Quiz 7b Rubric

1. (2 points) We want to create an object-oriented system for representing trees (the living ones, not the CS ones). What should each of the following be? Choose out of the following list:

Class variable, Instance variable, Method, Object, Subclass.

* 1. General Sherman (biggest tree in the world) Object
  2. Number of trees in the Amazon Class variable
  3. Photosynthesis (making food) Method
  4. Coniferous (cone-shaped) trees Subclass

0.5 points for each, all or nothing.

1. (3 points) Andrew doesn’t like Piazza. He thinks that we should write our own, better version of Piazza. However, he’s really busy and so has hired you to help. He wants you to create objects that represent the things he needs to work with in Piazza.

Write a question-thread class that represents an entire question thread on Piazza. It accepts a question as an instantiation argument. It should store the question, the student answer, the instructor’s answer, and a list of follow-ups. A follow-up is an object which has one method: add-comment, which adds a comment (taken as an argument) to that follow-up. Your question-thread class should support the following methods:

* unanswered? returns #t if there is no answer (student or instructor), else #f.
* set-student-answer accepts 1 argument, and sets the student answer to that argument.
* set-instructor-answer is the corresponding method for the instructor answer.
* new-follow-up accepts a follow-up as an argument, and stores it.
* add-comment accepts an index j and a comment c, and adds c to the jth follow-up. (The first follow-up to be added using new-follow-up would be the 0th follow-up.)
* There should be methods to access the answers and the follow-ups.

(define-class (question-thread question)

(instance-vars (student-answer #f) (instructor-answer #f) (follow-ups ‘()))

(method (unanswered?) (not (or student-answer instructor-answer)))

(method (set-student-answer ans) (set! student-answer ans))

(method (set-instructor-answer ans) (set! instructor-answer ans))

(method (new-follow-up f) (set! follow-ups (append follow-ups (list f))))

(method (add-comment j c)

(ask (list-ref follow-ups j) ‘add-comment c)

-0.5 points if they indexed into the list of follow-ups incorrectly for add-comment.

No points deducted if they added getters for everything (for the last bullet point).

No points deducted if they have multiple instance-vars clauses, although technically that is wrong. Apart from these guidelines, use your best judgment.

1. (1+1+3 points) Define a class ideal-fruit-shop that has a fruit and a corresponding price. An ideal-fruit-shop accepts the message buy, which takes as argument a quantity. buy should return the total price of the purchase.

> (define orange-shop (instantiate ideal-fruit-shop ‘orange 3))

> (ask orange-shop ‘buy 4)

12

(define-class (ideal-fruit-shop fruit price)

(method (buy quantity)

(\* quantity price)))

1 point for this part. Almost everyone should get 0.5 points – simply using define-class and method is enough to get 0.5 points.

Louis Reasoner looked at the code you wrote above, and was confused. “You take in a fruit as an argument when creating an ideal-fruit-shop, but you never use it in any of your methods! Why bother having it at all?”

You answer that there actually is a purpose for having the fruit as an instantiation variable. What is this purpose?

An ideal-fruit-shop gets a method called fruit. (This is all they need to say for the point.) This is necessary so that a user to find out which fruit he is buying from the fruit shop.

As the name suggest, an ideal-fruit-shop is ideal. Let’s make it more realistic. A fruit-shop inherits from ideal-fruit-shop, but it also has an amount of the fruit that the fruit shop has. When we send a buy message to a fruit-shop, it must check if the fruit shop has enough of the fruit to sell. If it doesn’t, it executes (error “Not enough fruit”). If it does, it must reduce the amount of the fruit by the appropriate amount (and then return the price of the purchase). Write the fruit-shop class. It **must** take advantage of inheritance wherever possible.

> (define orange-shop (instantiate fruit-shop ‘orange 3 10))

> (ask orange-shop ‘buy 6)

18

> (ask orange-shop ‘buy 6)

Error: Not enough fruit

(define-class (fruit-shop fruit price amount)

(parent (ideal-fruit-shop fruit price))

(method (buy quantity)

(if (> quantity amount)

(error “Not enough fruit”)

(begin (set! amount (- amount quantity))

(usual ‘buy quantity)))))

-0.5 points for incorrect parent clause, -1 for missing parent clause

-0.5 points for incorrect usage of usual, -1 for not using usual

-0.5 points for not checking whether the quantity is too much

-0.5 points for not updating the amount

No points deducted for saying >= or <= instead of > or < in the test.