1 合成模式 (Composite)

合成 (Composite) 模型模式属于对象的结构模式，有时又叫做部分-整体 (Part-Whole) 模式。合成模式将对象组织到树结构中，可以用来描述整体与部分的关系。合成模式可以使客户端将整体对象与部分对象同等看待。

在下面的情况下应当考虑使用合成模式：

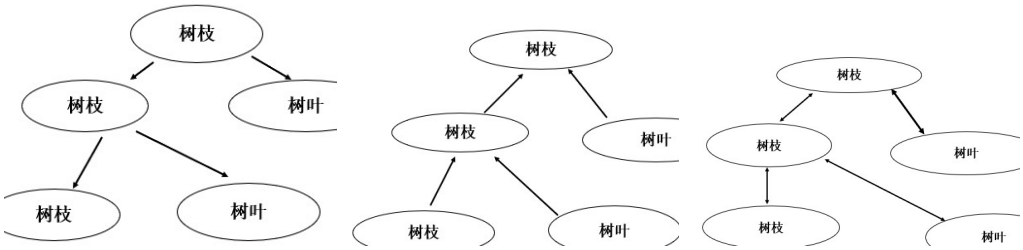
- 需要描述对象的部分和整体的等级结构;
- 需要客户端忽略掉整体对象和部分对象的区别。

合成模式的典型应用-文件系统: 一个文件系统就是一个典型的合成模式系统。文件系统是一个树结构。树有节点，节点有两种，一种是树枝节点，即目录，有子树结构；另一种是文件，即树叶节点，没有子树结构。

1.1 对象的树结构

有向树结构的分类: 根据信息传递的方向，有向树结构又可以分为 3 种, 从上向下，从下向上和双向的。在三种有向树图中，树的节点和他们的相互关系都是一样的，但是连接他们的关系的方向却不同。

由上向下的树图: 在由上向下的树图中，每一个树枝节点都有箭头指向它的所有的子节点，从而一个客户端可以要求一个树枝节点给出所有的子节点，而一个节点却不知道它父节点。在这样的树结构上，信息可以按照箭头所指的方向自上向下传播。



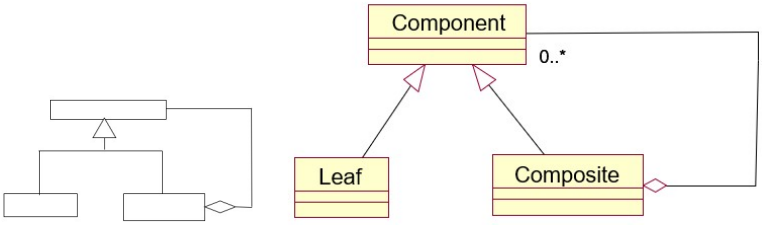
由下向上的树图: 在一个由下向上的树图中，每一个节点都有箭头指向它的父节点，但是父节点却不知道其子节点。信息可以按照箭头所指的方向自下向上传播。

双向的树图: 在一个双向的树图中，每一个节点都同时知道它的父节点和所有的子节点。在这样的树结构上，信息可以按照箭头所指的方向向两个方向传播。

树图中的两种节点: 一个树结构由 2 种节点组成：树枝节点和树叶节点。树枝节点可以有子节点，树叶节点不可以有子节点。注意一个树枝子节点可以不带任何叶子，但是它因为有带有叶子的能力，因此仍然是树枝节点而不会成为叶子节点。一个树叶节点则永远不可能带有子节点。

根节点: 一个树结构中总有至少一个节点是特殊的节点，称为根节点。一个根节点没有父节点，因为它是树结构的根。一个树的根节点一般是树枝节点，如果根节点是树叶节点的话，这个树就变成了只有这一个节点的树。

树结构的类图: 可以使用类图描述一个树结构的静态结构。下图所示的是合成模式的简略类图，同时也是一个典型的树结构的类图。



可看出上面的类图结构涉及到三个角色:

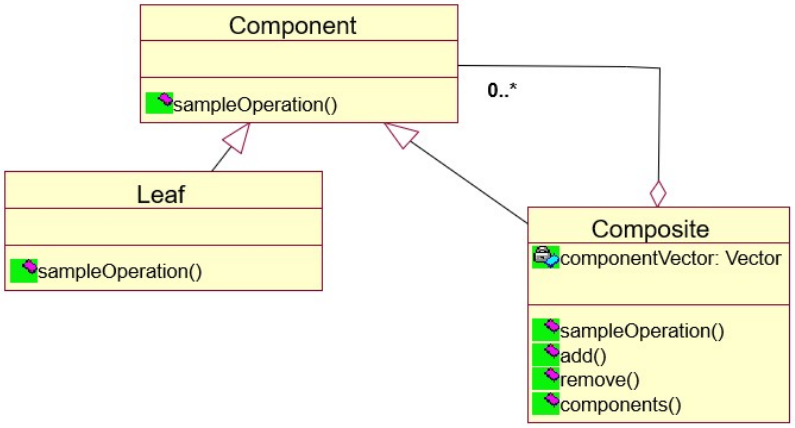
- 抽象构件 (Component) 角色：这是一个抽象角色，它给参加组合的对象规定一个接口。这个角色给出共有的接口及其默认行为。
- 树叶构件 (Leaf) 角色：代表参加组合的树叶对象。一个树叶没有下级的子对象。定义出参加组合的原始对象的行为。
- 树枝构件 (Composite) 角色：代表参加组合的所有包含子对象的对象，并给出树枝构件对象的行为。

1.2 合成模式的两种形式

透明方式: 作为第一种选择，在 Component 里面声明所有的用来管理子类对象的方法，包括 add()、remove() 以及 getChild () 方法。好处是所有的构件类都有相同的接口。缺点是不够安全，因为树叶类对象和合成类对象在本质上是有区别的。树叶类对象不可能有下一个层次的对象，因此 add()、remove() 以及 getCHild () 是没有意义的，但编译时期不会出错，只在运行时期才可能会出错。

安全方式: 第二种选择是在 Composite 类里面声明所有的用来管理子类对象的方法。这样的做法是安全的做法。这个选择的缺点是不够透明，因为树叶类和合成类将具有不同的接口。

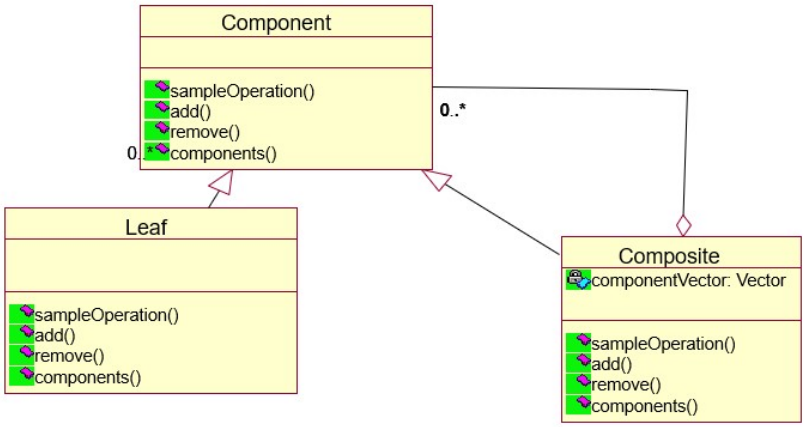
安全式的合成模式的结构: 安全式的合成模式要求管理聚集的方法只出现在树枝构件类中，而不出现在树叶构件类中。其类图如下:




```
1 import java.util.Vector;
2 import java.util.Enumeration;
3 public interface Component {
4   Composite getComposite(); // 返返自己的实例
```

```
5   void sampleOperation(); // 某个业务逻辑方法
6 }
7 public class Composite implements Component {
8   private Vector componentVector=new java.util.Vector();
9   public Composite getComposite() {
10    return this; /* 返还自己的实例 */
11  }
12  public void sampleOperation { /* 某业务逻辑方法 */
13   Enumeration enumeration=components();
14    while (enumeration.hasMoreElements()) {
15      ((Component)enumeration.nextElement()).sampleOperation();
16    }
17  }
18  /* 聚集管理方法，增加一个子构件 */
19  public void add(Component component) {
20    componentVector.addElement(component);
21  }
22  /* 聚集管理方法，删除一个子构件 */
23  public void remove(Component component) {
24    componentVector.removeElement(component);
25  }
26 }
27 public class Leaf implements Component {
28  /* 某业务逻辑方法 */
29  public void sampleOperation { }
30  public Composite getComposite() { /* 返还自己的实例 */
31    // write your code here
32    return null;
33  }
34 }
```

透明式的合成模式的结构：透明式的合成模式要求所有的具体构件类，不论是树枝还是树叶，均符合一个统一的接口。其示意类图如下：



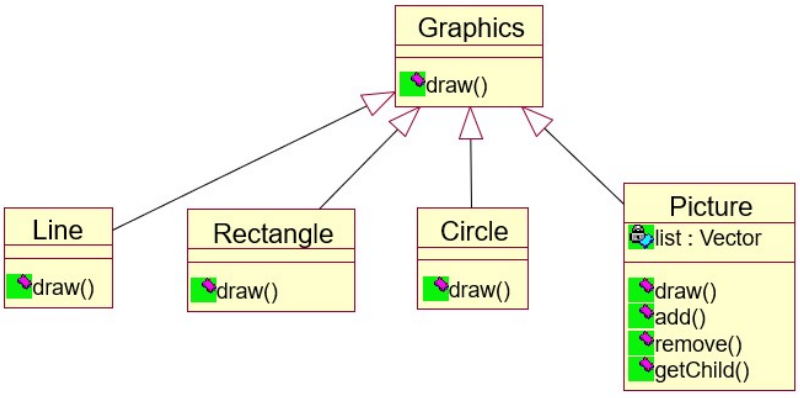

```
1  import java.util.Vector;
2  import java.util.Enumeration;
3  public interface Component {
4    Composite getComposite(); // 返返自己的实例
5    void sampleoperation(); // 某个业务逻辑方法
6    /* 聚集管理方法，增加一个子构件 */
7    void add(Component component);
8    /* 聚集管理方法，删除一个子构件 */
9    void remove(Component component);
10   /* 聚集管理方法，返还聚集的Enumeration对象 */
11   Enumeration components( );
12 }
13 public class Composite implements Component {
14   private Vector componentVector=new java.util.Vector();
15   public Composite getComposite() { /* 返还自己的实例 */
16     return this;
17   }
18   public void sampleOperation { /* 某业务逻辑方法 */
19    Enumeration enumeration=components();
20     while (enumeration.hasMoreElements()) {
21       ((Component)enumeration.nextElement()).sampleOperation();
22     }
23   }
24   /* 聚集管理方法，增加一个子构件 */
25   public void add(Component component) {
26     componentVector.addElement(component);
27   }
28   /* 聚集管理方法，删除一个子构件 */
29   public void remove(Component component) {
30     componentVector.removeElement(component);
31   }
```

```
32    /* 聚集管理方法，返回聚集的Enumeration对象
33    public   Enumeration components() {
34      return componentVector.elements();
35    }
36  }
37  public class Leaf implements Component {
38    public void sampleOperation { /* 某个业务逻辑方法 */
39      //write your code here
40    }
41    public Composite getComposite() { /* 返还自己的实例 */
42      return null;
43    }
44    /* 聚集管理方法，增加一个子构件 */
45    public void add(Component component) {}
46    /* 聚集管理方法，删除一个子构件 */
47    public void remove(Component component){}
48    /* 聚集管理方法，返回聚集的Enumeration对象 */
49    public Enumeration components() { return null; }
50  }
```

一个绘图的例子: 以一个绘图软件说明合成模式的应用。一个绘图系统给出各种工具用来描绘由线、长方形和圆形等基本图形组成的图形。一个复合的图形是由这些基本图形组成，可以运用合成模式。合成图形应当有一个列表，存储对所有基本图形的引用。复合图形的 draw() 方法在调用时，应当逐一调用所有列表上的基本图形的 draw() 方法。可以使用两种合成模式来实现：安全式和透明式。

1.3 应用安全式合成模式

安全式的合成模式意味着只有树枝构件才配有管理聚集的方法，而树叶则没有这些方法。下图所示是使用安全式的设计结构图



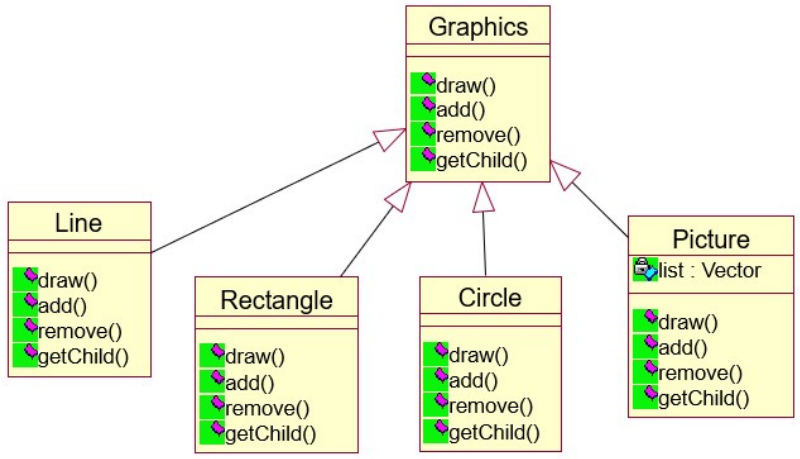
客户端可以调用 add() 方法加入一个基本图形，remove() 方法取消一个基本图形，getChild(index x) 得到组成复杂图形的第 x 个基本图形。

```
// 构件角色Graphics
package com.javapatterns.composite.drawingsafe;
public abstract class Graphics {
  public abstract void draw();
}
// Picture类是树枝构件角色，它实现了抽象构件角色所要求的方法，
// 还额外提供了用于管理子对象聚集的一系列方法
package com.javapatterns.composite.drawingsafe;
import java.util.Vector;
public class Picture extends Graphics {
  private Vector list=new Vector(10);
  public void draw() {
    for (int i=0; i< list.size(); i++) {
      Graphics g = (Graphics)list.get(i);
      g.draw();
    }
  }
  public void add(Graphics g) { /* 聚集管理方法,增加一个子构件 */
    list.add(g);
  }
  public void remove(Graphics g) { /* 聚集管理方法,删除一个子构件 */
    list.remove(g);
  }
  public void getChild(int i) {
    return (Graphics)list.get(i); /* 返还一个子构件 */
  }
}
//Line是树叶构件，它没有任何的子对象，
//因此不必提供管理子对象聚集的方法
package com.javapatterns.composite.drawingsafe;
public class Line extends Graphics {
  public void draw() {
    //write your code
  }
}
//树叶Rectangle
package com.javapatterns.composite.drawingsafe;
public class Rectangle extends Graphics {
  public void draw() {
    //write your code
  }
}
```

```
43 //树叶Circle
44 package com.javapatterns.composite.drawingsafe;
45 public class Circle extends Graphics {
46   public void draw() {
47     //write your code
48   }
49 }
```

1.4 应用透明式的合成模式

透明式合成模式无论树枝和树叶都有管理聚集的方法。其设计图如下:




```
1  // 构件角色Graphics
2  package com.javapatterns.composite.drawingtransparent;
3  abstract public class Graphics {
4     public abstract void draw();
5     public void add(Graphics g); /* 聚集管理方法,增加一个子构件 */
6     public void remove(Graphics g); /* 聚集管理方法,删除一个子构件 */
7     public Graphics getChild(int i); /* 返还一个子构件 */
8  }
9  // Picture类是树枝构件角色，它实现了抽象构件角色所要求的方法，
10 // 还额外提供了用于管理子对象聚集的一系列方法
11 package com.javapatterns.composite.drawingtransparent;
12 import java.util.Vector;
13 public class Picture extends Graphics {
14   private Vector list=new Vector(10);
15   public void draw() {
16     for (int i=0;i< list.size();i++) {
17       Graphics g=(Graphics)list.get(i);
18       g.draw();
```

```
    }
  }
  /* 聚集管理方法,增加一个子构件 */
  public void add(Graphics g) { list.add(g); }
  /* 聚集管理方法,删除一个子构件 */
   public void remove(Graphics g) { list.remove(g); }
  /* 返还一个子构件 */
   public Graphics getChild(int i) { return (Graphics)list.get(i); }
}
// Line是树叶构件，它必需实现抽象构件角色所要求的方法
package com.javapatterns.composite.drawingtransparent;
public class Line extends Graphics {
  public void draw() {
    //write your code
  }
  /* 聚集管理方法,增加一个子构件 */
  public void add(Graphics g) {
    //do nothing
  }
  /* 聚集管理方法,删除一个子构件 */
  public void remove(Graphics g) {
    //do nothing
  }
  /* 返还一个子构件 */
  public Graphics getChild(int i) { return null; }
}
// Rectangle是树叶构件，它必需实现抽象构件角色所要求的方法
package com.javapatterns.composite.drawingtransparent;
public class Rectangle extends Graphics {
  public void draw() {
    //write your code
  }
  public void add(Graphics g) { // 聚集管理方法,增加一个子构件
    //do nothing
  }
  /* 聚集管理方法,删除一个子构件 */
  public void remove(Graphics g) {
    //do nothing
  }
  /* 返还一个子构件 */
  public void getChild(int i) { return null; }
}
```

```
61 // Circle是树叶构件，它必需实现抽象构件角色所要求的方法
62 package com.javapatterns.composite.drawingtransparent;
63 public class Circle extends Graphics {
64   public void draw() {
65     //write your code
66   }
67   public void add(Graphics g) { // 聚集管理方法,增加一子构件
68     //do nothing
69   }
70   public void remove(Graphics g) { // 聚集管理方法,删除一个子构件
71     //do nothing
72   }
73   public void getChild(int i) { // 返还一个子构件
74     return null;
75   }
76 }
```