Iterator

Cel:

Uniwersalny sposób na sekwencyjne przeglądanie elementów agregacji, bez ujawniania jej reprezentacji.

Przykład:

```
class ArrayList {
     private Object[] ar;
     public int size() { return ar.length; }
     public Object get(int idx) { return ar[idx]; }
     ...
}
class Client {
     public static void main(...) {
          ArrayList al = new ArrayList();
          ...
          for(int i = 0; i < al.size(); ++i)
               System.out.println(al.get(i));
     }
}
```

```
// ale tę listę powinniśmy przeglądać w inny sposób
class LinkedList {
      private static class Element { public Object data;
                                     public Element next; }
      private Element first;
      public int size() {
            int s = 0;
            for(Element p = first; p != null; p = p.next, s++);
            return s;
      }
      public Object get(int idx) {
            int s = 0;
            for(Element p = first; p != null; p = p.next, s++)
                  if(s == idx) return p.data;
            return null;
      }
}
class Client {
      public static void main(...) {
            LinkedList ll = new LinkedList();
            ...
            for(int i = 0; i < ll.size(); ++i)
                  System.out.println(ll.get(i));
      }
}
```

```
// ale tę listę powinniśmy przeglądać w inny sposób
class LinkedList {
     public static class Element { public Object data;
                                   public Element next; }
     private Element first;
     public Element getFirst() {
          return first;
     }
     ...
}
class Client {
     public static void main(...) {
          LinkedList ll = new LinkedList();
          ...
          LinkedtList.Element p = ll.getFirst();
          while(p != null) {
               System.out.println(p.data);
               p = p.next;
          }
     }
}
```

```
// a jak bez ujawniania implementacji?
class LinkedList {
     private static class Element { public Object data;
                                    public Element next; }
     private Element first, actual;
     public void moveFirst() { actual = first; }
     public boolean moveNext() {
          actual = actual.next;
          return actual != null;
     }
     public Object getActual() { return actual.data; }
     ...
}
class Client {
     public static void main(...) {
          LinkedList ll = new LinkedList();
          ...
          ll.moveFirst();
          do {
               System.out.println(ll.getActual());
          } while(ll.moveNext());
     }
}
```

```
// jeszcze lepsze rozwiązanie – iterator
class LinkedList {
     private static class Element { public Object data;
                                    public Element next; }
     private Element first;
     public Iterator iterator() { return ......; }
     ...
}
interface Iterator {
     boolean hasNext();
     Object next();
}
class Client {
     public static void main(...) {
          LinkedList ll = new LinkedList();
          ...
          Iterator it = ll.iterator();
          while(it.hasNext())
          {
               System.out.println(it.next());
          }
     }
}
```

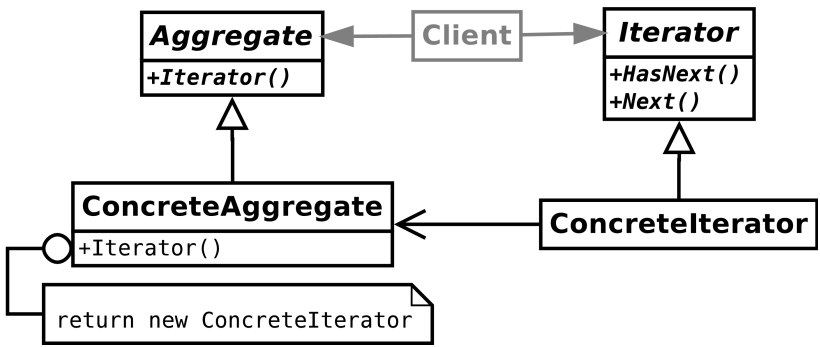
```
// a jak działa iterator? – w każdej klasie inaczej
class ArrayList {
     private Object[] ar;
     private class ArrayListIterator implements Iterator {
          private int idx = 0;
          public boolean hasNext() {
               return idx < ar.length;
          }
          public Object next() {
               Object data = ar[idx];
               idx++;
               return data;
          }
     }
     public Iterator iterator() {
          return new ArrayListIterator();
     }
     ...
}
```

```
// a jak działa iterator? – w każdej klasie inaczej
class LinkedList {
     private static class Element { public Object data;
                                    public Element next; }
     private Element first;
     private class LinkedListIterator implements Iterator {
          private Element actual = first;
          public boolean hasNext() {
               return actual != null;
          }
          public Object next() {
               Object data =  actual.data;
               actual = actual.next;
               return data;
          }
     }
     public Iterator iterator() {
          return new LinkedListIterator();
     }
     ...
}
```

Zastosowanie:

- Aby uzyskać dostęp do obiektów agregacji (kolekcji) bez ujawniania jej wewnętrznej reprezentacji.
- Aby umożliwić kilka niezależnych iteracji jednej kolekcji.
- Aby zapewnić jednolity interfejs iteracji dla różnych rodzajów agregacji (polimorficzna iteracja).

Struktura:



Składniki:

- Iterator – definiuje interfejs dający dostęp do kolejnych elementów
- ConcreteIterator – implementuje interfejs Iteratora, zapamiętuje aktualną pozycję iteracji
- Aggregate – definiuje interfejs tworzenia iteratora
- ConcreteAggregate – zwraca konkretnego Iteratora

Zależności:

ConcreteIterator przechowuje aktualną pozycję iteracji i potrafi znaleźć pozycję następną.

Konsekwencje:

1. Umożliwia różnicowanie algorytmów iteracji – złożone agregacje mogą być przeglądane na kilka różnych sposobów (np. w różnym porządku), aby to zapewnić wystarczy wymienić iteratora.
2. Upraszcza interfejs agregacji – metody odpowiedzialne za iterację znajdują się w innej klasie.
3. Możliwa więcej iż jedna iteracja przez strukturę jednocześnie, gdyż to sam iterator pamięta o stanie iteracji.

Implementacja:

1. Iterator z powyższych przykładów, to **iterator zewnętrzny** – klient zajmuje się pobraniem iteratora i jawnie wywołuje metody zwracające element i przechodzące do kolejnego obiektu. Inny rodzaj iteratora – **iterator wewnętrzny** – otrzymuje jedynie operację, którą ma wykonać na wszystkich elementach struktury (wzorzec *Visitor*).
2. Rodzaj iteratora – **kursor**. Algorytm iteracji nie znajduje się w iteratorze, a agregacji. Iterator to jedynie wskazanie na aktualny element (wzorzec *Memento* – zachowuje aktualny stan iteracji).

```
interface Cursor { }
class ArrayList {
     private class Index implements Cursor {
          int idx;
          Index(int idx) { this.idx = idx; }
     }
     private Object[] ar;
     public Cursor first() { return new Index(0); }
     public boolean end(Cursor c)
     {    return ((Index)c).idx >= ar.length; }
     public Object getNext(Cursor c)
     {    return ar[((Index)c).idx++]; }
}
```

```
class Client {
     public static void main(...) {
          ArrayList lista = new ArrayList(10);
          for(Cursor c = lista.first(); !lista.end(c); )
          {
               System.out.println(lista.getNext(c));
          }
     }
}
```

3. Modyfikacja agregacji (dodawanie/ usuwanie elementów) w trakcie jej przeglądania może być niebezpieczna. Rozwiązanie:
   - modyfikacja struktury powinna unieważniać iteratory (np. wzorzec *Observer*), próba dalszej iteracji spowoduje wyrzucenie wyjątku:

```
Iterator it = list.iterator();
while(it.hasNext()) {
     Point o = it.next();
     if(o.color() == Colors.red)
          list.remove(o);
}
```

```
Iterator it = list.iterator();
     Point o = it.next();
     o = it.next();
     o = it.next();
     list.remove(o);
     o = it.next();
```

- (trudniejsze) agregacja powinna aktualizować iteratory, aby zapewnić ciągłość iteracji
- (pośrednie) iterator może sam udostępniać pewne operacje modyfikacji agregacji:

```
Iterator it = list.iterator();
while(it.hasNext()) {
      Point o = it.next();
      if(o.color() == Colors.red)
            it.remove();
}
```

4. Dodatkowe operacje Iteratora: first (skok na początek), previous (poprzedni element), skipTo (skok do wskazanego elementu).
5. Iterator jako klasa wewnętrzna w Javie: ma dostęp do składowych prywatnych obiektu klasy zewnętrznej:

```
class LinkedList {
      ...
      private Element first;
      private class LinkedListIterator implements Iterator {
            private Element actual = first;
            ...
      }
      ...
}
```

6. Iterator jako klasa anonimowa w Javie:

```
class LinkedList {
     ...
     public Iterator iterator() {
          return new Iterator() {
               public boolean hasNext() { ... }
               public Object next() { ... }
               ...
          };
     }
     ...
}
```

7. Iteratory kompozytów – zewnętrzne (przechowujące ścieżkę do elementu) mogą być trudne w implementacji, prościej jest użyć wewnętrznego iteratora (z wywołaniem rekurencyjnym) lub kursora (przechowującego węzeł drzewa, który daje dostęp do dzieci, rodzeństwa i rodziców).

Powiązania:

- *Composite* – iteratory są często używane do przeglądania kompozytów.
- *Factory Method* – polimorficzne iteratory są tworzone przez metody fabrykujące.
- *Memento* – iterator może używać memento do przechowania stanu iteracji.