Indian Institute of Technology, Kharagpur

Department of Computer Science and Engineering

End-Semester Examination, Spring 2014-15

Software Engineering (CS 20006)

Students: 135 **Date**: 21-Apr-15 (AN)

Full marks: 100 **Time**: 3 hours

Instructions:

1. Marks for every question is shown with the question.
2. No clarification to any of the questions will be provided. If you have any doubt, please make suitable assumptions and proceed. State your assumptions clearly. While making assumptions, be careful that you do not contradict any explicitly stated fact in the question.

---

1. A course on Software Construction in IIT wants to manage the assignments to *Students*, the submissions of assignments by *Students*, and the evaluations of the submissions through an **Assignment Management System (AMS)**. The requirement specifications for the system are as follows:

   (a) Participants of the course are:

   - One *Instructor*. She / he is identified by an *Employee Code*, and has *Name*, *Email* and *Mobile Number*.
   - 5 or more *Teaching Assistants (TA)*. Every *TA* is identified by a *Roll No*, and has *Name*, *Email* and *Mobile Number*.
   - 100 or more *Students*. Every *Student* is identified by a *Roll No*, and has *Name*, *Department*, *Hall*, *Email* and *Mobile Number*.

   (b) The responsibilities of the *Instructor* include:

   - *Set-up*: Design and set up the assignments, decide on the date that an assignment is to be assigned to the *Students*, and on the date of submission for the assignment. Also decide on a *Coordinating TA* for an assignment.
   - *Allocation*: Allocate *Students* to *TA*. Every *Student* has one allocated *TA*.
   - *Approve*: Approve / Disapprove extensions for submissions beyond the submission date and decide on the penalty.
   - *Compilation*: Compile the evaluations (as performed by the *TA*s) and publish the final scores of every assignments on **AMS**.

   (c) The responsibilities of a *TA* include:

   - *Mentor*: Mentor and manage the *Students* allocated to the *TA*. She / he is the primary support for the allocated *Students* before she / he should approach the *Instructor*.
   - *Upload*: The *Coordinating TA* of an assignment uploads an assignment as set by the *Instructor* and sets up the required *Assign* and *Submission Dates* on the **AMS**.
   - *Download*: Download from the **AMS** and archive the submissions for the *Students* allocated to the *TA*. This needs to be done after the submission date in every assignment.
   - *Evaluate*: Peruse the submissions, discuss with the respective *Students* for clarifications, take demonstrations (if relevant), and evaluate.
   - *Report*: Report the evaluations to *Instructor*. Requests for submission date extension with full credit, on grounds of medical or personal exigencies, are also to be reported after proper authentication. Further, *TA*s are responsible for reporting Disciplinary actions (like plagiarism), if any.

(d) The responsibilities of a *Student* include:

- *Perform*: Complete every assignment within the submission date and submit to **AMS**.
- *Appeal*: Appeal to the *Instructor* for permission for special submission without penalty. Every appeal needs to go through the respective *TA* and must be authenticated by her / him.
- *Demonstrate*: Discuss and demonstrate the submission to the allocated *TA*.

(e) An *Assignment*:

- *Ownership*: Is to be completed individually by every *Student*.
- *Type*: Is one the following types:
  - *Programming Assignment*: The assignment has one *Problem* that asks to write a single program, specifies the language for coding (like C / C++ / Java), and has a single submission date.
  - *Systems Assignment*: The assignment asks to develop a System. It has one or more *Problem/s*. Each *Problem* is about a component module that builds up the System and has a separate submission date. No coding language is specified for such assignments.
- *Assign Date*: Has an *Assign Date* on which it is given to the *Students*.
- *Submission Date*: Has a *Submission Date* by which it is to be completed and submitted. For a *Systems Assignment* every constituent *Problem* has a separate *Submission Date* but a common *Assign Date*.
- *Marks*: Every *Problem* in an assignment has specified maximum marks.
- *Coordinator*: By turn a *TA* is allocated by the *Instructor* as a coordinator for an assignment. She / he manages the upload, and the dates for the assignment.

(f) A *Submission* :

- *Action*: Is performed for an *Assignment*, by a *Student* on a *Date*.
- *Valid*: Is *valid* if it is submitted within the *Submission Date* of the corresponding assignment. *Valid* submissions carry full credit.
- *Late*: Is *late* if it is done within 3 days from the *Submission Date* of the corresponding assignment. *Late* submissions carry 10% penalty.
- *Special*: Is *special* if it is done within 7 days from the *Submission Date* of the corresponding assignment. A *Student* needs to *appeal* for *Special* submissions to the *Instructor* through the *TA*. *Special* submissions are allowed on grounds of medical or personal exigency and carry no penalty.
- *Invalid*: Is *invalid* if it is not submitted within 3 days from the the *Submission Date* of the corresponding assignment and has not been granted extension as a *Special Submission*. *Invalid* submissions carry zero credit.

No submission is allowed before the *Assign Date* of the submission.

(g) The *Work flow* in the course is as follows:

- The *Instructor* designs an *Assignment* and mails it to the *Coordinating TA*.
- The *Coordinating TA* uploads the *Assignment* to the system and sets the corresponding *Assign* and *Submission Dates*.
- Once an *Assignment* has been set, **AMS** sends an email notification to all *Students*, *TA*s and the *Instructor* about the *Assignment*.
- The *Students* completes the *Assignment* and uploads the solution to **AMS**.
- Once the *Submission Dates* for an *Assignment* is over, **AMS** sends an email notification each to every *TA* with the respective list of completed submissions (*Students'* roll numbers are mentioned) by the *Students* allocated to the *TA*. Respective students are carbon-copied on the email. The *Instructor* is copied on every notification.
- Every *TA* downloads the respective submissions, discusses with the *Student*s, checks demonstrations, and evaluates the solution.
- Once a *TA* completes her / his evaluations for an *Assignment*, **AMS** sends an email notification to the *Instructor* with the *TA* on the carbon-copy. *TA* should complete the evaluations after 4 days from the *Submission Date* and before 10 days from it.

- The *Instructor* on receipt of completion report from all the *TA*s compiles the scores for an *Assignment* and publishes on the **AMS**.
- If a *Student* misses to submit by the *Submission Date*, but submits within 3 days from that, she / he is penalized by 10%. On such submissions, **AMS** sends an email notification to the *TA* and the *TA* would similarly evaluate the submission and report to the *Instructor*.
- A *Student* may appeal for an extension on grounds of medical or personal exigency. The *TA* would scrutinize the appeal and report to the *Instructor*, if authentic. **AMS** will send an email notification for the same. As the *Instructor* approves or disapproves the appeal, accordingly the *Student* and the *TA* are notified. If the appeal is approved, the process of submission and evaluation is followed.

You have been assigned as the software engineer for the **AMS**. You are required to analyse the specifications, design the system (using UML and DP) and also prepare the test plan. Answer the following questions in this background.

(a) Identify the actions in **AMS** and design the Use-Case Diagrams for the actions. Identify the actors, specify their types, and mark the relationships between the actors. Show the <<`include`>>, <<`extend`>>, and generalization relationships of the use-cases. **[4+4=8]**

(b) Design Class Diagrams for *Assignments* & *Submissions*. Show the attributes and operations with their associated properties. Highlight specialization hierarchies, if any. **[4+4=8]**

(c) Complete the Class Diagram of **AMS** showing all other classes (in addition to Question 1b) by their respective brief Diagrams (with name and key attributes). For the entire collection of classes (that is, including *Assignments* and *Submissions*) show the associations, aggregations / compositions, generalization / specialization, and abstract / concrete etc. **[6]**

(d) Design the State-Chart Diagrams for *Assignments* and *Submissions*. **[2+2=4]**

(e) Design Sequence Diagrams for the actions in **AMS** as specified in the Work flow. **[10]**

(f) Identify and justify the use of Iterator, Singleton and Command DPs in **AMS**. **[2*3=6]**

(g) Prepare a test plan for **AMS** to perform black-box tests. Clearly mark the scenarios for Unit Testing and Integration Testing. **[4+4=8]**

2. Let `unique_ptr` be an *Exclusive Ownership – No Copy* smart pointer. It is not possible to copy such pointers. The ownership can only be changed through `swap()`. The interface for `unique_ptr` is given below:

```
template<class T> class unique_ptr { T* ptr_;   // The raw pointer
    unique_ptr(unique_ptr<T>& p);               // Copy constructor
    unique_ptr& operator=(unique_ptr<T>& p);    // Copy assignment
public:
    explicit unique_ptr(T* p) throw();          // RAII Constructor
    unique_ptr() throw();                       // Default Constructor
    ~unique_ptr();                              // Releases the pointer (with destruction)
    T& operator*() const;                       // Dereference operator
    T* operator->() const throw();              // Indirection operator
    operator bool() const throw();              // Returns whether the unique_ptr is not empty
    T* get() const throw();                     // Gets the raw pointer (ptr_)
    T* release() throw();                       // Returns the raw pointer (ptr_) & nulls its value
                                                // (w/o destruction)
    void swap (unique_ptr& u) throw();          // Exchanges the contents (ptr_) of the unique_ptr
                                                // object with those of u -- w/o destruction
};
```

(a) Implement the `unique_ptr` class. **[1*7+1.5*2=10]**

(b) Write an application to black-box test all methods of the `unique_ptr` class as implemented. **[10]**

3. Write the output for the following program: **[1*9=9]**

```
#include <exception>
#include <iostream>
using namespace std;

struct Excp: public exception {
    int data;
    Excp(int d): data(d) {
        cout << "Excp(" << data << ")" << endl; }
    ~Excp() {
         cout << "~Excp(" << data << ")" << endl; }
};

int f(int n) {
    try {
        if (0 == n) {
            throw Excp(n);
            cout << "recur for n = " << n << endl;
        }
        return f(n-1);
    }
    catch (Excp& e) {
        throw Excp((e.data == 0)? 1: -n*e.data);
    }
    cout << "param = " << n << endl;
}
```

```
int main() {
    int n = 2, r = 0;
    try {
        r = f(n);
        cout << "output = " << r << endl;
    }
    catch (Excp& e) { r = e.data; }

    cout << "result = " << r << endl;
    return 0;
}
```

4. The following code uses two types of smart pointers from the `memory` component of C++ Standard Library:

- The behaviour of `auto_ptr` is defined as:

  `auto_ptr` objects have the peculiarity of taking ownership of the pointers assigned to them: An `auto_ptr` object that has ownership over one element is in charge of destroying the element it points to and to deallocate the memory allocated to it when itself is destroyed.
  When an assignment operation takes place between two `auto_ptr` objects, ownership is transferred, which means that the object losing ownership is set to no longer point to the element (it is set to the null pointer).

  Hence, an `auto_ptr` is an *Exclusive Ownership – Destructive Copy* smart pointer.

- The behaviour of `shared_ptr` is defined as:

  Objects of `shared_ptr` types have the ability of taking ownership of a pointer and share that ownership: once they take ownership, the group of owners of a pointer become responsible for its deletion when the last one of them releases that ownership.

  Hence, a `shared_ptr` is a *Shared Ownership – Reference Counting* smart pointer.

Read the code carefully to understand the resource (memory) management by the smart pointers and the ensuing lifetime of the objects. Based on your understanding write the output from the code.

The marks for this question are as follows:

Output from Block	Marks	Remarks
`using_shared_ptr() Blk`	**0.5*4 = 2**	Write output from this block only – not the nested blocks
`auto_ptr Blk`	**0.5*10 = 5**	Some output from this block may be printed after ...`END`
`shared_ptr Blk 1`	**0.5*4 = 2**	Some output from this block may be printed after ...`END`
`shared_ptr Blk 2`	**0.5*4 = 2**	Some output from this block may be printed after ...`END`
`shared_ptr Blk 3`	**0.5*2 = 1**	Some output from this block may be printed after ...`END`
Interleaving between blocks	**3**	Some blocks are nested in others
Total	[**15**]	

Code for Q 4. Write the output.

```
#include <memory>
#include <iostream>
#include <string>
using namespace std;

struct Node {
    int data;
    shared_ptr<Node> hard;
    Node* soft;

    Node(int d=0): data(d), hard(0), soft(0)
        { cout << "A::A() Data = "
               << data << endl; }
    ~Node() { cout << "A::~A() Data = "
                  << data << endl; }
};

void Write_auto_ptr(string name,
         auto_ptr<Node>& a) {
    cout << name << ": ";
    cout << ((a.get())? "!0 = " : "0");
    if (a.get())
        cout << a->data << endl;
    else
        cout << endl;
}

void using_shared_ptr() {
    cout << "using_shared_ptr() Blk START\n";

    Node n;

    {
        cout << "auto_ptr Blk START\n";

        auto_ptr<Node> p(new Node(100));
        auto_ptr<Node> q(new Node(200));
        Write_auto_ptr("p", p);
        Write_auto_ptr("q", q);

        q = p;
        Write_auto_ptr("p", p);
        Write_auto_ptr("q", q);

        auto_ptr<Node> r(q);
        Write_auto_ptr("r", r);
        Write_auto_ptr("q", q);

        cout << "auto_ptr Blk END\n\n";
    }
    shared_ptr<Node> p1(new Node(111));

    {
        cout << "shared_ptr Blk 1 START\n";

        shared_ptr<Node> p2(new Node(222));
        p2 = 0;

        shared_ptr<Node> p3(new Node(333));
        shared_ptr<Node> p3_copy(p3);
        p3 = 0;

        cout << "shared_ptr Blk 1 END\n\n";
    }

    {
        cout << "shared_ptr Blk 2 START\n";

        Node *p4 = new Node(444);
        shared_ptr<Node> p5(new Node(555));
        p5->hard = shared_ptr<Node>(p4);
        p4->soft = p5.get();

        cout << "shared_ptr Blk 2 END\n\n";
    }

    {
        cout << "shared_ptr Blk 3 START\n";

        Node *p6 = new Node(666);
        shared_ptr<Node> p7(new Node(777));
        p7->hard = shared_ptr<Node>(p6);
        p6->hard = p7;

        cout << "shared_ptr Blk 3 END\n\n";
    }

    cout << "using_shared_ptr() Blk END\n";
    return;
}

int main() {
    using_shared_ptr();

    return 0;
}
```

5. Write the output from the following code: **[0.5*12=6]**

```
#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;

struct product: public unary_function<double, void> { product() : prod(1), count(0) {}
        int prod; unsigned int count; void operator()(int i) { prod *= i; ++count; }
};

struct gen { gen() : item(0) {}
    int item; int operator()() { return (++item % 2)? -item: item; }
};

struct compare: public binary_function<int, int, bool> {
        bool operator()(int x, int y) { return x > y; }
};

int main() {
    vector<int> V(5);

    generate(V.begin(), V.end(), gen());
    cout << "Filled Vector is:" << endl;
    for(vector<int>::const_iterator it = V.begin(); it != V.end(); ++it)
        cout << *it << " ";
    cout << endl << endl;

    product result = for_each(V.begin(), V.end(), product());
    cout << "Product of " << result.count << " numbers is " << result.prod << endl << endl;

    sort(V.begin(), V.end(), compare());
    cout << "Sorted Vector is:" << endl;
    for(vector<int>::const_iterator it = V.begin(); it != V.end(); ++it)
        cout << *it << " ";
    cout << endl;

    return 0;
}
```