Homework 1: Introduction and 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.

Readings: Chapters 1 and 2 of the *Design Patterns* book by Gamma, *et al.* for background on OO design. TR #91-22a for reference material on Little Smalltalk. Books by Budd and by Goldberg and Robson on reserve for more background on Smalltalk and OO ideas. (See also the *Introduction to the Literature* handout for more readings.)

1. (10 points) Write (in Little Smalltalk) a class `Account`, which models an account with a bank or a savings and loan. To specify this problem, as in TR #91-22a, 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 returns 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 as the receiver.

2. (extra credit, 10 points) Implement `Account` in a significantly different way than you did in the above problem. (For example, if you used two instance variables before, use an array now, or vice versa). Call the new class `Account2`. Can every program work with both `Account` and `Account2` objects? Why or why not?

3. (extra credit, 10 points) Specify the class `Account` using the notation of TR #91-22a.

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 balance is the sum of the `PlusAccount` object's savings and checking balances. Deposits and withdrawals made using `deposit:` and `withdraw:` should be made to and from the savings account, but `withdraw:` should take as

much money from the checking account as needed if the savings account balance isn't sufficient. A `PlusAccount` should have extra operations `checkingDeposit:`, and `checkingWithdraw:`, `savingsBalance` and `checkingBalance`, and `equalAsPlusAccount:`. A checking withdrawal should use the savings account if the balance in checking 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. (20 points) This problem discusses binary methods.

   (a) When comparing `Account` and `PlusAccount` objects, the `=` method should compare the balances of the two accounts as if they were both `Account` objects; that is only the owner's name and the total balance should matter. Do you have to add any code to make this work?

   (b) Does the `=` method differ from the method `equalAsAccount:`? If so, how?

   (c) Add a message `equalAsPlusAccount:` that tests if the argument has the same owner's name, savings and checking balance.

   (d) Does the `equalAsAccount:` method differ from the `equalAsPlusAccount:` method? If so, how?

   (e) Is there any reason to have both an `equalAsAccount:` and an `equalAsPlusAccount:` method? Explain.

   (f) Suppose we added k different subtypes of `Account` to our program. How many methods for equality testing would we need to write?

6. (extra credit, 10 points) Write 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 a `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. Write code for these methods and `equal` methods that compares the objects contained in an `AccountCompare` or `PlusAccountCompare` objects. (You can take the `equalAsAccount:` and `equalAsPlusAccount:` methods out of `Account` and `PlusAccount`, or use them in your implementation.)

   Briefly describe how the code you wrote resembles generic functions in a multimethod language.

7. (50 points) Design and write a class `SavingsAndLoan`, to model all the accounts and plus-accounts in a savings-and-loan. Think of `SavingsAndLoan` as 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, 20 points) Pick some thing or concept that is specific to some application area (e.g., compilers, databases, or train robbery) and design a few Smalltalk classes that model these things or concepts.

9. (extra credit 15 points) The `PlusAccount` objects we designed above act like `Account` objects with a balance that is the sum of the checking and savings accounts. For this problem you

are to write a subclass of `Account`, call it `PlusAccount2`, whose objects act like `Account` objects with the checking account dropped. Hand in your code and an answer to the following question: which is a better `PlusAccount` or `PlusAccount2` and why?

10. Read a published research article that is related to OO programming languages (such as Smalltalk) from the "Introduction to the Literature" handout. By a published research article, I mean an article that is not in a trade journal (e.g., it has references at the end) and that is not on the web, but in the library. (It's okay to get a copy of a published article from the web, however.) An article from the OOPSLA or ECOOP conferences is acceptable. An article primarily about OO programming is not acceptable; the article should be about programming *languages*. I am happy to suggest articles for you, or evaluate your choices for suitability if you are in doubt.

    Write a short (1 or 2 page) review of the article, stating:

    - (5 points) what the problem was that the article was claiming to solve, (or if the article is a survey article, then the scope of the survey),
    - (10 points) the main points made in the article and what you learned from it,
    - (10 points) what contribution it make vs. any related work mentioned in the article (or if the article was a survey article, your evaluation of its completeness and clarity of thought and expression).