设计模式

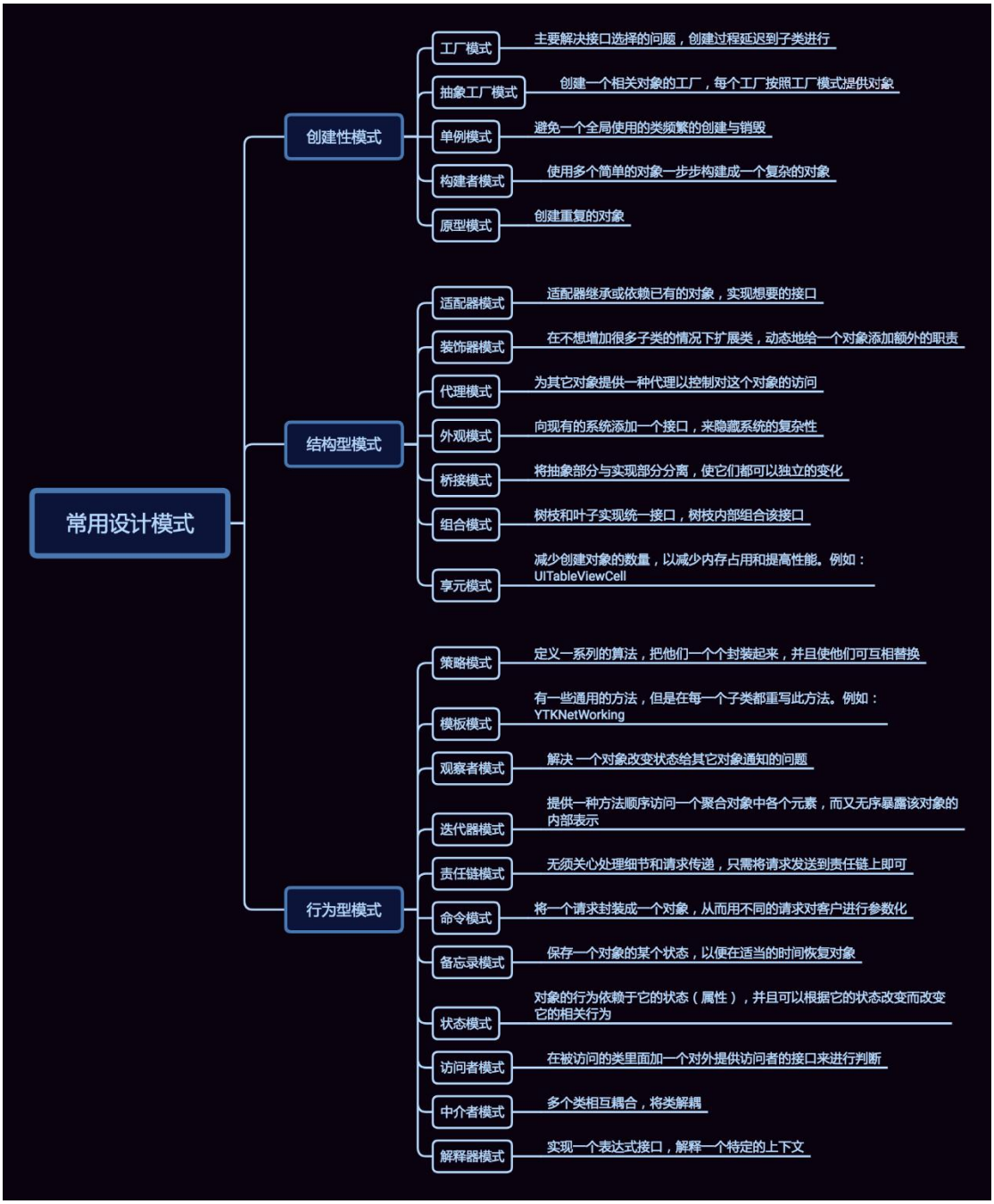


图 1.　　设计模式导图

一.　单例模式

提供一个访问该类的全局访问点，并保证该类仅有一个实例。单例模式不仅可以保证实例的唯一性，还可以严格的控制外部对实例的访问。

1. 问题

单例模式同时解决了两个问题，所以违反了单一职责原则。

1)保证一个类只有一个实例

为什么会有人想要控制一个类所拥有的实例数量？最常见的原因是控制某些共享资源

（例如数据库或文件）的访问权限。

它的运作方式是这样的：如果我们创建了一个对象，同时过一会儿后你决定再创建一个新对象，此时我们会获得之前已创建的对象，而不是一个新对象。

注意，普通构造函数无法实现上述行为，因为构造函数的设计决定了它必须总是返回一个新对象。

2)为该实例提供一个全局访问节点

还记得我们用过的那些存储重要对象的全局变量吗？它们在使用上十分方便，但同时也非常不安全，因为任何代码都有可能覆盖掉那些变量的内容，从而引发程序崩溃。

和全局变量一样，单例模式也允许在程序的任何地方访问特定对象。但是它可以保护该实例不被其他代码覆盖。

还有一点：我们不会希望解决同一个问题的代码分散在程序各处的。因此更好的方式是将其放在同一个类中，特别是当其他代码已经依赖这个类时更应该如此。

2. 解决方案

所有单例的实现都包含以下两个相同的步骤：

将默认构造函数设为私有，防止其他对象使用单例类的 new 运算符。

新建一个静态构建方法作为构造函数。该函数会“偷偷”调用私有构造函数来创建对象，并将其保存在一个静态成员变量中。此后所有对于该函数的调用都将返回这一缓存对象。

如果我们的代码能够访问单例类，那它就能调用单例类的静态方法。无论何时调用该方法， 它总是会返回相同的对象。

3. 真实世界类比

政府是单例模式的一个很好的示例。一个国家只有一个官方政府。不管组成政府的每个人的身份是什么， “某政府”这一称谓总是鉴别那些掌权者的全局访问节点。

4. 单例模式适合应用场景

如果程序中的某个类对于所有客户端只有一个可用的实例，可以使用单例模式。

单例模式禁止通过除特殊构建方法以外的任何方式来创建自身类的对象。该方法可以创建一个新对象，但如果该对象已经被创建，则返回已有的对象。

如果我们需要更加严格地控制全局变量，可以使用单例模式。

单例模式与全局变量不同，它保证类只存在一个实例。除了单例类自己以外，无法通过任何方式替换缓存的实例。

请注意，我们可以随时调整限制并设定生成单例实例的数量，只需修改获取实例方法，即 getInstance 中的代码即可实现。

5. Unity 泛型单例

1) 泛型参数的约束

在定义泛型类时，可以对客户端代码能够在实例化类时用于类型参数的类型种类施加限制。如果客户端代码尝试使用某个约束所不允许的类型来实例化类，则会产生编译时错误。这些限制称为约束。约束是使用 where 上下文关键字指定的。下表列出了六种类型的约束：

where T: struct

类型参数必须是值类型。可以指定除 Nullable 以外的任何值类型。有关更多信息，请参见使用可以为 null 的类型（C# 编程指南）。

where T : class

类型参数必须是引用类型；这一点也适用于任何类、接口、委托或数组类型。

where T：new()

类型参数必须具有无参数的公共构造函数。当与其他约束一起使用时，new() 约束必须

最后指定。

where T：<基类名>

类型参数必须是指定的基类或派生自指定的基类。

where T：<接口名称>

类型参数必须是指定的接口或实现指定的接口。可以指定多个接口约束。约束接口也可以是泛型的。

where T：U

为 T 提供的类型参数必须是为 U 提供的参数或派生自为 U 提供的参数。

2) 完整代码

PS：

```
using UnityEngine;

public class SingleComponent<T> : MonoBehaviour where T : SingleComponent<T>
{
    //设置为私有字段的变量，在派生类中是无法访问的
    private static T instance;
    public static T Instance//[单例模式]通过关键字 Static 和其他语句之生成对象的一个实例，确保该脚本在场景中的唯一性
    {
        get
        {
            return instance;
        }
    }

    //关键字 Virtual，使此函数变为一个虚函数，让其可以在一个或多个派生类中被重新定义。
    protected virtual void Awake()
    {
        if (instance != null)//场景中的实例不唯一时
        {
            Destroy(instance.gameObject);
        }
        else
        {
            instance = this as T;
        }
    }
    //设置为 protected 类型的变量，派生类可以访问，非派生类无法直接访问
    protected virtual void OnDestroy()
    {
        Destroy(instance.gameObject);
    }
```

```
        //如果遇到报错：Some objects were not cleaned up when closing the scene. (Did you spawn new GameObjects from OnDestroy?)
        //请在 OnDestory 调用单例的地方使用这个判断一下
        /// <summary>
        ///  单例是否已初始化
        /// </summary>
        public static bool IsInitialized
        {
            get
            {
                return instance != null;//返回的是一个 bool 值，即该实例是否为空
            }
        }
    }
```

6. 单例模式优缺点

1) 优点

可以保证一个类只有一个实例。

获得了一个指向该实例的全局访问节点。

仅在首次请求单例对象时对其进行初始化。

2)缺点

违反了单一职责原则。该模式同时解决了两个问题。

单例模式可能掩盖不良设计，比如程序各组件之间相互了解过多等。

该模式在多线程环境下需要进行特殊处理，避免多个线程多次创建单例对象。

单例的客户端代码单元测试可能会比较困难，因为许多测试框架以基于继承的方式创建模拟对象。由于单例类的构造函数是私有的，而且绝大部分语言无法重写静态方法，所以你需要想出仔细考虑模拟单例的方法。要么干脆不编写测试代码，或者不使用单例模式。

二. 观察者模式

观察者模式定义了一种一对多的依赖关系，让多个观察者对象同时监听某一个主题对象，这个主题对象在状态发生变化时，会通知所有观察者对象，使它们能够自动更新自己的行为。

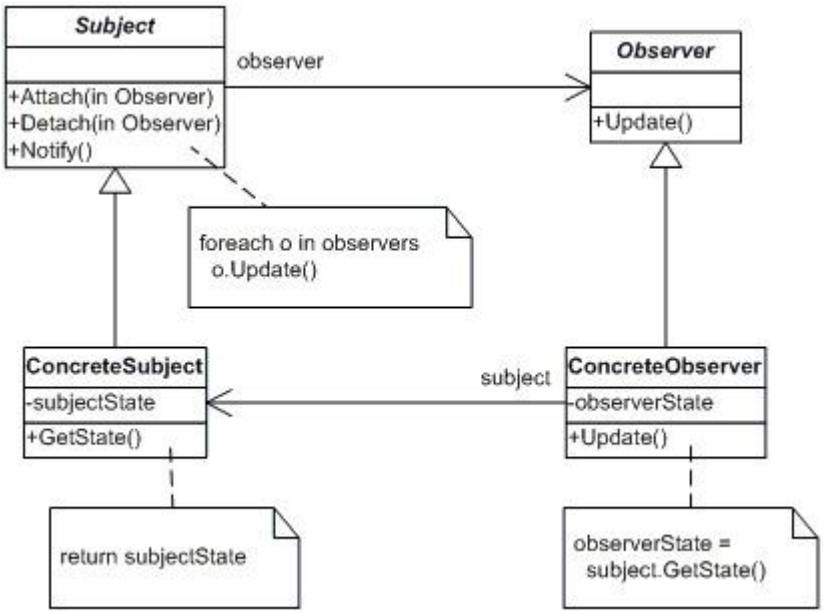
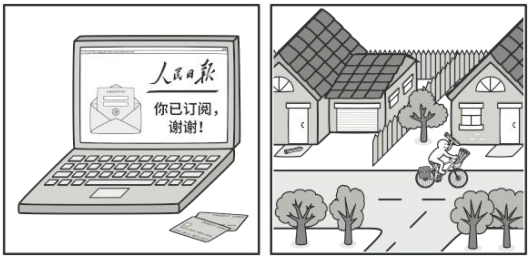


图 2. 图 2.1 观察者模式结构图

在 Unity 中，观察者模式的实现是基于 C#的事件委托机制，要彻底的解除通知者和观察者之间的耦合就可以采用事件委托（Delegate）。委托就是一种引用方法的类型，拥有参数和返回值，作为一种方法的抽象，是特殊的“函数类”，一旦我们为委托分配了方法，委托将会执行与该方法相同的行为，其实例代表了一个或多个具体的函数。C#中关于回调函数的实现机制便是委托，事件（Event）是基于委托的，事件为委托提供了一种用于实现类之间消息发布与接收的结构。在 C#中订阅与取消事件可以使用“+=”与“-=”，在 Unity 中可以通过添加 Handler 来监听事件消息，进而实现消息响应，可以方便的创建事件与编写响应，通过创建 Handle 来监听事件进而完成游戏逻辑。事件只能被其他的模块监听注册，但是不能由其他的模块触发。要触发事件，只能由模块内的公开方法在其内部进行事件触发，这样可以使得程序更加安全。但是在委托的使用过程中会出现过于依赖反射机制（reflection）来查找消息对应的被调用函数，这会影响部分性能。

1. 真实世界类比



如果我们订阅了一份杂志或报纸，那就不需要再去报摊查询新出版的刊物了。出版社（即应用中的“发布者”）会在刊物出版后（甚至提前）直接将最新一期寄送至你的邮箱中。

出版社负责维护订阅者列表，了解订阅者对哪些刊物感兴趣。当订阅者希望出版社停止寄送新一期的杂志时，他们可随时从该列表中退出。

2. 观察者模式适合应用场景

当一个对象状态的改变需要改变其他对象，或实际对象是事先未知的或动态变化的时，可使用观察者模式。

当我们使用图形用户界面类时通常会遇到一个问题。比如，创建了自定义按钮类并允许客户端在按钮中注入自定义代码，这样当用户按下按钮时就会触发这些代码。

观察者模式允许任何实现了订阅者接口的对象订阅发布者对象的事件通知。我们可在按钮中添加订阅机制，允许客户端通过自定义订阅类注入自定义代码。

当应用中的一些对象必须观察其他对象时，可使用该模式。但仅能在有限时间内或特定情况下使用。

订阅列表是动态的，因此订阅者可随时加入或离开该列表。

3. 优点

第一、观察者模式在被观察者和观察者之间建立一个抽象的耦合。被观察者角色所知道的只是一个具体观察者列表，每一个具体观察者都符合一个抽象观察者的接口。被观察者并不认识任何一个具体观察者，它只知道它们都有一个共同的接口。

由于被观察者和观察者没有紧密地耦合在一起，因此它们可以属于不同的抽象化层次。如果被观察者和观察者都被扔到一起，那么这个对象必然跨越抽象化和具体化层次。

第二、观察者模式支持广播通讯。被观察者会向所有的登记过的观察者发出通知，

观察者模式可以实现表示层和数据逻辑层的分离，并定义了稳定的消息更新传递机制，抽象了更新接口，使得可以有各种各样不同的表示层作为具体观察者角色。

观察者模式在观察目标和观察者之间建立一个抽象的耦合。

观察者模式支持广播通信。

观察者模式符合“开闭原则”的要求。

4. 缺点

第一、如果一个被观察者对象有很多的直接和间接的观察者的话，将所有的观察者都通知到会花费很多时间。

第二、如果在被观察者之间有循环依赖的话，被观察者会触发它们之间进行循环调用，导致系统崩溃。在使用观察者模式是要特别注意这一点。

第三、如果对观察者的通知是通过另外的线程进行异步投递的话，系统必须保证投递是以自恰的方式进行的。

第四、虽然观察者模式可以随时使观察者知道所观察的对象发生了变化，但是观察者模式没有相应的机制使观察者知道所观察的对象是怎么发生变化的。

5. 模式扩展

MVC 模式是一种架构模式，它包含三个角色：模型(Model)，视图(View)和控制器(Controller)。观察者模式可以用来实现 MVC 模式，观察者模式中的观察目标就是 MVC 模式中的模型(Model)，而观察者就是 MVC 中的视图(View)，控制器(Controller)充当两者之间的中介者(Mediator)。当模型层的数据发生改变时，视图层将自动改变其显示内容。

6. 与其他模式的关系

责任链模式、命令模式、中介者模式和观察者模式用于处理请求发送者和接收者之间的不同连接方式：

责任链按照顺序将请求动态传递给一系列的潜在接收者， 直至其中一名接收者对请求进行处理。

命令在发送者和请求者之间建立单向连接。

中介者清除了发送者和请求者之间的直接连接，强制它们通过一个中介对象进行间接沟通。

观察者允许接收者动态地订阅或取消接收请求。

中介者和观察者之间的区别往往很难记住。在大部分情况下，我们可以使用其中一种模式，而有时可以同时使用。让我们来看看如何做到这一点。

中介者的主要目标是消除一系列系统组件之间的相互依赖。这些组件将依赖于同一个中

介者对象。观察者的目标是在对象之间建立动态的单向连接，使得部分对象可作为其他对象的附属发挥作用。

有一种流行的中介者模式实现方式依赖于观察者。中介者对象担当发布者的角色，其他组件则作为订阅者，可以订阅中介者的事件或取消订阅。当中介者以这种方式实现时， 它可能看上去与观察者非常相似。

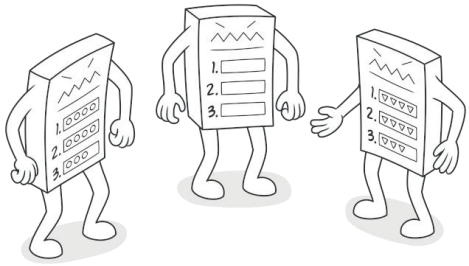
当感到疑惑时，记住可以采用其他方式来实现中介者。例如，可永久性地将所有组件链接到同一个中介者对象。这种实现方式和观察者并不相同，但这仍是一种中介者模式。

假设有一个程序，其所有的组件都变成了发布者，它们之间可以相互建立动态连接。 这样程序中就没有中心化的中介者对象，而只有一些分布式的观察者。

三. 模板方法模式——Template Method

1. 意图

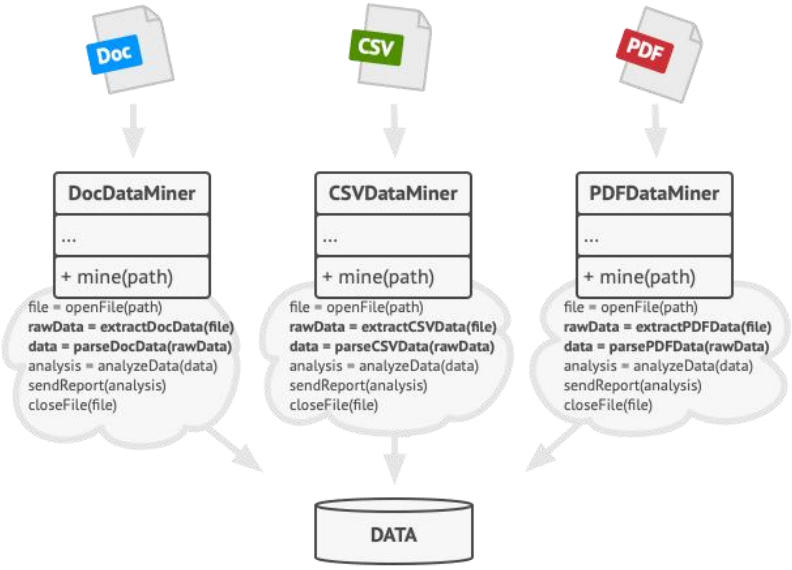
模板方法模式是一种行为设计模式，它在超类中定义了一个算法的框架，允许子类在不修改结构的情况下重写算法的特定步骤。就像填空题一样，只要填入的内容适当就没有问题。



2. 问题

假如我们正在开发一款分析公司文档的数据挖掘程序。用户需要向程序输入各种格式（PDF、DOC 或 CSV）的文档，程序则会试图从这些文件中抽取有意义的数据，并以统一的格式将其返回给用户。

该程序的首个版本仅支持 DOC 文件。在接下来的一个版本中，程序能够支持 CSV 文件。一个月后，又“教会”了程序从 PDF 文件中抽取数据。



一段时间后，我们就会发现这三个类中包含许多相似代码。尽管这些类处理不同数据格

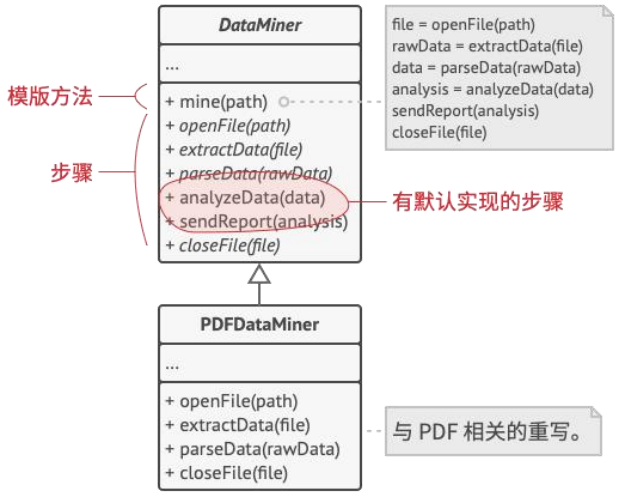
式的代码完全不同，但数据处理和分析的代码却几乎完全一样。如果能在保持算法结构完整的情况下去除重复代码，这难道不是一件很棒的事情吗？

还有另一个与使用这些类的客户端代码相关的问题：客户端代码中包含许多条件语句，以根据不同的处理对象类型选择合适的处理过程。如果所有处理数据的类都拥有相同的接口或基类，那么我们就可以去除客户端代码中的条件语句，转而使用多态机制来在处理对象上调用函数。

3. 解决方案

模板方法模式建议将算法分解为一系列步骤，然后将这些步骤改写为方法，最后在“模板方法”中依次调用这些方法。步骤可以是抽象的，也可以有一些默认的实现。为了能够使用算法，客户端需要自行提供子类并实现所有的抽象步骤。如有必要还需重写一些步骤（但这一步中不包括模板方法自身）。

让我们考虑如何在数据挖掘应用中实现上述方案。我们可为图中的三个解析算法创建一个基类，该类将定义调用了一系列不同文档处理步骤的模板方法。



首先，我们将所有步骤声明为抽象类型，强制要求子类自行实现这些方法。在我们的例子中，子类中已有所有必要的实现，因此我们只需调整这些方法的签名，使之与超类的方法匹配即可。

现在，让我们看看如何去除重复代码。对于不同的数据格式，打开和关闭文件以及抽取和解析数据的代码都不同，因此无需修改这些方法。但分析原始数据和生成报告等其他步骤的实现方式非常相似，因此可将其提取到基类中，以让子类共享这些代码。

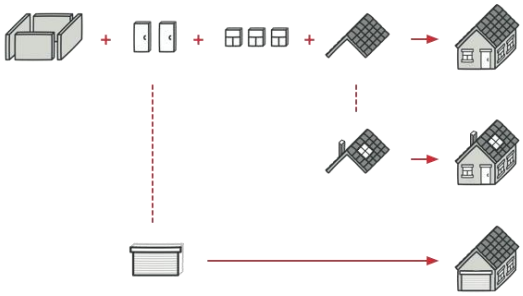
正如所看到的那样，我们有两种类型的步骤：

抽象步骤必须由各个子类来实现

可选步骤已有一些默认实现，但仍可在需要时进行重写

还有另一种名为钩子的步骤。钩子是内容为空的可选步骤。即使不重写钩子，模板方法也能工作。钩子通常放置在算法重要步骤的前后，为子类提供额外的算法扩展点。

4. 真实世界类比

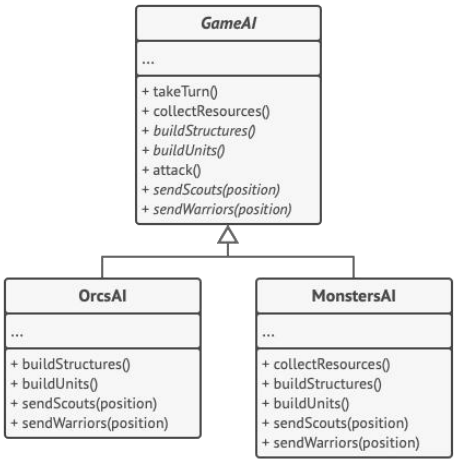


模板方法可用于建造大量房屋。标准房屋建造方案中可提供几个扩展点，允许潜在房屋业主调整成品房屋的部分细节。

每个建造步骤（例如打地基、建造框架、建造墙壁和安装水电管线等）都能进行微调，这使得成品房屋会略有不同。

5. 伪代码

本例中的模板方法模式为一款简单策略游戏中人工智能的不同分支提供“框架”。



游戏中所有的种族都有几乎同类的单位和建筑。因此我们可以在不同的种族上复用相同的 AI 结构，同时还需要具备重写一些细节的能力。通过这种方式，我们可以重写半兽人的 AI 使其更富攻击性，也可以让人类侧重防守，还可以禁止怪物建造建筑。在游戏中新增种族需要创建新的 AI 子类，还需要重写 AI 基类中所声明的默认方法。

PS：

```
// 抽象类定义了一个模板方法，其中通常会包含某个由抽象原语操作调用组成的算
// 法框架。具体子类会实现这些操作，但是不会对模板方法做出修改。
class GameAI is
    // 模板方法定义了某个算法的框架。
    method turn() is
        collectResources()
        buildStructures()
        buildUnits()
        attack()

    // 某些步骤可在基类中直接实现。
    method collectResources() is
```

```
            foreach (s in this.builtStructures) do
                s.collect()

        // 某些可定义为抽象类型。
        abstract method buildStructures()
        abstract method buildUnits()

        // 一个类可包含多个模板方法。
        method attack() is
            enemy = closestEnemy()
            if (enemy == null)
                sendScouts(map.center)
            else
                sendWarriors(enemy.position)

        abstract method sendScouts(position)
        abstract method sendWarriors(position)

// 具体类必须实现基类中的所有抽象操作，但是它们不能重写模板方法自身。
class OrcsAI extends GameAI is
        method buildStructures() is
            if (there are some resources) then
                // 建造农场，接着是谷仓，然后是要塞。

        method buildUnits() is
            if (there are plenty of resources) then
                if (there are no scouts)
                    // 建造苦工，将其加入侦查编组。
                else
                    // 建造兽族步兵，将其加入战士编组。

        // ……

        method sendScouts(position) is
            if (scouts.length > 0) then
                // 将侦查编组送到指定位置。

        method sendWarriors(position) is
            if (warriors.length > 5) then
                // 将战斗编组送到指定位置。

// 子类可以重写部分默认的操作。
class MonstersAI extends GameAI is
        method collectResources() is
```

```
        // 怪物不会采集资源。

    method buildStructures() is
        // 怪物不会建造建筑。

    method buildUnits() is
        // 怪物不会建造单位。
```

6. 模板方法模式适合应用场景

当我们只希望客户端扩展某个特定算法步骤，而不是整个算法或其结构时，可使用模板方法模式。

模板方法将整个算法转换为一系列独立的步骤，以便子类能对其进行扩展，同时还可让超类中所定义的结构保持完整。

当多个类的算法除一些细微不同之外几乎完全一样时，我们可使用该模式。但其后果就是，只要算法发生变化，我们就可能需要修改所有的类。

在将算法转换为模板方法时，我们可将相似的实现步骤提取到超类中以去除重复代码。子类间各不同的代码可继续保留在子类中。

7. 模板方法模式优缺点

1) 优点

我们可仅允许客户端重写一个大型算法中的特定部分，使得算法其他部分修改对其所造成的影响减小。

我们可将重复代码提取到一个超类中。

2)缺点

部分客户端可能会受到算法框架的限制。

通过子类抑制默认步骤实现可能会导致违反里氏替换原则。

模板方法中的步骤越多，其维护工作就可能会越困难。

8. 与其他模式的关系

工厂方法模式是模板方法模式的一种特殊形式。同时，工厂方法可以作为一个大型模板方法中的一个步骤。

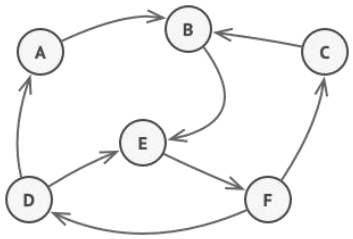
模板方法基于继承机制：它允许我们通过扩展子类中的部分内容来改变部分算法。策略模式基于组合机制：我们可以通过对相应行为提供不同的策略来改变对象的部分行为。 模板方法在类层次上运作，因此它是静态的。策略在对象层次上运作， 因此允许在运行时切换行为。

四. 状态模式——State

状态模式是一种行为设计模式，让我们能在一个对象的内部状态变化时改变其行为，使其看上去就像改变了自身所属的类一样。

1. 问题

状态模式与有限状态机的概念紧密相关。



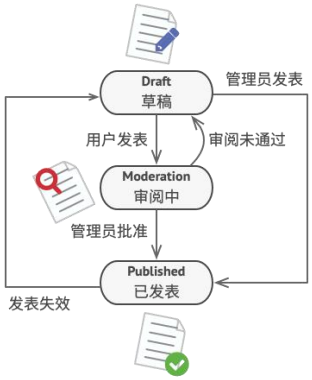
其主要思想是程序在任意时刻仅可处于几种有限的状态中。在任何一个特定状态中，程序的行为都不相同，且可瞬间从一个状态切换到另一个状态。不过，根据当前状态，程序可能会切换到另外一种状态，也可能会保持当前状态不变。这些数量有限且预先定义的状态切换规则被称为转移。

我们还可将该方法应用在对象上。假如我们有一个文档 Document 类。文档可能会处于草稿(Draft)、 审阅中(Moderation)和已发布(Publishe)三种状态中的一种。文档的 publish 发布方法在不同状态下的行为略有不同：

处于草稿状态时，它会将文档转移到审阅中状态。

处于审阅中状态时，如果当前用户是管理员，它会公开发布文档。

处于已发布状态时，它不会进行任何操作。



状态机通常由众多条件运算符(if 或 switch)实现，可根据对象的当前状态选择相应的行为。 “状态”通常只是对象中的一组成员变量值。即使之前从未听说过有限状态机，也很可能已经实现过状态模式。下面的代码应该能帮助回忆起来。

PS：

```
class Document is
    field state: string
    // ……
    method publish() is
        switch (state)
            "draft":
                state = "moderation"
                break
            "moderation":
                if (currentUser.role == "admin")
                    state = "published"
                break
            "published":
                // 什么也不做。
```

```
                break
        // ……
```

当我们逐步在文档类中添加更多状态和依赖于状态的行为后，基于条件语句的状态机就会暴露其最大的弱点。为了能根据当前状态选择完成相应行为的方法，绝大部分方法中会包含复杂的条件语句。修改其转换逻辑可能会涉及到修改所有方法中的状态条件语句， 导致代码的维护工作非常艰难。

这个问题会随着项目进行变得越发严重。我们很难在设计阶段预测到所有可能的状态和转换。随着时间推移，最初仅包含有限条件语句的简洁状态机可能会变成臃肿的一团乱麻。

2. 解决方案

状态模式建议为对象的所有可能状态新建一个类，然后将所有状态的对应行为抽取到这些类中。

原始对象被称为上下文（context），它并不会自行实现所有行为，而是会保存一个指向表示当前状态的状态对象的引用，且将所有与状态相关的工作委派给该对象。

如需将上下文转换为另外一种状态，则需将当前活动的状态对象替换为另外一个代表新状态的对象。采用这种方式是有前提的：所有状态类都必须遵循同样的接口，而且上下文必须仅通过接口与这些对象进行交互。

这个结构可能看上去与策略模式相似，但有一个关键性的不同——在状态模式中，特定状态知道其他所有状态的存在，且能触发从一个状态到另一个状态的转换；策略则几乎完全不知道其他策略的存在。

3. 真实世界类比

智能手机的按键和开关会根据设备当前状态完成不同行为：

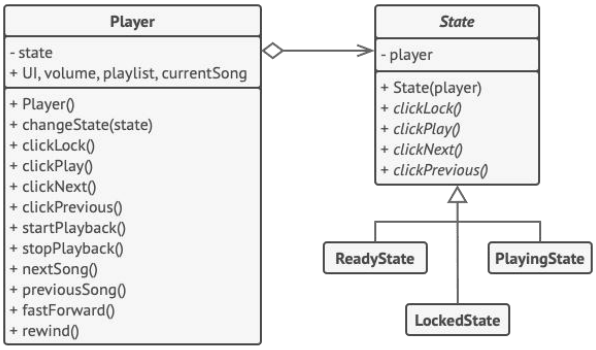
当手机处于解锁状态时，按下按键将执行各种功能。

当手机处于锁定状态时，按下任何按键都将解锁屏幕。

当手机电量不足时，按下任何按键都将显示充电页面。

4. 伪代码

在本例中，状态模式将根据当前回放状态，让媒体播放器中的相同控件完成不同的行为。



播放器的主要对象总是会连接到一个负责播放器绝大部分工作的状态对象中。部分操作会更换播放器当前的状态对象，以此改变播放器对于用户互动所作出的反应。

PS：

```
// 音频播放器（Audio- Player）类即为上下文。它还会维护指向状态类实例的引用，
// 该状态类则用于表示音频播放器当前的状态。
class AudioPlayer is
    field state: State
    field UI, volume, playlist, currentSong
```

```
    constructor AudioPlayer() is
        this.state = new ReadyState(this)

        // 上下文会将处理用户输入的工作委派给状态对象。由于每个状态都以不
        // 同的方式处理输入，其结果自然将依赖于当前所处的状态。
        UI = new UserInterface()
        UI.lockButton.onClick(this.clickLock)
        UI.playButton.onClick(this.clickPlay)
        UI.nextButton.onClick(this.clickNext)
        UI.prevButton.onClick(this.clickPrevious)

    // 其他对象必须能切换音频播放器当前所处的状态。
    method changeState(state: State) is
        this.state = state

    // UI 方法会将执行工作委派给当前状态。
    method clickLock() is
        state.clickLock()
    method clickPlay() is
        state.clickPlay()
    method clickNext() is
        state.clickNext()
    method clickPrevious() is
        state.clickPrevious()

    // 状态可调用上下文的一些服务方法。
    method startPlayback() is
        // ……
    method stopPlayback() is
        // ……
    method nextSong() is
        // ……
    method previousSong() is
        // ……
    method fastForward(time) is
        // ……
    method rewind(time) is
        // ……


// 所有具体状态类都必须实现状态基类声明的方法，并提供反向引用指向与状态相
// 关的上下文对象。状态可使用反向引用将上下文转换为另一个状态。
abstract class State is
    protected field player: AudioPlayer
```

```
    // 上下文将自身传递给状态构造函数。这可帮助状态在需要时获取一些有用的
    // 上下文数据。
    constructor State(player) is
        this.player = player

    abstract method clickLock()
    abstract method clickPlay()
    abstract method clickNext()
    abstract method clickPrevious()


// 具体状态会实现与上下文状态相关的多种行为。
class LockedState extends State is

    // 当你解锁一个锁定的播放器时，它可能处于两种状态之一。
    method clickLock() is
        if (player.playing)
            player.changeState(new PlayingState(player))
        else
            player.changeState(new ReadyState(player))

    method clickPlay() is
        // 已锁定，什么也不做。

    method clickNext() is
        // 已锁定，什么也不做。

    method clickPrevious() is
        // 已锁定，什么也不做。


// 它们还可在上下文中触发状态转换。
class ReadyState extends State is
    method clickLock() is
        player.changeState(new LockedState(player))

    method clickPlay() is
        player.startPlayback()
        player.changeState(new PlayingState(player))

    method clickNext() is
        player.nextSong()
```

```
        method clickPrevious() is
            player.previousSong()


class PlayingState extends State is
    method clickLock() is
        player.changeState(new LockedState(player))

    method clickPlay() is
        player.stopPlayback()
        player.changeState(new ReadyState(player))

    method clickNext() is
        if (event.doubleclick)
            player.nextSong()
        else
            player.fastForward(5)

    method clickPrevious() is
        if (event.doubleclick)
            player.previous()
        else
            player.rewind(5)
```

5. 状态模式优缺点

1) 优点

单一职责原则。将与特定状态相关的代码放在单独的类中。

开闭原则。无需修改已有状态类和上下文就能引入新状态。

通过消除臃肿的状态机条件语句简化上下文代码。

2)缺点

如果状态机只有很少的几个状态，或者很少发生改变，那么应用该模式可能会显得小题大作。

6. 与其他模式的关系

桥接模式、状态模式和策略模式（在某种程度上包括适配器模式）模式的接口非常相似。实际上，它们都基于组合模式——即将工作委派给其他对象，不过也各自解决了不同的问题。模式并不只是以特定方式组织代码的配方，我们还可以使用它们来和其他开发者讨论模式所解决的问题。

状态可被视为策略的扩展。两者都基于组合机制：它们都通过将部分工作委派给“帮手”对象来改变其在不同情景下的行为。策略使得这些对象相互之间完全独立，它们不知道其他对象的存在。但状态模式没有限制具体状态之间的依赖，且允许它们自行改变在不同情景下的状态。

五. 策略模式——Strategy

策略模式是一种行为设计模式，它能让我们定义一系列算法，并将每种算法分别放入独立的类中，以使算法的对象能够相互替换。

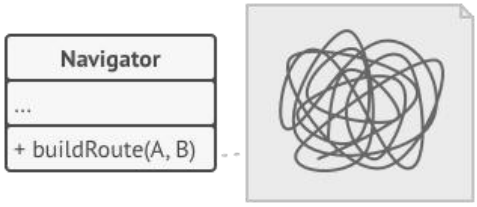
1. 问题

一天，我们打算为游客们创建一款导游程序。该程序的核心功能是提供美观的地图，以帮助用户在任何城市中快速定位。

用户期待的程序新功能是自动路线规划：他们希望输入地址后就能在地图上看到前往目的地的最快路线。

程序的首个版本只能规划公路路线。驾车旅行的人们对此非常满意。但很显然，并非所有人都会在度假时开车 因此在下次更新时添加了规划步行路线的功能。此后，又添加了规划公共交通路线的功能。

而这只是个开始。不久后，又要为骑行者规划路线。又过了一段时间，又要为游览城市中的所有景点规划路线。



尽管从商业角度来看，这款应用非常成功，但其技术部分却让你非常头疼：每次添加新的路线规划算法后，导游应用中主要类的体积就会增加一倍。终于在某个时候，觉得自己没法继续维护这堆代码了。

无论是修复简单缺陷还是微调街道权重，对某个算法进行任何修改都会影响整个类，从而增加在已有正常运行代码中引入错误的风险。

此外，团队合作将变得低效。如果在应用成功发布后招募了团队成员，他们会抱怨在合并冲突的工作上花费了太多时间。在实现新功能的过程中，我们的团队需要修改同一个巨大的类，这样他们所编写的代码相互之间就可能会出现冲突。

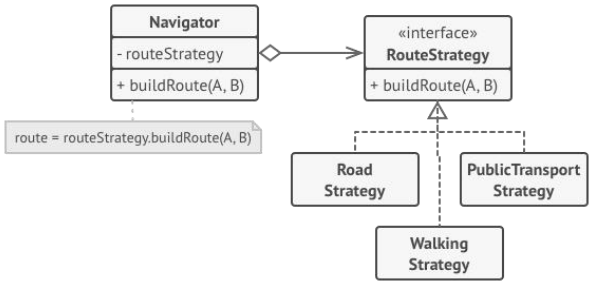
2. 解决方案

策略模式建议找出负责用许多不同方式完成特定任务的类，然后将其中的算法抽取到一组被称为策略的独立类中。

名为上下文的原始类必须包含一个成员变量来存储对于每种策略的引用。上下文并不执行任务，而是将工作委派给已连接的策略对象。

上下文不负责选择符合任务需要的算法——客户端会将所需策略传递给上下文。实际上，上下文并不十分了解策略，它会通过同样的通用接口与所有策略进行交互，而该接口只需暴露一个方法来触发所选策略中封装的算法即可。

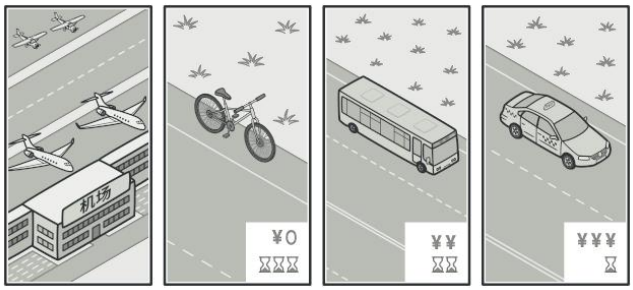
因此，上下文可独立于具体策略。这样我们就可在不修改上下文代码或其他策略的情况下添加新算法或修改已有算法了。



在导游应用中，每个路线规划算法都可被抽取到只有一个 build- Route 生成路线方法的独立类中。该方法接收起点和终点作为参数，并返回路线中途点的集合。

即使传递给每个路径规划类的参数一模一样，其所创建的路线也可能完全不同。主要导游类的主要工作是在地图上渲染一系列中途点，不会在意如何选择算法。该类中还有一个用于切换当前路径规划策略的方法，因此客户端（例如用户界面中的按钮）可用其他策略替换当前选择的路径规划行为。

3. 真实世界类比



假如需要前往机场。我们可以选择乘坐公共汽车、预约出租车或骑自行车。这些就是我们的出行策略。我们可以根据预算或时间等因素来选择其中一种策略。

4. 伪代码

PS：

```
// 策略接口声明了某个算法各个不同版本间所共有的操作。上下文会使用该接口来
// 调用有具体策略定义的算法。
interface Strategy is
    method execute(a, b)

// 具体策略会在遵循策略基础接口的情况下实现算法。该接口实现了它们在上下文
// 中的互换性。
class ConcreteStrategyAdd implements Strategy is
    method execute(a, b) is
        return a + b

class ConcreteStrategySubtract implements Strategy is
    method execute(a, b) is
        return a - b

class ConcreteStrategyMultiply implements Strategy is
    method execute(a, b) is
        return a * b

// 上下文定义了客户端关注的接口。
class Context is
    // 上下文会维护指向某个策略对象的引用。上下文不知晓策略的具体类。上下
    // 文必须通过策略接口来与所有策略进行交互。
    private strategy: Strategy

    // 上下文通常会通过构造函数来接收策略对象，同时还提供设置器以便在运行
    // 时切换策略。
```

```
        method setStrategy(Strategy strategy) is
            this.strategy = strategy

        // 上下文会将一些工作委派给策略对象，而不是自行实现不同版本的算法。
        method executeStrategy(int a, int b) is
            return strategy.execute(a, b)


// 客户端代码会选择具体策略并将其传递给上下文。客户端必须知晓策略之间的差
// 异，才能做出正确的选择。
class ExampleApplication is
        method main() is

            创建上下文对象。

            读取第一个数。
            读取最后一个数。
            从用户输入中读取期望进行的行为。

            if (action == addition) then
                context.setStrategy(new ConcreteStrategyAdd())

            if (action == subtraction) then
                context.setStrategy(new ConcreteStrategySubtract())

            if (action == multiplication) then
                context.setStrategy(new ConcreteStrategyMultiply())
            result = context.executeStrategy(First number, Second number)
            打印结果。
```

5. 策略模式适合应用场景

当我们想使用对象中各种不同的算法变体，并希望能在运行时切换算法时，可使用策略模式。

策略模式让我们能够将对象关联至可以不同方式执行特定子任务的不同子对象，从而以间接方式在运行时更改对象行为。

当我们有许多仅在执行某些行为时略有不同的相似类时，可使用策略模式。

策略模式让我们能将不同行为抽取到一个独立类层次结构中，并将原始类组合成同一个，从而减少重复代码。

如果算法在上下文的逻辑中不是特别重要，使用该模式能将类的业务逻辑与其算法实现细节隔离开来。

策略模式让我们能将各种算法的代码、内部数据和依赖关系与其他代码隔离开来。不同客户端可通过一个简单接口执行算法，并能在运行时进行切换。

当类中使用了复杂条件运算符以在同一算法的不同变体中切换时，可使用该模式。

策略模式将所有继承自同样接口的算法抽取到独立类中，因此不再需要条件语句。 原始对象并不实现所有算法的变体，而是将执行工作委派给其中的一个独立算法对象。

6. 策略模式优缺点

1) 优点

我们可以在运行时切换对象内的算法。

我们可以将算法的实现和使用算法的代码隔离开来。

我们可以使用组合来代替继承。

开闭原则。我们无需对上下文进行修改就能够引入新的策略。

2)缺点

如果我们的算法极少发生改变，那么没有任何理由引入新的类和接口。使用该模式只会让程序过于复杂。

客户端必须知晓策略间的不同——它需要选择合适的策略。

许多现代编程语言支持函数类型功能，允许我们在一组匿名函数中实现不同版本的算法。这样，我们使用这些函数的方式就和使用策略对象时完全相同，无需借助额外的类和接口来保持代码简洁。

7. 与其他模式的关系

桥接模式、状态模式和策略模式（在某种程度上包括适配器模式）模式的接口非常相似。实际上，它们都基于组合模式——即将工作委派给其他对象，不过也各自解决了不同的问题。模式并不只是以特定方式组织代码的配方，你还可以使用它们来和其他开发者讨论模式所解决的问题。

命令模式和策略看上去很像，因为两者都能通过某些行为来参数化对象。但是，它们的意图有非常大的不同。

你可以使用命令来将任何操作转换为对象。操作的参数将成为对象的成员变量。你可以通过转换来延迟操作的执行、将操作放入队列、保存历史命令或者向远程服务发送命令等。

另一方面，策略通常可用于描述完成某件事的不同方式，让你能够在同一个上下文类中切换算法。

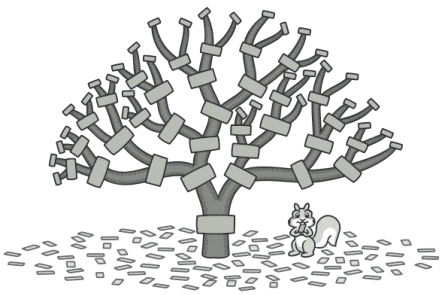
装饰模式可让你更改对象的外表，策略则让你能够改变其本质。

模板方法模式基于继承机制：它允许你通过扩展子类中的部分内容来改变部分算法。策略基于组合机制：你可以通过对相应行为提供不同的策略来改变对象的部分行为。模板方法在类层次上运作，因此它是静态的。策略在对象层次上运作，因此允许在运行时切换行为。

状态可被视为策略的扩展。两者都基于组合机制：它们都通过将部分工作委派给“帮手”对象来改变其在不同情景下的行为。策略使得这些对象相互之间完全独立，它们不知道其他对象的存在。但状态模式没有限制具体状态之间的依赖，且允许它们自行改变在不同情景下的状态。

六. 组合模式

组合模式是一种结构型设计模式，我们可以使用它将对象组合成树状结构，且能像使用独立对象一样使用它们。

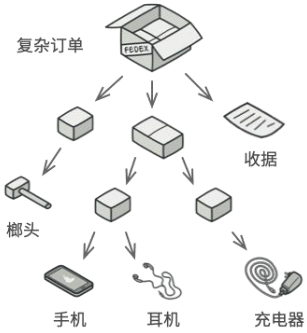


1. 问题

如果应用的核心模型能用树状结构表示，在应用中使用组合模式才有价值。

例如，有两类对象：产品和盒子。一个盒子中可以包含多个产品或者几个较小的盒子。这些小盒子中同样可以包含一些产品或更小的盒子，以此类推。

假设我们希望在这些类的基础上开发一个定购系统，订单中可以包含无包装的简单产品，也可以包含装满产品的盒子……以及其他盒子。此时我们要如何计算每张订单的总价格呢？



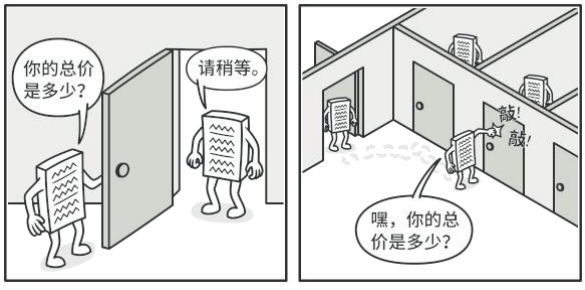
订单中可能包括各种产品，这些产品放置在盒子中，然后又被放入一层又一层更大的盒子中。整个结构看上去像是一棵倒过来的树。

我们可以尝试直接计算：打开所有盒子，找到每件产品，然后计算总价。这在真实世界中或许可行，但在程序中，并不能简单地使用循环语句来完成该工作。我们必须事先知道所有产品和盒子的类别，所有盒子的嵌套层数以及其他繁杂的细节信息。因此，直接计算极不方便，甚至完全不可行。

2. 解决方案

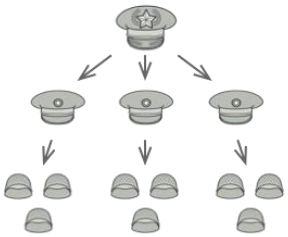
组合模式建议使用一个通用接口来与产品和盒子进行交互，并且在该接口中声明一个计算总价的方法。

那么方法该如何设计呢？对于一个产品，该方法直接返回其价格；对于一个盒子，该方法遍历盒子中的所有项目，询问每个项目的价格，然后返回该盒子的总价格。如果其中某个项目是小一号的盒子，那么当前盒子也会遍历其中的所有项目，以此类推，直到计算出所有内部组成部分的价格。我们甚至可以在盒子的最终价格中增加额外费用，作为该盒子的包装费用。



该方式的最大优点在于我们无需了解构成树状结构的对象的具体类，也无需了解对象是简单的产品还是复杂的盒子。只需调用通用接口以相同的方式对其进行处理即可。当我们调用该方法后，对象会将请求沿着树结构传递下去。

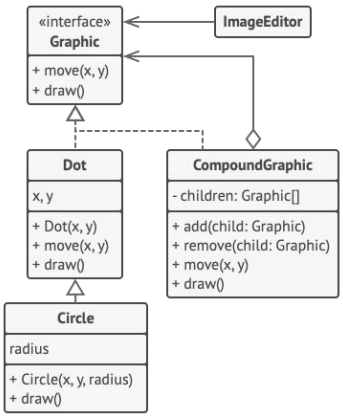
3. 真实世界类比



大部分国家的军队都采用层次结构管理。每支部队包括几个师，师由旅构成，旅由团构成，团可以继续划分为排。最后，每个排由一小队实实在在的士兵组成。军事命令由最高层下达，通过每个层级传递，直到每位士兵都知道自己应该服从的命令。

4. 伪代码

在本例中，我们将借助组合模式帮助你在图形编辑器中实现一系列的几何图形。



组合图形 Compound- Graphic 是一个容器，它可以由多个包括容器在内的子图形构成。组合图形与简单图形拥有相同的方法。但是，组合图形自身并不完成具体工作，而是将请求递归地传递给自己的子项目，然后“汇总”结果。

通过所有图形类所共有的接口，客户端代码可以与所有图形互动。因此，客户端不知道与其交互的是简单图形还是组合图形。客户端可以与非常复杂的对象结构进行交互， 而无需与组成该结构的实体类紧密耦合。

PS：

```
// 组件接口会声明组合中简单和复杂对象的通用操作。
interface Graphic is
    method move(x, y)
    method draw()

// 叶节点类代表组合的终端对象。叶节点对象中不能包含任何子对象。叶节点对象
// 通常会完成实际的工作，组合对象则仅会将工作委派给自己的子部件。
class Dot implements Graphic is
    field x, y

    constructor Dot(x, y) { …… }

    method move(x, y) is
        this.x += x, this.y += y
```

```
        method draw() is
            // 在坐标位置(X,Y)处绘制一个点。

// 所有组件类都可以扩展其他组件。
class Circle extends Dot is
        field radius

        constructor Circle(x, y, radius) { …… }

        method draw() is
            // 在坐标位置(X,Y)处绘制一个半径为 R 的圆。

// 组合类表示可能包含子项目的复杂组件。组合对象通常会将实际工作委派给子项
// 目，然后“汇总”结果。
class CompoundGraphic implements Graphic is
        field children: array of Graphic

        // 组合对象可在其项目列表中添加或移除其他组件（简单的或复杂的皆可）。
        method add(child: Graphic) is
            // 在子项目数组中添加一个子项目。

        method remove(child: Graphic) is
            // 从子项目数组中移除一个子项目。

        method move(x, y) is
            foreach (child in children) do
                child.move(x, y)

        // 组合会以特定的方式执行其主要逻辑。它会递归遍历所有子项目，并收集和
        // 汇总其结果。由于组合的子项目也会将调用传递给自己的子项目，以此类推，
        // 最后组合将会完成整个对象树的遍历工作。
        method draw() is
            // 1. 对于每个子部件：
            //       - 绘制该部件。
            //       - 更新边框坐标。
            // 2. 根据边框坐标绘制一个虚线长方形。


// 客户端代码会通过基础接口与所有组件进行交互。这样一来，客户端代码便可同
// 时支持简单叶节点组件和复杂组件。
class ImageEditor is
        field all: CompoundGraphic
```

```
method load() is
    all = new CompoundGraphic()
    all.add(new Dot(1, 2))
    all.add(new Circle(5, 3, 10))
    // ……

// 将所需组件组合为复杂的组合组件。
method groupSelected(components: array of Graphic) is
    group = new CompoundGraphic()
    foreach (component in components) do
        group.add(component)
        all.remove(component)
    all.add(group)
    // 所有组件都将被绘制。
    all.draw()
```

5. 组合模式适合应用场景

如果我们需要实现树状对象结构，可以使用组合模式。

组合模式为我们提供了两种共享公共接口的基本元素类型：简单叶节点和复杂容器。容器中可以包含叶节点和其他容器。这使得我们可以构建树状嵌套递归对象结构。

如果我们希望客户端代码以相同方式处理简单和复杂元素，可以使用该模式。

组合模式中定义的所有元素共用同一个接口。在这一接口的帮助下，客户端不必在意其所使用的对象的具体类。

6. 组合模式优缺点

1) 优点

我们可以利用多态和递归机制更方便地使用复杂树结构。

开闭原则。无需更改现有代码，我们就可以在应用中添加新元素，使其成为对象树的一部分。

2)缺点

对于功能差异较大的类，提供公共接口或许会有困难。在特定情况下，我们需要过度一般化组件接口，使其变得令人难以理解。

7. 与其他模式的关系

桥接模式、状态模式和策略模式（在某种程度上包括适配器模式）模式的接口非常相似。实际上，它们都基于组合模式——即将工作委派给其他对象，不过也各自解决了不同的问题。模式并不只是以特定方式组织代码的配方，我们还可以使用它们来和其他开发者讨论模式所解决的问题。

我们可以在创建复杂组合树时使用生成器模式，因为这可使其构造步骤以递归的方式运行。

责任链模式通常和组合模式结合使用。在这种情况下，叶组件接收到请求后，可以将请求沿包含全体父组件的链一直传递至对象树的底部。

可以使用迭代器模式来遍历组合树。

可以使用访问者模式对整个组合树执行操作。

可以使用享元模式实现组合树的共享叶节点以节省内存。

组合和装饰模式的结构图很相似，因为两者都依赖递归组合来组织无限数量的对象。

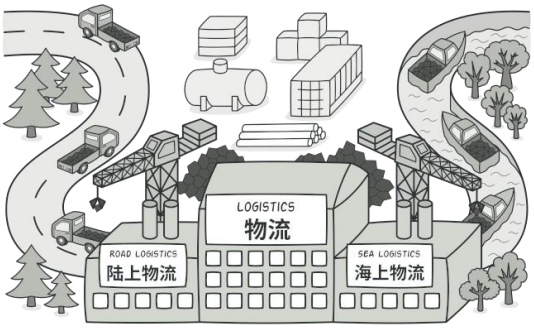
装饰类似于组合，但其只有一个子组件。此外还有一个明显不同：装饰为被封装对象添加了额外的职责，组合仅对其子节点的结果进行了“求和”。

但是，模式也可以相互合作：我们可以使用装饰来扩展组合树中特定对象的行为。

大量使用组合和装饰的设计通常可从对于原型模式的使用中获益。我们可以通过该模式来复制复杂结构，而非从零开始重新构造。

七. 工厂模式

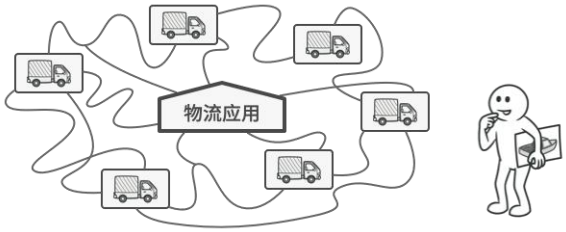
工厂方法模式是一种创建型设计模式，其在父类中提供一个创建对象的方法，允许子类决定实例化对象的类型。



1. 问题

假设我们正在开发一款物流管理应用。最初版本只能处理卡车运输，因此大部分代码都在位于名为卡车的类中。

一段时间后，这款应用变得极受欢迎。我们每天都能收到十几次来自海运公司的请求，希望应用能够支持海上物流功能。

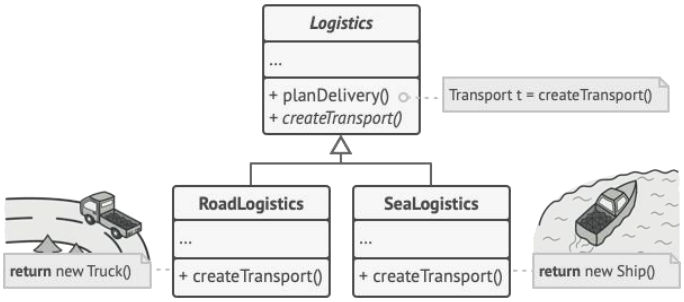


这可是个好消息。但是代码问题该如何处理呢？目前，大部分代码都与卡车类相关。 在程序中添加轮船类需要修改全部代码。更糟糕的是，如果我们以后需要在程序中支持另外一种运输方式，很可能需要再次对这些代码进行大幅修改。

最后，我们将不得不编写繁复的代码，根据不同的运输对象类，在应用中进行不同的处理。

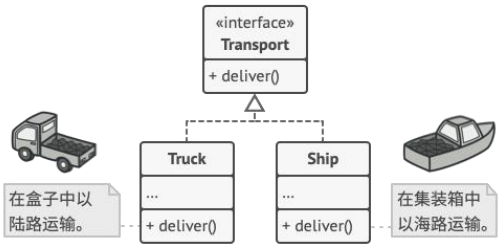
2. 解决方案

工厂方法模式建议使用特殊的工厂方法代替对于对象构造函数的直接调用（即使用 new 运算符）。不用担心，对象仍将通过 new 运算符创建，只是该运算符改在工厂方法中调用罢了。工厂方法返回的对象通常被称作“产品”。

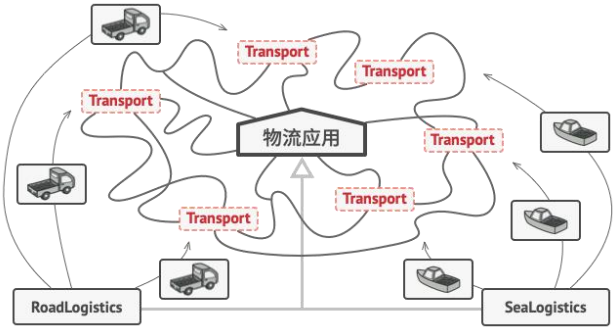


乍看之下，这种更改可能毫无意义：我们只是改变了程序中调用构造函数的位置而已。但是，仔细想一下，现在我们可以在子类中重写工厂方法，从而改变其创建产品的类型。

但有一点需要注意：仅当这些产品具有共同的基类或者接口时，子类才能返回不同类型的产品，同时基类中的工厂方法还应将其返回类型声明为这一共有接口。



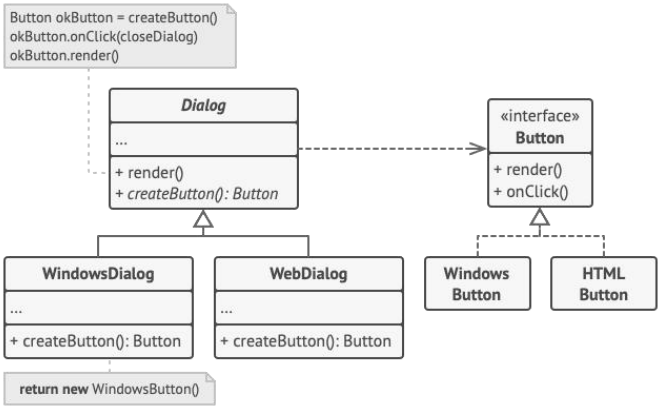
举例来说，卡车 Truck 和轮船 Ship 类都必须实现运输 Transport 接口，该接口声明了一个名为 deliver 交付的方法。每个类都将以不同的方式实现该方法：卡车走陆路交付货物，轮船走海路交付货物。陆路运输 Road-Logistics 类中的工厂方法返回卡车对象，而海路运输 Sea-Logistics 类则返回轮船对象。



调用工厂方法的代码（通常被称为客户端代码）无需了解不同子类返回实际对象之间的差别。客户端将所有产品视为抽象的运输。客户端知道所有运输对象都提供交付方法， 但是并不关心其具体实现方式。

3. 伪代码

以下示例演示了如何使用工厂方法开发跨平台 UI（用户界面）组件，并同时避免客户代码与具体 UI 类之间的耦合。



基础对话框类使用不同的 UI 组件渲染窗口。在不同的操作系统下，这些组件外观或许略有不同，但其功能保持一致。Windows 系统中的按钮在 Linux 系统中仍然是按钮。

如果使用工厂方法，就不需要为每种操作系统重写对话框逻辑。如果我们声明了一个在

基本对话框类中生成按钮的工厂方法，那么我们就可以创建一个对话框子类，并使其通过工厂方法返回 Windows 样式按钮。子类将继承对话框基础类的大部分代码，同时在屏幕上根据 Windows 样式渲染按钮。

如需该模式正常工作，基础对话框类必须使用抽象按钮（例如基类或接口），以便将其扩展为具体按钮。这样一来，无论对话框中使用何种类型的按钮，其代码都可以正常工作。

我们可以使用此方法开发其他 UI 组件。不过，每向对话框中添加一个新的工厂方法，我们就离抽象工厂模式更近一步。我们将在稍后谈到这个模式。

PS：

```
// 创建者类声明的工厂方法必须返回一个产品类的对象。创建者的子类通常会提供
// 该方法的实现。
class Dialog is
    // 创建者还可提供一些工厂方法的默认实现。
    abstract method createButton():Button

    // 请注意，创建者的主要职责并非是创建产品。其中通常会包含一些核心业务
    // 逻辑，这些逻辑依赖于由工厂方法返回的产品对象。子类可通过重写工厂方
    // 法并使其返回不同类型的产品来间接修改业务逻辑。
    method render() is
        // 调用工厂方法创建一个产品对象。
        Button okButton = createButton()
        // 现在使用产品。
        okButton.onClick(closeDialog)
        okButton.render()


// 具体创建者将重写工厂方法以改变其所返回的产品类型。
class WindowsDialog extends Dialog is
    method createButton():Button is
        return new WindowsButton()

class WebDialog extends Dialog is
    method createButton():Button is
        return new HTMLButton()


// 产品接口中将声明所有具体产品都必须实现的操作。
interface Button is
    method render()
    method onClick(f)

// 具体产品需提供产品接口的各种实现。
class WindowsButton implements Button is
    method render(a, b) is
        // 根据 Windows 样式渲染按钮。
```

```
        method onClick(f) is
            // 绑定本地操作系统点击事件。

    class HTMLButton implements Button is
        method render(a, b) is
            // 返回一个按钮的 HTML 表述。
        method onClick(f) is
            // 绑定网络浏览器的点击事件。


    class Application is
        field dialog: Dialog

        // 程序根据当前配置或环境设定选择创建者的类型。
        method initialize() is
            config = readApplicationConfigFile()

            if (config.OS == "Windows") then
                dialog = new WindowsDialog()
            else if (config.OS == "Web") then
                dialog = new WebDialog()
            else
                throw new Exception("错误！未知的操作系统。")

        // 当前客户端代码会与具体创建者的实例进行交互，但是必须通过其基本接口
        // 进行。只要客户端通过基本接口与创建者进行交互，你就可将任何创建者子
        // 类传递给客户端。
        method main() is
            this.initialize()
            dialog.render()
```

4. 工厂方法模式适合应用场景

当我们在编写代码的过程中，如果无法预知对象确切类别及其依赖关系时，可使用工厂方法。

工厂方法将创建产品的代码与实际使用产品的代码分离，从而能在不影响其他代码的情况下扩展产品创建部分代码。

例如，如果需要向应用中添加一种新产品，我们只需要开发新的创建者子类，然后重写其工厂方法即可。

如果我们希望用户能扩展我们软件库或框架的内部组件，可使用工厂方法。

继承可能是扩展软件库或框架默认行为的最简单方法。但是当我们使用子类替代标准组件时，框架如何辨识出该子类？

解决方案是将各框架中构造组件的代码集中到单个工厂方法中，并在继承该组件之外允许任何人对该方法进行重写。

让我们看看具体是如何实现的。假设我们使用开源 UI 框架编写自己的应用。我们希望在应用中使用圆形按钮，但是原框架仅支持矩形按钮。我们可以使用圆形按钮 Round-

Button 子类来继承标准的按钮 Button 类。但是，你需要告诉 UI 框架 UIFramework 类使用新的子类按钮代替默认按钮。

为了实现这个功能，我们可以根据基础框架类开发子类圆形按钮 UIUIWith-Round-Buttons，并且重写其 create-Button 创建按钮方法。基类中的该方法返回按钮对象，而我们开发的子类返回圆形按钮对象。现在，就可以使用圆形按钮 UI 类代替 UI 框架类。就是这么简单！

如果希望复用现有对象来节省系统资源，而不是每次都重新创建对象，可使用工厂方法。

在处理大型资源密集型对象（比如数据库连接、文件系统和网络资源）时，会经常碰到这种资源需求。

让我们思考复用现有对象的方法：

首先，需要创建存储空间来存放所有已经创建的对象。

当他人请求一个对象时，程序将在对象池中搜索可用对象。

… 然后将其返回给客户端代码。

如果没有可用对象，程序则创建一个新对象（并将其添加到对象池中）。

这些代码可不少！而且它们必须位于同一处，这样才能确保重复代码不会污染程序。

可能最显而易见，也是最方便的方式，就是将这些代码放置在我们试图重用的对象类的构造函数中。但是从定义上来讲，构造函数始终返回的是新对象，其无法返回现有实例。

因此，需要有一个既能够创建新对象，又可以重用现有对象的普通方法。这听上去和工厂方法非常相像。

5. 工厂方法模式优缺点

1) 优点

可以避免创建者和具体产品之间的紧密耦合。

单一职责原则。可以将产品创建代码放在程序的单一位置，从而使得代码更容易维护。

开闭原则。无需更改现有客户端代码，你就可以在程序中引入新的产品类型。

2)缺点

应用工厂方法模式需要引入许多新的子类，代码可能会因此变得更复杂。最好的情况是将该模式引入创建者类的现有层次结构中。

6. 与其他模式的关系

在许多设计工作的初期都会使用工厂方法模式（较为简单，而且可以更方便地通过子类进行定制），随后演化为使用抽象工厂模式、原型模式或生成器模式（更灵活但更加复杂）。

抽象工厂模式通常基于一组工厂方法，但我们也可以使用原型模式来生成这些类的方法。

我们可以同时使用工厂方法和迭代器模式来让子类集合返回不同类型的迭代器，并使得迭代器与集合相匹配。

原型并不基于继承，因此没有继承的缺点。另一方面，原型需要对被复制对象进行复杂的初始化。工厂方法基于继承，但是它不需要初始化步骤。

工厂方法是模板方法模式的一种特殊形式。同时，工厂方法可以作为一个大型模板方法中的一个步骤。

7. 代码

1) C#

PS：

```
using system;
```

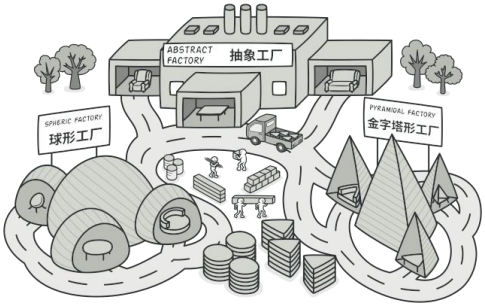
```
namespace refactoringguru.designpatterns.factorymethod.conceptual
{
    //creator 类声明了应该返回的工厂方法
    //product 类的对象创建者的子类通常提供这个方法的实现
    abstract class creator
    {
        //注意，创建者也可以提供一些默认的实现。
        //工厂方法
        public abstract iproduct factorymethod();

        //另外请注意，尽管它的名字是创造者，但主要责任不是创造产品。
        //通常，它包含一些依赖于产品对象的核心业务逻辑。子
        //类可以间接地改变业务逻辑，通过覆盖工厂方法并返回一个不同类型的产品
        public string someoperation()
        {
            // call the factory method to create a product object.
            var product = factorymethod();
            // now, use the product.
            var result = "creator: the same creator's code has just worked with "
                + product.operation();

            return result;
        }
    }

    //具体的创建者覆盖工厂方法以更改结果产品的类型。（子类重写）
    class concretecreator1 : creator
    {
        // 注意，方法的签名仍然使用抽象产品类型，即使从方法实际返回的是具体
的产品。这样，创建者就可以独立于具体的产品类。
        public override iproduct factorymethod()
        {
            return new concreteproduct1();
        }
    }

    class concretecreator2 : creator
    {
        public override iproduct factorymethod()
        {
            return new concreteproduct2();
        }
    }
```

```
//product 接口声明了所有具体产品必须实现的操作。
public interface iproduct
{
    string operation();
}

//具体产品提供了产品接口的各种实现。
class concreteproduct1 : iproduct
{
    public string operation()
    {
        return "{result of concreteproduct1}";
    }
}

class concreteproduct2 : iproduct
{
    public string operation()
    {
        return "{result of concreteproduct2}";
    }
}

class client
{
    public void main()
    {
        console.writeline("app: launched with the concretecreator1.");
        clientcode(new concretecreator1());

        console.writeline("");

        console.writeline("app: launched with the concretecreator2.");
        clientcode(new concretecreator2());
    }

    // 客户端代码与具体创建者的实例一起工作，尽管是通过其基本接口。
    // 只要客户端继续通过基接口与创建器一起工作，您就可以向它传递任何创建器的子类。
    public void clientcode(creator creator)
    {
        // ...
        console.writeline("client: i'm not aware of the creator's class," +
            "but it still works.\n" + creator.someoperation());
```

```
                // ...
            }
        }

        class program
        {
            static void main(string[] args)
            {
                new client().main();
            }
        }
    }
```

八. 抽象工厂模式

抽象工厂模式是一种创建型设计模式，它能创建一系列相关的对象，而无需指定其具体类。



1. 问题

假设正在开发一款家具商店模拟器。我们的代码中包括一些类，用于表示：

一系列相关产品，例如椅子 Chair 、沙发 Sofa 和咖啡桌 Coffee-Table。

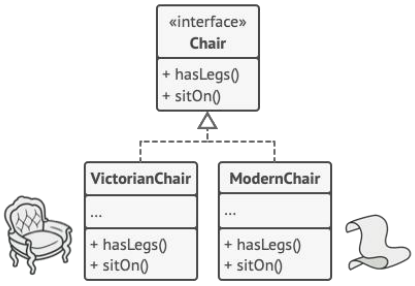
系列产品的不同变体。例如，我们可以使用现代 Modern 、维多利亚 Victorian 、装饰风艺术 Art-Deco 等风格生成椅子、沙发和咖啡桌。

我们需要设法单独生成每件家具对象，这样才能确保其风格一致。如果顾客收到的家具风格不一样，他们可不会开心。

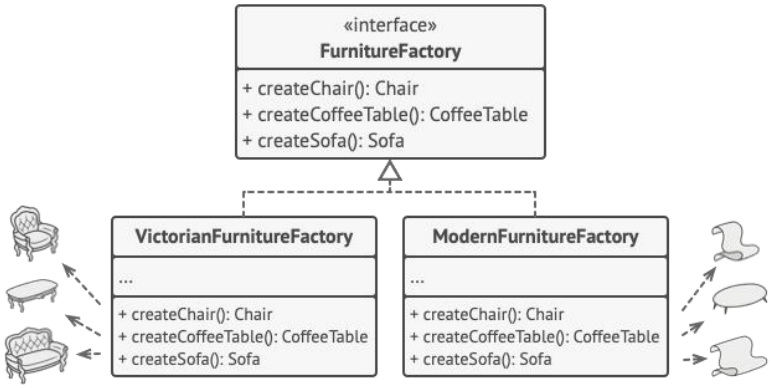
此外，我们也不希望在添加新产品或新风格时修改已有代码。家具供应商对于产品目录的更新非常频繁，我们不会想在每次更新时都去修改核心代码的。

2. 解决方案

首先，抽象工厂模式建议为系列中的每件产品明确声明接口（例如椅子、沙发或咖啡桌）。然后，确保所有产品变体都继承这些接口。例如，所有风格的椅子都实现椅子接口； 所有风格的咖啡桌都实现咖啡桌接口，以此类推。

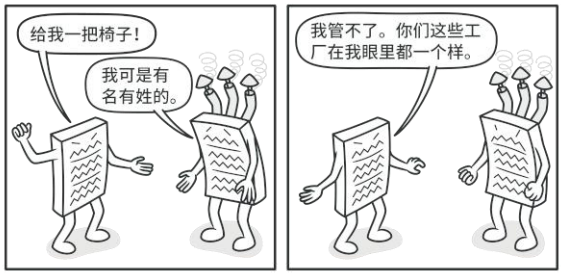


接下来，我们需要声明抽象工厂——包含系列中所有产品构造方法的接口。例如 create-Chair 创建椅子、create-Sofa 创建沙发和 create-Coffee-Table 创建咖啡桌。这些方法必须返回抽象产品类型，即我们之前抽取的那些接口：椅子，沙发和咖啡桌等等。



那么该如何处理产品变体呢？对于系列产品的每个变体，我们都将基于抽象工厂接口创建不同的工厂类。每个工厂类都只能返回特定类别的产品，例如，现代家具工厂 Modern-Furniture-Factory 只能创建现代椅子 Modern-Chair、现代沙发 Modern-Sofa 和现代咖啡桌 Modern-Coffee-Table 对象。

客户端代码可以通过相应的抽象接口调用工厂和产品类。我们无需修改实际客户端代码，就能更改传递给客户端的工厂类，也能更改客户端代码接收的产品变体。

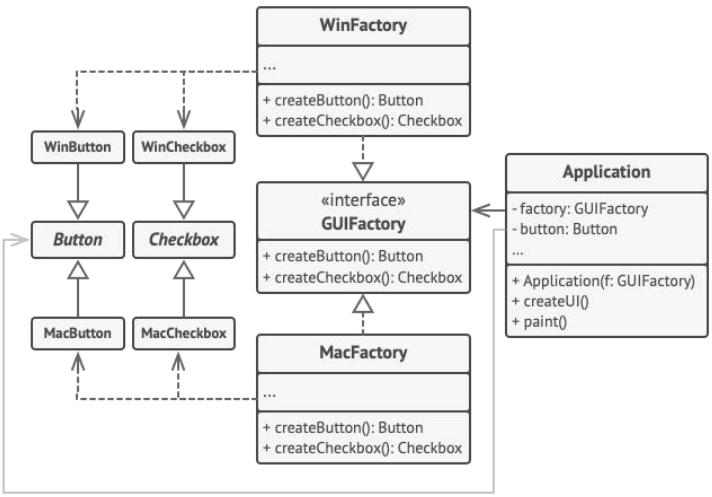


假设客户端想要工厂创建一把椅子。客户端无需了解工厂类，也不用管工厂类创建出的椅子类型。无论是现代风格，还是维多利亚风格的椅子，对于客户端来说没有分别，它只需调用抽象椅子接口就可以了。这样一来，客户端只需知道椅子以某种方式实现了 sit-On 坐下方法就足够了。此外，无论工厂返回的是何种椅子变体，它都会和由同一工厂对象创建的沙发或咖啡桌风格一致。

最后一点说明：如果客户端仅接触抽象接口，那么谁来创建实际的工厂对象呢？一般情况下，应用程序会在初始化阶段创建具体工厂对象。而在此之前，应用程序必须根据配置文件或环境设定选择工厂类别。

3. 伪代码

下面例子通过应用抽象工厂模式，使得客户端代码无需与具体 UI 类耦合，就能创建跨平台的 UI 元素，同时确保所创建的元素与指定的操作系统匹配。



跨平台应用中的相同 UI 元素功能类似，但是在不同操作系统下的外观有一定差异。 此外，我们需要确保 UI 元素与当前操作系统风格一致。我们一定不希望在 Windows 系统下运行的应用程序中显示 macOS 的控件。

抽象工厂接口声明一系列构建方法，客户端代码可调用它们生成不同风格的 UI 元素。每个具体工厂对应特定操作系统，并负责生成符合该操作系统风格的 UI 元素。

其运作方式如下：应用程序启动后检测当前操作系统。根据该信息，应用程序通过与该操作系统对应的类创建工厂对象。其余代码使用该工厂对象创建 UI 元素。这样可以避免生成错误类型的元素。

使用这种方法，客户端代码只需调用抽象接口，而无需了解具体工厂类和 UI 元素。 此外，客户端代码还支持未来添加新的工厂或 UI 元素。

这样一来，每次在应用程序中添加新的 UI 元素变体时，我们都无需修改客户端代码。只需创建一个能够生成这些 UI 元素的工厂类，然后稍微修改应用程序的初始代码，使其能够选择合适的工厂类即可。

PS：

```
// 抽象工厂接口声明了一组能返回不同抽象产品的方法。这些产品属于同一个系列
// 且在高层主题或概念上具有相关性。同系列的产品通常能相互搭配使用。系列产
// 品可有多个变体，但不同变体的产品不能搭配使用。
interface GUIFactory is
    method createButton():Button
    method createCheckbox():Checkbox


// 具体工厂可生成属于同一变体的系列产品。工厂会确保其创建的产品能相互搭配
// 使用。具体工厂方法签名会返回一个抽象产品，但在方法内部则会对具体产品进
// 行实例化。
```

```
class WinFactory implements GUIFactory is
    method createButton():Button is
        return new WinButton()
    method createCheckbox():Checkbox is
        return new WinCheckbox()

// 每个具体工厂中都会包含一个相应的产品变体。
class MacFactory implements GUIFactory is
    method createButton():Button is
        return new MacButton()
    method createCheckbox():Checkbox is
        return new MacCheckbox()


// 系列产品中的特定产品必须有一个基础接口。所有产品变体都必须实现这个接口。
interface Button is
    method paint()

// 具体产品由相应的具体工厂创建。
class WinButton implements Button is
    method paint() is
        // 根据 Windows 样式渲染按钮。

class MacButton implements Button is
    method paint() is
        // 根据 macOS 样式渲染按钮

// 这是另一个产品的基础接口。所有产品都可以互动，但是只有相同具体变体的产
// 品之间才能够正确地进行交互。
interface Checkbox is
    method paint()

class WinCheckbox implements Checkbox is
    method paint() is
        // 根据 Windows 样式渲染复选框。

class MacCheckbox implements Checkbox is
    method paint() is
        // 根据 macOS 样式渲染复选框。

// 客户端代码仅通过抽象类型（GUIFactory、Button 和 Checkbox）使用工厂
// 和产品。这让你无需修改任何工厂或产品子类就能将其传递给客户端代码。
class Application is
    private field factory: GUIFactory
```

```
        private field button: Button
        constructor Application(factory: GUIFactory) is
            this.factory = factory
        method createUI() is
            this.button = factory.createButton()
        method paint() is
            button.paint()


// 程序会根据当前配置或环境设定选择工厂类型，并在运行时创建工厂（通常在初
// 始化阶段）。
class ApplicationConfigurator is
    method main() is
        config = readApplicationConfigFile()

        if (config.OS == "Windows") then
            factory = new WinFactory()
        else if (config.OS == "Mac") then
            factory = new MacFactory()
        else
            throw new Exception("错误！未知的操作系统。")

        Application app = new Application(factory)
```

4. 抽象工厂模式适合应用场景

如果代码需要与多个不同系列的相关产品交互，但是由于无法提前获取相关信息， 或者出于对未来扩展性的考虑，不希望代码基于产品的具体类进行构建，在这种情况下，我们可以使用抽象工厂。

抽象工厂为我们提供了一个接口，可用于创建每个系列产品的对象。只要代码通过该接口创建对象，那么就不会生成与应用程序已生成的产品类型不一致的产品。

如果我们有一个基于一组抽象方法的类，且其主要功能因此变得不明确，那么在这种情况下可以考虑使用抽象工厂模式。

在设计良好的程序中，每个类仅负责一件事。如果一个类与多种类型产品交互，就可以考虑将工厂方法抽取到独立的工厂类或具备完整功能的抽象工厂类中。

5. 抽象工厂模式优缺点

1) 优点

可以确保同一工厂生成的产品相互匹配。

可以避免客户端和具体产品代码的耦合。

单一职责原则。可以将产品生成代码抽取到同一位置，使得代码易于维护。

开闭原则。向应用程序中引入新产品变体时，无需修改客户端代码。

2)缺点

由于采用该模式需要向应用中引入众多接口和类，代码可能会比之前更加复杂。

6. 与其他模式的关系

在许多设计工作的初期都会使用工厂方法模式（较为简单，而且可以更方便地通过子类

进行定制），随后演化为使用抽象工厂模式、原型模式或生成器模式（更灵活但更加复杂）。

生成器重点关注如何分步生成复杂对象。抽象工厂专门用于生产一系列相关对象。抽象工厂会马上返回产品，生成器则允许你在获取产品前执行一些额外构造步骤。

抽象工厂模式通常基于一组工厂方法，但我们也可以使用原型模式来生成这些类的方法。

当只需对客户端代码隐藏子系统创建对象的方式时，我们可以使用抽象工厂来代替外观模式。

可以将抽象工厂和桥接模式搭配使用。如果由桥接定义的抽象只能与特定实现合作，这一模式搭配就非常有用。在这种情况下，抽象工厂可以对这些关系进行封装，并且对客户端代码隐藏其复杂性。

抽象工厂、生成器和原型都可以用单例模式来实现。

九. 懒汉模式

1. 解决初始化与脚本加载顺序的矛盾

十. UI 框架

1. MVC

M：Model，UI 的数据以及对于 UI 数据的操作函数。数据部分可以理解为 UI 组件的具体渲染数据。以 UGUI 为例，一个 UI 上挂载了一个 Text 组件，一个 Button 组件，那么 Model 里可能就会有一个 string 属性代表 Text 显示的内容，一个 Action 属性表示 Button 被点击时会调用的回调函数。逻辑部分就是我们在游戏逻辑中会对 UI 数据进行的操作，可能还会有一些复杂的变换操作，比如一个人物最大生命值是 100，并且血条本身只显示血量百分比，那么这个 Model 里的 SetHPBarValue()函数就会执行下面的操作，把传进来的当前人物生命值（这里假设是 50）换算到一个百分比的值，即：当前人物生命值/人物最大生命值 = 0.5。然后会被 Controller 调用，继而把这个值给 View，让它显示 50%。

V：View，UI 的渲染组件，以 UGUI 为例，一个 UI 上挂载了一个 Text，一个 Button，那么这两个 UI 组件都将写到 View 里。也会包含一些操作，例如获取一个 UI 组件，对一个 UI 组件进行操作。

C：Controller，本身 Model 和 View 他们是互不耦合的，所以需要一个中介者把他们联系起来，从而达到数据更新和渲染状态更新同步。它本身是没什么代码量的，代码几乎都在 Model 和 View 中。

然后就是很多人说 MVC 会有很多冗余代码，写起来不方便，应该是他们的使用方法有问题，可能并没有搞清楚 M、V、C 的分工，可能把 M 的工作写到了 C 里，或者把 C 里的工作写到了 V 里，总之概念就搞的一团糟，更不用说写出的代码如何了。

2. MVVM

上面我已经提到了 MVVM，下面来详细介绍一下 MVVM 到底是个什么东西。

M：Model，还是和 MVC 的 M 差不多。

V：View，还是和 MVC 的 V 差不多。

VM：双向耦合（可能这里叫双向绑定更加贴切一点）M 和 V，什么是绑定呢，这里用观察者模式来解释可能更好懂一些。以 M 绑定 V 为例，M 的更新会执行一个委托，而我们此时已经将 V 加入到委托链里了，所以 M 的更新会执行我们指定的一个委托函数，从而达到 V 也更新的目的。V 绑定 M 也是同理。最后，M 更新会导致 V 更新，V 更新也会导致 M 相关数据更新的效果（避免循环应该加一个判重机制）。

我感觉 MVVM 和 MVC 并没有本质上的区别，他只是更加强制制定了更加明显的规范，所以写起来感觉 MVVM 更加厉害一些。

架构

一. OOP 思想

1. 什么是 OO

OO(Object–Oriented)面向对象，OO 方法(Object-Oriented Method，面向对象方法，面向对象的方法)是一种把面向对象的思想应用于软件开发过程中，指导开发活动的系统方法，简称 OO (Object-Oriented)方法，Object Oriented 是建立在“对象”概念基础上的方法学。对象是由数据和容许的操作组成的封装体，与客观实体有直接对应关系，一个对象类定义了具有相似性质的一组对象。而继承性是对具有层次关系的类的属性和操作进行共享的一种方式。所谓面向对象就是基于对象概念，以对象为中心，以类和继承为构造机制，来认识、理解、刻画客观世界和设计、构建相应的软件系统。

2. 核心思想

封装（Encapsulation）、继承（Inheritance）、多态（Polymorphism）。

二. ECS 架构

在游戏项目开发的过程中，一般会使用 OOP 的设计方式让 GameObject 处理自身的业务，然后框架去管理 GameObject 的集合。但是使用 OOP 的思想进行框架设计的难点在于一开始就要构建出一个清晰类层次结构。而且在开发过程中需要改动类层次结构的可能性非常大，越到开发后期对类层次结构的改动就会越困难。

经过一段时间的开发，总会在某个时间点开始引入多重继承。实现一个又可工作、又易理解、又易维护的多重继承类层次结构的难度通常超过其收益。因此多数游戏工作室禁止或严格限制在类层次结构中使用多重继承。若非要使用多重继承，要求一个类只能多重继承一些 简单且无父类的类

也就是说在大型游戏项目中，OOP 并不适用于框架设计。但是也不用完全抛弃 OOP，只是在很大程度上，代码中的类不再具体地对应现实世界中的具体物件，ECS 中类的语义变得更加抽象了。

ECS 有一个很重要的思想：数据都放在一边，需要的时候就去用，不需要的时候不要动。ECS 的本质就是数据和操作分离。传统 OOP 思想常常会面临一种情况，A 打了 B，那么到底是 A 主动打了 B 还是 B 被 A 打了，这个函数该放在哪里。但是 ECS 不用纠结这个问题，数据存放到 Component 种，逻辑直接由 System 接管。借着这个思想，我们可以大幅度减少函数调用的层次，进而缩短数据流传递的深度。

1. 何为 ECS 架构

ECS，即 Entity-Component-System（实体-组件-系统）的缩写，Entity 由多个 Component 组成，Component 由数据组成，System 由逻辑组成。其模式遵循组合优于继承原则，游戏内的每一个基本单元都是一个实体，每个实体又由一个或多个组件构成，每个组件仅仅包含代表其特性的数据（即在组件中没有任何方法），例如：移动相关的组件 MoveComponent 包含速度、位置、朝向等属性，一旦一个实体拥有了 MoveComponent 组件便可以认为它拥有了移动的能力，系统(System)便是来处理拥有一个或多个相同组件的实体集合的工具，其只拥有行为（即在系统中没有任何数据），在这个例子中，处理移动的系统仅仅关心拥有移动能力的实体，它会遍历所有拥有 MoveComponent 组件的实体，并根据相关的数据（速度、位置、朝向等），更新实体的位置。

实体与组件是一个一对多的关系，实体拥有怎样的能力，完全取决于其拥有哪些组件，通过动态添加或删除组件，可以在（游戏）运行时改变实体的行为。

2. 实体、组件与系统

先有一个 World，它是系统和实体的集合，而实体就是一个 ID，这个 ID 对应了组件的集合。组件用来存储游戏状态并且没有任何行为，系统拥有处理实体的行为但是没有状态。

1) 实体—Entity

实体只是一个概念上的定义，指的是存在你游戏世界中的一个独特物体，是一系列组件的集合。为了方便区分不同的实体，在代码层面上一般用一个 ID 来进行表示。所有组成这个实体的组件将会被这个 ID 标记，从而明确哪些组件属于该实体。由于其是一系列组件的集合，因此完全可以在运行时动态地为实体增加一个新的组件或是将组件从实体中移除。比如，玩家实体因为某些原因（可能陷入昏迷）而丧失了移动能力，只需简单地将移动组件从该实体身上移除，便可以达到无法移动的效果了。

Entity 需要遵循立即创建和延迟销毁原则，销毁放在帧末执行。因为可能会出现这样的情况：systemA 提出要在 entityA 所在位置创建一个特效，然后 systemB 认为需要销毁 entityA。如果 systemB 直接销毁了 entityA，那么稍后 FxSystem 就会拿不到 entityA 的位置导致特效播放失败（你可能会问为什么不直接把 entityA 的位置记录下来，这样就不会有问题了。这里只是简单举个例子，不要太深究(●'◡'●)）。理想的表现效果应该是，播放特效后消失。

PS：

Player(Position, Sprite, Velocity, Health)

Enemy(Position, Sprite, Velocity, Health, AI)

Tree(Position, Sprite)

注：括号前为实体名，括号内为该实体拥有的组件。

2) 组件—Component（只有变量，没有函数）

一个组件是一堆数据的集合，可以使用 C 语言中的结构体来进行实现。它没有方法，即不存在任何的行为，只用来存储状态。（Component 之间不可以直接通信）一个经典的实现是：每一个组件都继承（或实现）同一个基类（或接口），通过这样的方法，我们能够非常方便地在运行时动态添加、识别、移除组件。每一个组件的意义在于描述实体的某一个特性。例如，PositionComponent（位置组件），其拥有 x、y 两个数据，用来描述实体的位置信息，拥有 PositionComponent 的实体便可以说在游戏世界中拥有了一席之地。当组件们单独存在的时候，实际上是没有什么意义的，但是当多个组件通过系统的方式组织在一起，才能发挥出真正的力量。同时，我们还可以用空组件（不含任何数据的组件）对实体进行标记，从而在运行时动态地识别它。如，EnemyComponent 这个组件可以不含有任何数据，只是拥有该组件的实体会被标记为“敌人”。

根据实际开发需求，这里还会存在一种特殊的组件，名为 Singleton Component （单例组件），顾名思义，单例组件在一个上下文中有且只有一个。

PS：

PositionComponent(x, y)

VelocityComponent(X, y)

HealthComponent(value)

PlayerComponent()

EnemyComponent()

注：括号前为组件名，括号内为该组件拥有的数据

3) 系统—System

理解了实体和组件便会发现，到这里我们还未曾提到过游戏逻辑相关的话题。系统便是 ECS 架构中用来处理游戏逻辑的部分。何为系统，一个系统就是对拥有一个或多个相同组

件的实体集合进行操作的工具，它只有行为，没有状态，即不应该存放任何数据。举个例子，游戏中玩家要操作对应的角色进行移动，由上面两部分可知，角色是一个实体，其拥有位置和速度组件，那么怎么根据实体拥有的速度去刷新其位置呢，MoveSystem（移动系统）登场，它可以得到所有拥有位置和速度组件的实体集合，遍历这个集合，根据每一个实体拥有的速度值和物理引擎去计算该实体应该所处的位置，并刷新该实体位置组件的值，至此，完成了玩家操控的角色移动了。

System 之间的执行顺序需要严格制定，且 System 之间不可以直接通信。

注意，我强调了移动系统可以得到所有拥有位置和速度组件的实体集合，因为一个实体同时拥有位置和速度组件，我们便认为该实体拥有移动的能力，因此移动系统可以去刷新每一个符合要求的实体的位置。这样做的好处在于，当我们玩家操控的角色因为某种原因不能移动时，我们只需要将速度组件从该实体中移除，移动系统就得不到角色的引用了，同样的，如果我们希望游戏场景中的某一个物件动起来，只需要为其添加一个速度组件就万事大吉。

一个系统关心实体拥有哪些组件是由我们决定的，通过一些手段，我们可以在系统中很快地得到对应实体的集合。

前面提到过的 Singleton Component（单例组件），明白了系统的概念更容易说明，还是玩家操作角色的例子，该实体速度组件的值从何而来，一般情况下是根据玩家的操作输入去赋予对应的数值。这里就涉及到一个新组件 InputComponent（输入组件）和一个新系统 ChangePlayerVelocitySystem（改变玩家速度系统），改变玩家速度系统会根据输入组件的值去改变玩家速度，假设还有一个系统 FireSystem（开火系统），它会根据玩家是否输入开火键进行开火操作，那么就有 2 个系统同时依赖输入组件，真实游戏情况可能比这还要复杂，有无数个系统都要依赖于输入组件，同时拥有输入组件的实体在游戏中仅仅需要有一个，每帧去刷新它的值就可以了，这时很容易让人想到单例模式（便捷地访问、只有一个引用），同样的，单例组件也是指整个游戏世界中有且只有一个实体拥有该组件，并且希望各系统能够便捷的访问到它，经过一些处理，在任何系统中都能通过类似 world->GetSingletonInput() 的方法来获得该组件引用。

系统这里比较麻烦，还存在一个常见问题：由于代码逻辑分布于各个系统中，各个系统之间为了解耦又不能互相访问，那么如果有多个系统希望运行同样的逻辑，该如何解决，总不能把代码复制 N 份，放到各个系统之中。UtilityFunction（实用函数） 便是用来解决这一问题的，它将被多个系统调用的方法单独提取出来，放到统一的地方，各个系统通过 UtilityFunction 调用想执行的方法，同系统一样， UtilityFunction 中不能存放状态，它应该是拥有各个方法的纯净集合。

PS：

MoveSystem(Position, Velocity)

RenderSystem(Position, Sprite)

注：括号前为系统名，括号内为该系统关心的组件集合