考试科目：软件体系架构与设计模式
考试形式：开卷
考试时间：2015 年秋
考试时长：120 分钟

一、 (共 25 分) When discussing the topic of design pattern, there are four types of design patterns that can be applied to various layers of software or program. Answer the following questions about the design patterns in software design:

(a) （5 分） draw a layered model to show the four types/layers of design patterns and give a few examples for each layer;

答:

Architectural Pattern	Client/Server, B/S, Web Service, SOA, ... （2 分）
Component Design	Singleton, Adapator, Abstract Factory, ... （1 分）
Class Design	Abstract Class, Virtual Class, Interface, ... （1 分）
Data Structure	Default data tyeps: int, float, string, ...; User-defined structures: linked list, tree, graph, ... （1 分）

(b) （5 分） list and explain the three aspects that describing the design pattern at component level;

答:

Context: describe the work scenario or functional requirement that the design pattern can be applied to; （1 分）

Problem: the design problem that repeatly occurs in the work scenario; （2 分）

Solution: a design that provides a solution to the described design problem. （2 分）

(c) （5 分） describe the difference and/or relationship between the design patterns in various layers;

答:

Data Structure provides a basic unit to form a class or component design （2 分）; class design provides a basic block to the component design （2 分）; various architectural pattern may use certain types of component design patterns. （1 分）

(d) （5 分） in a software design in which the distributed computing system is selected as the architectural pattern, please list a few component design patterns that are most likely to be used for this software architecture, and explain why;

答:

Broker, Connector, Singleton, Proxy, Abstract Factory. （1 分）

Broker, Connector and Proxy provide the needed features for communication architecture used in the distributed system; （2 分）

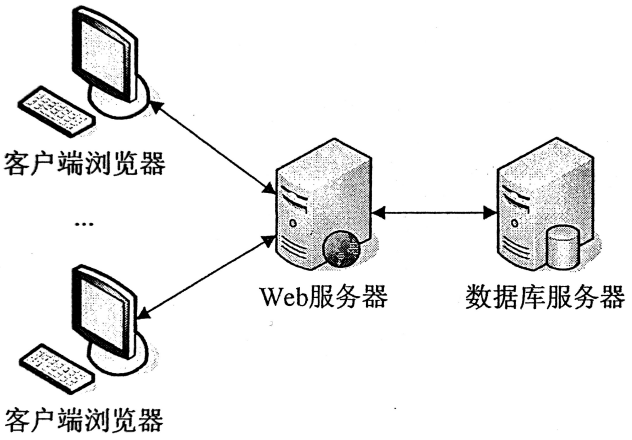
Abstract Factory provides an efficient way to create and manage a group of components/objects that will be dynamically configured in execution. （2 分）

(e) （5 分） comment on the advantages and disadvantages of component design patterns.

答:
Advantages: design reuse, ease software development, better software quality, facilitate software changes; （3 分）
Disadvantages: need learning curve, higher requirement on software developer. （2 分）

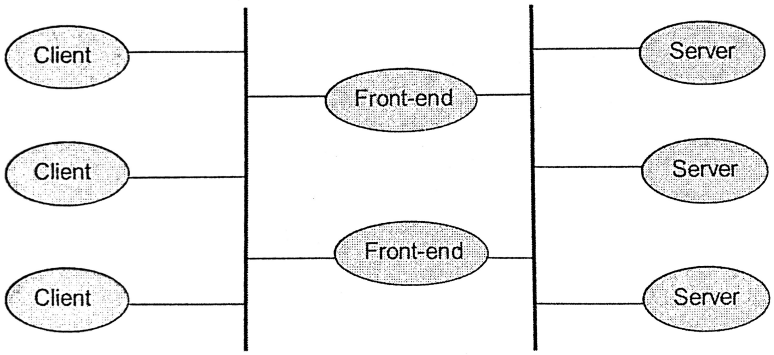
二、 (共 25 分) 下图使一个典型的 B/S 架构图，只有一台前端机（Web 服务器）适用于访问量有限的情况。后端服务器与本地数据库相连，通过 ODBC 或 JDBC 实现数据读取。

1. (7 分) 当前端用户访问量剧增，1 台前端机不能支持时，该如何改进架构设计？画出改进后的前端架构图。
2. (6 分) 如果后端的数据库服务器不再是本地设置，而是在异地通过网线连接，且 Web 服务器与数据库服务器运行在不同平台上（不同的硬件系统、不同的 O/S），该如何改进后端架构设计？阐述理由。
3. (6 分) 如果采用 Web Service 架构，画出新的架构图并标注三方。
4. (6 分) 在你的改进设计中，Web Service 架构三方之间的通信和数据交换采用何种协议或技术?



答:

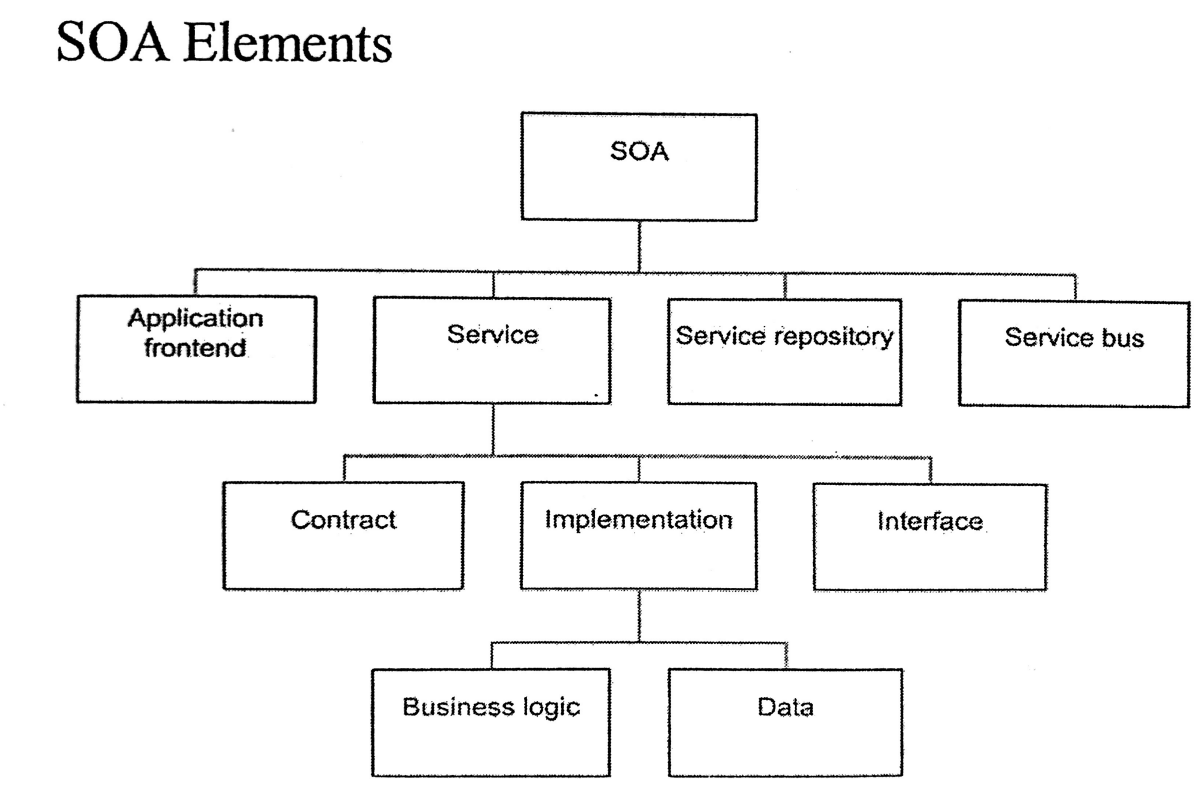
1. (7 分) 可采用多重架构（multi-tier）多个前端机的解决方案（3 分）。改进后的前端架构图如下（4 分）:



2. (6 分) 后端架构可采用 Web Service 或分布式通讯中间件架构（如 CORBA），理由如下：
   (1) 基于网络的远程数据交换（3 分）
   (2) 跨平台性（3 分）
3. (6 分) (1) 画出 Web Service 架构图（3 分）
   (2) 标注服务需求方（1 分），服务提供方（1 分），注册发布方（1 分）
4. (6 分) 服务提供方 <—> 注册发布方：WDSL 或 UDDI（2 分）
   服务需求方 <—> 注册发布方：WDSL 或 UDDI（2 分）
   服务提供方 <—> 服务提供方：SOAP（2 分）

三、 (共 25 分) 下图为面向服务架构（SOA）的结构示意图，它包括了 SOA 架构的主要元素，请回答如下问题：

1. (6 分) 与传统的 OOA 设计比较，SOA 的设计特点有何不同？
2. (7 分) SOA 架构的基本单元或模型是什么？它包括哪些要素？
3. (6 分) SOA 架构强调的现代软件架构设计的一个重要的基本原则是什么？是如何实现这一原则的？
4. (6 分) SOA 架构与 00 技术是如何发生关联的？



答:

1. (6 分) 与传统的 OOA 设计比较，SOA 的设计特点有何不同？

(1) SOA 架构基于服务模型（service model），而 OOA 架构基于实体（object）(3 分)

(2) SOA 架构向上易于支持或与商务模式和业务流程集成，OOA 向下强调开发平台，软件设计，实现技术（3 分）

2. (7 分) SOA 架构的基本单元或模型是什么？它包括哪些要素?

   Service Model (2 分)，Service Contract (2 分)，Service Reusability (1 分)，Service Discovery (2 分)

3. (6 分) SOA 架构强调的现代软件架构设计的一个重要的基本原则是什么？是如何实现的?

   (1) 一个重要基本原则：模型（model）与实体（instance）分离，或界面（interface）与实现（implementation）分离（3 分）

   (2) 实现方式：建立抽象模型（或服务模型）(1 分)，用 XML 语言描述模型或界面 (1 分)，不同平台的软件实现（implementation）支持同一界面（1 分）

4. (6 分) SOA 架构与 00 技术是如何发生关联的?

   SOA 架构完成服务建模和架构设计后（3 分），具体的软件模块的实现可采用 00 编程技术，这是发生关联点的地方（3 分）

四、 (共 25 分) In a software component design, there is a legacy class OldMgr that already created and used, with the following code pieces:

```
class OldMgr { // legacy manager class
    private:
        OldMgr* mgrHandle;
        void OldMgr();
    Public:
        OldMgr* GetOldMgr();
};

OldMgr* OldMgr::GetOldMgr(): public
{
    if (mgrHandle == NULL)
        mgrHandle = new OldMgr();
    return mgrHandle;
}
```

Now a new manager class NewMgr is required to be designed with the following functions:

Requirement 1) the NewMgr class can and can only create one instance anytime and anywhere when it gets instantiated;

Requirement 2) the NewMgr class shall have a public function NewMgr::GetNewMgr() that allows other classes to call it and to get the pointer/handle to the NewMgr instance;

Requirement 3) when NewMgr::GetNewMgr() is called, actually the OldMgr::GetOldMgr() function gets called to return the pointer/handle to the sole manager instance.

Based on above design requirements, answer the following questions:

1. (3 分) What design pattern does the OldMgr class use?
2. (6 分) How does the OldMgr class ensure that one and only one instance of the manager class is created?
3. (3 分) What design pattern shall be selected for the NewMgr class to enforce the Requirement 1)?

4. (3 分) What design pattern shall be used for designing the NewMgr class so that Requirement 2) is supported?
5. (5 分) Write the code of NewMgr class to show your design.
6. (5 分) Write the code of NewMgr::GetNewMgr() function.

答:

1. Singleton
2. (1) the constructor of OldMgr is declared as Private to prevent other objects call its constructor to instantiate is; (2 分)
   (2) the OldMgr class provides a public function OldMgr::GetOldMgr() to return the pointer or handle of the manager instance; (2 分)
   (3) within the body of OldMgr::GetOldMgr() function, if the manager instance is not created yet, create and save it; otherwise simply return the pointer/handle. (2 分)
3. Singleton
4. Adapter
5.

```
NewMgr: public AbstractMgr {
    private:                              //  (2分)
        OldMgr* old_mgr;                  //  (1分)
        void NewMgr();                    //  (1分)
    public:
        AbstractMgr* GetNewMgr();         //  (1分)
};

AbstractMgr* NewMgr::GetNewMgr(){
    if (old_mgr != Null)                                  //  (1分)
        return old_mgr -> GetOldMgr();                    //  (2分)
    else {
        cout << "Error: no manager object existed!" << endl;
        exit (1);                                         //  (2分)
    }
}
```