Composite (kompozyt)

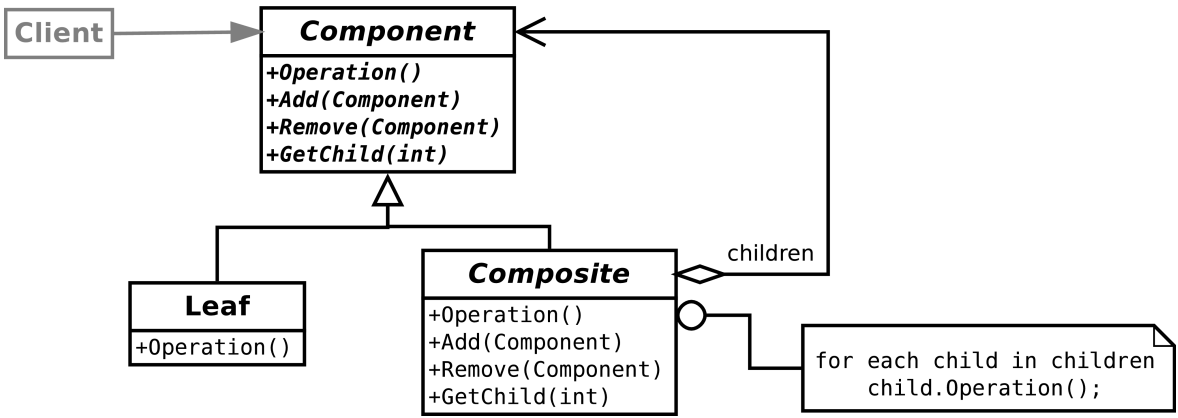
Cel:

Łączenie obiektów w drzewiastą strukturę do reprezentacji hierarchii typu część-całość. Pozwala klientom jednakowo traktować kompozycję obiektów, jak i pojedyncze obiekty.

Przykład:

```
interface ElementGraficzny {
     void rysuj(Ekran e);
}
class Linia implements ElementGraficzny {
     public void rysuj(Ekran e) { /* ... */ }
}
class Prostokąt implements ElementGraficzny {
     public void rysuj(Ekran e) { /* ... */ }
}
class Tekst implements ElementGraficzny {
     public void rysuj(Ekran e) { /* ... */ }
}
class Grupa { /* ??? */ }
```

Struktura:


Client
Component
+Operation()
+Add(Component)
+Remove(Component)
+GetChild(int)
children
Leaf
+Operation()
Composite
+Operation()
+Add(Component)
+Remove(Component)
+GetChild(int)
for each child in children
child.Operation();

Składniki:

- ***Komponent***
  - deklaruje interfejs obiektów w kompozycji
  - implementuje domyślne zachowanie we wspólnym interfejsie
  - deklaruje interfejs dostępu do dzieci
  - (opcjonalnie) definiuje interfejs dostępu do ojca
- ***Liść***
  - reprezentuje obiekt liścia w kompozycji (komponent bez dzieci)
  - definiuje zachowanie obiektów podstawowych
- ***Kompozyt***
  - definiuje zachowanie komponentów z dziećmi
  - przechowuje komponenty-dzieci
  - implementuje operacje zarządzania dziećmi
- ***Klient***
  - posługuje się obiektami przez interfejs Komponent

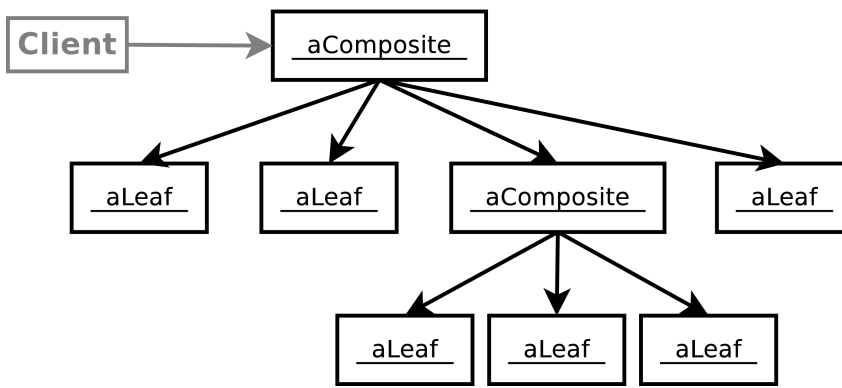
Zależności:

Klient posługuje się obiektami w kompozycji przez interfejs Komponentu, jeśli odbiorcą jest Liść, spełnia on żądanie bezpośrednio, jeśli jest nim Kompozyt, zazwyczaj odsyła je do swych dzieci, niekiedy dodając własne operacje (przed lub po).

```
class Grupa implements ElementGraficzny {
     private ArrayList<ElementGraficzny> children;
     public Grupa() {
          children = new ArrayList<ElementGraficzny>();
     }
     public void add(ElementGraficzny child) {
          children.add(child);
     }
     public void remove(ElementGraficzny child) {
          children.remove(child);
     }
     public ElementGraficzny getChild(int index) {
          return children.get(index);
     }
     public void rysuj(Ekran e) {
          for(ElementGraficzny child : children)
               child.rysuj(e);
     }
}
```

```
// ... Client:
Grupa grupa1 = new Grupa();
grupa1.add(new Linia(...));
grupa1.add(new Linia(...));
grupa1.add(new Prostokąt(...));
Grupa grupa2 = new Grupa();
grupa2.add(new Prostokąt(...));
grupa2.add(new Tekst(...));
grupa2.add(grupa1);
grupa2.add(new Tekst(...));

grupa2.rysuj(new Ekran(...));
```



Konsekwencje:

1. Definiuje hierarchię klas składającą się z obiektów prostych i złożonych. Obiekty proste mogą wchodzić w skład bardziej złożonych, które znów mogą być częścią innych obiektów i tak dalej. Klient w ten sam sposób, co obiektem prostym może posłużyć się i złożonym.
2. Prosty klient – bo jednakowo traktuje obiekty proste i złożone struktury. Nie powinien wiedzieć, czy ma do czynienia z liściem, czy z kompozytem.
3. Łatwo dodawać nowe komponenty (kolejne klasy liści lub kompozytów) – nie trzeba modyfikować kodu klienta.
4. Struktura niekiedy zbyt uniwersalna. Jeśli chcemy, by jakiś kompozyt składał się wyłącznie z pewnych określonych komponentów, musimy dbać o to ręcznie, nie ma wbudowanej kontroli typów.

Implementacja:

1. Jawna referencja do ojca – upraszcza poruszanie się po drzewie *(przyda się też przy Chain of Resposibility)*. Definiowana najpewniej w klasie Komponent. Aby uniknąć błędów, najlepiej ustawiać tę wartość tylko w metodach Add i Remove klasy Kompozyt.

```
abstract class ElementGraficzny {
     Grupa ojciec;
     public void rysuj(Ekran e);
}

class Grupa extends ElementGraficzny {
     public void add(ElementGraficzny child) {
          children.add(child); child.ojciec = this;
     }
     public void remove(ElementGraficzny child) {
          children.remove(child); child.ojciec = null;
     }
     ...
}
```

2. Współdzielone komponenty – pozwalają na oszczędność miejsca. Rozwiązanie podpowiadane przez wzorzec Flyweight.

3. Maksymalizacja interfejsu Komponentu – skoro klient ma nie wiedzieć, czy korzysta z Liścia, czy Kompozytu, wszelkie operacje powinniśmy zadeklarować w interfejsie Komponentu (wraz z domyślną implementacją, przesłanianą w razie potrzeby). Operacje nie mające sensu dla liści mogą być pozostawione puste (np. w implementacji domyślnej Komponentu).

```
abstract class ElementGraficzny {
     public void add(ElementGraficzny child) { }
     public void remove(ElementGraficzny child) { }
     public ElementGraficzny getChild(int index) { /*...*/ }
}
```

4. Operacje zarządzania dziećmi – są implementowane w Kompozycie, ale można je zadeklarować już w Komponencie. Deklaracja w Kompozycje wymusza niekiedy sprawdzanie typu i rzutowanie. Z kolei deklaracja w Komponencie jest wygodniejsza, ale mniej bezpieczna – klient może próbować wywołać te metody z liści (dobrym rozwiązaniem jest rzucać wówczas wyjątek).

```
abstract class ElementGraficzny {
     public void add(ElementGraficzny child) {
          throw new UnsupportedOperationException(); }
}
```

5. Umieszczenie listy Komponentów w Komponencie, a nie Kompozycie wiązałoby się z dodatkowym kosztem miejsca w wypadku Liści.

```
class Grupa implements ElementGraficzny {
     private ArrayList<ElementGraficzny> children;
     ...
}
```

6. Kolejność dzieci – niekiedy jest istotna (np. kolejność wyświetlania figur). Należy wówczas odpowiednio zaprojektować interfejs zarządzania dziećmi, aby pozwalał odzwierciedlić ich kolejność. Z pomocą przyjdzie też wzorzec *Iterator*.
7. Kto (w języku bez *garbage collectora*) odpowiada za niszczenie komponentów? Najpewniej Kompozyt – usuwa dzieci, gdy sam jest usuwany.
8. Cachowanie informacji – jeśli struktura jest często przeglądana, możemy w Kompozycie przechowywać część informacji o jego komponentach lub wyniki operacji. Zmiana dziecka powinna unieważnić cache ojca.

```
abstract class Komponent {
     abstract public int wartość();
     proteced Kompozyt ojciec;
     // ...
}
```

```
class Liść extends Komponent {
     private int wartość;
     public int wartość() { return wartość; }
     public void zmień(int wartość) {
          this.wartość = wartość;
          ojciec.emptyCache();
     }
}
class Kompozyt extends Komponent {
     private ArrayList<Komponent> dzieci;
     private int cache;
     boolean emptyCache = true;
     public void emptyCache() { emptyCache = true;
                                ojciec.emptyCache(); }
     public int wartość() {
          if(emptyCache) {
               cache = 0;
               for(Komponent k : dzieci) cache+= k.wartość();
               emptyCache = false;
          }
          return cache;
     }
}
```

9. Wybór struktury danych na komponenty ściśle zależy od zastosowania. Niekiedy różne klasy Kompozytów mogą mieć własne implementacje.

```
abstract class Wyrażenie {
     abstract public boolean wartość();
     // ...
}

class Zmienna extends Wyrażenie {
     private boolean wartość;
     public boolean wartość() { return wartość; }
     // ...
}

class Suma extends Wyrażenie {
     private Wyrażenie a, b;
     public boolean wartość() {
          return a.wartość() || b.wartość();
     }
     // ...
}
```

```
class Iloczyn extends Wyrażenie {
     private Wyrażenie a, b;
     public boolean wartość() {
          return a.wartość() && b.wartość();
     }
     // ...
}

class Negacja extends Wyrażenie {
     private Wyrażenie a;
     public boolean wartość() {
          return !a.wartość();
     }
     // ...
}

// ... wyrażenie: (true && false) || (!false)
Wyrażenie wyr = new Suma(
          new Iloczyn(new Zmienna(true), new Zmienna(false)),
          new Negacja(new Zmienna(false)));
System.out.println(wyr.wartość());
```

Powiązania:

- Połączenie komponent-rodzic jest wykorzystywane przez Chain of Responsibility.
- Flyweight pomaga współdzielić komponenty.
- Iterator może pomóc iterować przez strukturę.
- Visitor – do lokalizacji operacji, które byłyby rozproszone po elementach struktury.
- Może być tworzony przez Buildera.
- Jest dobrym kandydatem do bycia Prototypem.