Design Patterns & Software Architecture

Iterator

dr. Joost Schalken-Pinkster

Windesheim University of Applied Science

The Netherlands

The contents of these course slides is (in great part) based on:
Chris Loftus, *Course on Design Patterns & Software Architecture for NEU*. Aberystwyth University, 2013.
Jeroen Weber & Christian Köppe, *Course on Patterns and Frameworks*. Hogeschool Utrecht, 2013.
Leo Pruijt, *Course on Software Architecture*. Hogeschool Utrecht, 2010-2013.

Session overview

- Iterator

Iterator design pattern

Iterator design pattern:
Let's start with a very simple example

Lets print all elements of an array or list…

Sequences it all seems so simple, no?

Going through an array with for (.. : ..) :

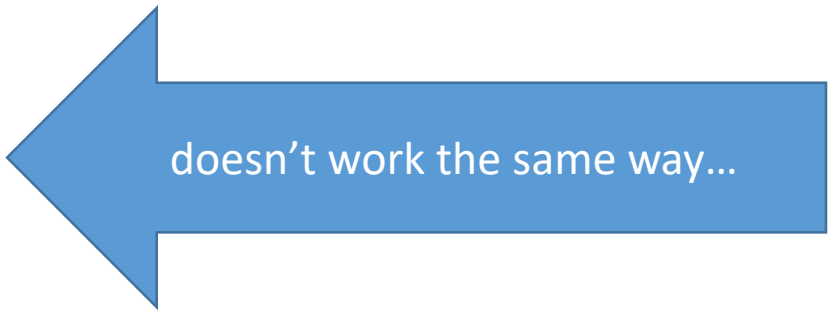
```
package simple_example;


public class ArrayIterator {
        public static void main(String[] args) {
                String[] aStringArray = {"a", "b", "c"};

                for(String s: aStringArray) {
                        System.out.println(s);
                }
        }
}
```

Going through array before for(.. : ..) : (Java < 1.5)

```
package simple_example;
public class ArrayIteratorNoIterator {
        public static void main(String[] args) {
                String[] aStringArray = {"a", "b", "c"};

                for(int i = 0; i < aStringArray.length; i++) {
                        System.out.println(aStringArray[i]);
                }
        }
}
```

Going through List before for(.. : ..) :

```
package simple_example;
import java.util.*;

public class ListIteratorNoIterator {
        public static void main(String[] args) {
                String[] aStringArray = {"a", "b", "c"};
                List<String> aStringList = Arrays.asList(aStringArray);

                for(int i = 0; i < aStringList.size(); i++) {
                        System.out.println(aStringList.get(i));
                }
        }
}
```

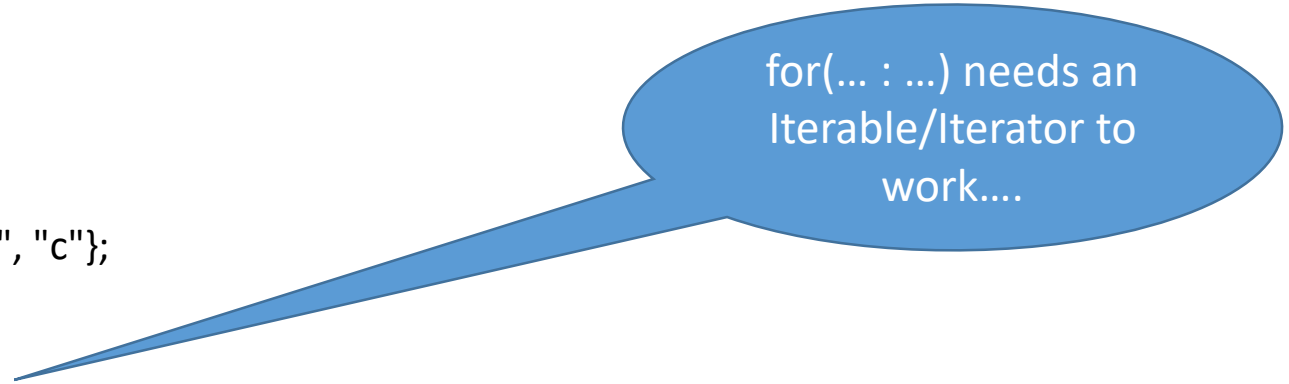


How can we make clients that are unaware of what we traverse?

- Iterator to the rescue
  - We need to “encapsulate what varies”
  - Get the two collection classes to provide an iterator object that has methods:
    - hasNext(): boolean
    - next(): Object
- These objects hide the way looping is done and the type of data-structure used

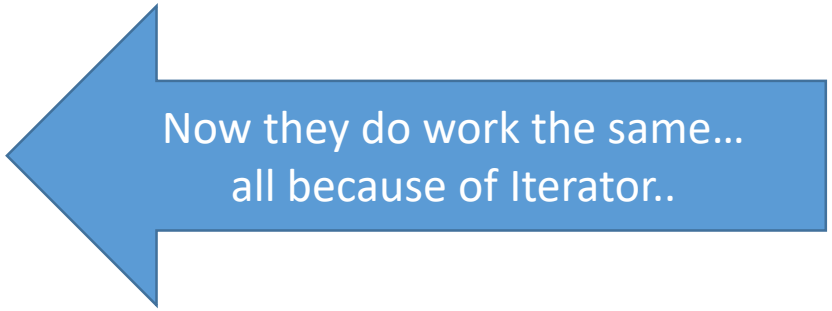
Going through an array with for (.. : ..) :

```
package simple_example;


public class ArrayIterator {
        public static void main(String[] args) {
                String[] aStringArray = {"a", "b", "c"};

                for(String s: aStringArray) {
                        System.out.println(s);
                }
        }
}
```



Iteration through an a list:

```
package simple_example;

import java.util.*;


public class ListIterator {
        public static void main(String[] args) {
                String[] aStringArray = {"a", "b", "c"};
                List<String> aStringList = Arrays.asList(aStringArray);

                for(String s: aStringList) {
                        System.out.println(s);
                }
        }
}
```



```
public class xxxxIterator implements Iterator<Aggregate> {
        Aggregate a = a;
        public GraphicalIterator(Aggregate a) {
                this.a = a;
        }
        @Override
        public boolean hasNext() {
                // ADD CODE HERE
        }
        @Override
        public Aggregate next() {
                // ADD CODE HERE
        }
        @Override
        public void remove() {
                throw new UnsupportedOperationException("Cannot remove from iterator");
        }
}
```

Let’s find a design pattern

Will now present, on the board,
and using Eclipse,
a solution that utilises
the Iterator design pattern…

Case: Zoo Requirements

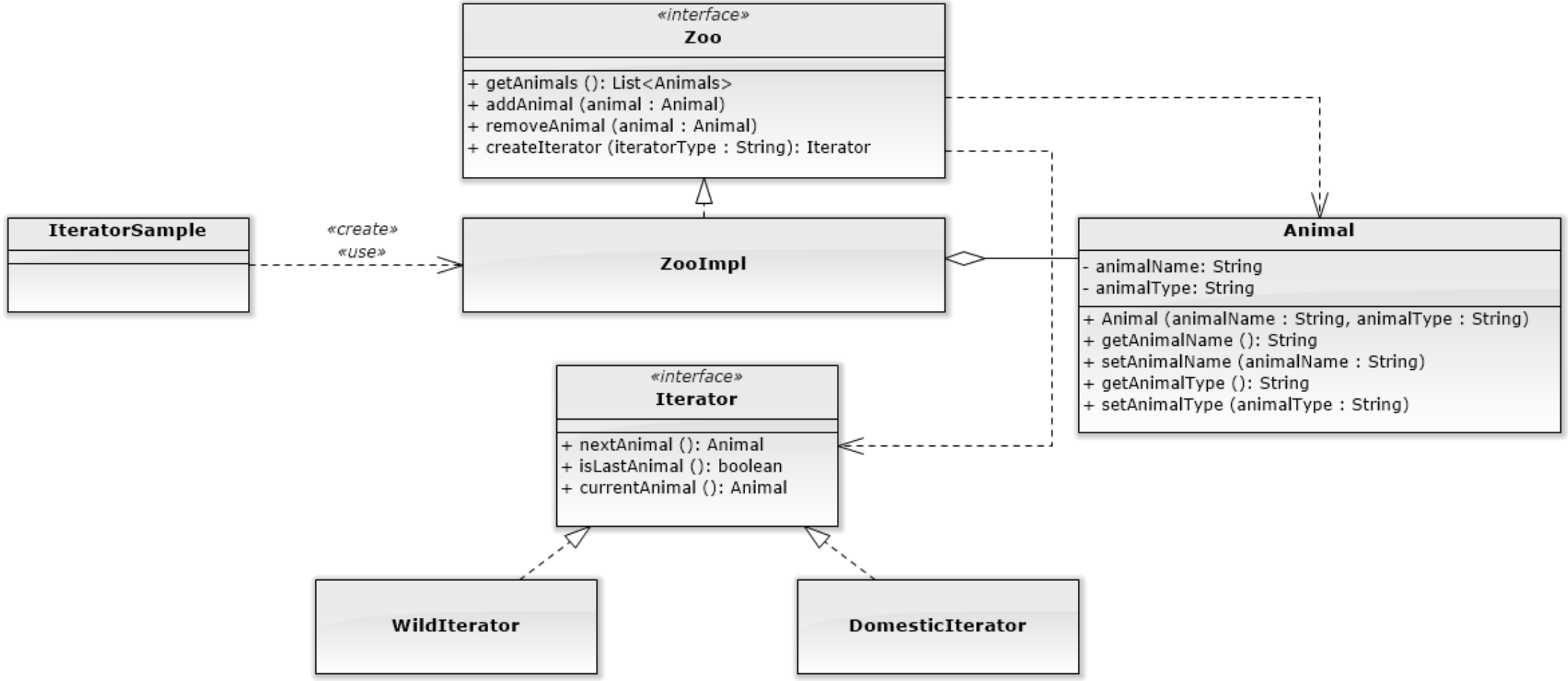
Lets create a zoo that contains some animals…

Lets implement behaviour to show all animals..

And implement behaviour to show all animals that are wild or domestic..

Source: Joe (2013), *Iterator Design Pattern*. JavaPapers, June 30th , 2013. http://javapapers.com/design-patterns/iterator-design-pattern/

Case: Zoo Design

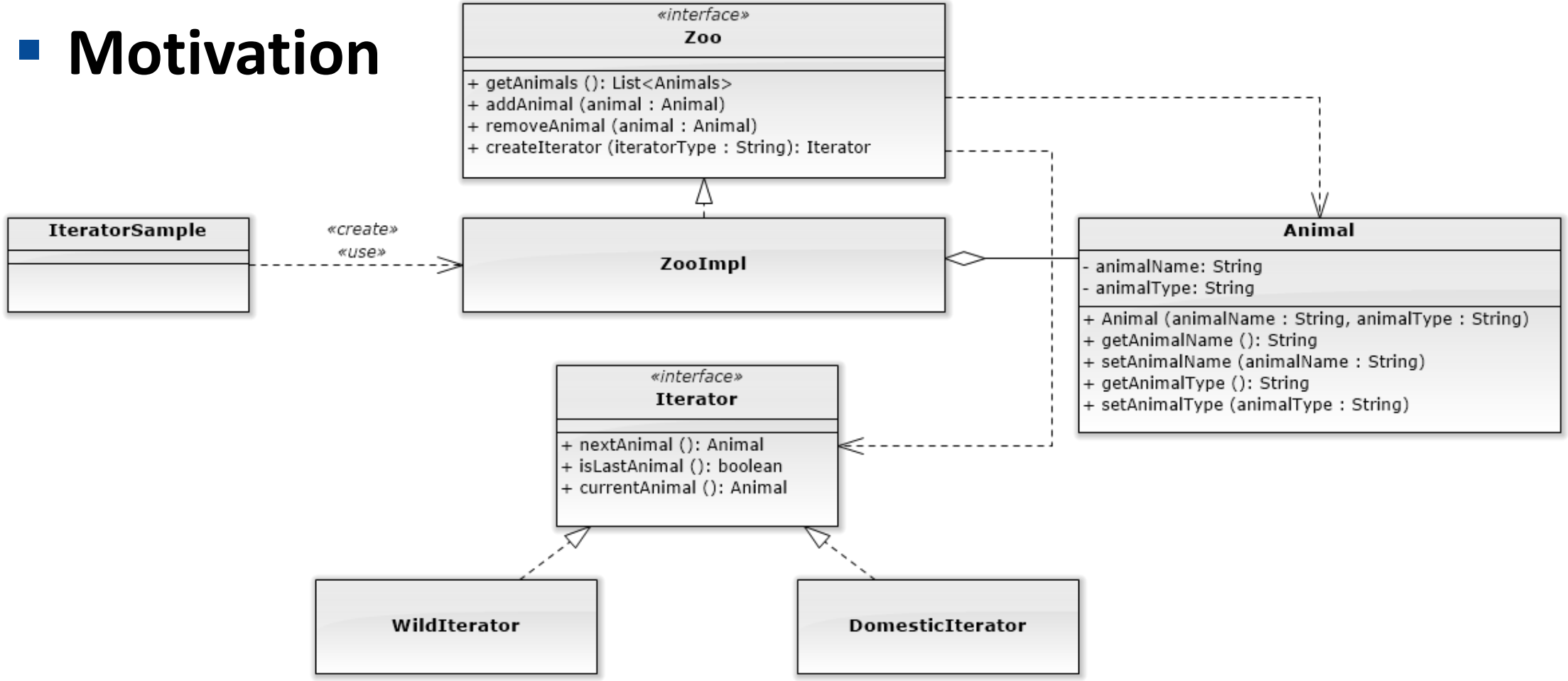


Source: Joe (2013), *Iterator Design Pattern*. JavaPapers, June 30th , 2013. http://javapapers.com/design-patterns/iterator-design-pattern/

Iterator pattern definition

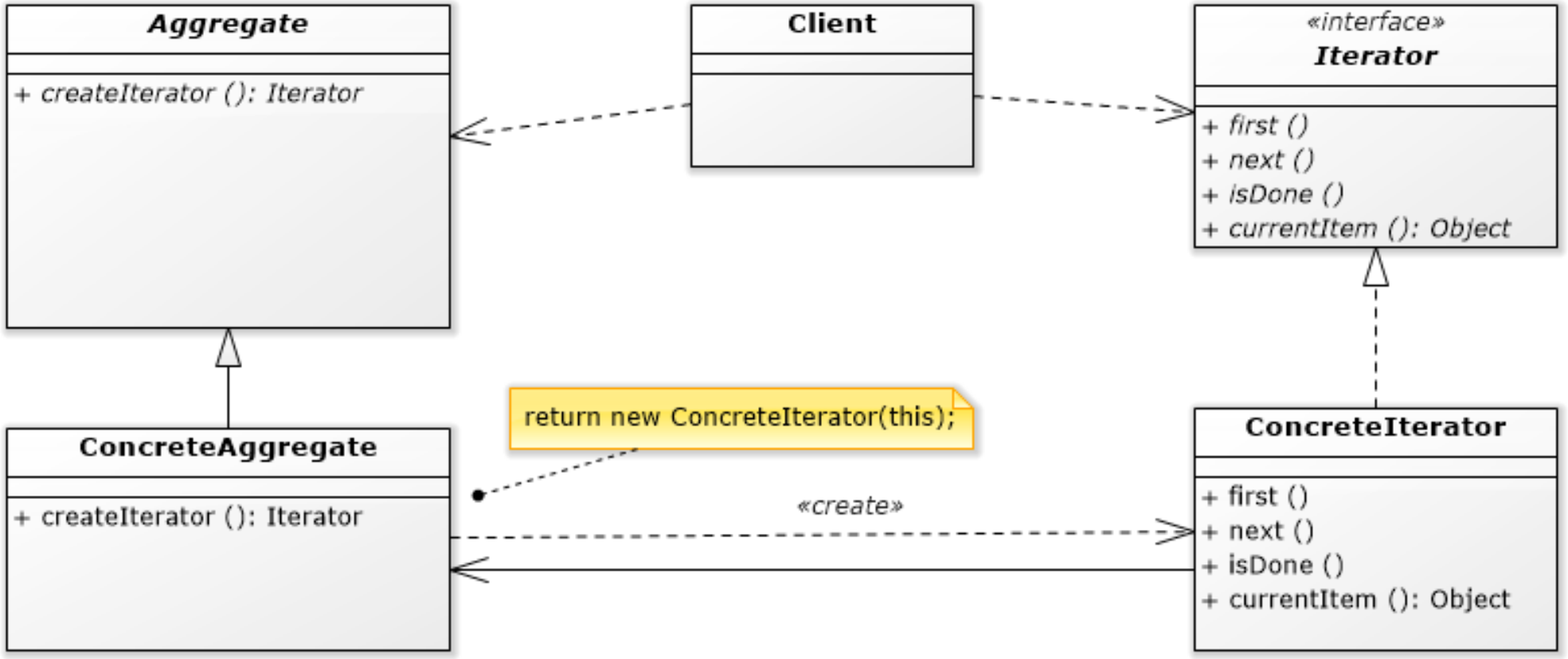
- **Intent**: Provide a way to access the elements of an aggregate objects sequentially without exposing its underlying representation.
- **Motivation**:
  - An aggregate object such as a list should give you a way to access its elements without exposing its internal structure.
  - Moreover, you might want to traverse the list in different ways, depending on what you want to accomplish.
  - The key idea in this pattern is to take the responsibility for access and traversal out of the list object and put it into an iterator object.

Motivation


«interface»
Zoo
+ getAnimals (): List<Animals>
+ addAnimal (animal : Animal)
+ removeAnimal (animal : Animal)
+ createIterator (iteratorType : String): Iterator
IteratorSample
«create»
«use»
ZooImpl
Animal
- animalName: String
- animalType: String
+ Animal (animalName : String, animalType : String)
+ getAnimalName (): String
+ setAnimalName (animalName : String)
+ getAnimalType (): String
+ setAnimalType (animalType : String)
«interface»
Iterator
+ nextAnimal (): Animal
+ isLastAnimal (): boolean
+ currentAnimal (): Animal
WildIterator
DomesticIterator

- **Applicability**: Use Iterator when:
  - to access an aggregate object's contents without exposing its internal representation.
  - to support multiple traversals of aggregate objects.
  - to provide a uniform interface for traversing different aggregate structures (that is, to support polymorphic iteration).

- **Structure:**


Aggregate
+ createIterator (): Iterator
Client
«interface»
Iterator
+ first ()
+ next ()
+ isDone ()
+ currentItem (): Object
return new ConcreteIterator(this);
ConcreteAggregate
+ createIterator (): Iterator
«create»
ConcreteIterator
+ first ()
+ next ()
+ isDone ()
+ currentItem (): Object

- **Participants**:
  - Iterator
    - defines an interface for accessing and traversing elements.
  - ConcreteIterator
    - implements the Iterator interface.
    - keeps track of the current position in the traversal of the aggregate.
  - Aggregate
    - defines an interface for creating an Iterator object.
  - ConcreteAggregate
    - implements the Iterator creation interface to return an instance of the proper ConcreteIterator.

- **Consequences**: The iterator pattern:
  - It supports variations in the traversal of an aggregate
  - Iterators simplify the Aggregate interface.
  - More than one traversal can be pending on an aggregate.

- **Implementation**:
  - Who controls the iteration
    - Internal vs external
  - Who defines the traversal algorithm
  - How robust is the iterator
  - Additional Iterator operations
  - Using polymorphic iterators
    - issue for C++ and other non-virtual languages
  - Iterators may have privileged access
  - Iterators for composites
  - Null iterators

External vs Internal Iterators

- External iterator: iteration is under the control of the client class: e.g. next() or just a loop…
- Internal iterator: iteration is under the control of the aggregate class: e.g. a method of Zoo…

Polymorphic Iterator

- If an iterator has common contract across multiple collections, the client code which interacts with the iterator need not be changed for different collections.
- This is called polymorphic iterator.

- Polymorphic iterator are sometimes more expensive to call.

Robust Iterator

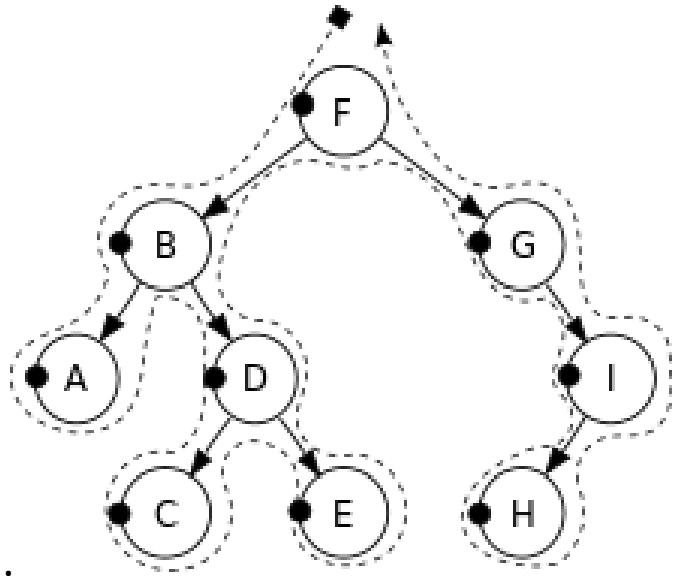
What would happen if the contents of the collection is modified like a new element is inserted or an existing element is deleted?

- It will behave inconsistently resulting in erroneous output.
- A simplistic iterator will create a copy of the collection when it is instantiated and use that for traversal. So that during the iteration process, even if the collection is modified nothing would happen.
- A robust iterator is the one that works intact even if the associated collection is modified.

Tree traversion

Tree-like aggregated structures (like Composites) can be iterated to/traversed through in three ways:

- Pre-order (i.e. F, B, A, D, C, E, G, I, H)
  1. Display the data part of root element (or current element)
  2. Traverse the left subtree by recursively calling the pre-order function.
  3. Traverse the right subtree by recursively calling the pre-order function.
- In-order (i.e. A, B, C, D, E, F, G, H, I)
  1. Traverse the left subtree by recursively calling the in-order function.
  2. Display the data part of root element (or current element).
  3. Traverse the right subtree by recursively calling the in-order function.
- Post-order (i.e. A, C, E, D, B, H, I, G, F)
  1. Traverse the left subtree by recursively calling the post-order function.
  2. Traverse the right subtree by recursively calling the post-order function.
  3. Display the data part of root element (or current element).



Wikipedia (2014), *Tree traversal*. Wikipedia Foundation, December 15thm 2014. http://en.wikipedia.org/wiki/Tree_traversal

Null Iterator

- Imagine we are traversing a tree data structure. During the tree traversal, when we ask for the nextElement, we will get a concrete iterator which will help us to traverse through the tree.
- If we ask for the nextElement in the leaf node, we will get a null iterator returned by the collection signifying the leaf node.
- This behavior will allow us to design the tree traversal pattern.

Iterator can do more than traversal

- Iterators are not limited to traversal alone. It is entirely left to the purpose and implementation.
- There can be functional logic involved in an iterator.
  - We can have a FilteringIterator, which can filter out certain required values and provide for traversal.
    - For example in a list containing wild and domestic animals, we can have two different iterators as WildAnimalIterator and DomesticAnimalIterator.

Design Principle: Single Responsability

A class should have only one reason to change…

E. Freeman, E. Freeman, B. Bates, K. Sierra, Head First Design Patterns. O'Reilly, 2004. p. 343

Single responsibility

Design principle: A class should have only one reason to change…

- The main responsibility of Zoo is to implement an internal collection of Animal and provide operations…
- Should not also be to support iteration
  - Change one thing and you might break the other…
  - Prevents multiple iterations…
  - Makes it difficult to switch implementations…

Reading

For this lesson please read:

- Chapter 9 (Well managed collections) of Head First Design Patterns.