3F6

SOFTWARE ENGINEERING AND DESIGN

Engineering Tripos Part IIA

2008 Solutions

Steve Young

Question 1

(a) A class diagram shows the architecture of a software system in terms of its classes, their attributes and methods and the relationships between them. In particular, the derivation of one class from another and the navigability between classses are shown. By contrast, an object diagram shows the actual instantiated objects that exist at some moment in time when the software is executing including the connections between them and the instantiated member values.

(b) The required object diagram is shown in Fig. 1. Note that any reasonable naming convention for the nested list items can be used. The convention followed here uses the index notation suggested in lectures.

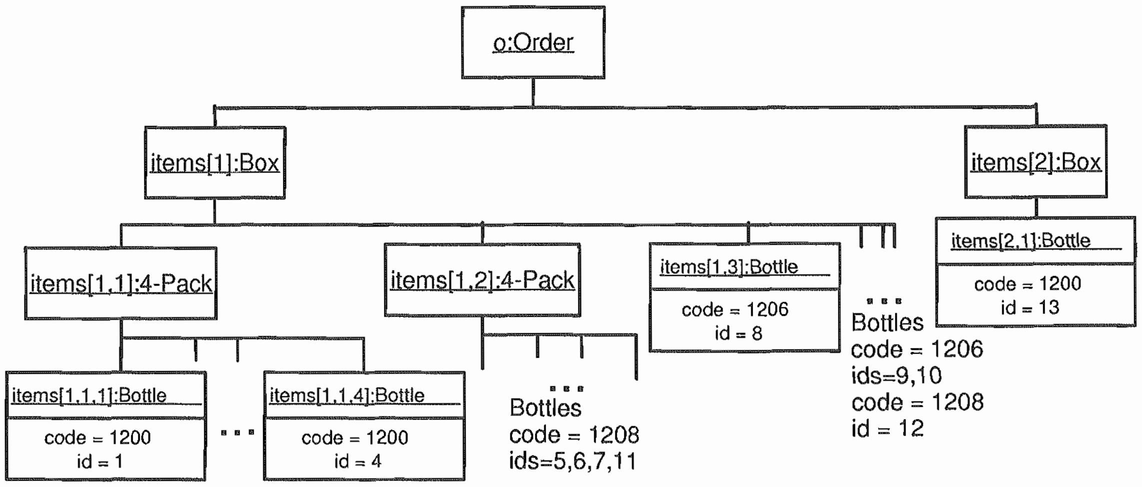


Figure 1: Question 1 Part b: UML Object Diagram

(c) The required sequence diagram is shown in Fig. 2. The same naming convention is used as in part (b).

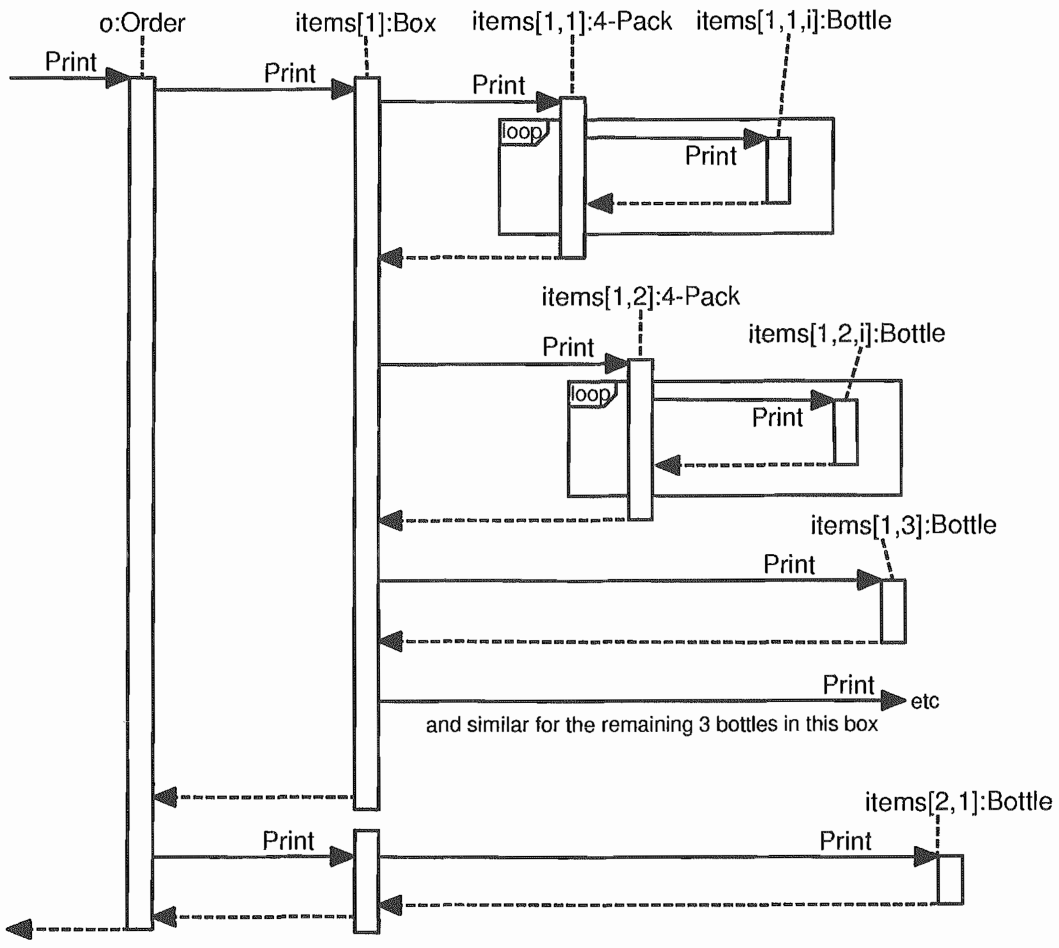


Figure 2: Question 1 Part c: UML Sequence Diagram

(d) There are a variety of strategies that could be used to implement the `Pack()` method. However, in this case the supplied class diagram offers a strong clue in that each class has its own `Pack()` method except the `Box` class. This suggests a design whereby each object "packs itself". To do this it is clear that the `Order` class will need to supply some helper functions. In particular, a function is needed to find and remove n identical bottles from its order list and a function to return a box with at least m spaces, if no existing box exists then a new one is created and added to the list. This would lead to code something like the following:

```
bool Bottle::Pack()
{
   ItemList fourbottles;
   if (order->GetBottles(4,code,fourbottles)){
      order->Add4Pack(fourbottles);
      return true;    //
```

```
    }else {
       Box * b = order->GetBox(1);
       b->AddItem(this);
       return false;
    }
}
```

and

```
bool FourPack::Pack()
{
   Box * b = order->GetBox(4);
   b->AddItem(this);
   return false;
}
```

where the return value indicates whether or not the item has already removed itself from the list. Given the above, `Order::Pack()` just needs to keep packing the first item in its order list until it is a box:

```
bool FourPack::Pack()
void Order::Pack()
{
   while (items.front()->GetType() != ABox){
      bool removed = items.front()->Pack();
      if (!removed) items.pop_front();
   }
}
```

Note that the type of each item is recorded explictly by adding a `type` field to class `Item`.

For the sake of a concrete illustration, the above solution is given in C++. This is more detailed than would be required in an exam where pseudo-code could be used to gloss over the details of list handling etc.

Question 2

(a) This is bookwork. A semaphore is used to give mutually exclusive access to a sequence of instructions (called a *critical section*). The semaphore method `enter()` is called on entry to the critical section and `leave()` is called on exit. If when calling `enter` some other thread is already executing the critical section, the calling thread is blocked. When leave is called and there are one or more waiting threads, then one of the threads is unblocked and allowed to proceed.

Signals are used to synchronise concurrently executing threads. When the wait method is called, the calling thread blocks immediately until some other thread calls the send method of the same signal.

Semaphores and signals are represented within the operating system as process queues. When a thread blocks on a semaphore or signal, it is placed in an associated process queue. When the thread is subsequently unblocked, it is removed from the semaphore or signal queue and moved the processor's ready queue.

(b) Code for bput and bget is as follows:

```
void bput(x:T)
{
   s.enter();                // enter critical section
   q.put(x);                 // store x in queue
   notempty.send();          // signal in case a thread is waiting
   s.leave();                // leave critical section
}

T bget()
{
   s.enter();                // enter critical section
   while (q.size() == 0)     // block until queue is
      notempty.wait();       // not empty
   T x = q.get();            // get element from queue
   s.leave();                // leave critical section
   return x;                 // return value
}
```

Note that this answer assumes that enter and wait have the same semantics as in the lectures where it is noted that calling wait inside a critical section must automatically release the associated semaphore and then automatically reacquire the semaphore when the wait terminates.

(c) Neither of the proposed solutions is acceptable. When the if statement is included, then the concentrator is effectively polling its input queues waiting for a packet to forward. When there is no traffic, this process will just execute a busy waiting loop needlessly wasting cpu. Removing the if statement avoids the busy waiting inefficiency but means that the concentrator will block as soon as it encounters an empty input channel. Meanwhile, the other input channels might be filling up.

(d) (i) A solution using an extra input buffer could be to create an extra protected `control` buffer holding integer channel numbers. The code is then modified to ensure that every time a packet is input to any of the input channels, the index of that channel is input to the control buffer. The concentrator would then operate as follows:

```
do {
   int i = control.bget();
   p = in[i].bget();
   out.bput(p);
} until forever;
```

(ii) A solution using multiple threads could be to create a thread for every input channel. Each thread can then read its input directly and happily block when there is nothing to do i.e. the i'th thread does

```
void worker(int i)
{
    do {
       p = in[i].bget();
       out.bput(p);
    } until forever;
}
```

Then the main body of the concentrator program would simply create N threads and sleep

```
for (i=0; i<N; i++)
    CreateThread(worker,i);
sleep(forever);
```

Question 3

(a) An IDL is used to describe interfaces to remote objects. Each interface gives a name to the object and describes its access functions. A number of interfaces can be bundled together to form a module along with additional type definitions to describe the data entities which are input and output to/from the objects.

A key feature of an IDL is that the interfaces it describes must be accessable and implementable in a range of programming languages. Furthermore, the internal implementation language need not be the same as the language used by the client to manipulate it. Thus, the constructs provided by an IDL must be supportable by all languages. C++ is too general-purpose for this. For example, C++ supports pointers and defines its parameter passing modes in terms of implementation (ie by value or by address) rather than logically (ie in, out or inout).

(b) (i) In a distributed system architecture, remote method parameters must be transferred across the network from client to server and back. Enforcing an explicit indication of which direction is intended allows an efficient and correct implementation of the transfer process.

(ii) The provided interface is an example of the factory design pattern. The `PatientFactory` interface provides access to the remote server without specifying which specific patient record is required. Once a `PatientFactory` object has been created on the client machine, it can be used to generate as many patient record objects as are required. The `Patient` interface allows access to the medical record of the person for whom the associated object was created.

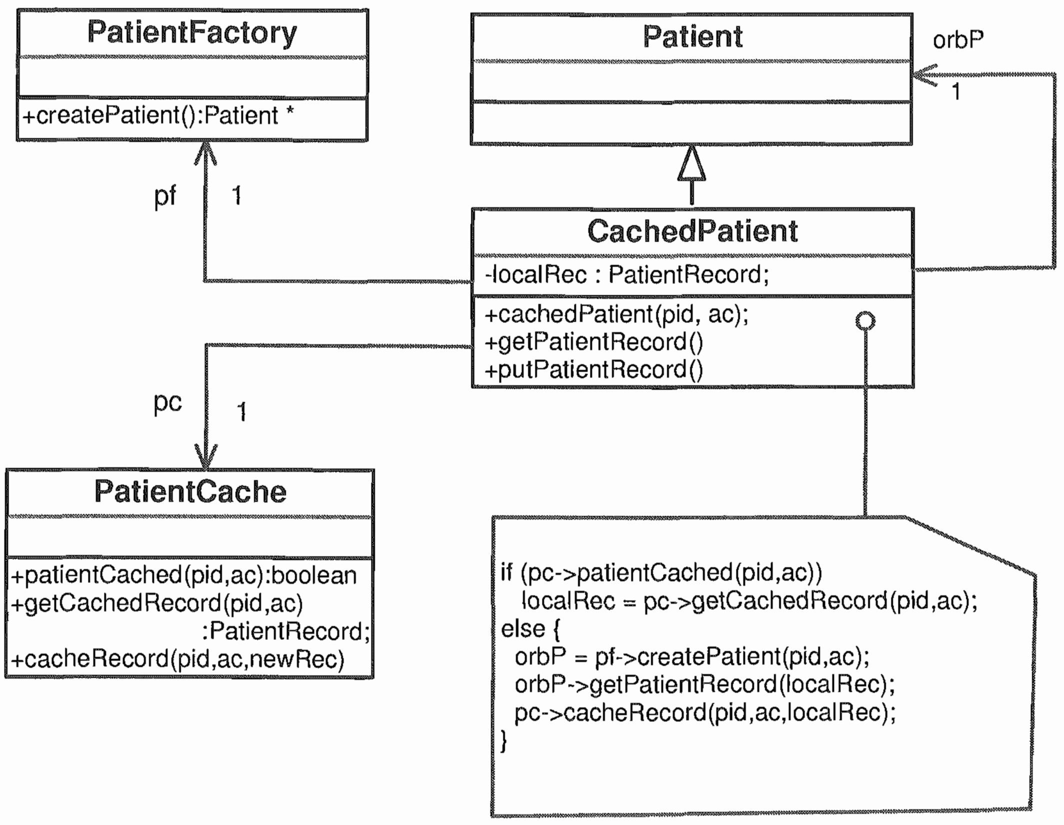
(c) Outlines of the C++ code are as follows:

```
PatientFactory::PatientFactory(CORBA::ORB_var orb, string ior)
{
   // create remote object via generic pointer
   CORBA::Object_var obj = orb->string_to_object(ior);
   if (obj==NULL) throw error();
   // narrow the object to the required type
   pfvar = PatientFactory::_narrow(obj);
   if (pfvar==NULL) throw error();
}

Patient * PatientFactory::getPatient(PatientId pid, AccessCode ac)
{
   Patient * p = pfvar->getPatient(pid,ac);
   if (p==NULL) throw error();
   return p;
}
```

(d) The obvious approach to performance enhancement is to wrap the `Patient` class in a derived class called say `CachedPatient` which examines a local cache of patient records and loads the record from the cache if possible, otherwise it loads the record from the remote server. Fig. 3 illustrates this.

Note that any cached patient record would need to be updated if `putPatientRecord` is called.



```
if (pc->patientCached(pid,ac))
   localRec = pc->getCachedRecord(pid,ac);
else {
  orbP = pf->createPatient(pid,ac);
  orbP->getPatientRecord(localRec);
  pc->cacheRecord(pid,ac,localRec);
}
```

Figure 3: Question 3 Part d: Class Diagram for Cached Patient Records

Question 4

(a) This is bookwork. A virtual function is a function whose definition can be redefined in a derived class. This implies that when a virtual function is referenced as in `p->vfunc()`, the actual function to execute can only be determined at run-time since it will depend on which specific derived class `p->` refers to. Virtual functions are implemented by generating a jump table of virtual function addresses for each derived class. Each instantiated object then holds a reference to the appropriate jump table for its class. Executing `p->vfunc()` then simply requires derefencing `p` to find the jump table, then looking up the address of `vfunc` in the jump table. An example diagram illustrating this was given in the lecture notes.

(b) The required class diagram for the on-line store is shown in Fig. 4. The only difficult part is the representation of stock levels. In effect there are 2 states: normal stock levels and low stock levels. Since the handling of orders and provision of delivery dates will be different for the two cases, the state design pattern is used. Note that there is an argument for introducing three states (normal, low stock and out of stock). Either solution would be satisfactory.

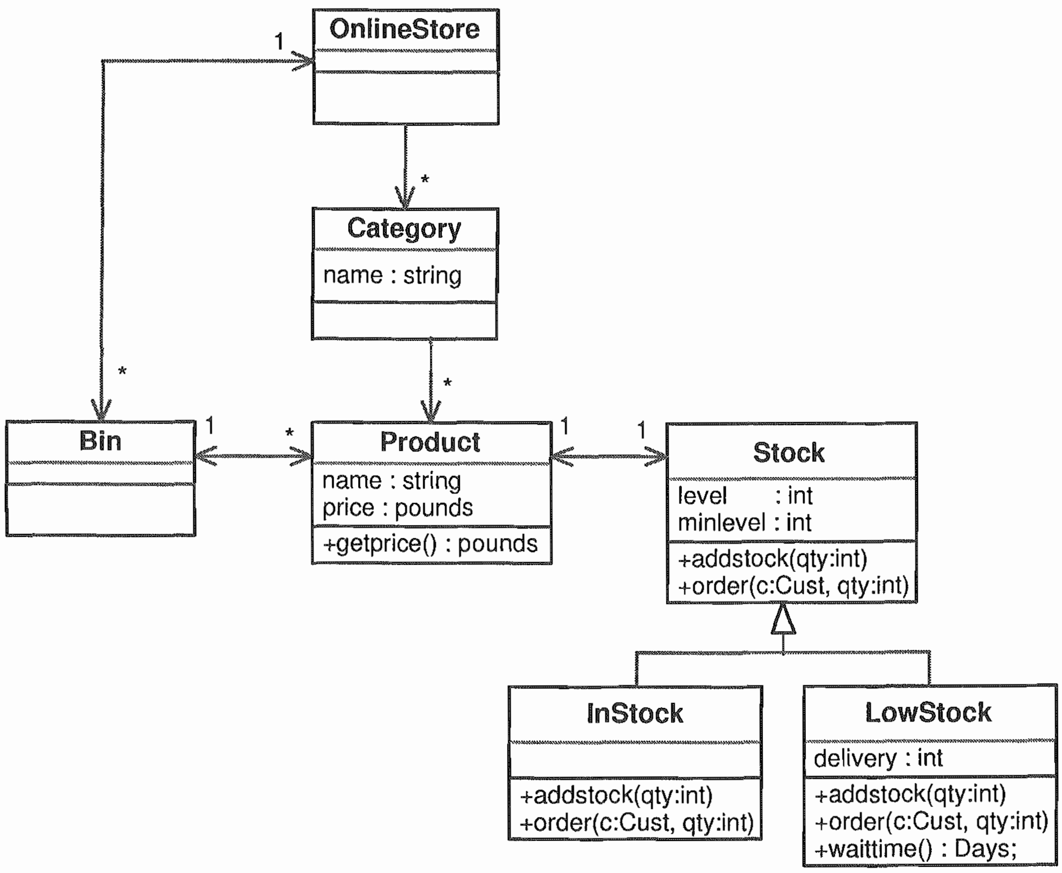


Figure 4: Question 4 Part b: Class Diagram for Online Store

(c) The key idea of the decorator design pattern is to extend the functionality of a class whilst preserving its interface. This means that some other client class cannot distinguish the decorated class from the original class allowing multiple layers of decoration whilst retaining the ability to treat instantiated objects as a homogeneous set. Virtual functions are key to the implementation of this pattern since they enable the behaviour of interface functions to be modified in each layer of decoration without changing the interface signature. Note finally that the decorator pattern is a run-time structure i.e. bjects are decorated at run-time by linking the decorator object to the decorated object via a pointer.

(d) A solution to providing multiple special offer schemes using the decorator pattern is shown in Fig. 5. Note that the concept of customer reward points has been added to the `Product` class. This is the only change to the existing system. The abstract `Offer` class forwards requests for price and points to the `Product` class. Concrete special offers then provide a layer around the product class which modifies the price and/or bonus points as required.

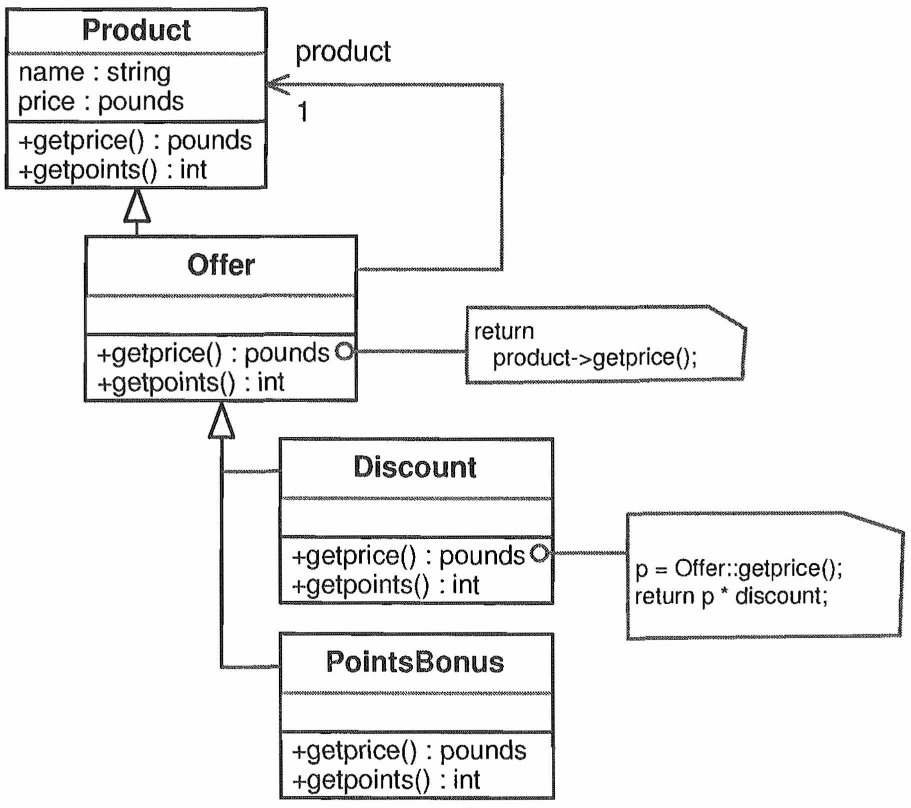


Figure 5: Question 4 Part d: Using the decorator pattern to implement special offer schemes