Master copy. June 08 E2.12

SOLUTIONS 2008

Q1

['bookwork'] [2]

(a i) An *object* data member. - `theObject` [1]

(a ii) The name(s) of member function(s) that break encapsulation.
`TOuter::shallowCopy(),TOuter::midCopy()` [1]

['bookwork'] [6]

(b i) Default and copy constructors, assignment operator and destructor. [1]

(b ii) Default constructor ensures that the every instance of class has a defined default state.

A copy constructor is used the copy the state of an existing instance of a class to a newly created instance of the same class.

A copy constructor is used the clone the state of an existing instance of a class to another instance of the same class.

A destructor performs housekeeping functions (e.g. returning memory to the heap) before an instance of a class goes out of scope. [1]

(b iii) Default constructor ensures that the every instance of class has a defined default state thus avoiding the threat of using uninitialized data. By explicitly assigning values to every data member we improve the documentation quality of the code and avoid the need to memorise language specific default initialisations.

A copy constructor is used the copy the state of an existing instance of the same class to a newly created instance of the same class.

A copy constructor is used the clone the state of an existing instance of the same class to another instance of the same class.

The destructor, although, typically empty is used as an aide-memoire to encourage thinking about consequences of change. For example, after a change to the underlying data e.g. from stack to heap based. [1]

(b iv) The assignment operator and the copy constructor copy the state of an existing object to another object.

The issue of concern is that an object that has an object data member and uses a shallow copy can lead to the breaking of encapsulation when the object data member is defined as a pointer or reference variable i.e. not an ordinary object. A recursive deep copy can solve the problem. [1]

(b v) The assignment operator and the copy constructor copy can be distinguished by the fact that former can be invoked at anytime in the lifecycle of an object and the later is invoked at the moment of creation of an object. [1]

(b vi) Why might we include OCCF in an abstract class?

If the abstract class were a base class, then the OCCF members could be reused in classes derived from the base class. [1]

['new Application of theory'] [8]

(c i)
```
ostream& Tinner::operator<< (ostream &o,
                                    const TInner &v){
   o << v.value;
   return o;
}
```
[2]

(c ii)
```
istream& Tinner::operator>> (istream &i, TInner &v){
   int anInt;
   i >> anInt;
   v.setValue(anInt);
   return i;
}
```
[2]

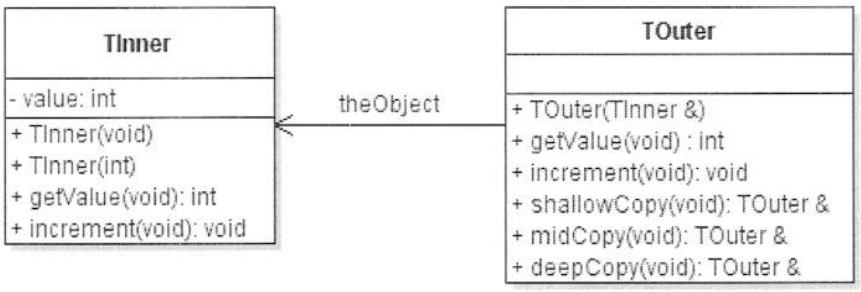
(c iii)
```
TInner & Tinner::operator= (const TInner &v) {
   this->value = v.value;
   return *this;
}
```
[2]

Changes:
```
class TInner{
public:
  etc…
   friend ostream& operator<< (ostream &, const TInner &); // Needed for (i)
   void setValue(int aValue);                              // Needed for (ii)
   TInner & operator= (const TInner &);                    // Needed for (iii)
  etc…
};
```

(c iv) The declaration of a C++ *friend* function within a class specification gives an ordinary function the special privilege of access to the private data members of the class. It is a privilege that the class author can grant in order to knowingly break encapsulation i.e. not use public member functions to access private data members. [2]

['new Application of theory'] [4]

(d i)

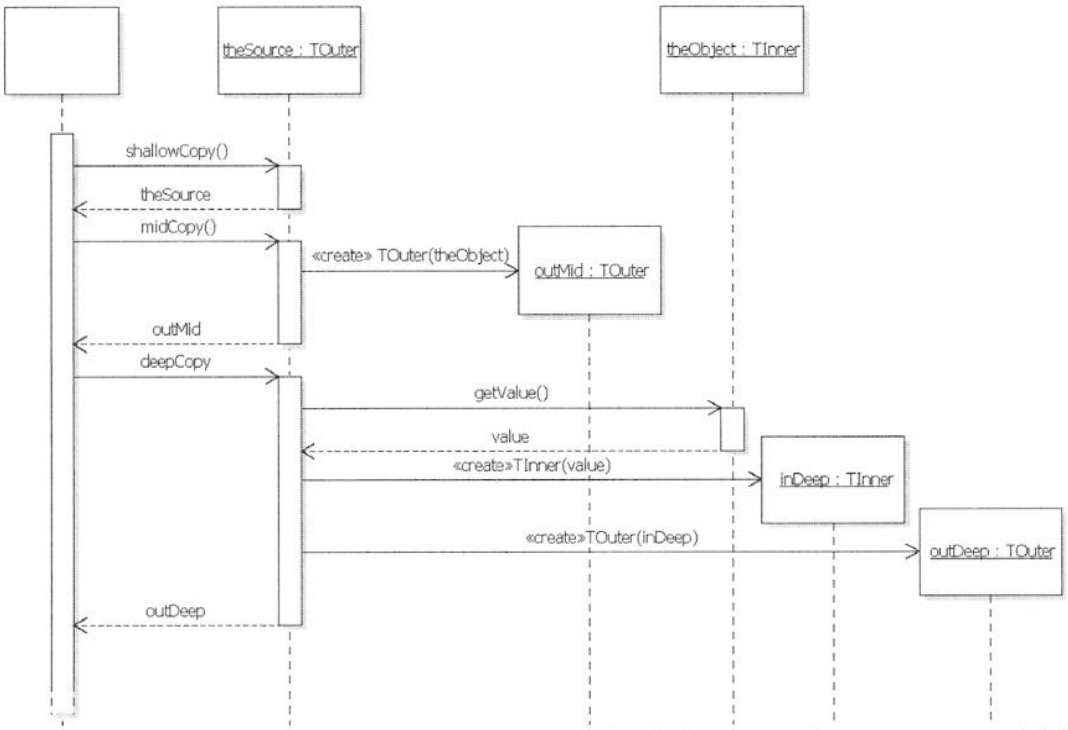


[2]

(d ii)

```
theSource.shallowCopy();
theSource.midCopy();
theSource.deepCopy();
```

[2]

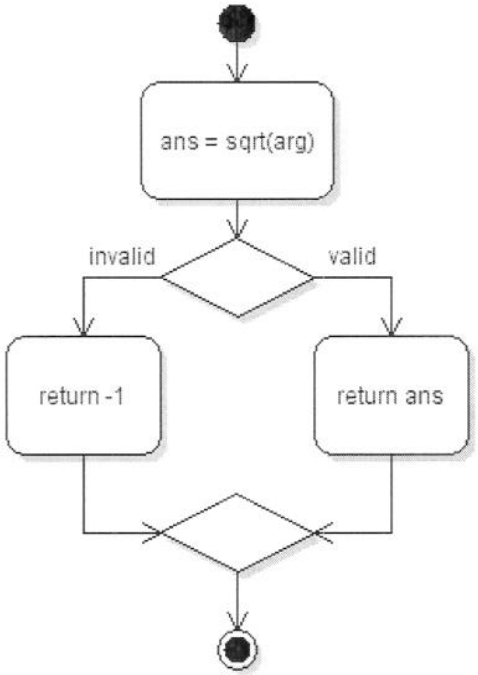


Note: only the `deepCopy()` message send (and the nested message sends) is required in the diagram above.

Q2

['new Application of theory'] [6]

(a i)



[2]

(a ii)

```
class IllegalArgumentException: public runtime_error {
  public:
    IllegalArgumentException(double
     illegalArgumentValue): runtime_error("argument error"){
       this->illegalArgumentvalue=illegalArgumentValue; //or initialiser
    }

    double getIllegalArgumentValue(void){
       return illegalArgumentValue;
    }

  private:
    double illegalArgumentValue;

};
```

[4]

['new Application of theory'] [8]

(b i) – (b iv)

The called service code	The calling client code

double newSqrt(double arg){ // Fig2.1
  errno = 0;
  double ans = sqrt(arg);
  if (errno == 0)
  return ans;
  else
  return -1.0;
}
	double arg, ans; // Fig 2.2
	cout << "Enter a value: ";
	cin >> arg;
	
	ans = newSqrt(arg);
	if (ans != -1.0)
	  printf("newSqrt: %lf\n", ans);
	else
	  perror("newSqrt: argument error");

double newSqrt(double arg){ //(a iii)
  if (arg >= 0.0)
  return sqrt(arg);
  else
  return -1.0;
}
	ans = newSqrt(arg);
	if (ans != -1.0)
	  cout << "newSqrt: " << ans;
	else
	  cout << "newSqrt: argument error"

double newSqrt(double arg){ //(a iv)
  assert (arg >= 0.0);
  return sqrt(arg);
}
	cout << "newSqrt: "<< newSqrt(arg);

double newSqrt(double arg){ //(a v)
  if (arg >= 0.0)
  return sqrt(arg);
  else
  throw IllegalArgumentException(arg);
}
	try {
	  ans = newSqrt(arg);
	  cout << "newSqrt: " << ans;
	} catch (IllegalArgumentException e){
	  cout << "newSqrt: " << e.what()
	  << " " << e.getArgument();
	}

double newSqrt(double arg){ //(a vi)
  return sqrt(arg);
}
	if (arg >= 0.0){
	  ans = newSqrt(arg);
	  cout << "newSqrt: " << ans;
	}
	else
	  cout << "newSqrt: argument error";

4 x 1.5	4 x 0.5

['bookwork'] [6]

(c i)

- Reuse as a result of using generic algorithms e.g. `find()`, `sort()`
- Reuse as a result of using common protocol within a container group e.g. `front()`
- Reuse of a programming idiom e.g. iterators are used for iteration through the elements of a container.

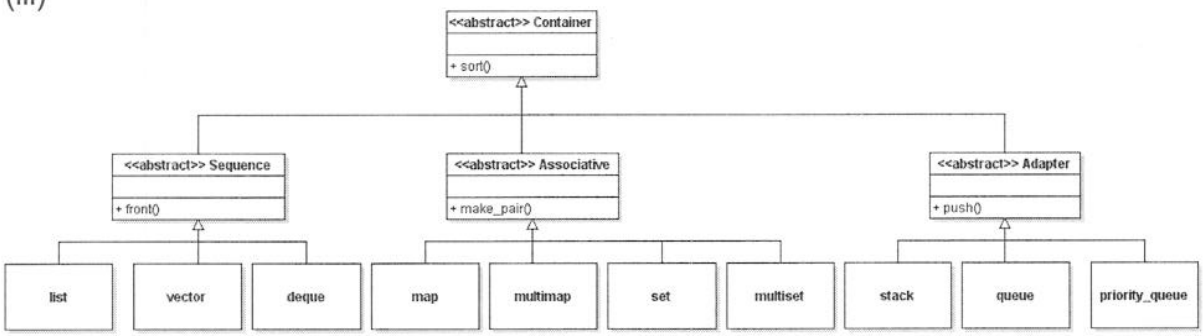
Need 1 of the above [1]

- Adaptability from using template data for container elements allowing user written classes to be stored.
- Adaptability from swapping within a container group e.g. `vector`, `deque` and `list` are all interchangeable.

Need 1 of the above [1]

(ii) In conventional OO, the generic algorithms would be located in the abstract class at the root of the container class hierarchy. Shared protocol amongst classes of the same category would be located in abstract classes for the appropriate category. Each concrete class would inherit from both abstract classes. [1]

(iii)



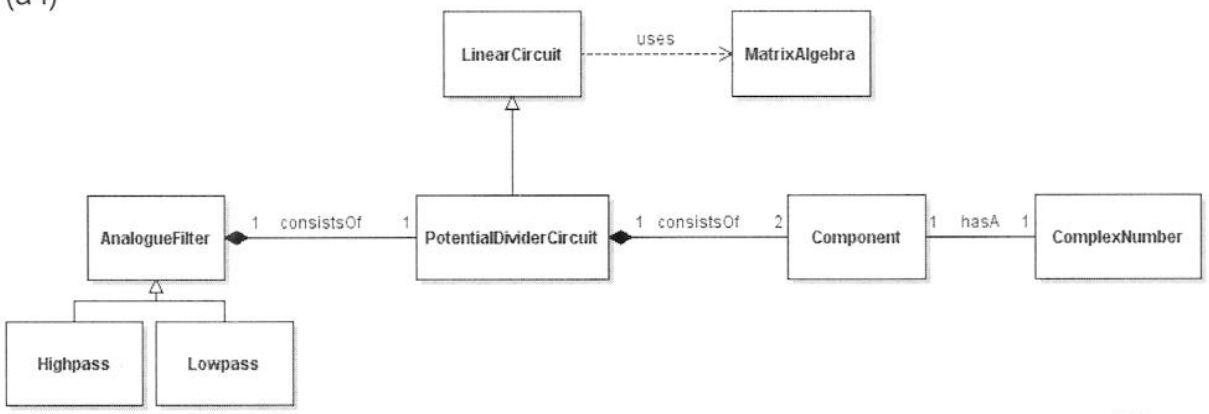
Looking for:

- "grandparent" container class,
- the 3 categories of container "parent" classes which would be abstract base classes,
- concrete derived "grandchild" classes corresponding to the STL classes containing a container specific member function. [3]

Q3

['new Application of theory'] [8]

(a i)



[2]

(a ii) 1 example of each dependency needed.

- Generalisation/Specialisation e.g. `Highpass`, `Lowpass` and `AnalogueFilter`, `LinearCircuit` and `PotentialDividerCircuit`.
- Association e.g. `Component` and `ComplexNumber`, `PotentialDividerCircuit` and `Component`, `AnalogueFilter` and `PotentialDividerCircuit`.
- Dependency e.g. `LinearCircuit` uses `MatrixAlgebra`

[2]

(a iii)

- Inheritance e.g.

```
class Highpass: public Filter { //etc…
```

- Composition e.g.

```
class Component { //etc…
  private:
    ComplexNumber cn1; /etc…
```

- Dependency e.g.

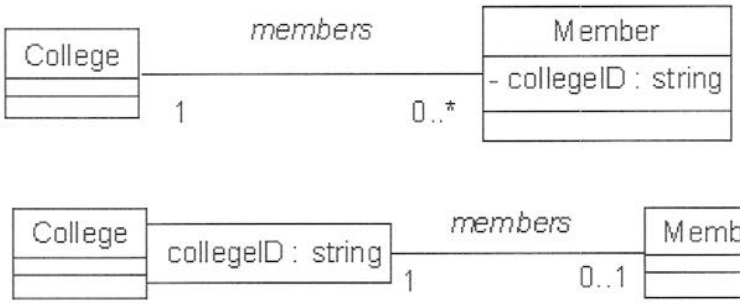
```
double getUnknownCurrent(Matrix aMatrix);
Matrix doNodalAnalysis(void);
```

[2]

(a iv) The conceptual model class diagram will have classes, obvious association, generalisation and dependency relationships between classes, obvious attributes, association names and multiplicities. The implementation model class diagram may have more of all of the aforementioned. Extra details include design and code level details e.g. an orchestrating class, navigation arrows on associations, member functions, visibility modifiers, abstract member functions and classes. Further design refinements include association classes, derived attributes and derived associations, qualified associations, aggregation/ composition associations. [2]

['new Application of theory'] [4]

(b)



[2x2]

i.e. attribute in `Member` is moved to qualified class `College` changing a 1-to-many association into a a 1-to-1 association.

['new Application of theory'] [4]

(c) Any 4 distinct statements e.g.

- Primary key and Alternate keys are kinds of Candidate key
- Every Relation must have a Candidate key
- Every Relation may have a Foreign key
- Every Relation consists of a heading and a body
- A Heading has many attributes
- A Body has many tuples
- A Candidate Key is associated with many attributes
- Candidate key must be minimal and unique (from note)

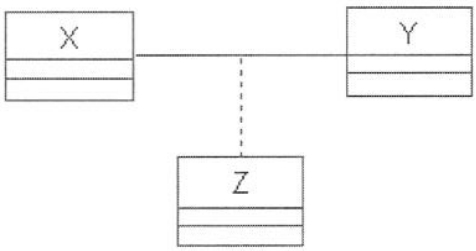
[4x1]

['new Application of theory'] [1]

(d) Invalid because can only be associated with 1 college, not 2 as in the object diagram.

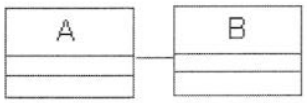
['new Application of theory'] [3]

(e i)



(e ii)

(e iii)



[3x1]

Q4.

['new Application of theory'] [9]

(a i) Any 5 distinct statements e.g. [5]

Top level functionality includes:

- Set input high
  - Deals with "Not primary input" error
- Initialise circuit
- Display all nodes
- Load circuit
  - Deals with "file" exceptions
- Simulate circuit
- All functions use an STL container class for collections of objects
- Border around diagram implies this part of the implementation model
- There is only one type of user for the program

(a ii) [2]

Identifier and name: UC#1, Set input high
Initiator: DLS User
Goal: To set a primary input logically high
Pre-condition: Node must exist, node must be primary
Post-condition: Primary input node is high

	DLS user	System
Success scenario	`H a`	
		Get object reference for node `a` in collection of nodes
		Check node is a primary input
		Set state high for object
Failure scenario	`H z`	
		Get object reference for node `z` in collection of nodes
		Error because node `z` not found

(a iii) In a test driven development (TDD) approach, tests are defined *at the same time as requirements are defined* i.e. before any code is written. This second deliverable from the requirements process is then used by the client as part of the acceptance testing that takes place by the client in their own environment. The fact that reverse engineering has taken place to provide scenarios reveals that not all tests were created initially, as the scenarios provide the basis for testing.

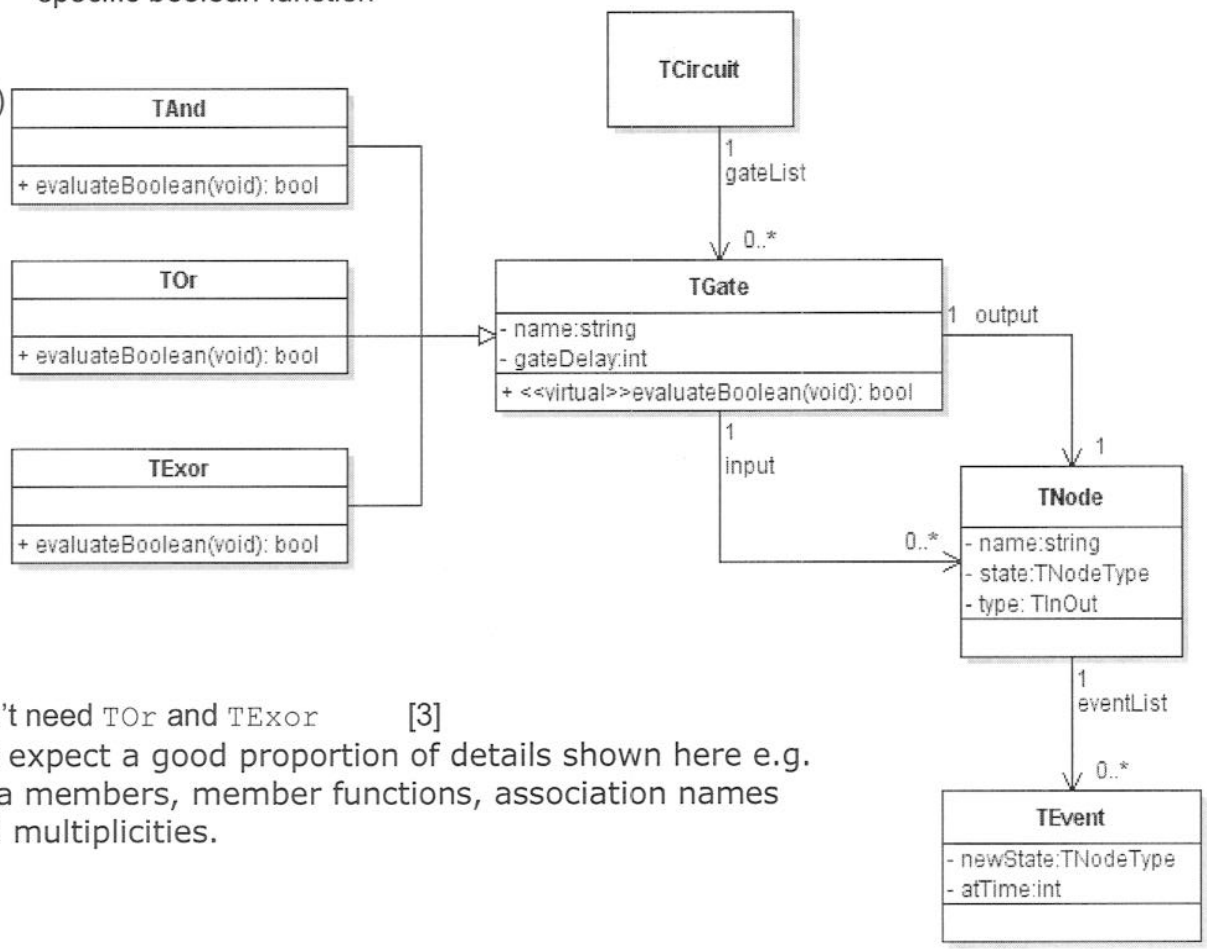
[2]

['bookwork'] ['new Application of theory'] [6]

(b i) Any 3 of the following: [3]

- Classification e.g. Tangible, Role, Event, Organisation. In the RS the class *Event* is obvious.
- Textual analysis e.g. noun identification. In the RS a *Gate* is a noun suggestive of a class.
- Textual analysis e.g. verb identification. In the RS associations between classes are apparent e.g. "Each circuit *has* a number of logic gates"
- Use of standard phrase to identify an association e.g. *has-A*. In the RS "Each gate has one output node"
- Use of standard phrase to identify a generalisation e.g. *is-A*. In the RS "an and-gate is a gate."
- Association multiplicities may appear in the text. In the RS "Each gate has *one* output node and has *one or more* input nodes"
- The attributes of classes are revealed and sometimes the primitive type of the attribute. In the RS "A gate is *named* e.g. "`nand2`" and there is a *gate delay* (measured in *nanoseconds*)."
- If it is very obvious then we may record in a supplementary text some protocol details, but not in the class diagram. In the RS "Each gate *evaluates* some specific boolean function"

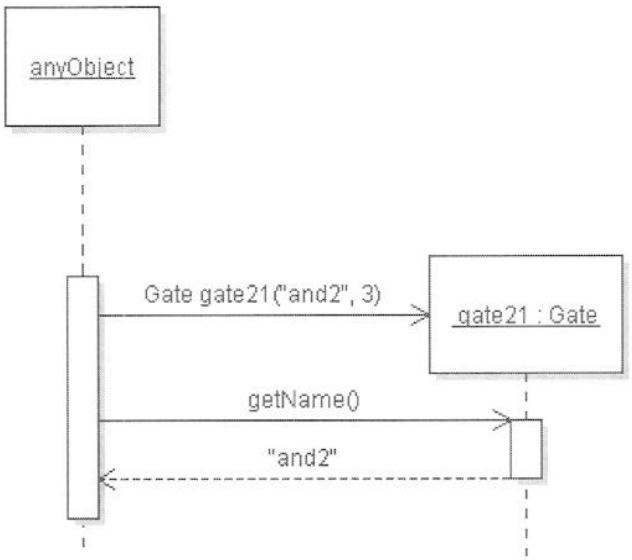
(b ii)



Don't need `TOr` and `TExor` [3]

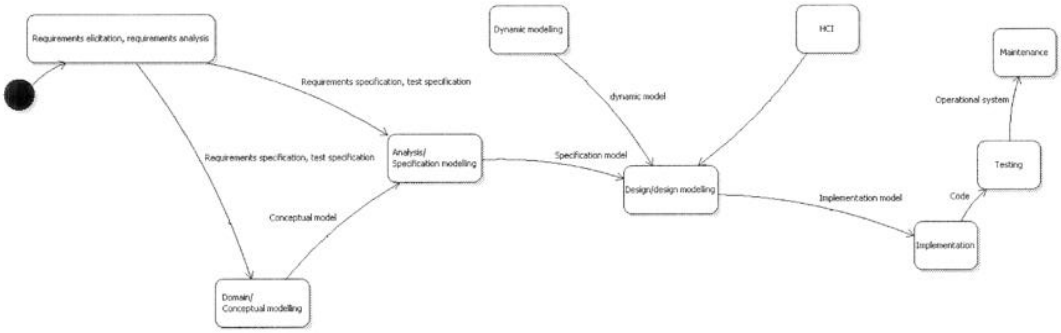
But expect a good proportion of details shown here e.g. data members, member functions, association names and multiplicities.

(c) ['new Application of theory'][2]



- `gate21` object must be at a lower horizontal level than `anyObject`
- `gate21` must be receiver of `getName()` message and generate a string as a message answer

(d) ['bookwork'][3]



Must show the Conceptual, Specification and Implementation models.