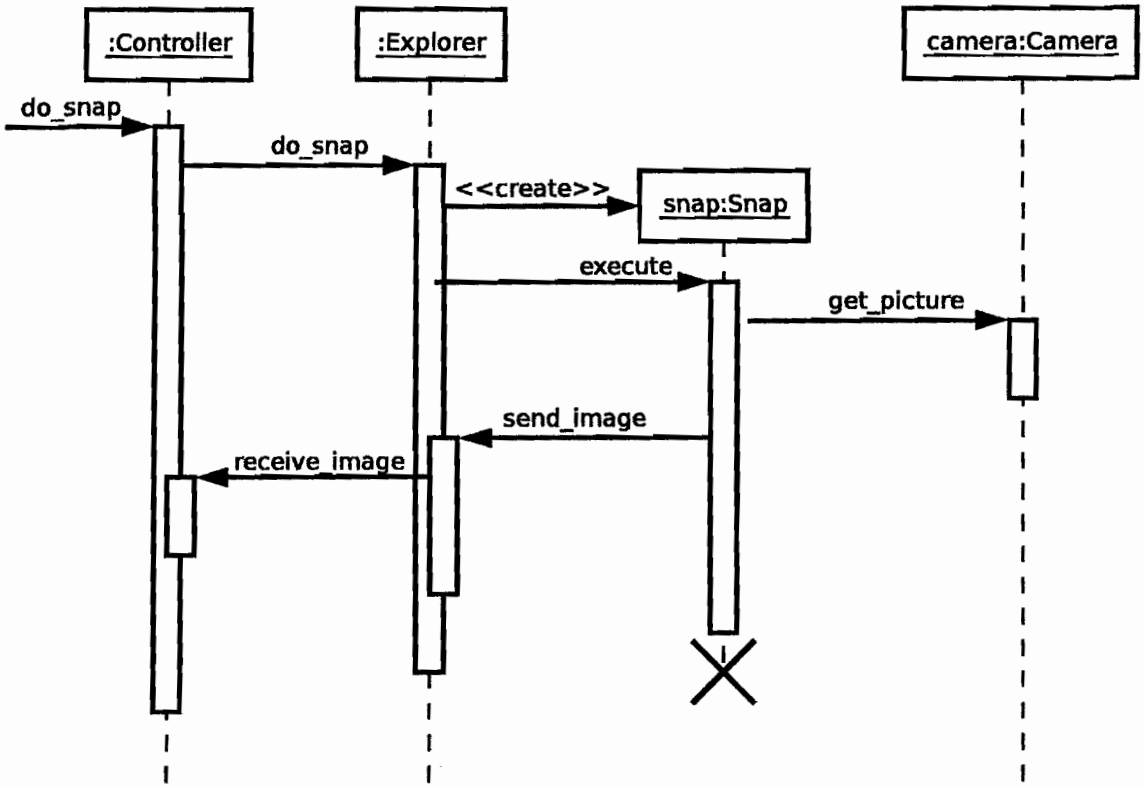
PART IIA 2007
3F6 Software engineering and design
Dr T Drummond

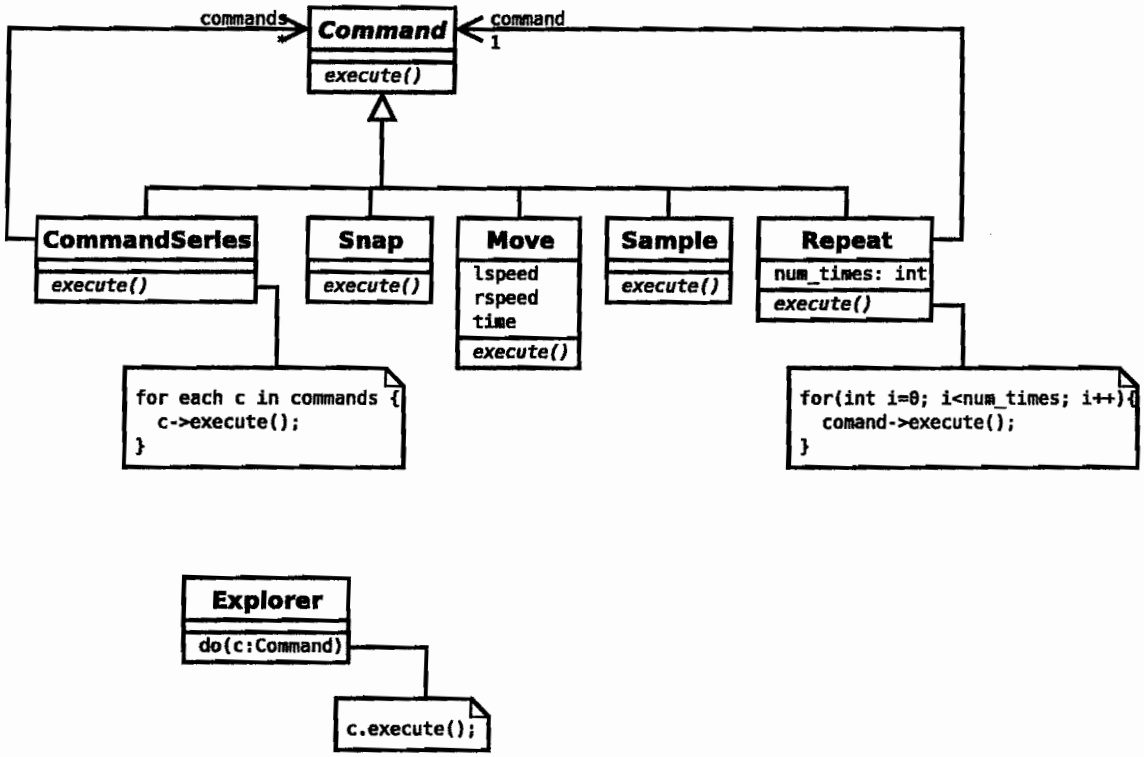
Solutions

1 (a) The sequence diagram looks like this:

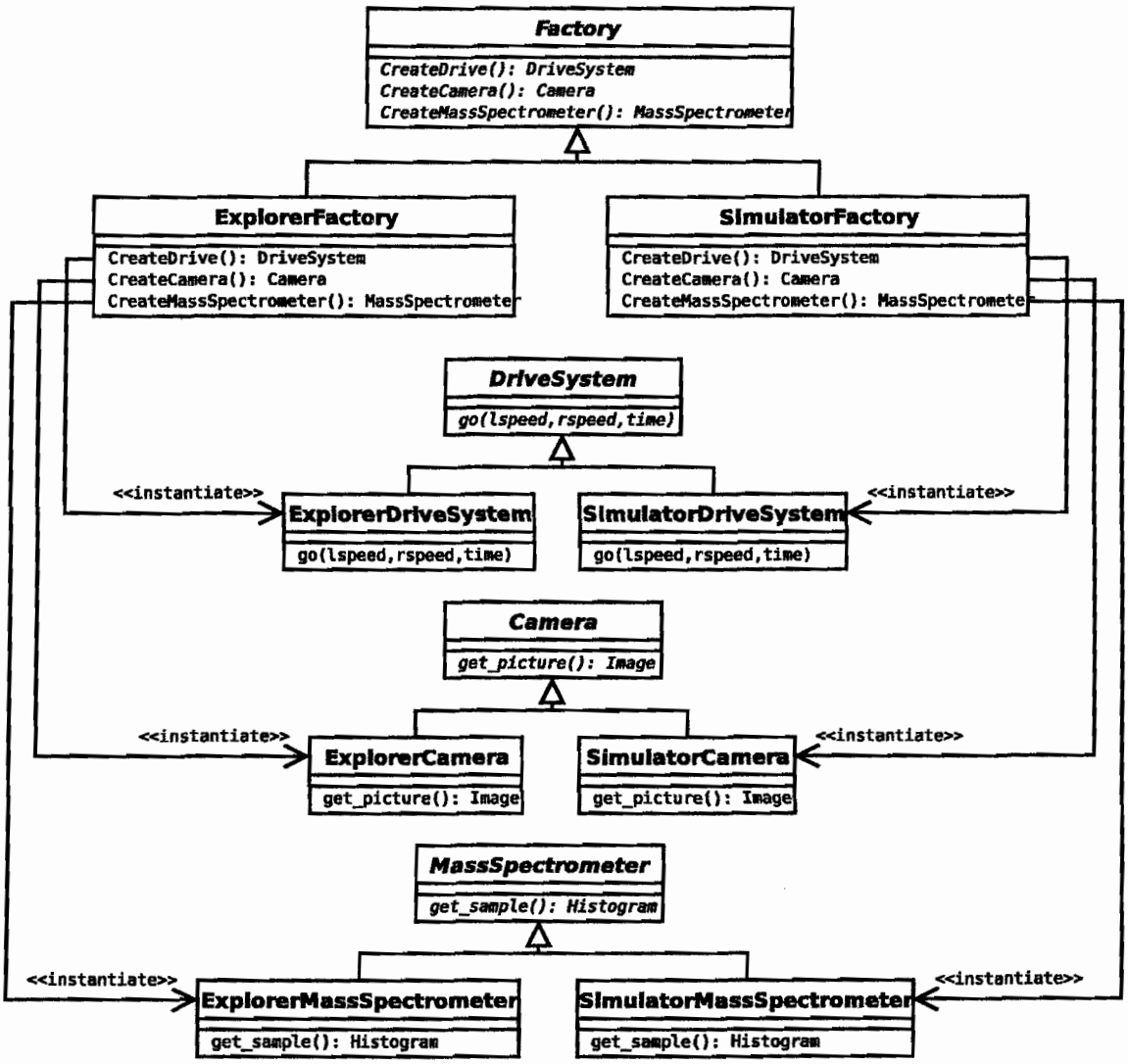


Version: 1

(b) The simplest approach is to combine the three existing commands (`Move`, `Snap` and `Sample`) into a class hierarchy. Two new classes can then be added to this hierarchy to encapsulate the new features. The `CommandSeries` class embodies the Composite Design Pattern and the `Repeat` class embodies the Decorator Design Pattern.



(c) The Abstract Factory Design Pattern can be used in one of two ways. It can be used to manufacture commands for simulation or the real explorer, or it can be used to manufacture components (Drive System, Camera, Mass Spectrometer) for either the simulator or the real explorer. The second approach is shown below:

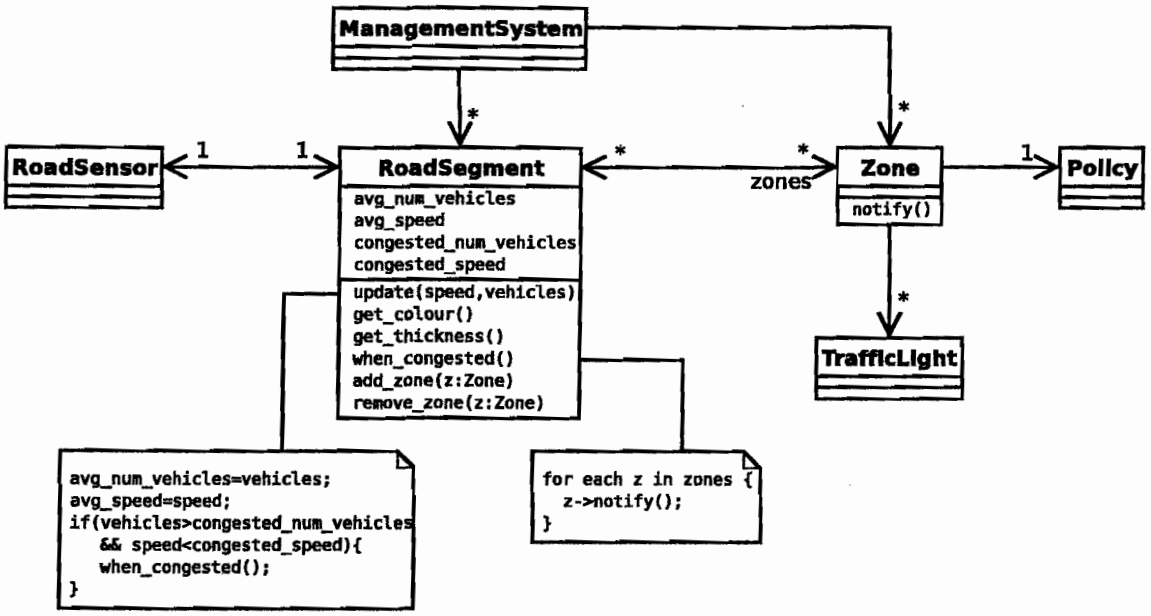


The lower half of this diagram contains repeated structure and candidates may reasonably omit two of the three repeated structures provided that they indicate that the architecture for e.g. the DriveSystem applies also to the other two components.

2 (a) Programming Interface can be decoupled from implementation by using the data hiding capabilieis of classes. Classes provide encapsulation which embodies both data and functions. By making the functions public and the data private, classes can decouple their internal implementation from their interface, allowing the implementation to be changed without changing the interface.

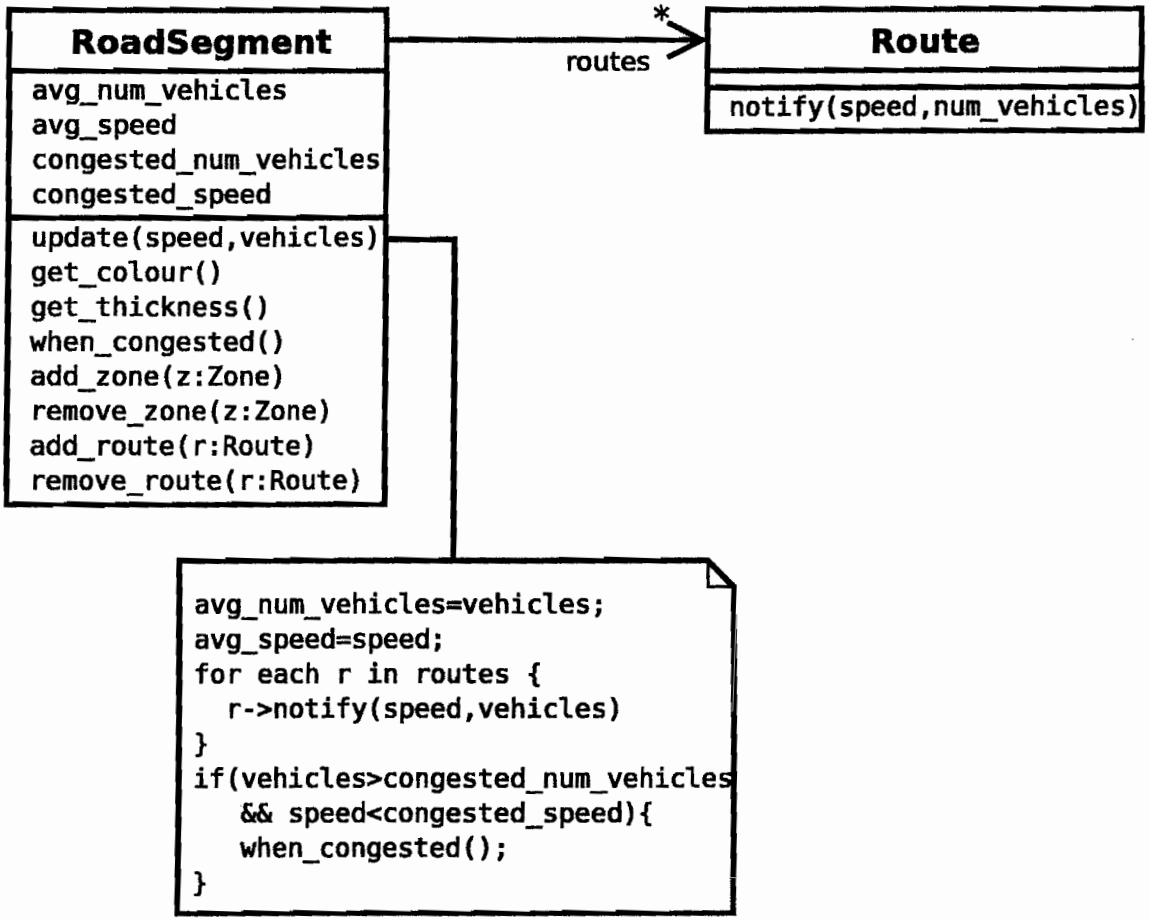
User Interface can be decoupled from internal state by using the Observer Design Pattern. This allows the internal state to be manipulated without knowing about the user interface code. The user interface observes the state, represents it for user display and provides the user with ways of controlling it. This approach is also called Model-View-Controller (MVC).

(b) A suitable UML class diagram is shown below:



Candidates could note that Zone is an Observer of RoadSegment.

(c) This can be achieved by including another observer of RoadSegment (namely the Route Planning software):



3 (a) An IOR is a form of universal resource locater (URL). It enables a CORBA object to be accessed from anywhere on any system. It support multiple protocols so that local private networks with propriety networking protocols can be be used as well as public networks such as the internet. It contains

(i) the object's fully qualified interface name

(ii) the host address and port number of the server

(iii) metadata about the server ORB to facilitate optimisation

(b) The steps needed to access the object are

(i) If not already running, create a local ORB.

(ii) Ask the ORB to construct an object ref given the IOR

(iii) Narrow the obj ref to create a smart pointer to the object itself

(iv) Access the objects exported attributes and methods using the smart pointer

(c) To convert to a factory pattern, simply add an interface of the form

```
interface VFactory {
   Verifier getservant(in string name, in string passNum);
};
```

also delete the `init` method from Verifier and remove `transaction_id` from all other calls. Calling `getservant` creates a Verifier object initialised with the supplied name and passport number.

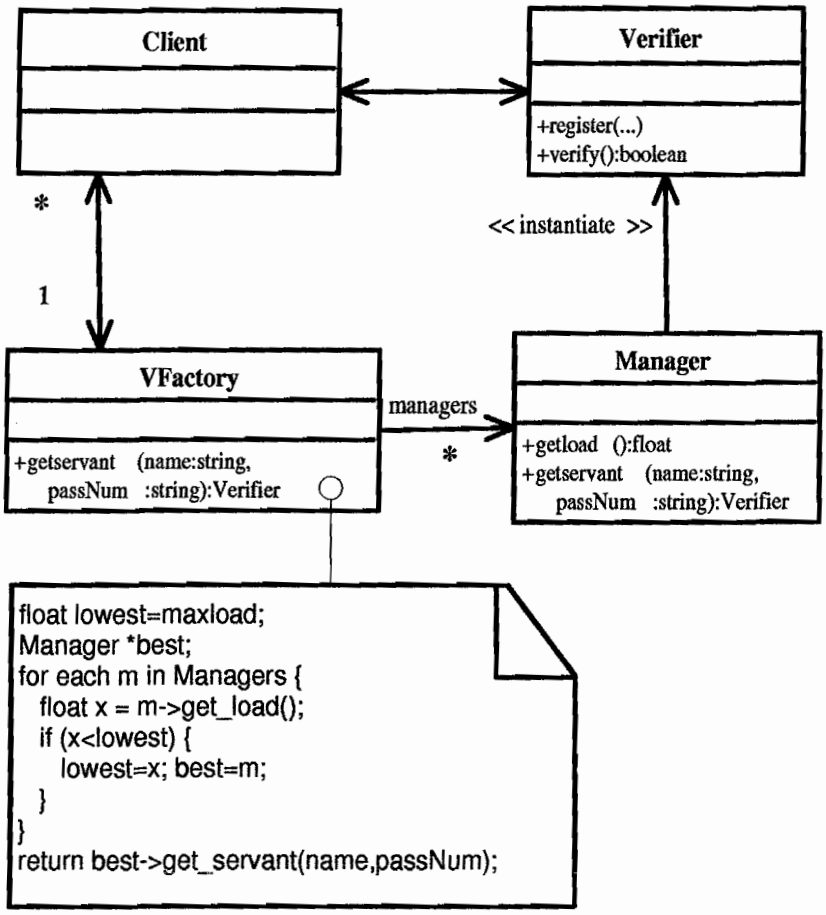
Advantages include:

(i) Simplification of the interface and increase in efficiency

(ii) Opportunity to perform load balancing (as in part d)

(iii) Better separation of per-client data. This is now distinct in each client rather than shared and hence does not need to be protected by mutexes.

(d) the outline of a suitable load manager is sketched out in the following figure



(e) The simplest solution to the upgrade problem is to add a new Finger Print type and derive an extended interface

```
module BioCheck {
   // existing module contents

   typedef short FingerPrint2[128][128];

   interface Verifier2 : Verifier {
      void register2(in Hand h, in FingerPrint2 fp);
   };
   interface VFactory2 {
      Verifier2 getservant(in string name, in string passNum);
   };
};
```

Version: 1

4 (a) There are 3 types of code review: inspections, walk-throughs and readings. All 3 share common features

(i) the goal is to detect errors not to fix them

(ii) the reviewers should be familiar with the overall design and the organisation's software standards

(iii) management does not attend or see the outcome

(iv) all participants must be circulated with all relevant documents before the meeting and must prepare for the meeting.

The differences are mostly a matter of emphasis:

(i) an inspection is quite formal, it must follow a defined procedure.

(ii) an inspection team is typically 5 or 6 people each of whom as an assigned role: moderator, author, secretary, user-rep, standards-bearer, QA-inspector. A walk-thru consists of 1 to 3 reviewers plus the author who manages the process.

(iii) a reading is like a walk-thru except that the emphasis is on the preparation phase, the review meeting is simply a reporting of results.

(b) Before the meeting each reviewer should be provided with

(i) copies of all related design documentation

(ii) a copy of the organisation's coding standards

(iii) copies of previous reviews on this or similar code

(iv) a list of any special use cases that the manager wishes that reviewer to focus on

(c) Code review:

(i) Car park empty - no issues although the unrestricted access to the global shared variables `freeBays` and `map` appears to be risky.

(ii) Car park full - when full the index `i` in `getNextFreeBay` will increment to `NumBays`. The subsequent reference to `map[i]` will then fail with an array bound error.

(iii) Simultaneous departure. Simultaneous execution of `++freeBays` in `car_leaving` will have undefined results.

Note that since code reviews are only intended to detect errors not fix them, solutions to these problems were not required. For completeness, the required solutions are (a) change `i<=NumBays` to `i<NumBays` and (b) all code in the 3 routines are critical sections and should be protected by a semaphore.

(d) If there were multiple entry barriers then the statement

```
if (freeBays==0) notfull.wait();
```

may fail because when there is just 1 free bay in the car park, a car could go through another barrier in parallel and take the bay before the first car can claim it. [Solution, though not required, is to change `if` to `while` so that the full condition is rechecked before claiming a bay.]

(e) If cars park in unallocated bays, the pre-allocated unused slots will never be reset. So eventually the car park will become "full" even though there are no cars in it.

The solution is to extend map entries to record 3 status levels: free (0), allocated (1) and occupied (2). The `getNextFreeBay` routine will then set the bay as allocated, and when `car_arriving` is called the status will change from allocated to occupied. If the bay was not already allocated then this is a wayward car so the map is scanned, the allocated bay is found and reset to the free state.