1. 名词解释
   a) 设计的 5 个准则（设计要考虑功能性、审美性）（写出准则，或者解释准则的含义）
      1) 准则 1：**用“美”的方式实现功能，是设计的价值**（而用软件功能解决用户的问题是需求分析的价值）。设计的审美需要考虑**简洁性**、**一致性**（概念完整）、**坚固性**（高质量）
      2) 准则 2：**设计复杂度 = 事物复杂度 + 载体与事物的适配复杂度**。设计的难度包含设事物复杂度、载体复杂度
      3) 准则 3：**设计重在内部结构，不是外在表现。**外部表现是对外表现的能力，外部表现是为了满足职责；内部结构则是为了完成的质量，通过罗列方式完成外部表现的内部结构不是好的内部结构。设计的目的就是为了保证质量，因此其重在内部结构
      4) 准则 4：**只有高层设计良好，低层设计才可能良好**。需求分析->体系结构设计->详细设计->编码
      5) 准则 5：**只有写完并测试完代码之后，才能算是完成了设计**（敏捷视点）
   b) 4+1 view，即逻辑视图、开发视图、进程视图、部署视图 + 用例视图，前四个为体系结构视图，后一个为需求视图
      1) **用例视图 Use case view**，用例视图关注的是软件体系结构的需求，说明了体系结构设计的出发点，同时可用于验证和评估体系结构的设计。包括来自于需求说明书中的概要功能需求和质量需求
      2) **逻辑视图 Logical view**，逻辑视图从总体功能组织的角度来描述软件系统的高层结构，其主要内容是软件体系结构的抽象规格，主要关注点是满足用户的各项需求，尤其是功能需求、质量属性需求和约束。以面向对象的方式对系统进行分解，分解为关键抽象，组织成对象和对象集，最终分解出 component、connector、configuration 等模块。
      3) **开发视图 Development view**，开发视图描述了软件体系结构在开发中的实现与演化。对子系统进行划分，说明了模块之间和子系统之间的导入导出关系。主要考虑软件模块的组织和实现（分层、复用、变更、软件管理）
      4) **进程视图 Process view**，进程视图关注软件体系结构的运行时表现，考虑在静态结构中难以分析的性能、可靠性等非功能需求，也会描述为保障系统完整性和容错性而需要的进程并发及其分布。进程是一组任务组成的一个执行单元，程序需要被切分成不同的进程来支持并发，从而带来较高的性能和吞吐量
      5) **部署视图 Deployment view**，部署视图描述软件体系结构的基础设施和网络分布，将软件映射到硬件上，即放在不同的网络节点上。考虑的是非功能需求（可靠性、吞吐量、容错性、容量），主要是底层资源的使用和性能
   c) OO 协作与协作设计
      1) 协作：
         1. 系统行为由对象间协作实现。一个应用可以被拆分成许多不同的行为，每一个行为都由一组对象通过协作的方式实现。无论协作的大小，每一个协作都实现了应用的一个行为
         2. 一个面向对象的应用实际上是一个由对象间关系连接起来的对象网络，而协作就是在该网络中实现某种应用行为的消息模式，协作分散在对象网络之中。设计对象协作之前要先找到系统行为
      2) 协作设计：是对对象间的协作模式进行设计的过程，协作描述了系统行为在对象网络中的分布逻辑，可分为集中式、分散式、委托式三种

d) OO 职责与职责分配
   1) 职责来自于
      1. 需求用例所描述的系统行为，包括用户和系统的交互、需要隐藏的 secret
      2. 软件体系结构设计，包括体系结构设计的决策和约束
      3. 详细设计的因素，即详细设计的决策
      4. 其他内容，比如对象的创建与销毁，关系的创建与销毁，数据维护，协作设计
   2) 职责分配：
      1. 在面向对象的系统中是由多个对象组成的，这些对象必需能够向其它对象发送消息或操作，这就是对象的交互和职责分配
      2. 职责分配是将职责分配到结构中的一个或多个类中，主要的职责分配模式是 GRASP 模式，职责的分配需要遵循低耦合、高内聚的原则

e) GRASP 模式（通用的职责分配软件模式）
   1) **低耦合**：分配职责时保证低耦合，即降低依赖并增加复用性
   2) **高内聚**：分配职责后保证模块内部的内聚性很高，将复杂度控制在可管理的范围之内。不可能完全消除时序内聚、过程内聚、通信内聚，必须将这些内聚的拥有者变成转发者。一个对象仅实现单一的职责，而不实现复杂的职责组合
   3) **专家模式**：（促进高内聚）解决面向对象设计中的职责分配问题，同时保证对信息的封装。分配职责要看对象所拥有的信息，谁拥有哪方面的信息，谁就负责哪方面的职责，并且一个对象只做这一件事情
   4) **创建者模式**：（促进低耦合）通过考虑对象间的关系，来决定某个对象该由谁来创建。如果某个对象与其他对象已经有聚合/包含关系，则这个对象应该由聚合/包含它的对象来创建。这样就不需要依赖第三方，不会增加新的耦合
   5) **控制者模式**：对象协作设计中的一种风格，解决“处理系统事件”这一职责的分配问题
      1. 如果一个程序可以接收外部事件（如 GUI 事件），它需要在事件源和事件处理模块之间加一个事件管理模块，从而将这二者解耦
      2. 控制者（可以是外观控制者，或者一个纯虚构对象）接受所有的外界请求并转发，很可能形成集中式风格
      3. Controller 肯定会跟两边对象都有很严重的耦合，并且由于它负责转发很多信息，所以它的内聚度不高
   6) **多态模式**：处理基于类型的变更情况。当遇到根据对象的类型来选择系统行为的情况时，需要使用多态的方法调用，而不是通过 if/else 语句来做选择。
   7) **纯虚构模式**：（促进高内聚）额外创建一个虚构的类，将一些相关的、内聚度很高的职责放入这个实际不存在的类中，从而促进高内聚、低耦合、可复用。比如将数据库的处理方法放在一个虚构的类中，相关的还有与显示、平台（硬件）相关的处理代码，以及复杂的行为和数据结构，都可以放在虚构的类中
   8) **间接模式**：在需要解耦的两个对象之间增加一个中间对象（并将可能引发耦合的职责分配到这个对象中），防止它们直接耦合，从而提高可复用性。在 filter-pipe 方法中，filter 中间的 pipe 就是间接的对象；从详细设计的角度来考虑，适配器模式就是加了一个间接对象来适配
   9) **保护差异模式**：识别对象的可能变更和不稳定性，设计一个稳定的接口来应对将来可能发生的变化，由稳定的接口来承担对象的访问职责

2. 设计的审美

a) 设计的审美标准有哪些：
   1) 简洁性：模块化
   2) 结构一致性（概念完整性）：体系结构的风格，模块化
   3) 坚固性（高质量）：易开发、易修改、易复用、易调试、易维护
b) 列举已知的软件设计方法与技术（至少 5 种），并说明它们促进了哪些审美标准的达成？
   1) **模块化**：进行模块划分，隐藏一些程序片段（数据结构和算法）的实现细节，暴露接口与外界；且保证模块内部的内聚度较高，模块与外界的耦合较低。模块隐藏实现细节，通过接口访问模块，因此促进了简洁性；且因为功能内聚，对外提供统一的外部接口，因此促进了结构一致性
   2) **信息隐藏**：将系统分成模块，每个模块封装一个重要决策，且只有该模块知道实现细节。决策类型可以是需求、变更，不同的决策之间相互独立。信息隐藏和模块化都在一定程度上促进了简洁性，但是只是体现在外部接口部分。信息隐藏为了处理一些不需要对外表现的决策（内部实现的可修改性）而进行了片段分割，这又在一定程度上牺牲了简洁性而达到了坚固性
   3) **运行时注册**：针对系统变化，将可能变化的部分与其他部分解耦，不直接发生程序调用，而是在运行时注册。因为这个技术针对可能的变更而使用，本来可以用一个部件处理的事情，被分解为多个部分，而且这些不同部分之间还有着复杂的交互规则，所以牺牲了简洁性，以提高坚固性（灵活性）
   4) **配置式编程**：针对系统变化，主要解决共性与差异性问题。将可能变化的部分写在一个配置文件中，当要发生变化时，直接修改配置文件。因为需要充分考虑可能的变更来组织配置文件，并且需要在系统启动时对配置文件进行解析，所以牺牲了简洁性，以提高坚固性（灵活性）
   5) **设计模式**：设计模式牺牲简洁性达到坚固性，保证程序的可维护性和可扩展性。同时，在设计模式中是用同样的方法做同样的事情，因此促进了程序的结构一致性的达成

3. 设计的层次性【高层设计、中层设计和低层设计各自的出发点、主要关注因素（即哪些审美要素）、主要方法与技术和最终制品】
   a) 高层设计
      1) 出发点：弥补详细设计机制的不足，将一组模块组合起来形成整个系统，进行整体结构设计，同时关注系统的质量属性。隐藏详细设计中的导入导出关系和单词匹配，设计带有质量属性的部件 component，以及部件之间的关系连接件 connector。同时，体系结构也是一系列对系统设计所做的设计决策
      2) 主要关注因素：质量属性，比如可用性、可修改性、效率、安全性、可测试性；项目环境，包括开发环境、业务环境、技术环境；业务目标。为了达成以上目标，要求体系结构满足简洁性、一致性、坚固性
      3) 主要方法与技术：
         1. 方法：4+1 view、场景驱动、体系结构风格
         2. 技术：模块的表示方法可以是 box-line、formal language（ADL，架构描述语言）、UML（4+1 view 模式使用 UML 技术实现）
      4) 最终制品：体系结构
   b) 中层设计
      1) 出发点：进行模块划分，隐藏一些程序片段（数据结构和算法）的实现细节，暴露接口与外界。模块化要做到尽可能独立，模块内部的内聚度高，而模块与

外界的耦合度低

2) 主要关注因素：简洁性（易开发、易修改、易复用），可观察性（看上去“显然是正确的”，易开发、易调试、易维护）
3) 主要方法与技术：
    1. 低耦合（将模块之间的关系最小化），高内聚（模块内部元素之间的联系最大化）
    2. 信息隐藏，一个模块只封装一个 secret（主要秘密是需求决策，次要秘密是修改决策），给出要修改部分的接口，隐藏待修改部分的实现细节
    3. 结合模块化和信息隐藏的方式，再加上封装、继承、多态等技术，进行面向对象的设计
4) 最终制品：模块与类结构

c) 低层设计

1) 出发点：将基本的语言单位（类型与语句）组织起来，建立高质量的数据结构和算法（数据结构合理易用，算法可靠、高效、易读）。屏蔽程序中复杂的数据结构与算法的实现细节
2) 主要关注因素：数据结构与算法的简洁性（易读）、坚固性（可靠，易维护）
3) 主要方法与技术：防御式编程，断言式编程，测试驱动开发，异常处理，配置式编程，表驱动编程，基于状态机编程
4) 最终制品：算法与数据结构，单个的函数

4. 软件体系结构风格

a) 描述或比较相关风格

1) Share Data 共享数据风格：repository 使用 pull 风格，需要 agent 不断地查询中间的数据是否发生变化，如果发生变化 agent 会做出响应；blackboard 使用 push 风格，每一个 agent 会向中间数据注册，如果中间数据发生变化，则会通知每一个 agent。它的部件包括中心数据结构以及一系列操作于中心数据结构之上的部件，连接件是程序调用或者直接的内存访问
    1. 优点：能够储存大量数据，因为数据是共享的；集中区可以做一些复杂的设计；安全控制、并发控制、复用、备份都比较容易处理
    2. 缺点：所有的 agent 都依赖于中间的数据区，这里是瓶颈；如果要修改中间数据，则会大幅度提高修改成本
2) Pipe-Filter 管道过滤器风格：类似于一条流水线，流水线上的模块并发地做自己的工作，数据传递通过 pipe 传递。各个 filter 之间并发（只传递数据流）且独立（无控制流，且不共享数据），每一个 filter 都不知道其他的 filter，其正确性完全独立，不依赖于上一个/下一个 filter。它的部件是 filter，连接件是 pipe
    1. 优点：一个系统的整体 I/O 行为就是对所有 filter 行为的组合；只要两个 filter 之间传递的数据一致，则它们就可以连起来使用，因此可复用性较高；因为 filter 是独立的，所以增加、删除 filter 比较容易；所有 filter 的操作通常可以并发，提高系统的性能
    2. 缺点：不适用于交互式系统；传输数据需要额外的空间；两个 filter 的数据格式不一样的情况下，对数据打包、接包、转换格式会增加程序复杂度
3) Implicit Invocation 隐式调用风格：基于事件/消息的方式相互调用。部件向广播媒介中声明一个事件，其他的部件可以注册监听这个事件，并将自己的程序与这个事件联系起来，当事件发生后，广播媒介调用所有监听了该事件的程序。它有以下约束：每一个事件的抛出者并不知道谁是接收者，也没有默认，因为

它们之间是严格解耦的；事件抛出后，不能假设这个事件的处理顺序，甚至它是否会被处理都是未知的。它的部件是所有的代理（可以是对象、进程、方法），它的连接件是广播媒介（即事件管理）

1. 优点：只要部件注册到这个系统中的某一个事件，它就可以被用在这个系统中，因此可复用性较高；因为部件之间非常独立，部件的变更和替换不会需要其他部件修改接口
2. 缺点：因为事件是否会被处理是未知的，故而程序完整性和正确性很难被保证；程序很难测试和诊断

4) Main Program and Subroutine 主程序和子程序风格：结构化，程序被分为一定的层次。控制权转移有严格要求，单线控制。这种方法的算法中，所有的子程序都由主程序调用。它的部件是程序、方法、模块，连接件是程序调用

1. 优点：程序清晰、容易理解，且有很强的正确性控制
2. 缺点：难以变更或复用，可能造成公共耦合

5) Layered 分层风格：系统分成很多层，每一层都是一个独立的抽象部分。层之间通过程序调用来交互，只允许上层调用下层，不允许下层调用上层，而且不允许跨层调用。它的部件是一组特定的程序和对象，连接件是遵守可见性限制的程序调用

1. 优点：分层式结构适用于并行开发，只需要将各层的协议定义出来，各层内部的开发交由不同的开发小组各自负责即可；任何变更、增加都最多只会影响到两层，影响只会向上转移一层，绝对不会向下转移；只要接口相一致，就有很高的复用性
2. 缺点：不是所有的系统都能使用分层结构；逐层的调用会使得调用本身产生延迟，造成性能下降

6) Object-Oriented 面向对象风格：对象将秘密封装起来，不与外部环境耦合，以获得较高的可修改性；对象是自治的，负责自己内部的数据表达和完整性；对象是平等的，不受其他对象控制，只能通过调用方法来访问一个对象。它的部件是对象或模块，连接件是方法调用

1. 优点：能够将数据封装起来，不会因为数据共享而产生相互之间的影响；因为对象的自治性质，可以将系统逻辑分成一个个小的逻辑
2. 缺点：如果要使用某一个对象，则必须先知道某一个对象的身份，则如果该对象的身份发生了变化，会对所有调用这个对象的其他对象产生影响；对象协作可能引发副作用

b) 对给定场景，判断需要使用的风格

1) Main Program and Subroutine 主程序和子程序风格：需要顺序执行任务的系统；对正确性要求很高的系统
2) Object-Oriented 面向对象风格：适用于有一个中心问题并且需要保护相关信息的应用中；数据展示和相关操作被封装在抽象数据类型中
3) Layered 分层风格：能够把程序分成不同的层次，层次之间的协议是稳定的，程序可能只在某一层内部发生修改。比如 OSI 七层模型，比如 UNIX 系统
4) Pipe-Filter 管道过滤器风格：如果有一系列任务需要操作一个有顺序性的数据，且这些任务需要独立执行，那么便适用于此风格。比如有顺序的批处理程序、UNIX/Linux 命令管道、编译器、信号处理、大数据量处理
5) Implicit Invocation 隐式调用风格：通常适用于松散耦合系统，部件之间关联比较少，每一个部件都有它独立的操作，但是会有事件发生需要用到某些操作。

比如调试系统、数据库管理系统、GUI

6) Share Data 共享数据风格：大型信息系统，能够将数据做集中处理，并且需要维护一个复杂的中心信息。常见如数据库、专家系统、编程环境。在网络应用中，网络日志使用 repository 风格，而网络聊天室使用 blackboard 风格

5. 职责分配与协作设计

a) 协作设计（控制风格）的比较和场景判定（给定一个场景，判定应该使用哪一种）

1) 集中式，将整个系统逻辑集中在某一个 controller 模块中。controller 并不做具体的事情，它从其他模块中收集信息并做出决策，而后根据决策来指挥其他模块完成任务。不推荐使用

2) 分散式，将整个系统逻辑分散在系统模块中，没有一个集中的 controller。虽然分散式控制风格符合面向对象的思想，但是它的控制流太分散，需要大量对象的合作，因此耦合度高，内聚性低，并且很难隐藏信息。不推荐使用

3) 委托式，决策制定工作分散在对象网络中，但是会有几个 controller 做主要决策，每一个 controller 只绑定少数几个 component，支持信息隐藏以及部分的分散。委托式位于完全集中和完全分散之间。推荐使用

b) 对给定场景和要求的控制风格，根据 GRASP 模式，判断特定职责的分配

1) **低耦合**：不要发生程序调用的地方就不要调用。比如 Post 和 Database 之间已经被 DatabaseManager 给隔开了，应当 Post 调用 DatabaseManager，就不要 Post 直接调用 Database 的方法了

2) **高内聚**：一个对象内部的方法应当相关，如果不相关则应该放在不同的对象中，保证一个对象只封装一个职责。比如一个对象内有三个方法，但是都是凑在一起的，这样内聚度不高；它可以将这三个方法做成三个职责，分别放在三个不同的对象中，然后再由该对象进行调用

3) **专家模式**：比如 PPT 上的例子，要获取订单总额，只有 Sale 对象知道订单中的所有商品，因此“获取订单总额”这一职责应当放在 Sale 对象中。专家模式即为“知道什么就做什么”

4) **创建者模式**：B 聚合了 A，或者 B 包含 A，或者 B 引用到了 A，则应当由 B 来创建 A，而不是 B 调用 C 来创建 A。比如 Sale 对象包含 SaleLineItem 的信息，因此应当由 Sale 来创建 SaleLineItem

5) **控制者模式**：当有外部事件发生时，应当使用一个事件转发对象。比如 OnEnterItem()事件来自于 GUI 显示层，应当先转发到 Post 对象，再由 Post 对象调用 Sale 对象，来对这个事件做真正的处理。不能让 GUI 与 Sale 直接耦合

6) **多态模式**：当需要根据对象类型做方法的选择时，不使用 switch 语句，而是使用一个接口来声明这个方法，再让所有的类型实现这个接口。这样在使用某个对象的时候，直接调用的就是它配套的方法

7) **纯虚构模式**：比如数据库的操作方法，增删改查和数据库连接之类的方法，都放在一个在问题域中不存在的对象中

8) **间接模式**：当希望两者不直接耦合时，就在中间加一个间接的对象，然后把相关的职责放在这个对象中。上面纯虚构模式中的数据库处理对象也是这里的间接对象

9) **保护差异模式**：将职责分配到外部的稳定接口之中

c) 根据分析类图和体系结构模块接口，建立基本的设计类图（20 分左右）

1)

6. 设计模式

a) 设计模式部分的所有思考题（全部整理，可能会有两题及以上）

1) 有一个数据列表 DataList，其基本类型是 3 维向量 ThreeD<x,y,z>。假设有三个外部对象 A,B,C，分别对其 x、y、z 维度感兴趣，希望访问 DataList 在相应维度的数据并进行处理（分别使用迭代器与 Proxy）。要求：（1）用 Java 语言实现该数据列表的数据结构；（2）请定义其对外的数据接口

1. **迭代器模式：**

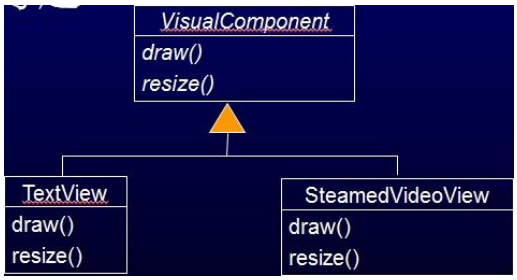
```
public class ThreeD { //这个类是集合对象的基本类型
        private Object X, Y, Z;
        public Object getX(); public Object getY(); public Object getZ();
}
public interface DataList {
        public Iterator CreateIteratorX();
        public Iterator CreateIteratorY();
        public Iterator CreateIteratorZ();
        public int Count();
        public ThreeD GetItem(int index);
}
public class ConcreteDataList implements DataList {
        private ArrayList<ThreeD> dataList = new ArrayList<ThreeD>;
        public Iterator CreateIteratorX() {return new IteratorX(this);}
        public Iterator CreateIteratorY() {return new IteratorY(this);}
        public Iterator CreateIteratorZ() {return new IteratorZ(this);}
        public int Count() {return dataList.size(); }
        public ThreeD GetItem(int index) {return dataList.get(index); }
}
public interface Iterator{
        First(); Next(); IsDone(); CurrentItem();
}
public class IteratorX Implements Iterator { //在这个对象中，其只访问遍历 X 维度的数据
        private DataList dataList;
        private int index;
        public IteratorX(DataList dataList) {this. dataList = dataList; }
        public Object CurrentItem() {dataList.getItem(index).getX(); }
}
//另外，又有 IteratorY 和 IteratorZ 两个对象，分别遍历 Y/Z 维度的内容
clientA {
        DataList dataList = new ConcreteDataList();
        Iterator iterator = dataList. CreateIteratorX();
}
```

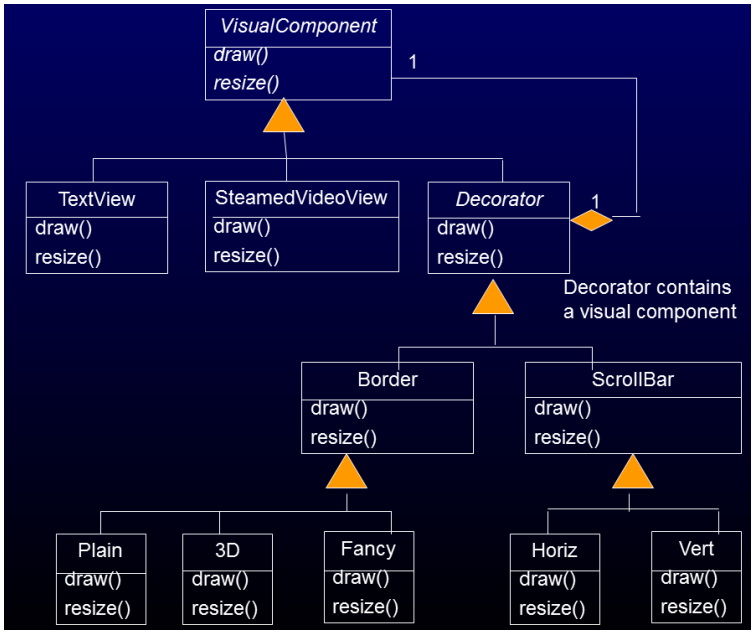
2. 代理模式

```
public class ThreeD { //这个类是集合对象的基本类型
        private Object X, Y, Z;
        public Object getX(); public Object getY(); public Object getZ();
```

```
}
public interface DataList {
        public int Count();
        public Object getObject();
}
public class ConcreteDataList implements DataList {
        private ArrayList<ThreeD> dataList = new ArrayList<ThreeD>;
}
public class proxyX implements DataList { //只访问 X 维度
        DataList dataList;
        public Object getObject() { return ((ThreeD)dataList.getObject()).getX(); }
}
public class proxyY implements DataList { DataList dataList; } //只访问 Y 维度
public class proxyZ implements DataList { DataList dataList; } //只访问 Z 维度
public class Factory {
        DataList createDataListX() {
                ProxyX proxy;
                ConcreteDataList dataList = new ConcreteDataList ();
                proxy.setRealDataList(ConcreteDataList);
                return proxy;
        }
        //同理，又有 Y/Z 这两个维度的代理的创建方法
}
```

2) 如果要对该系统做扩展，增加上 Border（可能有 Plain、3D、Fancy 这三种类型），以及 ScrollBar(可能有 Horiz、Vert 这两种类型)（装饰者模式）





1. 为所有可能添加的内容创建一个接口 Decorator，它与 View 一起实现 VisualComponent 接口，同时 Decorator 聚合一个 VisualComponent 的实例。
2. public abstract class Decorator implements VisualComponent {
   private VisualComponent visualComponent;
   public Decorato(VisualComponent visualComponent) {
   this.visualComponent = visualComponent;
   }
   public void draw() {
   visualComponent.draw();
   //在实际的 Decorator 中（即 Decorator 的子类），需要加上自己将自己画出来的代码，是装饰的代码
   }
   }
   Client {
   VisualComponent scrolledView = new Horz(new TextView());
   scrolledView.draw();
   }

3) 比较 facade 与 controller(collaboration design)的异同：
   1. 相同：它们不自己承担具体的职责，都是进行一些任务转发的功能；它们都是对两个模块或外部和内部进行解耦；都是抽出来的一层，使用一个同一的接口屏蔽了底层的实现
   2. 不同：controller 是面向用户需求的，需要在对象网络中收集信息并做出决策，而后将信息转发到承担这个职责的对象。但是 façade 不一定，也许只是单纯地抽取接口，并不做信息收集、决策、转发这些工作。所以 controller 通常是 façade，而 façade 不是 controller
4) 在策略模式中，为什么使用“聚合”而不是“关联”关系？
   1. 在策略模式中，主程序和发生变化的算法之间的关系必须是聚合关系，而不能是关联关系。因为策略可能是选择 1 个算法，也可能选择多个算法，

选择算法的个数根据所聚合的算法对象的个数而定

2. 而如果仅是普通的关联关系的话，一次只能选择一个算法，因此如果同时使用多个策略则必须使用聚合关系

5) 共性和差异性问题

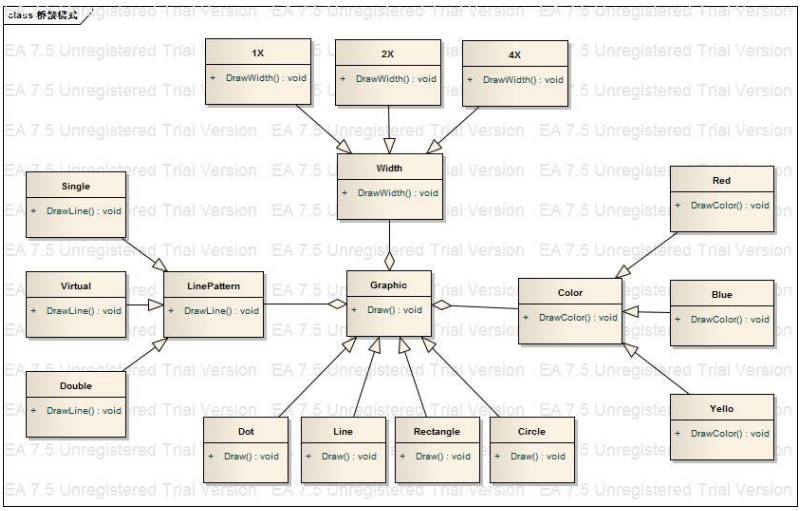
1. 如果一个对象有超过 2 个的差异行为，如何处理？使用多个策略树，一个策略树对应一个差异行为，将会发生差异的地方独立出来并进行封装
2. 如果一个对象集的部分行为组（比如两个方法 f1()和 f2()，可以都是一种实现，也可以都是另一种实现）存在差异性，如何处理？将一个行为组做成一个策略树，即两个方法都放在一个接口中实现
3. 如果一个对象集的部分属性（以及依赖于这些属性的方法）存在差异性，如何处理？将发生变化的属性独立出来做成策略树，同时封装相关的方法。在策略树的实现对象中，将属性和方法封装在一起实现
4. 一个对象的行为需要协作对象来完成，但是协作对象存在差异性，如何处理？比如类中的 f 方法，它有时调用 A.f1()，有时调用 B.f2()：这是普通的行为差异性，同样将行为封装出来做成策略树
5. 如果一个对象集的行为因为属性的取值而存在差异性，如何处理？仍旧做成一个策略树，通过属性的条件来判断使用策略树中的哪一个策略（状态模式）

6) 比较 Strategy 与 State

1. 与策略比较，策略是 N 选 M，状态一般情况下是 N 选 1
2. 在 Strategy 模式中，行为的差异性变化是由 Client 与 Context 来控制的，Strategy 与其实现类本身不能够控制变化，只能由 Client 与 Context 来选择当前的策略对象；而在 State 模式中，Client 只在 Context 初始化的时候为其设定初始的状态，在之后的状态变化中，皆由 Context 与 State 的实现类 ConcreteState 来控制状态的变化，选择当前的状态对象

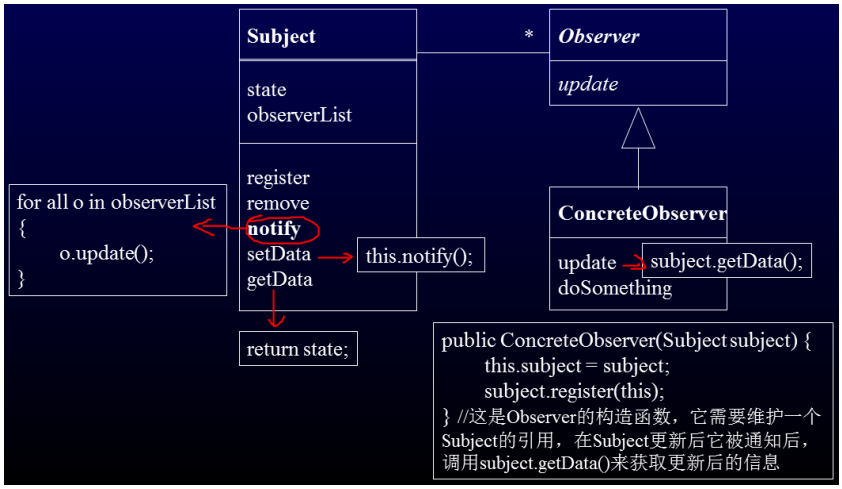
7) 使用 Bridge Pattern 实现一个画图程序的主体框架（解答：首先判断哪一个是接口，哪一个是实现，而后将接口和实现分别做成继承树）

1. 图形：点、线、矩形、圆…
2. 线形：实线、虚线、双线…（继承树）
3. 线粗细：1x、2x、4x（继承树）
4. 颜色：RGB（属性）
5. **解答**：图形是抽象，而线形、线粗细、颜色是对图像的实现



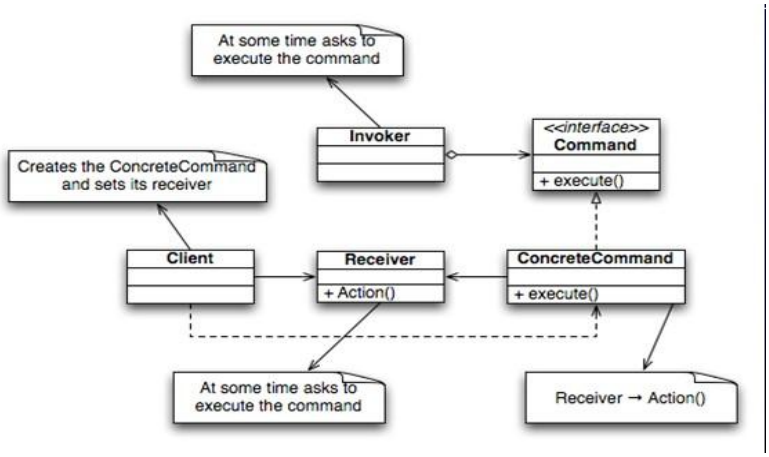
8) 与真正自实现的 Event Style 相比，Observer Pattern 的不同在哪里？

1. Event Style 可以实现多个事件对应多个不同的接口，事件发生后，事件控制中心一一调用注册了该事件的方法；在 Observer 模式中，如果要实现多个事件，则必须维护多个 Observer 的 list 以及相关的注册/移除方法，一个 ObserverList 中只能包含实现了同一个接口的 Observer
2. 相对于 Event Style 来讲，Observer 和 Subject 的耦合度要更强，因为 Observer 需要维护一个 Subject 的引用来获取它的状态，而 Event Style 中的事件源和事件处理者是解耦的
3. 在 Observer 模式中，Subject 中的 notify()方法直接调用 Observer 的 update()方法，可见它们的耦合；而在 Event Style 中，Subject 发生了变化是通知事件控制中心，再由控制中心来通知监听对象
4. 在 Observer 模式中，Observer 需要有一个继承树（必须有相同的接口），但是 Event Style 中的 listener 不需要处在一个继承树中



9) 如果众多 Observer 的接口不相同怎么办？（不都是 update 接口，甚至不是同一种类型）（用 Command 模式改良 Observer 模式）

1. 在 Command 模式中，操作被作为对象封装起来，实现一个 Command 接口。ConcreteCommand 聚合一个实际职责对象的引用，在的 execute()方法中调用这个对象的方法，ConcreteCommand 并不真正执行操作

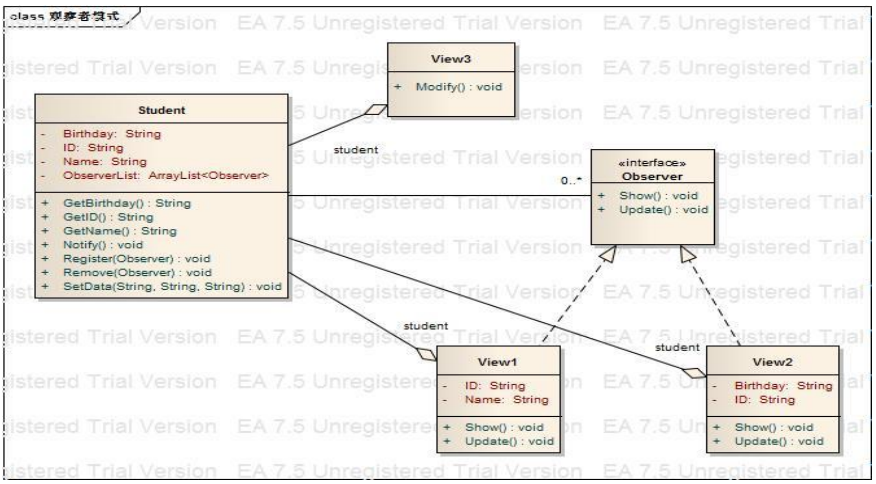


2. 使用一个 Command 的接口，一个 Observer 对应一个 ConcreteCommand，在 ConcreteCommand 的统一的 execute()方法中调用 Observer 的实际方法。在 Subject 包含一个 CommandList，在 notify()方法中一次调用每一个

Command 的 execute()方法

3. Client {
   Subject subject = new ConcreteSubject();
   Observer observer = new Observer(subject);
   Command command = new ConcreteCommand(observer);
   subject.Register(command);
   }
   class ConcreteCommand implements Command {
   Observer observer;
   ConcreteComman(Observer observer) { this.observer = observer; }
   void Execute() { observer.action(); };
   }
   class Observer {
   Subject subject;
   Observer(Subject subject) { this.subject = subject; }
   void action() { subject.getData(); //do something; }
   }
4. 如果要实现 Redo 和 Undo 的话，必须在 Command 中加上操作执行前后的环境信息，以及 Unexecute()方法来还原这个操作；同时在接收 Undo/Redo 操作的对象中保存一个 Undo 的 List 以及一个 Redo 的 List。在 Undo 后将撤销了的 Command 加入到 Redo 的 List 中，在 Redo 后将撤销了的 Cmmand 加入到 Undo 的 List 中

10) 用 observer 编写程序：Model: student（ID, name, birthday），View1:显示 ID+name，View2：显示 ID+birthday，View3：修改 student 的三个列；View1~2 为 observer，model 为 subject；实现每次 view3 中实现修改后，View1~2 自动更新

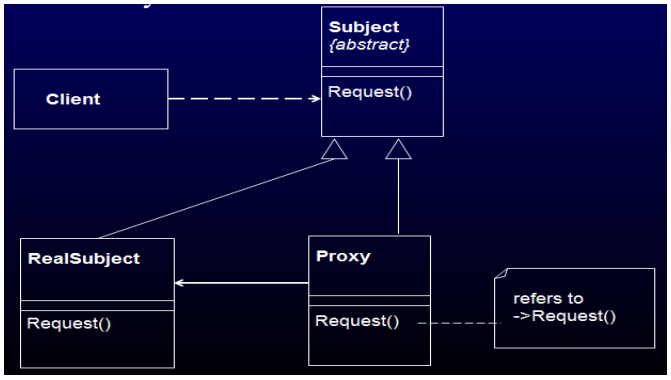


1. 首先在 View1 和 View2 中加上各自的构造函数，在构造函数中传入 Student，设置 this.Student = Student，然后将自己注册 Student.Register(this)
2. View3.Modify() -> Student.SetData() -> Student.Notify
3. Notify()中，for all Observer in ObserverList { Observer.update(); }
4. Observer.update -> Student.getXXX()，从 Student 中获取修改后的信息

b) 普通 Programming to Interfaces 有哪些手段？集合类型 Programming to Interfaces 有哪些手段？

1) 普通 1：代理模式

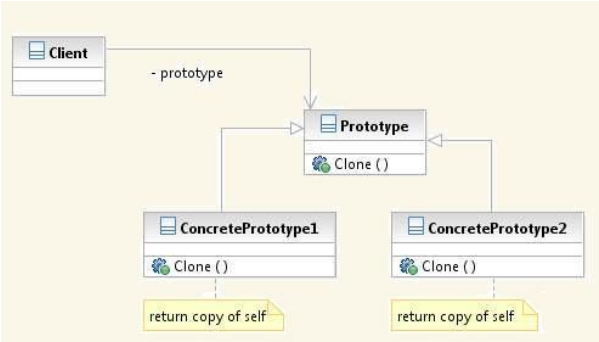
1. client 请求创建一个 subject，创建者创建一个代理交给 client，client 通过代理来访问实际的 subject（client -> 代理 -> subject）。为了能够让 client 在使用代理的时候感觉像是使用 subject 一样，需要实际 subject 和代理继承自同一个接口
2. 代理模式分为三种：远程 proxy（client -> proxy -> | -> real subject），proxy 在本地，client 访问 proxy，proxy 再通过网络 socket 访问实际的 subject；虚拟 proxy，先创建代理，在需要的时候再创建实际的 subject，这个方法用在创建实际 subject 非常消耗资源的情况下，用于进行性能的优化；保护 proxy，在 proxy 的代码中限制对实际 subject 的访问



3.
```
Client {
    private Subject subject;
    Client () {subject = new Factory().createSubject(); }
}
Factory {
    Subject createSubject() {
        Proxy proxy;
        RealSubject realSubject = new RealSubject();
        proxy.setRealSubject(realSubject);
        return proxy;
    }
}
```

2) 普通 2：原型模式

1. 原型模式要求对象自己创建自己（最重要的是初始化），而不是在 client 对对象手动地进行创建和初始化。client 有 Prototype 类型的引用，当它需要再创建一个对象的时候，可以调用 prototype->Clone()方法创建这个对象



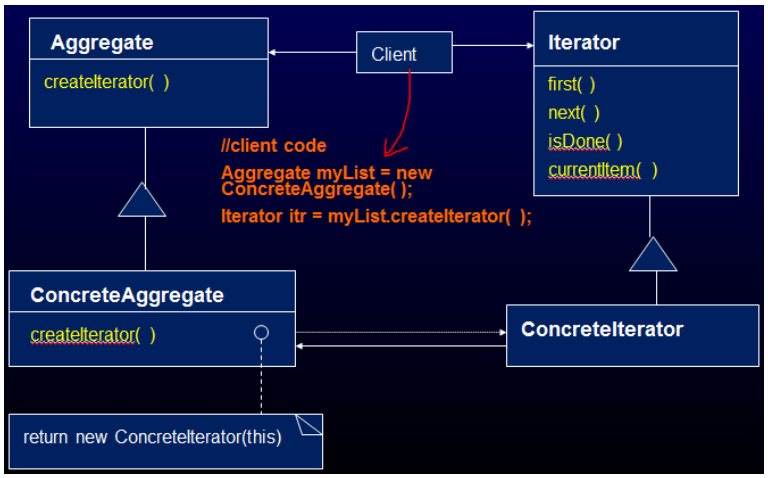
3) 集合：迭代器模式

1. 在迭代器模式中，为了实现多种不同的遍历方法，则将遍历对象从集合对象中独立出来，单独成为 Iterator 对象，这个对象承担遍历的职责。将 Iterator 做成接口，则实现这个接口的 ConcreteIterator 可以实现不同的遍历方法。Iterator 还需要包含它所访问的集合，以实现对集合的访问

```
Iterator {
    private Aggregate aggregate;
    Iterator(Aggregate agg) { aggregate = agg; }
    First(); Next(); IsDone(); CurrentItem();
}
```

2. Iterator 和 Aggregate 都是接口，但是在 ConcreteIterator 对象中，真正的 Aggregate 变量却是实现了 Aggregate 接口的对象。一个 ConcreteIterator 对应一个 ConcreteAggregate，用户只需要在创建对象的时候指定它们的具体类型，在以后的使用中都不会再讨论到具体类型
3. 因为这个模式的目的是实现 Iterator 独立于 Aggregate（Iterator 可以独立地变更而不会影响到 Aggregate），因此 Iterator 不能了解具体的 Aggregate 的信息，自然也不能够创建一个 ConcreteAggregate。在这个模式中，是由客户代码先创建集合类型，再由集合类型创建它自己的迭代器。如果集合类型需要修改自己的迭代器，则应在自己的 createIterator()中修改

```
ConcreteAggregate {
    iterator createIterator() {return new ConcreteInterator(this);}
}
```



c) OCP 的手段有哪些？【提示：不仅仅是继承】

1) 使用多态的方式，做一个继承树。此方案针对会发生修改但是并不严重的地方，让需要扩展的实体继承已经存在的实体
2) 使用装饰者模式，可以在不修改原来代码的情况下进行方法的扩展
3) 延迟绑定
    1. **继承**：使用继承、多态的方式进行动态绑定，在运行时根据对象的实际类型来执行其所实现的方法
    2. **运行时注册**：使用进行事件处理的运行时注册方式，当需要进行扩展时，就让扩展的方法监听某个事件，事件发生时这个方法就会被调用
    3. **配置文件**：使用配置文件进行启动时绑定，将需要修改、扩展的信息写在

配置文件中，通过解析配置文件来决定做什么事情

4. **构建更替**：如果需要修改、扩展，则在加载模块的时候使用修改/扩展的模块，进行加载时绑定。将变化的部分写在一个.dll 文件中，变化的时候直接更新.dll 文件

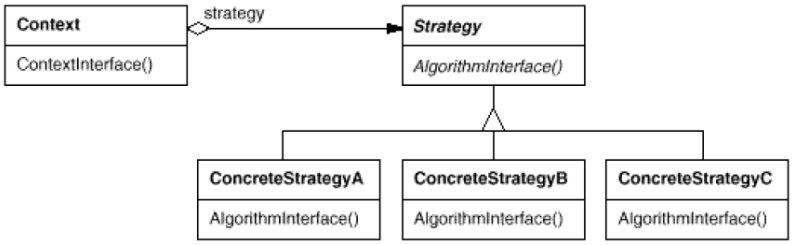
d) 一个模块的信息隐藏有哪两种基本类型，各自有哪些典型的处理手段？

1) 决策类型 1：**需求**，即一个模块的接口功能与模块内部的程序细节分离。

1. **外观模式**：将复杂的内部实现封装起来，并提供一个简单的调用接口。通常用在一个子系统中，作为这个子系统的外部接口，封装并实现了这个子系统对外提供的功能。用户不需要访问子系统内部，而只需要访问这个 Façade 对象即可，Façade 将用户与子系统解耦

2) 决策类型 2：**变化**，将要发生变化的程序部分独立出来，这样以后对这个部分的修改不会影响到其他部分。这里的变化可能是方法的变化、属性的变化、部分代码内容的变化。

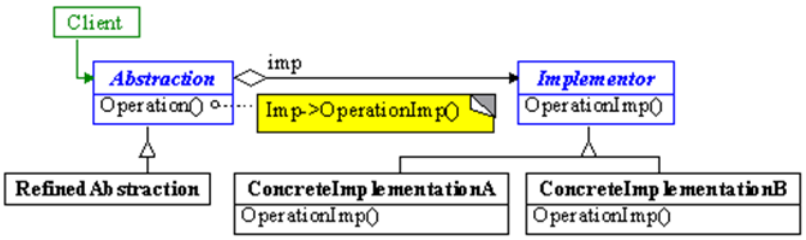
1. **策略模式**：将要变更的算法独立出来，按照 OCP 的方法将其封装起来成为一个对象，同时给这个算法建立一个继承树（为其设置一个接口，让所有这个算法的可能变更的版本实现这个接口）



2. **状态模式**：对象的行为根据状态的变化（属性的取值变化）而变化，将变化的状态独立出来做成一棵继承树，每一个实现的子类都实现了一种状态下的行为。原来的对象包含一个对状态对象的引用，表示当前的状态，一切行为都使用这个状态对象的行为。（需要由 Context 和 State 来处理状态的变化，不要由 Client 来处理）

变化的方法：（1）在 context 中调用一个 set 方法来设定状态；（2）在 ConcreteState 中设定状态，当前状态的差异性对象调用 set 来设定状态，同时将自己 destroy 掉（少见）

3. **桥接模式**：在抽象和实现需要独立变化的情况下，将抽象和实现做成两个不同的继承树。抽象接口和实现接口是两个完全不同的接口，实现接口一般包含很基本、原始的方法，而抽象接口包含的是基于原始方法实现的更加高级的方法。在抽象的接口中包含一个对实现对象的引用，抽象对象的方法中需要调用实现对象的方法（抽象接口的方法基于实现接口的方法），来完成实现的过程

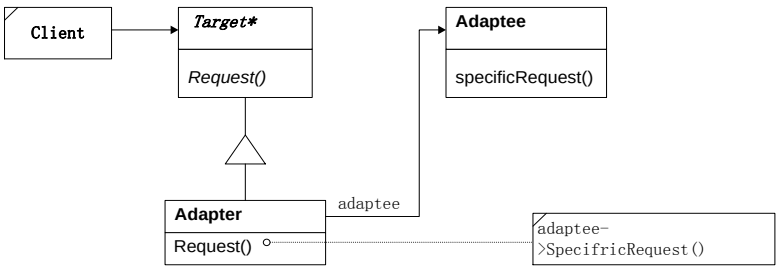


e) 实现共性与可变性有哪些手段？对给定的场景，给出共性与可变性的设计方案

1) **继承**：将共性交给父类，在子类中实现差异性，多态只能做到 N 选 1。将 N 个方案做成 N 个子类，在使用的时候使用其中一个子类即可
2) **聚合**：（即策略模式、状态模式、桥接模式，总体聚合出现差异性的策略）将共性交给总体，将差异性交给部分，可以做到 N 选 M。在共性和差异性问题的处理中，聚合是更好的方案
3) **运行时注册/事件机制**：这一部分与其他部分解耦，不直接发生程序调用
4) **配置式编程**：将可能变化的部分写在一个配置文件中，当要发生变化时，直接修改配置文件
5) **构件更替**，将变化的部分写在一个.dll 中，变化的时候直接更新.dll 文件

f) 在解决 De-Coupling 时，常常使用哪些 Indirection 的手段？对给定的场景，给出 Indirection 的解决方案

1) **依赖倒置**：如果抽象实体需要依赖于具体实体的话，那么为具体实体做一个接口，抽象实体可以依赖于这个接口，具体实体则实现这个接口
2) **外观模式**：Façade 是用户和子系统之间的一层间接，它封装子系统的内部实现并对用户提供访问接口，将这二者解耦
3) **代理模式**：Proxy 是用户和实际实体之间的一层间接，它负责转发用户的请求到实际实体，对实际实体进行访问控制，将这二者解耦
4) **适配器模式**：应用中使用了一个接口 Target，这个 Target 的一个实现需要使用到一个其他实现，但是那个实现与当前的接口不一致。让 Target 的实现实体聚合一个 Adaptee 的对象，做成对象 Adapter，当用户对 Adapter 有请求的时候，Adapter 便将请求转发给 Adaptee。同时，Adapter 还需要实现用户要求的但是在 Adaptee 没有实现的职责，不仅仅是转发请求。（client->Adapter->Adaptee）



5) **Event-Style 事件风格**：使用一个事件处理机制，其他模块可以向处理对象提供事件，也可以将自己的方法注册到某个事件，当这个事件发生之后，处理对象会调用注册到这个事件的所有方法，实现事件源和注册方法的解耦

g) MVC 与分层方式的区别【要具体到实现】

1) **分层**：禁止下层调用上层（只能调用自己下一层的方法），禁止跨层调用；下层返回上层的时候只能是普通的 Value Object 或者是值传递。也就是说，分层方式中，每一层只能了解并使用自己层和下一层
2) **MVC**：在 MVC 中，三者互相都用关系，view ⇔ model，view ⇔ control，control

-> model

1. view 拥有 model 的引用，view 先提前注册感兴趣的 model，当 model 发生变化通知 view 时，view 向 model 获取更新数据（只可以使用，不可以修改）（类似于 Observer 模式）
2. model 拥有 view 的引用，当 model 发生变化的时候需要通知 view，这个引用是 view 自己注册给 model 的
3. view 拥有 control 的引用，当用户发起事件的时候，view 把事件传递给相应的 control（如果 control 需要变化的话，则类似于策略模式）
4. control 拥有 model 和 view 的引用，它有两个作用：1）根据 view 传递的用户事件，交给相应的 model 进行处理，起到转发的作用（类似于 Controller），2）选择相应的 view 进行显示

3) 总结：分层需要遵守严格的规定，MVC 也有其相应的风格规定；MVC 中 view 需要在 model 中注册，model 和 view 之间需要有回调的关系；MVC 三者之间的关系比分层更为紧密；分层可以有 2~n 层，MVC 只能有 3 个部分

h) 对象的创建有哪些常见解决方法？【提示：这里要求常见解决方法，不是设计模式】（GRASP 模式中的高内聚、低耦合、创建者）

1) **创建者**：对于 A、B 两个对象，在以下情况下 A 创建 B 对象（在 A 中 new B()）：A 聚合了 B 的对象；A 包含了 B 的对象；A 记录了对 B 对象的引用；A 使用了 B 的对象；在 A 中包含初始化 B 对象的数据
2) **低耦合**：在 A 聚合、包含、记录、使用 B 的情况下，如果将创建 B 对象的职责赋予其他对象，则 A 需要与其他对象产生多余的耦合
3) **高内聚**：在 A 中包含初始化 B 的数据的情况下，此时 A 拥有创建 B 对象所需要的信息，是信息专家，根据高内聚的原则，A 应当承担创建 B 对象的职责