## 第27章 设计模式总结

## 27.1 回到设计模式的起点

熊小猫：学习完设计模式，你有什么感想吗？

兔小白：现在的程序员能站在大神肩膀上，简直太幸福了！设计软件如探囊取物，匹配合适的设计模式就可以了。

熊小猫：如此说来，还得感谢最开始发明面向对象语言的人。如果没有面向对象语言，也不可能诞生如此一应俱全的设计模式。

兔小白：设计原则也很重要！设计原则贯穿于各种设计模式，是软件设计不可或缺的准则。

熊小猫：面向对象的程序，由对象和对象的关系构成。面向对象程序的设计，本质上是设计对象以及对象间的关系。高内聚、低耦合，再加上可扩展，就是设计的目标。设计原则无一例外都是为了达成这个目标。

兔小白：我有一个困惑，是先有设计原则，还是先有设计模式呢？

熊小猫：哈哈，这是先有鸡还是先有蛋的问题。其实二者都源于人们用程序解决问题时的思考，是长期共同演化，从产生再到逐步完善的过程。

早在设计模式的概念被提出前，开发者已经在反复做设计方案的过程中，不断总结经验，提炼设计原则。有了初步的设计原则，设计质量和速度得到提升，从而加快了程序设计的发展。更多设计原则被总结出来，更多优秀的设计方案也涌现了出来。

这个时候，虽然没有设计模式的概念，但被反复使用的优秀设计方案已经被总结、沉淀下来，并在小范围内传播。直到GOF将问题分类，映射到固定的解决方案，设计模式的概念终于在软件领域横空出世。从此，设计模式被提升到理论的高度，被高度规范化。

兔小白：我理解二者是对设计经验在不同方向上的总结。设计原则是理论方向，设计模式是实践方向。设计模式虽然好用，但初学者面对如此多的设计模式，很容易选择困难，甚至犯错。

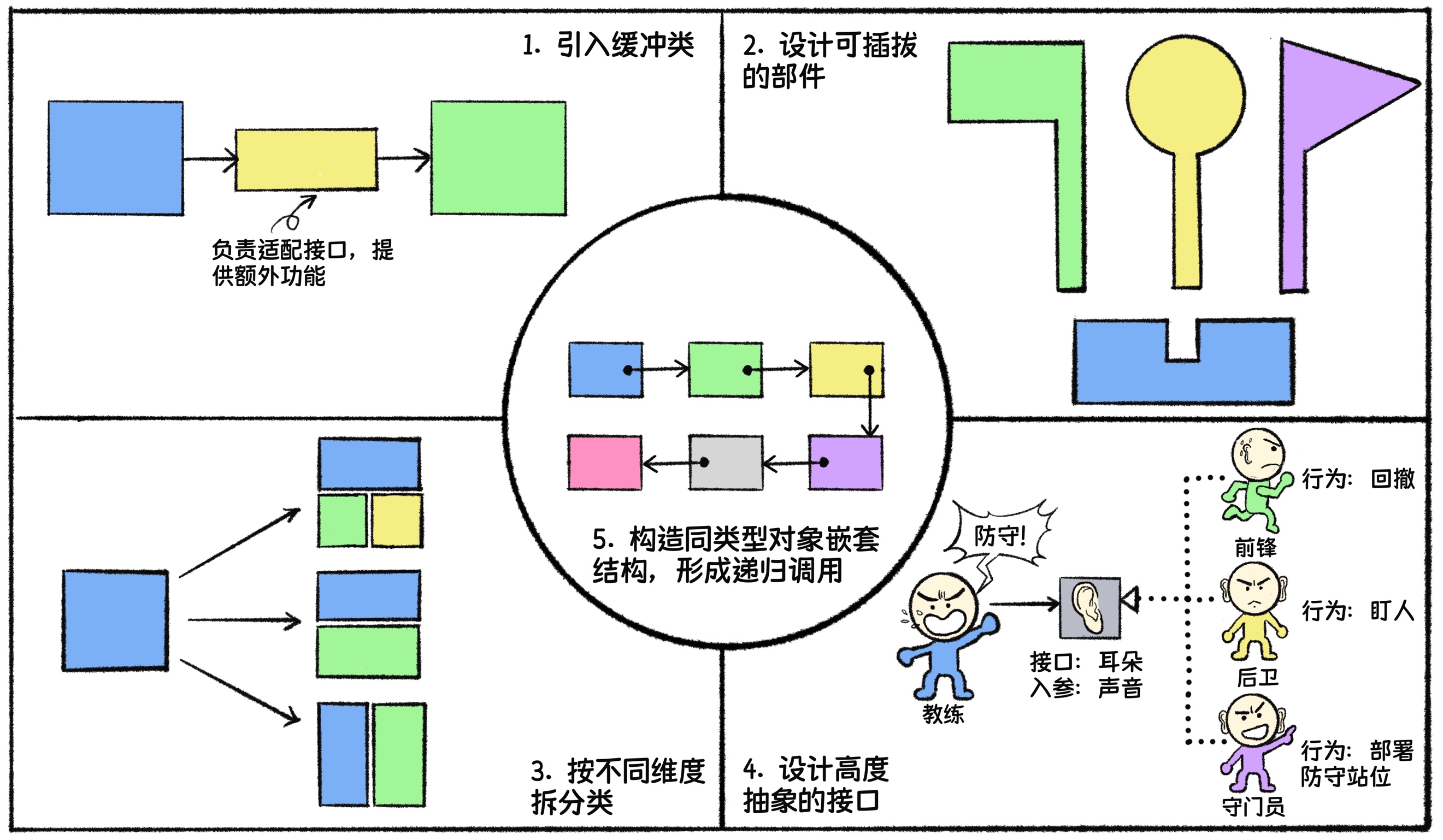
熊小猫：设计模式的数量虽多，但都是为了达到高内聚、低耦合、可扩展的设计目的。各种设计模式使用的手法[[1]](#footnote-1)1，其实存在很多相同之处。当你了解了这些设计手法，即使不去刻意匹配设计模式，也能设计出可复用的软件。

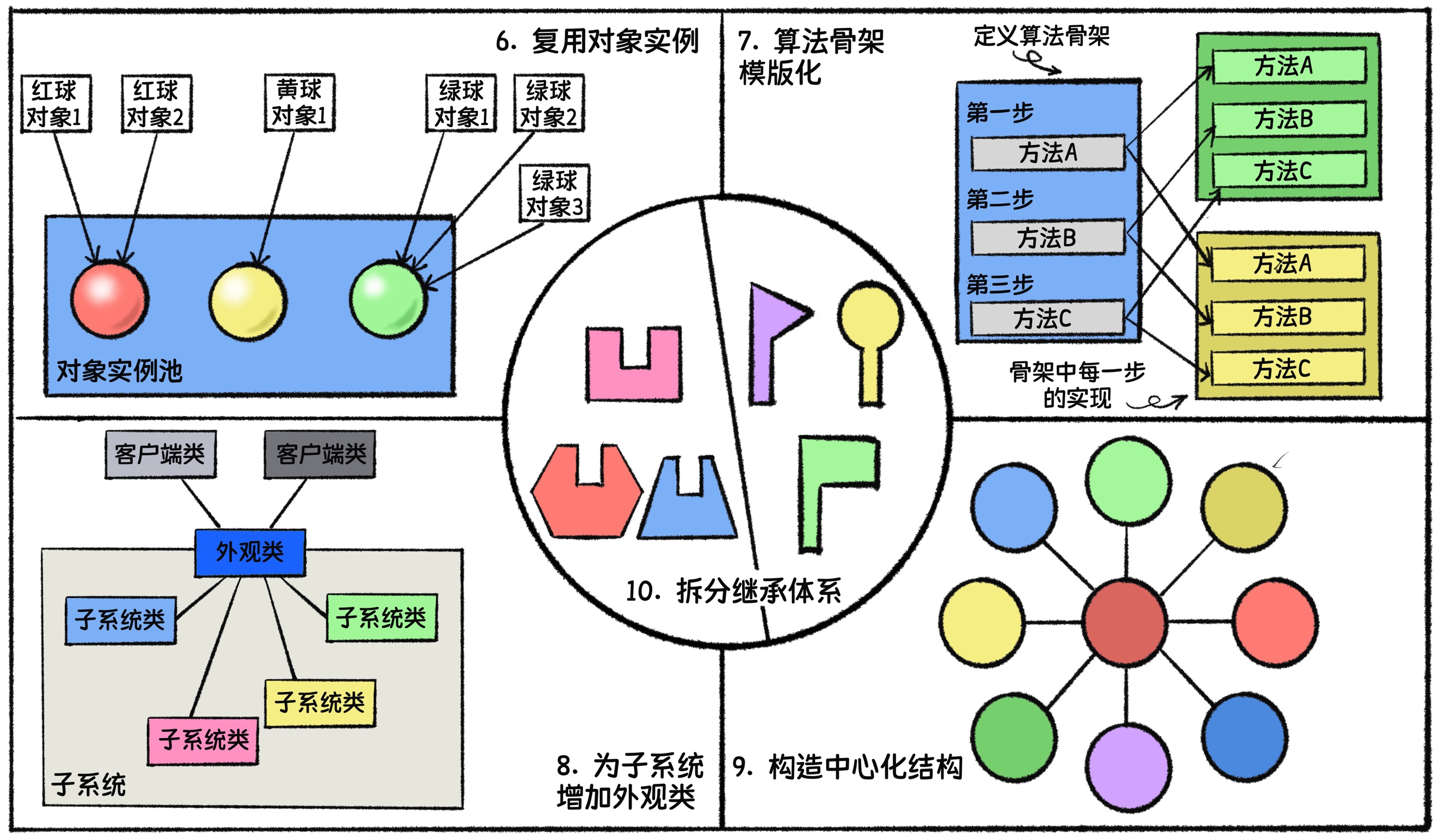
兔小白：设计手法？这是个新的概念，展开详细说说吧！

## 27.2 10种常用的设计手法

熊小猫：我在长期的软件设计工作中，总结出下面10种设计手法。这些手法是一些实用技巧，不涉及特定的问题，只是为了达到高内聚、低耦合、可扩展的设计目的。

我已经提前画好了这10种设计手法的示意图，你可以边听我讲，边对照示意图理解。





1. **引入缓冲类**

* 解释

在两个存在依赖关系的类间，引入一个新的类。这个类提供了缓冲空间，带来了两个好处。一是使得两端的主要部件类能够保持稳定，变化的部分放在缓冲类中实现。二是能够在缓冲类上实现增强，比如提供额外的功能、接口适配等等。

* 例子
* 适配器模式。Adapter是Target和Adaptee之间的缓冲类，承担接口对接的职责。Adapter的引入，使得Target可以调用到Adaptee，而不需要它们双方做任何改变。
* 代理模式。Proxy使得RealSubject可以专注于处理稳定的核心业务。多变的周边业务或者基于核心的增强业务，放在Proxy中实现。未来的变化，将更多发生在Proxy上，从而保证了RealSubject的稳定。
* 还有很多种设计模式都体现了这种思想。比如命令模式，通过Command类，将不同的Reveiver的封装成相同的接口，对外提供一致的调用方式。

1. **设计可插拔的部件**

* 解释

在程序中，不同功能的类通过组合，构成提供完整功能的工具。构成工具的功能部件类，应该遵循依赖倒置的原则，设计成可以替换的形式，提升软件的扩展性。

* 例子
* 生成器模式。Director和Builder类协作完成构造对象的职责。Director可以插拔不同类型的Builder实现对象，从而改变构造对象的行为逻辑。
* 桥接模式。Abstraction和Implementor共同完成某个功能。Abstraction可以插拔不同类型的Implementor实现对象，从而改变自己的功能。
* 其它例子还有状态模式，Context可以插拔各种State实现对象。而在策略模式中，Context可以插拔各种Strategy的实现对象。

1. **按不同维度拆分类**

* 解释

为了满足单一职责和开闭原则，往往需要将类进行拆分。我们基于对现实世界的直觉，一般是按功能对类进行垂直拆分。在此之外，还有以下几个拆分原则可以使用：**分离稳定和不稳定行为，分离数据和行为，分离高层和低层逻辑。**

* 例子
* 桥接模式。该模式将原本的一个继承体系拆分为两个继承体系。Abstraction和Implementor，这两个继承体系配合工作，独立演变，互不影响。
* 生成器模式。该模式将生产对象的职责拆分到Director和Builder两个类中。Director负责高层、稳定的主流程实现。Builder负责易发生变化的每个步骤的细节实现。
* 访问者模式。该模式拆分了元素和作用于元素的行为。Element为元素实现，主要负责数据维护。Visitor负责作用于元素的行为。
* 其他例子还有很多。例如模版方法，它类似于生成器模式，分离了稳定的主流程和不稳定的每个流程步骤。而备忘录模式将对象的状态分离成为Memento类，保存状态数据的快照。

1. **设计高度抽象的接口**

* 解释

接口的抽象程度越高，个性化就越弱。高度抽象的接口可以将不同行为统一成相同接口，提供一致的调用方式。但随之而来的是，由于要考虑所有行为的输入参数，接口的入参会变得复杂。另外接口命名不能准确反应业务逻辑，降低了程序的可读性。解决办法是通过构造单一职责的类，通过类名表示接口的真正业务。

* 例子
* 命令模式。命令模式将Command实现类的调用方法统一为execute。每种Command类只实现一种命令。通过Command实现类的命名，来反映该命令的业务含义。
* 观察者模式。在该模式中，所有的Observer都是通过update方法对外提供接收通知的接口。这是一个高度抽象的方法，完全忽略了观察者业务的不同。任何观察者所需要的数据，都需要被定义在Subject中，作为参数，传递给update方法。
* 其他例子还有组合模式，Leaf和Composite类的行为要保持一致。而在访问者模式中，无论Visitor子类所代表的行为是什么，都统一对外提供visit方法。

1. **构造同类型对象嵌套结构，形成递归调用**

* 解释

这是一种去中心化的思想。该手法将职责细化，分解到各个单一职责的子类中。然后运用插拔的思想，将若干个子类以嵌套的方式连接起来。对外提供的调用方法，以递归的方式，让所有连接在一起的子类，依次执行各自的逻辑。这种方式不但易于扩展职责，而且可以灵活组装已有职责，构成新的职责，而不需要编写新的类。

* 例子
* 装饰模式。装饰模式是该手法的典型应用案例。Decorator维护一个Component类型的对象，而它自己也是Component的子类。因此Decorator对象间可以随意插拔，串联成链条。Decorator以递归的方式，依次调用链条上每一个Component类型对象的operation方法。
* 职责链模式。职责链模式是该手法的另外一种典型应用案例。多个不同职责的Handler对象，通过插拔的方式串联成职责链。每个Handler根据自己的职责，对请求进行处理。非自己职责范围的请求，则传递给下一个Handler。
* 其他例子还有组合模式和解释器模式。其中解释器模式更具代表性，它将TerminalExpression和NoterminalExpression的子类对象，组装成抽象语法树，递归调用每个对象的interpret方法，完成语句的解释执行。

1. **复用对象实例**

* 解释

该手法限制一个类只能被创建出有限数量的实例，可能只有一个实例，也可能按照对象种类生成相应数目的实例。目的是节省存储空间。此外，某些场景只允许存在一个实例。

* 例子
* 单例模式。Singleton类限制只能通过instance方法创建它的实例。instance方法需要保证只会创建一个实例。
* 享元模式。FlyweightFactory维护一个享元对象池，每类享元对象，只会维护一个实例。例如围棋的棋子对象，只需要维护黑色、白色两个棋子实例。

1. **算法骨架模版化**

* 解释

该手法将职责分层。顶层是稳定的算法骨架，将其模板化，以便复用。下一层是骨架中每一步的细节逻辑，变化相对频繁，由子类负责不同实现。算法骨架模版化，既确保了算法过程的一致性，又使得算法实现具备扩展性。

* 例子
* 模版方法模式。父类AbstractClass中的模版方法templateMethod，定义了算法骨架。骨架中每一步的primitiveOperation方法由子类实现。
* 生成器模式。Director将构建对象的步骤模版化为construct方法。Builder负责每一步的具体实现。

1. **为子系统增加外观类**

* 解释

由多个类构成的子系统，可以通过增加类似外观类的方式，对外提供一致的入口类和统一的调用方式。外观类屏蔽了子系统内部的复杂性，将依赖关系局限在子系统内部，降低了程序的整体耦合度。

* 例子
* 外观模式。Facade类为子系统提供一致的入口。它将客户端的请求代理给适合的子系统对象，避免客户端类和子系统多个内部类直接依赖。
* 策略模式。Context类屏蔽了Client和不同Strategy子类的直接耦合。策略模式结合简单工厂后，Context作为策略组件的外观类，对外提供统一的调用接口。

1. **构造中心化结构**

* 解释

虽然去中心化已经成为主流，但是中心化也有其存在的价值。在关系中心的类，负责转发请求、联通组件内的不同类。当一个系统内部类的关系复杂，呈现出网状时，可以用中心化结构简化类的关系。

* 例子
* 中介者模式。中介者模式中的Mediator有着丰富的“人脉”，它依赖每一个Colleague子类，集所有Colleague能力为一身。Colleague不需要相互调用，只需要调用Mediator。Mediator会负责对接合适的Colleague。中介者模式将依赖关系集中在Mediator身上，消除了Colleague之间的网状依赖关系。

1. **拆分继承体系**

* 解释

大部分设计模式中，只需要一套继承体系便可以获得想要的扩展性。但在一些比较复杂的场景中，主体对象存在两个不同维度的扩展方向，使用一套继承体系不够灵活。将一套继承体系拆分成两套继承体系后，两套继承体系可以在不同维度上独立发展，互不影响。而两套继承体系的对象构成一个整体，配合工作，实现原有功能。

* 例子
* 桥接模式。桥接模式中存在Abstraction和Implementor两套继承体系。Abstraction中那些变化频繁的行为，可以自立门户，形成Implementor体系。Abstraction使用Implementor来实现原有行为。
* 访问者模式。访问者模式将作用于Element的操作，抽象成Visitor继承体系。虽然访问者模式中存在Element和Visitor两套继承体系，但是要注意，Element的扩展性并不好，扩展Element，需要修改所有已存在的Visitor。

## 27.3 实践是唯一出路

熊小猫：我全部的软件设计经验，都毫无保留地传授给了你。但是经验属于不可言传，只可意会的知识，还需要你自己用心去体会。

兔小白：你给我讲的内容，我都记录下来了，回去好好研究！

熊小猫：研究不能停留在纸面上。掌握设计模式的唯一出路就是去大胆实践，别怕犯错。

兔小白：放心吧，熊老师！我会努力将设计模式的思想和方法融入到我的开发工作中，设计出可复用的面向对象软件。

## 27.4 尾声

兔小白和熊小猫学习设计模式的故事，到此便告一段落。兔小白在日后的工作中，更加重视程序设计，考虑问题也更加全面。虽然，他最初对设计模式的使用有些生硬，甚至还犯过错误，但他积极吸收别人的建议，及时反思和总结，成长的非常快。现在他不但自己能够灵活运用设计模式，还经常给别的同事提出设计建议。他一直保持着学习的习惯和追求技术卓越的热情，在通往优秀架构师的道路上持续前进！

兔小白学习技术的旅程才刚刚开始，后面还有更精彩的故事等待发生......

1. 1 这里的手法指不考虑特定问题，达到设计目的的技巧。 [↑](#footnote-ref-1)