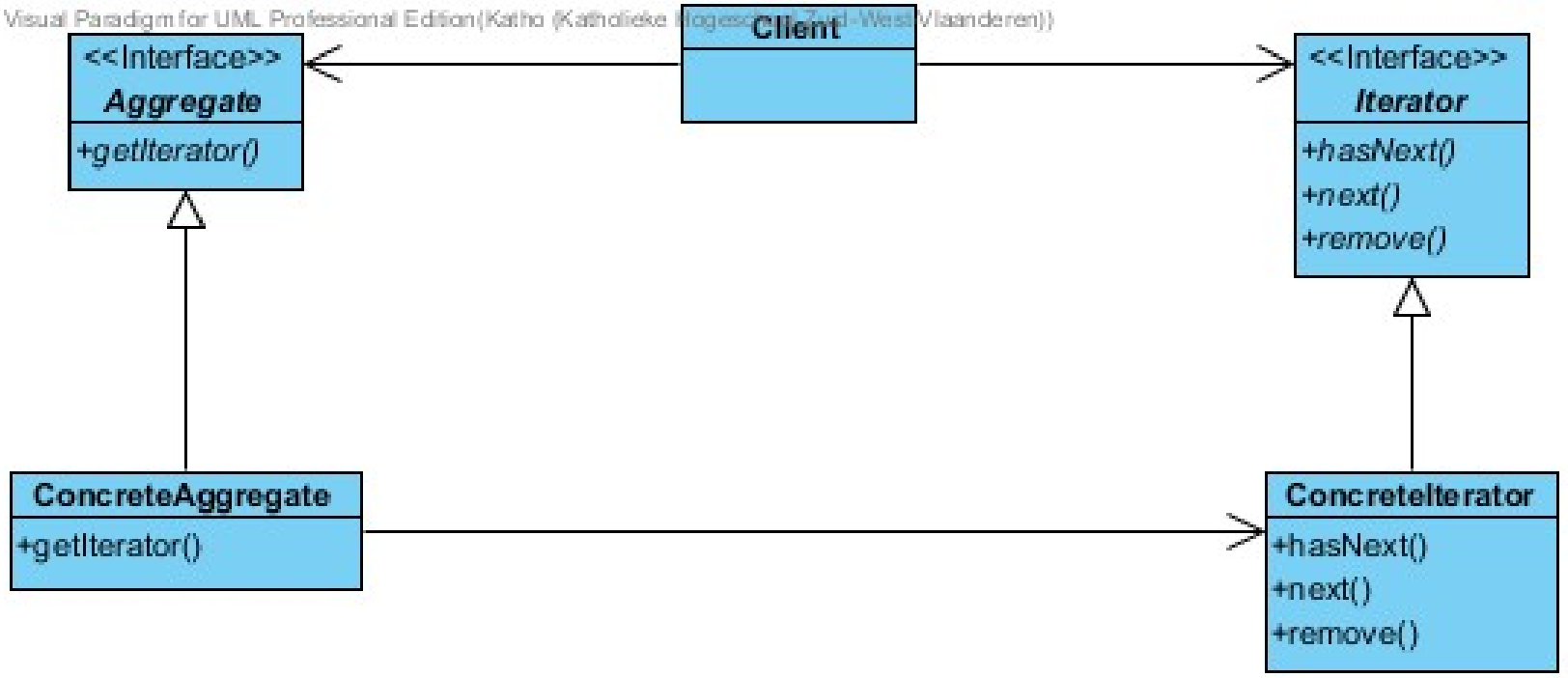
(Een Client moet niet weten hoe de collectie concreet in elkaar zit.)

Programmeer naar een interface, niet naar een implementie

Wat moet een iterator kunnen?

- **toegang** geven tot de elementen van het aggregaat (*element access*)
- een **startpunt** voor een iteratie voorzien
- naar het **volgende** element gaan (*element traversal*)
- aangeven wanneer de iteratie ten **einde** is

Structuur van het pattern



katholieke hogeschool
associatie KU Leuven
vives

Iterator scheiden van aggregaat?

Waarom worden iterator en aggregaat gescheiden?

Single responsability principle

“Een klasse heeft 1 welomschreven verantwoordelijkheid.”

- Een verzameling bijhouden is 1 verantwoordelijkheid
- Itereren over die verzameling is een andere verantwoordelijkheid

Je kunt hierover discussiëren…

Iterator in java

In JDK: **Iterator<E>** interface

hasNext()	Is er nog een volgend element?
next()	Geef het volgende element
remove()	Verwijder het element dat we kregen van de meest recente next().

Ondersteund door zowat alle collectieklassen uit de JDK.

(HashTable: op zich niet, maar Keys en Values wel.)

Iterator in java

Java collecties ondersteunen allemaal het iterator pattern.

Wat als je een eigen collectie itereerbaar wil maken?

Iterable<T> interface

Iterable<T> heeft 1 methode:

```
// retourneert een iterator voor de collectie
Iterator<T> iterator();
```

Iterator in .NET

- interface **IEnumerable, IEnumerable<T>**:

  GetEnumerator(): retourneert een IEnumerator, waarmee je over het aggregaat kunt itereren.

- interface **IEnumerator, IEnumerator<T>**:

  Current(): retourneert het huidige element (ongeldig als je niet eerst minstens 1 keer MoveNext aangeroepen hebt)

  MoveNext(): ga naar het volgende element. (Returnwaarde is *false* als er geen volgende element is, anders *true*.)

  Reset(): ga terug naar de beginsituatie (Current is ongeldig; eerst MoveNext() nodig)

Itereren korte stijl

Programmeertalen hebben meestal een constructie die het gebruik van een iterator gemakkelijker maken.

Java: for / in – loops (sinds Java 5)

```
for (Object obj : Collection) {
    // doe iets met element
}
```

.NET: foreach – loop

```
foreach (Object o in Collection) {
    // doe iets met element
}
```

Externe vs interne iterators

- Wat we nu gezien hebben zijn **externe iterators**

  -> De client is verantwoordelijk voor het opschuiven (moet next() aanroepen).

- Een andere stijl zijn **interne iterators**

  -> De iterator zorgt voor het opschuiven.
  De client moet aan de iterator zeggen wat hij bij elk element moet doen.

Interne iterator in java

Je kunt **streams** beschouwen als interne iteratie.

```
List<String> myList =
    Arrays.asList("a1","a2","b1","c2","c1");

myList
    .stream()
    .map(String::toUpperCase)
    .forEach(System.out::println);
```

Interne iterator in C#

Interne interatie kan met de extension method ForEach:

```
List<int> intlijst = new List<int>()
  { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

intlijst.ForEach(i => Console.WriteLine(i));
```

Generator
‘just in time’ collectie

.NET heeft een mechanisme om collecties te implementeren die hun elementen ‘just in time’ fabriceren.

Zo een collectie heet een generator.

Nodig:

- methode die **IEnumerable<T>** als returntype heeft
- in die methode een **lus** met daarin **yield return**

De lus wordt maar zoveel keer uitgevoerd als de gebruiker van de iteratie vraagt.

Generator voorbeeld

Machten van 2 genereren.

De methode MachtenVan2() kan een methode van een klasse zijn, of een statische methode.

```
public IEnumerable<double> MachtenVan2()
{
    double power = 1.0;

    while (true)
    {
        yield return power;
        power = power * 2;
    }
}
```

Generator voorbeeld

De eerste 50 machten van 2 opvragen.

De lus in MachtenVan2() wordt maar 50 keer uitgevoerd.

```
foreach (double macht in MachtenVan2().Take(50))
  Console.WriteLine(macht);
```

Generator in java

Een generator in java kan met streams, maar het is niet zo eenvoudig als in C#.