SOFTWARE ENGINEERING 2: OBJECT ORIENTED SOFTWARE ENGINEERING

1. Consider the UML class diagram in Fig. 1.1 (member data are omitted).

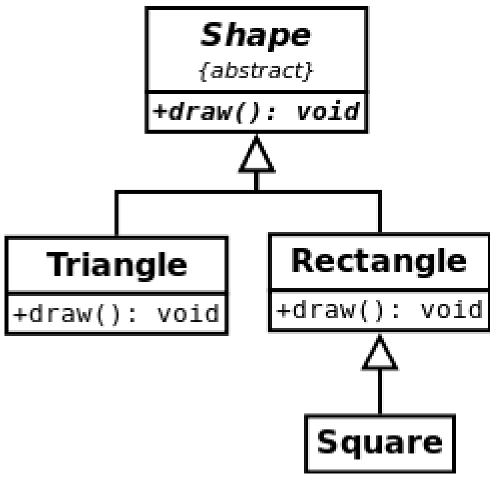

Figure 1.1 The UML diagram to consider for the question

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

[4]

[bookwork, example analysis]

The diagram represents an abstract base class Shape with a pure virtual void member function draw which has no parameters. Classes Triangle and Rectangle inherit from Shape and override function draw. Class Square inherits from Rectangle.

b) Write in C++ code the declaration of all the classes and functions visible in the diagram.

[5]

[bookwork, example analysis, programming skills]

```
class Shape{
    public:
    virtual void draw() =0;
};

class Triangle : public Shape {
    public:
    void draw();
};
```

```
class Rectangle : public Shape {
    public:
    void draw();
};

class Square : public Rectangle {

};
```

c) Explain what upcasting is. Give an example in C++ code using the classes in the diagram.

[7]

[bookwork, example analysis, programming skills]

Inheritance is (syntactically) an "is-a" relationship between the derived class and the base class. Upcasting in this context means that when a class inherits from a base class, it is possible to syntactically use (pointers and references to) the type of the base class also for objects instance of the derived class.

For example:

```
Shape* s = new Triangle;
```

d) Explain how the Liskov substitution principle should affect the design of inheritance hierarchies.

[6]

[bookwork]

The Liskov substitution principle prescribes that inheritance hierarchies (for mutable objects) should be designed so that using objects instance of derived classes in place of objects instance of base classes keeps the desirable properties of the program.

The derived class should always (behaviourally and semantically) be a specialization or extension of the base class.

e) Assuming objects instances of the classes in the diagram to be mutable, explain why the architecture respects (or does not respect) the Liskov substitution principle.

[5]

[bookwork, example analysis]

The architecture does not respect the principle because although a Square is geometrically a kind of Rectangle, since its representation as an object introduces a constraint with respect to class Rectangle (all the sides of a square need to be equal) then we can't consider class Square to be an extension of class Rectangle and thus there can be cases in which it is not be suitable to use an object instance of Square instead of an object instance of Rectangle.

f) Explain what polymorphism is. Give an example in C++ code using the classes in the diagram.

[8]

[bookwork, example analysis, programming skills]

Consider a base class and a derived class featuring a member function with the same name and parameter list but with a different behaviour. Polymorphism is a language programming feature which means that, when the function is called on an object instance of the derived class through a pointer or reference to the base class, the behaviour of the function in the derived class is obtained.

For example in:

```
Shape* s1 = new Triangle;
Shape* s2 = new Rectangle;

s1->draw();
s2->draw();
```

The draw member function, called on two pointers to objects of the same (abstract) class (Shape) will have a different behaviour in the two cases (Triangle and Rectangle).

g) Explain how polymorphism is useful to write extensible code with an example related to the architecture in the diagram.

[5]

[bookwork, example analysis]

In the architecture of the example it is likely that other classes inheriting from Shape (e.g. Circle) will be defined in the future. If a certain function has a parameter of type reference to Shape on which it calls member function draw, because of polymorphism it will be able to work consistently also with objects of type Circle.

2. Consider C++ templates and the Standard Template Library (STL).

a) Briefly explain what is the purpose of template functions and classes in C++. Expand on your explanation using commented examples of definition and use of a template function and of declaration (a stub is enough) and use of a template class.

[12]

[bookwork, programming skills]

Templates are used in C++ in order to write generic functions and classes.

For example when an object instance of the following class which represents a complex number is instantiated, the type of the member data containing the real and imaginary parts of the number can be chosen (at compile time) according to the needed numerical precision (float, double etc).

```
template <typename T>
class Complex{
    private:

    T re;
    T im;

    // other member data and member functions

};
```

The class is used to instance objects as follows:

```
Complex<double> z;
```

The following function can swap objects of any type on which the default constructor and the assignment operator are defined:

```
template <typename T>
void swap(T& e1, T& e2){
    T tmp;
    tmp = e1;
    e1 = e2;
    e2 = tmp;
}
```

It is used as follows:

```
string s1("a"), s2("b");
swap(s1, s2);
```

b) What is the purpose of the Standard Template Library? What does it contain? Illustrate your answer providing examples in code.

[11]

[bookwork, programming skills]

The purpose of the STL is to provide an implementation of: generic container classes, algorithms which operate on the containers, and iterators to access their elements.

An example of container class is vector, for instance:

```
vector<int> v;
```

declares v as a vector (a resizeable sequence of contiguous memory locations) of int variables.

An example of algorithm is sort. For instance:

```
sort(v.begin(), v.end());
```

sorts the elements in vector v.

The line above also shows examples of iterators: member functions begin() and end() return respectively iterators to the beginning and to the end of the vector.

c) Are generic programming and object oriented programming mutually incompatible? Illustrate your answer with an example using the classes in Fig 1.1.

[7]

[bookwork, programming skills]

Generic programming and object oriented programming are based on different and arguably divergent principles, however in practice it is common to make use of a combination of the two, consider for instance iteration on the elements of a Standard Template Library container class (generic programming) calling on each element a polymorphic function (object oriented programming):

```
vector<Shape*> v;

v.push_back(new Triangle);
v.push_back(new Rectangle);

for(vector<Shape*>::iterator it=v.begin(); it!=v.end(); ++it){
    (*it)->draw();
}
```

3. Consider C++ exceptions.

a) Does the compiler perform any checks on exceptions in C++? Why is it so? What does it entail? Is it any different in other programming languages?

[8]

[bookwork]

In C++ the compiler doesn't perform any checks on exceptions.

This design choice is probably encouraged by the fact that exceptions weren't in the language from the beginning and the consequence is that C++ programmers working on code which makes use of exceptions need to be careful and rely on experience and documentation in order to recognize which instructions might throw exceptions.

In Java, for instance, some classes of exceptions are checked by the compiler.

b) Explain what is the purpose of exceptions in C++ and show an example of their use.

[11]

[bookwork, programming skills]

The purpose of exceptions is writing robust programs which can deal with unexpected occurrences without the need to return error codes and avoiding the proliferation of conditional statements.

For instance a function which computes the square root of a number could throw an exception if the argument is a negative number:

```
double except_sqrt(double n){
    if(n<0){
        throw invalid_argument("negative number");
    }
    return sqrt(n);
}
```

The exception should be caught as follows:

```
try{
    result = except_sqrt(num);
}
catch(const invalid_argument& e){
    cout << "couldn't compute square root: " << e.what() << endl;
    // further exception handling e.g. setting result
}
```

c) What is exception safe code? What is the Resource Acquisition As Initialization idiom? How is it related to exceptions?

[11]

[bookwork]

Exception safe code keeps the state of the domain of the application consistent and handles resources accordingly (e.g. flushing buffers, releasing locks) even if an exception is thrown.

In the RAII idiom a resource is acquired by the constructor of an associated object and it is released by its destructor.

The paradigm works well with exceptions because the destructor of objects instantiated in a function is still called even if the function throws an exception and its execution is interrupted (as long as the exception is handled e.g. by the caller).