这本书提供的并不是一个固定设计模式的写法，而是的是编程和设计的思想

命令模式

将命令独立封装成一个类，使得命令与行为的解耦，命令类不关注是谁调用了命令的实例，将原本的强耦合关系变为单项依赖；命令模式中实现双向队列将行为变为了数据，保存执行行为前的状态实现撤销、重做操作，例如在Moba游戏中可以把鼠标点击输入和英雄移动解耦，并且使得可以随时打断英雄单位的状态，使得英雄朝最新的目标点移动，命令模式也可以通过工厂实例化命令类。但是命令模式的缺点一是可能会导致大量的实例化浪费内存，二是每次添加新的特殊的行为时就需要重新写一个对应的命令类。

1. 享元模式

分内部状态和外部状态，让不同的实例共享相同的内部状态，同时定义自身的外部状态，通常用于共享材质、网格，保留可修改的位置信息，缩放相关信息，能显著减少Draw Call提升性能。享元模式着重于共享现有对象。

1. 观察者模式

将事件从推送变为拉取，使得不需要在该实例中存储被影响者的引用，从而降低耦合程度。观察者拉取被观察者发出的事件从而更改自身的状态。

1. 原型模式

通过一个对象生成与之相似的其他对象。可以通过在基类中定义虚函数，继承它的子类通过重写这个虚函数的方式来生成相似特征的对象，同样也可以使用模板类或则泛型来完成原型模式。原型模式着重于克隆现有对象并创建相似的新对象。

1. 单例模式

保证全局存在唯一一个可被全局访问的实例。在需要保存状态时使用单例模式，而在只需要提供全局的方法时使用只包含静态方法的类会更好。单例模式实际上促成了耦合，并破坏了封装性，需要时刻注意全局的状态是否被改变。

1. 状态模式

即有限状态机的实现。

1. 双缓冲模式

在渲染中广泛使用，一帧准备好后才会被渲染到屏幕上，减少了单缓冲模式下用户会看到“部分更新的内容”——画面撕裂的问题，并提高了渲染效率。但仍存在卡顿（帧率低于显示器刷新率并开启垂直同步）、跳帧和画面撕裂（帧率高于刷新率未开启垂直同步）等问题。三缓冲模式，双缓冲模式中在未收到信号的情况下，后台的缓冲区中已经绘制完成后会停止绘制等待信号，而三缓冲模式在后台维护两个缓冲区，两个缓冲区的绘制不会停下，在收到信号后选最新被绘制完成的缓冲区替换屏幕上显示的图像，这减小了画面的延迟。借助更大一个缓冲区池，可以完成简单的动态模糊。

1. 游戏循环

在游戏循环中，对于AI和物理等模块，固定时间的循环可以使得这些模块更加稳定，毕竟如果是帧驱动，如果两台性能不同的机器上，时机表现差距会很大；对于渲染这些参与表现的模块，对实时性要求高、性能要求高，灵活的动态时间循环更加适合。固定更新时间步长，动态渲染，设置一个固定的时间，在一次循环开始时记录当前时间，接收输入后开始另一个嵌套在主线程循环中的循环，在其中更新物理、AI等需要固定更新时间的组件，直到耗尽被预先设置好的固定时间，此时记录超出预定的时间，并依据这个时间更新渲染，因为渲染器会提前绘制，它知道此时各个对象的速度，可以根据这个超出的时间与固定时间之比获取正确的应当显示在屏幕上的图像。

1. 更新方法

让每个实体定义自身的行为，继承统一的一个抽象层，在抽象层中维护一个更新列表，按顺序调用。比如在Unity中可能是通过继承MonoBehaviour，每帧通过反射获取类型对象的实例，并调用它的Update方法。因为更新并不是同步，所以调用顺序会影响最终的结果。

1. 字节码

通过把行为存储在数据中，通过定义命令，从文本文件等文件中读取数据和执行对应的操作（或许可以通过位标志组合来组合操作？）。实际上我对它的应用场景知道得很少，因此对于这个模式还是一头雾水。

1. 子类沙箱

将子类共同的行为定义在一个共同的父类中，其他模块与对应子类模块的耦合实际上只会发生在父类中，但是会存在父类代码量大且繁冗的问题，可以把影响大量子类的共同行为定义在父类中，子类使用父类的方法，影响范围小的行为则放在子类中，这样即使耦合了，影响范围小且修改难度也不高。

1. 类型对象

创造一个类来允许灵活的创造新的类，而该的每个实例都代表了不同类型的对象。适用于游戏需要后续更新的情况，可以不使用硬编码而使用数据文件。感觉上和原型模式与享元模式相似，类型对象应该着重点在于数据的分享而不是前两者的代码的分享。

1. 组件模式

在单一的场景对象的实例中需要包含多个不同领域的功能时，将各个模块封装为组件，只有一个管理所有组件状态的类持有组件并更新各个组件的状态。

1. 事件序列

通过完成一个循环队列，将消息和事件在各个组件间进行时间上的解耦。用发布订阅模式完成一个全局的事件系统，但是就我自己的看法来说，可能只适合全局中不同模块间的解耦，比如UI和战斗模块间可以使用，但如果在战斗模块之中，比如玩家对场景中的一个敌人造成伤害，这种情况下或许直接耦合会更好，因为使用事件序列会通知场景中的每一个敌人，让这些敌人自己去判断是否应该扣血，这样明显不合理。

1. 服务定位器

提供服务的全局接入点，而不必让用户和实现它的具体类耦合。通过一个中间的服务定位器类，提供合适的全局访问类，解开了单例模式中隐藏起来的耦合，但是它的实现更加复杂，这意味着更加脆弱，比如服务定位器类并不知道将被提供的全局接入点是否为空，这可能会导致错误，因此和单例模式一样需要尽量少地使用。

1. 数据局部性

提高缓存命中率，从而提高运行速度，将代码中原本通过引用跳转的部分变为数组，连续地存放在内存中。但这需要舍弃代码的灵活性。

1. 脏标识模式

为了避免不必要的计算或者传输开销，在需要结果时才去执行工作。我对这个模式也是相当陌生的，尽管我可能知道它是如何实现的，总之印象很浅

1. 对象池模式

为了优化内存的设计模式，避免反复创建和销毁对象以此降低开销、减少内存碎片，提前统一创建放入池中，需要时改变激活和失活状态，放到需要的地方。将池子中的激活对象放在列表的两个部分，这样便于快速找到从池中取出一个对象并激活。

1. 空间分区

建立细分空间用于存储对象，可以帮助告诉对象位置，降低算法复杂度。