UNIVERSITY OF LONDON
IMPERIAL COLLEGE OF SCIENCE, TECHNOLOGY AND MEDICINE

EXAMINATIONS 2004

MSc in Computing for Industry
MEng Honours Degree in Information Systems Engineering Part IV
MSci Honours Degree in Mathematics and Computer Science Part IV
MEng Honours Degrees in Computing Part IV
MSc in Advanced Computing
for Internal Students of the Imperial College of Science, Technology and Medicine

This paper is also taken for the relevant examinations for the Associateship of the City and Guilds of London Institute
This paper is also taken for the relevant examinations for the Associateship of the Royal College of Science

PAPER C475=I4.16

SOFTWARE ENGINEERING - ENVIRONMENTS

Tuesday 4 May 2004, 10:00
Duration: 120 minutes

Answer THREE questions

Paper contains 4 questions
Calculators not required

Section A(Use a separate answer book for this Section.)

1 **(Agile Programming)**

a "Although Software Engineering works for some projects it is inappropriate for the vast majority of software being developed today."

i) What is meant by Software Engineering in this context? Define it by listing three Software development processes that could be considered part of modern software engineering processes (not the waterfall lifecycle as this has been generally discredited).

ii) Is coding analogous to building / manufacturing in other engineering disciplines? What does the analogy lead to when planning the effort of a software project?

iii) Compare the standard Software Engineering view of change with the Agile view.

b i) Give an example of a problem which is not amenable to Agile programming techniques. What are the characteristics that make it unsuitable?

ii) Give an example of problem amenable to Agile programming techniques. What are the characteristics that make it suitable for Agile programming techniques?

c Several methodologies fit under the Agile banner. These include XP, Crystal, Open Source, Adaptive Software Development, Scrum, Feature Driven Development, Pragmatic Programming and Dynamic System Development Method. While all of them share many characteristics, there are also some significant differences. For your example in part b ii) choose two of these methodologies and describe in each case how implementation would proceed. For your problem which of the two methodologies would lead to better results? Justify your choice.

The three parts carry 45%, 30% and 25% of the marks, respectively.

2 **(Program Maintenance and Design)**

a Java was designed to support distributed program development explicitly. Library developers should be able to develop products for a client base who have no access to source code. According to the Java Language Specification

> "The Java language was designed to adapt to evolving environments. Classes are linked in as required and can be downloaded from across networks."

i) As a system evolves what happens to its client base?

ii) What problems arise do to having a distributed development environment?

iii) The basis of the design of Javas dynamic loading mechanism is its *anonymous/autonomous* philosophy. What does this mean?

iv) In order to implement this philosophy both *separate compilation* and *dynamic linking* are necessary. Why?

b Here is the Software Maintenance Process Model from the IEEE:

> 1) receive and log request, 2) analyze request and define requirements, 3) identify impact on test site, 4) formulate design and identify impact on test cases, 5)implement, test and system document, 6) document, issue and test.

Criticism of the IEEE maintenance model is that several essential steps are missing. List three steps that should have been included. For each of these steps describe what might go wrong with the maintenance process if your step is omitted.

c Information hiding is the idea that each module of a program is "characterized by its knowledge of a design decision which it hides from all others", as described by David Parnas in a seminal 1972 paper, "On the Criteria to Be Used in Decomposing Systems into Modules".

i) Since Parnas's paper this idea has arisen again and again in slightly different guises. Describe briefly three of these techniques.

ii) The FORTRAN programming language was designed well before Parnas's ideas became popular. Nonetheless there are constructs in FORTRAN that can be used to help with program modularization. What are they and how can they be used to aid modularization? Your answer can be in pseudo-code, concrete syntax is not necessary.

The three parts carry 40%, 30% and 30% of the marks, respectively.

Section B (Use a separate answer book for this Section.)

3 **(Alloy as a constraint-based declarative language)**

a Alloy uses `fun`-statements and `assert`-statements for analysis. In Alloy `fun`-statements have the general form

```
fun F(x1:T1, ..., xn:Tn){ C1 C2 ...  Cm }
```

where `xi` are variables of type `Ti` and `Ci` are constraints. Translate this general `F` into an `assert`-statement `G` such that the analysis of the `fun`-statement `F` and the analysis of the `assert`-statement `G` always render the same result. Justify that correspondence. Why would, or wouldn't, this translation be of use for invocations of `F`?

b Multiplicity constraints, e.g. "*at most one* failure occurs," are important declarative mechanisms but they may give rise to inconsistencies. Write a short Alloy module that uses three *different* explicit multiplicity keywords such that their interaction causes an inconsistency. Identify the location(s) of that inconsistency and suggest a way of resolving that inconsistency.

c We discussed how `fun`-statements can be used in several ways.

i) Consider the two `fun`-statement patterns

```
fun P(s,s':State) { C1 => C2 } run P for 3
fun Q(s,s':State) { C1 && C2 } run Q for 3
```

where `Ci` are constraints with `s` and `s'` as only free variables. Explain in what senses these patterns differ and give a prominent example use for each such pattern, justifying the use of the pattern for each example.

ii) A directed graph is *strongly connected* if, and only if, every node in that graph has a finite path to every node of that graph. Given the declarations

```
sig Element {}
sig DirectedGraph {
 nodes : set Element,
 edges : nodes -> nodes }
```

write a consistent constraint in Alloy whose analysis can only generate a strongly connected graph with exactly five nodes.

The three parts carry, respectively, 25%, 25%, and 50% of the marks.

4 **(Root contention protocol for IEEE 1394 ("FireWire"))**

The IEEE 1394 high-performance serial bus is "hot-plugable," allowing quick and dependable transfer of multi-media data.

a Given the Alloy declarations (*nodes* are any components connected to the bus)

```
sig Node { to, from :  set Link }
sig Link { target, source :  Node, reverse :  Link }
```

i) Add constraints that ensure "Each node has links connecting to other nodes. Each link has at most one connection."

ii) Add constraints that guarantee that the connection topology of `Node` is a rooted tree.

b Messages (sent along outgoing links) are stored in queues of length at most one with a possible overflow where `Msg` is a partition of scalars `Req` and `Ack`:

```
sig Queue { slot:  option Msg, overflow:  option Msg }
```

Write a parameterized constraint named `SameQueue` specifying that two queues are identical.

c Given that `Op` is a partition of scalars `Init, AssignParent, ReadReqOrAck, Elect, WriteReqOrAck, ResolveContention, Stutter` consider the state declaration

```
sig State {
part waiting, active, contending, elected:  set Node,
parentLinks :  set Link,
queue :  Link ->!  Queue,
op :  Op } -- the operation that produced the state
```

i) Write a parameterized constraint `SameState` stating that its two argument states are the same.

ii) In a transition function `fun Trans(s,s' :  State) { ... }` write constraint(s) for the case that the transition is caused by stuttering, where "stuttering" means that the state is idle.

d Considering that `s.elected` denotes the nodes elected as roots of the tree in state `s`, write a `fun`-statement `Evolution()` for linearizing the state space and use it to state, for all protocol runs, `assert`-statements `AtMostOneElected`, `OneEventuallyElected`, and `NoOverflow`.

The four parts carry, respectively, 25%, 15%, 25%, and 35% of the marks.