Homework 1: Introductory Smalltalk Programming

Due:

The purpose of this homework is to ensure basic skills in Smalltalk programming, and to help you explore the literature a bit. This is an individual homework; that is, for this homework, you are to do the work on your own, not in groups.

For all Smalltalk programs, you must run your code with little Smalltalk. Hand in a printout of your code and your testing. (You can use the `script` command to give a transcript, or run Smalltalk from within an emacs shell to get test output captured, or you can use the Smalltalk printing primitives to print test output to a file.)

Don't write code that looks like C! We'll take points off for poor style.

1. (10 points) Define (in Little Smalltalk) a class `Account`, which models accounts at a savings-and-loan (bank). As in TR91-22a, we model the state of each object as a mathematical entity, called its "abstract value". The abstract value of an account object is a pair; this pair contains the owner's name (a string) and a balance (a number). A newly created account should have a balance of 0 and an owner's name which is empty. This class should have the following instance operations:
   - `owner:`, which sets the account's owner's name,
   - `owner`, which return's the account's owner's name,
   - `deposit:`, which adds the argument amount to the balance. The argument must be positive.
   - `withdraw:`, which subtracts the argument amount from the balance. The argument must be positive.
   - `balance`, which returns the account's balance.
   - `payInterest:`, which takes as an argument a *rate* (a fraction between 0 and 1), and pays interest in the account at the given rate. For example, if `acc` is an `Account` whose balance is 100, then the state after the message send `acc payInterest: 0.04` is that the balance of `acc` is 104.
   - `equalAsAccount:`, which tests to see whether its argument has the same owner and balance.
2. (extra credit, 10 points for the programming and semantics tracks) Implement `Account` in a different way than you did in the above problem. Call the new class `Account2`. (For example, if you used two instance variables before, use an array now, or vice versa). Can a program work with both `Account` and `Account2` objects? Why or why not?
3. (extra credit, 10 points for applications—software engineering) Specify the class `Account` using the notation of TR91-22a.
4. (15 points) Design and implement a class `PlusAccount` as a subclass of `Account`. A `PlusAccount` object's abstract value is a tuple of an owner's name, a savings balance, and a checking balance. A `PlusAccount` acts like an `Account` in which the checking balance is dropped; that is, the savings balance of a `PlusAccount` object is given by sending it the message `balance`. `PlusAccount` should have extra operations `checkingDeposit:`, and `checkingWithdraw:`, and `CheckingBalance`, and `equalAsPlusAccount:`. The message `equalAsPlusAccount:` should test if the argument has the same owner's name, savings and checking balance. A checking withdrawal should use the savings account if the balance in checking isn't sufficient. Similarly, make `withdraw:` should use the balance in checking if the savings account balance isn't sufficient.

(Hint: during your implementation of `PlusAccount`, you should not need to edit the implementation of `Account`. If you feel the need to do this, you aren't doing inheritance right.)

5. (extra credit, 15 points, programming and semantics) Methods such as `equalAsAccount:` are called binary methods. The Smalltalk method `=` is also a binary method, and is defined for both `Account` and `PlusAccount`. Describe how `=` differs from the methods `equalAsAccount:` and `equalAsPlusAccount:`. Is there any reason to prefer one over the other?

6. (extra credit, 10 points, programming and semantics) Define two classes: `AccountCompare` and `PlusAccountCompare`. The abstract value of an `AccountCompare` object is a pair of `Account` objects (which might actually be objects of type `PlusAccount`). The abstract value of an `PlusAccountCompare` object is a pair of `PlusAccount` objects (which again can be any two objects that obey the protocol of `PlusAccount`). The class `AccountCompare` has a method, `for:and:`, which takes two objects that satisfy the protocol for `Account`, and makes the abstract value of the receiver be these two objects. Similarly, `AccountCompare` has a method, `for:and:`, which takes two objects that satisfy the protocol for `PlusAccount`, and makes the abstract value of the receiver be these two objects. You can take the `equalAsAccount:` and `equalAsPlusAccount:` methods out of `Account` and `PlusAccount`. What should the definitions of the `=` method be for the classes `AccountCompare` and `PlusAccountCompare`? Briefly describe how this resembles multi-methods.

7. (50 points) Design and define a class `SavingsAndLoan`, to model all the accounts and plus-accounts in a savings-and-loan. Think of `SavingsAndLoan` a like a bank: it may contain many account and plus account objects. The methods should include `openAccount` and `openPlusAcount`, as well as `upgrade:` which converts a person's account from a plain `Account` to a `PlusAccount`. You should also have some method for getting a report that gives information on all the accounts (and `PlusAccounts`). (Note: you may make subclasses of `Account` and `PlusAccount` for this, but do not change their code.) You'll probably want to have additional methods also. I've intentionally left this problem vague so that you have freedom to design the class. There's no one right answer. You should include an overview of your design in what you turn in.

8. (extra credit, 10–30 points depending, applications track) Pick some thing or concept that is specific to some application area (e.g., databases or train robbery) and design a few Smalltalk classes that model these things or concepts.

9. (extra credit 15 points, semantics and applications—software engineering) The `PlusAccount` objects we designed above act like `Account` objects with the checking account dropped. For this problem you are to define a subclass of `Account`, call it `PlusAccount2`, whose objects act like an `Account` whose balance is the total of the savings and checking account balances in the `PlusAccount2` object. Hand in your code and an answer to the following question: which is a better `PlusAccount` or `PlusAccount2` and why?

10. (40 points) Read an article (or part of a book) that is related to OOP or OO programming languages (such as Smalltalk) from the "Introduction to the Literature" handout. (This doesn't include the books by Watt.) Write a short (1 or 2 page) review of the article, stating: what the problem was that the article was solving, the main points, what you learned from it, and what contribution it make vs. any related work mentioned in the article.