第21章

模板方法（Template Method）模式

普通方法的方法体定义了一系列的指令。方法常常会调用当前对象或其他对象的其他方法。此时，可以将该普通方法看做“模板”，它给出了计算机执行的指令序列。然而，就模板方法模式而言，它还引入了一种更为特殊的模板。

在编写某个方法时，可能需要先定义出算法的轮廓，因为该算法的实现可能五花八门。在这种情况下，就可以定义一个方法，将它的一些步骤定义为抽象方法，或者是存根方法；又或者将它们定义到一个独立的接口中。这样就产生了一个更为严格的“模板”，它特别指出了算法中的哪些步骤是由其他类提供的。

模板方法模式的意图是在一个方法里实现一个算法，并推迟定义算法中的某些步骤，从而让其他类重新定义它们。

经典范例：排序

排序是一种常用的经典算法。假设一位古代女工设计了一种可以根据箭簇的锋利程度对箭进行排序的算法。假定她将箭从左到右排成一排，逐个进行比较，并将更为锋利的箭放在左侧。设计完该算法后，她又认为或许可以根据箭的飞行距离或者其他属性来重新设计排序算法。

不同排序算法的方式和速度是不相同的，但无论是哪种排序算法，都遵循比较两个元素或者属性的基础步骤。如果你设计了一个排序算法，可以比较任意两个元素的某个属性，就可以

通过该属性对元素的集合进行排序。

排序算法是模板方法模式的经典范例。只需修改一个关键步骤——两个对象的比较，就可以对其他对象的各种集合的各种属性进行算法的重用。

近些年来，排序算法可能是重新实现频率最高的算法之一，实现次数或许已经超过了当今程序员的数量。但是，除非你需要对一个超大的集合排序，否则并不需要实现自己的排序算法。

Arrays 类和 Collections 类都提供了 sort()方法。这是一个静态方法，使用一个数组作为参数，并且接收一个可选的比较器 Comparator。ArrayList 类的 sort()方法是一个静态方法，会对 sort()消息的接收者进行排序。换句话说，这些方法共享一个公共策略，该策略依赖于 Comparable 和 Comparator 接口。如图 21.1 所示。

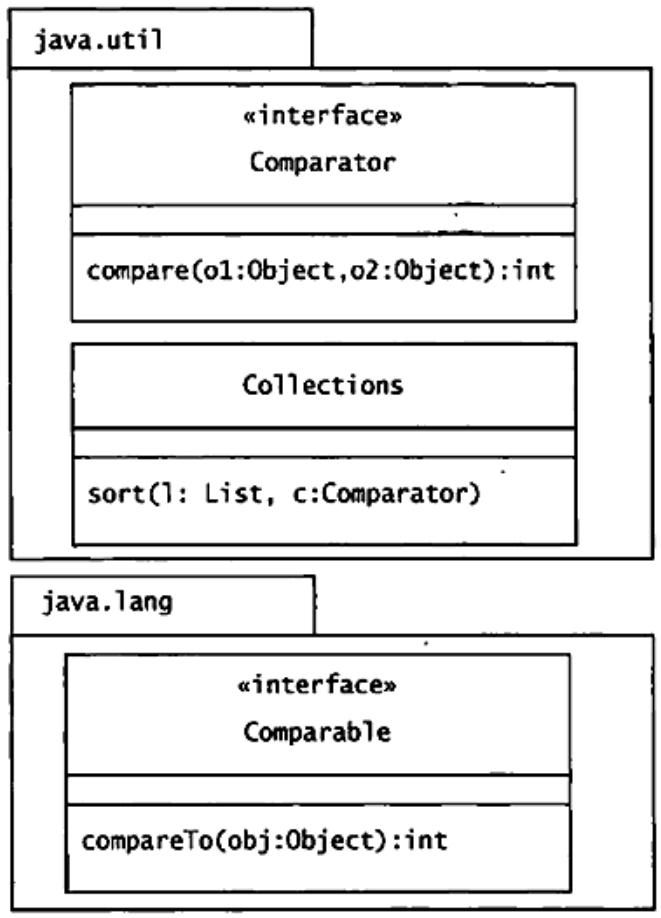

图 21.1　Collections 类中的 sort()方法使用图中的接口

如果需要，Arrays 类和 Collections 类的 sort()方法可以提供一个 Comparator 接口的实例。如果在使用 sort()方法时，没有提供 Comparator 接口的实例，该方法将会依赖 Comparable 接口的 compareTo()方法。如果某些元素没有实现 Comparable 接口，而你却又

未曾提供 Comparator 接口实例，对这些元素排序就会出现异常。当然，在 Java 语言中，绝大多数基础类型，如 String 都实现了 Comparable 接口。

sort()方法为模板方法模式提供了一个范例；类库提供算法，可以让你自己实现比较两个元素的关键步骤。Compare()方法返回一个小于、等于或者大于 0 的数字。这些数字分别对应着对象 o1 小于、等于或者大于对象 o2。例如，下面的代码分别根据火箭的最远点和名称对火箭集合进行排序（火箭构造函数接收的参数包括名字、质量、价格、最远点以及推动力）。

```
package app.templateMethod;

import java.util.Arrays;
import com.oozinoz.firework.Rocket;
import com.oozinoz.utility.Dollars;

public class ShowComparator {
    public static void main(String args[]) {
        Rocket r1 = new Rocket(
            "Sock-it", 0.8, new Dollars(11.95), 320, 25);
        Rocket r2 = new Rocket(
            "Sprocket", 1.5, new Dollars(22.95), 270, 40);
        Rocket r3 = new Rocket(
            "Mach-it", 1.1, new Dollars(22.95), 1000, 70);
        Rocket r4 = new Rocket(
            "Pocket", 0.3, new Dollars(4.95), 150, 20);
        Rocket[] rockets = new Rocket[] { r1, r2, r3, r4 };

        System.out.println("Sorted by apogee: ");
        Arrays.sort(rockets, new ApogeeComparator());
        for (inti = 0; i<rockets.length; i++)
            System.out.println(rockets[i]);
            System.out.println();
            System.out.println("Sorted by name: ");
            Arrays.sort(rockets, new NameComparator());
            for (inti = 0; i<rockets.length; i++)
                System.out.println(rockets[i]);
    }
}
```

下面是 ApogeeComparator 比较器的代码：

```
package app.templateMethod;
```

```
import java.util.Comparator;
import com.oozinoz.firework.Rocket;

public class ApogeeComparator implements Comparator {
    // 挑战!
}
```

NameComparator 的代码如下：

```
package app.templateMethod;

import java.util.Comparator;
import com.oozinoz.firework.Rocket;

public class NameComparator implements Comparator {
    // 挑战!
}
```

程序的输出取决于 Rocket 是如何实现 toString()方法的，但火箭的显示则依照两种排序方式：

```
Sorted by apogee:
Pocket
Sprocket
Sock-it
Mach-it

Sorted by name:
Mach-it
Pocket
Sock-it
Sprocket
```

挑战 21.1

请补充完成 ApogeeComparator 类和 NameComparator 类的代码，使程序可以正确地对火箭集合进行排序。

答案参见第 347 页

排序是一种通用算法，除了一个关键步骤外，和你的领域与应用几乎没有关系。该关键步骤就是对元素的排序。没有哪个通用排序算法包括诸如比较两个火箭最远点的步骤，而是由应用程序来实现这一步骤。Sort()方法和 Comparator 接口可以为通用排序算法提供对该特定步骤的支持。

模板方法模式并不局限于仅缺少领域步骤的情况。有时，会将整个算法应用到具体的应用领域。

完成一个算法

模板方法模式与适配器模式类似，它们都允许开发者简化代码，并允许其他开发者来完成代码的设计。在适配器模式中，设计者可能会为特定对象指定设计需要的接口，而由其他开发者提供该接口的实现。在实现时，应使用实现了不同接口的现有类所提供的服务。在模板方法模式中，设计者可能会提供一个通用的算法，而其他开发者仅提供该算法关键步骤的实现。考虑图 21.2 所示的 Aster 火药球填压机。

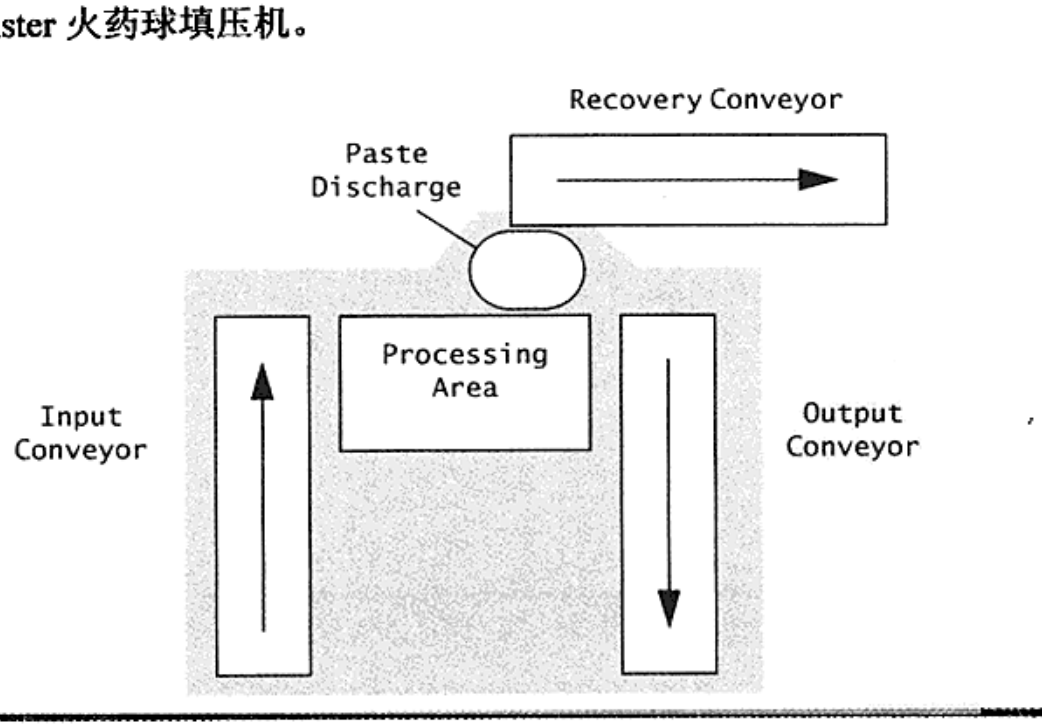

图 21.2 Aster 火药球填压机利用传送带的输入输出来传送火药球模具。Oozinoz 公司增加了一个回收传送带，用于回收被丢弃的火药糊

Aster 公司的火药球填压机接收一个空的金属模具，并填充火药。这个机器有一些料斗，可

以将化学药品混合成糊，然后压进模具。当机器关闭时，它将停止在处理区工作的模具。输入传送带上的模具不会被处理，而是被送到输出传送带上。接着，填压机将卸下当前的火药糊，并用水清洗工作区。填压机通过板载的计算机程序来完成这些操作。**AsterStarPress** 类如图 21.3 所示。

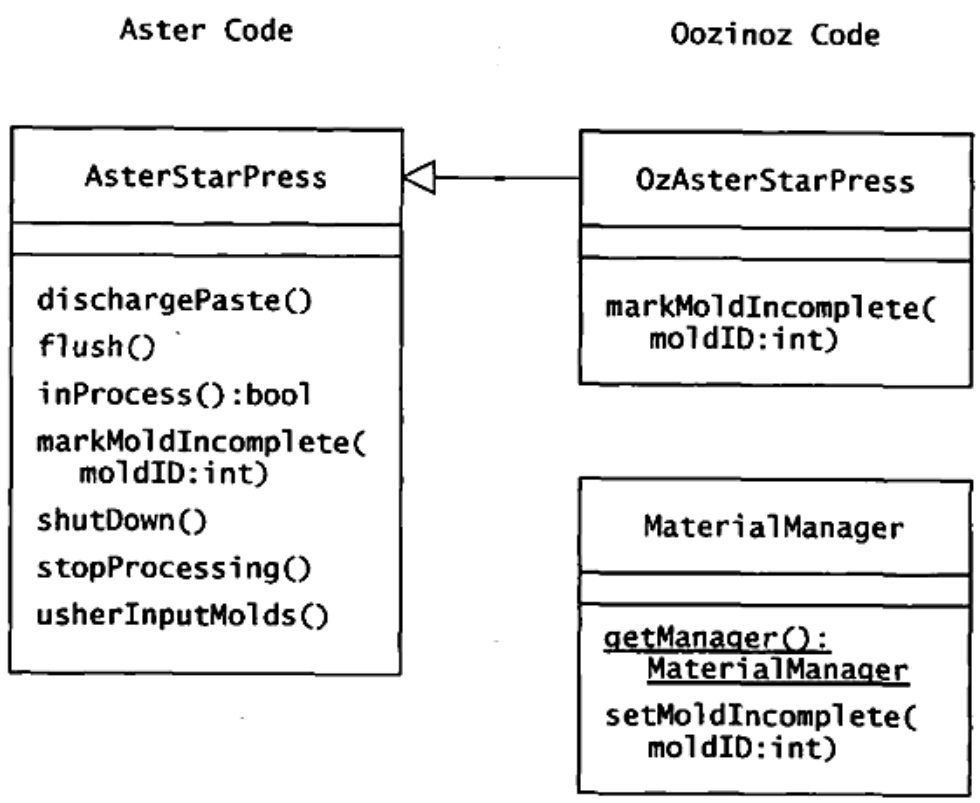

图 21.3　Aster 公司的火药球填压机有一个抽象类，在 Oozinoz 系统中，必须实现该抽象类才能工作

Aster 公司的火药球填压机属于可独立运行的智能机器，公司意识到该机器可能会被用于智能化工厂，并且需要具备通信能力。例如，如果正在处理的填压机尚未处理完毕，**shutDown()** 方法会通知工厂。

```
public void shutdown() {
    if (inProcess()) {
        stopProcessing();
        markMoldIncomplete(currentMoldID);
    }
    usherInputMolds();
    dischargePaste();
    flush();
}
```

markMoldIncomplete()方法和 AsterStarPress 类都是抽象的。Oozinoz 系统需要创建一个子类来实现这些方法，并将这些代码下载到火药球填压机的板载计算机中。可以通过向 MaterialManager 单例传递未完成的模具信息，实现 markMoldIncomplete()方法，以便跟踪原料的状态。

挑战 21.2

写出 OzAsterStarPress 类的 markMoldIncomplete()代码实现：

```
public class OzAsterStarPress extends AsterStarPress {
    public void markMoldIncomplete(int id) {
        // 挑战！
    }
}
```

答案参见第 347 页

Aster 公司负责开发火药球填压机的技术人员对焰火工厂的工作方式了如指掌，并完成了填压机与工厂之间的通信工作。但是，依然有些地方是 Aster 公司的开发人员所没有预见到的，因此需要你去完成这些通信点。

模板方法钩子

钩子（hook）是一个方法回调，它可以让其他开发者将自己的代码插入到程序的指定位置。当你希望使用其他开发者的代码时，或者希望在别的程序中插入自己的代码时，就可以使用钩子。开发者可以在你需要的时候增加一个方法调用。通常，开发者会为钩子方法提供一个存根实现。一旦其他客户端不再需要钩子方法，就没有必要再重写它了。

考虑如下情况：当 Aster 公司的火药球填压机被关闭时，它会倒出火药糊，并自动用水清洗自身。压缩机必须倒出火药糊，以防止它干燥之后会堵塞机器。Oozinoz 公司会回收这些火药糊，以便将它们切成罗马蜡烛中的小火药球（罗马蜡烛是一个固定立方体，它由炸药混合而成，点燃后会有焰火效果）。在火药填压机倒出火药糊后，可以安排一个机器人来将火药糊移动到一个独立的传送带上，如图 21.2 所示。有一点至关重要，就是一定要在机器冲洗工作区前将

火药糊移走。这带来的问题是：需要在 shutdown()方法的两条语句之间获得控制。

```
dischargePaste();
flush();
```

你可能需要重写 dischargePaste()方法，使之增加一个调用，用于收集火药糊。

```
public void dischargePaste() {
    super.dischargePaste();
    getFactory().collectPaste();
}
```

这个方法会在倒出火药糊之后插入一个步骤。增加的步骤使用一个单例工厂方法来收集倾倒的火药糊。执行 shutdown()方法时，工厂机器人会在清洗填压机前收集完所有倾倒的火药糊。不幸的是，dischargePaste()方法存在危险性，因为收集火药糊的方法会产生一些边际效应。Aster 公司的程序员们显然并不知道你是如何定义 dischargePaste()方法的。如果他们修改代码，使得在你不想收集火药糊的时候倾倒它，就会产生错误。

程序员们都会竭力编写代码来解决问题。但是，这里面临的挑战是需要在与其他程序员沟通后，再编写代码解决问题。

挑战 21.3

请为 Aster 公司的程序员写一个通知，要求他们在清洗工作区前，确保已经安全地收集完倾倒的火药糊。

答案参见第 348 页

在模板方法模式中，子类支持的步骤可能被用来完成某些算法；但也可能这些步骤是可选的，通常是根据其他程序员的要求，在子类代码中实现这些钩子方法。尽管该模式的意图是将算法中的某部分分离为一个单独的类，但是，倘若程序多处出现重复的算法方法，也可以将它们重构为模板方法模式。

重构为模板方法模式

在使用模板方法模式时，你会发现在类继承关系中，超类提供的是算法的描述，而子类提

供的是算法具体的步骤。当发现不同的方法具备相似的算法时，就可以将它们重构为模板方法模式（重构是在不改变程序功能的前提下，用更好的设计来重新实现）。考虑第 16 章中的 Machine 和 MachinePlanner 的并行层次。如图 21.4 所示，Machine 类提供了一个 createPlanner()工厂方法，返回一个合适的 MachinePlanner 类的子类。

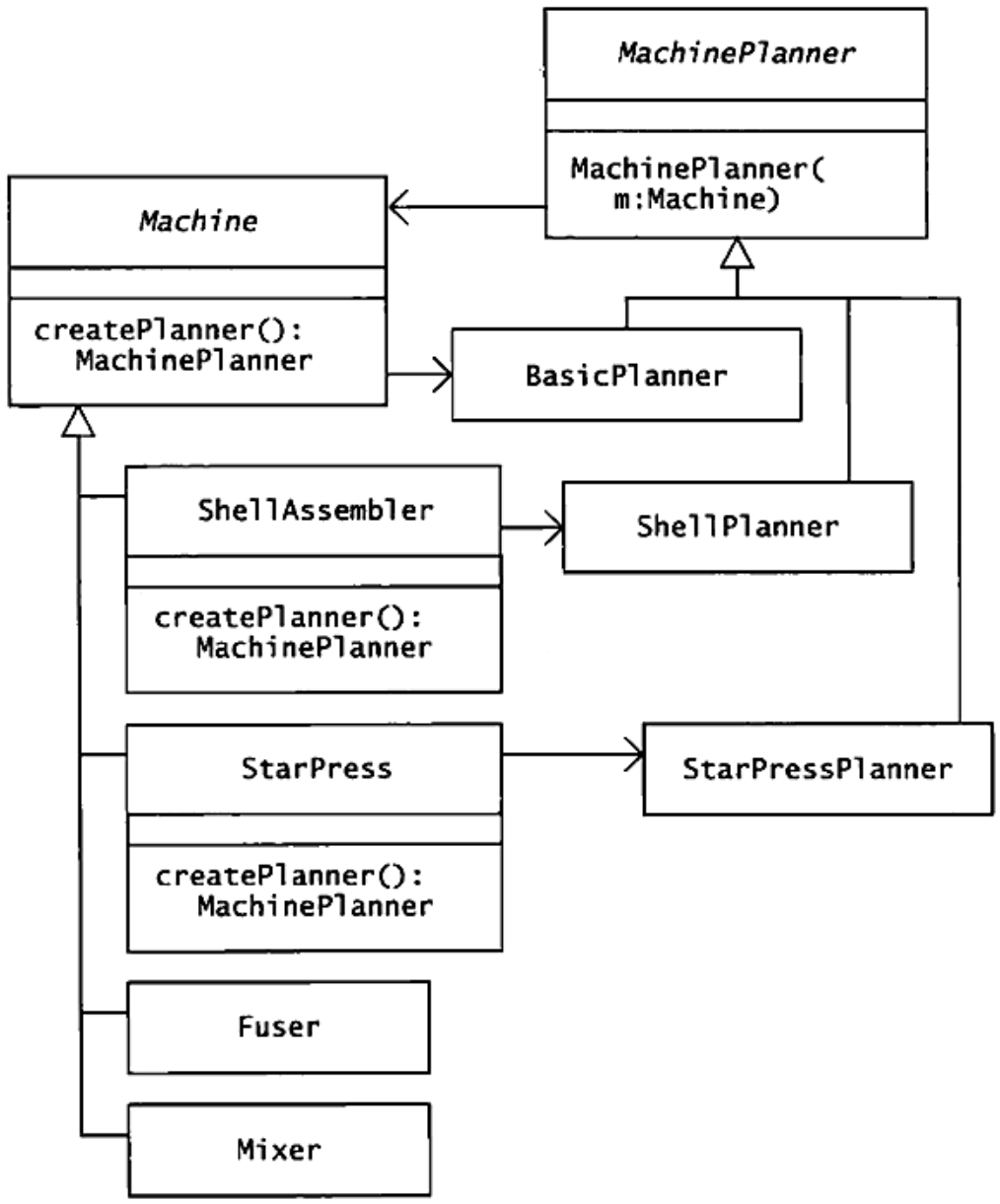

图 21.4 Machine 对象可以为它自己创建合适的 MachinePlanner 对象实例

当要求创建一个规划器时，Machine 类的两个子类会实例化 MachinePlanner 类层次关系中指定的子类。ShellAssembler 类和 StarPress 这两个类有一个共同的需求：它们都希望在需要的时候创建 MachinePlanner 类。

阅读这些类的代码，你会发现子类都使用了同样的延迟初始化技术来创建规划器。例如，

ShellAssembler 类有一个 getPlanner()方法来延迟初始化 planner 成员。

```
public ShellPlanner getPlanner() {
    if (planner == null)
        planner = new ShellPlanner(this);
    return planner;
}
```

在 ShellPlanner 类中，planner 的类型是 ShellPlanner。StarPress 类也有一个 planner 成员，但是类型为 StarPressPlanner。StarPress 类的 getPlanner()方法也会延迟初始化 planner 属性：

```
public StarPressPlanner getPlanner() {
    if (planner == null)
        planner = new StarPressPlanner(this);
    return planner;
}
```

Machine 类的其他子类使用了相同的技术，仅在第一次需要使用规划器的时候才创建它。这就可以通过重构来清理代码，减少我们需要维护的代码。假设你决定提供一个 Machine 类，其中包含类型为 MachinePlanner 的 planner 属性，就可以从子类中删除这个属性，以及已有的 getPlanner()方法。

> **挑战 21.4**
>
> 请写出 Machine 类中 getPlanner()方法的代码。
>
> 答案参见第 348 页

我们经常可以通过抽象出相似的方法轮廓，将其移到超类，并让不同的子类实现算法所需要的不同步骤，从而将代码重构为模板方法模式。

小结

模板方法模式的意图是在一个方法里定义一个算法，抽象某些步骤，或者将它们定义在接

口中，以便其他类可以实现这些步骤。

模板方法模式可以作为开发者之间的一种契约。一些开发者开发算法的框架，而另一些开发者负责实现算法的具体步骤。这些步骤可以是完成算法所必需的步骤，也可以是算法开发者在程序中特定位置设置的钩子。

模板方法模式的意图并不是要求我们在定义子类前写出模板方法。你可能会在已有的类层次关系中发现一些相似的方法。在这种情况下，你可以提取算法的框架，并将它移到超类中，从而运用模板方法模式来简化和组织你的代码。