Modern Method Software Engineering

Home work 4

Sike Huang	**850414-A394**	**sike.huang@gmail.com**	**0762320173**
Shanbo Li	**840810-A478**	**shanboli@Gmail.com**	**0704646157**

2007-10-07

Question 1: Consider the classification of Design goals (given in Section 6.4.2). Assign one or more categories of design goals to each of following goals.

1. User must be given feedback within one second after they issue any command.

Answer:

Response time (Performance)

2. The ticketdistributor must be able to issue train tickets, even in the event of network failure.

Answer:

Fault tolerance (Dependability)

3. The housing of ticketdistributor must allow for new buttons to be installed in case the enumber of fares increases.

Answer:

Extensibility (Maintance)

4. The atm must withstand dictionary attacks (i.e. users attempting to discover an identification number by systematic trials).

Answer:

Security (Dependability)

Question 2: Consider closed and open architectures (Fig 6-10 and Fig 6-11).

1. Compare design goals which are achieved and which could be difficult to meet in each (use bullets, first analysis design goals for Open and then similarly discuss the same for closed architecture).

Answer:

To open architecture, the main design goal is **Runtime efficiency**. Any layer can invoke operations from any layers below. In general, the openness of the architecture allows developers to bypass the higher layers to address performance bottlenecks. But an open architecture may very complex, so it is not easy to change a layer without touch other layer. And it leads to make it difficult to maintain the architecture.

To closed architecture, the main design goals are **High maintainability** and **flexibility**. Any layer can only invoke operations from the immediate layer below. It is very easy to change a layer on closed architecture. But for one layer can only access the layer below, there may be some bottlenecks that cannot be avoid.

2. Under which scenario (set of design goals) you will consider developing an Open architecture and under which developing a Closed architecture.

Answer:

Open architecture

Desktop application

Design goals:

- Operating system independent (java)
- Desktop application
- Based on Java Swing
- High performance required
- No database needed

Closed architecture

A website application

Design goals:

- Interactive with users
- Database involved
- Database changeable
- Web layer art engineer do not know database acknowledgement

Question 3: Consider Repository, Model/View/Controller, Client-Server, Peer-to-Peer, n-Tier, and 'Pipe and filter' architectural styles (Section 6.3.5). Discuss how these architectures affect the following design goals. I) Extensibility II) Response Time III) Modifiability IV) Access Control V) Maintenance cost VI) Adaptability.

1. Consider a concrete example first (preferably a single example) and then try to decompose into above mentioned architectural styles. And then discuss how architecture affects the mentioned design goals. (Use bullets against each design goal, better create a table comparing architectures with respect to design goals)

Answer:

Let's consider developing a large-scale online game, such as World of Warcraft. First of all, we decompose the whole game into some subsystems, where each subsystem utilizes certain design pattern philosophy.

Gaming data management system: data is crucially important in a game, and the data has to be kept in a unified manner, so **repository** architectural style is chosen. We set up a central repository, or rather a database to hold all users' data as well as those data used to construct the scenarios and behaviors in the game.

Interactive graphical user interface: though the game itself is unique, the same data can form different view from user's aspect, by using **model-view-control** architectural style, we are able to render various user views in respect of the gaming environment, and separate the domain model and controller infrastructure as well.

Between end user client and gaming server: it's natural to have **client-server** architectural style, where every client communicates with a central gaming server to exchange data and messages. So the central gaming server acts as a master and leads the game.

Among end user clients: if one player is communicating with another player, it's clumsy that every message is delivered and forwarded by server, in such case, the **peer-to-peer** architectural style will connect the player directly, and significantly relieve the server load.

Gaming system as a whole: a game such as World of Warcraft is very complex, and the software system to handle this complexity has to consider many things, one after another. The **n-tier** architectural style supports design based on increasing levels of abstraction, and it allows implementers to partition a complex problem into a sequence of incremental steps.

Events processors in virtual world: there are numerous events in a game, events can happen in any order at any time, the combination of events can be regarded as another event as well, under such condition, **pipe and filter** architectural style suits the best, we can define event processor like a filter, and set up event flow channel as a pipe, then it's possible to deal with no matter what kinds of events.

	Repository	MVC	CS
Extensibility	Moderate, new functionalities can be added to other subsystems based on accessing central repository.	Good, controller, model and view are independent from each other, easy to change.	Poor, it's possible to add new functionalities in server, but it will affect the clients.
Response Time	Poor, central repository can soon become the bottleneck.	Moderate, controller takes care of all transactions, it may overload.	Poor, too many clients will dramatically decrease the performance.
Modifiability	Moderate, adding new services in the form of additional subsystems.	Good, easy to add new functionalities to controller.	Moderate, the update in server will propagate to its entire clients.
Access Control	Good, data can be strongly protected in the central repository.	Moderate, controller has to take care of security issues.	Good, actions are centralized in one place.
Maintenance cost	Low, since data is most important factor, central control of data is feasible.	Low, bugs usually lies in the controller.	Normal, server must be updated to balance the load.
Adaptability	Poor, different application has various data types to be handled.	Moderate, domain model can be changed.	Moderate, clients only need to know about the specific domain server.

Table 3-1 Architectural Styles and Design Goals

	P2P	n-Tier	Pipe&Filter
Extensibility	Poor, any changes has to be distributed.	Good, new functions are added to its corresponding tier.	Good, pipes can be added at run time.
Response Time	Fast, direct link between clients, no middleman.	Poor, elaboration of work.	Fast, all filters are processes that run at the same time.
Modifiability	Poor, any changes has to be distributed.	Good, changes in one tier only affect adjacent layers.	Good, filters are independent to each other.
Access Control	Poor, no central supervision.	Poor, control must be spread out through all tiers.	Poor, nothing prevents adding pipes or filters.
Maintenance	High, distributed problem is hard to find and solve.	Low, only one or few ties may have to fix.	Low, pipes and filters are flexible to fix.
Adaptability	Good, peer is not blind to certain application domain.	Good, ties can be adapted to different domain.	Good, domain independency.

Table 3-2 Architectural Styles and Design Goals (Cont.)

2. Make sets of design goals (not just limited to above mentioned design goals, consult section 6.4.2) for each architectural style that could serve as guideline for choosing certain system architecture.

Answer:

Repository: Robustness, Reliability, Availability, Security, Administration cost

Repositories are well suited for applications with constantly changing, complex data-processing tasks, once a central repository is well defined, we can easily add new services in the form of additional subsystems.

Mode/View/Controller: Maintenance cost, Extensibility, Modifiability, Portability

The most important of MVC is the ability to have multiple views that rely upon a single model, with MVC, it doesn't matter if the user wants a Flash interface or a WAP one, since the same model can handle either. Because the model returns data without applying any formatting, the same components can be used and called for use with any interface, and since model is self-contained and separate from the controller and view, it's much less painful to change your data layer or business rules.

Client-Server: Adaptability, Maintenance cost, Update cost, Security

A client-server architecture enables the roles and responsibilities of a computing system to be distributed among several independent computers. It's possible to replace, upgrade a server while its clients remain both unaware and unaffected. Since all the data are stored on the servers, which have far greater security controls than most clients, servers can better control access and resources to guarantee appropriate permissions to clients, and updates to those data are far easier to administer.

Peer-to-Peer: Memory, Fault tolerance,

In p2p, all clients are able to provide resources, including bandwidth, storage space, and computing power, thus as node arrive and demand on the system increases, the total capacity of the system also increases, and the distributed nature of peer-to-peer architecture also increases robustness in case of failures by replicating data over multiple peers.

n-Tier: Modifiability, Adapterbility, Portability

n-Tier divided the application into logically isolated pieces, therefore ending or reducing tight coupling between the user interface, business logic and database. It's pretty easy to change or add to your application new functionalities, without breaking or recompiling the entire client-side code, and a change in one tier or layer does not affect the rest of it.

Pipe and Filter: Extensibility, Modifiability, Readability, Throughput

Pipe and filter systems allow the designer to understand the overall input/output behavior of a system as a simple composition of the behaviors of the individual filters. Second, they support reuse: any two filters can be hooked together. Third, systems can be easily maintained and enhanced: new filters can be added to existing systems and old filters can be replaced by improved ones. Finally, they naturally support concurrent execution.

Question 4: Consider eBay auction. Customers browse and view products and make an offer before end-date of given auction or buy theatrical on 'But-it now price'. Users are also registered with system and create profiles etc. etc. ….. System registers buyers, displays them the items active in auction etc. etc. …. You can check eBay's process. Formalize the general usage of eBay in terms of actors you can identify. And give a very brief usage scenario for each identified actor. (These will form assumptions for your answer)

Your task is to design an access control policy or discuss already in-use access control policy on eBay.

Answer:

Scenario name	**UserRegister**
Participating actor instances	Shanbo: seller
Flow of events	1. Shanbo has some secondhand book and want to sale them. 2. He thought it should be a good way to sale the books via internet. He wants to sale books with eBay. 3. To be a seller, he has to register first. He registers and applies to sale books. 4. Now Shanbo can start to sale books on ebay.

Table 4-1 UserRegister Scenario

Scenario name	**AddItem**
Participating actor instances	Shanbo: seller
Flow of events	1. Shanbo add some books to eBay. 2. He sets the end-date and buy it now price.

Table 4-2 AddItem Scenario

Scenario name	**BuyItem**
Participating actor instances	Sike: buyer
Flow of events	1. Sike needs a book named OOSE 2. He wants to buy it from eBay so that he can get a low price. 3. He searches the book and found a seller named Shanbo gives a very low price. 4. He chooses buy-it-now, before he could continue, eBay asks him to login first. 5. He logins with his account, and continues on his shopping. 6. He pay the money to PayPal. Input his address and wait for the book at home.

Table 4-3 BuyItem Scenario

Scenario name	**ItemDeliver**
Participating actor instances	Shanbo: seller Sike: buyer
Flow of events	1. eBay notifies Shanbo that a buyer want to buy his book and already transfer money to PayPal. 2. Shanbo deliver the book to Sike. And wait for Sike's confirm so that he can get the money from PayPal.

Table 4-4 ItemDeliver Scenario

Scenario name	**ConfirmItem**
Participating actor instances	Sike: buyer Shanbo: seller
Flow of events	1. Sike received the book which post from Shanbo 2. Sike satisfies with the book and confirms that he has received the book to eBay. 3. PayPal transfers the money to Shanbo's bank account.

Table 4-5 ConfirmItem Scenario

Our access control policy:

My goal is that all access to any pages must to be authorized. Without authorization user can not access the certain page. As the MVC model shown below, we stored users with role-control in the database. Each user has a password. The password has to be stored with encryption, so that even the database manager cannot see the user's password.

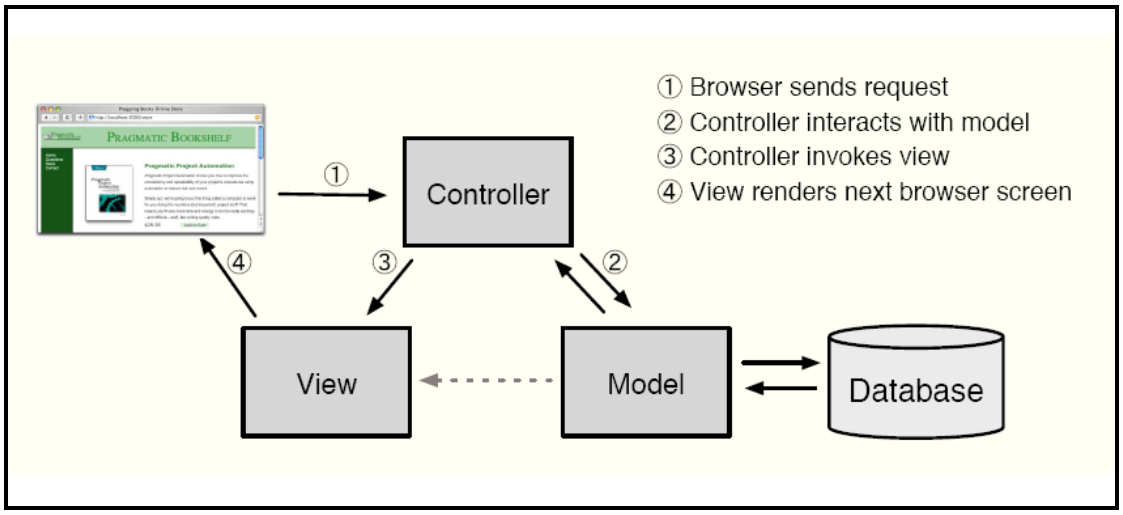

Figure 4-1 MVC Model

When a user tries to login, the controller will encrypt the password which he just inputted. And compare the encrypt password with the password in database. If it is not correct, the user will be rejected. Otherwise, if the password is correct, his name and role will be stored in his cookie. And every page has a filer. Before displays the page's content, the filter will check if the user's role is allowed to access this page. Figure 4-1 shows this issue of a normal role user.

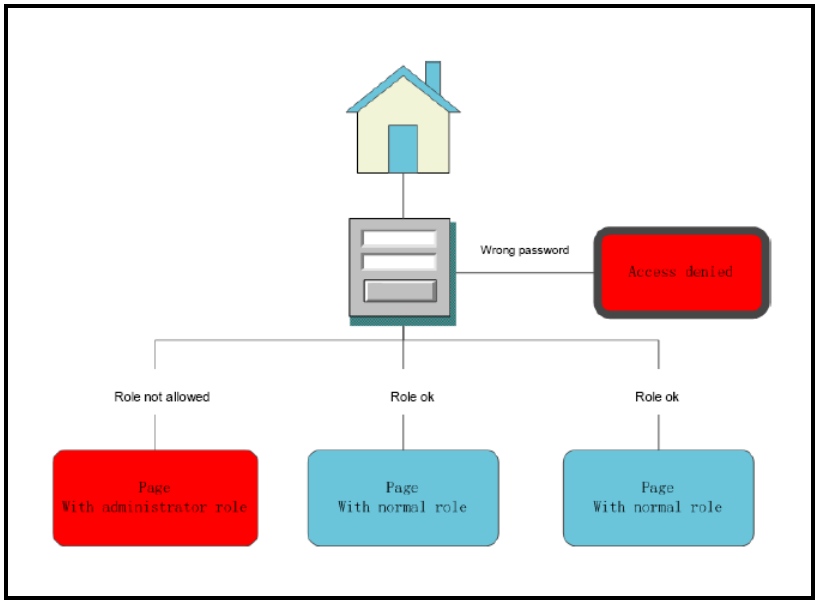

Figure 4-1 Access Control

Question 5: List the factors due to which you can't describe boundary use-cases in Requirements elicitation or Requirement Analysis phase?

Answer:

It is normal that boundary use cases are not described during requirement elicitation/analysis, or rather they are considered separately from the common use cases. For example, many system administration functions, such as registering and deleting users, managing access control, can be derived from the daily user requirements, whilst, many other functions are consequences of design decisions, say, cache sizes, location of database/backup servers, which are not of requirement decisions.

Question 6: Tetris field comprises of a collection of stones, some representing empty fields, and some representing parts of a particular Tetris block, whenever a Tetris block lands, it dissembles into its individual stones, which are considered independent from then on. It makes sense, therefore, to represent each stone as an individual object in an object-oriented Tetris application. Doing so naively results in a large amount of objects being allocated, deleted, and moved all the time. What design pattern can be applied for optimization, taking advantage of the fact that the stones are really very similar?

Answer:

To draw the stones we can use *Factory Method* pattern.

The factory method pattern is an object-oriented design pattern. Like other creational patterns, it deals with the problem of creating objects (products) without specifying the exact class of object that will be created. The factory method design pattern handles this problem by defining a separate method for creating the objects, which subclasses can then override to specify the derived type of product that will be created.

So**, Factory method pattern** can be used to draw different style of stones. We can introduce new identifiers for new kinds of Shapes, or we can associate existing identifiers with different shape objects. Template method pattern can be applied for optimization

To control the movement we can use *Template Method* pattern

A template method defines the program skeleton of an algorithm. The algorithm itself is made abstract, and the subclasses override the abstract methods to provide concrete behavior.

Using Template method for class MoveController, which is the base class of LeftKeyController, RightKeyController, UpKeyController and DropController. For handling any movement, it has to perform the following actions: erase(), action(), draw(). Although different event has different action(), the three steps have to be done in the same order, that's why Template Method is used here.

Question 7: What design pattern(s) will you use if you are supposed to realize following commands in your own developed UNIX shell:

$ cat apple.txt | wc | mail -s "The count" nobody@december.com

Answer:

The commands above utilizes pipe-filter architectural style, it provides a structure for systems, having components that process a stream of data (filters) and connections that transmit data between adjacent components (pipes). We can use **decorator pattern** as an alternative or related to pipes-and-filters, as shown in the UML class diagram below. So we would create a *Decorator* as an abstract class that takes a *Command* interface as a parameter in its constructor and will itself implement the *Command* interface. In conclusion, the same problem can be implemented by using both Pipes-and-Filters architectural pattern and Decorator pattern. The basic idea is the dynamic arrangement of filters on one side and dynamic pluggablility of responsibilities (wrappers) on the other side. One can see how pipes-and-filters with sequential processing maps to the decorator pattern exactly. So they may be related patterns in a sense that in Pipes-and-Filters, pipes are stateless and serve as conduits for moving streams of data across multiple filters, while filters are stream modifiers which process the input streams and send the resulting output stream to the pipe of another filter, and in the Decorator pattern we can see decorators as filters.

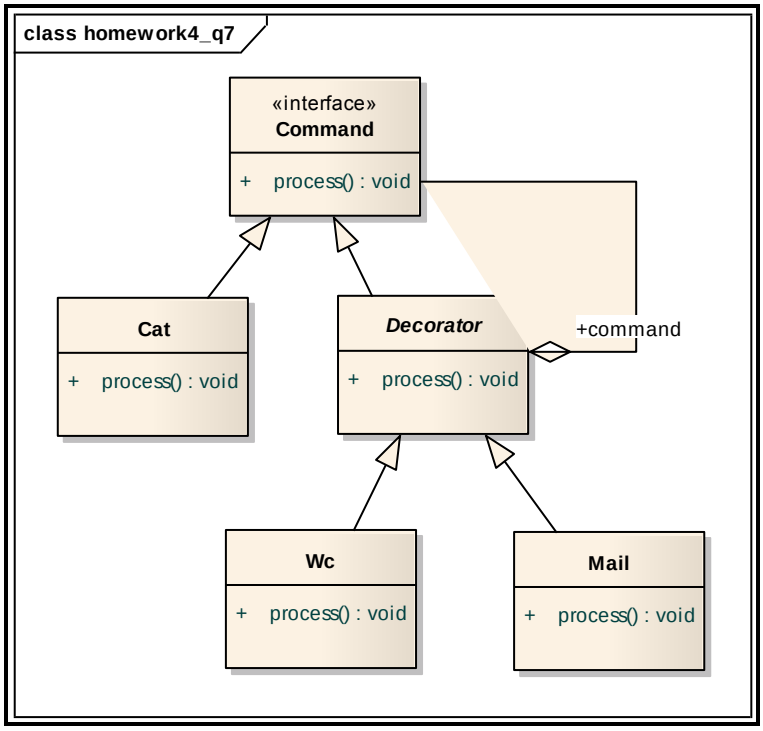

Figure 7-1 Decorator Pattern

Question 8: List at least 3 general dangers and benefits of using design patterns.

Answer：

Benefits：

- Reuse the core of the solution
  - It is just why we need design patterns.
- Make the interfacing between many modules or classes more manageable.
  - Like Façade Pattern.
- Reduce the Complexity of Models
  - Like Composite pattern

Dangers:

- Functionality may lost in lower level of classes
  - Like Façade Pattern
- Design pattern may lead to more class generate, that means more objects will be generated at run time, which consume more memory resources to some degree and will slightly slow down the running speed.
- Some pattern novices understand Singleton first and believe that since it has been given a pattern name they should use it more often.
- Sometimes they indicate a language deficency. Many of the GoF patterns are rarely applied in

Question 9: Develop a recursive composite design pattern and represent it using UML Class diagram. (Bonus Point 02)

Answer:

The intent of recursive composite design pattern is to compose objects into tree structures to present part-whole hierarchies, and treat individual objects and compositions of objects uniformly.

An obvious and concrete example is how the files and directories are treated in the operation systems:

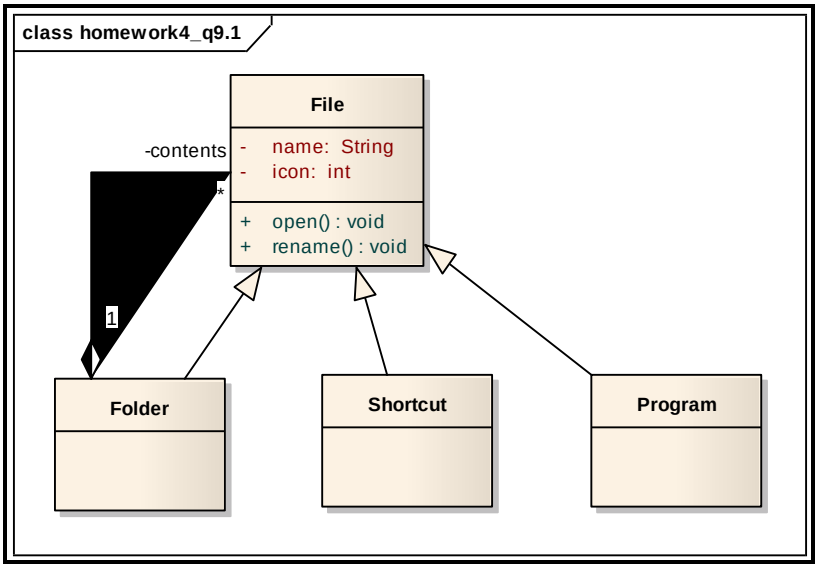

Figure 9-1 UML Class Diagram for Files and Folders

We can abstract the idea above into the recursive composite design pattern depicted in the UML class diagram below:

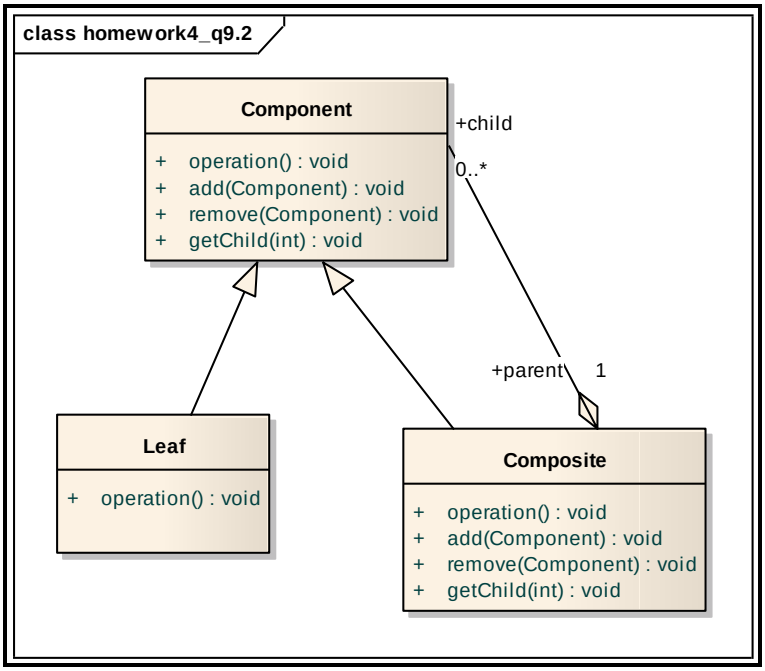

Figure 9-2 Recursive Composite Pattern

Component is the abstraction for all components, including composite ones, it declares the interface for objects in the composition and implements default behavior for the interface common to all classes, as appropriate.

Leaf represents leaf objects in the composition and implements all Component methods.

Composite represents a composite Component i.e. component having children, and implements methods to manipulate children.

Question 10: Based on the problem description for "Advert Consultancy Inc" (Introduced in Question 5 of Homework 2), and you results from homework 2 and homework 3, and the examples starting on pages 288, 337 and 380 of the textbook, Develop a system design and object design document which includes:

1. Subsystem decomposition structure (using class diagrams).

Answer:

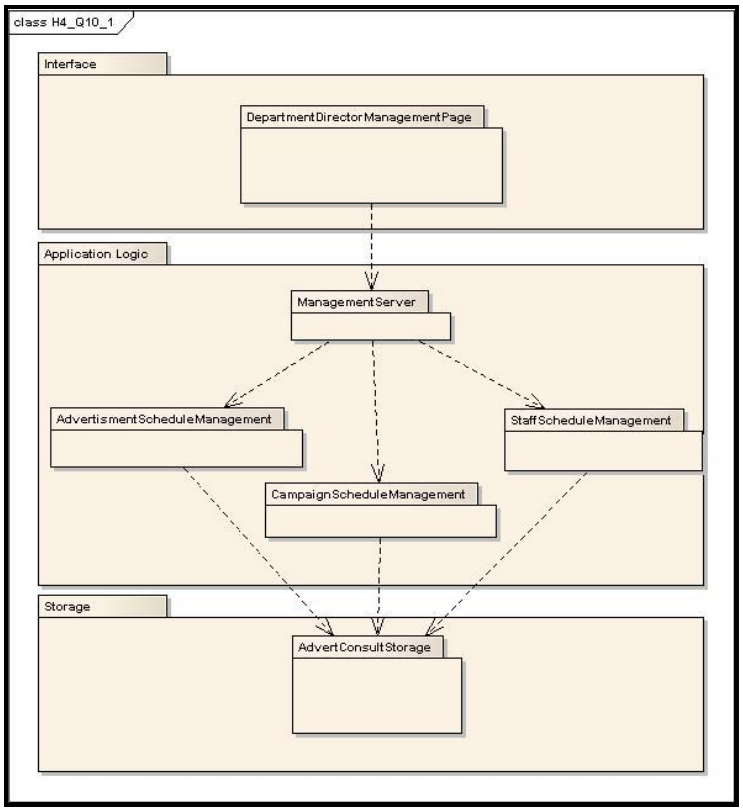

Figure 10-1 AdvertConsult system decomposition campaign management part (UML class diagram)

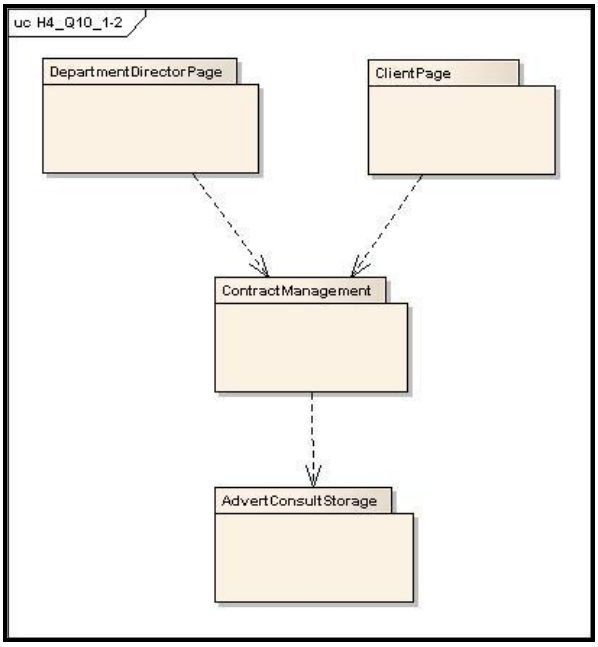

Figure 10-2 AdvertConsult system contract management part (UML class diagram)

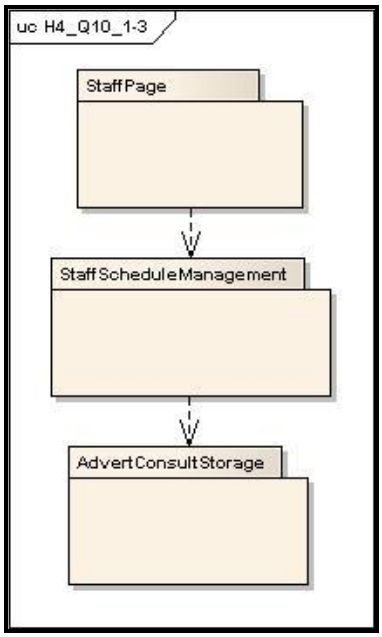

Figure 10-3 AdvertConsult system StaffSchedule part (UML class diagram)

2. Mapping subsystems to processors and components - hardware/software mapping (using UML deployment diagrams).

Answer:

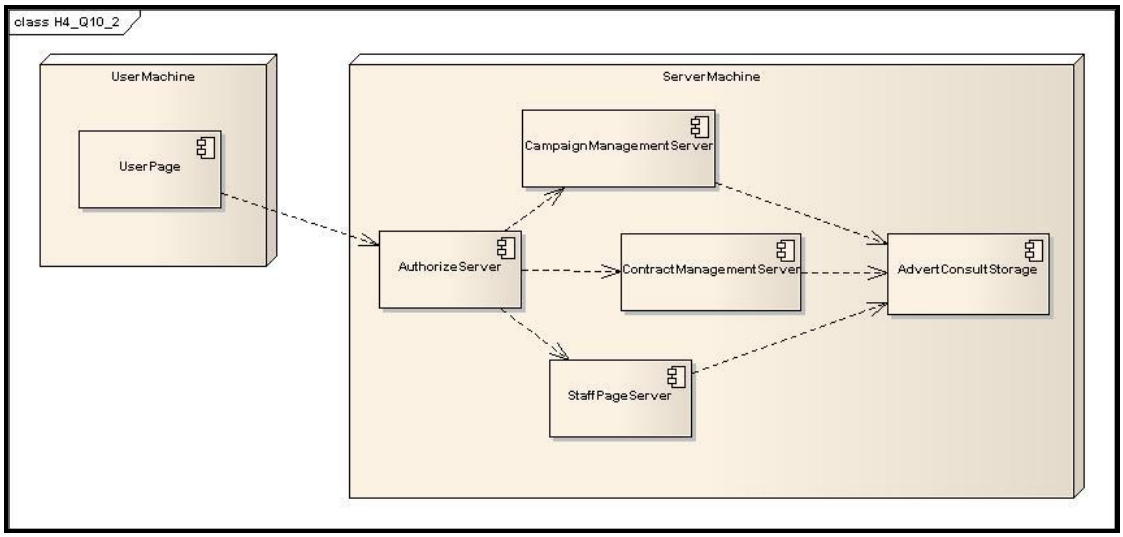

Figure 10-4 hardware/software mapping

3. Persistent storage solution for the problem.

Answer:

In database there are several tables. A table named ***Users*** contains all user information, including their name, password, role and other information which may needed. Stuffs' current states are also in the table of *Users*. Managers, staffs, clients are all items of table of Users. This table is used for login and role authorize. Table of Users also shows the stuffs' current state and clients' information. A table named ***campaigns*** stores campaigns' information. A table named ***contracts*** stores contracts' information.

4. Access control, global control flow and boundary conditions.

Answer:

Access control

Objects Actors	Campaign	Contract	Campaign Schedule	Advert Schedule	Staff Schedule
Department Director	<<create>> addNew modify	<<create>> addNew modify	<<create>> addNew modify	<<create>> addNew modify	<<create>> addNew modify
Staff			view	view	view
Board of Governor	view	view	view	view	view
Client	view requestModify	view requestModify	View requestModify	View requestModify	View requestModify

Table 10-1 Access control

Global control flow

We use Ruby on Rails to realize the whole system, the global control flow shows as the figure fellow:

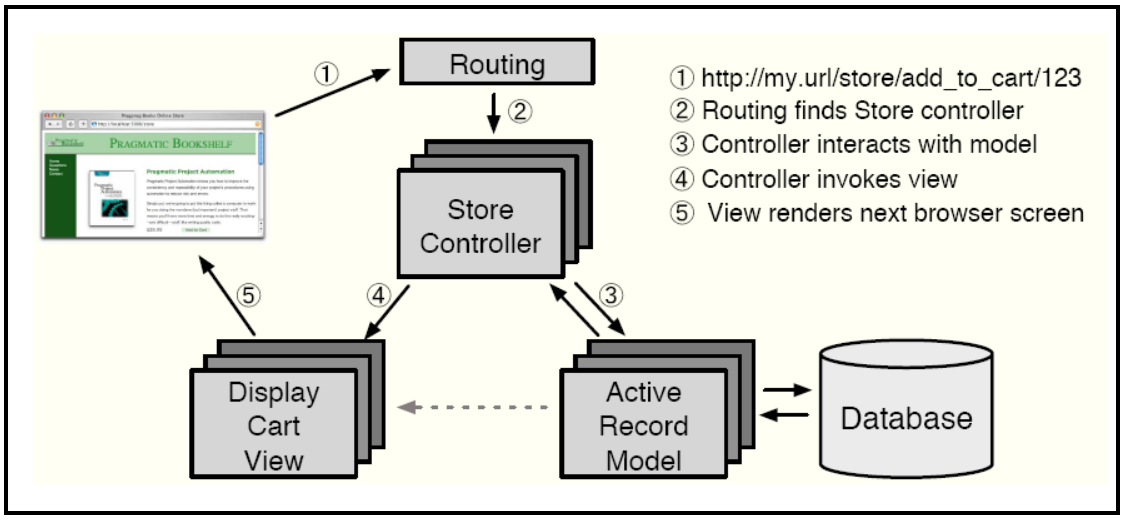

Figure 10-5 Global control flow

Boundary Condition

Install AC system	AC system operator creates AC system, installs it on a server machine, and configures resource parameters.
Initial AC system Infomation	AC system operator input all initial information about Advert Consult Inc. Initial information includes staffs' information and campaigns' information.
Convert Persistent Storage	When the AC system is shut down, AC system can convert the persistent storage from a flat file storage to a database storage or from a database storage to a flat file storage.

Table 10-2 Boundary Condition (1)

StartAC systemServer	The AC system operator starts the AC system. If the server was not cleanly shut down, the system will recover these operations when AC system starts. Once AC system start, users can initiate any of their use cases.
ShutDownAC systemServer	The AC system operator stops the AC system. Server will save ongoing modify or add operation and cached data. Once AC system shut down, no one can access AC system.

Table 10-3 Boundary Condition (2)

5. Applying design patterns to designing object model for the problem (using class diagrams).

Answer:

As different kinds of advertisements can be finished by different subgroups, if there is not a standard it should be very hard to manage the advertisements. **Abstract Factory** can be used to handle Campaigns.

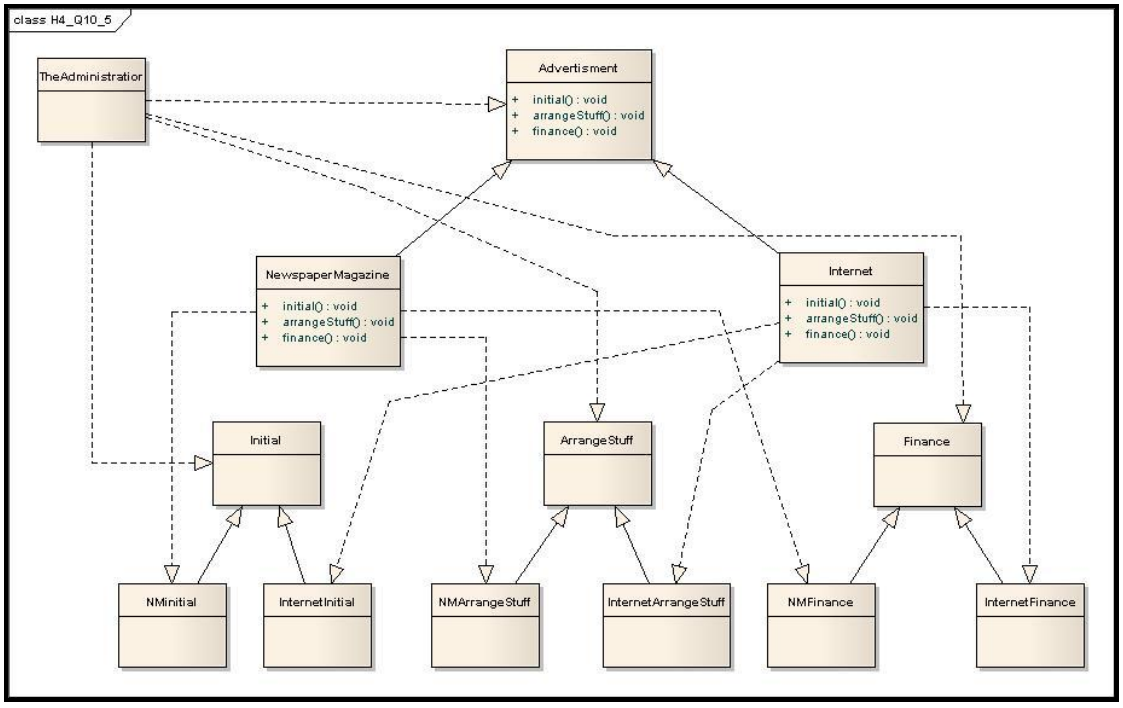

Table 10-6 Abstract Factory design pattern

6. Writing contracts for noteworthy classes

CampaignMgr class requires the attribute maxNumCampaigns to be positive.

```
context CampaignMgr inv.
    self.getMaxNumCampains() > 0
```

acceptCampagin(c) assumes that c has not yet been accepted in the CampaignMgr

```
context CampaignMgr::acceptCampaign(c) pre:
    !isCampaignNegotiated(c)
```

State that the CampaignMgr must not yet have reached the maximum number of Campaigns before invoking acceptCampaign().

```
context CampaignMgr:acceptCampaign(c) pre:
    getNumCampaigns() < getMaxNumCampaigns()
```

State that the Campaign c should be known to the CampaginMgr after acceptCampaign() returns.

```
context CampaignMgr::acceptCampaign(c) post:
    isCampaignNegotiated(c)
```

State that number of Campaigns in the CampaignMgr increases by one with the invocation of acceptCampaign().

```
context CampaignMgr::acceptCampaign(c) post:
    getNumCampaigns() = @pre.getNumCampaigns() + 1
```

Invoke getActiveCampaigns() from AdvertConsult results in all ongoing Campaigns, where each Campaign contains a Director and a Client, and the Staffs who are assigned with.

```
context AdvertConsult::getActiveCampaigns post:
    result = campaigns -> asSet
```

Ensure all Advertisements in a Campaign occur within the Campaign's time frame.

```
context Campagin inv:
    advertisements -> forAll(a:Advertisement | a.start.after(c.start) and a.end.before(c.end)
```

Question 11: Consider a simple intersection with two crossing roads and four traffic lights. Assume a simple algorithm for switching lights, so that the traffic on one road can proceed while the traffic on the other road is stopped. Model each traffic light as an instance of a TrafficLight class with a state attribute that can be red, yellow, or green. Write invariants in OCL on the state attribute of the TrafficLight class that guarantee that the traffic cannot proceed on both roads simultaneously. Add associations to the model to navigate the system, if necessary. Note that OCL constraints are written on classes (as opposed to instances).

Answer:

First and foremost, let's assume the traffic light changes in the following way:

… -> green -> yellow -> red -> green -> yellow -> red -> …

Traffic can move forwards when the light is green and yellow, however in yellow situation the light blinks to foretell the driver that red is coming, whereas the red light forbids any movement of the traffic.

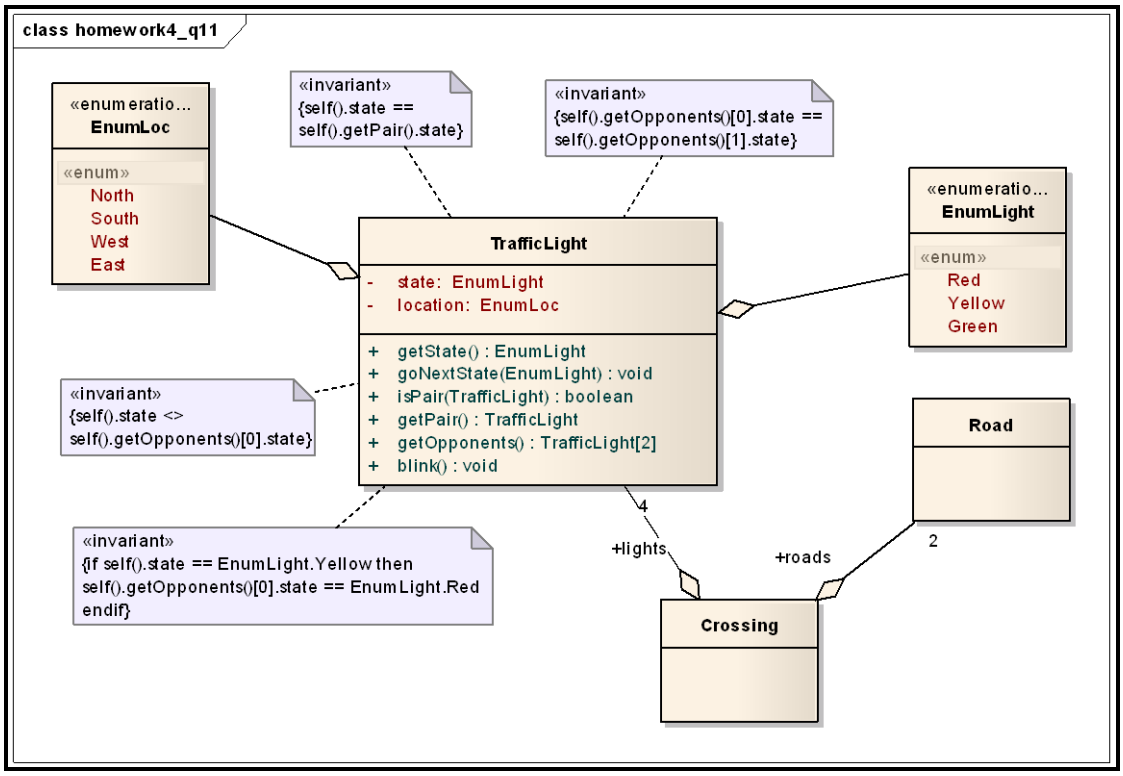

Figure 11-1 UML Class Diagram with OCL constrains

EnumLight stands for the three possible colors that can be shown in a light.

EnumLoc holds four geographic locations for four traffic lights in a crossing, north and south, west and east are a pair, and those two pairs are opponents to each other.

TrafficLight has two important methods, getPair() returns the other traffic light in the pair, whereas getOpponents() gives the two opponent traffic lights from current pair of view.

A sequence of invariants on state attribute of the TrafficLight class is defined as below:

1. inv. self().state == self().getPair().state
2. self().getOpponents()[0].state == self().getOpponents()[1].state
3. self().state <> self().getOpponents()[0].state

4. if self().state == EnumLight.Yellow then self().getOpponents()[0].state == EnumLight.Red endif

Within a pair, the color of traffic light should be the same.
Between two pairs, the color of traffic light should be different.
If a pair has yellow color, then the other pair must be red.

Question 12: Consider a sorted binary tree structure for storing integers. Write invariants in OCL denoting that

- All nodes in the left subtree of any node contain integers that are less than or equal to the current node, or the subtree is empty.
- All nodes in the right subtree of any node contain integers that are greater than the current tree, or the subtree is empty.
- The tree is balanced.

Answer:

Predefined:

```
class TreeNode {
    public int item;            // The data in this node.
    public TreeNode left;       // Pointer to the left subtree.
    public TreeNode right;      // Pointer to the right subtree.
    public int length;          // the length of tree
}
```

context BinaryTree **inv**:
TreeNodes -> forAll (b:TreeNode | b.left.item <= b.item or b.left == null)
context BinaryTree **inv**:
TreeNodes -> forAll (b:TreeNode | b.right.item >= b.item or b.right ==null)
context BinaryTree **inv**:
TreeNodes -> forAll (b:TreeNode | abs(b.left.length – b.right.length) <= 1 || b.length ==0)