NATIONAL UNIVERSITY OF SINGAPORE

SCHOOL OF COMPUTING

EXAMINATION FOR
Semester 1 AY2010/2011

CS2103 – SOFTWARE ENGINEERING

Nov 2010 Time Allowed: 2 Hours

INSTRUCTIONS TO CANDIDATES

1. This examination paper contains **FOUR (4)** questions and comprises **TEN (10)** printed pages, including this page.
2. Answer **ALL** questions within the space in this booklet
3. This is an **Open Book** examination.
4. Please write your Matriculation Number below.

MATRICULATION NO: _____

This portion is for examiner's use only

Question	Marks	Remarks
Q1	/11	
Q2	/13	
Q3	/13	
Q4	/13	
Total	/50	

Q1 [11 marks]:

(a) [6 marks] Assume you are building a *Module management system* for a university. Here is some information about the domain.

> A module is taught by a team of instructors and coordinated by one of those instructors. Each module has exactly one coordinator. The maximum number of modules an instructor could be teaching is three while an instructor is not allowed to coordinate more than two modules at a time. Some instructors do not teach or coordinate any modules in some semesters. A module has one or more tutorial groups. A student should belong to one tutorial group from each module he/she is taking. Each tutorial is taught by only one instructor. If the module has a team project, each project group is supervised by one supervisor. The maximum and minimum team size varies from module to module.

The system should be able to answer questions such as 'who is teaching module CS2103?', 'Which tutorial groups in which modules are attended by the student John Doe?' However, the system need not answer questions about previous semesters.

Build an object oriented domain model (OODM) for the information given above. State all multiplicities, all necessary attributes, and any important constraints. You may use association labels and role names where they help to clarify the diagram further. Use the skeletal diagram given below as the starting point.

TutorialGroup

Module

Instructor

ProjectGroup

Student

(b) [2 marks]
Which classes (if any) in the domain model in (a) can be improved by applying the *Abstraction occurrence* pattern? For each class on which the pattern is applied, show the resulting OODM fragment after the pattern is applied.

(c) [3 marks]
Now, assume that the system explained in part (a) is also required to answer questions such as the below.

- Did Jane Doe attend tutorial 3 of module CS2103?
- Did Jan Doe submit answers for tutorial 8 of module CS2103?

Assuming you have already modified the OODM to accommodate these new requirements, give the relevant part of the OODM that will help to answer these questions. State all multiplicities and all necessary attributes. You may use association labels and role names where they help to clarify the diagram further.

Q2 [13 marks]

(a) [4 marks]
Assume you are building a software tool called *ExplorerPlus* that has some powerful file handling functionalities. One of those functionalities is the 'remove duplicates' feature described below.

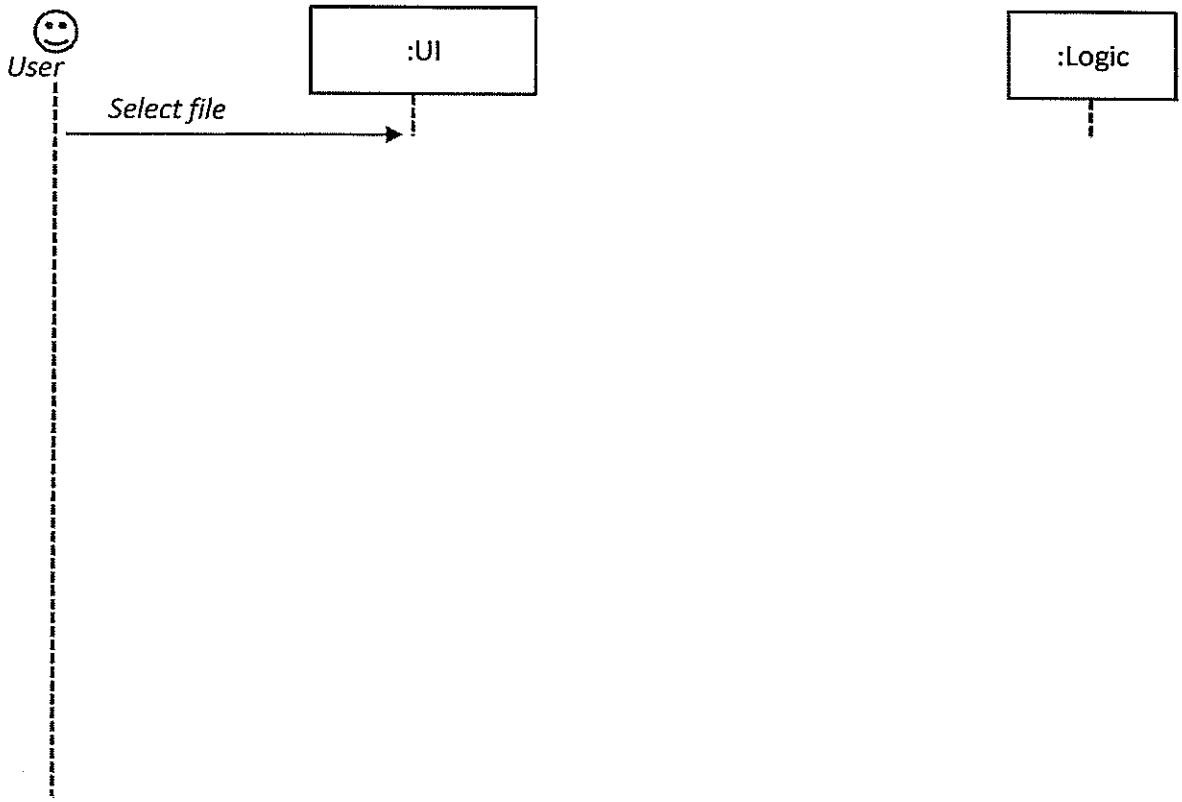
Remove duplicates: Sometimes, we end up having copies of the same file in different locations of the hard disk. This feature allows users to search for such duplicates of a given file and selectively delete some of the duplicates.

Complete the *Remove duplicates* use case given below. You need not show more than three extensions. However, it is possible that the use case does not need three extensions.

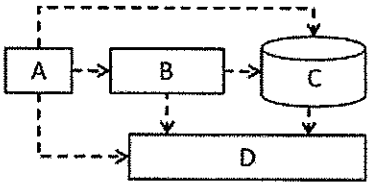
Use case: Remove duplicates
System: Explorer Plus
Actor: user
Preconditions: none
Guarantees: none
MSS:

(b) [4 marks]
Assume the *ExplorerPlus* is to be built following the architecture given below.

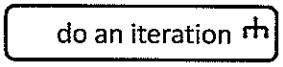
Use a sequence diagram to discover the interaction between the *UI* and *Logic* that is required to support the use case described in (a) above. Use proper sequence diagram notation. Clearly specify which operations of the *Logic* API are called by the *UI*. Show the main success scenario and up to one extension (if any).



(c) [5 marks]
A project team is building a software product that has the following architecture.



The main components of the product – i.e., A, B, C and D - are developed in parallel by four different developers. The team is using an agile process. After an initial study of the requirements, the rest of the project is done in iterative fashion. Each iteration adds the next highest priority feature to the product. The iteration starts with a discussion that selects which feature to implement in that iteration. The iteration ends with acceptance testing done by the project team in the presence of the customer. The system integration is done once per iteration and is done in bottom-up fashion. Since the company does not have a separate QA department, system testing is done by the same project team. The project keeps iterating until the customer stops asking for more features.
Show the workflow of the project as activity diagrams. You may omit unit testing from the activity diagram. Some fragments from two partial activity diagrams are given for your guidance.



Q3 [13 marks]

(a) [4 marks]

Given below is an operation from the Logic component of a *Snakes and Ladders* game.

movePiece(PLAYER *p*, int *face_value*):void - This operation moves the player *p*'s piece by *face_value* number of squares to a destination square, calculated as destination_square = current_square+ face_value.

If it is not possible for the piece to move because the destination_square is not free or does not exist, it remains in the same square. If the piece does move and ends up at a snake head or a ladder foot, the piece is moved to the snake tail or the ladder top, respectively.

Other things to note:

- The board is numbered 1-100. The game starts from square 1. The game ends when a player reaches the 100th square.
- The game can be played by 2-4 players. Only one die is used.
- PLAYER is an enumeration that has the values : P1, P2, P3, P4.
- The game does not allow having multiple pieces on the same square.

If we are using a defensive approach to testing, what are the equivalence partitions for p, face_value, and the destination_square? The table already gives one sample for each.

p	p has turn,
face_value	[1..6],
destination	Empty square,

(b) [4 marks]
Use the equivalent partitions from (a) to design an efficient and effective set of test cases to test the movePiece operation. One test case is already given. You may not use more than 10 test cases in total.

p	face_value	Destination square
Has the turn	[1..6]	empty

(c) [5 marks] Fill the table below to show the minimum number of test cases required to achieve each type of test coverage of class A. Note that NumberGen.random and NumberGen.high are class level variables from the NumberGen class.

```
class A{
        int x = 0;
        void foo(){
                x = 5;
                bar()
        }//end of foo

        void bar(){
                if((NumberGen.random==6) && (NumberGen.high==34)) { print("ok");}
                else{print("not ok");}
        }//end of bar
}//end of class A
```

Type of coverage	Number of test cases required
Function coverage	
Statement coverage	
Condition coverage	
Branch coverage	
Path coverage	

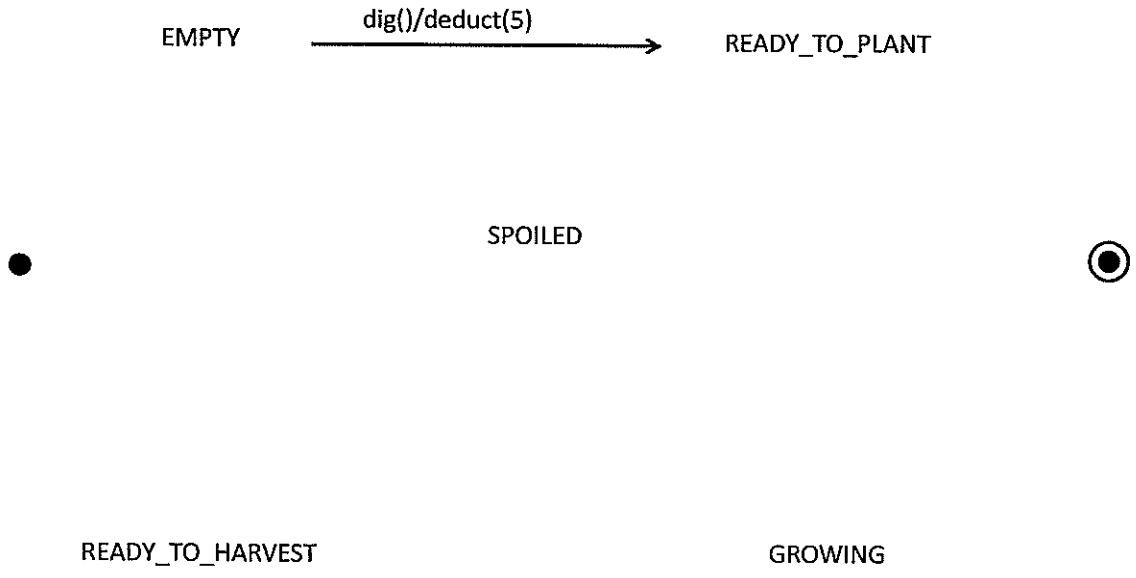
Q4 [13 marks]

(a) [7 marks] Given below is the behavior of a patch (an area used to grow crops) in a FarmVille-like farm simulation game.

Creating a patch costs 15 coins. A new patch needs to be dug before seeds can be planted. Digging costs 5 coins and planting costs 10 coins. It takes 24 hours for the plants to grow and reach harvesting age. When a patch is harvested, the player earns 50 coins. A harvested patch needs to be dug again to prepare it for planting again. A patch that has growing crops or harvest-age crops can be attacked by stray cows. If attacked, the crop in the patch will be destroyed and the patch becomes spoiled. Digging a spoiled patch to make it ready for planting again costs 10 coins. An unspoiled patch that has growing crops or harvest-age crops can be sold for 20 coins. Selling here simply means the owner is changed while the patch continues to function as before. Patches in other conditions can only sell for 5 coins. A dug patch or a patch that already has crops cannot be dug again. Removing a patch does not cost or earn anything and can be done at any stage.

Capture the above behavior as a state machine diagram. The skeletal diagram below indicates all the states that need to be included in the diagram. Given below are the events and activities you may consider including in the diagram. You may add superstates if required.

create(), dig(), harvest(), attack(), plant(), remove(), sell(*newOwner*), changeOwner(*newOwner*)
deduct(*c*) , earn(*c*) //*c* is the number of coins to be deducted from the owner
grow(*h*) // *h* is the number of hours taken to grow



(b) [3 marks] Complete the dig() operation of the Patch class given below. It should match the state machine diagram you gave for (a). Use the switch-case technique when implementing the state-dependent behavior.

```
void dig(){

}
```

(c) [3 marks]The phrase 'same stimulus, different responses' can be used to explain both *polymorphic behavior* and *state-dependent behavior* of a software component. What is the difference between them?
Use a game console software as an example (you are free to use a different example too) to illustrate how both types of behavior can exist in a single software component.

---End of paper---