  \* 名詞解釋：  
  
      \* 設計的5個準則：  
  
         \* 用“美”的方式實現功能，是設計的價值  
  
            \* 簡潔性  
            \* 一致性（概念完整性）  
            \* 堅固性（高質量）  
         \* 設計複雜度＝事物複雜度+載體與事物的適配複雜度  
         \* 設計重在內部結構，不是在外在表現  
         \* 只有高層設計良好，底層設計才可能良好  
         \* 只有寫完兵測試代碼之後，才能算是完成了設計（敏捷視點）  
      \* 4+1 View:  
  
         \* logical view（邏輯視圖）：關注系統的邏輯結構和重要的設計機制，描述系統提供的功能和服務  
         \* development view（開發視圖）：關注系統的實現結構，描述系統開發的組織  
         \* process view（進程視圖）：關注系統的運行時表現，描述系統的並行進程組織  
         \* physical view（物理視圖）：關注系統的基礎設施，描述系統的部署與分佈  
         \* scenarios（場景視圖）：關注系統最為重要的需求，描述系統應該實現的場景與用例。  
      \* OO協作與協作設計：  
  
         \* 每個協作，無論大小，都是一個行為（行為一般是職責或者任務）  
         \* 面向對象的程序可以看做是一個以object為節點，通過關係連接的網絡  
         \* 協作是實際行為發生時在網絡中消息傳遞的模式  
         \* 協作分散在網絡中的各個對象中，不會單獨存在於一個地方  
      \* OO職責與職責分配：  
  
         \* 什麼是職責：職責是從需求來的，由體系結構加工之後，處理得到  
         \* 為什麼職責分配：為了滿足需求，實現模塊化，信息隱藏。手段是GRASP。  
      \* GRASP模式：General Responsibility Assignment Software Pattern  
  
         \* 低耦合  
         \* 高內聚  
         \* Expert  
         \* Creator  
         \* Controller  
         \* Polymorphism  
         \* Pure Fabrication（純虛構模式）  
         \* Indirection  
  
            \* pipe/event/share data styles  
            \* adapter pattern  
            \* proxy pattern  
            \* broker pattern  
            \* delegator pattern  
            \* mediator pattern  
            \* publish-subscribe or observer pattern    
  
   \* 軟件設計的審美標準有哪些，列舉已知的軟件設計方法與技術，並說明它們促進了哪些審美標準的達成？  
  
      \* 簡潔性  
      \* 一致性  
      \* 堅固性  
      \* 例如模塊化：促進了堅固性和簡潔性和一致性  
      \* 信息隱藏：促進了簡潔性和堅固性  
      \* 使用軟件體系結構風課編程，促進了一致性，堅固性  
   \* 設計的層次性：高層設計，中層設計，底層設計各自的出發點，主要關注因素（即那些審美要素），主要方法與技術和最終製品。  
  
      \* 底層設計：代碼設計，屏蔽程序中複雜數據結構與算法的實現；主要進行類型適配；出現原因：邏輯語句，適配類型有限，主要關注因素：可靠性，易讀性，性能。常見技術：防禦式編程，斷言式編程，表驅動編程，基於狀態機的編程，最終製品：數據結構或算法，單個的函數  
      \* 中層設計：模塊與類結構設計，隱藏程序片段的實現細節，通過接口對外交互；分解成盡可能獨立的片段；出現原因：模塊無法完全獨立，主要關注因素：簡潔性，一致性，易修改，可靠性，常見技術：低耦合，高內聚，信息隱藏，面向對象，最終製品：類和模塊  
      \* 高層設計：體系結構設計，關注總體設計和質量屬性，關注點：全部；總體功能組織；出現原因：大規模軟件被導入導出關係的侷限所依賴，方法技術：場景驅動， 最終製品：體系結構  
   \* 軟件體系結構風格：  
  
      \* 描述比較相關的風格  
      \* Physic unit-level style  
  
         \* Client-Server  
         \* Three-tier(N-tier)  
         \* Peer-to-Peer  
         \* Distributed System  
  
            \* Distributed object (middleware)  
            \* Distributed Resource (http)  
            \* Distributed Service (web service)  
      \* Process level  
  
         \* broadcast style  
         \* blackboard style  
         \* point -to-point style  
         \* pipe-filter style  
      \* Model level  
  
         \* implicit-invocation style  
  
            \* repository pull style(web blog)  
            \* blackboard push style(chat room)  
         \* pipe-and-filter style  
         \* call-and-return style:  
  
            \* main-program-and-subroutine  
            \* object-oriented  
            \* layer-style  
      \* 對給定的場景，判斷需要使用的風格  
   \* 職責分配與協作設計：  
  
      \* 協作設計（控制風格）的比較和場景判定  
  
         \* 集中式  
         \* 委託式  
         \* 分佈式  
      \* 對給定場景和要求的控制風格，根據GRASP模式，判定特定職責的分配  
      \* 根據分析類圖和體系結構模塊接口，建立基本的設計類圖。  
   \* 設計模式：  
  
      \* 設計模式部分的所有思考題  
  
         \* 有一個數據列表DataList，其基本類型是3維向量Three<x,y,z>，用java語言實現該數據列表的數據結構  
                                   public interface container()  
                                   {  
                                        public Iterator createIterator  
                                   }  
                                   public interface Iterator<Class T>  
                                   {  
                                        public  T first();  
                                        public T next();  
                                        public T currentItem();  
                                   }  
                                   public class DataList implements container  
                                   {  
                                             private ArrayList<Three<x,y,z>> data;  
                                             public Iterator createIterator()  
                                             {  
                                                  return data.iterator();  
                                             }  
                                             public void addData(Three<x,y,z> \_data)  
                                             {  
                                                   data.addObject(\_data);  
                                             }  
                                             public void removeData(There<x,y,z> \_data)  
                                             {  
                                                  data.removeObject(\_data);  
                                             }  
                                             public void removeData(int index)  
                                             {  
                                                  data.removeObjectAt(index);  
                                             }  
                                   }  
                                   public class MyIterator<Class T> implements Iterator  
                                   {  
                                             private ArrayList<T> bindingDataList;  
                                             private int currentCursor;  
                                             public MyIterator(ArrayList<T> data)  
                                             {  
                                                  bingdingDataList = data;  
                                                  currentCursor = 0;  
                                             }  
                                             public T first()  
                                             {  
                                                  if(!data.empty())  
                                                       return data.ObjectAt(0);  
                                                  return null;  
                                             }  
                                             public T next()  
                                             {  
                                                      currentCursor++;  
                                                       if(currentCursor<data.szie())  
                                                           return data.ObjectAt(currentCursor);  
                                                      currentCursor--;  
                                                      return null;   
                                             }  
                                             public T currentItem()  
                                             {  
                                                       if(!data.empty())  
                                                             return data.ObjectAt(currentCursor);  
                                                       return null;  
                                             }  
                                   }  
  
  
  
         \* 假設有3個外部對象A，B，C，分別對其x，y，z維度感興趣，希望訪問DataList在相應維度的數據並行處理，請定義其對外的數據接口，分別使用iterator與proxy  
  
            \* iterator:  
  
               \* CreateXiterator();  
               \* CreateYiterator();  
               \* CreateZiterator();  
               \* 剩下的參考上一題  
            \* proxy:  
  
               \* interface proxySubject  
               \* Class XProxyObject implements proxySubject  
               \* Class YProxyObject implements proxySubject  
               \* class ZproxyObject implements proxySubject  
               \* class DataLsit implements proxySubject  
               \* A,B,C分別創建對應的proxyObject就行了  
         \* 下面一道是使用Decorator的典型情況，看一下就行  
         \* 比較facade與controller(collaboration design)的異同  
  
            \* facade是為了將複雜接口隱藏，對上層提供簡單的接口，controller一般為了控制邏輯處理（出發點不一樣）  
            \* facade可以有有多個不同職責接口（介面），在委託式中，一般一個controller只有一個職責。  
            \* 相同之處在於它們都不實現自己的數據處理而僅僅負責轉發分配任務。  
         \* 在策略模式中，為什麼使用“聚合”而不是“關聯”關係？  
  
            \* 聚合的意思是總體擁有部分，關聯之間的關係是平等的，而在策略模式中，對象和策略之間的關係是對象擁有策略，而不是策略是對象的一種。  
         \* 如果一個對象集之間除了共性之外，有超過兩個的差異行為，該如何處理  
  
            \* 將差異行為抽出做成strategy，聚合在對象中  
         \* 如果一個對象集的部分行為組存在差異性，如何處理  
  
            \* 同上，但strategy中是一個行為組而不是一個行為  
         \* 如果一個對象集的部分屬性（以及依賴於這些屬性的方法）存在差異性，如何處理  
  
            \* 將這部分屬性方法抽出，做成協作對象聚合來做（這個不確定，上課似乎說過，不太記得了，求大神指導）  
         \* 如果一個對象集的一個行為需要協作對象來完成，但是它們的協作對象存在差異性，如何處理  
  
            \* 還是將協作對象聚合到對象中  
         \* 如果一個對象集的行為因為屬性的取值差異存在差異性，如何處理  
  
            \* 典型的State Pattern  
         \* 比較Strategy和State  
  
            \* Strategy是M of N ; State 是 1 of N  
            \* Strategy有對象來控制策略，State由對象特定狀態來確定策略  
         \* 使用Bridge Pattern實現一個畫圖程序的主體框架，PPT上例子有  
         \* 與真正自我實現的Event Style相比，Observer Pattern的不同在哪裡  
  
            \* Observer把EventRoute省去了，將功能放在Subject中了  
            \* Observer中subject只能被觀察一種Event了  
            \* 相比Event Style，Observer Pattern subject的功能變複雜了  
            \* 求補充....  
         \* 如果眾多Observer的接口不相同怎麼辦  
  
            \* 使用一個新的對象將它們的接口統一  
         \* 用Observer編寫程序（這個很簡單，我就不班門弄斧了）  
         \* SingleTon的解決問題（這個也很簡單）  
         \* 如果有其他類的實例創建都需要子類來決定怎麼辦  
  
            \* Factory Method ＋ Abstract Factory的用法，需要分情況討論  
         \* 如果多個其他類實例之間存在類型依賴怎麼辦  
  
            \* 典型的Abstract Factory的用法  
      \* 普通program to interfaces有哪些手段  
  
         \* 模塊化  
         \* 信息隱藏  
      \* 集合類型的program to interfaces有哪些手段  
  
         \* iterator pattern  
         \* proxy pattern  
         \* prototype pattern  
      \* ocp的手段有哪些？  
  
         \* open/closed principle  
  
            \* Dynamic Binding: (polymorphism)  
            \* Runtime registration: (event style)  
            \* Startup binding: (configuration file)  
      \* 一個模塊的信息隱藏有哪兩種類型，各自有哪些典型的處理手段  
  
         \* 對於可變性的處理  
         \* 外部表現和內部結構的區別  
      \* 實現共性和可變性有哪些手段，對給定的場景，給出共性和可變性的設計方案。  
  
         \* 聚合  
         \* 繼承  
      \* 在解決 De-coupling 時，常常使用哪些 Indirection 的解決方案。  
  
         \* proxy  
         \* bridge  
      \* MVC 與 分層方式的區別（要具體的實現）  
      \* 對象的創建有哪些常見的解決方案？   
  
         \* creator  
         \* 高內聚  
         \* 低耦合