Принципы проектирования и дизайна ПО

Лекция №11

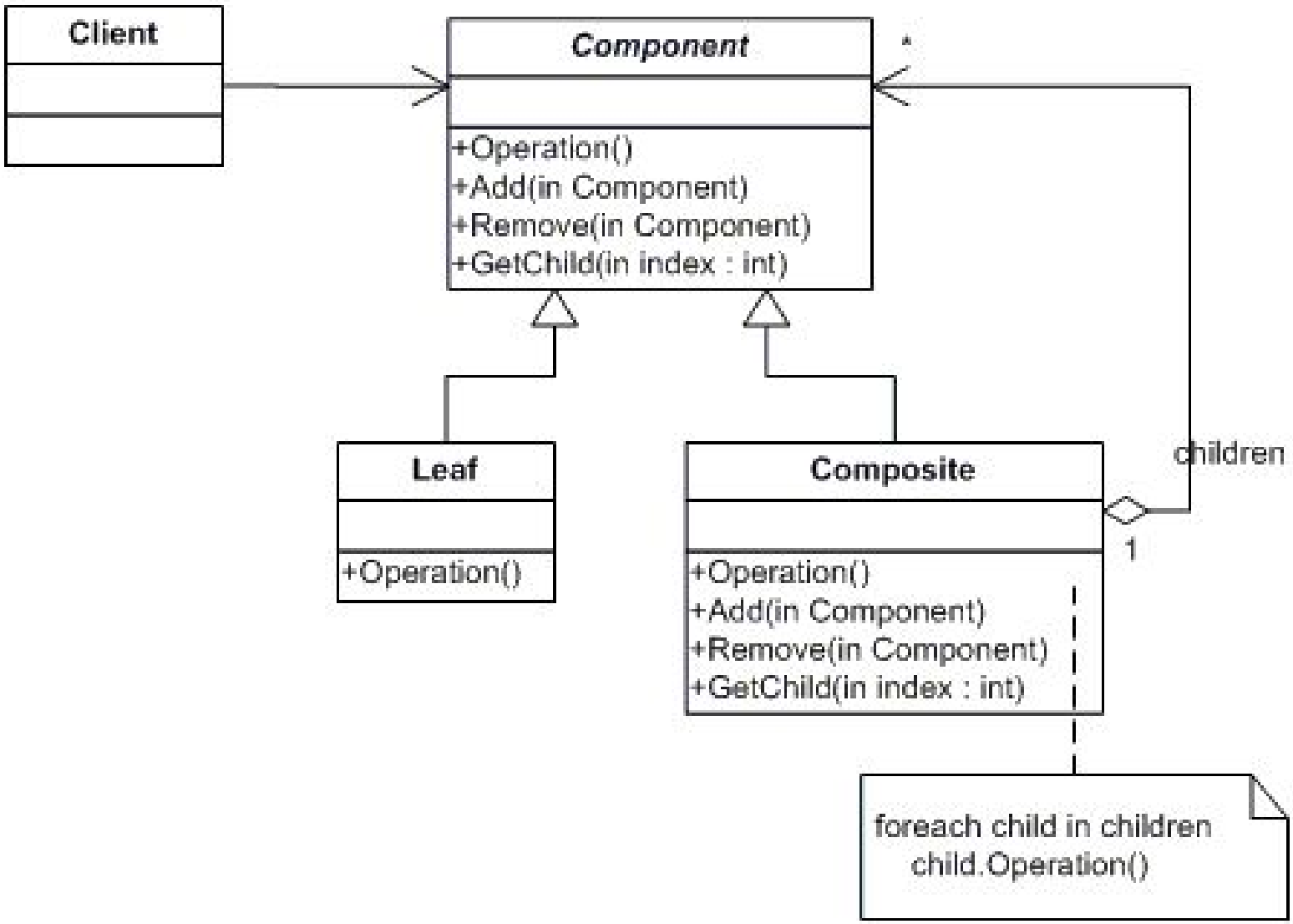
Агошков Илья 2016

Composite

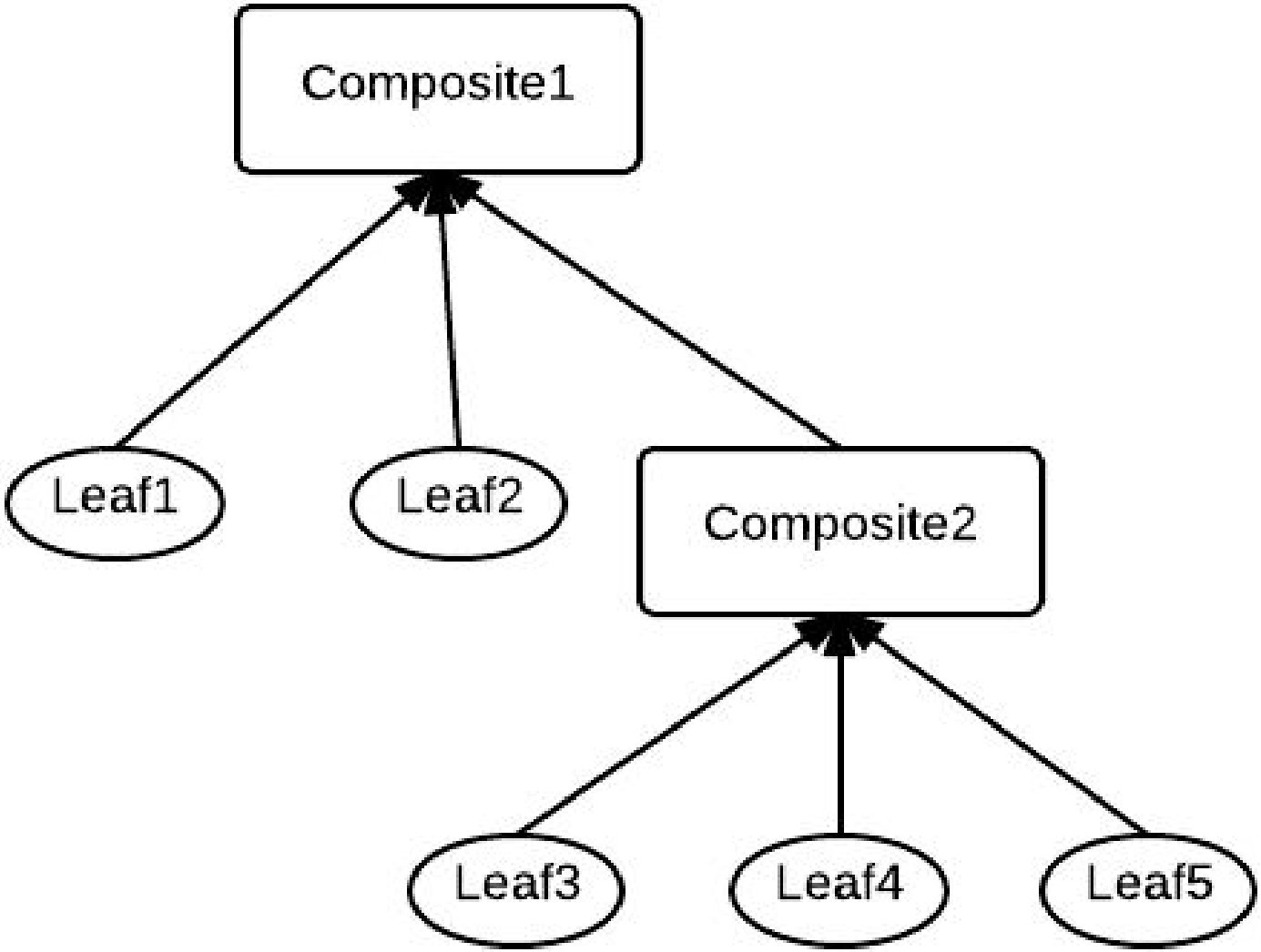
Intent

Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

Composite


Client
Component
*
+Operation()
+Add(in Component)
+Remove(in Component)
+GetChild(in index : int)
Leaf
+Operation()
Composite
+Operation()
+Add(in Component)
+Remove(in Component)
+GetChild(in index : int)
children
1
foreach child in children
child.Operation()

Composite



Composite

Applicability

Use the Composite pattern when
• you want to represent part-whole hierarchies of objects.
• you want clients to be able to ignore the difference between compositions of objects and individual objects. Clients will treat all objects in the composite structure uniformly.

Decorator

Intent

Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Also known as

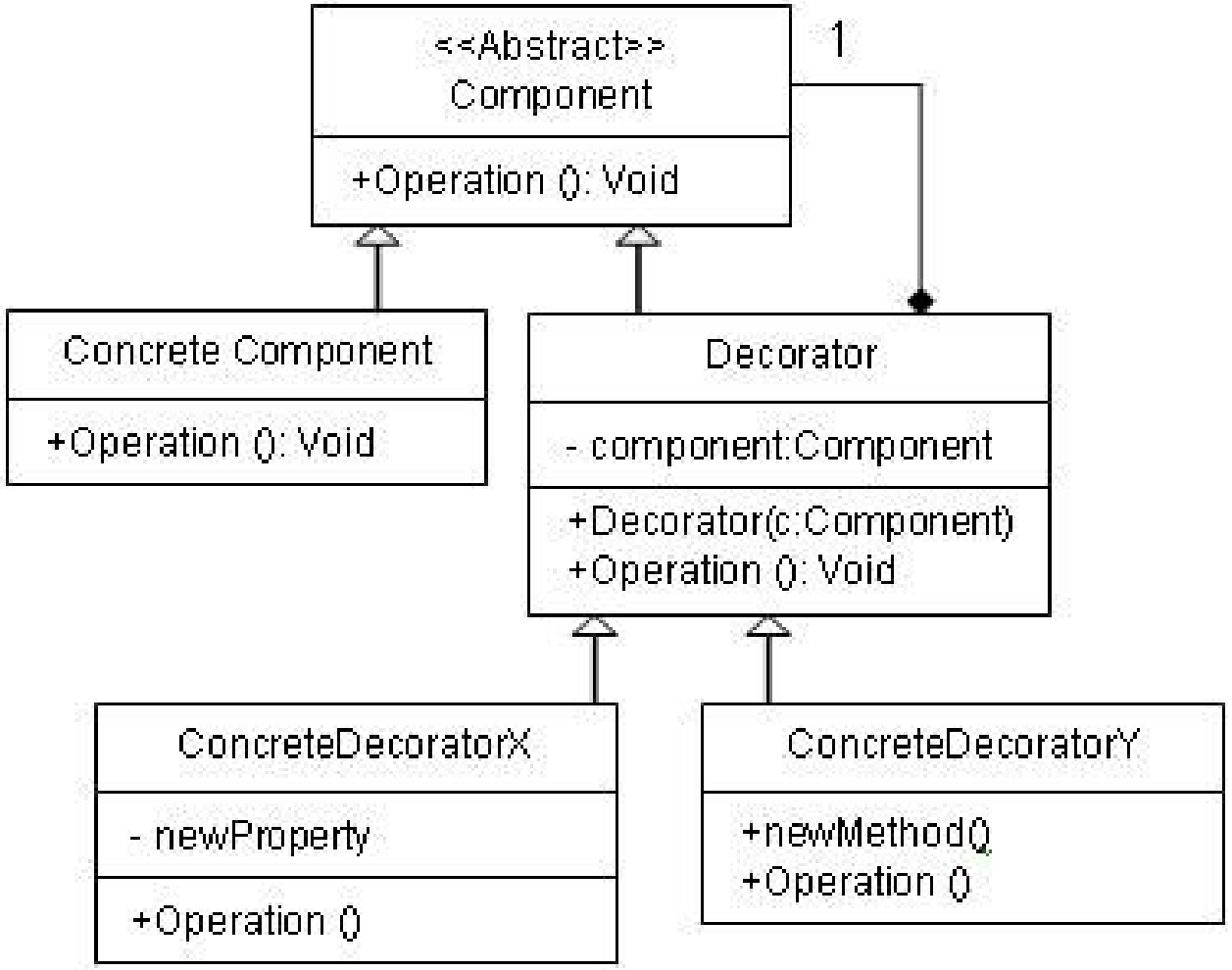
Wrapper

Decorator

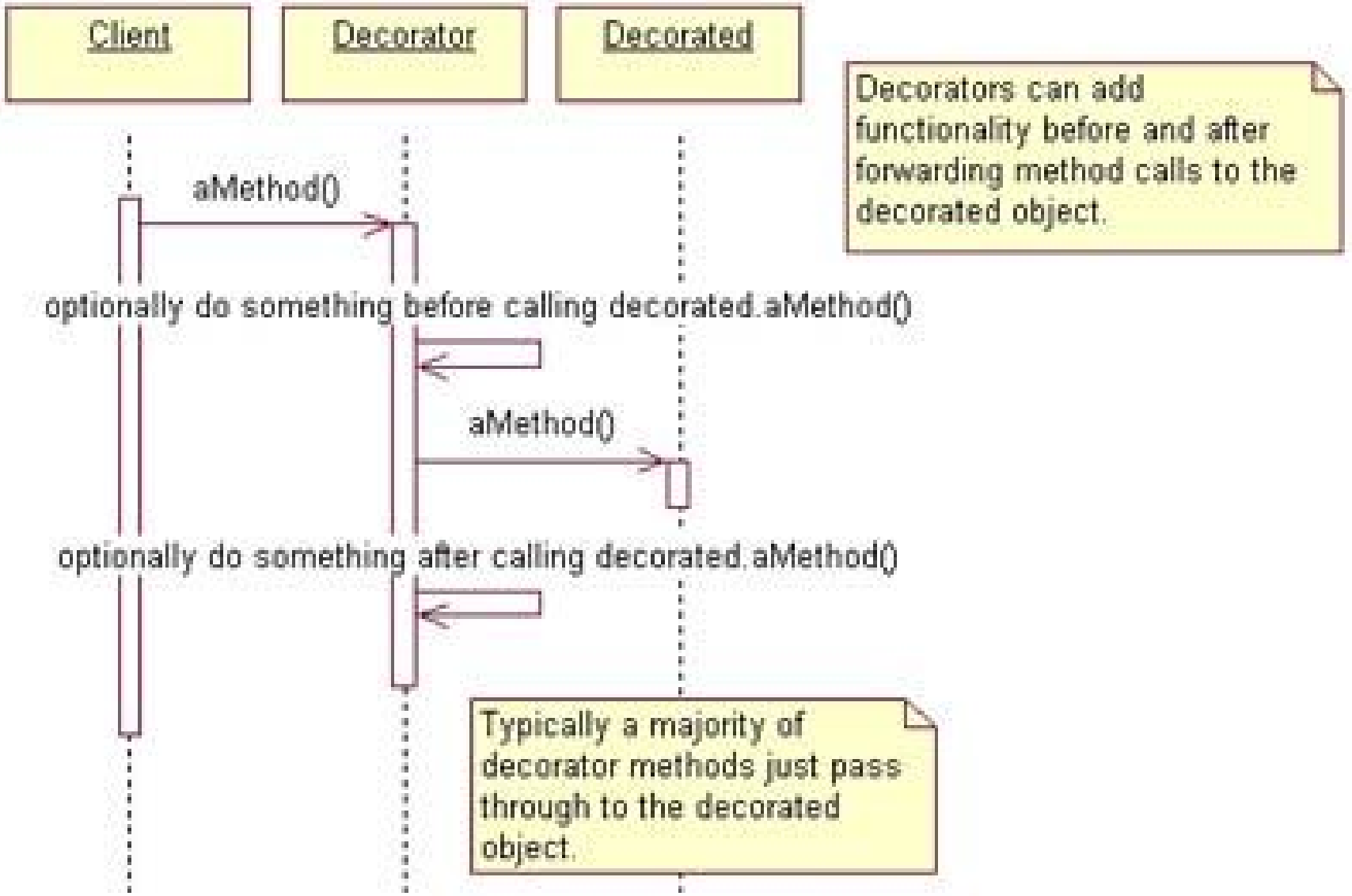
Applicability

Use Decorator
• to add responsibilities to individual objects dynamically and transparently, that is, without affecting other objects.
• for responsibilities that can be withdrawn.
• when extension by subclassing is impractical. Sometimes a large number of independent extensions are possible and would produce an explosion of subclasses to support every combination. Or a class definition may be hidden or otherwise unavailable for subclassing.

Decorator


<<Abstract>>
Component
+Operation (): Void
1
Concrete Component
+Operation (): Void
Decorator
- component:Component
+Decorator(c:Component)
+Operation (): Void
ConcreteDecoratorX
- newProperty
+Operation ()
ConcreteDecoratorY
+newMethod()
+Operation ()

Decorator


Client
Decorator
Decorated
Decorators can add functionality before and after forwarding method calls to the decorated object.
aMethod()
optionally do something before calling decorated.aMethod()
aMethod()
optionally do something after calling decorated.aMethod()
Typically a majority of decorator methods just pass through to the decorated object.

Unit

```
public interface Unit {
  void draw();
  int getHealth();
  int getEnergy();
  int getAmmo();
}
```

Tank

```
public class Tank implements Unit {
  public void draw() {
    System.out.println(“The Tank”);
  }
  public int getHealth() {
    return 200;
  }
}
```

Soldier

```
public class Soldier implements Unit {
  public void draw() {
    System.out.println(“The Soldier”);
  }
  public int getHealth() {
    return 50;
  }
}
```

Draw tank and soldier with health

Decorator

Consequences

1. More flexibility than static inheritance.
2. Avoids feature-laden classes high up in the hierarchy.
3. A decorator and its component aren’t identical.
4. Lots of little objects.

Iterator

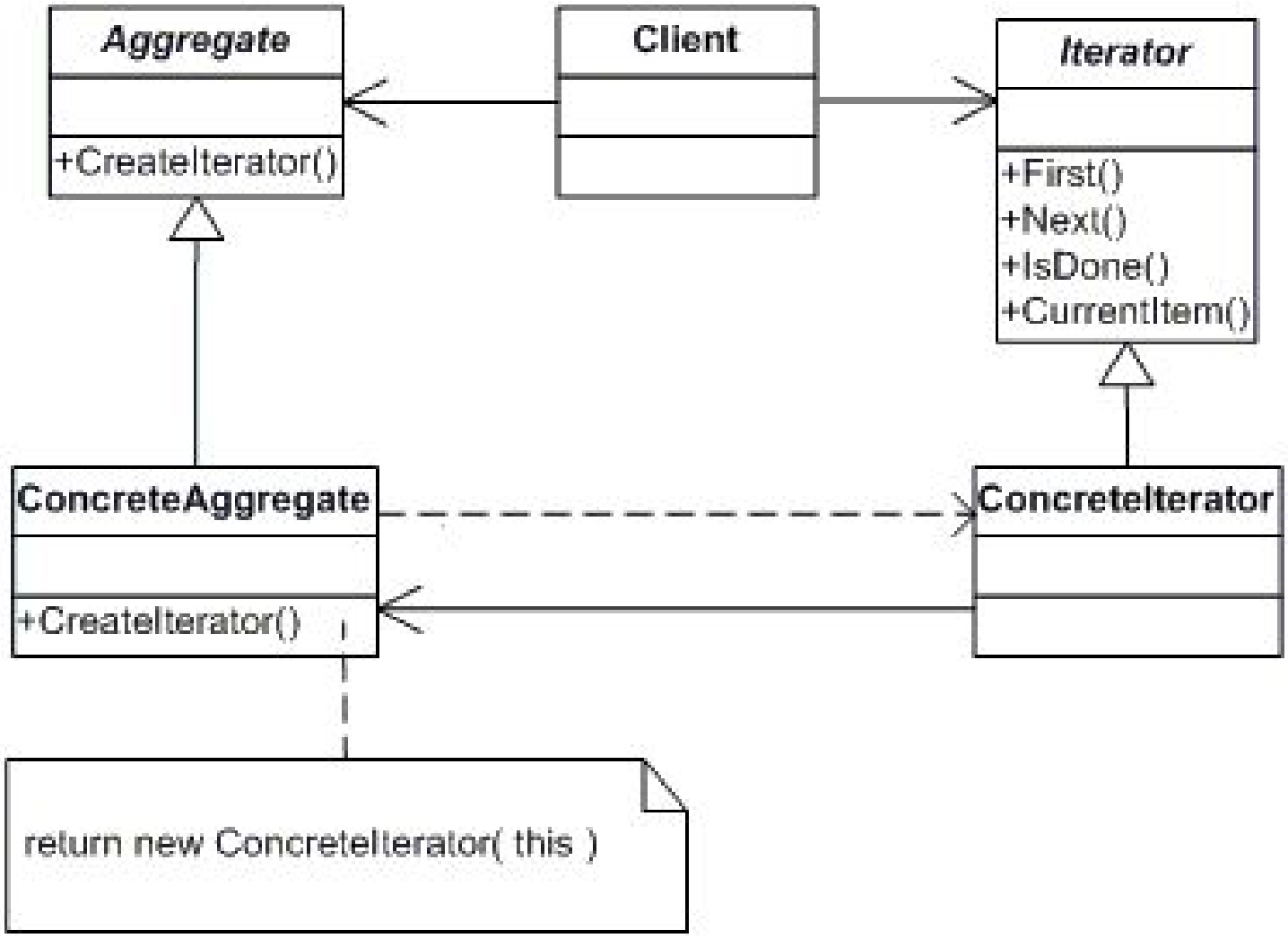
Intent

Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Also known as

Cursor

Iterator


Aggregate
+CreateIterator()
Client
Iterator
+First()
+Next()
+IsDone()
+CurrentItem()
ConcreteAggregate
+CreateIterator()
ConcreteIterator
return new ConcreteIterator(this)

Iterator

Implementation details

internal vs external
different traversals (back/forward)
more than one traversal at the same time
robustness + multithreading
additional operations
iterators may have “privileged access”
iterators for composites
null iterators