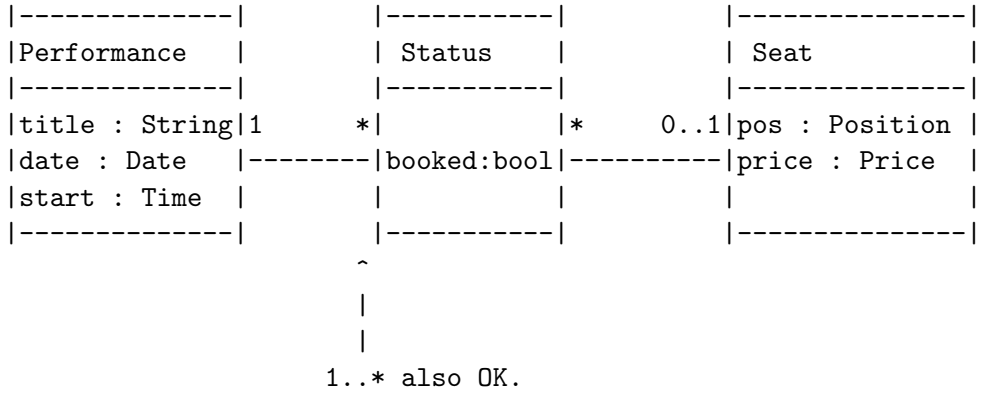
Module Title: Inf2C-SE
Exam Diet: Dec 2016
Brief notes on answers:

1. (a) *Application of knowledge.* Expect a solution similar to:



Marking:
2 marks: Correct notation for classes, associations and typed attributes
2 marks: Appropriate presentation of each class as box or attribute
2 marks: Correct associations
2 marks: Correct multiplicities on association ends.

(b) *Bookwork.*

(i). In lecture it was mentioned that collective ownership helps to keep code easy to understand and maintain, to spread knowledge of the system around the team, and to give team members a sense of empowerment.
Marking:
3 marks for giving two reasons why it is effective

(ii). Related practices include

- coding standards,
- refactoring,
- simple design.

All contribute to keeping code easily understandable and maintainable, and so help collective ownership to be effective.
Marking:
For each of the two other activities, 1 mark for naming it and 1.5 marks for explaining why.

Answers here might well vary from those above.

(c) *Bookwork, problem solving.*

(i). The test runner should execute the method before *each* test method.
Marking:
Award 2 marks if answer makes clear that method is run before each, rather than maybe run just once before running all the test methods. Award 1 mark if answer is ambiguous. Award 0 marks if answer says method is run before all the tests.

(ii).
```
public class KeepMaxTest {

    @Test
    public void updateUpdates() {
       KeepMax k = new KeepMax(3);
       assertEquals("max of 3 and 5 is 5",5, k.update(5));
    }
    @Test
    public void updateDoesNotUpdate() {
       KeepMax k = new KeepMax(3);
       assertEquals("max of 3 and 2 is 3", 3, k.update(2));
    }
}
```

Marking:
1 mark: `@Test` annotations.
2 marks: Sensible setups exercising condition both ways.
1 mark: Use of most appropriate assert methods for checks afterwards: `assertEquals()` rather than `assertTrue( ... == ...)`.
1 mark: Expected argument of `assertEquals()` before actual argument. (Order is important as it can affect full message when exception thrown.)
1 mark: Correct conditions are actually checked.
1 mark: Use of comment strings in assert method calls.

2. (a) *Application of knowledge.* Four approaches were discussed in lecture: *interviews*, *walking through scenarios*, *using prototypes* and *observation*. Of these observation would probably be most important, so as to understand exactly what is involved. The safety criticality justifies the investment. Next most important would likely be making use of prototypes because of the high quality of feedback. Again, the safety criticality justifies the investment. However, students might offer good plausible arguments in favour of other approaches too.

Marking:
For each approach selected, 1 mark for naming it and up to 2 marks for a good plausible explanation.

(b) *Problem solving.* The *Template Method* pattern is appropriate. To use this pattern, a framework is set up using an abstract class as follows:

```
abstract class GameFramework {
  abstract void setup();
  abstract void makeStep();
  abstract boolean gameOver();
  abstract void printScore();
  void playGame() {
    setup();
    do {
      makeStep();
    } while (!gameOver());
    printScore();
  }
```

The colleague would then have their `Game` class extend this class and would delete the definition of the `playGame()` method from their class.

Marking:
1 mark for correct design pattern name.
4 marks for characterisation of framework class. Acceptable answers are as above or some equivalent description.
3 marks for how they would adapt their `Game` class.

(c) *Bookwork.* Regression testing involves regularly, perhaps daily, running a full suite of tests after each modification to code. Regression testing ensures that any bugs introduced by modifications are quickly found and fixed. It ensures the whole code base is maintained in a fully operational form.

Marking:
2 marks for definition,
2 marks for why useful.

(d) *Bookwork.* MTTF = Mean Time To Failure. MTTF is not sufficient for predicting likelihood of non-failure in a given period. For that one needs more information on the distribution of failure times. For example, if the time to failure is approximately normally distributed, it would be useful to know the standard deviation of the time to failure.

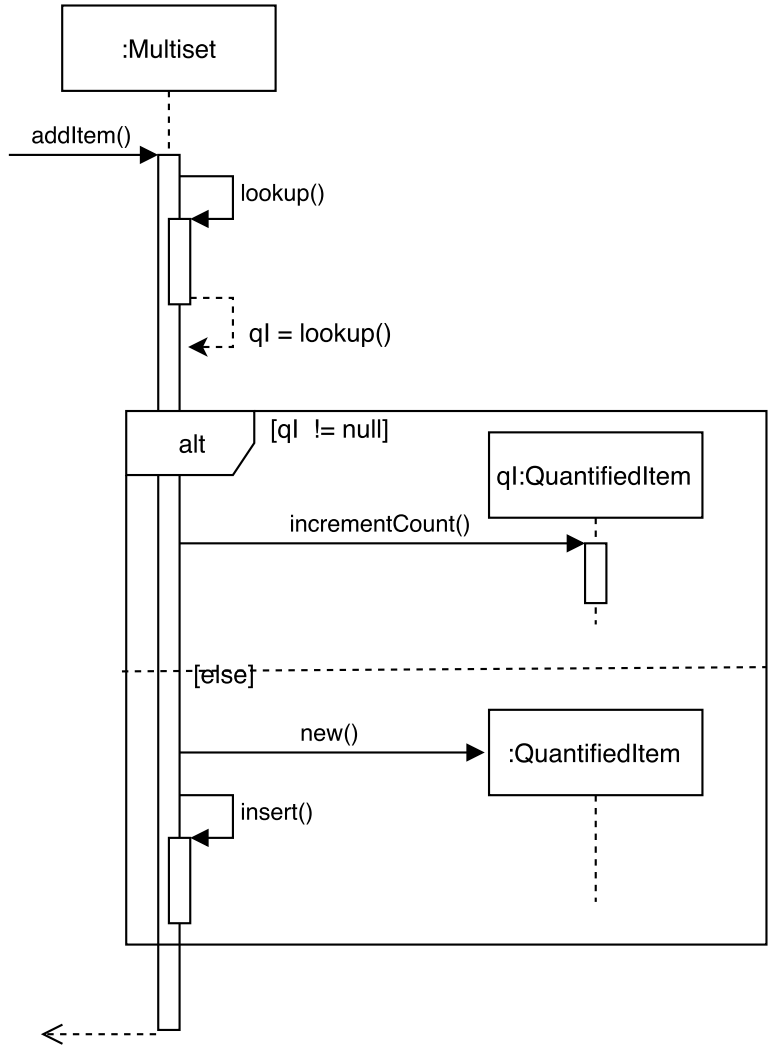
Marking:
1 mark for abbreviation.
1 mark for for stating it is not sufficient.
2 marks for explaining why.

(e) *Bookwork:* User familiarity is about using concepts and entities from the existing experience of users.

For example, a standard part of the widely-used desktop metaphor is to represent directories as folders and refer to them as such.

Marking: 1.5 marks for the idea, 1.5 marks for a good example.

3. (a) *Application of knowledge / problem solving:* The sequence diagram should closely resemble this.



Issues to look for in the diagram include:

1 Basic diagram syntax captured (object rectangles with dashed lifelines descending below. Horizontal solid arrow messages and dashed arrow responses.)

2 Correct ordering of messages and messages going between correct objects.

3 Use of UML *alt* block with appropriate condition for if region and [else] for else region.

4 Local variable value qI set on method return.

5 QuantifiedItem objects in distinct parts of alt block are distinct.

6 Lifeline of second QuantifiedItem object starting with its construction, but first should already exist when incrementCount is called.

7 Activation bars for lookup() and insert() calls half overlap activation bar for addItem() invocation.

Marking: 2 marks for each of first two. 1 mark for each of further 5 points.

(b) *Bookwork, application of knowledge.*

(i). A make rule has a *target*, one or more *dependencies* and one or more *commands*. For example, in the first rule, the target is `a`, the dependencies are `a.o` and `b.o`, and the one command is `gcc -o a a.o b.o`. (Answers without the make variables expanded are fine too.)
Marking:
3 marks: Components of a rule correctly named and examples of each correctly identified.

(ii). The make command uses a rule when the target does not exist or at least one of the dependencies is newer than the target.
Marking:
1.5 marks for each of these two main circumstances.

(iii). On running the command `make a`,
1. the 3rd rule fires, generating a new file `b.o`, and then
2. the 1st rule fires, generating a new file `a`.

Marking:
3 marks

(c) *bookwork:* We re-engineer software if it has become old or unmaintainable, but we need to keep it running and perhaps evolve it further. Activities mentioned in lecture were

- *Source code translation* e.g. from obsolete language, or assembly, to modern language.
- *Reverse engineering* i.e. analysing the program, possibly in the absence of source code.
- *Structure improvement*, especially *modularization*, *architectural refactoring*
- *Data re-engineering*, reformatting and cleaning up data.
- *Adding adaptor interfaces* to users and newer other software

Marking:
2 marks for definition. 2.5 marks for each of two activities.