**Composite pattern**

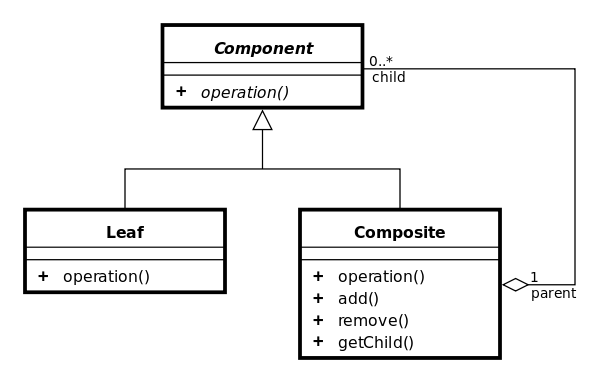
In [software engineering](http://en.wikipedia.org/wiki/Software_engineering), the **composite pattern** is a partitioning [design pattern](http://en.wikipedia.org/wiki/Design_pattern_(computer_science)). The composite pattern describes that a group of objects are to be treated in the same way as a single instance of an object. The intent of a composite is to "compose" objects into tree structures to represent part-whole hierarchies. Implementing the composite pattern lets clients treat individual objects and compositions uniformly.[[1]](http://en.wikipedia.org/wiki/Composite_pattern#cite_note-GangOfFour-1)

When dealing with Tree-structured data, programmers often have to discriminate between a leaf-node and a branch. This makes code more complex, and therefore, error prone. The solution is an interface that allows treating complex and primitive objects uniformly. In [object-oriented programming](http://en.wikipedia.org/wiki/Object-oriented_programming), a composite is an object designed as a composition of one-or-more similar objects, all exhibiting similar functionality. This is known as a "[has-a](http://en.wikipedia.org/wiki/Has-a)" relationship between objects.[[2]](http://en.wikipedia.org/wiki/Composite_pattern#cite_note-2) The key concept is that you can manipulate a single instance of the object just as you would manipulate a group of them. The operations you can perform on all the composite objects often have a [least common denominator](http://en.wikipedia.org/wiki/Least_common_denominator) relationship. For example, if defining a system to portray grouped shapes on a screen, it would be useful to define resizing a group of shapes to have the same effect (in some sense) as resizing a single shape.

**When to use**

Composite can be used when clients should ignore the difference between compositions of objects and individual objects.[[1]](http://en.wikipedia.org/wiki/Composite_pattern#cite_note-GangOfFour-1) If programmers find that they are using multiple objects in the same way, and often have nearly identical code to handle each of them, then composite is a good choice; it is less complex in this situation to treat primitives and composites as homogeneous.

**Structure**

[](http://en.wikipedia.org/wiki/File:Composite_UML_class_diagram_(fixed).svg)

[http://bits.wikimedia.org/static-1.22wmf1/skins/common/images/magnify-clip.png](http://en.wikipedia.org/wiki/File:Composite_UML_class_diagram_(fixed).svg)

Composite pattern in [UML](http://en.wikipedia.org/wiki/Unified_Modeling_Language).

Component

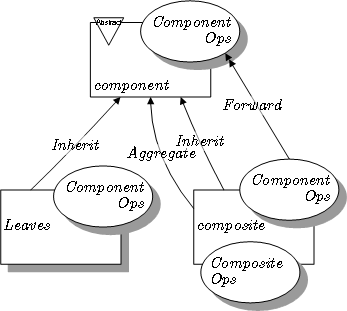
* is the abstraction for all components, including composite ones
* declares the interface for objects in the composition
* (optional) defines an interface for accessing a component's parent in the recursive structure, and implements it if that's appropriate

Leaf

* represents leaf objects in the composition .
* implements all Component methods

Composite

* represents a composite Component (component having children)
* implements methods to manipulate children
* implements all Component methods, generally by delegating them to its children

[](http://en.wikipedia.org/wiki/File:Composite_pattern_in_LePUS3.png)

[http://bits.wikimedia.org/static-1.22wmf1/skins/common/images/magnify-clip.png](http://en.wikipedia.org/wiki/File:Composite_pattern_in_LePUS3.png)

Composite pattern in [LePUS3](http://en.wikipedia.org/wiki/Lepus3).

**Variation**

As it is described in [Design Patterns](http://en.wikipedia.org/wiki/Design_Patterns), the pattern also involves including the child-manipulation methods in the main Component interface, not just the Composite subclass. More recent descriptions sometimes omit these methods.[[3]](http://en.wikipedia.org/wiki/Composite_pattern#cite_note-3)

|  |
| --- |
| **public** **interface** Parca {  **public** **int** fiyat();  } |

|  |
| --- |
| **import** java.util.ArrayList;  **public** **class** Bilgisayar **implements** Parca {  **private** ArrayList<Parca> parcalar = **new** ArrayList<Parca>();  **public** **void** ekle(Parca parca) {  parcalar.add(parca);  }  **public** **void** sil(Parca parca) {  **if** (parcalar.contains(parca))  parcalar.remove(parca);  }  **public** Parca parcaVer(**int** no) {  Parca parca = **null**;  parca = parcalar.get(no);  **if** (parca == **null**)  **throw** **new** ParcaYokHatasi(no + " nolu parca bulunamadi..");  **return** parca;  }  **public** ArrayList<Parca> parcaListesi() {  **return** parcalar;  }  @Override  **public** **int** fiyat() {  **int** fiyat = 0;  **for** (**int** i = 0; i < parcalar.size(); i++) {  fiyat += parcalar.get(i).fiyat();  }  **return** fiyat;  }  } |

|  |
| --- |
| **public** **class** ParcaYokHatasi **extends** RuntimeException {  **public** ParcaYokHatasi(String mesaj) {  **super**(mesaj);  }  } |

|  |
| --- |
| **public** **class** Monitor **implements** Parca{  @Override  **public** **int** fiyat() {  // **TODO** Auto-generated method stub  **return** 250;  }  } |

|  |
| --- |
| **public** **class** Fare **implements** Parca {  @Override  **public** **int** fiyat() {  // **TODO** Auto-generated method stub  **return** 25;  }  } |

|  |
| --- |
| **public** **class** Klavye **implements** Parca {  @Override  **public** **int** fiyat() {  // **TODO** Auto-generated method stub  **return** 50;  }  } |

|  |
| --- |
| **public** **class** Ram **implements** Parca {  @Override  **public** **int** fiyat() {  // **TODO** Auto-generated method stub  **return** 100;  }  } |

|  |
| --- |
| **public** **class** Program {  /\*\*  \* **@param** args  \*/  **public** **static** **void** main(String[] args) {  Bilgisayar bilgisayar = **new** Bilgisayar();  Parca monitor = **new** Monitor();  Parca klavye = **new** Klavye();  Parca fare=**new** Fare();  bilgisayar.ekle(monitor);  bilgisayar.ekle(fare);  bilgisayar.ekle(klavye);  System.*out*.println("monitor fiyati:"+monitor.fiyat());  System.*out*.println("bilgisayar fiyati:"+bilgisayar.fiyat());  }  } |

**Example**

The following example, written in [Java](http://en.wikipedia.org/wiki/Java_(programming_language)), implements a graphic class, which can be either an ellipse or a composition of several graphics. Every graphic can be printed. In Backus-Naur form,

Graphic = ellipse | GraphicList

GraphicList = empty | Graphic GraphicList

It could be extended to implement several other shapes (rectangle, etc.) and methods ([translate](http://en.wikipedia.org/wiki/Translation_(geometry)), etc.).

/\*\* "Component" \*/

interface Graphic {

//Prints the graphic.

public void print();

}

/\*\* "Composite" \*/

import java.util.List;

import java.util.ArrayList;

class CompositeGraphic implements Graphic {

//Collection of child graphics.

private List<Graphic> childGraphics = new ArrayList<Graphic>();

//Prints the graphic.

public void print() {

for (Graphic graphic : childGraphics) {

graphic.print();

}

}

//Adds the graphic to the composition.

public void add(Graphic graphic) {

childGraphics.add(graphic);

}

//Removes the graphic from the composition.

public void remove(Graphic graphic) {

childGraphics.remove(graphic);

}

}

/\*\* "Leaf" \*/

class Ellipse implements Graphic {

//Prints the graphic.

public void print() {

System.out.println("Ellipse");

}

}

/\*\* Client \*/

public class Program {

public static void main(String[] args) {

//Initialize four ellipses

Ellipse ellipse1 = new Ellipse();

Ellipse ellipse2 = new Ellipse();

Ellipse ellipse3 = new Ellipse();

Ellipse ellipse4 = new Ellipse();

//Initialize three composite graphics

CompositeGraphic graphic = new CompositeGraphic();

CompositeGraphic graphic1 = new CompositeGraphic();

CompositeGraphic graphic2 = new CompositeGraphic();

//Composes the graphics

graphic1.add(ellipse1);

graphic1.add(ellipse2);

graphic1.add(ellipse3);

graphic2.add(ellipse4);

graphic.add(graphic1);

graphic.add(graphic2);

//Prints the complete graphic (four times the string "Ellipse").

graphic.print();

}

}