SOFTWARE ENGINEERING 2: OBJECT ORIENTED SOFTWARE ENGINEERING

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

   a) Consider the concept of *state* in Object Oriented Software Engineering.

      i) Explain what is meant by “state of an object”. Illustrate your answer using an example in C++ code dealing with points on the Cartesian plane.

      [6]

[bookwork, programming]

Objects represent entities and include several data fields which contribute to model the entity. For instance a point on the Cartesian plane has two coordinates, thus a class whose instances are objects representing points will have member data in order to represent the coordinates as follows:

```
class point{
    private:
        double x;
        double y;
    ...
};
```

At different stages in the life cycle of the object these member data will probably have different values which represent different states of the object (in this case, corresponding to different positions on the plane).

[Most students provided a valid answer, although not all establishing clear descriptive links between the explanation and the example in code.]

      ii) Explain what “inconsistent state” means in this context. Illustrate your answer using an example in C++ code expanding on the previous one.

      [6]

[bookwork, programming]

The state of an object can be represented using member data whose values are mutually related, for instance a point might be represented including also, besides its coordinates, its distance from the origin:

```
class point{
    public:
```

```
        double x;
        double y;
        double distance_origin;
    ...
};
```

Since in this example these fields are public, somewhere in the code an instance of this class could be set to an inconsistent state, e.g. as follows:

```
point p;
p.x = 0;
p.y = 0;
p.distance_origin = 1;
```

[A few students did not provide a valid explanation or example, presenting instead just cases in which member data are modified, e.g. because public, but not to a state that could be considered inconsistent according to some criteria.]

iii) Explain how encapsulation and abstraction can avoid inconsistent states while keeping the object mutable. Illustrate your answer using an example in C++ code expanding on the previous one.

[6]

[bookwork, programming]

The use of encapsulation (in this case using the visibility modifier `private:`) prevents the member data from being directly accessible from outside the class. Following the principle of abstraction, the object can be kept mutable while keeping the state consistent by e.g. defining for the coordinates setter member functions which also update the member data representing the distance from the origin:

```
class point{
    private:
        double x;
        double y;
        double distance_origin;
    public:
        void set_x(double ix){
            x = ix;
            distance_origin = sqrt(x*x + y*y);
        }
        void set_y(double iy){
            y = iy;
            distance_origin = sqrt(x*x + y*y);
        }
    ...
};
```

[Most students correctly explained encapsulation, although many gave vague answers as far as abstraction is concerned and some did not provide a meaningful example.]

b) Consider an application domain related to geometric entities, in particular dealing with triangles, circles and points. Shapes are represented in terms of points (and other attributes when needed). A translation operation should be available for any entity (effect of applying a geometric vector expressed by a point). For instance if a point `p1` is at coordinates (1, 2) and a point `p2` is at coordinates (3, 4), after translating `p1` by `p2`, `p1` should be at coordinates (4, 6).

i) Describe in words how you would model this domain in an object oriented architecture.

[6]

[bookwork, new example]

This domain could be modeled with a concrete class `Point` and concrete classes `Triangle` and `Circle` inheriting from an abstract class `Shape`. A method `translate` would be defined in `Point`, declared as abstract in the base class and overridden in the derived classes.

The derived classes would contain objects instance of class `Point` (composition), in particular class `Triangle` would contain three points for the vertices, while `Circle` would contain one point for the center and another numerical attribute for the radius.

The translation in the derived classes of `Shape` would be implemented by delegation using the same operation defined in class `Point`: a `Triangle` is translated by translating the points representing its vertices, a `Circle` by translating the point representing its center.

[Most students got the main outline of the architecture description correctly, although not everyone in a clear or complete way.]

ii) Draw a UML class diagram of the architecture.

[6]

[bookwork, new example]

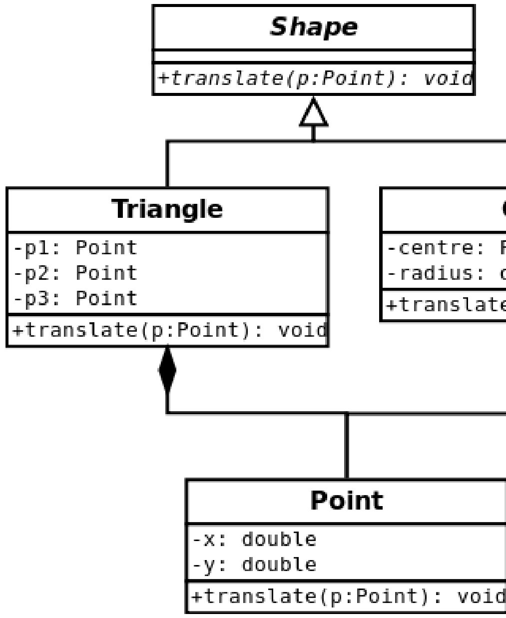

Figure 1.1 Often attributes that are involved in composition relationships are not also included in the class box: the diamond arrow is annotated instead. Either way is considered acceptable in this case. The parameter for function translate is not marked as const reference in the diagram as this aspect is primarily part of the implementation rather than the design, but this information may be included too.

[Most students sketched the main outline and features of the UML diagram correctly, although in some cases the diagram was left quite incomplete.]

iii) Write a declaration for all the classes. The declarations can be kept to the essential skeleton (e.g. constructors can be omitted) but all the relevant elements needed in order to express the architecture should be included. Moreover, include the definition (where needed) for the member function which implements the translation.

[10]

[bookwork, new example, programming]

```
class Shape {
    virtual void translate(const Point& p) = 0;
};

class Triangle : public Shape {
    public:
        void translate(const Point& p){
            p1.translate(p);
            p2.translate(p);
            p3.translate(p);
        }
```

```
    private:
        Point p1;
        Point p2;
        Point p3;
};

class Circle : public Shape {
    public:
        void translate(const Point& p){
            centre.translate(p);
        }
    private:
        Point centre;
        double radius;
};

class Point {
    public:
        void translate(const Point& p){
            x = x + p.x;
            y = y + p.y;
        }
    private:
        double x;
        double y;
};
```

[A relatively common mistake here: not using delegation (from composition) correctly and altering the coordinates of the points at a low level, e.g. using setters, instead of using member function translate.]

2. This question deals with container classes and memory management in C++. Write the implementation (you can keep the definition and the declaration together) of a container class representing a *stack* of integers, i.e. a data structure on which elements (in this case integer numbers) can be pushed (making the stack grow) and from which elements can be popped (making the stack shrink). Stacks are "last in, first out" data structures.

a) Declare suitable member data for the class. The implementation should be based on a dynamically allocated array.

[3]

b) Define a constructor which takes as argument an integer representing the initial physical size of the stack.

[4]

c) Define the copy constructor.

[4]

d) Define the assignment operator.

[6]

e) Define the destructor.

[3]

f) Define a `push` member function which models the operation of pushing an element (passed as argument) on the stack.

[6]

g) Define a `pop` member function which models the operation of popping an element (which is also returned) off the stack. The attempt to pop an element off an empty stack does not need to be handled, it is assumed that it may result in undefined behaviour.

[3]

h) Define an `empty` member function which returns true if the stack is empty and false otherwise.

[1]

[bookwork, new example, programming]

```
class intstack{
    public:

        // b)
        intstack(int iphsize) : phsize(iphsize), lsize(0) {
            data = new int[phsize];
        }

        // c)
        intstack(const intstack& s){
            lsize = s.lsize;
            phsize = s.phsize;
            data = new int[phsize];
            for(int i = 0; i < lsize; i++){
                data[i] = s.data[i];
            }
        }

        // d)
        intstack& operator=(const intstack& s){
            if(s.data != data){
                delete[] data;
                lsize = s.lsize;
                phsize = s.phsize;
                data = new int[phsize];
                for(int i = 0; i < lsize; i++){
```

```
                    data[i] = s.data[i];
                }
            }
            return *this;
        }

        // e)
        ~intstack(){
            delete[] data;
        }

        // f)
        void push(int n){
            if(phsize == lsize){
                phsize = phsize * 2;
                int* tmp = new int[phsize];
                for(int i = 0; i < lsize; i++){
                    tmp[i] = data[i];
                }
                delete[] data;
                data = tmp;
            }
            data[lsize] = n;
            lsize = lsize + 1;
        }

        // g)
        int pop(){
            lsize = lsize - 1;
            return data[lsize];
        }

        // h)
        bool empty() const {
            return (lsize == 0);
        }

    // a)
    private:
        int* data;
        int phsize;
        int lsize;
};
```

[Students who remembered from the lecture notes how to build a (basic, proof of concept) example of container class based on a dynamically allocated array using abstraction and encapsulation, generally got the outline of the answer right. Some common mistakes on details: using delete instead of delete[], getting some indices wrong, resizing when not needed. Students who didn't remember the fundamental aspects of this kind of task mostly just went off track.]

3. This question deals with C++ templates and related concepts.

a) Explain why the headers where template classes and functions are declared usually also contain their definitions.

[8]

[bookwork]

A template class or function cannot be compiled “as is”: an instance of the actual class or function with the actual type or types needs to be generated (instantiated) and this happens when the compiler finds a call of such a function or an instantiation of such a class in some source file.

At this stage, the code of the function or the class needs to be available in order to allow the compiler to generate the actual code from the template, thus this cannot be in a different source file (source files are compiled individually before being linked): it needs to be in the same unit. This is usually achieved by having also the definitions for template classes and functions in the header file which contains their declarations: when such a header is included (by the source file using the template functions or classes declared and defined in it) all the needed information is available to the compiler.

[Some students erroneously commented that this is just a matter of convenience or readability. Some students got the outline of the answer right but did not explain with sufficient clarity or detail.]

b) The Standard Template Library includes a container template class `pair` with two (public) member data fields, `first` and `second` (not necessarily both of the same type). Write a skeleton of this class including the member data and a constructor initializing them. Show with a code snippet how the class can be instantiated and used (e.g. in the main).

[8]

[bookwork, new example, programming]

```
template<typename T1, typename T2>
class m_pair{
    public:
        m_pair(const T1& f, const T2& s) : first(f), second(s) { }
        T1 first;
        T2 second;
};

    ...
    m_pair<int, bool> mp(3, true);
    if(mp.second){
        cout << mp.first << endl;
    }
    else{
        cout << -mp.first << endl;
    }
    ...
```

> [Relatively common mistakes: not including the types of the template class instantiation, i.e. `m_pair mp(3, true);` instead of `m_pair<int, bool> mp(3, true);` and declaring the member data fields as private although it is stated in the question that they should be public.]

c) Using the following code as a starting point, explain why iterators are useful illustrating your answer with an example in C++ code.

Let `items` be a `vector` (e.g. containing some integers).

```
for(int i = 0; i < items.size(); i++){
    cout << items[i] << endl;
}
```

[8]

> [bookwork, programming]
>
> We consider a `for` loop using indexing of a `vector`:
>
> ```
> for(int i = 0; i < items.size(); i++){
>     cout << items[i] << endl;
> }
> ```
>
> However if later on we change the design of the program and choose a different container class for `items`, e.g. `list`, the loop would need to be entirely rewritten. Using iterators, changes are more localized because the generic action of iterating on a container is abstracted (the availability of forward iterators is still needed in this case):
>
> ```
> vector<int>::iterator it;
> // or
> // list<int>::iterator it;
>
> // same in both cases
> for(it = items.begin(); it != items.end(); ++it){
>     cout << *it << endl;
> }
> ```
>
> [Students who remembered the concept of iterators mostly provided a correct example, although not everyone explained in a clear way the advantages of using them. Students who did not remember mostly seemed to erroneously think that "iterators" was used as synonymous for indexing.]

d) Consider the (global) function `sort` included in the header `<algorithm>` which takes as arguments the initial and final iterators of a sequence (contained in a container class) in order to sort its elements. Describe the conditions that the container and its elements need to respect in order to be used with this function.

[6]

[bookwork]

The elements included in the container need to be of some type providing an overloading of operator less than [although there is also an overloaded version of the `sort` function which takes one additional argument in input, i.e. a function that can be used to perform the comparison instead of the less than operator]. The container needs to have random access iterators (for instance this function cannot be used with list containers).

[Many students mentioned at least one of these aspects.]