PART IIA 2006
3F6: Software engineering and design
Principal Assessor: Dr T W Drummond

ENGINEERING TRIPOS PART IIA *Datasheet: None*

Module 3F6

SOFTWARE ENGINEERING AND DESIGN CRIB

STATIONERY REQUIREMENTS SPECIAL REQUIREMENTS

Version: 1

1 Concurrency and mutexes

(a) A multi-threaded program has two or more separate threads of execution. However, unlike a multi-process system, all threads of execution share a single common memory space. Multi-threading provides a natural framework for event-driven, real-time systems. The two principle hazards are:

1) contention in access to shared resources, especially memory.

2) deadlock caused by all threads blocked waiting for each other to release a lock or send a signal.

These are both failure modes. A third less catastrophic hazard is failing to meet real time responses due incorrect or inappropriate scheduling.

(b) A critical section is a block of code where it must be guaranteed that at most one thread is executing it at any one time. Usually the code in a critical section is dependent on accessing shared resources such as a shared global variable. The obvious approach to providing safe access is to maintain a variable which is set on entry to the section and cleared on exit. Eg

```
bool mutex;
while (mutex); mutex=true;
// critical section
mutex=false;
```

However, there are two problems with this. Firstly, it has to be assumed that the operations of reading and setting the mutex are indivisible. Secondly, the busy wait on the spin-lock "while(mutex)" is grossly inefficient. These problems are solved by making the locking and unlocking of mutex an operating system call. The OS can then ensure indivisibility of the critical operations and it can interact with the scheduler to ensure that blocked threads are simply put to sleep until the shared resource is made available.

(c) Implementations for the Put and Get operations are as follows:

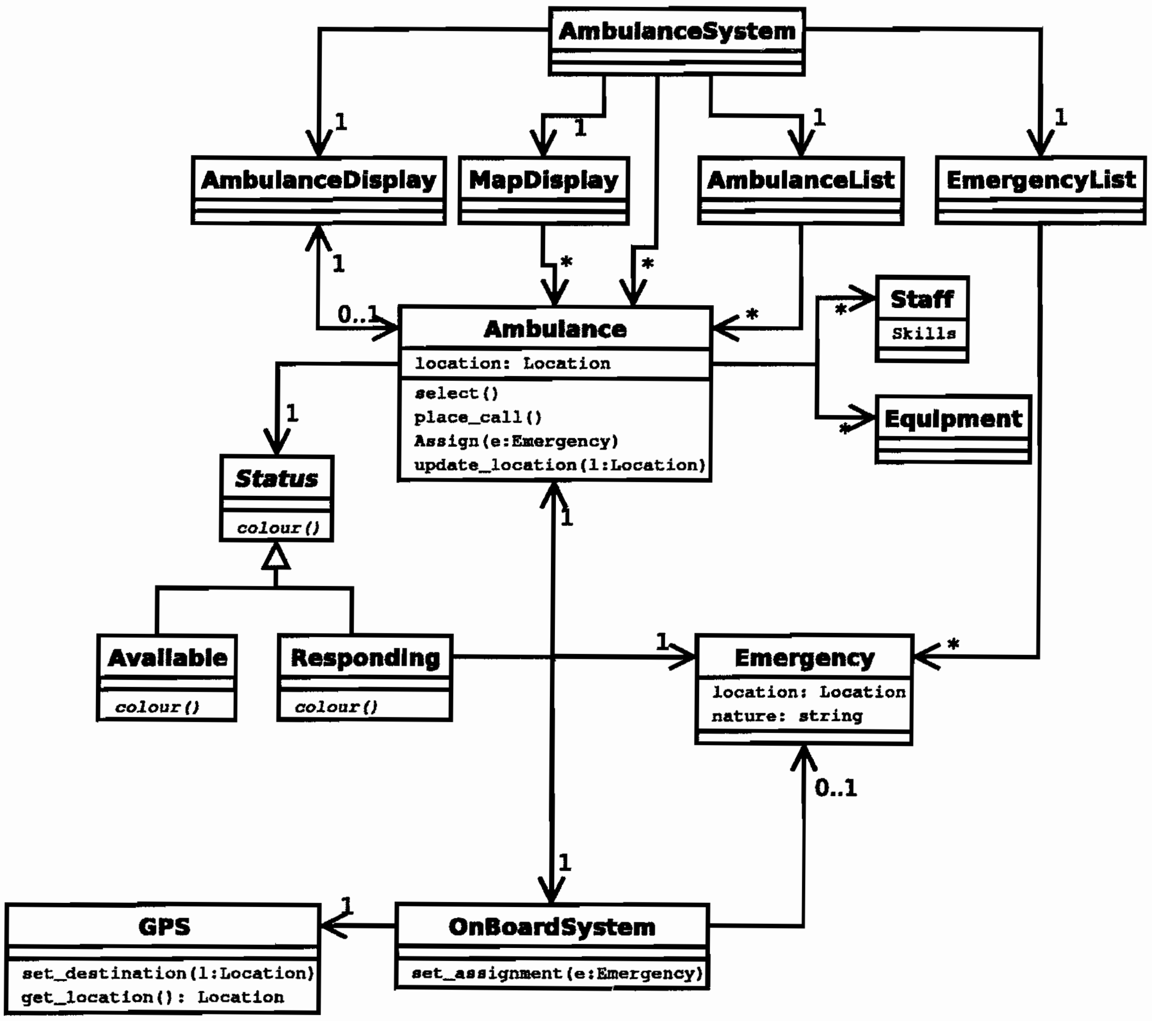
```
// class data - std circular buffer
Message buf[N];
int inx,outx,used;       // all init to zero
pthread_mutex_t lock;            // mutex lock
pthread_cond_t notFull, notEmpty  // signals

void Buffer::Put(Message m)
{
   pthread_mutex_lock(&lock);
   while (used == N) {
      pthread_cond_wait(&notFull, &lock);
   }
   buf[inx] = m; ++used;
   inx = (inx + 1) % N;
   pthread_cond_signal(&notEmpty);
   pthread_mutex_unlock(&lock);
}

Message Buffer::Get()
{
   Message m;
   pthread_mutex_lock(&lock);
   while (used == 0) {
      pthread_cond_wait(&notEmpty, &lock);
   }
   m = buf[outx]; --used;
   outx = (outx + 1) % N;
   pthread_cond_signal(&notFull);
   pthread_mutex_unlock(&lock);
   return m;
}
```

(TURN OVER for continuation of Question 1

2 Specifications and class diagrams.

(a) Bookwork: A virtual function is one that is dynamically determined by class of an object. It is specified in a base class from which each derived class may specify its own implementation. A common example would the method for drawing objects of different types in a Drawing program.

(b)



Note that the buffer full and empty conditions should be tested by while loops because there is no guarantee that the buffer state will not change between the time that the thread is reawoken and reaquiring the mutex lock.

(d) The difficulty with the above scheme is that a consumer thread must commit to getting a message from a buffer, and will block until one arrives. If the consumer is servicing multiple buffers this is unacceptable.

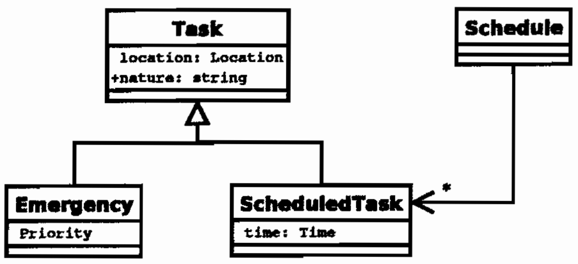
The simplest non-polling solution is to add an extra buffer control, whose messages are the buffer index. The i'th producer would then send a message m to the consumer by

```
control->Put(Message(i));
in[i]->Put(m);
```

the consumer would read incoming messages in two stages, first it would get the producer index from the control buffer, then it would read the message, i.e.

```
do {
   i = control->Get();
   m = in[i]->Get();
}
```

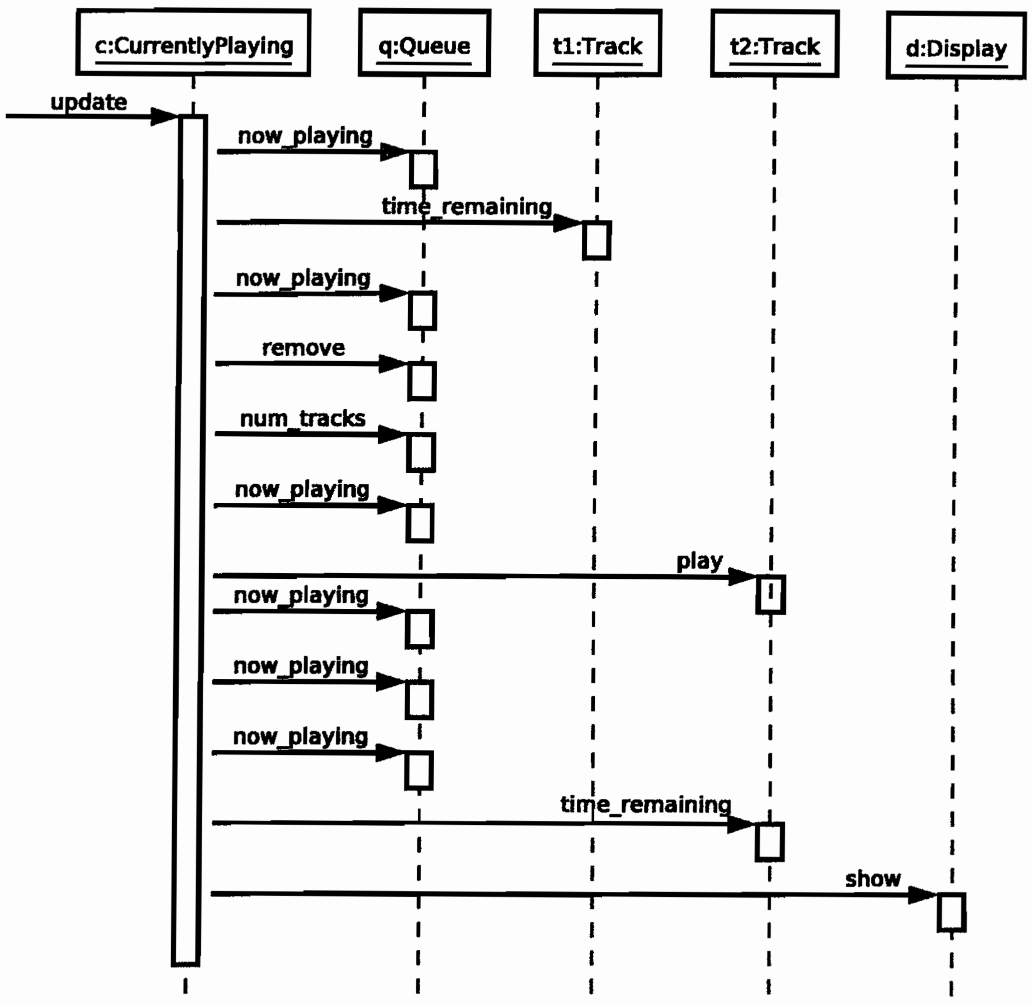
(c)


Task
location: Location
+nature: string
Schedule
Emergency
Priority
ScheduledTask
time: Time
*

3 Design patterns and sequence diagram

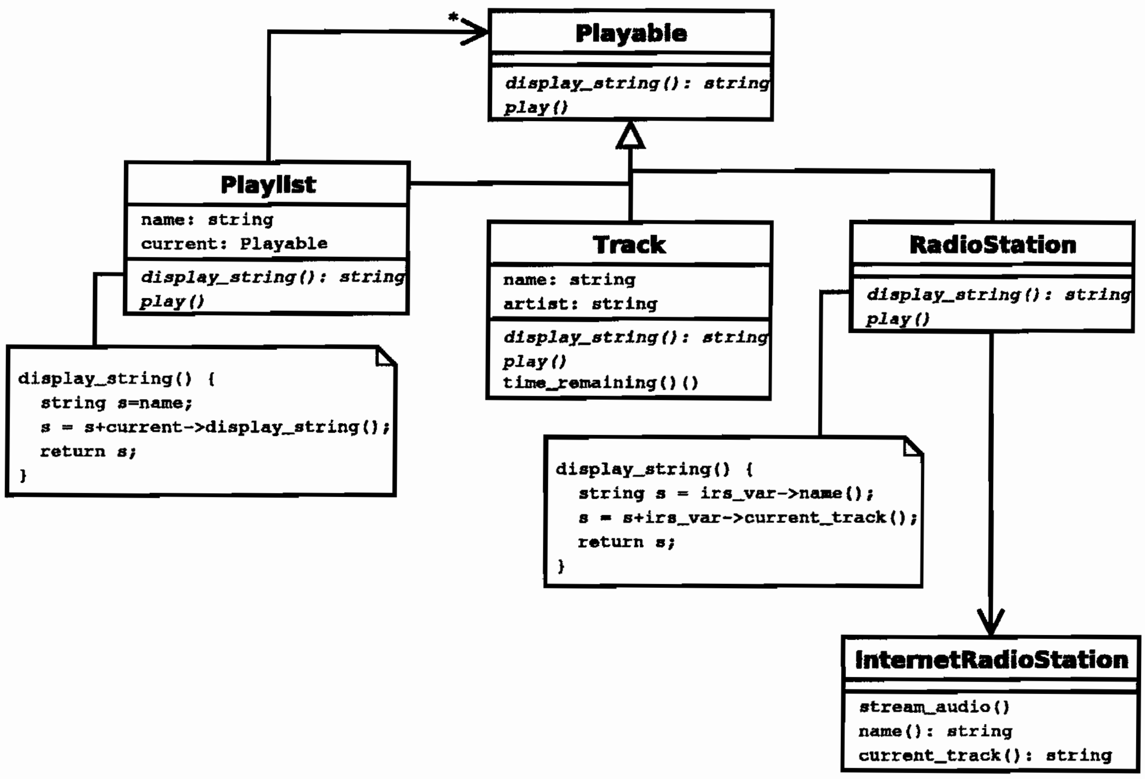
(a) There are seven classes present. Display, DisplayMode, TrackBrowser, CurrentlyPlaying, Library, Queue and Track. Each Display has a DisplayMode and each DisplayMode knows which display it belongs to. TrackBrowser and CurrentlyPlaying are derived from DisplayMode. A TrackBrowser has a (or can navigate to) a Library. Library has a (can navigate to the) Queue and both have many Tracks. Currently Playing has a Queue. Display, DisplayMode, CurrentlyPlaying and TrackBrowser constitute a State Pattern.

(b)



(TURN OVER for continuation of Question 3

(c) The main requirement is to turn Track into a class hierarchy. The Playlist class is most easily represented with a Composite Pattern.



```
display_string() {
  string s=name;
  s = s+current->display_string();
  return s;
}
```

```
display_string() {
  string s = irs_var->name();
  s = s+irs_var->current_track();
  return s;
}
```

4 bbb

END OF PAPER

(a) Formal methods permit the exact expression of the system specification in a mathematical or defined notation, so that automatic and verified tools can be used to transform the specification into error-free software components. This approach would be highly suited to the development of the core of the command and control system, which must be correct and bug-free.

Prototyping, whereby a cheap throw-away version of the system is built and then used to refine the specification in conjunction with the users and the customer, would be suitable for designing the maintenance system, which is not safety-critical in 'real time' but which must present data in a usable and unambiguous way.

(b) (i) Any three from: [book-work]

* a culture of quality
* information hiding and encapsulation
* use of strongly-typed languages
* design & program for readability and understandability
* avoid dangerous programming constructs
* use defined and documented development processes
* hold open code reviews and software inspections

(ii) In N-version programming, at least three versions of the system are designed, implemented and tested separately. These versions then execute in parallel and the outputs are compared in a voting system. Hopefully all will agree, but if not the error is recorded and the majority view is used.

(iii) [beyond book-work] Ideally, each version would be designed, implemented and tested by completely separate teams, sharing only the specification of the component. Within each team, the design and implementation of tests should not be carried out by the same people who design and implement the software.

Each team should use a different programming language, and each version should ideally run on a different operating system and hardware platform. In this way, the chances of more than one version encountering the same bug are minimised.

(c) User interaction permits the technicians to interrogate and update the monitoring system. The interface should aim to be familiar, consistent and straightforward. Since it will run on a laptop, it can make use of standard menu selection, forms and dialog components.

Information presentation will probably consist of graphs, dials and event lists. Judicious use of colour would assist in identifying warning and error conditions. The interface should be easy to learn, robust and quick to use.

If prototypes are constructed, the team can observe the technicians as they try out the prototype, and can adapt and refine the design of the user interface in collaboration with them.

3E6 Answer
Michael Gray

Draft
18 January 2006