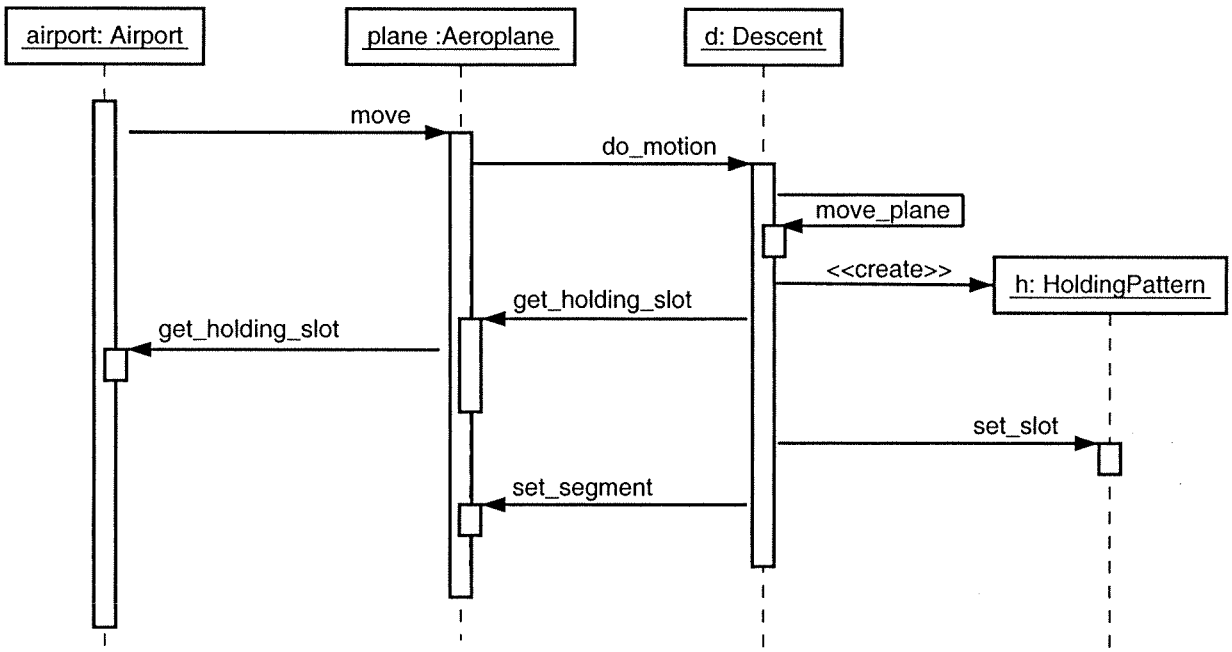
Solutions to 3F6 Software Engineering and Design questions, 2003

1. *UML, design patterns, sequence diagrams*

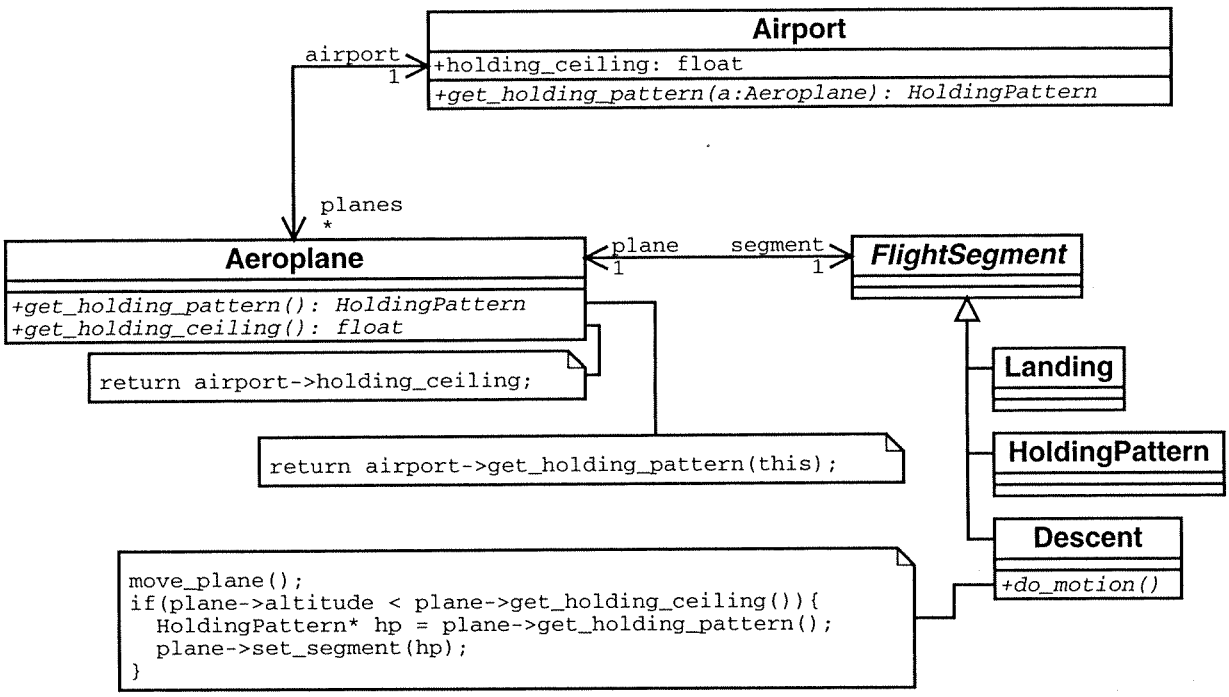
   (a) There are six classes in the diagram. Each `Airport` has many `Aeroplanes`. Each `Aeroplane` has one `FlightSegment`. Each `FlightSegment` is a `Landing`, `HoldingPattern`, or a `Descent`. The relationships between `Airport` and `Aeroplane` and between `Aeroplane` and `HoldingPattern` have bi-directional navigability shown by the arrows.

   The relationship between `Aeroplane` and the `FlightSegment` class hierarchy represent an instance of the *State* pattern.

   (b) The sequence diagram is shown below:



(c) The simplest solution to this doesn't require many changes. The software is already capable of handling multiple airports, but in this new version of the software Airports have to provide some additional services; they need to know their holding pattern ceiling and they need to be able to provide differing holding patterns. If airports are now going to provide holding patterns, this function can absorb get_holding_slot since the returned HoldingPattern can already have its slot set correctly.



Examiner's remarks: Some students noted that this modification embodies the *Factory Pattern* (Airports now create Holding Patterns). Others solved the problem using a variant of the *Visitor Pattern*. Other solutions involved sub-classing the Airport and HoldingPattern classes which is not strictly necessary.

2. *CORBA*

(a) The remote proxy pattern is sued when an object on one machine needs to talk to an object on a remote machine. A proxy for the remote object exists on the local machine and the client talks to this proxy as though it were the remote object. The proxy responds to requests and handles communication with the remote object.

(b) AccountingSystem, Account and Invoice will all require proxy classes in the client.

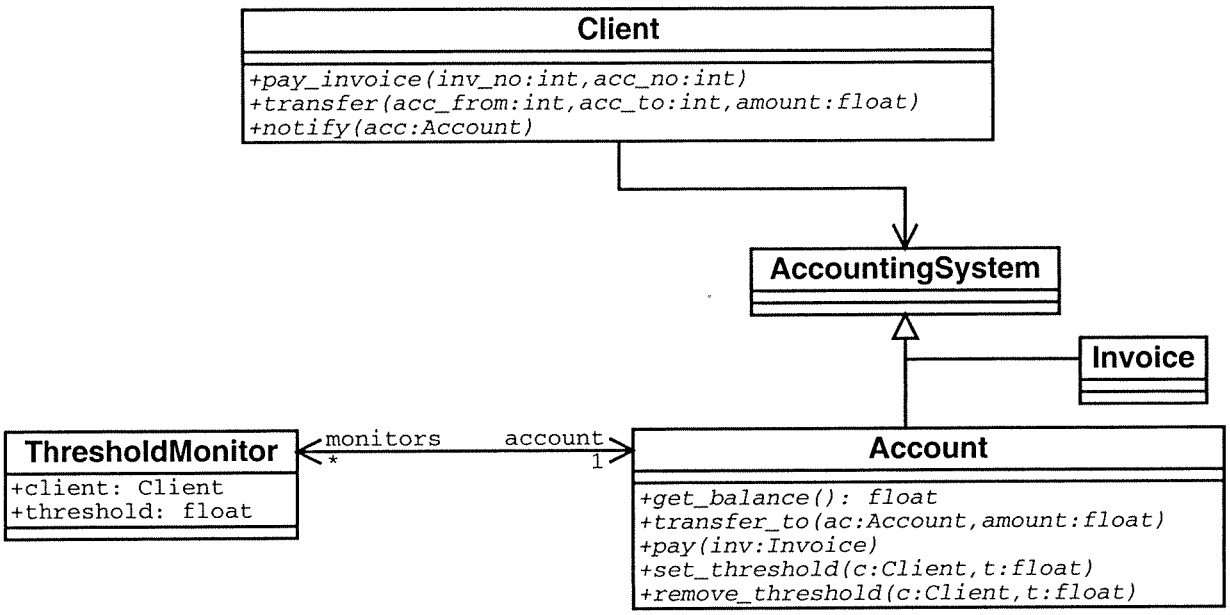
(c)

```
interface Invoice {
  string get_payee();
  float get_amount();
};

interface Account {
  float get_balance();
  void transfer_to(in Account ac, in float amount);
  void pay(in Invoice inv);
};

interface AccountingSystem {
  Account get_account(in long acc_no);
  Invoice get_invoice(in long ref_no);
};
```

(d) To do this, the client will now need to present an interface so that the Accounting system can notify when an account drops below threshold. The Account will also need to present an interface for imposing (and preferably removing) thresholds.



The idl is given below:

```
interface Client {
  void notify(in Account ac);
};

interface Account {
  // interfaces as before plus
  void set_threshold(in Client c, in float t);
  void remove_threshold(in Client c, in float t);
};
```

3. *Concurrent systems, monitors and fault minimisation*

(a) A condition variable is a binary flag which states whether a condition has occurred or not. When `cwait` is called, the process blocks, joining a queue, until the condition is met. When `csignal` is called, this signals that the condition has now been met. The first process waiting in the queue is immediately unblocked and continues from where it left off. This means that `csignal` can only be called as the last command in a function. [3]

(b)
```
void exit(int direct)
{
   // A car has left the bridge, so one fewer is crossing
   numCrossing--;

   // Swap direction if it was the last one
   if(numCrossing == 0)
   {
      currentDirection = 1 - currentDirection;
   }

   // Signal a waiting car to start crossing. This uses
   // the currentDirection instead of direc so that if
   // the direction has changed, it can start a car
   // that is waiting at the other side
   csignal(waitForDirection[currentDirection]);
}
```
[6]

(c) Fault minimisation is concerned with trying to avoid creating faults in the program, and so avoiding design and programming decisions which are likely to lead to lead to these faults. Monitors automatically provide mutual exclusion, whereas semaphores rely on pairs of calls to `WAIT` and `SIGNAL` on mutexes. Mistakes can be made in the pairing of these `WAIT`s and `SIGNAL`s, so semaphores are inherently more dangerous and lead to more faults. Object-oriented languages are are often used to aid fault minimisation, since their strong structure and encapsulation removes the opportunity for some faults. Monitors, as objects, fit neatly into this object-oriented framework, which gives them an additional advantage.. [3]

(d) Fault minimisation involves making architectural, algorithmic, or implementational decisions which reduce the chance of faults being created while writing the program. Any of the following decisions (or similar) would be appropriate:

Data hiding and encapsulation Ensuring that each software component does a particular well-defined job (encapsulation), and that as much of it's inner workings as possible are hidden from the other programmers (data hiding) reduces the chances of a component being misunderstood, or misused.

A strongly-typed language Languages such as Java, Ada, or (to some extent) C++, make it very difficult for variables to be used for any purpose than that for which they are intended (e.g. `int`s must always represent integer numbers). This reduces the chance of variables being misused.

Designing programs for readability Many faults come from simple programming errors, often brought about by misunderstanding or misreading existing code. Making programs as readable as possible reduces the chances of this occurring and also makes it easier to visually inspect programs for errors.

Avoiding dangerous constructs Constructs such as `goto`s, pointers, recursion, interrupts, or inheritance are very powerful, but have serious consequences if misused. Avoiding their use all together is the best way to ensure that faults due to these cannot occur.

A well-defined development process All of the developers involved in the project should be aware of the sequence of events needed to produce a piece of software. They should follow the process and communicate with each other, or faults can easily creep in. [4]

(e) The direction only switches once the bridge is clear, so the car will have to wait until the entire queue of traffic has cleared the bridge. The current solution therefore suffers from *starvation*. One simple solution would be to keep count of the number of cars that have crossed the bridge, and switch direction after a certain number have crossed. [4]

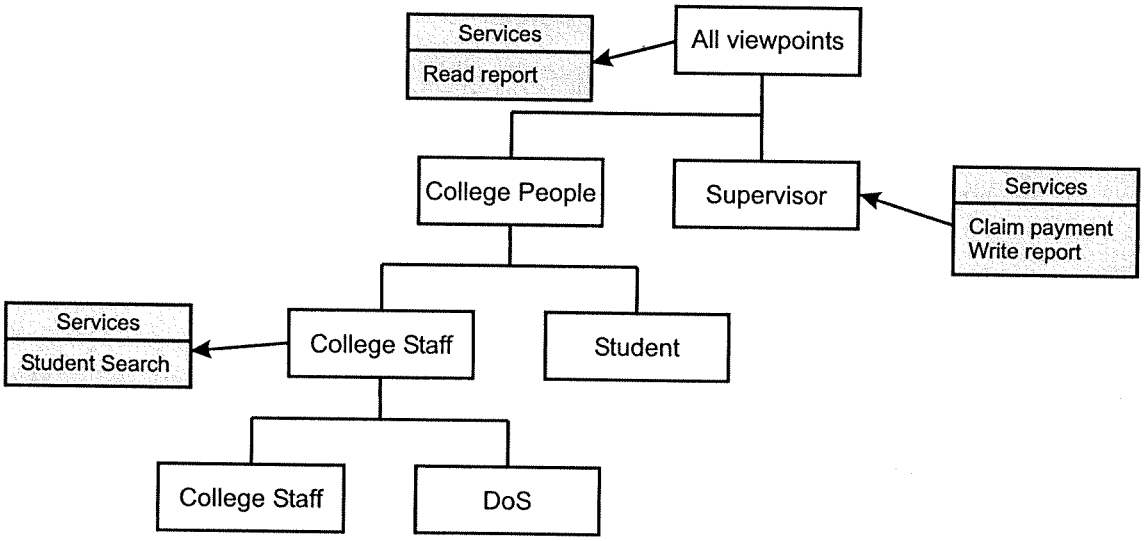
Examiner's remarks: This question was about concurrent systems (using monitors) and fault minimisation. It was a very unpopular question, presumably because monitors have not been examined for many years, but was well-answered by most of the students that attempted it. There was some confusion about the use of condition variables, and many students confused fault minimisation (avoiding creating faults in the first place) with fault tolerance (making the program robust to any faults in the program). Most students, however, correctly answered the 'sting in the tail'.

4. *Database systems, software specification and development*

(a) *Viewpoints* are those elements of the whole system that are external to the software, often the various users. Possible viewpoints are: student, supervisor, tutor, director of studies, senior tutor, data entry personnel, pay clerk, database maintainer...

The software provides *services* to the viewpoints. Possible services are: Student search, read report, write report, claim payment, sort, add student... [4]

A possible structure chart is



(b) Databases are standard, and are known quantities. They can be configured to behave as required, without having to develop the whole database in-house. A commercial database is likely to be delivered faster, cheaper, and to contain fewer errors. What is more, the database's requirements can be specified entirely up-front, so standard Waterfall development will be fine for this element.

The front end is specific to this application, and the users will have specific requirements. As with most user interfaces, this will be difficult to specify, and it is better to develop the system and the specification in parallel, with plenty of feedback from the users through the use of prototypes and user trials. This is known as *evolutionary development.* [3]

(c) The *stakeholders* are all those who have some influence on the system requirements. The database requirements can be specified at the beginning and are unlikely to change, so the stakeholders need only be consulted once, at the start of the development process. This will specify both the functional and non-functional requirements which will enable a choice of commercial database to be made, and the database suitably configured.

The evolutionary development of the front end will require repeated prototypes, each of which will be delivered to some of the stakeholders (although not all), for evaluation and feedback. Thus, although all of the stakeholders are consulted at the start of the design process, some are also involved throughout the development of the front end. [2]

(d) Pessimistic concurrency control assumes that other users may want to edit the same database records at the same time as you do. It avoids this by first locking each record involved in the transaction, so that no-one else can modify them. This guarantees a consistent database state, but can slow down the start of the transaction, and also runs the risk of deadlock, if two transactions try to lock the same set of records at the same time.

Optimistic concurrency control assumes that everything conflicts rarely occur. The database records are not locked, and instead operations are performed on copies of the records. The database management system then checks to see if these original records have been modified by anyone else since. If not, the new copies are copied back into the database. If there has been a conflict, the copies are thrown away and the transaction starts again on new copies of the new database records. As a result, optimistic concurrency control can be slow in committing finished transactions.

Optimistic concurrency control is most appropriate for this system, since clashes are unlikely to occur, and locking every record when someone wants to read it would add to the time taken to read and edit reports. [4]

(e) The crash recovery algorithm described in lectures makes periodic backups of the entire database (checkpoints), as well as maintaining a log of every change made to the database. If a system failure occurs, the most recent checkpoint is reloaded, and the log then analysed to undo or redo any other transactions as necessary, and in particular to ensure that the ACID properties are maintained:

Atomicity requires that either all of a transaction is completed, or none at all. A simple re-load of the checkpoint after a system failure is not enough to guarantee this, since the checkpoint could have been taken in the middle of a transaction. The crash recovery-algorithm ensures atomicity by either completing any partial transactions at the checkpoint by reference to the log, or (if the crash occurred before the transaction was finished, and therefore not logged), by undoing the entire transaction.

Consistency requires that transactions move the the database from one consistent state to the next. The re-loaded checkpoint is likely to be inconsistent, but redoing any transactions that it can from the log, and undoing the others, ensures that consistency is achieved in the recovered database.

Isolation requires that the changes due to a transaction are not revealed until the entire transaction is complete. This is usually guaranteed in the database's normal operation, so by restoring the database to a consistent

state and removing any transactions which were part-finished at the system failure, this property is ensured.

Durability requires that the results of any committed transaction are persistent, even in the event of system failures. This is the aim of the recovery algorithm. The results of all committed transactions will either be in the checkpoint, or the stages necessary to redo the entire transaction are in the log, and so will be repeated in the system-recovery process. (The log entries are written before the database is changed each time, to ensure that the log is always completely up-to-date). [5]

Examiner's remarks: This question was about software specification, development, and databases. It was generally well-answered, although a number of students had clearly attempted it last and run out of time. Most candidates could discuss approaches to development and database design satisfactorily, but many students had not considered the structure of a viewpoint structure chart important, and had not given enough thought to it. Many candidates had no hierarchy in the structure chart, while Director of Studies and Tutor (for example) could both be considered as College Staff, as in this crib. The hierarchy is important as it could prompt the identification of other viewpoints which might otherwise be missed. In the last part, to discuss the ACID properties in the context of recovery algorithms, many candidates described one or the other (or both), without actually linking the two.