第5章

合成（Composite）模式

合成模式的英文为 Composite，是一组对象的组合，这些对象可以是容器对象，表现为组的概念；另外一些对象则代表了单对象，或称为叶子对象。在对组合进行建模时，必须注意两个重要的概念。第一个概念是组对象允许包含单对象，也可以再包含其他的组对象（常见的错误是将组对象设计为只允许包含叶子对象）。第二个概念则是要为组合对象和单对象定义共同的行为。结合这两个概念，就可以为组对象与单对象定义统一的类型，并将该组对象建模为包含同等类型对象的集合。

合成模式的意图是为了保证客户端调用单对象与组合对象的一致性。

常规组合

图 5.1 展示了一个经典的组合结构。Leaf 类和 Composite 类都实现自一个抽象的 Component 通用接口，同时，Composite 对象又包含了其他的 Composite 和 Leaf 对象的集合。

注意，图 5.1 中的 Component 类是一个抽象类，它未包含任何一个具体方法，因而可以将其定义为接口，让 Leaf 类和 Composite 类去实现它。

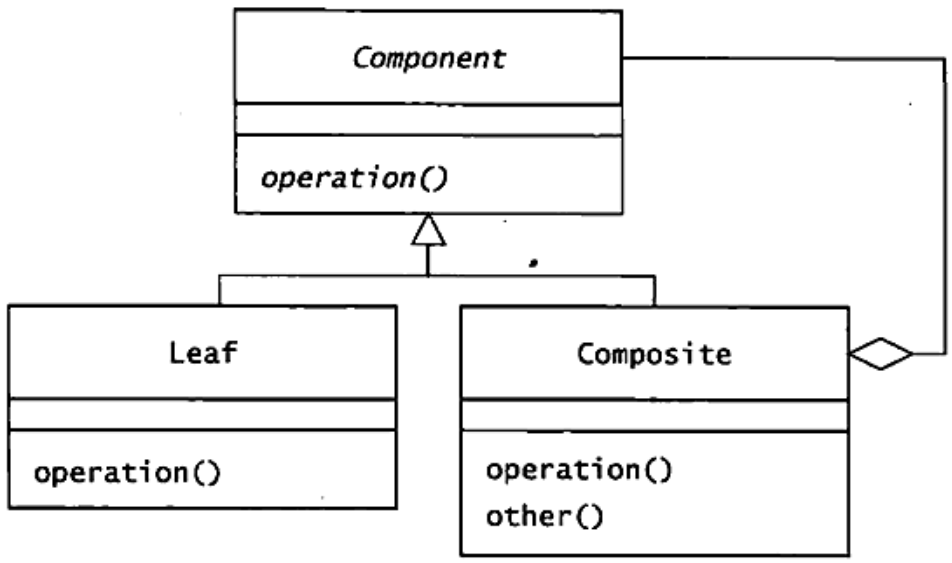

图 5.1 合成模式的关键在于组合对象可以包含其他组合对象（不仅仅是叶子对象），且 Composite 和 Leaf 节点共享了一个共同的接口

挑战 5.1

在图 5.1 中，为何 Composite 类维持了包含 Component 对象的集合，而不是仅包含叶子对象？

答案参见第 305 页

合成模式中的递归行为

在对生产焰火的机器进行处理时，Oozinoz 的工程师们观察到组合的特征。工厂是由车间组成的，每个车间有一条或多条生产线。一条生产线上有很多机器，机器之间相互协作，以保证生产进度。Oozinoz 的开发人员设计了图 5.2 所示的类图，将这些工厂、车间、生产线看做是“机器”的组合来进行建模。

如图 5.2 所示，单台机器与机器的组合都定义了 getMachineCount()方法，它会返回给定组件中机器的数量。

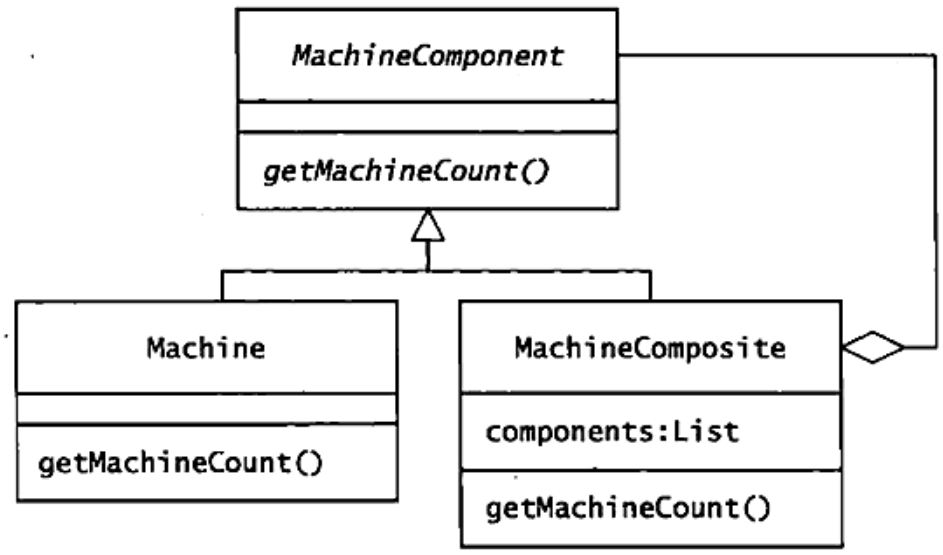

图 5.2　对于单台机器与组合机器而言，getMachineCount()方法都是它们需要的

挑战 5.2

写出 Machine 与 MachineComposite 中 getMachineCount()方法的实现代码。

答案参见第 305 页

假设我们考虑给 MachineComponent 增加如下方法：

方　法	行　为
isCompletelyUp()	判断组件中的所有机器是否都处于“up”状态
stopAll()	停止组件中所有机器的运作
getOwners()	返回负责管理机器的一组工程师
getMaterial()	返回机器组件中所有正在处理的原料

在 MachineComponent 中，每个方法的定义和操作都是递归的。例如，某个组合中机器的数量是该组合中所有组件包含的机器数量总和。

挑战 5.3

为 MachineComponent 声明的每个方法，写出表 5.1 中 MachineComposite 中的递归定义与 Machine 中的非递归定义。

表 5.1 为方法定义

方 法	类	定 义
getMachineCount()	MachineComposite	返回组合中每个组件的数量和
	Machine	返回 1
isCompletelyUp()	MachineComposite	? ?
	Machine	? ?
stopAll()	MachineComposite	? ?
	Machine	? ?
getOwners()	MachineComposite	? ?
	Machine	? ?
getMaterial()	MachineComposite	? ?
	Machine	? ?

答案参见第 306 页

组合、树与环

在合成结构中，如果一个节点拥有对其他节点的引用，则该节点就是一棵树。然而，这个定义太宽泛了。我们可以引用图论中的一些理论为对象建模，以示其精确性。倘若将对象看做节点，将对象间的引用看做边，就可以将对象模型绘制为图形结构。

考虑一个鉴定或分析化学药品的对象模型。Assay 类拥有一个 Batch 类型的 batch 属性，Batch 类则拥有一个 Chemical 类型的 chemical 属性。假设存在一个特殊的 Assay 对象 a，它的 batch 属性引用 Batch 对象 b，b 的 chemical 属性引用 Chemical 对象 c。图 5.3 展示了该对象模型的两种可供选择的对象图。（若要了解更多关于如何使用 UML 描述对象模型的信息，请参考附录 D。）

a 引用 b，b 引用 c，这一系列的引用导致出现了一条从 a 到 c 的路径。环（cycle）指的是路径中包含了出现两次的节点。倘若 Chemical 对象 c 反向引用 Assay 对象 a 的话，将会在对象模型中出现一个环。

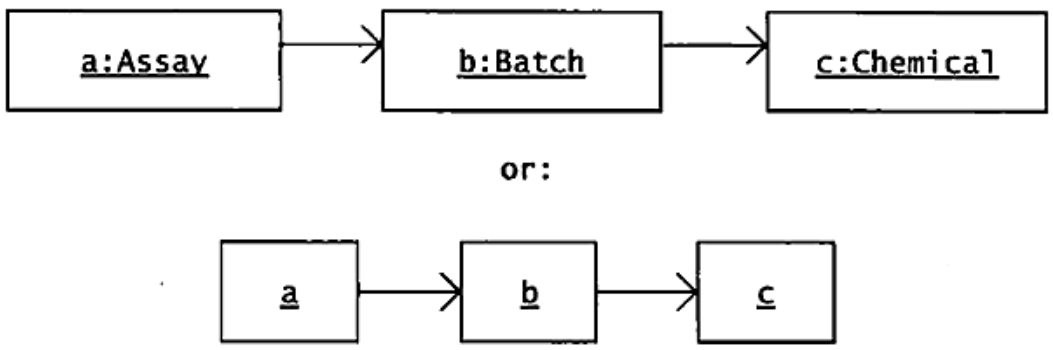

图 5.3　两幅 UML 图展示了描述相同信息的两种不同体现：对象 a 引用对象 b，对象 b 引用对象 c

对象模型是有向图，每个对象引用都会有一个方向。图论中的术语“树”通常指的是无向图，然而，只要满足下列条件，一个有向图也可以被称为树：

- 存在一个没有任何引用指向自身的根节点。
- 其他每个节点都只有一个引用该节点的父节点。

为何要考虑树中图的概念呢？因为合成模式特别适合这种结构形式。（如你所见，可以使用有向无环图甚至有环图实现合成模式，但是这需要额外的工作，必须加倍小心。）

图 5.3 中的对象模型描述的是一棵简单树。对于更大的对象模型，很难去鉴别模型是否为树。图 5.4 展示了 plant 的工厂对象模型，plant 是 MachineComposite 类的对象。这个 plant 包含一个含有三台机器的车间：mixer、press 和 assembler。plant 对象的机器列表组件包含对 mixer 的直接引用。

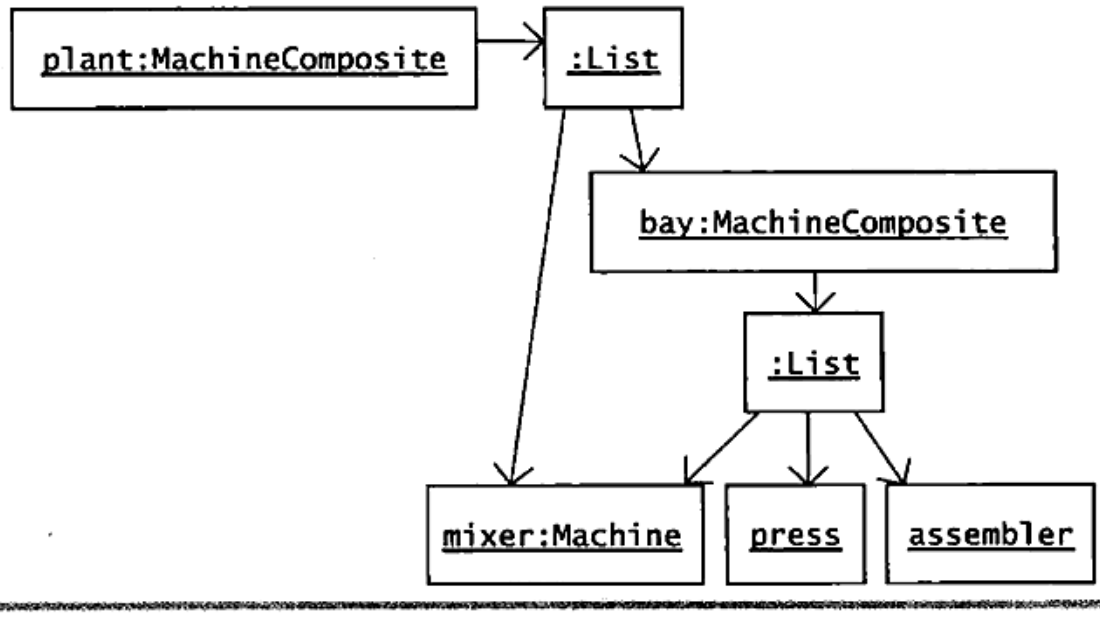

图 5.4　既不是环也不是树的对象模型图

图 5.4 中的对象图不包括环，但由于有两个对象同时引用了 mixer 对象，因此它也不是树。如果删除或者忽视 plant 对象和它相应的 list，bay 对象就成为树的根了。

假设合成模式是树形结构，而系统又允许使用不为树形结构的 Composite，则合成模式中的方法就存在问题。挑战 5.2 要求定义一个 getMachineCount()操作，Machine 类对此方法的实现无疑是正确的：

```
public int getMachineCount() {
    return 1;
}
```

MachineComposite 类同样正确地实现了 getMachineCount()方法，返回组合对象中每个组件的数量总和：

```
public int getMachineCount() {
    int count = 0;
    Iterator i = components.iterator();
    while (i.hasNext()) {
        MachineComponent mc = (MachineComponent) i.next();
        count += mc.getMachineCount();
    }
    return count;
}
```

只要 MachineComponent 对象是树形结构，这些方法都是正确的。然而，这种假设并非总是成立，特别是当用户可以编辑这一组合模型时，可能会突然变成非树形结构。考虑如下的 Oozinoz 实例。

Oozinoz 的焰火工程师使用图形界面应用程序来记录和更新工厂中的机器组合对象模型。某天，他们报告称工厂中现有机器的数量有误。我们可以在 OozinozFactory 类中定义 plant()方法，重新创建对象模型。

```
public static MachineComposite plant() {
    MachineComposite plant = new MachineComposite(100);
    MachineComposite bay = new MachineComposite(101);
    Machine mixer = new Mixer(102);
    Machine press = new StarPress(103);
    Machine assembler = new ShellAssembler(104);
    bay.add(mixer);
```

```
    bay.add(press);
    bay.add(assembler);
    plant.add(mixer);
    plant.add(bay);
    return plant;
}
```

这正是图 5.4 中创建 plant 对象的代码实现。

挑战 5.4

下面的程序将会输出什么？

```
package app.composite;
import com.oozinoz.machine.*;

public class ShowPlant {
   public static void main(String[] args) {
        MachineComponent c = OozinozFactory.plant();
        System.out.println(
            "Number of machines: " + c.getMachineCount());
   }
}
```

答案参见第 306 页

Oozinoz 的工程师们通常会使用 GUI 应用程序为工厂中的机器进行对象建模，它可以在第二次添加节点时，检查该节点是否已经在组件树中存在。一个简单的实现是维护一个已存在节点的集合。然而，你可能无法控制一个组合对象的构成。因此需要编写一个 isTree()方法去检查组合对象是否为树形结构。

倘若在遍历对象模型之后，没有哪个节点被访问两次，我们就认为这个对象模型是一棵树。可以在抽象类 MachineComponent 中实现 isTree()方法，以便它可以维护访问过的节点列表。MachineComponent 类可以将带参数的 isTree(set:Set)方法作为抽象方法。图 5.5 展示了 isTree()方法的实现方式。

在 MachineComponent 类的实现中，isTree()方法调用了自身的抽象方法

isTree(s:Set)，代码如下：

```
public bool eanisTree() {
    return isTree(new HashSet());
}
protected abstract boolean isTree(Set s);
```

该方法使用了 Java 类库中的 Set 类。

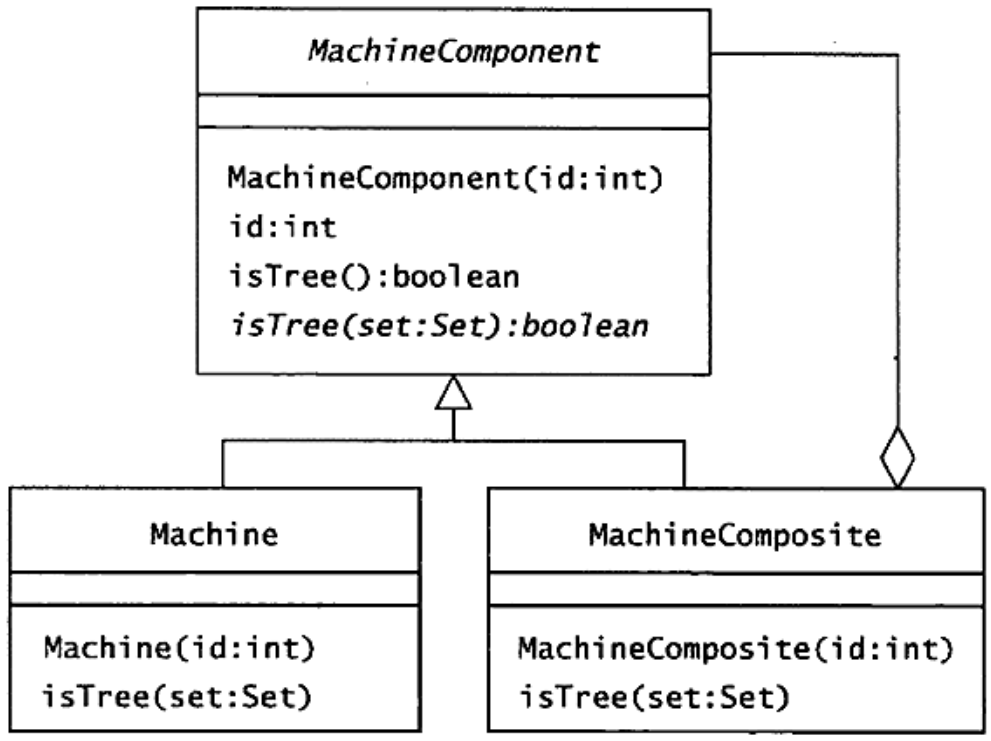

图 5.5 isTree()方法可以检测该组合模型是否为树形结构

Machine 和 MachineComposite 类都必须实现 isTree(s:Set)方法，Machine 类的 isTree()实现非常简单，反映了单个机器始终是树形结构：

```
protected boolean isTree(Set visited) {
    visited.add(this);
    return true;
}
```

MachineComposite 实现的 isTree()必须把调用它的对象加到 visited 列表中，并且迭代调用组合对象中各个组件的 isTree()方法。如果任何一个被访问过的组件返回 false，或者任何一个组件本身并不是树形结构，该方法就会返回 false，否则返回 true。

挑战 5.5

写出 `MachineComposite.isTree(Set visited)`的代码。

答案参见第 307 页

只需多加注意，防止 `isTree()`返回 `false`，就可以保证对象模型是一个树状模型。另一方面，也可能需要允许组合对象不是树形结构，尤其是当正在建模的问题领域包含环的时候。

含有环的合成模式

挑战 5.4 中提到的非树形合成模式是一个失误，原因在于用户将 machine 同时当做 plant 和 bay 的一部分。若是物理对象，可能并不允许它被包含在多个对象中。然而，问题域可以包括多个非物理元素，此时环状包含关系是有意义的。这种场景在对工作流建模时经常发生。

考虑图 5.6 描述的礼花弹的构造过程。我们点燃位于底部提供动力的黑火药，形成一股推动力，把焰火弹发射出去。当焰火弹升空后，它的二级引线点燃，最终燃至内核，然后内核燃烧并且爆炸，形成焰火的效果。

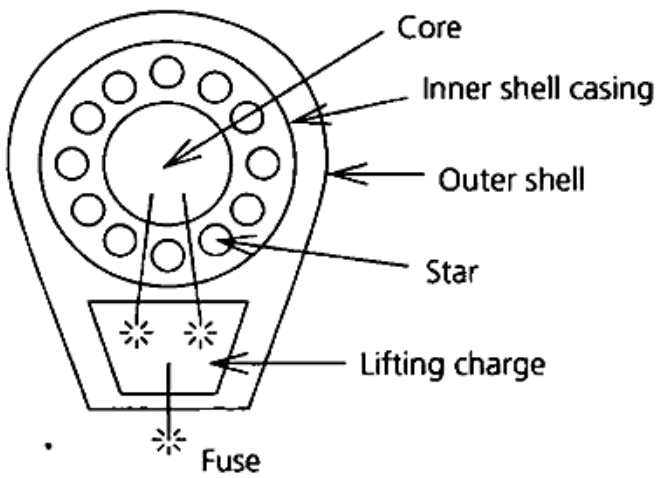

图 5.6　一个焰火弹使用两份火药：一份用于开始升空时提供动力，另一份用于在焰火弹到达顶点时爆炸并产生焰火效果

制作焰火弹的工艺流程是：制作内层的焰火弹，进行检测，然后确定它是否需要返工或者完成组装。

内层焰火弹的制作流程为：操作员使用焰火弹装配器把火药放入一个半球状的盒子里，插入黑火药核，然后在核的顶部放入更多的火药，再把它密封在其他半球状的盒子中。

检察员验证内层焰火弹是否安全，质量是否符合标准。如果不是，操作员就需要拆解内层焰火弹，重新制作。如果内层焰火弹通过检查，操作员将使用引线将内层焰火弹与提供上升动力的部分联系起来。最后，操作员手工给焰火弹包上外壳完成组装。

和之前的机器实现合成模式一样，Oozinoz 工程师用一个 GUI 程序来描述组合的过程。图 5.7 给出了支持建模过程的类图结构。

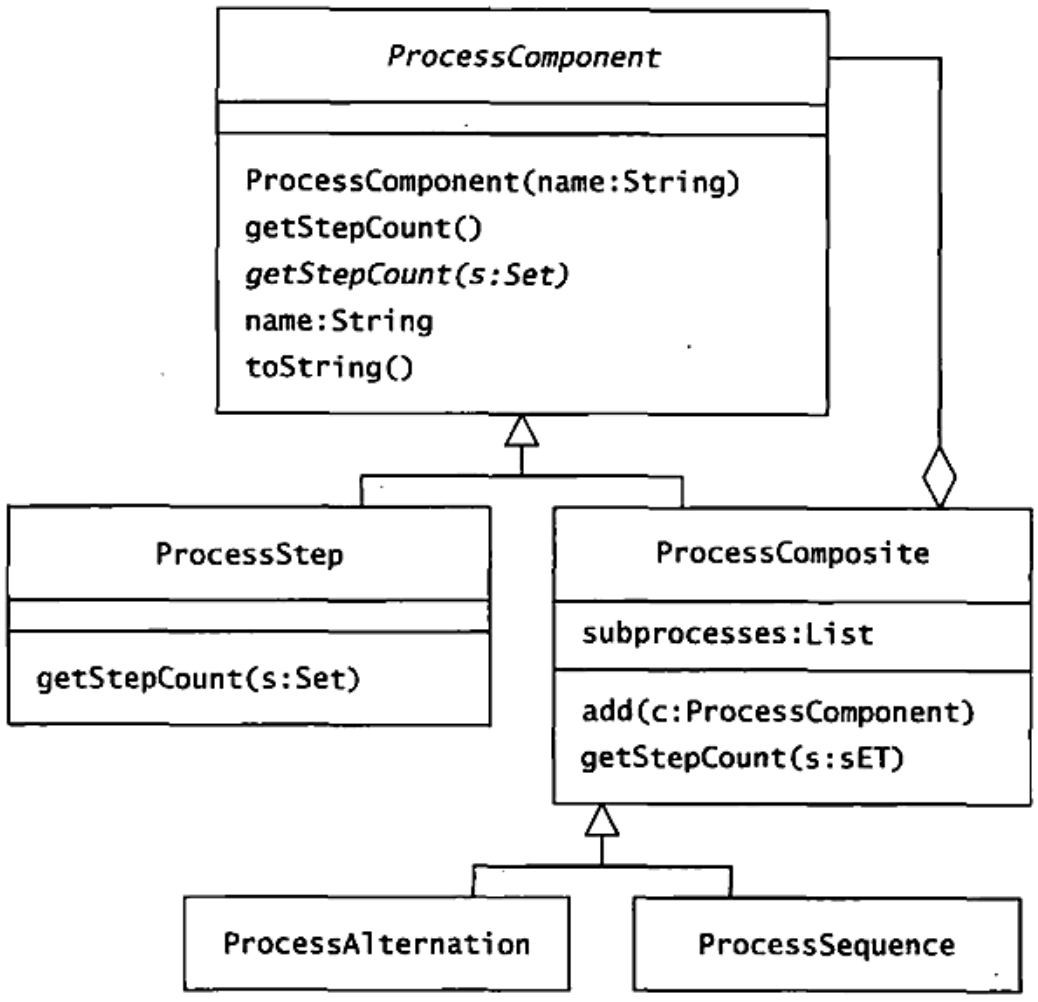

图 5.7 制作焰火弹的流程，包括一些可相互替代的或者有顺序的步骤

图 5.8 展示了代表制作焰火弹工艺流程的对象。make 流程按顺序包含 buildInnerShell 步骤、inspect 步骤以及 reworkOrFinish 子步骤。reworkOrFinish 子步骤含有两个可相互替换的路径。make 流程的最后可能是 disassemble 步骤或者仅仅是 finish 步骤。

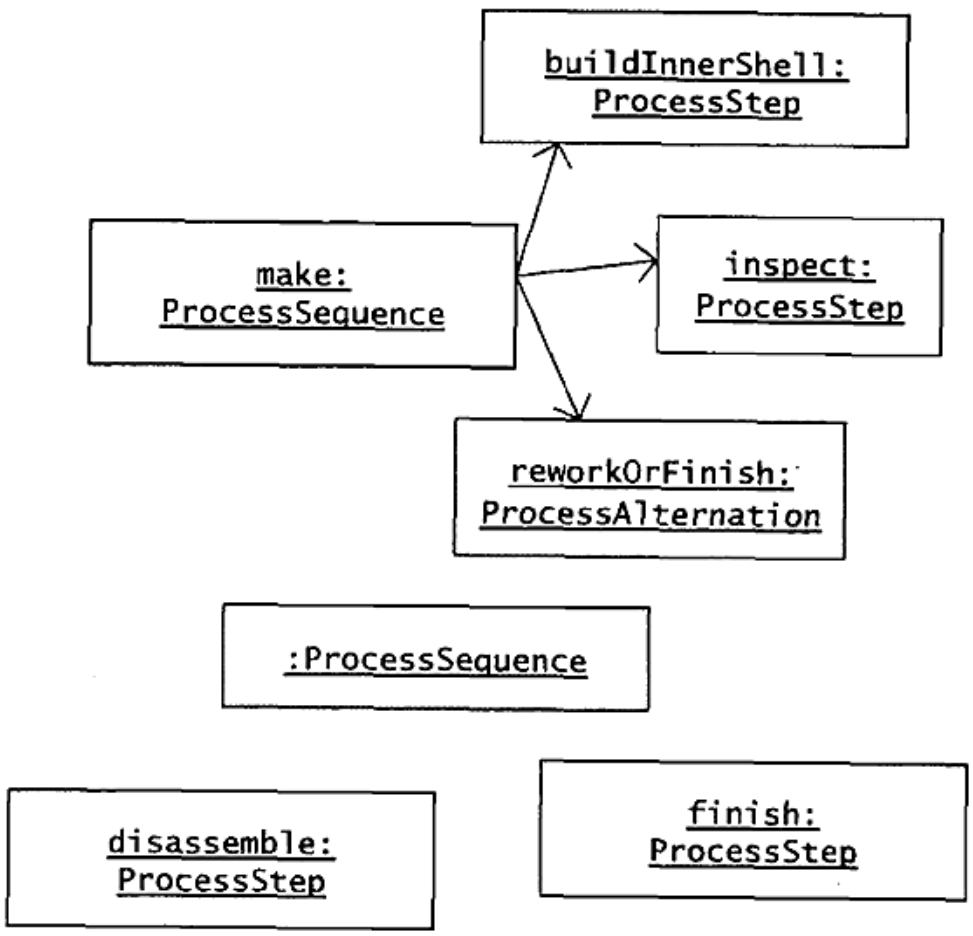

图 5.8　当你完成这幅图后，将会展示出 Oozinoz 在制作焰火弹流程时的对象模型

挑战 5.6

图 5.8 展示了焰火弹组装流程模型中的对象。一幅完整的对象图应该包含各个对象间的相互引用关系。例如，该图已经包含了 make 对象包含的引用关系。请补充图中缺失的引用关系。

答案参见第 307 页

位于 ProcessComponent 层级的 getStepCount()操作负责统计工艺流程中单个处理步骤的数量。注意，这个数量并非流程的深度，而是图中叶子节点处理步骤的数量。getStepCount()方法必须确保每个步骤的统计只有一次，如果流程中包含了环，应避免出现死循环。ProcessComponent 类实现了 getStepCount()方法，以便通过该方法获得访问过的节点集合。

```
public int getStepCount() {
    return getStepCount(new HashSet());
}
```

```
public abstract int getStepCount(Set visited);
```

ProcessComposite 类在实现 getStepCount()方法时应避免再次访问之前已经访问过的节点：

```
public int getStepCount(Set visited) {
    visited.add(getName());
    int count = 0;
    for (inti = 0; i<subprocesses.size(); i++) {
        ProcessComponent pc =
            (ProcessComponent) subprocesses.get(i);
        if (!visited.contains(pc.getName()))
            count += pc.getStepCount(visited);
    }
    return count;
}
```

ProcessStep 类中的 getStepCount()方法的实现很简单：

```
public int getStepCount(Set visited) {
    visited.add(name);
    return 1;
}
```

com.oozinoz.process 包提供了一个 ShellProcess 类。该类定义了图 5.8 描述的返回 make 对象的 make()方法。com.oozinoz.testing 包有一个 ProcessTest 类，该类提供各类流程图的自动化测试。例如，ProcessTest 类包含一个方法，该方法测试 getStepCount()操作能否正确返回具有环状结构的 make 流程中的步骤数。

```
public void testShell() {
    assertEquals(4, ShellProcess.make().getStepCount());
}
```

这些测试在 JUnit 测试框架下运行并通过。若要了解更多关于 JUnit 的信息，请访问 www.junit.org。

环的影响

许多组合对象上的操作，例如统计叶子节点的数量，在组合对象不是树形结构时仍有意义。

通常，非树形与树形结构组合对象的唯一区别在于，你必须小心谨慎地避免重复操作相同的节点。然而，如果组合对象包含了环，则某些操作是没有意义的。例如，我们不能通过算法决定在 Oozinoz 中制作焰火弹所需要的最大步骤数。在任何一个包含环的组合对象中，依赖路径长度的操作都是无意义的。因此，尽管我们可以讨论树的高度——从根到叶子的最长距离，但在一个有环图中却不存在。

倘若允许合成模式不是树形结构，带来的另一个后果是不能假设每个节点有且仅有一个父节点，而是可能具有多个父节点。例如，图 5.8 的流程模型可能有多个组合对象引用 inspect 步骤，则 inspect 对象就将存在多个父节点。一个节点拥有多个父节点本身没有任何问题，但在建模和编码时，必须考虑这个问题。

小结

合成模式包含两个相互关联的重要概念。一个概念是一个组合对象可以包含单对象或者其他组合对象。与之相关的另一个概念就是组合对象与单对象共享同一个接口。将这些概念应用于对象建模上，就可以在组合对象与单对象上创建抽象类或 Java 接口，以此定义公共行为。

在对组合对象建模时，通常需要给组合节点引入递归定义。倘若存在递归定义，编写代码时，需要注意防止死循环。为避免这一问题，可以确保组合对象都是树形结构。此外，虽然允许环出现在合成模式中，但必须修改算法避免可能出现的死循环。