IMPERIAL COLLEGE LONDON

DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING
EXAMINATIONS 2011

ISE PART II: MEng, BEng and ACGI

SOFTWARE ENGINEERING 2

Tuesday, 7 June 2:00 pm

Time allowed: 2:00 hours

There are THREE questions on this paper.

Answer ALL questions.
Q1 carries 40% of the marks. Questions 2 and 3 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
Second Marker(s) : J.V. Pitt

The Questions

1.

All the parts of this question refer to Figure 1.1. The figure shows an architecture that is similar but is not identical to the analogue filter case study developed in EE2-12. Your answers should be based only on what is shown in the figure.

a) For **all** the question parts below **you must identify elements from the architecture to illustrate your answer**. Use fully qualified names where appropriate.

i) List the *four* core properties of Object Oriented Programming.

[8]

ii) Briefly describe *three* different types of implementation code inheritance shown in the analogue filter architecture.

[6]

iii) Briefly describe *two* ways that code in a derived class may *explicitly* invoke code in a C^{++} base class i.e. exclude the code inheritance mechanisms described in part ii.

[4]

iv) What is the meaning of the term *Programming by differences*?

[2]

v) Briefly explain why *Programming by differences* is beneficial if this architecture was extended.

[2]

vi) What is a *Polymorphic* member function?

[2]

vii) Briefly explain why invoking a *Polymorphic message* on a *Polymorphic variable* is beneficial if this architecture was extended.

[4]

b) List the element(s) of the C^{++} *Generic programming* paradigm that are *not* shown in the architecture.

[2]

c) Briefly explain why the object oriented approach to handling errors shown in the architecture is particularly well suited to working with library code.

[2]

d) For any *one* of the template classes in the architecture briefly justify the benefit of the class having a *Template datatype* data member.

[2]

e) List an example of *Multiple inheritance* shown in the architecture. Draw a UML class diagram for each of the *two* generic problems that can result from the use of *Muultiple inheritance*.

[6]

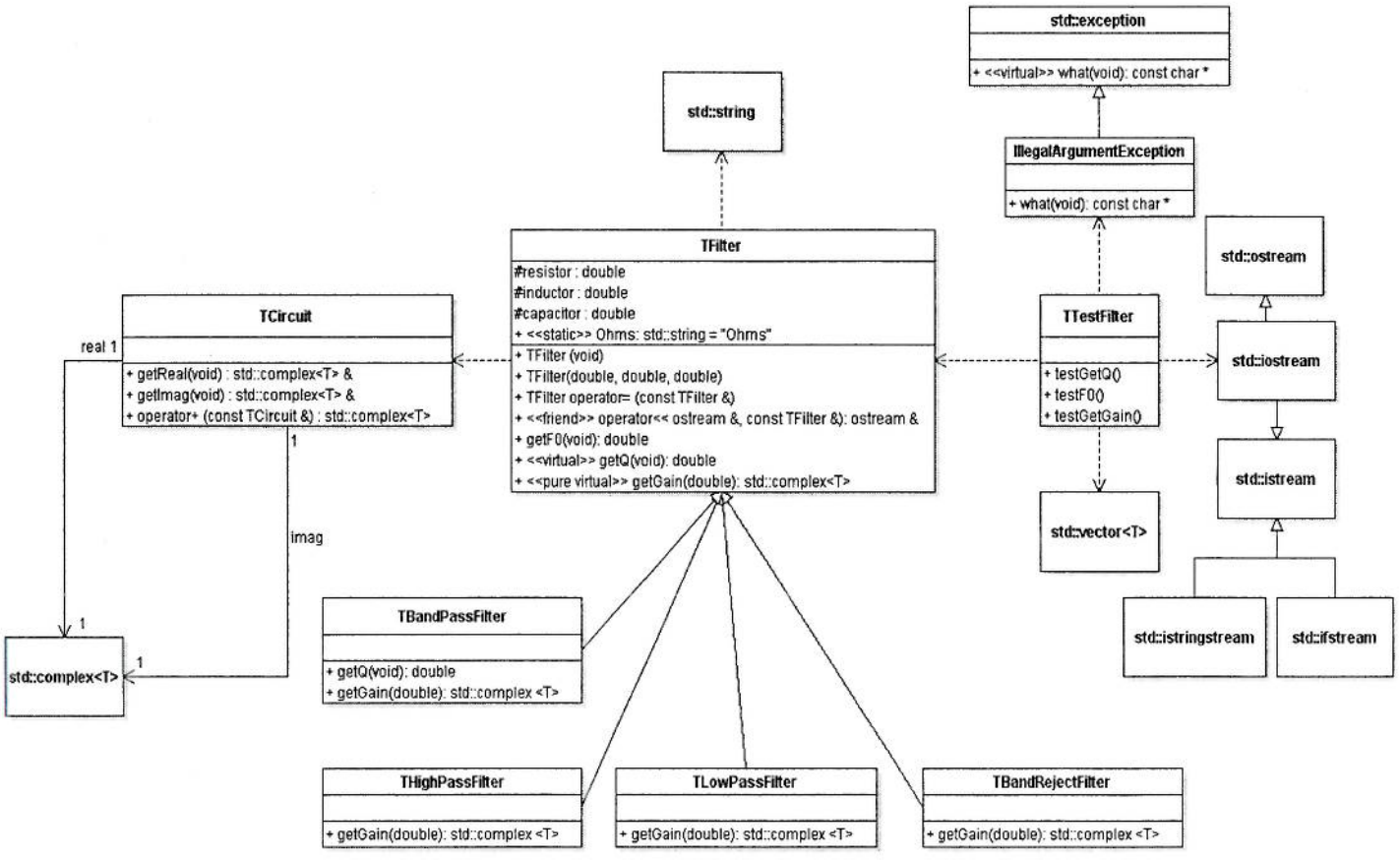


Figure 1.1 Architecture based on the EE2-12 Computing laboratories

2.

All the parts of this question refer to Figure 2.1. The figure shows a problem domain description that is similar but is not identical to the robot world case study developed in EE2-12. Your answers should be based only on what is shown in the figure.

a) Write C++ *Specification* code that would be found in the *Specification files* for the *four* classes in the complete robot world description. Include the relationships between the classes and the data members but *not* the member functions (*Implicit* or *Explicit*).

[8]

b) Choose a class that is *not* a derived class. For your chosen class write C++ *Implementation* code that would be found in an *Implementation file* including *Implicit* and *Explicit* member functions. Create a stub for any member function that does not yet have any useful functionality.

[8]

c) Using the class you selected for your answer to (b) write C++ *Implementation* code:

i) For the *Stream Insertion* operator;

ii) For the *Stream Extraction* operator.

[6]

d) Write a short C++ code extract to demonstrate the invocation of a *Polymorphic message* on a *Polymorphic variable.*

[3]

e) Write a C++ *Exception class* to deal with a request to add a robot to the robot carrier when the robot carrier is already full.

[3]

f) For each of the following concepts briefly describe where the concept might be used in the robot world C++ code:

i) A *class* constant;

ii) A *constant* function.

[2]

A robot moves around a two-dimensional grid. The position of a robot on the grid is specified using `X` and `Y` coordinates. A robot records its current position on the grid and is located at (0, 0) when instantiated. The robot protocol includes the following:

- set the `X` position;
- set the `Y` position;
- get the `X` position;
- get the `Y` position.

A robot may be free standing on the grid or it may be loaded onto a robot carrier for transportation to a grid location in order to perform some task. The robot carrier may carry up to four robots at a time. Like the robot, the robot carrier records its current position on the grid. Robots loaded onto the carrier have the same location as the robot carrier.

The robot carrier shares the same protocol as the robot but it can also include some additional messages:

- indicate how many places are available for carrying robots i.e. 0 to 4;
- add a robot to the carrier;
- remove a robot from the carrier;

Every robot has a fixed amount of data storage. The quantity of storage determines the type of the robot because a team leader robot has a large amount of storage and a drone robot has a small amount of storage. There must be exactly one team leader and between one and three drone robots on the robot carrier before it moves to a new location.

A drone robot can also respond to a number of additional messages:

- harvest the data (associated with the task to be performed by the drone);
- transfer the data to the team leader robot.

A team leader robot can also respond to an additional message:

- receive and store all the data harvested by the drone robots.

Figure 2.1 The robot world problem domain description

3.

All the parts of this question refer to Figure 3.1. The figure shows a problem domain description that is similar but is not identical to the nodal analysis case study developed in EE2-12. Your answers should be based only on what is shown in the figure. The figure shows a basic resistive circuit, the nodal analysis and a solution coded in Matlab™.

For all the UML class diagrams include the relationships between classes (where appropriate), the data members and *only* the member functions required for the algebra. You do *not* need to include constructors, destructors, getters, setters, operators, helpers and the functions that map between one and two dimensional data structures.

a) Draw a UML class diagram for the class `TSequence` which implements a dynamic linear array containing *Template type* elements. The storage for the array is allocated when the `TSequence` is instantiated using the row and column integer dimensions.

[6]

b) Draw a UML interaction diagram which replicates the Matlab™ code but uses the protocol specified in your answer to (a). Use the UML stereotype `<<create>>` to show the creation of an object. You can assume that the function specified below is also available:

```
void put(int row, int col, double value);
// inserts value at specified row and column
```

You should use the same identifiers and numeric values shown in the Matlab™ code for the objects. Use `double` for all the numeric data.

[6]

c) Draw a UML class diagram where the class `TSequence` has been replaced by two new classes (i.e. `TMatrix` and `TVector`). The *two* classes are involved in an *Association* relationship.
Note:

- The class `TMatrix` represents a 2 by 2 matrix.
- The class `TVector` represents a 2 by 1 vector.

[6]

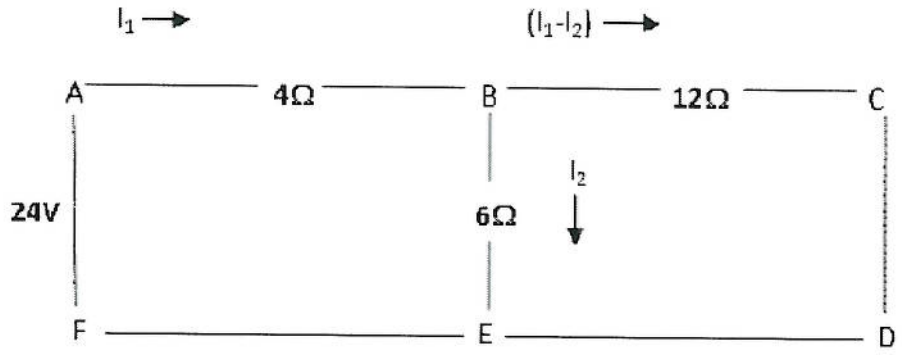
d) Draw a UML class diagram where the class `TSequence` has been refactored and augmented with two new classes (i.e. `TMatrix` and `TVector`). The *three* classes are involved in an *Inheritance* relationship.
Note:

- The class `TMatrix` represents a 2 by 2 matrix.
- The class `TVector` represents a 2 by 1 vector.

[6]

e) Draw a UML class diagram for the class `TSequence` which implements a dynamic linear array using a C^{++} standard library class containing *Template type* elements.

[6]



Using Kirchoff's Laws for the loop ACDF: $16* I_1 - 12* I_2 = 24$ (Eqn. 4.1)

and for the loop ABEF: $4* I_1 + 6* I_2 = 24$ (Eqn. 4.2)

Solving (Eqn. 4.1) and (Eqn. 4.2) gives $I_1 = 3$ and $I_2 = 2$

The simulataneous equations may be solved in Matlab™ as follows

```
>> resistorsMatrix = [16 -12; 4 6];

   voltagesVector = [24; 24];

   resistorsInverted = resistorsMatrix ^-1;

   currentsVector = resistorsInverted * voltagesVector
```

Note: ^-1 represents 2x2 matrix inversion.

Figure 3.1 Nodal analysis of a simple resistive circuit

Solutions 2011

EE2-12

Q1. (a i) [8]

- Abstraction: Each class describes what we need to know e.g. `TFilter`
- Encapsulation: The scope of the class and its visibility modifiers controls (e.g. + - #) encapsulate what we are able to know.
- Inheritance: Many examples e.g. `TFilter` is a base class to `TBandPassFilter`
- Polymorphism: Many examples e.g. polymorphic functions include
  `TFilter::getF0(),TFilter::getQ(),TFilter::getGain().`

(ii) [6]

- Drop-through code that is used unchanged e.g. `TFilter::getF0(),`
- Drop-through code that *may* be overriden e.g. `TFilter::getQ(),`
- Drop-through code that *must* be overriden e.g. `TFilter::getGain().`

(iii) [4]

A derived class constructor can call a base class constructor using C++ constructor initialiser syntax e.g.

```
TBandPassFilter(double R,double L,double C):TFilter(R,L,C){}
```

A base class member function can be called by using the FQN e.g.

```
TFilter::1/getQ()
```

(iv) [2]

Programming by differences is where we focus on the differences between a base and derived class. The code may be new and/or it may extend and/or modify existing code e.g. the only difference between the class `TLowPassFilter` and `TFilter` is that the former overrides the inherited member function `getGain()`.

(v) [2]

Starting from a generic specification of a concept we focus on the differences that are required to deliver the new specialisation of the concept. This should require less development effort than if we started without any existing code and a full specification of the new concept. For example, the 3 filter member functions would require 12 code implementations without inheritance but here there are 8 code implementations i.e. 3 + 2 + 1 + 1 +1. This mechanism facilitates code reuse.

(vi) [2]

A polymorphic member function is a function with the same siqnature as an inherited member function. The inherited code may be overridden therefore the response can be receiver dependent e.g.

`TBandPassFilter::getGain(), TLowPassFilter::getGain()` etc...

(vii) [4]

Invoking a polymorphic message on a polymorphic variable facilitates adaptability and offers future-proofing of code. The polymorphic variable may be assigned to an instance of a newly invented or debugged derived class. If all the instances of the derived classes are logically related and only use the shared inherited protocol then it can mean minimal changes to the overall program. For example, a program might allow the user to select one of the filters, then instantiate an object for the filter, assign the object to a polymorphic variable and invoke one of three member functions on the object. The user may then select another object and reassign the variable to the new object etc... The steps followed may be the same for 4 or 40 filter types, so in the future new filters may be created with minimal changes to the overall program.

b) [2]

Generic algorithms and iterators are *not* in the architecture. Not required but examples of container classes can be found in the architecture e.g. `std::vector<T>`

c) [2]

The exception handler is useful when the source of the source of the problem is decoupled from the source of the solution. C++ programming relies heavily on the use of utility classes which may throw an exception object with state containing information about the source of the problem. The caller of the utility then provides a catch block which provides the solution to the problem.

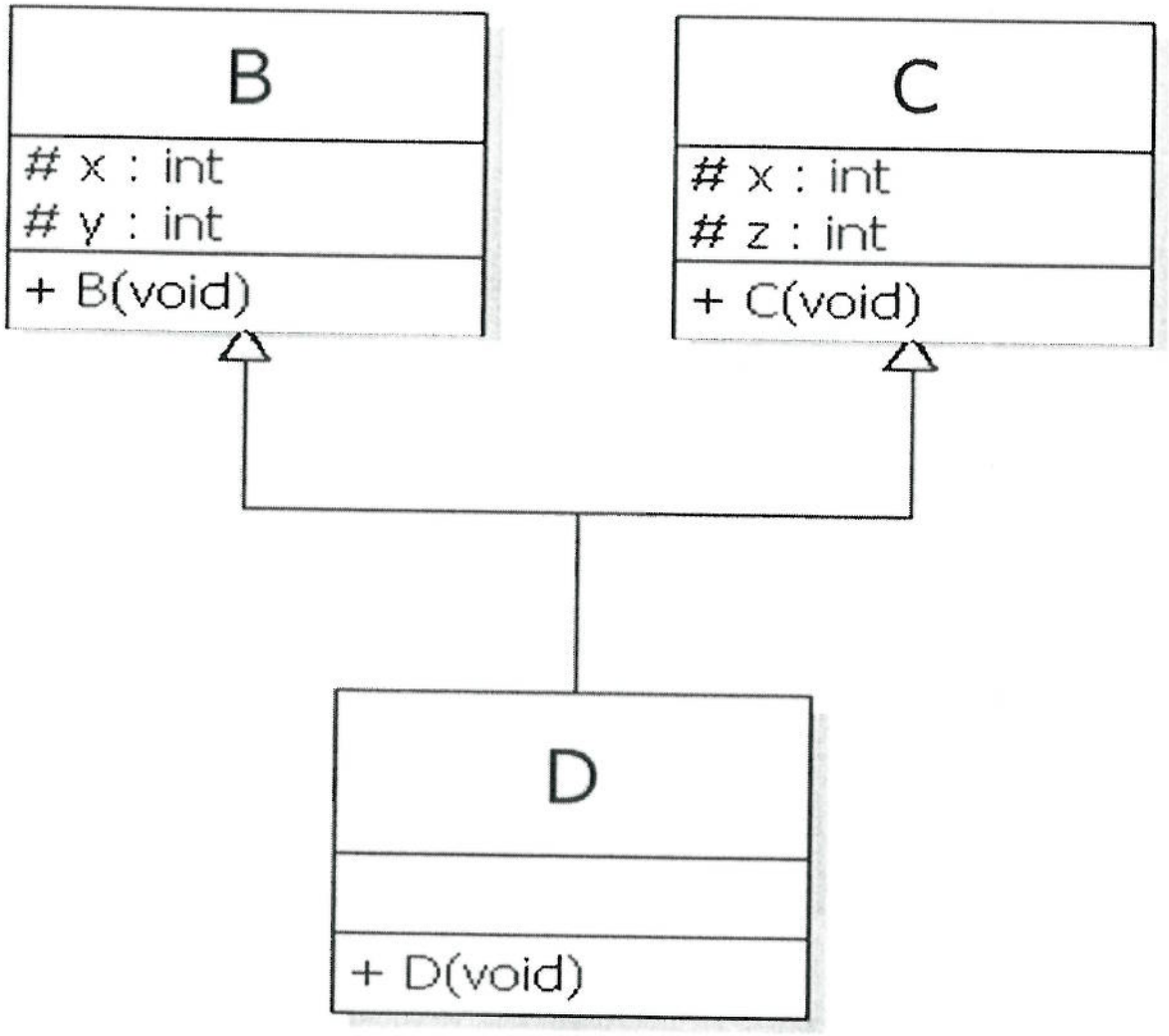
d) [2]

The container class `std::vector<T>` provides a linear sequence data structure (i.e. a dynamic array) whose elements are a template type. Therefore gains is a useful utility since the elements may be of any datatype.

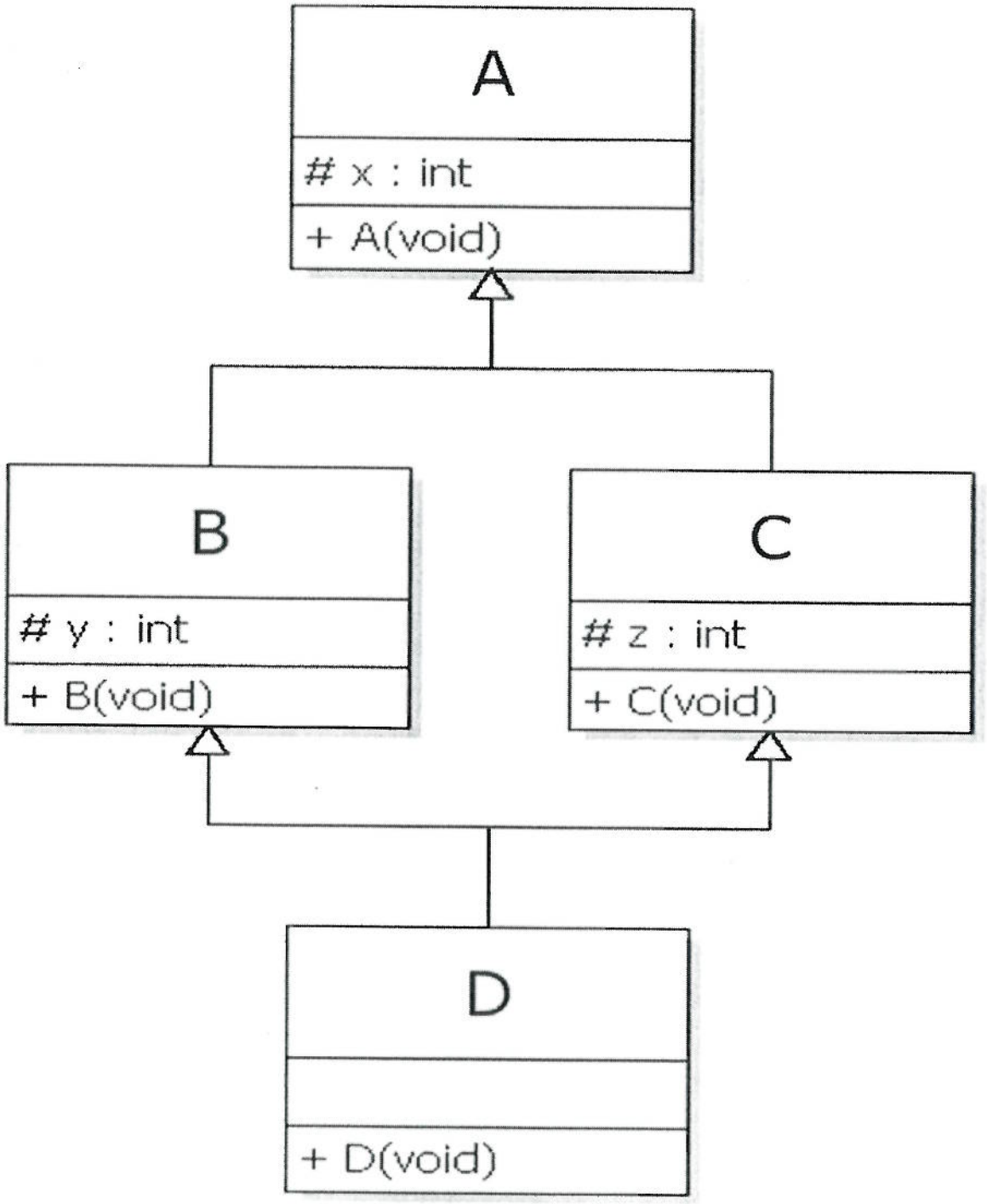
e) [6]

`std::iostream` inherits from both `std::istream` and `std::ostream`

(i) Parents `B` and `C` provide the same class member eg: `B::x` and `C::x`



(ii) Parents `B` and `C` provide the same class member where they have both inherited the member from the common grandparent e.g. `A::x` , `B::x` and `C::x`



Q2.

(a) [8]

```
class TRobot {
  private:
    int x, y, storage;
};
```

```
class TDroneRobot: public TRobot { };

class TLeaderRobot: public TRobot { };

class TRobotCarrier {
 private:
  int x, y;
  TRobot payload[4];
  // assumes that the number of spaces free is a derived attribute
};
```

(b) [8] `TRobot` or `TRobotCarrier` may be used.

```
TRobot::TRobot() : x(0), y(0) { }

TRobot::TRobot(const TRobot& orig) : x(orig.x), y(orig.y) { }

TRobot::~TRobot() { }

TRobot TRobot::operator= (const TRobot  &tc) {
  if (this == &tc)
    return tc;
  else {
    this->x = tc.x; this->y = tc.y; return *this;
  }
}

void TRobot::SetY(int y) { this->y = y; }
int TRobot::GetY() const { return y; }
void TRobot::SetX(int x) { this->x = x; }
int TRobot::GetX() const { return x; }
```

(c) [6]

`TRobot` or `TRobotCarrier` may be used.

(i)

```
ostream& operator << (ostream &os, const TRobot &obj)
{
      os << "X = " << obj.GetX() << " ,Y -= " << obj.GetX(;
      return os;
}
```

(ii)

```
istream& operator >> (istream &is, TRobot &obj)
{
      int x, y;
      is >> x >> y; obj.SetX(x); obj.SetY(y);
      return is;
}
```

(d) [3]

```
TDroneRobot *drone1 = new TDroneRobot;
TLeaderRobot *leader1 = new TLeaderRobot;
TRobot *aRobot;

aRobot = drone1;
cout << aRobot->GetX();
aRobot = leader1;
cout << aRobot->GetX();
```

(e) [3]

```
class RobotCarrierFullException: public runtime_error
{
  public:
    RobotCarrierFullException(void);
};


RobotCarrierFullException::RobotCarrierFullException(void):
                                runtime_error("Robot carrier full"){};
```

(f) [2]

(i) The maximum number of robots that can be carried by the robot carrier.

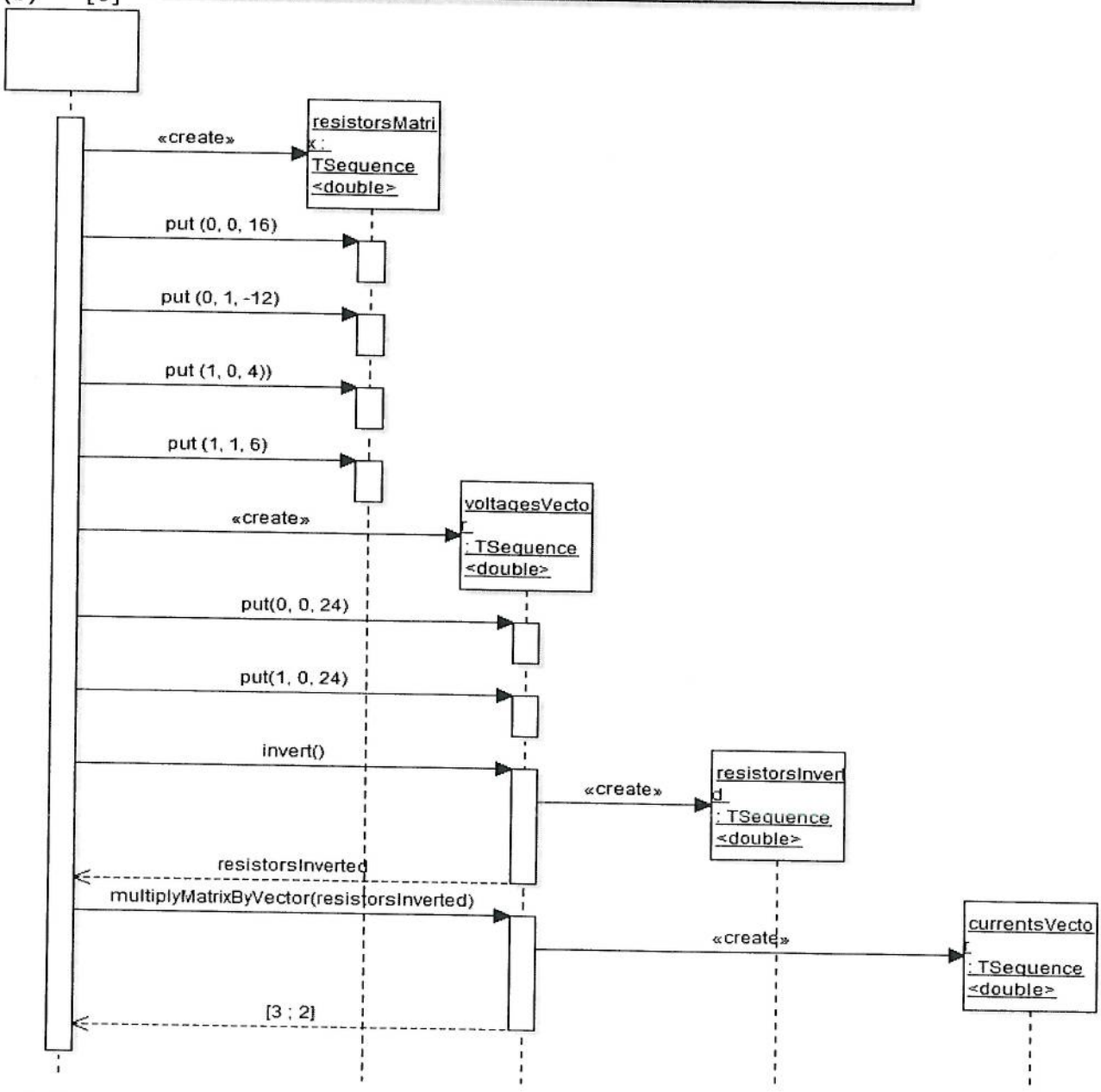
(ii) The getter functions for any of the classes.

Q3. Note: constructors, destructors, getters, setters, operators, helpers and the functions that map between one and two dimensional data structures need not be shown.

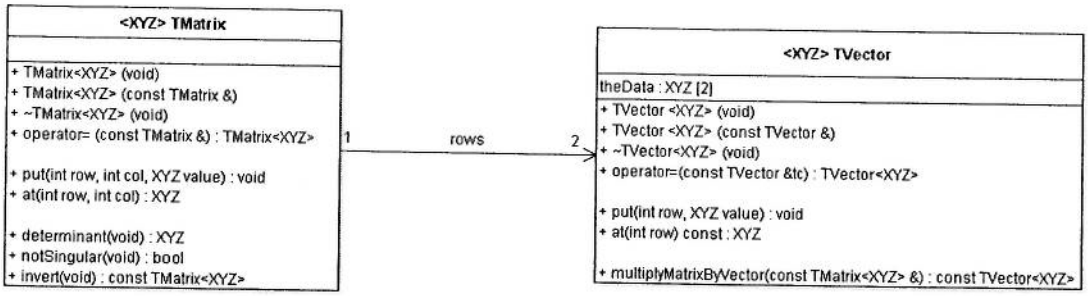
(a) [6]

TSequence <XYZ>
noRows : int # noCols : int # theData[ ] : XYZ
+ TSequence <XYZ> (int rows, int cols) + put (int row, int col, XYZ value) : void + at (int row, int col) : XYZ + determinant(void) : XYZ + notSingular(void) : bool + multiplyMatrixByVector(const TSequence <XYZ> & matrix2by2) + invert(void) : TSequence <XYZ>

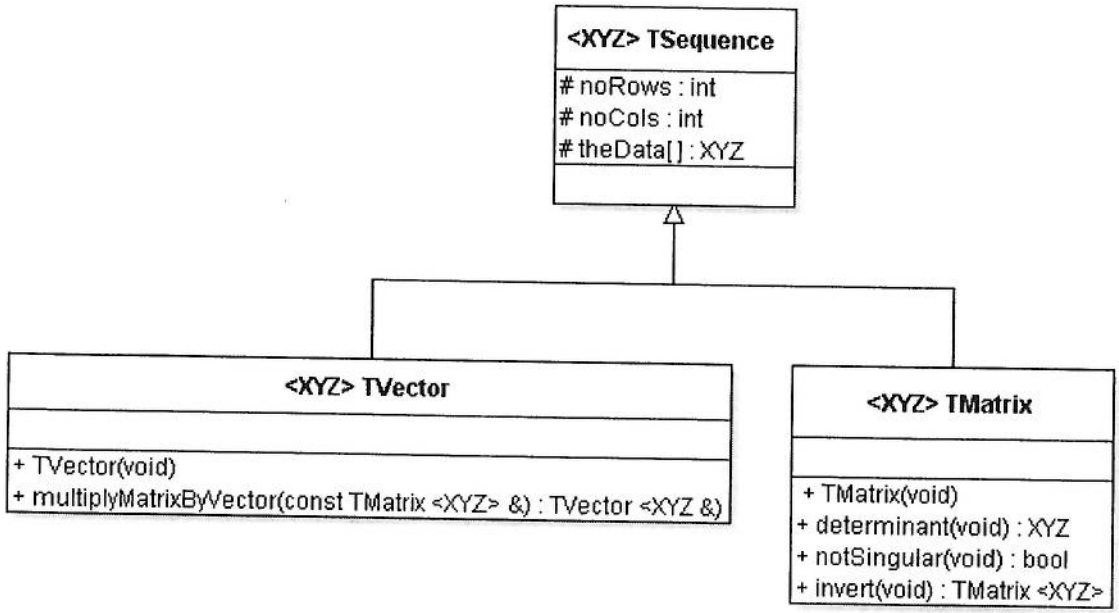
(b) [6]



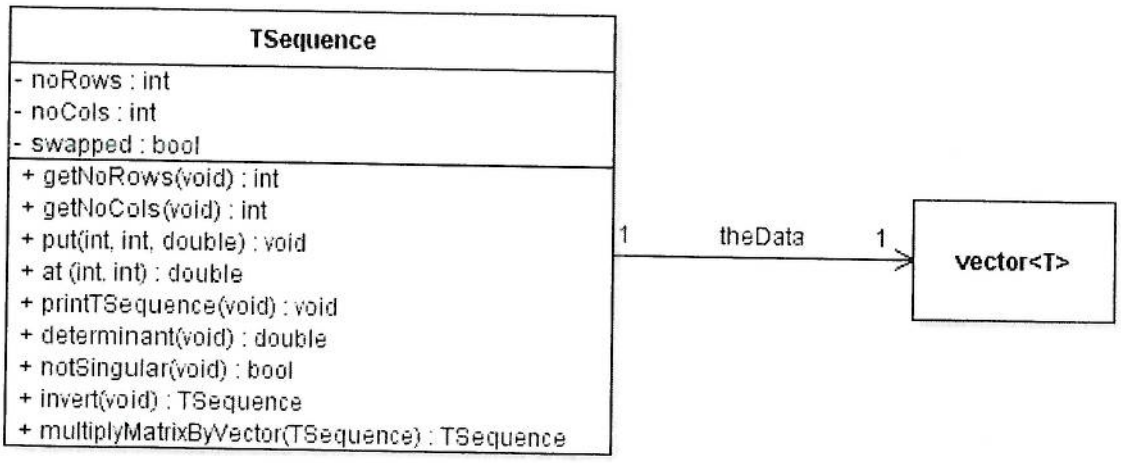
(c) [6]


<XYZ> TMatrix
+ TMatrix<XYZ> (void)
+ TMatrix<XYZ> (const TMatrix &)
+ ~TMatrix<XYZ> (void)
+ operator= (const TMatrix &) : TMatrix<XYZ>
+ put(int row, int col, XYZ value) : void
+ at(int row, int col) : XYZ
+ determinant(void) : XYZ
+ notSingular(void) : bool
+ invert(void) : const TMatrix<XYZ>
1
rows
2
<XYZ> TVector
theData : XYZ [2]
+ TVector <XYZ> (void)
+ TVector <XYZ> (const TVector &)
+ ~TVector<XYZ> (void)
+ operator=(const TVector &tc) : TVector<XYZ>
+ put(int row, XYZ value) : void
+ at(int row) const : XYZ
+ multiplyMatrixByVector(const TMatrix<XYZ> &) : const TVector<XYZ>

(d) [6]


<XYZ> TSequence
noRows : int
noCols : int
theData[] : XYZ
<XYZ> TVector
+ TVector(void)
+ multiplyMatrixByVector(const TMatrix <XYZ> &) : TVector <XYZ &)
<XYZ> TMatrix
+ TMatrix(void)
+ determinant(void) : XYZ
+ notSingular(void) : bool
+ invert(void) : TMatrix <XYZ>

(e) [6]


TSequence
- noRows : int
- noCols : int
- swapped : bool
+ getNoRows(void) : int
+ getNoCols(void) : int
+ put(int, int, double) : void
+ at (int, int) : double
+ printTSequence(void) : void
+ determinant(void) : double
+ notSingular(void) : bool
+ invert(void) : TSequence
+ multiplyMatrixByVector(TSequence) : TSequence
1
theData
1
vector<T>