Please note the following:

- Q4 (b) and (d) of the 2003 exam (the first presentation of E2.12) includes some XML and Distributed computing issues which have been dropped from the course content in order to include more material on UML and REUSE.

- Question 4(c) concerns the translation of Java into a UML Activity diagram. This activity is still examinable, however, if it appears in the future it will use C^{++} and not Java.

- The solutions provided here are often much more verbose than would be expected in the exam. This will be clarified in the summer term revision lecture.

- The only change to content of the exam is that Q4 is now all about REUSE (in all its forms). Thus the paper will consist of the following:

  - Q1 (compulsory and worth 40%) examines **all the course contents**, then you may select 2 out of 3 (each worth 30%) i.e.

    - Q2 **C^{++}**

    - Q3 **UML**

    - Q4 **REUSE**

IMPERIAL COLLEGE LONDON

DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING
EXAMINATIONS 2003

ISE II: M. Eng., B.Eng. and ACGI

SOFTWARE ENGINEERING 2

12 June 2003, 2pm

Time allowed: 2:00 hours

There are FOUR questions on this paper.

Q1 is compulsory.
Answer Q1 and any two of the questions 2-4.

Q1 carries 40% of the marks. Questions 2 to 4 carry equal marks.

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

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

1. (a) What are the three main characteristics of an Object Oriented Programming language?

[3]

(b) A feature of some Object Oriented languages (e.g. Smalltalk) is that every new class is integrated into a core class hierarchy, whilst other Object Oriented languages (e.g. C^{++}) do not have a core class hierarchy. Describe one advantage and one disadvantage in C^{++} as a result of this feature.

[2]

(c) Give one advantage and one disadvantage of the Iterative and Incremental development process model.

[2]

(d) Give four categories of classification that may be used for identification of candidate classes during modelling. Give an example of each category in the context of sitting this exam.

[4]

(e) What is a Qualified Association? Explain why is it beneficial to use a Qualified Association. Give an example of a Qualified Association using UML.

[4]

(f) Explain how the proper application of the guiding principles of High Cohesion and Low Coupling could have helped reduce the consequences of the Y2K problem, i.e. changing the date representation in a large monolithic program.

[2]

1. (g) The following questions are all based upon observation of the Borland C^{++} Builder (BCB) Integrated Development Environment:

(i) Identify the architectural style used in compiling a C^{++} source file under BCB.

[1]

(ii) A framework is domain specific, what type of domain does BCB address?

[1]

(iii) Identify those elements of BCB that can be described as Software Components.

[1]

2. (a) Explain why the C^{++} code shown in Figure 2.1 could form the basis of a robust Graphical User Interface developed in Borland C^{++} Builder.

[4]

(b) (i) Use the class diagram shown in Figure 2.2 to write C^{++} source code for defining the three classes as they should appear in an interface file. You can assume a suitable constructor already exists. The UML stereotypes provide additional C^{++} specific implementation details.

[4]

(ii) Write a short code extract showing how concrete filter objects (as defined in Figure 2.2) are created using a polymorphic variable.

[2]

(c) (i) Briefly explain why the C^{++} program shown in Figure 2.3 is a good demonstration of reusable and adaptable code.

[2]

(ii) Rewrite the C^{++} `Component` class definition so that the data members of the class `std::complex` use template classes instead of the type `double`. You may write stubs for the member function bodies.

[3]

(iii) Write a short C^{++} code extract showing how `Component` objects can be created with different actual datatypes for the data members of the class `std::complex` e.g. `float`, `double` and `long double`.

[1]

(d) What is the Common Grandparent problem in C^{++}? Illustrate your answer with a UML diagram.

[4]

```
try {
    double d = StrToFloat(Edit1->Text);
    Form1->Caption = "Good Input";
}
catch (EConvertError &e {
// ShowMessage() displays the argument in a message box.
    ShowMessage("Bad Input");
}
```

Figure 2.1

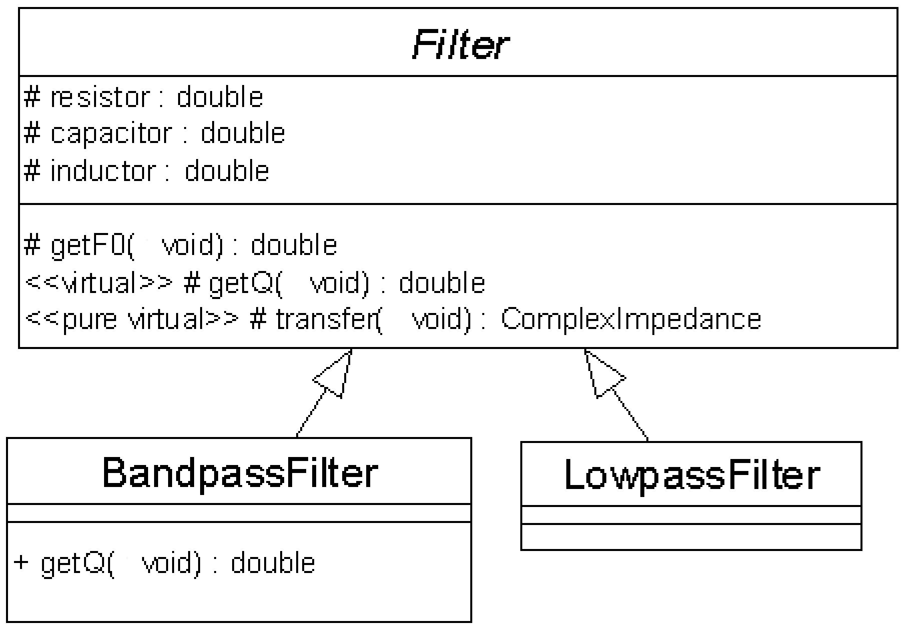


Figure 2.2

```
class Component {
  public:
    Component(double r, double i):z(r,i) {}

    complex<double> series(Component z1){
      return this->z + z1.z;}

    complex<double> parallel(Component z1){
      return pow((pow(this->z,-1)+pow(z1.z,-1)),-1);}

    complex<double> potDiv(Component z1){
      return z1.z/(this->z+z1.z);}

  private:
    complex<double> z;
};
```

Figure 2.3

3. (a) Perform a textual analysis of the text below to identify the classes, then draw a Class-Association diagram:

> A quadraphonic audio system consists of an amplifier which is connected with many kinds of audio player devices. Players include cd, record, cassette and radio tuner. Each player device connects to a single amplifier. The amplifier is connected to 4 speakers and each speaker is connected to a single ampilifier. Amplifiers are classified as preamplifier, power amplifier or integrated (i.e. contains both a preamplifier and a power amplifier).

[6]

(b) Draw a Use-Case diagram from the perspective of a user of a standard laboratory signal generator. Include at least three Use-Cases. You can assume that the system will be implemented in software.

[2]

(c) Identify three different features of Figure 3.1 that suggest that this is *not* an Initial Object Model.

[3]

(d) Convert the Sequence Collaboration diagram shown in Figure 3.2 into a Sequence Interaction diagram.

[5]

(e) Convert the C^{++} code shown in Figure 3.3 into a Class-Association diagram.

[4]

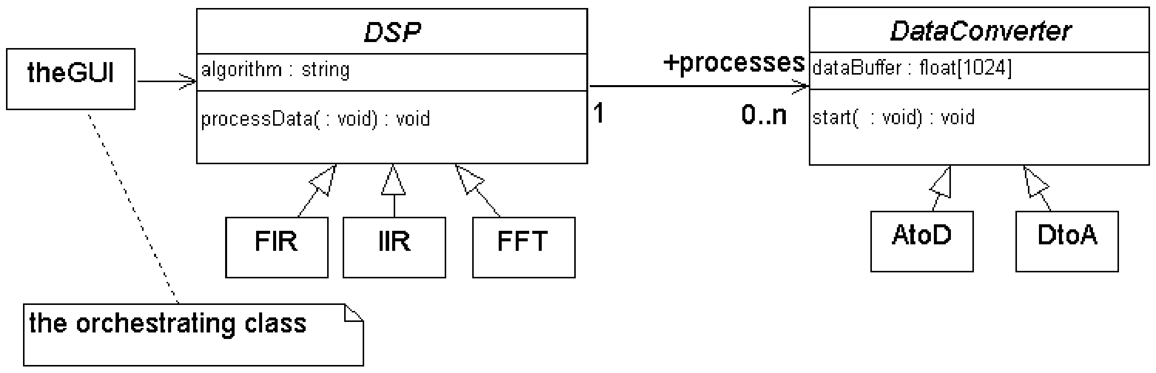


Figure 3.1

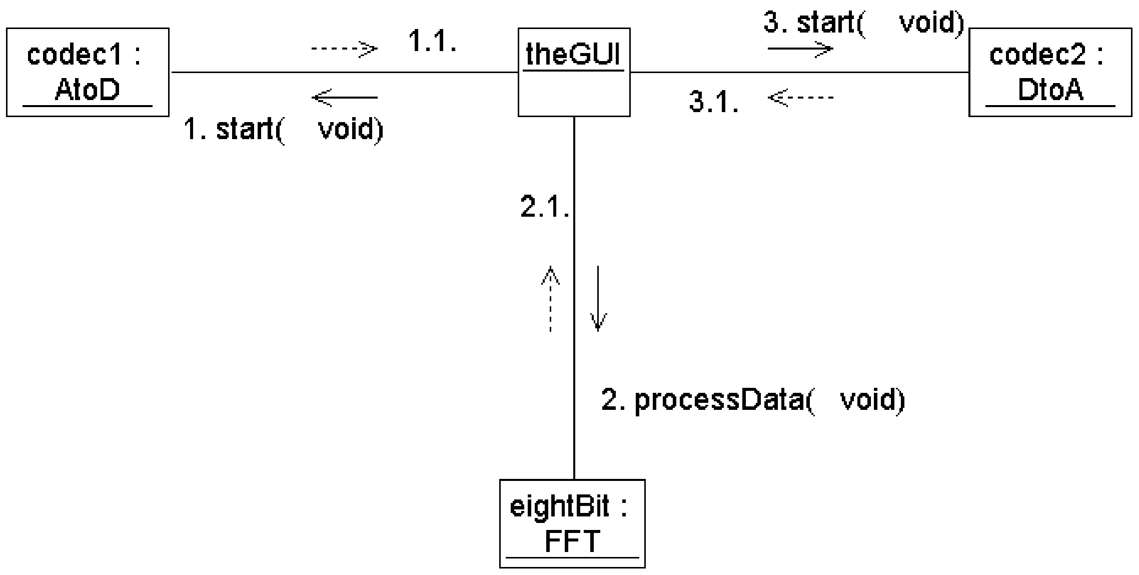


Figure 3.2

```
class FullAdder{
};

class FourBitAdderSubtractor{
  public:
      FullAdder  *includes;
};

class FourBitAdder:
               public FourBitAdderSubtractor{
};

class FourBitSubtractor:
               public FourBitAdderSubtractor{
};
```

Figure 3.3

4. (a) (i) Briefly describe what it means to say that "Design patterns are discovered (i.e. mined) not invented".

[2]

(ii) Name one design pattern and briefly describe what problem it solves.

[2]

(b) Explain what it means to say that a Distributed Computing System can be distinguished by how far it distances itself from the underlying details of the network.

[4]

(c) The Java program shown in Figure 4.1 is a very basic WWW server application. Use a UML activity diagram to highlight the key elements of this server application, which is typical of any server in a client-server architecture.

[4]

(d) The XML DTD shown in Figure 4.2 provides a partial description for a typical Object Oriented class definition.

(i) Suggest three potential benefits that might be gained from expressing a class definition in XML.

[3]

(ii) Using the DTD in Figure 4.2 write some XML data for the abstract Filter class shown in Figure 2.2 of Question 2.

[5]

```
Socket s;
ServerSocket ss;

int port = 8080; // experimental WWW server port number

ss = new ServerSocket(port);

while(true){
    s = ss.accept();

    BufferedReader br=new BufferedReader(
                new InputStreamReader(s.getInputStream()));
    PrintWriter pw=new
                     PrintWriter(s.getOutputStream(),true);

    String read1=br.readLine();
    bool notDone=true;
    while(notDone){
        read1 = br.readLine();
        if(read1.equals(""))//blank line terminates request
           notDone=false;
     }

    pw.print("<HTML><BODY><P>EEE Dept.</P></BODY></HTML>";
    // more print messages would appear here …

    pw.close(); br.close(); s.close();
}
```

Figure 4.1

```
<!-- DTD for a defining a CLASS -->
<!DOCTYPE CLASSDEFINITION[
                        <!ELEMENT CLASSDEFINITION (CLASS)*>

<!ELEMENT CLASS
            (ClassName, (ClassAttribute)*, (ClassMethod)*)>

<!ELEMENT ClassName (#PCDATA)>
<!ELEMENT ClassAttribute (#PCDATA)>  <!-- instance var. -->
<!ELEMENT ClassMethod (#PCDATA)>  <!-- method signature -->

<!ATTLIST ClassName Abstract CDATA #REQUIRED>
                 <!-- "Yes" if abstract, "No" otherwise -->
<!ATTLIST ClassAttribute Type CDATA #REQUIRED>
 <!-- datatype of attribute/instance var. e.g. "double" -->
] >
```

Figure 4.2

A1(a) "bookwork" [3]

- Encapsulation [1]
- Polymorphism [1]
- Inheritance [1]

A1(b) "bookwork" [2]

Some Object Oriented languages like Smalltalk demand that every new class is a subclassed from an existing class in the overall class hierarchy. This has the advantage that the class at the root of the hierarchy provides a common protocol shared by every class e.g. print state, inspect state, list sub/super-classes. The new subclasses may then override the inherited methods if required. The disadvantage is that every program includes a collection of core classes which makes the binary bigger.

The C^{++} language has been designed so as not to demand the absorption of new classes into an existing class hierarchy which leads to the advantage that the resultant binaries are smaller [1]. The disadvantage is that in not being able to inherit a common protocol requires that encapsulation is intentionally broken, in order to allow some classes access to the otherwise private/protected class members. In C^{++} this is done through friend functions [1]. Typically, friend functions are added to a class protocol in order that the I/O stream classes may display state information.

A1(c) "bookwork" [2]

Advantage [1]	Disadvantage [1]
many iterations of short sequential stages, resulting in a process of constant updating and revisiting of previous models and clarifying requirements as we proceed e.g. like doing a jigsaw or crossword.	it is harder to track and time project stages as there might be many iterations and stages feed to earlier and later stages
parallel development of the requirements subsets (reducing complexity and scope), final product integrates all the subsets.	the final integration of the subsets requires careful organisation
tests defined and developed at each stage	subsets need careful planning and are based on importance and risk factors
slicing up the requirements encourages reuse.	
early feedback and partial deliverables reduces the risk of solving the wrong problem.	

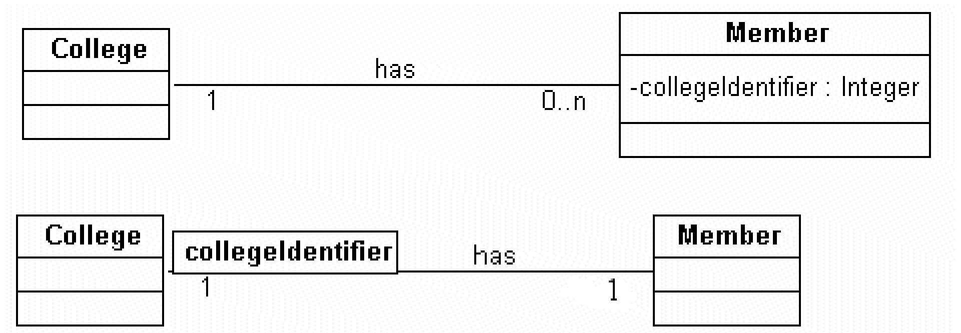
A1(d) "bookwork/new computed example" [4]

Tangible [0.5]	e.g. Exam paper, Answer script, pen, desk… [0.5]
Role [0.5]	e.g. Invigilator, Examiner, Examinee [0.5]
Event [0.5]	e.g. Exam [0.5]
Organisation [0.5]	e.g. Department, College, University [0.5]

A1(e) "bookwork" [4]

A Qualified Association is often used to select a unique object from the set of objects linked to one object e.g. to select a Member (from the set of members) given a membership code [1]. It adds precision to a design document [1]. Without qualified associations it is necessary to spell out the definition of an association in the glossary. e.g. College has Member is one-to-many, becomes College+collegeIdentifier has Member is one-to-one.

The property accountNo has been moved out of Account into the qualification, putting more information into the model and less into the glossary. It is also required during design to map from an object identity to a value identity. [2]



A1(f) "bookwork/new computed example" [2]

Coupling is a measure of the dependency between classes resulting from collaboration between objects to provide a service. It is desirable to have low coupling between classes, which means that one class is not too dependent on another, which leads to less maintenance and facilitates reuse. In the context of Y2K, a class to handle dates could have been unplugged and replaced with an updated version. [1]

Cohesion is a measure of how strongly related and focussed the responsibilities in a class are. It is desirable that a class exhibit high cohesion, which leads to less maintenance and facilitates reuse. In the context of Y2K, the existence of a Date class would have narrowed the focus compared with scrutinising vast quantities of monolithic code. [1]

A1(g) "bookwork/new computed example" [3]

(i) Dataflow e.g. pre-processor -> C++ compiler -> C compiler -> assembler -> linker [1]
(ii) User-Interface [1]
(iii) The palettes of the Visual Component Library contain components (i.e. classes) [1]

A2(a) "new computed example" [4]

If the program user enters a numeric value, the code in the try block is executed which results in the legend "Good input" appearing in the title bar of the window. [2]

If the program user enters a non-numeric value, the code in the catch block is executed which results in the legend "Bad input" appearing a message box. [2]

A2(b) "new computed example" [6]

```
class Filter {
    protected:                                              [0.5]
        double getF0(void);                                 [0.5]
        virtual double getQ(void);                          [0.5]
        virtual ComplexImpedance transfer(void)=0;          [0.5]
        double resistor;
        double capacitor;
        double inductor;
};
class BandpassFilter : public Filter{                       [0.5]
  public:                                                   [0.5]
    double getQ(void);                                      [0.5]
};
class LowpassFilter : public Filter { };                    [0.5]
```

```
Filter *f;                                                    [1]
f = new BandpassFilter(10.0, 0.001, 100.0);
f = new LowpassFilter(10.0, 0.001, 100.0)                     [1]
```

A2(c) "new computed example" [6]

- The program reuses a Standard Template Library class for the complex number ADT [1]
- Since it is a STL class its data members are template classes [1]

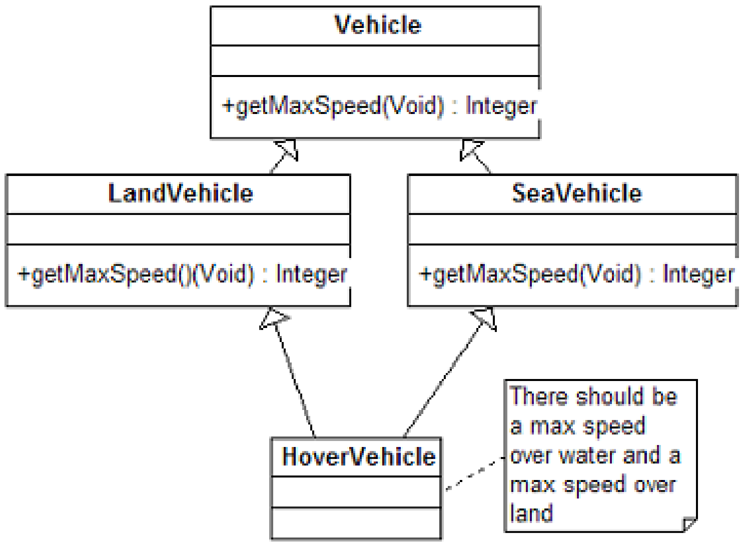
```
template <class A> class Component {                           [1]
  public:
    Component(A r, A i) : z(r,i) { }                         [0.5]
    complex<A> series(Component z1)  {etc …}                 [0.5]
    complex<A> parallel(Component z1){etc …}                 [0.5]
    complex<A> potDiv(Component2 z1) {etc …}
  private:
    complex<A> z;                                            [0.5]
};
```

```
double freq = 100/(2.0*M_PI), angFreq = 2.0*M_PI*freq;
Component <long double> zR (100.0, 0.0);                     [0.5]
Component <float> zL(0.0, 1.0*angFreq);                      [0.5]
Component <double> zC(0.0, -1.0/(0.001*angFreq));
```

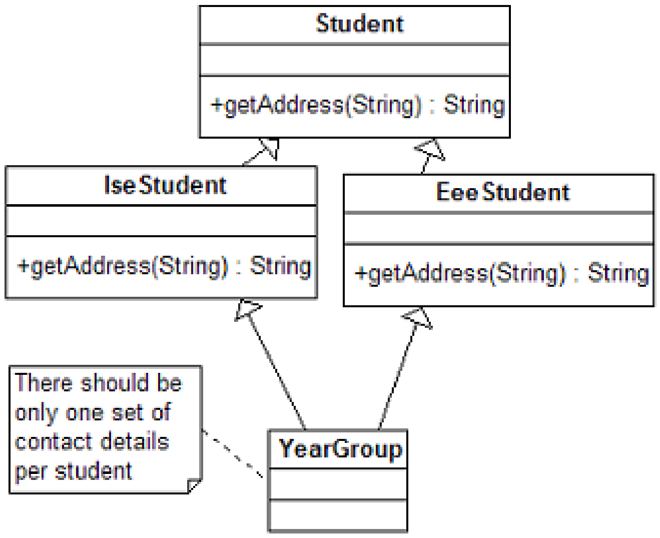
A2(d) "bookwork/new computed example" [4]

Since C^{++} allows a class to inherit from multiple classes, it is theoretically possible that two or multiple superclasses might have identical members inherited from a common grandparent. If the member has been overriden in the parent classes then the scope-resolution operator enables specification of a specific class and member e.g. LandVehicle::getMaxSpeed()

If the member is not over-ridden in the parent superclasses then to access the member of the grandparent the parent classes must be defined as virtual e.g.

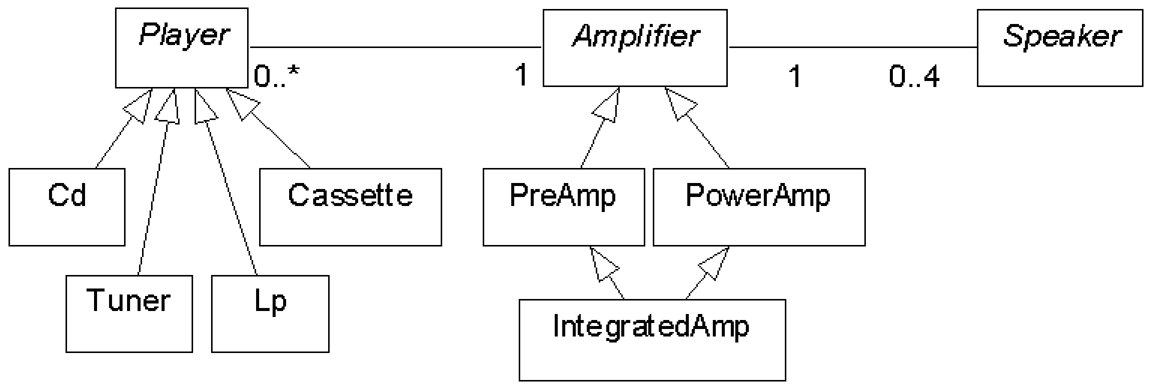

Vehicle
+getMaxSpeed(Void) : Integer
LandVehicle
+getMaxSpeed()(Void) : Integer
SeaVehicle
+getMaxSpeed(Void) : Integer
HoverVehicle
There should be a max speed over water and a max speed over land

```
class Student{ };

class IseStudent: public virtual Student{ };

class EeeStudent: public virtual Student { };

class YearGroup: public IseStudent, EeeStudent { };
```


Student
+getAddress(String) : String
IseStudent
+getAddress(String) : String
Eee Student
+getAddress(String) : String
YearGroup
There should be only one set of contact details per student

- description of problem [2]
- uml diagram [2]

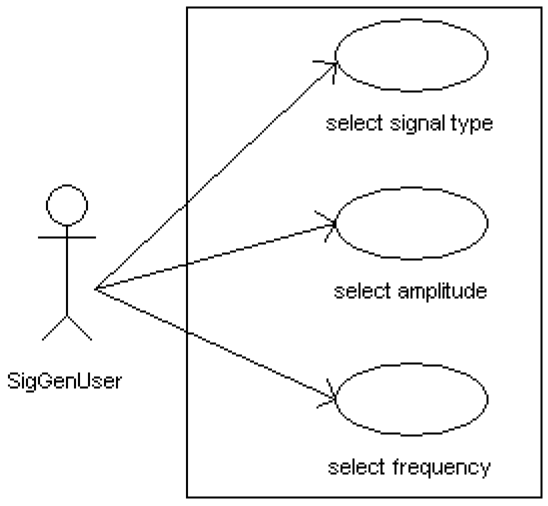
A3(a) "new computed example" [6]



Classes [1], Associations [1], Multiplicites [2], Generalisations [2]

A3(b) "new computed example" [2]

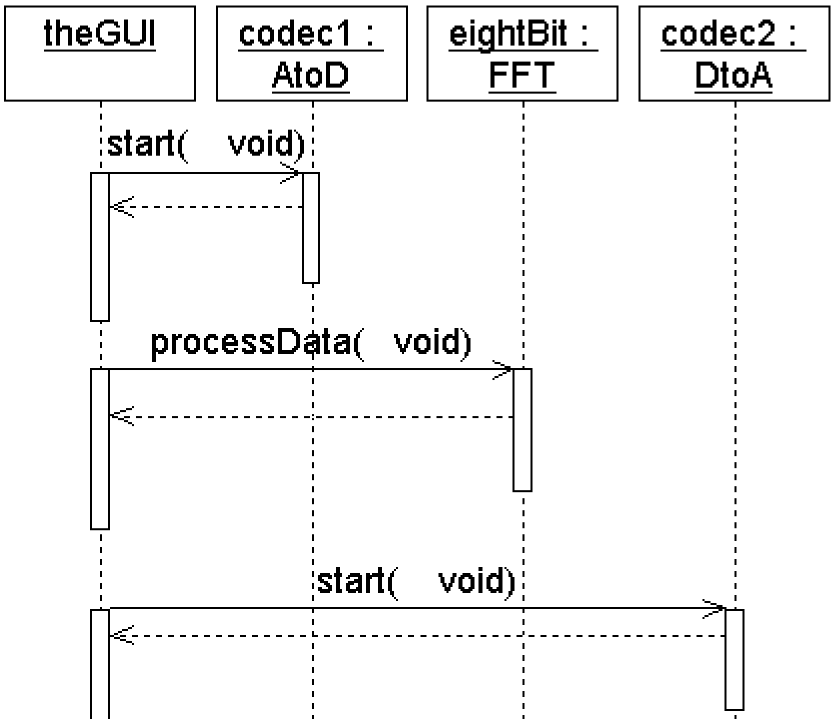
three Use Case's 3x[0.5]
system boundary required since system will be implemented in software [0.5]



A3(c) "new computed example" [3]

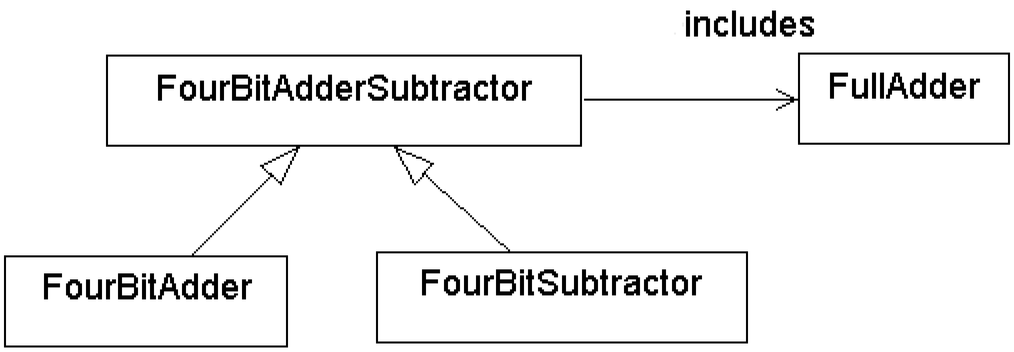
1. inclusion of the orchestrating instance which is responsible for initiating each use-case i.e. dynamic behaviour [1]
2. navigation of associations can only be determined after consideration of dynamic behaviour [1]
3. methods can only be determined after consideration of dynamic behaviour [1]

A3(d) "new computed example" [5]



Objects [2], Messages[2], Message answers[1]

A3(e) "new computed example" [4]



Classes [1], Generalisation [1], Association [2]

A4(a) "bookwork" [4]

A design pattern is a simple and elegant design that captures a general solution to a general problem [1] (i.e. not domain specific) that has been developed over time and is widely applied and accepted as current "best practice". [1]

name: Decorator (or Wrapper) [1]
problem: when additional properties and/or behaviour are to be added dynamically on an instance by instance basis. For example, subclassing from Text for all permutations of Bold, Italic, Underline etc… would be clumsy. The decorator pattern should be transparent and nestable so that the decorator can be decorated. [1]

OR

name: Observer [1]
problem: in a small system, the object with the changed state can send messages to its dependents to inform of the state change. However, it is not always logical for an object to *know* details about all its dependents. The Observer pattern is used when there is a one to many dependency between objects, such that if there is a state change for a particular object, a number of other objects need be notified. For example, if a file is created in a MS-Console window it is reasonable to expect an MS-Explorer display of the same folder to instantly include the new file. These two applications should be de-coupled so MS-Explorer should be registered as an dependent of the directory details (Model). [1]

OR

name: Adaptor [1]
problem: enables classes to collaborate with one another even though they have incompatible interfaces. Converts an interface into another interface as expected by a client. For example, a WWW browser could have an adapter for specific languages e.g. Cantonese. [1]

A4(b) "bookwork" [4]

a) the **Client-Server model** is the most prevalent architecture and uses **message passing** e.g. an email/web/name server. Two programs communicate via a protocol, which is driven by the messages passed between the two programs. Usually a program sending a message receives a response and then acts on the response accordingly.

b) Other architectures are based upon the **Object Model** which is growing in popularity due to the growth in popularity of OO in general. The emerging architectures include:

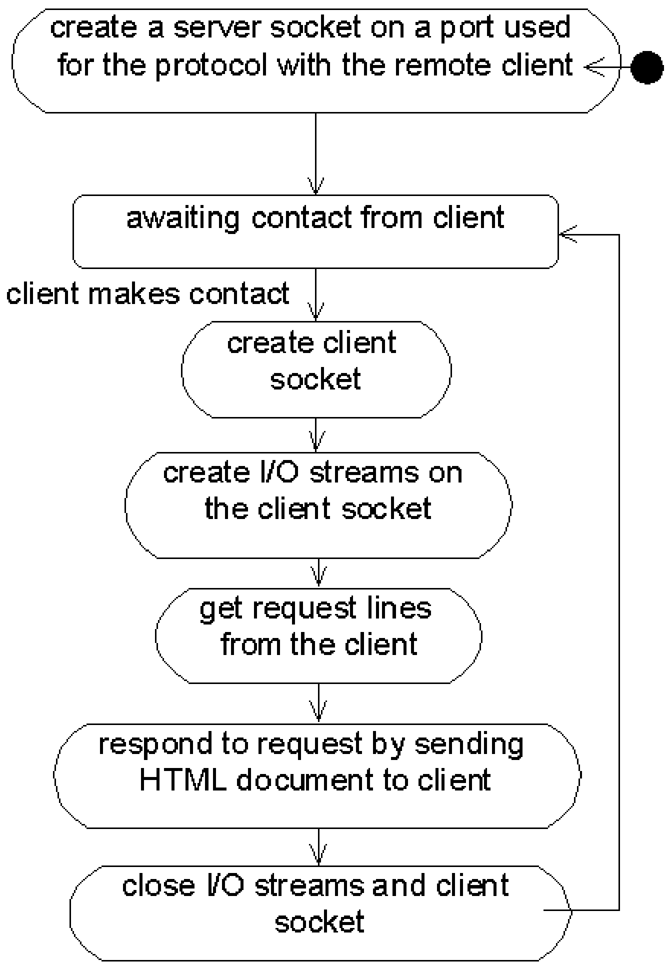
1. **Distributed object** which is based on communication between objects via message (i.e. calling methods), where the underlying communication and transport mechanisms, and the object locating process, are hidden from the application. This requires a distributed object framework e.g. CORBA, RMI, DCOM
2. **Event-based** uses code to monitor events e.g. a user-interface monitor events such as a button click. An event that causes a change of state triggers activity in the monitoring software. In a Server-Push application bus architecture (e.g. a stocks and shares tracker), listener objects would be registered with the bus, and monitor the bus for events in the form of objects that trigger processing in their domain. An early user of this technology was www.Pointcast.com that pushes out breaking news to its subscribers when it happens
3. **Tuple-spaces** are persistent object stores known as spaces. Clients access the spaces via a strict API, allowing concurrent access to the stored objects, which they copy, update and then return to the store.

All of the above is just to make a "teaching point" so this level of detail is not expected for the answer. Expect mention of:

- relationship between abstraction and performance
- underlying details e.g. sockets, ports etc…
- client-server comparison with object model
- example of high/low abstraction [4]

e.g. The client-server architecture reveals the underlying network details e.g. sockets, serversockets, ports, protocol etc… The object model based architectures (i.e. distributed object, event-based and tuple-spaces) incrementally reveal less and less of the underlying network. The price to pay for increased abstraction is performance overhead, so that tuple spaces is easy to use due to its high level of abstraction but at the cost of the slowest performance. Abstraction and performance are inversely proportional.

A4(c) "new computed example" [4]


create a server socket on a port used for the protocol with the remote client
awaiting contact from client
client makes contact
create client socket
create I/O streams on the client socket
get request lines from the client
respond to request by sending HTML document to client
close I/O streams and client socket

A4(d) "bookwork/new computed example" [8]

Advantages: [3]

- UML CASE tools can exchange UML data
- Compilers can read UML data directly
- Compilers can generate UML documents from source code
- XML is language neutral and can generate code in any suitable language
- Design Patterns can be loaded into initial code editors
- etc…

```
<!-- XML data for the Filter CLASS -->
<CLASSDEFINITION>                                          [1]

<CLASS>
  <ClassName Abstract="YES"> Filter </ClassName>           [1]
  <ClassAttribute Type="double">
                           Resistor </ClassAttribute> [0.5]
  <ClassAttribute Type="double">
                          Capacitor </ClassAttribute> [0.5]
  <ClassAttribute Type="double">
                           Inductor </ClassAttribute> [0.5]

  <ClassMethod>
       getF0(void):double </ClassMethod>                 [0.5]
  <ClassMethod>
       getQ(void):double </ClassMethod>                  [0.5]
  <ClassMethod>
       transfer(void):ComplexImpedance </ClassMethod> [0.5]
</CLASS>

</CLASSDEFINITION>
```