CS 635 Advanced Object-Oriented Design & Programming
Fall Semester, 2022
Doc 5 Pattern Intro, Iterator Pattern
Sep 8, 2022

Pattern Beginnings

"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice"

"Each pattern is a three-part rule, which expresses a relation between a certain context, a problem, and a solution"

A Pattern Language, Christopher Alexander, 1977

A Place To Wait

The process of waiting has inherent conflicts in it.

Waiting for doctor, airplane etc. requires spending time hanging around doing nothing

Cannot enjoy the time since you do not know when you must leave

Classic "waiting room"
Dreary little room
People staring at each other
Reading a few old magazines
Offers no solution

Fundamental problem
How to spend time "wholeheartedly" and
Still be on hand when doctor, airplane etc arrive

Fuse the waiting with other activity that keeps them in earshot
- Playground beside Pediatrics Clinic
- Horseshoe pit next to terrace where people waited

Allow the person to become still meditative
- A window seat that looks down on a street
- A protected seat in a garden
- A dark place and a glass of beer
- A private seat by a fish tank

A Place To Wait

Therefore:

"In places where people end up waiting create a situation which makes the waiting positive. Fuse the waiting with some other activity - newspaper, coffee, pool tables, horseshoes; something which draws people in who are not simple waiting. And also the opposite: make a place which can draw a person waiting into a reverie; quiet; a positive silence"

Chicken And Egg

Problem

Two concepts are each a prerequisite of the other
To understand A one must understand B
To understand B one must understand A
A "chicken and egg" situation

Constraints and Forces

First explain A then B
Everyone would be confused by the end
Simplify each concept to the point of incorrectness to explain the other one
People don't like being lied to

Solution

Explain A & B correctly but superficially

Iterate your explanations with more detail in each iteration

Patterns for Classroom Education, Dana Anthony, pp. 391-406, Pattern Languages of Program Design 2, Addison We
1996

Design Principle 1

Program to an interface, not an implementation

Use abstract classes (and/or interfaces in Java) to define common interfaces for a set of classes

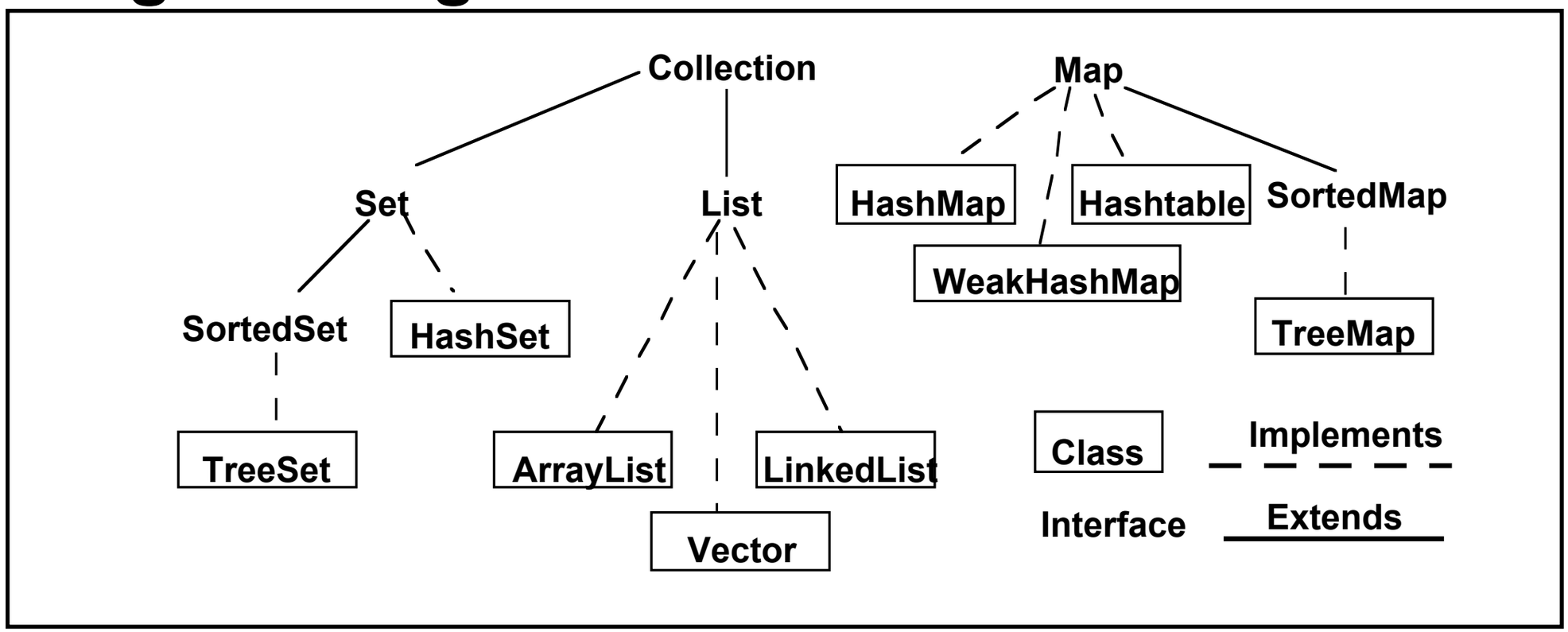
Declare variables to be instances of the abstract class not instances of particular classes

Benefits of programming to an interface

Client classes/objects remain unaware of the classes of objects they use,
as long as the objects adhere to the interface the client expects

Client classes/objects remain unaware of the classes that implement these objects.
Clients only know about the abstract classes (or interfaces) that define the interface.

Programming to an Interface




```
Collection students = new XXX;
students.add( aStudent);
```

students can be any collection type

We can change our mind on what type to use

Interface & Duck Typing

In dynamically typed languages programming to an interface is the norm

Dynamically typed languages tend to lack a way to declare an interface

Design Principle 2

Favor object composition over class inheritance

Composition

- Allows behavior changes at run time
- Helps keep classes encapsulated and focused on one task
- Reduce implementation dependencies

Inheritance

```
class A {
        Foo x
        public int complexOperation() {    blah }
}

class B extends A {
        public void bar() { blah}
}
```

Composition

```
class B {
        A myA;
        public int complexOperation() {
                return myA.complexOperation()
        }

        public void bar() { blah}
}
```

Designing for Change

Algorithmic dependencies
Builder, Iterator, Strategy,
Template Method, Visitor

Inability to alter classes conveniently
Adapter, Decorator, Visitor

Dependence on specific operations
Chain of Responsibility, Command

Dependence on hardware and software platforms
Abstract factory, Bridge

Tight Coupling
Abstract factory, Bridge, Chain of Responsibility,
Command, Facade, Mediator, Observer

Extending functionality by subclassing
Bridge, Chain of Responsibility, Composite,
Decorator, Observer, Strategy

Dependence on object representations or implementations
Abstract factory, Bridge, Memento, Proxy

Extending functionality by subclassing
Bridge, Chain of Responsibility, Composite,
Decorator, Observer, Strategy

Creating an object by specifying a class explicitly
Abstract factory, Factory Method, Prototype

Kent Beck's Rules for Good Style

One and only once

In a program written in good style, everything is said once and only once

Methods with the same logic
Objects with same methods
Systems with similar objects

rule is not satisfied

Lots of little Pieces

"Good code invariably has small methods and small objects"

Small pieces are needed to satisfy "once and only once"

Make sure you communicate the big picture or you get a mess

Rates of change

Don't put two rates of change together

An object should not have a field that changes every second & a field that change once a month

A collection should not have some elements that are added/removed every second and some that are add/removed once a month

An object should not have code that has to change for each piece of hardware and code that has to change for each operating system

Replacing Objects

Good style leads to easily replaceable objects

"When you can extend a system solely by adding new objects without modifying any existing objects, then you have a system that is flexible and cheap to maintain"

Moving Objects

"Another property of systems with good style is that their objects can be easily moved to new contexts"

Iterator Pattern

Provide a way to access the elements of a collection sequentially without exposing its underlying representation

Java Iterators

External

Iterator

hasNext()

next()

remove() Optional

forEachRemaining

Internal

forEach

ListIterator

add(), remove(), set() Optional

hasNext(), hasPrevious()

next(), previous()

nextIndex(), previousIndex()

SplitIterator

forEachRemaining() + others

For concurrent processing

Java Iterator

```
LinkedList<Strings> strings =  new LinkedList<Strings>();

code to add strings

Iterator<String> list = strings.iterator();
while (list.hasNext()){
    String element = list.next();
    if (element.size % 2 == 0)
    System.out.println(element);
    }
}
```

```
for (String element : strings) {
    if (element.size % 2 == 0)
        System.out.println(element);
}
```

Syntax sugar for above

Python Iterator

```
a = ['house', 'car', 'bike']

for x in a:
        print(x)

items_iterator = iter(a)
print( next(items_iterator))
print( next(items_iterator))
print( next(items_iterator))          #error raised here
```

Lambda & Closure

Lambda

Function without a name

Closure

Store the environment with the function

Lambda Expression - Java 8+

Anonymous Function

(Integer a, Integer b) -> a + b

arguments

body

```
(Integer start, Integer stop) -> {
    for (int k = start; k < stop; k++)
        System.out.println(k);
}
```

Short Version of Lambda Syntax

(String text) -> text.length();

text -> text.length();

(Integer a, Integer b) -> a + b

(a, b) -> a + b

Using Lambdas

Function<String,Integer> length = **text -> text.length();**

int nameLength = **length.apply**("Roger Whitney");

BiFunction<Integer,Integer,Integer> adder = **(a, b) -> a + b;**

int sum = **adder.apply**(1, 2);

Other Types of Lambdas

```
Predicate<Integer> isLarge =  value -> value > 100;
if (isLarge.test(59))
    System.out.println("large");
```

```
Consumer<String> print = text -> System.out.println(text);
print.accept("hello World");
```

```
int size = xxx;
Supplier<List> listType = size > 100 ? (()-> new ArrayList()): (() -> new Vector());
List elements = listType.get();
System.out.println(elements.getClass().getName());
```

Lambda Types

New - See java.util.function Interfaces

- Predicate<T> -- a boolean-valued property of an object
- Consumer<T> -- an action to be performed on an object
- Function<T,R> -- a function transforming a T to a R
- Supplier<T> -- provide an instance of a T (such as a factory)
- UnaryOperator<T> -- a function from T to T
- BinaryOperator<T> -- a function from (T, T) to T

Pre-existing

- java.lang.Runnable
- java.util.concurrent.Callable
- java.security.PrivilegedAction
- java.util.Comparator
- java.io.FileFilter
- java.beans.PropertyChangeListener
- etc.

Functional Interfaces

Interface with one method

Can be used to hold a lambda

java.lang.Runnable

void run()

Runnable Example

```
Runnable test = () -> System.out.println("hello from thread");
Thread example = new Thread(test);
example.start();
```

OnClickListener Example

```
button.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View source) {
        makeToast();
    }
});
```

```
button.setOnClickListener( event -> makeToast());
```

Internal Iterator - forEach

```
String[] rawData = {"cat", "can", "bat", "rat"};
List<String> data = Arrays.asList(rawData);
data.forEach( word ->System.out.println(word) );
```

Lambda Expression - Python

```
inc = lambda n : n + 1
result = inc(11)
print(result)                # 12
```

```
multi_args = lambda a, b : a + b
result = multi_args(1,2)
print(result)                # 3
```

```
def adder(n):
        return lambda k : k + n

add5 = adder(5)
add9 = adder(9)
result = add5(1)
print(result)                    # 6

result = add9(1)
print(result)                    # 10
```

adder shows that Python lambdas are also closures

Motivating Example - Sorting

```
a = ['house', 'car', 'bike']

a.sort()
print(a)
```

['bike', 'car', 'house']

```
a.sort(key = lambda x: len(x))
print(a)
```

['car', 'bike', 'house']

```
a.sort(key = len)
```

Java Sorting

List Method

sort(Comparator<? super E> c)

Sorts this list according to the order induced by the specified Comparator.

```
public class PidCompare implements Comparator<Process> {
    @Override
    public int compare(Process a, Process b) {
        return a.pid() - b.pid();
    }
}
```

```
aList.sort( new PidCompare() );
```

New Options

```
Comparator<Process> compareById = Comparator.comparing(e -> e.pid());

 aList.sort( compareById);



aQueue.sort((Process a, Process b) -> a.pid().compareTo( b.pid()));
```

Documentation

Java lambda Tutorial
http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html

Java 8 Lambdas, Warburton, O'Reilly Media, 2014
http://libproxy.sdsu.edu/login?url=http://proquest.safaribooksonline.com/

Java Iterators

External

Iterator

ListIterator

When you don’t need all items

When complicated logic

```
Iterator<String> list = strings.iterator();
while (list.hasNext()){
    String element = list.next();
    if (element.size % 2 == 0)
    System.out.println(element);
    }
}
```

Internal

forEach

When you want all items

Easier to implement

Pattern Parts

Intent

Motivation

Applicability

Structure

Participants

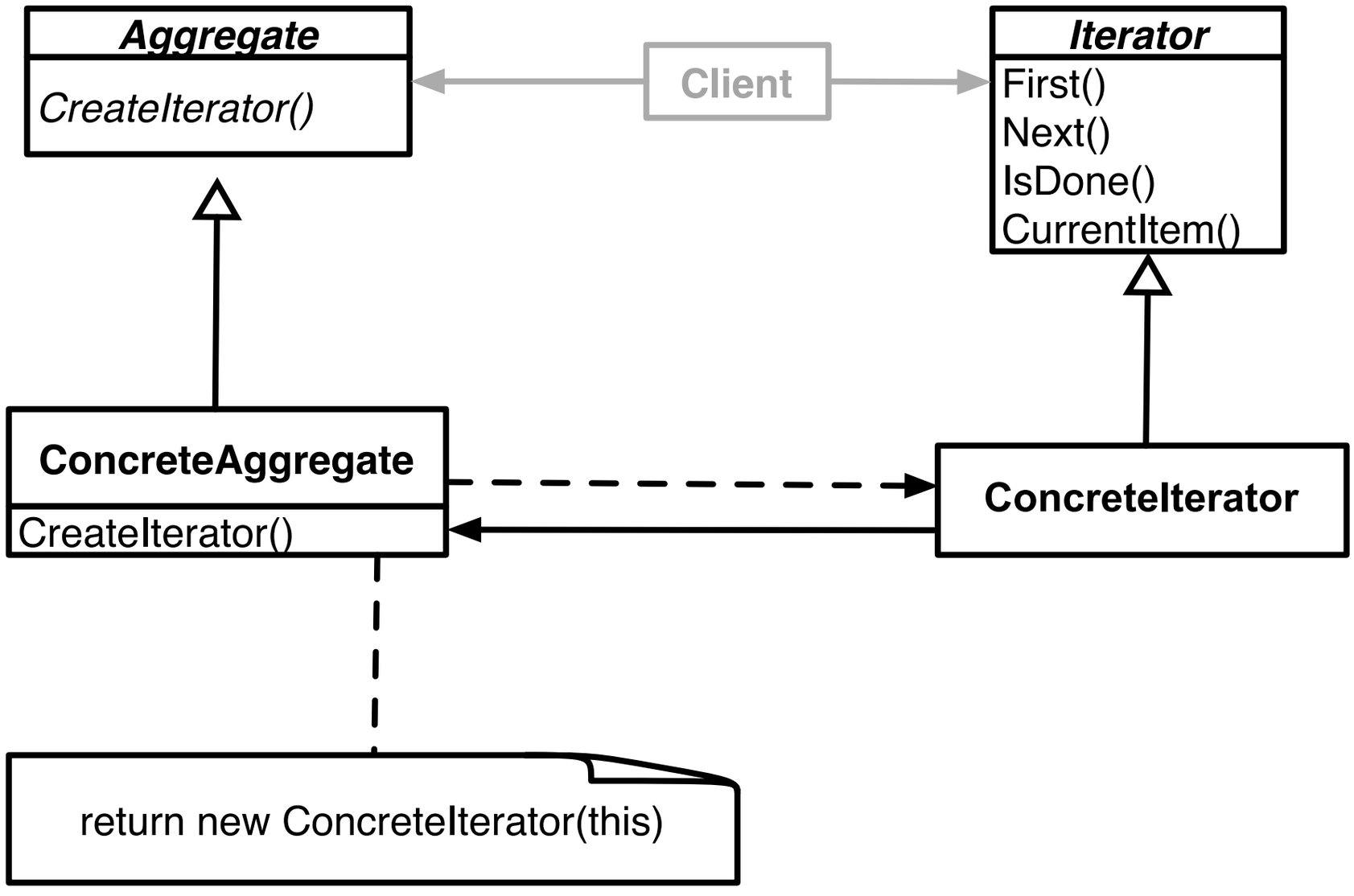
Collaborations

Consequences

Implementation

Sample Code

Iterator Structure

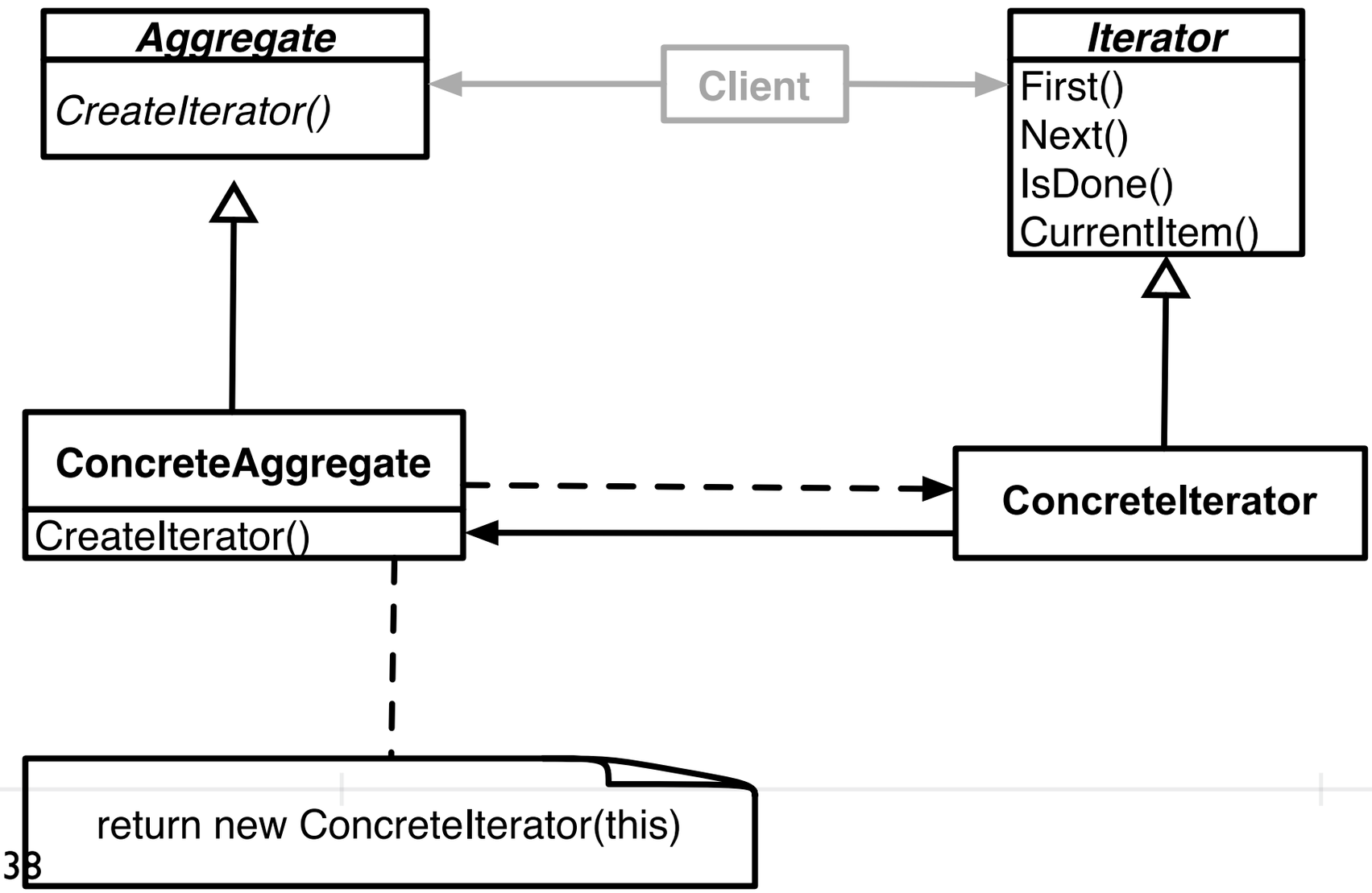


Iterator Structure & Names

Aggregate, ConcreteAggregate, Client, ConcreteIterator

Roles that classes will perform

Classes will not have those names



Issues - Concrete vs. Polymorphic Iterators

Concrete

```
Reader iterator = new StringReader( "cat");
int c;
while (-1 != (c = iterator.read() ))
    System.out.println( (char) c);
```

Polymorphic

```
Vector listOfStudents = new ArrayList();

// code to add students not shown

Iterator list = listOfStudents.iterator();
while ( list.hasNext() )
    System.out.println( list.next() );
```

Memory leak issue in C++, Why?

Issue - Who Controls the Iteration?

External (Active)

```
var numbers = new LinkedList();

code to add numbers

Vector evens = new Vector();
Iterator list = numbers.iterator();
   while ( list.hasNext() ) {
    Integer a = (Integer) list.next();
    int b = a.intValue();
    if ((b % 2) == 0)
        evens.add(a);
   }
```

Internal (Passive)

```
numbers  = LinkedList.new

code to add numbers

evens = numbers.find_all { |element| element.even? }
```

Issue - Who Defines the Traversal Algorithm

Object being iterated

Iterator

Issue - Robustness

What happens when items are added/removed from the iteratee while an iterator exists?

```
Vector listOfStudents = new Vector();

// code to add students not shown

Iterator list = listOfStudents.iterator();
listOfStudents.add( new Student( "Roger") );

list.hasNext();              //What happens here?
```

Java Streams

Stream java.util.stream.Stream

Sequence of values

Operations on the values

Operations are chained together into pipelines

Example

```
String[] words = {"a", "ab", "abc", "abcd", "bat"};
List<String> wordList = Arrays.asList(words);
List<String> longWords
longWords = wordList.stream()
                        .filter( s -> s.length() > 2)
                        .filter( s -> s.charAt(0) == 'a')
                        .map( s -> s.toUpperCase())
                        .collect( Collectors.toList());
System.out.println(longWords);
```

Lazy Evaluation

```
String[] words = {"a", "ab", "abc", "abcd", "bat"};
List<String> wordList = Arrays.asList(words);
List<String> longWords
longWords = wordList.stream()
                    .filter( s -> s.length() > 2)
                    .filter( s -> s.charAt(0) == 'a')
                    .map( s -> s.toUpperCase())
                    .collect( Collectors.toList());
System.out.println(longWords);
```

Only One pass of List to do all operations

4.0 gpa

```
List<Student>  = students.stream()
                                    .filter( student -> student.gpa() >= 4.0)
                                    .collect(Collectors.toList());
```

Stream methods

count()
distinct
filter
findAny
findFirst
flatMap
forEach
forEachOrdered
limit
map
max
min
nonMatch
reduce
sorted

For More Information

State of the Lambda: Libraries Edition

http://cr.openjdk.java.net/~briangoetz/lambda/lambda-libraries-final.html

http://tinyurl.com/mshjfkj

State of the Lambda

http://cr.openjdk.java.net/~briangoetz/lambda/lambda-state-final.html

http://tinyurl.com/kg5m9zu

Ruby Iterator Examples

a = [1, 2, 3, 4]

a.each {|x| puts x}	1 2 3 4
result = a.collect {|x| x + 10} puts result	11 12 13 14
result = a.find_all {|x| x > 2 } puts result	3 4
puts a.any? {|x| x > 2}	true
puts a.detect {|x| x > 2 }	3

Python

```
a = ['house', 'car', 'bike']

def is_even(n) :
        return n % 2 == 0

result = map(len, a)
b = list(result)
print(b)                                    # [5, 3, 4]

even = filter(is_even, map(len, a))
print(list(even))                           # [4]
print(list(even))                           # [ ]
```

Some Higher Order Functions

reduce

Processes a collection to a single value (which could be a collection)

filter

Select elements of a collection

map

Transforms elements of a collection

reduce

Common pattern

loop through a collection to compute some result

```
let data = [1,1,2,3,5,8]

var sum = 0
for n in data {
    sum += n
}
```

```
let easySum = data.reduce(0, +)
```

```
var product = 1
for n in data {
    product *= n
}
```

```
let easyProduct = data.reduce(1, *)
```

More Reduce Examples

```
let words = ["The", "cat", "in", "the", "hat"]

let title = words.reduce("", {$0 + " " + $1})                    // “ The cat in the hat”


let data = [1,8,1,2,3,5]
let maxElement = data.reduce(data[0], {max($0, $1)})
```

Filter

```
let data = [1,8,1,2,3,5]

let foo = data.filter( {$0 > 3})        // [ 8, 5]

let foo = data.filter {$0 > 3}          // don’t need the ( )


let largeSum = data.filter {$0 > 3}
                   .reduce(0, +)
```

Rust Example

```
pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    let mut results = Vec::new();

    for line in contents.lines() {
        if line.contains(query) {
            results.push(line);
        }
    }
    results
}
```

```
test bench_search_for  ... bench:  19,620,300 ns/iter (+/- 915,700)
test bench_search_iter ... bench:  19,234,900 ns/iter (+/- 657,200)
```

```
pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    contents
        .lines()
        .filter(|line| line.contains(query))
        .collect()
}
```

map

```
let data = [1,1,2,3,5,8]

let fiveAdded = data.map{$0 + 5}                    // [6, 6, 7, 8, 10, 13] Swift 2.3 & 3.0



let students = ["Sam": 3.2, "Pete": 3.9, "Jill": 3.7]

let scores = students.map({$0.1})                   // [3.2, 3.9, 3.7]

let sumOfSquares = data.map {$0 * $0}.reduce(0, +)          // 104
```

Selecting 6 most recent Photos in transcript

```
var photos: [PhotoItem] = [ ]
for time in transcript.reversed() {
    if let photo = item as? PhotoItem {
        photos.append(photo)
        if photos.count == 6 {
            break
        }
    }
}
```

```
transcript
    .reversed()
    .compactMap { $0 as? PhotoItem }
    .prefix(6)
```

```
transcript
    .compactMap { $0 as? PhotoItem }
    .suffix(6)
    .reversed()
```