Maths - June 08

E2.12

IMPERIAL COLLEGE LONDON

DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING
EXAMINATIONS 2008

ISE PART II: MEng, BEng and ACGI

SOFTWARE ENGINEERING 2

Tuesday, 3 June 2:00 pm

Time allowed: 2:00 hours

There are FOUR questions on this paper.

Q1 is compulsory.
Answer Q1 and any two of questions 2-4.
Q1 carries 40% of the marks. Questions 2 to 4 carry equal marks (30% each).

Any special instructions for invigilators and information for candidates are on page 1.

Examiners responsible First Marker(s) : L.G. Madden, L.G. Madden
Second Marker(s) : J.V. Pitt, J.V. Pitt

The Questions.

1. **[Compulsory]**

(a) Examine the C++ code in Figure 1 and list an example of each the following:

(i) An *object* data member.

(ii) A member function that can *accidentally* break encapsulation. If there are none, then state NONE. [2]

(b) (i) List the members of the Orthodox Canonical Class Format (OCCF).

(ii) For each of the OCCF members explain in a sentence what it does.

(iii) For each of the OCCF members, explain briefly why it is important to include the OCCF member in every new class, even when there is no implementation code, as is often the case for one of the OCCF members.

(iv) Care must be taken to avoid *accidentally* breaking encapsulation with two of the OCCF members. Identify these members and explain briefly how it is possible to *accidentally* break encapsulation.

(v) The two OCCF members identified in part (iv) appear to have similar functionality. Focus on the lifecycle of an object to identify a feature that can be used to distinguish between them.

(vi) Why might we include OCCF in an abstract class? [6]

(c) After answering each of the parts (c i) - (c iii) below, if you think the new code also requires a change to the *specification* of `TInner` shown in Figure 1 then describe the change in words (or code), otherwise, state SPECIFICATION UNCHANGED.

(i) Write *implementation* code for an *insertion* operator, as a *friend* function, for the class `TInner` shown in Figure 1.

(ii) Write *implementation* code for an *extraction* operator, as an *ordinary* function, for the class `TInner` shown in Figure 1.

(iii) Write *implementation* code for an *assignment* operator, as a *member* function, for the class `TInner` shown in Figure 1.

(iv) What does it mean to *knowingly* break encapsulation? Which C++ syntactic feature facilitates this concept? [8]

(d) (i) Using the C++ implementation code in Figure 1, draw a UML class diagram. You do not need to include any members not shown in the code.

(ii) Draw a UML sequence interaction diagram for the statement labelled `#3` in the `main()` function in Figure 1 showing all object collaborations. [4]

```
class TInner{
 public:
   TInner(void):value(0)                { }
   TInner(int value):value(value)       { }
   int getValue(void)                   { return value; }
   void increment(void)                 { value++; }

 private:
   int value;
};

class TOuter{
 public:
   TOuter(TInner &x):theObject(x)   { }

   int getValue(void){
      return this->theObject.getValue();
   }

   void increment(void){
      this->theObject.increment();
   }

   TOuter& shallowCopy(void){
      return *this;
   }

   TOuter& midCopy(void){
      TOuter outMid(this->theObject);
      return outMid;
   }

   TOuter& deepCopy(void){
      TInner inDeep(this->theObject.getValue());
      TOuter outDeep(inDeep);
      return outDeep;
   }

 private:
   TInner &theObject;
};

int main(void){
  TInner theObject(42);                 // #1
  TOuter theSource(theObject);          // #2
  theSource.deepCopy();                 // #3 - Q1. d(ii)
}
```

Figure 1

2. The aim of this question is to explore the defensive programming mechanisms available in C++.

(a) Figure 2.1 shows the implementation code for a more robust version of the library (ordinary) function for calculating square roots. The code will compile successfully in a C or a C++ compiler. Figure 2.3 demonstrates how the function can be called with a valid and an invalid argument.

(i) Draw a UML activity diagram (i.e. not a flowchart) for the algorithm implemented in the C++ code shown in Figure 2.1.

(ii) Translate the UML class diagram shown in Figure 2.2 into C++ implementation code for the exception `IllegalArgumentException` [6]

(b) For the parts (b i) - (b iv) below you should provide modified code equivalent to the lines labelled `#1-#6` of the *called* code (Figure 2.1). Each change to the implementation of the `newSqrt()` will have consequences on how `newSqrt()` is called. For each part (b i) - (b iv) you should *also* provide the modified code equivalent to lines `#1-#5` of the *calling* code (Figure 2.3).

(i) Modify `newSqrt()` to remove the residual C and make proper use of standard C++ conditional logic.

(ii) Modify `newSqrt()` to use an assertion instead of returning an error value.

(iii) Modify `newSqrt()` to use the exception created in part (a ii) to handle an illegal value i.e. `IllegalArgumentException` instead of an assertion.

(iv) Modify `newSqrt()` to use the design principle, design by contract (DbC). [8]

(c) (i) Why does a generic program facilitate Reuse? Why does a generic program facilitate Adaptability?

(ii) In generic programming the protocol of a STL container class differs from the protocol that would used in a conventional object oriented approach. Briefly, describe this difference.

(iii) Illustrate your answer to part (b ii) by drawing a UML class diagram for the container classes *as if* they were implemented using a conventional object oriented approach i.e. not the actual generic programming approach. Your class diagram should include *at least* the following :

- Concrete classes corresponding to some of the STL container classes.
- Abstract classes, that you should invent,
- and the following member functions:
  `sort(), front(), make_pair(), push()` [6]

```
double newSqrt(double arg){
// Precondition: none.
// Postcondition: if argument valid returns the square root
//                otherwise returns -1.
   errno = 0;                                    // #1
   double ans = sqrt(arg);                       // #2
   if (errno == 0)                               // #3
     return ans;                                 // #4
   else                                          // #5
     return -1.0;                                // #6
}
```

Figure 2.1

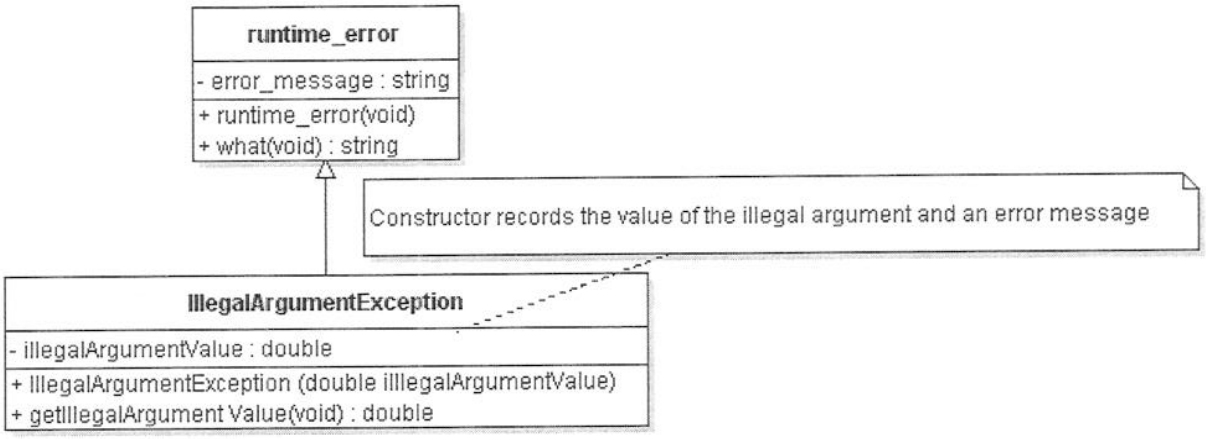

Figure 2.2

```
int main(){
   double arg, ans;
   cout << "Enter a value: ";
   cin >> arg;

   ans = newSqrt(arg);                           // #1
   if (ans != -1.0)                              // #2
      printf("newSqrt1: %lf\n", ans);            // #3
   else                                          // #4
      perror("newSqrt: argument error");         // #5
}
```

Figure 2.3

3. The aim of this question is to explore the expressiveness of the UML V1.0

Partial requirements specification for an analogue filter:

> High-pass and low-pass filters are both kinds of an analogue filter. An analogue filter consists of a potential divider circuit, which in turn consists of exactly two components, where each component has a complex number representation. A potential divider circuit is a kind of linear circuit which uses matrix algebra for circuit analysis.

(a) (i) Draw a UML class diagram, based upon a textual analysis of the requirements specification above. Include as much detail and precision as can be derived *only* from the textual description.

(ii) List the basic relationships between classes in object oriented analysis and design (OOAD). Illustrate your answer with an example of each relationship found in the class diagram produced for part (a i).

(iii) List the basic relationships between classes in object oriented programming (OOP). Illustrate your answer with short excerpts of C++ code demonstrating each relationship using the classes and relationships identified in part (a i).

(iv) List the kind of details that you would expect to see in an implementation model class diagram, but would not expect to see in a conceptual model class diagram. [8]

(b) For each of the two code excerpts listed in Figures 3.1(a) and (b) draw a UML class diagram (only involving `Member` and `College`) that accurately represents the C++ code that is shown.

Note: `Member` and `College` are user-written classes. A college identifier (CID) e.g. `cid1` is a string that uniquely identifies a member of the college. The member function `Student::getCollegeId()` returns the CID for the receiver. [4]

(c) Figure 3.2 provides a partial summary of the subject of relational algebra. Without any background in relational algebra, but with knowledge of UML, it should be possible to deduce from the class diagram some facts about relational algebra.

Express in plain English (i.e. avoid UML and OOAD terminology) *four* facts about relational algebra that can be deduced from the diagram. Pay close attention to the numerical values in the diagram. [4]

(d) Examine the class diagram in Figure 3.3(a) and the object diagram in Figures 3.3(b). State whether the object diagram is valid or not valid with respect to the class diagram. Justify briefly your answer. [1]

(e) Using you own class names draw class diagrams demonstrating each of the following:

(i) An association class.
(ii) A derived association.
(iii) A bi-directional association in a conceptual model. [3]

(a)
```
typedef vector <Member> College;
College members;
String cid1 = "abc123";
College::iterator index;
//enrol students e.g.
members.push_back(Member(cid1)); // etc…
// locate specific student
for (index=members.begin(); index != members.end(); index++)
    if ((*index).getCollegeId() == cid1)
        cout << (*index); // member found
```

(b)
```
typedef map <string, Member> College;
College members;
String cid1 = "abc123";
//enrol students e.g.
members.insert(make_pair(cid1,Member(cid1))); // etc…
// locate specific student
cout << members[cid1]; // member found
```

Figure 3.1

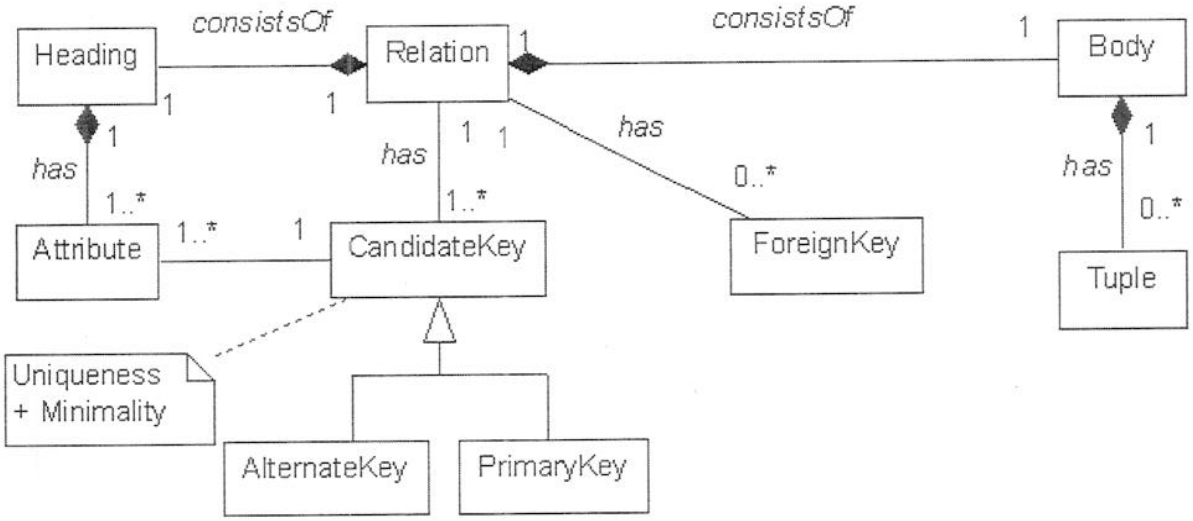

Figure 3.2

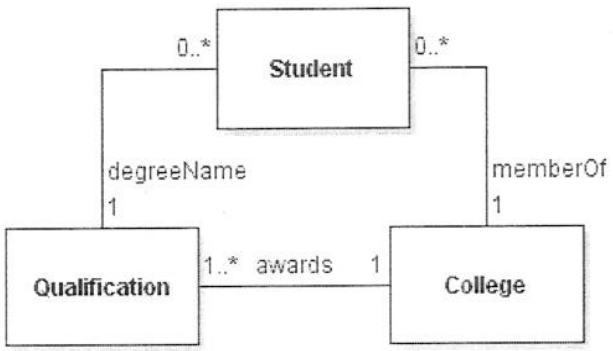

Figure 3.3(a)

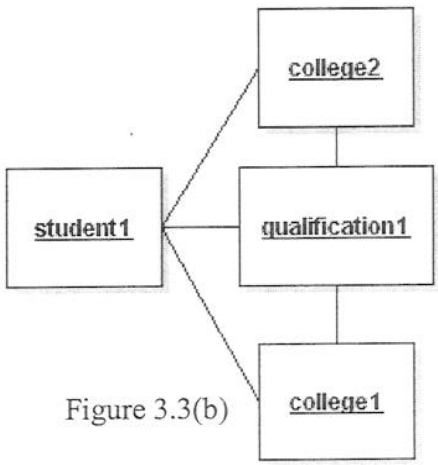

Figure 3.3(b)

4. The aim of this question is to explore the object oriented analysis and design (OOAD) process.

Partial requirements specification for an event-driven digital logic simulator (DLS):

> A DLS simulates the behaviour of a circuit. Each circuit has a number of logic gates. A gate has a fixed delay (measured in nanoseconds) and is named (e.g. "`nand2`"). There are several different kinds of gates, for example, an AND-gate is a kind of gate. Each gate evaluates a specific boolean function (e.g. AND, inclusive/exclusive OR etc...) after the fixed delay. Each gate has one output node and has one or more input nodes. A node is named (e.g. "`input32`") and is designated as an input or an output and it maintains a logical state e.g. `high`, `low`, or `undefined`. Each node has a list of events associated with it. An event records information including the logic state it will transition to and the time of the transition.

The DLS project has reached an advanced stage of development. However, the implementation model is incomplete and requires reverse engineering to complete the model.

(a) (i) Figure 4.1 is a partial Use case diagram for the DLS. By focusing upon the syntactic elements of the diagram it is possible to extract information about the DLS. Using bullet-points, describe several details about the DLS that can be discovered in the diagram. These details augment the requirements specification shown above.

(ii) Write a formal textual description for the "Set Input High" use case, including a success and failure scenario.

(iii) What is test driven development (TDD)? Describe how TDD should be implemented during object oriented analysis and design (OOAD).

Given the activities above, is there evidence to suggest that TDD has not been properly implemented throughout this project? Justify your answer. [9]

(b) (i) List some techniques that may be used to identify the basic architecture from a requirements specification. Using the requirements specification above, give one example of each kind of detail that can be discovered in the requirements specification and what technique was used to discover it.

(ii) The basic architecture has been translated into the skeleton C++ code shown in Figure 4.2.Translate the given code into a detailed class diagram that is appropriate to an implementation model. [6]

(c) The code shown below is an extract from the `main()` program for the DLS. Draw the corresponding sequence interaction diagram.

```
Gate gate21 = new Gate("and2", 3);
string s = gate21.getName();
```

[2]

(d) Draw a UML activity diagram showing the complete object oriented analysis and design process i.e. from requirements elicitation to implementation. The diagram should include the different models produced and the major activities performed. [3]

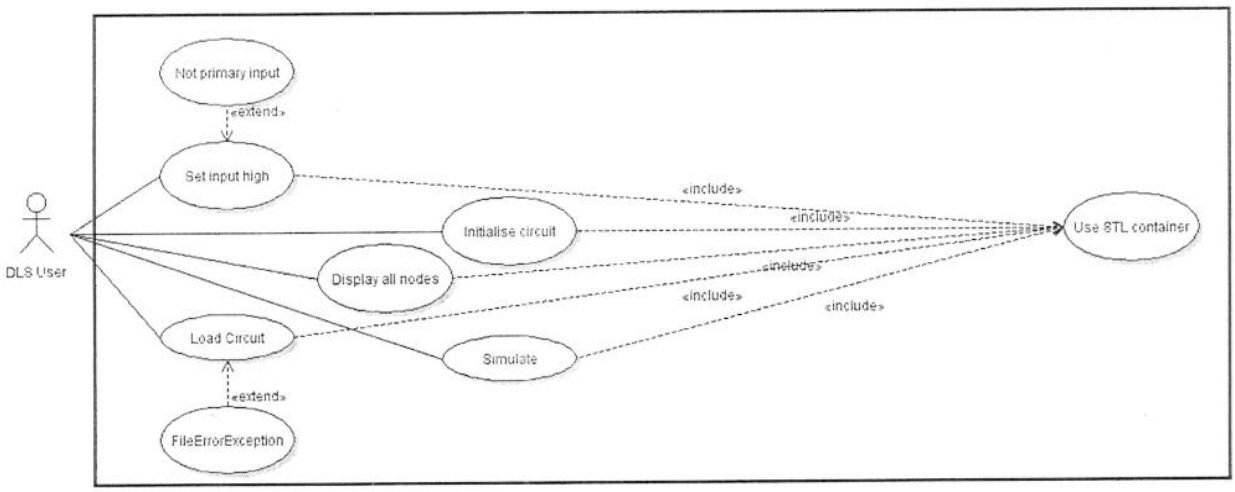

Figure 4.1

```
typedef enum {high, low, undefined} TNodeType;
typedef enum {input, output} TInOut;

class TEvent{
private:
   TNodeType newState;
   int atTime;
};

class TNode{
private:
   string name;
   TNodeType state;
   TInOut type;
   TEvent eventList[];  // no dimension as yet
};

class TGate{
public:
   virtual bool evaluateBoolean(void);
protected:
   string name;
   int gateDelay;
   TNode input;
   TNode output[];  // no dimension as yet
};

class TAndGate : public TGate{
public:
   bool evaluateBoolean(void);
};

class TCircuit{
private:
    TGate gateList[];  // no dimension as yet
};
```

Figure 4.2