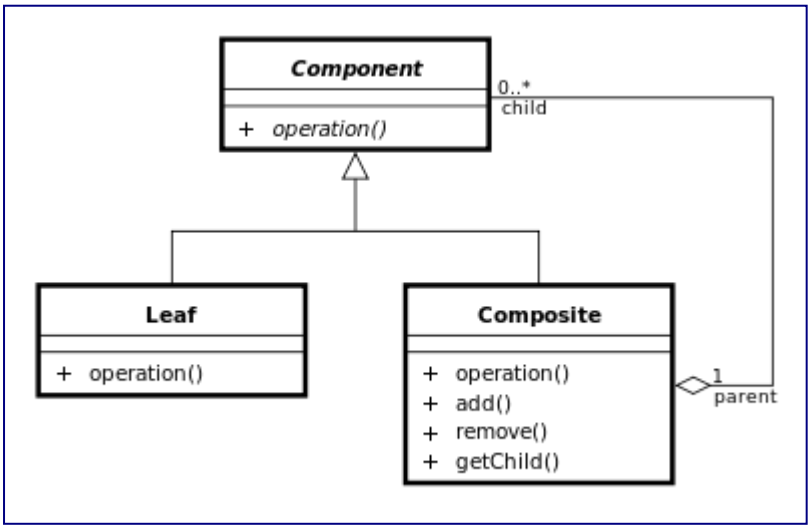
63. Реалізувати шаблони безпечний Composite та внутрішній Iterator для обходу “в глибину” ієрархічних структур на його основі.

Компонувальник **Composite** — структурний шаблон який об'єднує об'єкти в ієрархічну деревовидну структуру, і дозволяє уніфіковане звертання для кожного елемента дерева.

Призначення
Дозволяє користувачам будувати складні структури з простіших компонентів. Проектувальник може згрупувати дрібні компоненти для формування більших, які, в свою чергу, можуть стати основою для створення ще більших.

Структура



Ключем до паттерну компонувальник є абстрактний клас, який є одночасно і примітивом, і контейнером(Component). У ньому оголошені методи, специфічні для кожного виду об'єкта (такі як Operation) і загальні для всіх складових об'єктів, наприклад операції для доступу і управління нащадками. Підкласи Leaf визначає примітивні об'єкти, які не є контейнерами. У них операція Operation реалізована відповідно до їх специфічних потреб. Оскільки у примітивних об'єктів немає нащадків, то жоден з цих підкласів не реалізує операції, пов'язані з управління нащадками (Add, Remove, GetChild). Клас Composite складається з інших примітивніших об'єктів Component. Реалізована в ньому операція Operation викликає однойменну функцію відтворення для кожного нащадка, а операції для роботи з нащадками вже не порожні. Оскільки інтерфейс класу Composite відповідає інтерфейсу Component, то до складу об'єкта Composite можуть входити і інші такі ж об'єкти.

Учасники

- Component (Component)

Оголошує інтерфейс для компонуемих об'єктів; Надає відповідну реалізацію операцій за замовчуванням, загальну для всіх класів; Оголошує єдиний інтерфейс для доступу до нащадків та управління ними; Визначає інтерфейс для доступу до батька

компонента в рекурсивної структурі і при необхідності реалізує його (можливість необов'язкова);

- Leaf (Leaf_1, Leaf_2) — лист.

Об'єкт того ж типу що і Composite, але без реалізації контейнерних функцій; Представляє листові вузли композиції і не має нащадків; Визначає поведінку примітивних об'єктів в композиції; Входить до складу контейнерних об'єктів;

- Composite (Composite) — складений об'єкт.

Визначає поведінку контейнерних об'єктів, у яких є нащадки; Зберігає ієрархію компонентів-нащадків; Реалізує пов'язані з управлінням нащадками (контейнерні) операції в інтерфейсі класу Component;

Про шаблон Ітератор – дивись питання 65.

Код програми

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;


   /**
    * Pattern Composite + Iterator
    */

   public class Main {

       public static void main(String[] args) {

           //Initialize 8 areas
           Area a1 = new Area();
           Area a2 = new Area();
           Area a3 = new Area();
           Area a4 = new Area();
           Area a5 = new Area();
           Area a6 = new Area();
           Area a7 = new Area();
           Area a8 = new Area();

           //Initialize 3 composite levels
           CompositeLevel level1 = new CompositeLevel();
           CompositeLevel level2 = new CompositeLevel();
           CompositeLevel level3 = new CompositeLevel();

           //Initialize first level
           Level[] FirstLevel = new Level[3];
           FirstLevel[0] = level1;
           FirstLevel[1] = level2;
           FirstLevel[2] = level3;

           //Allocation of areas on the levels
           level1.add(a3);
           level1.add(a7);
           level1.add(a2);

           level2.add(a1);
           level2.add(a6);

           level3.add(a5);
           level3.add(a8);
           level3.add(a4);
```

```
        ConcreteAggregate conAgr = new ConcreteAggregate(list);
        Iterator<Integer> iterator = conAgr.createIterator();

        while(iterator.isDone()){
              System.out.println(iterator.next());
        }
        System.out.println();

        ArrayList<Integer> sortedList = new ArrayList<Integer>(list);
        Collections.sort(sortedList);
        ConcreteAggregate conAgr2 = new ConcreteAggregate(sortedList);
        Iterator<Integer> iterator2 = conAgr2.createIterator();

        while(iterator2.isDone()){
              System.out.println(iterator2.next());
    }
}

import java.util.List;
import java.util.ArrayList;

/**
 * Class-composite
 */

class CompositeLevel implements Level {

    //Collection of the levels
    private List<Level> myLevels = new ArrayList<Level>();

    //Add the level to the composition
    public void add(Level level) {
        myLevels.add(level);
        System.out.println("Level added.");
    }

    //Remove the level from the composition
    public void remove(Level level) {
      myLevels.remove(level);
      System.out.println("Level removed.");
    }
}

public interface Iterator<Integer> {

   public Integer first();
   public Integer next();
   public boolean isDone();
   public Integer currentItem();
}

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public interface Aggregate {

   public Iterator<Integer> createIterator();
}

/**
 * Interface-component
 */

interface Level {
}
```

```
import java.util.ArrayList;

public class ConcreteIterator implements Iterator {

   ConcreteAggregate list;
   public ConcreteIterator(ConcreteAggregate concreteAggregate){
         this.list = concreteAggregate;
   }

   private int index;

   @Override
   public Object first() {
         return list.get(0);
   }

   @Override
   public Object next() {
         return list.get(++index);
   }

   @Override
   public boolean isDone() {
         return index < list.size()-1;
   }

   @Override
   public Object currentItem() {
         return list.get(index);
   }
}

import java.util.ArrayList;

public class ConcreteAggregate implements Aggregate{
   protected ArrayList<Integer> list;

   public ConcreteAggregate(ArrayList list){
         this.list = list;
   }

   public Iterator<Integer> createIterator() {
         return new ConcreteIterator(this);
   }

   public Integer get(int i) {
         return list.get(i);
   }

    public int size(){
         return list.size();
    }
}

/**
 * Class-leaf
 */

class Area implements Level {
}

public class SomeClass{

    public SomeClass() {
    }
}
```

64. Управління вимогами.

Вимоги – це властивості, які повинно мати програмне забезпечення для адекватного визначення функцій, умов та обмежень виконання ПЗ, а також об'ємів даних, технічного забезпечення і середовища функціонування.

Типи вимог:

- *Програмні* вимоги — Software Requirements — властивості програмного забезпечення, які повинні бути належним чином представлені в ньому для вирішення конкретних практичних задач. Дана галузь знань стосується питань вилучення (збору), аналізу, специфицирования і затвердження вимог.
- *Користувальницькі* вимоги - визначають головну мету системи і, як мінімум, відповідять на наступні **питання:**
  - ✓ Вимоги експлуатації або розгортання: Де система буде використовуватися?
  - ✓ Профіль місії або сценарій: Як система досягне цілей місії?
  - ✓ Вимоги продуктивності: Які параметри системи є критичними для досягнення місії?
  - ✓ Сценарії використання: Як різні компоненти системи повинні використовуватися?
  - ✓ Вимоги ефективності: Наскільки ефективною має бути система для виконання місії?
  - ✓ Експлуатаційний життєвий цикл: Як довго система буде використовуватися?
  - ✓ Навколишнє середовище: Яким оточенням система повинна буде ефективно управляти?
- *Функціональні* вимоги пояснюють, що повинно бути зроблено. Вони ідентифікують завдання або дії, які повинні бути виконані. Функціональні вимоги визначають дії які система повинна бути здатна виконувати, зв'язок входу / виходу в поведінці системи.
- *Нефункціональні* вимоги — вимоги, які визначають критерії роботи системи в цілому, а не окремі сценарії поведінки. Нефункціональні вимоги визначають системні властивості такі як продуктивність, зручність супроводу, розширюваність, надійність, середовищні фактори експлуатації.
- Вимоги *продуктивності* — ступінь, до якої повинні бути виконані місія або функція; в загальному випадку виміряні з точки зору кількості, якості, покриття, своєчасності або готовності.
- *Похідні* вимоги — Вимоги, які маються на увазі або перетворені з високорівневих вимог. Наприклад, вимога для більшого радіусу дії або високої швидкості може призвести до вимоги низької ваги.

Управління вимогами до ПЗ передбачає контроль за виконанням вимог і планування використання ресурсів (людських, програмних, технічних, періодичних, вартісних) в процесі розробки проміжних робочих продуктів на етапах життєвого циклу.

65. Шаблон Iterator. Призначення, структура, учасники. Яким чином клієнт, не знаючи конкретного агрегату створює ітератор, що здатний працювати із агрегатом?

Ітератор (англ. Iterator) — шаблон проектування, належить до класу шаблонів поведінки.

Призначення

Надає спосіб послідовного доступу до всіх елементів складеного об'єкта, не розкриваючи його внутрішнього улаштування.

Мотивація

Складений об'єкт, скажімо список, повинен надавати спосіб доступу до своїх елементів, не розкриваючи їхню внутрішню структуру. Більш того, іноді треба обходити список по-різному, у залежності від задачі, що вирішується. При цьому немає ніякого бажання засмічувати інтерфейс класу Список усілякими операціями для усіх потрібних варіантів обходу, навіть якщо їх усі можна передбачити заздалегідь. Крім того, іноді треба, щоб в один момент часу існувало декілька активних операцій обходу списку.

Все це призводить до необхідності реалізації шаблону Ітератор. Його основна ідея у тому, щоб за доступ до елементів та обхід списку відповідав не сам список, а окремий об'єкт-ітератор. У класі Ітератор означений інтерфейс для доступу до елементів списку. Об'єкт цього класу прослідковує поточний елемент, тобто він володіє інформацією, які з елементів вже відвідувались.

Застосовність

Можна використовувати шаблон Ітератор у випадках:

- для доступу до змісту агрегованих об'єктів не розкриваючи їхнє внутрішнє улаштування;
- для підтримки декількох активних обходів одного й того ж агрегованого об'єкта;
- для подання уніфікованого інтерфейсу з метою обходу різноманітних агрегованих структур (тобто для підтримки поліморфної ітерації).

Структура

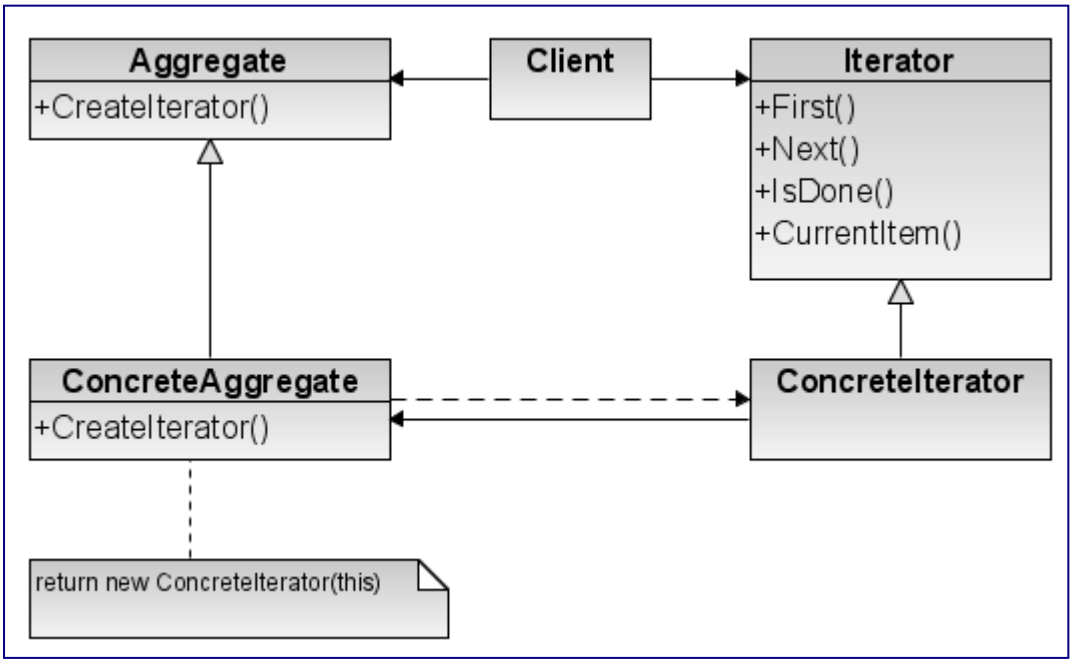


Рис. 9.1
UML діаграма, що описує структуру шаблону проектування Ітератор

- Iterator
  - визначає інтерфейс для доступу та обходу елементів
- ConcreteIterator
  - реалізує інтерфейс класу Iterator;
  - слідкує за поточною позицією під час обходу агрегату;
- Aggregate
  - визначає інтерфейс для створення об'єкта-ітератора;
- ConcreteAggregate
  - реалізує інтерфейс створення ітератора та повертає екземпляр відповідного класу ConcreteIterator

Відносини

ConcreteIterator відслідковує поточний об'єкт у агрегаті та може вирахувати наступний.

66. Реалізувати шаблони безпечний Composite та Iterator-курсор для обходу “в ширину” ієрархічних структур на його основі.

Оскільки завдання абсолютно аналогічне до питання 63, списуй інформацію звідти, і все буде добре!