ENGINEERING TRIPOS PART IIA 2004

Solutions to Module 3F6
Software engineering and design
Principal Assessor: Dr T W Drummond
Second Assessor: Dr P A Smith

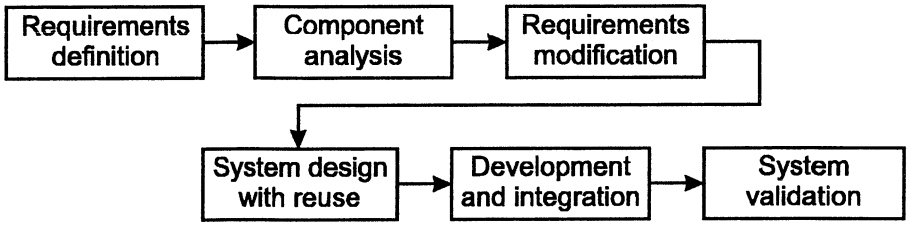
Solutions to 3F6 Software Engineering and Design questions, 2004

1. *Software processes, reusable components, critical systems*

   (a) Most software projects are too large for one person to complete alone, and too large to visualise in their head; they need a team of people writing software, and people to manage the project. In order that the project can be properly managed and, in particular, that the progress properly monitored, both the managers and the software developers need to be aware of the sequence of events that will be followed to develop the software; this is the *software process.* The software developers need to know what documentation they must produce to provide feedback to the managers, and the managers must know what jobs the developers are currently doing, and how much further there is to go in the process. Many software projects fail because a proper software process is not followed, and developers and managers just develop away at the project without knowing how progress is being made, and when it will be complete.

   Software should always be developed using a software process. However, applying the wrong process to a software job can lead to problems. For example, trying to apply a rigourous scheme such as the Waterfall model to a task where the requirements are uncertain will inevitably lead to problems at later stages, and the final product not meeting the real requirements. [25%]

   (b) The diagram below shows the Reusable Components process model:



   Like the Waterfall model, the Reusable Components model has a step-by-step march towards the final product, which begins with talking to the client and specifying the software requirements (the *Requirements definition*), and finishes with building the product (*Development and integration*) and testing it (*System validation*). However, the design stage involves an iteration: given the specification, the existing available components are analysed to see whether a system that meets the requirements can be built from them (*Component analysis*). This system is then proposed to the client and, through some negotiation, the specification is revised to make use of these components (*Requirements modification*). The *system design* then involves designing the correct combination of these components, and the design of any additional software components. [25%]

(c) The Reusable Components model can be described as both *top-down* and *bottom-up*. It is driven initially from the requirements definition, which makes it top-down. However, the component analysis and requirements modification stages are bottom-up, since they start with the individual components and build up from those. [10%]

(d) Advantages include:

- Faster development, since less code needs to be written.
- Less time spent on component testing, since components are already written and tested: only system tested is needed.
- Lower cost of development.
- Fewer risks in the development process, since the components are already known and understood.

Disadvantages include:

- Some compromise is likely to be needed on the system requirements.
- The process depends on the availability of high-quality components. If the components do not do the job as advertised, the whole process is compromised.
- The system as described in section (b) is a variant of the Waterfall model and will suffer from many of its disadvantages, particularly that if mistakes are made in one of the stages, the process does not allow for backtracking and the problems must instead be worked around. [20%]

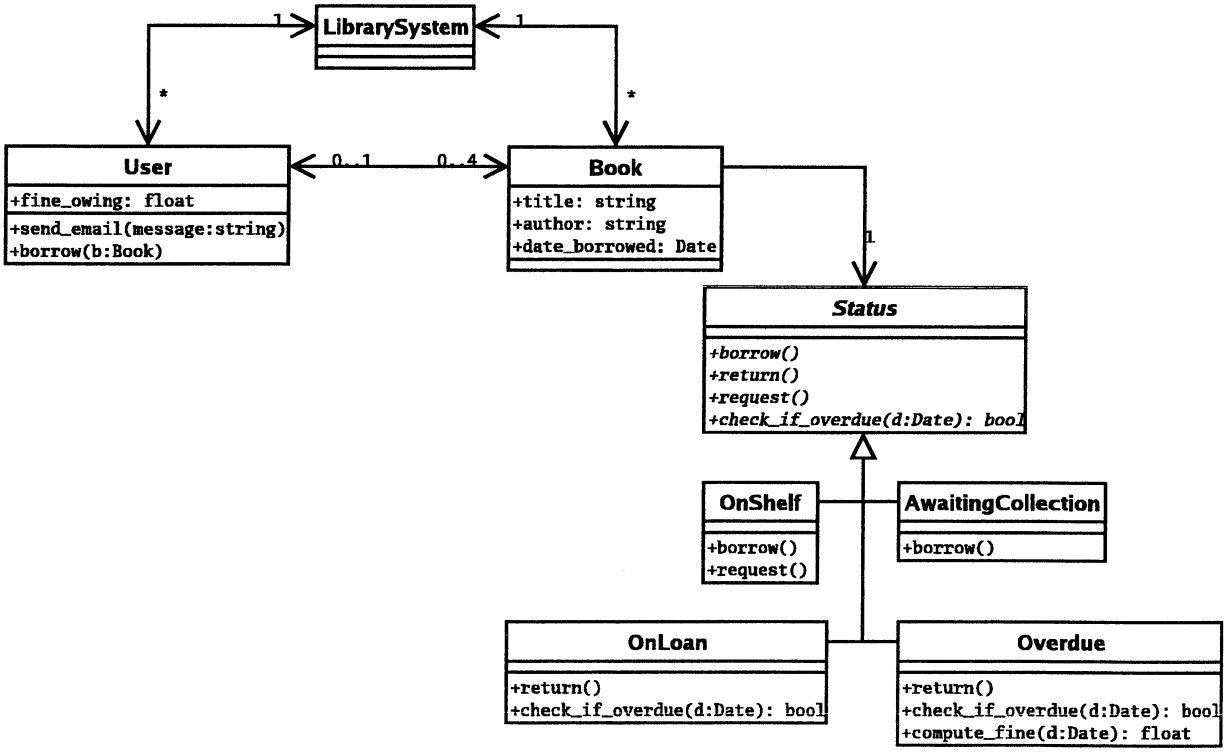
(e) A safety-critical system requires absolute attention to detail to be given in all the stages of the software process. A formal specification is often used, or a rigid development process such as Cleanroom. Reusable Components could be an appropriate process model, provided that the components themselves can be demonstrated to have been developed by such a process, and to be suitable for a safety-critical system, and that the whole system is developed as a critical system, with the appropriate safety specification and care in the system design. If the components have already successfully been used in a critical system, this can reduce the risk of this new project, and the need for extensive component testing. A substantial amount of integration testing, and system testing, will be needed as usual. [20%]

2. *Specification, Object-oriented design*

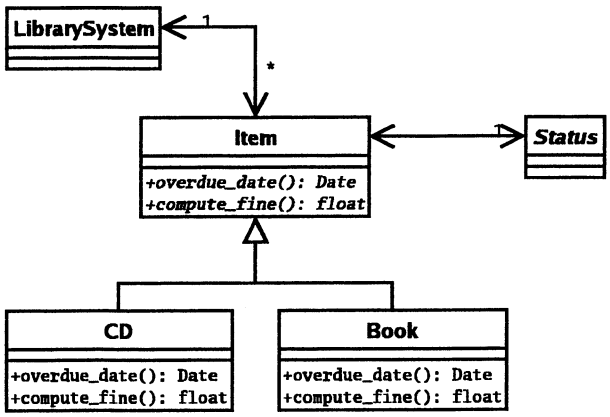
(a) *Encapsulation* and *data hiding* mean that an objects internal data is hidden and can only be modified by functions in the class of that object. This means that it is easier to ensure that the invariants of objects of a class are maintained (ie that data is in a valid state).

Class derivation and *polymorphism* allow complex classes to be derived from simpler ones. These complex classes can then be used in a function designed to operate with the simpler base class without modifying (or even recompiling) the function code.

(b) A class diagram depicting a suitable design is shown below. This design embodies the *state* pattern (books have a status which is one of four states). The abstract Status class shows four virtual functions, but in the subclasses only those functions where the default behaviour needs to be overridden are shown; Candidates may ligitimately choose to show all of these virtual functions in the derived classes.



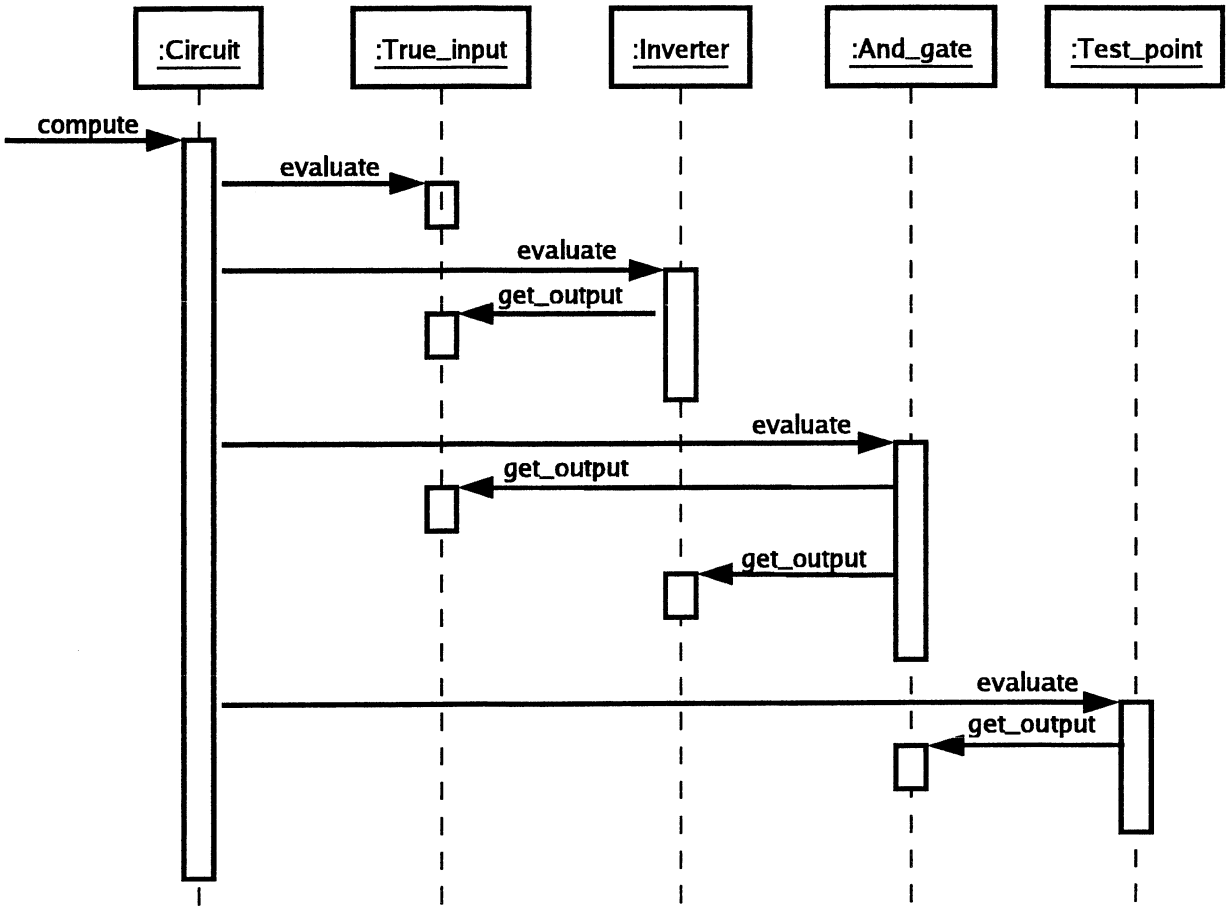
(c) In order to do this, the Book class must be replaced with a class hierarchy containing both Book and CD with a base class Item. In this hierarchy, the behaviours that differ between Book and CD must be implemented with virtual functions, thus the computation to see if an item is overdue, and the computation of the fine when an item is returned must be virtual. Class diagram shown below:



3. *UML, design patterns*

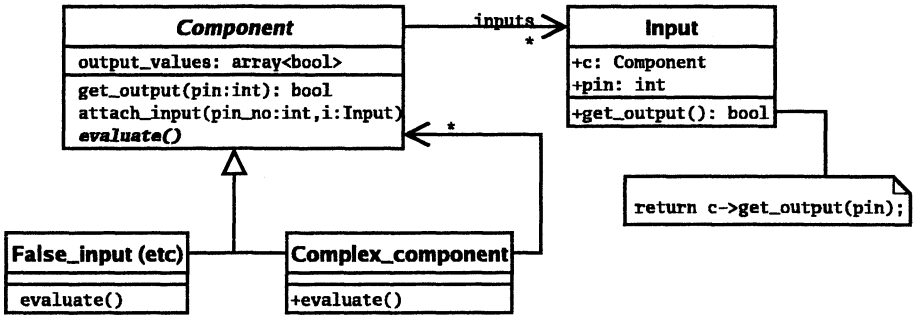
(a) There are ten classes present: Circuit, Component, False_input, True_input, Or_gate, And_gate, Nor_gate, Nand_gate, Inverter and Test_point. Each Circuit has many components and all the other classes are derived from component.

(b) Sequence diagram shown below



(c) In order for the Inverter, And_gate and Test_point components to compute their value, they have to call the get_output function on other components. If they do this before evaluate has been called on those components, there is no guarantee about the value returned by get_output. Hence the order in which evaluate is called on the components *does* matter.

(d) In order to allow JK flip-flops or adders, components with multiple outputs must be supported. This means that the output value field in Component must now store multiple output values (e.g. in an array). It also means that the inputs field in Component must be modified so that it refers to both a Component and an output pin on that Component. In the diagram below this is done by introducing a new class Input. Also the attach_input function has to be changed to reflect this. Finally, composite components must be supported using the *Composite* pattern. The UML class diagram is shown below:



4. *CORBA, concurrent systems, databases*

(a) A CORBA `.idl` file defines the interface(s) exposed to the network for objects which receive requests in a distributed system. It does this using a formal language which is independent of the programming language used for the implementation.

To implement a CORBA object, the `.idl` file is passed through an IDL compiler for the particular target language, which provides skeleton code to make an object with that interface. In the case of C++, the IDL compiler turns the `BookSheets` interface into the abstract C++ class `POA_BookSheets`, whose virtual functions are the same as the interface's methods. The `BookSeats_i` class derives from this base class, and provides the implementation of these functions. [15%]

(b) CORBA is naturally a concurrent system, since the client and server typically run on separate processors on separate machines on a network. One server can service numerous clients and, if it didn't handle requests concurrently, when two or more clients tried to access the server at the same time, one would have to wait for the other one. Concurrent processing guarantees that the server will (almost always) be able to service a request immediately. [10%]

(c) The `book_seat` method first checks that the seat is free and then, if so, it books that seat. If this method is called simultaneously by two clients, for the same empty seat, the first client will find that the seat is empty and will then start to book the seat. If the second client tests for an empty seat before the first client has written the name, it will also find the seat empty and will begin to book the seat. The booking by the first client will be over-written by the second, but the server will return a success code to both clients, and both will think they have the seat booked.

The solution is to ensure that only one client has access to the seat for the duration of its booking process (i.e. checking that the seat is empty, and then booking it)—the process must be made *atomic*. This can be done by using the semaphore for that seat, as provided in the class:

```
CORBA::Boolean BookSeats_i::book_seat(CORBA::Long seat_no,
                                      const char* name)
{
  // First wait until I have exclusive access to this seat
  my_seat_mutex[seat_no].wait();

  // Now test and set
  if(my_seats.name[seat_no] == "")
  {
    my_seats.name[seat_no] = string_dup(name);
    my_seat_mutex[seat_no].signal();  // Need to signal in both cases
```

```
    return true;
  }
  else
  {
    my_seat_mutex[seat_no].signal();  // Need to signal in both cases
    return false;
  }
}
```

[30%]

(d) To ensure backwards compatibility, the new interface should be *derived* from the existing interface, so that the original interface still exists and is usable. Defining a structure to hold the (up to 5) seats requested helps the efficiency of passing the parameters.

```
struct SeatBlock
{
  long num_seats;      // Number of seats being requested
  string names[5];     // Names and
  string seat_nos[5]; //   seat numbers
};

interface BookSeatsEx : BookSeats
{
  boolean book_seat_block(in SeatBlock block);
};
```

[20%]

(e) The `book_seat_block` function will necessarily have to book each of the seats in the block in turn. To ensure that it can successfully do this, it will first need to gain mutually-exclusive access to each of the seats, by locking each in turn. If another user is also trying to lock some of the same seats, *deadlock* can occur, with each thread waiting for the other to release its seats.

Booking multiple seats is, in database terms, an example of a *transaction.* Databases adopt a number of strategies for coping with deadlock, which can also be used here. A pessimistic strategy is to not use per-seat semaphores, but instead a single semaphore which grants exclusive access to the whole object (and thus all the seats). This the easiest solution in this case and, given that booking a block of seats is fairly quick, will not cause other clients to wait for an unacceptable length of time.

Other solutions would be to include the ability to recover from deadlock (by timing out after a while and trying again); or to adopt an optimistic strategy, which doesn't lock the seats at all before writing the names, but afterwards checks that they have been correctly written and if they haven't, unwinds the booking process. There is also an algorithmic solution: if seats are always locked in strictly numerical order, deadlock cannot occur. [10%]