SOFTWARE ENGINEERING 2: OBJECT ORIENTED SOFTWARE ENGINEERING

1. This is a general question about C++ and Object Oriented Software Engineering.

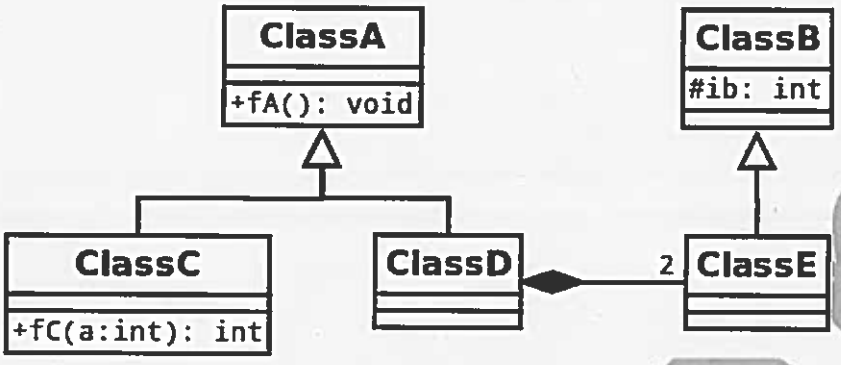

Figure 1.1 An UML diagram

a) Describe in words the software architecture represented in the diagram in Fig. 1.1.

[6]

[new example]

The diagram contains five classes. Classes ClassC and ClassD inherit from classA and class ClassE inherits from ClassB. Classes ClassD and ClassE are in a relationship of composition (each object instance of ClassD "has" two objects instances of ClassE). ClassA has a public member function fA which has no parameters and doesn't return anything. ClassC has a public member function fC with an int parameter and an int return value. ClassB has a protected member data field ib of type int.

Allocation of marks:

Classes: 1

Member data and member functions: 2.5

Inheritance: 1

Composition: 1.5

b) Write C++ declarations for all the classes in the UML diagram in Fig. 1.1. The declarations can be kept to the essential skeleton (e.g. constructors can be omitted) but all the elements related to available information (including relationships) should be included.

[9]

[new example, programming]

```
class ClassA {
```

```
    public:
        void fA();
};

class ClassB {
    protected:
        int ib;
};

class ClassC : public ClassA {
    public:
        int fC(int a);
};

class ClassE : public ClassB {

};

class ClassD : public ClassA {
    private:
        ClassE ce1;
        ClassE ce2;
};
```

Allocation of marks:

Classes: 2

Inheritance: 2.5

Member data in ClassD (composition): 2

Member data and member functions in other classes (including parameters, return types, access modifiers): 2.5

Perfect syntax is not required, for instance it doesn't matter whether or not all the semicolons are in place. But the students need to show competence of some defining syntax constructs such as the colon to express inheritance.

c) We want to create an example of polymorphism based on `ClassB` and `ClassE` from the software architecture in Fig. 1.1.

i) Explain why this requires a change in these classes.

[3]

ii) Write C++ code for an amended version of the declaration of the classes that would be suitable for an example of polymorphism.

[3]

iii) Write C++ code that could e.g. be in the `main` and that would represent an example of polymorphism based on `ClassB` and `ClassE`.

[3]

[new example, programming]

i) In order to have an example of polymorphism the base class `ClassB` should include a virtual member function that is then overridden in the derived class `ClassE`.

ii) This could be reflected in the declaration for instance as follows:

```
class ClassB {
    protected:
        int ib;

    public:
        virtual void fP();
};

class ClassE : public ClassB {

    public:
        virtual void fP();
};
```

iii) In the `main` an example of polymorphism might look like this:

```
ClassB* o = new ClassE;
o->fp();
```

The key issues in question 1.c are about showing understanding of which components are needed in order to have an operational example of polymorphism and enough competence in terms of syntax and implementative details working with pointers (or references). As in the question above, perfect syntax (semicolons etc) is not required.

d) i) List all the access modifiers available in C++ for member data and member functions and explain their meaning.

[4]

ii) Explain which access modifiers are most often associated respectively to member data and member functions and why.

[4]

[bookwork]

i) C++ has three access modifiers: public, private, protected. Fields which are public can be accessed from inside and outside the class. Fields which are private only from inside the same class. Even subclasses of a certain class cannot access its private fields. Fields which are protected can be accessed only from within the

same class and its subclasses.
Allocation of marks:
List of all the keywords: 1
Explanation of each keyword: 1 each (3 in total)

ii) Member data are usually declared as private or protected, while member functions are usually declared as public. This is because member data represent the state of the object instance of a certain class and we want the state to be encapsulated and only accessible from the outside through the abstraction provided by member functions.
Allocation of marks:
Distinction of the cases: 1
Explanation of why: 3

e) Consider the insertion (<<) operator.

i) Describe the cases (if any) in which its overloading is defined as member function and the cases (if any) in which it is defined as global function. [3]

ii) Discuss why it is so. [5]

[bookwork]

i) The insertion operator is defined as member function of ostream for basic types such as int, double etc. For all other cases, including objects from the standard library such as strings and user defined types, it is defined as a global function (sometimes as friend function of the type for which it is overloaded).

ii) The reason for this is that a binary operator defined as a member function needs to be a member function of the type of the left hand side operand. However in the case of the insertion operator the lhs tends to be always of type ostream while the overloading is on the type of the right hand side operand. If the operator was to be overloaded by being defined as member function for a user defined type this would entail having to change the code from the standard library for ostream, while defining it as a global (possibly friend) function doesn't require this.

The key issues in question 1.e are about understanding the way operator overloading works, in particular related to the knowledge that for an infix binary operator the lhs is either the first parameter of a corresponding function with two parameters or the object on which a member function is called.

Allocation of marks for 1.e.i:

Mention of the member function case: 1.5

Mention of the global/friend function case: 1.5

Allocation of marks for 1.e.ii:

Explanation of << as binary operand (in which the ostream object is the lhs) with reference to the key issue outlined above: 2.5

Explanation of the implications in terms of implementation: 2.5

2. This question deals with C++ templates and the Standard Template Library.

a) i) Write a template function which, given in input an `std::list` containing elements of a generic type, changes the list so that its minimum and maximum elements are swapped.

[12]

ii) Explain which are the characteristics that the generic type mentioned above (i.e. the type of the elements contained in an `std::list` given in input to this function) must have.

[6]

[new example, programming, workbook]

i)
```
template<typename T>
void swapminmaxl(std::list<T>& v){
    typename std::list<T>::iterator min, max, idx=v.begin();
    if(v.begin() != v.end()){
        min = idx;
        max = idx;
        for(; idx != v.end(); ++idx){
            if(*idx < *min){
                min = idx;
            }
            if(*max < *idx){
                max = idx;
            }
        }

        T tmp = *min;
        *min = *max;
        *max = tmp;
    }
}
```

Allocation of marks:
Loop and update algorithm: 3
Swap algorithm: 2
Correct use and syntax for templates: 3.5
Correct use and syntax for iterators: 3.5

ii) The type of the elements needs to have an order relationship defined and the less than (<) operator appropriately overloaded. Moreover the elements need to be swappable, which for instance means that the assignment (=) operator needs to be appropriately overloaded.
Allocation of marks:
Mention of less than operator, with explanation: 1, 2 (3 in total)
Mention of assignment operator, with explanation: 1, 2 (3 in total)

b) i) Discuss the main difference between the iterators that can be used (respectively) with `std::vector` and with `std::list`.

[2]

ii) Illustrate your answer using C++ code.

[5]

[bookwork, programming]

i) The iterators that can be used on vectors allow random access, while those that can be used on lists can only be used to iterate from one element to the next.

ii) For instance in code:

```
std::vector<int> v;
std::list<int> l;

//...
// assuming content is added at this point to both containers
//...

std::vector<int>::iterator itv = v.begin();
std::list<int>::iterator itl = l.begin();

itv++; // ok, next element
itv=itv+3; // ok, random access on vector iterators

itl++; // ok, next element
// itl=itl+3; no, no random access on list iterators
```

In question 2.b.ii students should show competence about writing basic code using iterators, including the distinction between operations involving random access iterators and sequential access iterators. As above, perfect syntax (semicolons etc) is not required, but competence of aspects of syntax which are core to the question (e.g. how to declare and initialise an object of iterator type) is relevant and part of the assessment.

c) Explain what is a `const_iterator` and when it is used.

[5]

[bookwork]

A `const_iterator` is a type of iterator that can be used for read-only access to a container. It is used for instance when we need to iterate through a container which is declared as `const` (for instance because it is passed by const reference) because in that case the use of an iterator with both read and write access would not be allowed by the compiler.

Allocation of marks:

What it is: 2

When it is used: 3

3. This question deals with Java.

a) Compare and contrast how the process of turning code into an executable program works in C++ and in Java. Explain how this affects the portability of Java applications.

[8]

[bookwork]

In C++, code in one or several source files is first compiled into several so-called object files which are then linked into an executable which can be directly executed by a machine of the target architecture of the compiler. In Java, when source code is compiled, a ".class" file is created for each of the classes in the code and these files contain an intermediate representation (bytecode). The bytecode is not architecture-specific and cannot be directly executed. It can be executed on machines of potentially any architecture by the Java virtual machine for that architecture.

Allocation of marks:

Explanation of the process in C++: 2.5

Explanation of the process in Java: 2.5

Explanation of Java portability: 3

b) Write Java code (roughly equivalent to the C++ code requested in question 1.b) for all the classes in the UML diagram in Fig. 1.1. The body of the functions can be kept empty.

[8]

[new example, programming]

```
class ClassA {
    public void fA(){}
}

class ClassB {
    protected int ib;
}

class ClassC extends ClassA {

        public int fC(int a) {}
}

class ClassE extends ClassB {

}

class ClassD extends ClassA {
        private ClassE ce1;
        private ClassE ce2;
```

```
}
```

Allocation of marks:

Classes: 2

Inheritance: 2

Member data in ClassD (composition): 2

Member data and member functions in other classes (including parameters, return types, access modifiers): 2

As above, perfect syntax is not required (semicolons etc), however the students need to show competence of some defining syntax constructs in particular as far as they are specific to Java, e.g. the `extends` keyword.

c) Explain why Java doesn't have the initialisation list feature that C++ has.

[7]

[bookwork]

In Java objects are always constructed dynamically and are always represented in classes as pointers/handles to the memory area hosting the object therefore there is no need for a feature like the C++ initialisation list, whose purpose is to (correctly) construct objects before the actual scope of the constructor function body is entered.

Allocation of marks:

Explanation of how objects are constructed in Java: 3

Explanation of how this relates to the initialisation list in C++: 4

d) Explain what `interfaces` are in Java and how they are used.

[7]

[bookwork]

An interface in Java is akin to a C++ abstract class in which all the member functions are pure virtual. Similarly to abstract classes, the role of an interface is to state which methods will be provided by other classes which implement that interface although the behaviour will differ depending on the specific class implementing the interface and is not specified in the interface itself. Interfaces can be used as type declarations that will be then associated to objects instance of a specific class implementing the interface and this allows polymorphic calls of the methods declared in the interface.

The key issues in this question are about operationally describing the characteristics of an interface (the methods are not defined) and explaining how and why this is useful (using them as types in polymorphism).

Allocation of marks:

Explanation of what: 2.5

Explanation of how: 4.5