Homework 3: Introductory Smalltalk Programming

Due: problems 1-3 and 5 on October 2, 2003; 6-10 on October 9, 2003.

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 teams.

For all Smalltalk programs, you can run your code with either Squeak or VisualWorks Smalltalk, although we recommend using Squeak. See the "running smalltalk" web page off the course web page. Hand in a printout of the "file out" of your code and your testing. For classes you write, you can file out the entire class by selecting the Classes menu in the class browser, then choosing "file out ...". You can also do this for a whole category. For code you add to an existing class, just file out the new or changed methods (from the Methods menu in the browser, choose "file out ..."). This is also a good way to save your work.

For testing, save your work first! Then test your programs in a workspace; if your program has textual output, use "print it" from the Smalltalk menu and then save the workspace in a file (using the File menu, and choosing "Save As ...").

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

Unless we specifically direct you to edit an existing Smalltalk class, don't edit existing classes. Instead, make subclasses of existing classes.

Readings: First, see the course web page's "Running Smalltalk" section. Although we will primarily use Squeak in class, you should be able to use VisualWorks as well. Start VisualWorks and then use the brightly colored "Lesson Runner" window to learn some of the basics of Smalltalk. Do the first 4 lesson topics (Introduction, Fundamentals, Class Library 1, and Class Library 2). Then you can exit VisualWorks.

Second go to `http://www.squeak.org/` and look in their documentation section. From the first topic's link, which as a reference for the Squeak user interface by Andrew Greenberg (and Andrew Black), read the section "Using Squeak: the Basics" which tells you how the mouse, menus, and keys in Squeak work.

Now, with Squeak running, read carefully chapters 1-6 of Goldberg and Robson's book *Smalltalk-80: The Language*. Try things out using a workspace in Squeak. You should enter into Squeak the example classes, such as the one on pages 43-44, so you can do this. You can then use the rest of this book as a reference as you go along.

(See also the *Introduction to the Literature* handout for more readings.)

1. (5 points) This problem is about creating a very simple class and reading the notation for specifications. (See the appendix.) This problem also creates something that will be useful in later problems.

   Using Squeak, get a System Browser, and create a new category `ComS541-Homework`. Add to this category a new class named `HanoiDisk` with instance variable `size`. Add an instance method category `accessing` and methods `size` and `size:`, and another category `printing` with method `printOn:`. Also add a class method category `instance creation` with the class method `ofSize:`.

   The following is the specification, in the notation described in the appendix of this homework, of these operations.

   size: *self: HanoiDisk, anInteger: Integer*
   Modifiable: the size of *self*.
   Effect: make the size of **post**(*self*) be *anInteger*.

   size *self: HanoiDisk* $\rightarrow$ *i: Integer*
   Ensures: *i* is the size of *self*.

   printOn: *self: HanoiDisk, aStream: Stream*

Requires: *aStream* is open for writing.
Modifiable: the output of *aStream*.
Effect: prints information about *self* on *aStream*.

ofSize: *self: HanoiDisk class*, *anInteger: Integer* → *hd: HanoiDisk*
Ensures: *hd* is a new HanoiDisk, and the size of *hd* is *anInteger*.

File out your code for this class, and then test it in a workspace (from the Tools menu), by typing in expressions and using "print it" on them. When your class works, file it out again, and then print it and the contents of your workspace. Hand in this printout, clearly labeled with your name and this homework's problem number at the top.

2. (15 points) Smalltalk does not have "structures" or "records", so sometimes one has to implement such a class with methods that simply set or get the values of instance variables. Such designs don't really distribute the responsibility very well, but they are often needed.

   Design an implement a class `HanoiMove`, which should also be in the category `ComS541-Homework`. This class is responsible for knowing 3 things: a `HanoiDisk` object, and two `Integer`s, which identify the tower the move originates from and the tower to which the disk is sent. In the "towers of Hanoi" there are three towers, which we'll identify by the numbers 1, 2, and 3.

   In your design try to make use of objects of this class convenient for the callers.

   File out your code for this class, and then test it in a workspace (from the Tools menu), by typing in expressions and using "print it" on them. When your class works, file it out again, and then print it and the contents of your workspace. Hand in this printout, clearly labeled with your name and this homework's problem number at the top.

3. (10 points) A more interesting data type is one that contains an unknown number of objects. In this problem you will implement a very simple kind of stack.

   In this same category, implement the class `HanoiTower` according to the following specification.

identity: *self: HanoiTower*, *anInteger: Integer*
Modifiable: the identity of *self*.
Effect: make the identity of **post**(*self*) be *anInteger*.

identity *self: HanoiTower* → *i: Integer*
Ensures: *i* is the identity of *self*.

push: *self: HanoiTower*, *aDisk: HanoiDisk*
Modifiable: the disks of *self*.
Effect: make *aDisk* be the topmost disk on **post**(*self*), leaving the existing disks of **pre**(*self*) alone.

pop *self: HanoiTower* → *top: HanoiDisk*
Requires: the size of *self* is not 0.
Modifiable: the disks of *self*.
Ensures: the disks of **post**(*self*) are all but the topmost disk of **pre**(*self*), and the result is the topmost disk of **pre**(*self*).

size *self: HanoiTower* → *i: Integer*
Ensures: *i* is the size of *self*.

Hint: you can add other methods to this class if you need or desire them. You may want to use an `OrderedCollection` in your implementation, but make `HanoiTower` a subclass of `Object`. (That is, represent these objects using composition instead of inheritance.)

File out your code for this class, and then test it in a workspace (from the Tools menu), by typing in expressions and using "print it" on them. When your class works, file it out again, and then print it and the contents of your workspace. Hand in this printout, clearly labeled with your name and this homework's problem number at the top.

4. (15 points, extra credit) Use the "SUnit" unit testing tool found in the `SUnit` categories and write a test class for testing the class `HanoiTower`. Call your test class `HanoiTowerTest`. It should be a subclass of `TestCase` (in the category `SUnit-Core`). To execute the tests evaluate the expression `TestRunner runTests`.

   See also `http://ANSI-ST-tests.sourceforge.net/SUnit.html`.

5. (a) (15 points) Read Chapter 19 of *Smalltalk-80: The Language*, where the class `Pen` is described. (See also chapters 31 and 32 of *On to Smalltalk*.)

   In the category `ComS541-Homework`, make a class `NGon`, which will be a subclass of `Pen`.

   Add a method `sides:length:` to the class `NGon`. This method should take two integer arguments, `n`, which is the number of sides and thus strictly greater than 2, and `len`, which is the number of pixels on each side and thus positive. It draws an `n`-gon each of whose sides is `len` pixels, that is to the right and below the pen's current point.

   Also add methods in a category `examples` to show how your method works, and use these in your testing. For example, you might have the following example.

```
drawSquare
    "NGon new drawSquare"
    Display restoreAfter:[
       | np |
       np := self class new.
       np color: Color blue.
       np home.
       np sides: 4 length: 50.
    ]
```

   should draw a square starting at the center of the screen, with sides 50 pixels long (in blue). When you have finished looking at it click the mouse (several times) to make the drawing go away. (Your testing should do more than just the above.)

   Hint: if you don't see anything, be sure you have the directions right, you may be drawing off the screen.

   The following example works in a morhpic project. To get rid of the resulting morph, hold down the "alt" key while you left click on the morph. Then click on the "x" in the circle at the top left in the "halo" of buttons that appear.

```
drawMorphicHeptagon
    "NGon new drawMorphicHeptagon"
    | f m np len |
    f := Form fromUser.
    f fillColor: Color white.
    len := f extent x min: f extent y.
    len := len / 7.
    m := SketchMorph withForm: f.
    np := NGon newOnForm: f.
    np color: Color blue.
    np home; up; turn: -180; go: len/2.
    np turn: 90; go: len; north.
    np sides: 7 length: (f extent x / 7).
    m revealPenStrokes.
    m openInWorld.
```

   (b) (10 points) Add a example method `drawNested:` to the class `NGon`, with the following specification.

   drawNested: *self: NGon*, *n: Integer*
   Requires: *n* is strictly greater than 2.

Modifiable: the identity of *self*.
Effect: draw n-gons with sizes from 3 to n, all of which share one side.

File out your code for this class, and then test it in a workspace (from the Tools menu), by typing in expressions and using "print it" on them. When your class works, file it out again, and then print it and the contents of your workspace. Hand in this printout, clearly labeled with your name and this homework's problem number at the top.

6. (40 points) An interesting part of Smalltalk is the `Stream` classes hierarchy. Read about `Stream` in chapter 12 of *Smalltalk-80: The Language*.

   In this problem you will define a class `HanoiMoveStream` as a subclass of `Stream`. This class acts like a stream of moves for solving the towers of Hanoi problem. This problem concerns 3 pins, on which disks can be placed one at a time. The disks all have different sizes, and the disks must always be placed so that each disk sits upon disks of a larger size. The disks are initially arranged on one of the three pins, forming the original tower, called tower 1. A move consists of taking one disk from one pin, and moving it to another pin; of course this is only permitted if the disk is smaller than the topmost disk on the other pin. The task is to use such moves to move all the disks from the tower on pin 1 to form a tower on pin 3. Of course, one uses tower 2 in the process.

   Your problem is to implement the following methods, and any others that are necessary to define a subclass of `Stream`.

   next *self: HanoiMoveStream* $\rightarrow$ *m: HanoiMove*
   Requires: *self* is not at its end.
   Modifiable: the current position of *self*.
   Ensures: m is the next move in the solution to the Towers of Hanoi problem.

   ofHeight: *self: HanoiMoveStream*, *anInteger: Integer*
   Requires: *anInteger* is strictly greater than 0.
   Modifiable: the height of the tower of disks of *self*.
   Effect: the position of **post**(*self*) is not at its end, and the next move of **post**(`self`) is the first move in the Towers of Hanoi solution for an initial tower of height *anInteger*.

   Hints: You will need several other methods, including `printOn:` for testing. You may find it helpful to generate all the moves by the time the first call to the `next` method completes. Use recursion.

   File out your code for this class, and then test it in a workspace (from the Tools menu), by typing in expressions and using "print it" on them. When your class works, file it out again, and then print it and the contents of your workspace. Hand in this printout, clearly labeled with your name and this homework's problem number at the top.

7. (40 points) One of the most interesting parts of Smalltalk is the collection class hierarchy. See chapters 9-11 and 13 of *Smalltalk-80: The Language*.

   In this problem you will implement a subclass of `Collection`, `BinaryRelation`, which models binary relations. Abstractly, a `BinaryRelation` object is modeled as containing a set of pairs of objects. For example, a binary relation that contains the set $\{(a, b), (c, d), (e, f)\}$ relates the object a to b, c to d, and e to f. Note that this is a relation on object identities; the values of the objects don't matter.

   You are to define the class `BinaryRelation` as a subclass of `Collection`. (It can be a subclass of some subclass of `Collection`, but must inherit from `Collection` to receive credit.)

   The class methods should include `new`, which returns a binary relation with an empty set. However, you don't need to override `new` if what you inherit works correctly.

   Instance methods should include the following, which are specified in the notation described in the appendix.

relate:to: *self: BinaryRelation*, *key: Object*, *value: Object*

Modifiable: the set of *self*.

Effect: make the set of **post**(*self*) be the union of the set of **pre**(*self*) with the set $\{(key, value)\}$.

relates:to: *self: BinaryRelation*, *key: Object*, *value: Object* → *b: Boolean*

Ensures: *b* is true just when the set of *self* contains the pair (*key*, *value*).

do: *self: BinaryRelation*, *aBlock: Block*

Requires: *aBlock* takes one argument, which has type `Association`, and does not modify the instance variables of *self*.

Modifiable: the locations that are modifiable by the execution of *aBlock*.

Effect: evaluate *aBlock* for each `Association` object that represents a pair, (*key*, *value*), such that *self* relates *key* to *value*.

at: *self: BinaryRelation*, *key: Object* → *s: Set*

Ensures: *s* is a new `Set` object whose abstract value is the set of objects to which *key* is related in *self*; that is, **post**$(s) = \{v \mid (key, v) \in$ the set of $self\}$.

Be sure that your methods for `do:` and `at:` do not expose `BinaryRelation`'s representation (that is, do not allow clients to have pointers into the representation). Also be sure that printing works as desired for binary relations, so that you can do adequate testing; you may have to redefine `printOn:` to make this work as needed.

Note: the `=` method in class `Association` only compares the keys, and ignores values. You may need to make a subclass of `Association`, or to create an `Association` to pass to the block passed to the `do:` method to avoid difficulties with this problem.

8. (30 points) Add a subclass `ReflexiveRelation` to `BinaryRelation`. `ReflexiveRelation`'s instances relate each object in their domain to itself by the `relates:to:` method. One way to think of this is that the `relate:to:` method satisfies the following specification.

   relate:to: *self: BinaryRelation*, *key: Object*, *value: Object*

   Modifiable: *self*.

   Effect: make the set of **post**(*self*) be the union of the set of **pre**(*self*) and the set $\{(key, value), (key, key), (value, value)\}$.

   This does not mean that you need to store the reflexive pairs (*key*, *key*) and (*value*, *value*) in the instance variables of `ReflexiveRelation`. Indeed, to make the problem more interesting, and to make the implementation more space efficient, we will prohibit you from doing that.

   So, your problem is to figure out a suitable representation, and override the appropriate methods to make `ReflexiveRelation` work as desired.

9. This problem discusses binary methods. (See the paper "On Binary Methods" [1] for more details.)

   (a) (5 points) What is the difference between the `==` and the `=` methods in Smalltalk?

   (b) (5 points) Implement an `=` method for the class `BinaryRelation`. This method should follow the general pattern for `=` methods. If you don't need to add any code to your class to have `=` work correctly say so instead of writing the code.

   (c) (5 points) Does your `=` method for binary relations work equally well when comparing arguments that may be reflexive relations? Why or why not?

   (d) (5 points) Suppose the type `Graph` were added as a subclass of `BinaryRelation`. Let's imagine that a `Graph`, besides storing the points of a relation, also stores a `SketchMorph` window. Does the `=` method inherited from `BinaryRelation` do the right thing for `Graph` arguments? Does it do the right thing for mixed `Graph` and `BinaryRelation` arguments?

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 from a refereed journal or conference. *Publication* means the article actually appeared in print, and was not just submitted somewhere. So beware of technical reports on the web. (It's okay to get a copy of a published article from the web, although I highly encourage you to physically go to the library.) An article from the *OOPSLA* or *ECOOP* conferences is fine. An article primarily about OO programming is not acceptable; the article should be about programming *languages*; one way to tell is if it introduces new syntax and features into a language, or if it introduces a new language or a new way to check a language. Do not get an article that is a survey article, or one that deals with primarily with language implementation (compilers, virtual machines, optimizations, etc.). Articles on security issues are fine if they are tied in with type checking or other language design issues. 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, in your own words:

    - (10 points) What the problem was that the article was claiming to solve?
    - (20 points) What were the main points made in the article, and what did learn from it?
    - (20 points) What contribution it make vs. any related work mentioned in the article?

    In your writing, be sure to digest the material; that is, don't just select various quotes from the article and string them together, instead, really summarize it. If you quote any text from the paper, be sure to mark the quotations with quotation marks (" and ") and give the page number(s).

    Hand in a copy of the article with your review.

Appendix

This appendix describes the structured English specification notation, which is slightly modified from ISU Com S TR #91-22a [3].

"Specifications of methods are given informally, but in a stylized manner, following Liskov and Guttag's book [5]. A specification consists of a header, a requires clause, and an ensures clause. The *header* describes the signature of the operation and gives formal names to the arguments and the result. These names are used in the requires and ensures clauses. The *requires* clause states a pre-condition on the operation's arguments; that is, a property of the arguments that the caller is expected to satisfy. The requires clause is omitted if no requirements are put on the caller. The *ensures* clause states a post-condition on the results, described in terms of the arguments and the formal result identifier. The post-condition is a property of the result that the method establishes.

"Consider the following simple example, the at: operation on Arrays.

at: *self: Array*, *i: Integer* → *o: Object*

Requires: i is greater than 0, and less than or equal to the size of *self*.

Ensures: o is the i^{th} element of *self*.

The header says that at: is a message that can be sent to an instance of the class Array or a subtype[1] of Array. The receiver, the object to which the message is sent, is given the name *self*. The at: message also needs an additional argument, which must be an Integer (or a subtype); this additional argument is denoted by i. The object returned by the at: method must be an Object (or one of its subtypes, which is not restrictive), and this object is represented by o. The requires clause says that, the value of i must be a legal index into the array *self*. If this condition is true, the operation

[1]A "subtype of Array" is a type whose instances behave like arrays in the sense that each instance of the subtype simulates some array. At the least, an object q simulates r if a sequence of message sends cumulating in a boolean or integer result would yield the same final result for both q and r. See [4] [2] [6] for more details.

behaves as described in the ensures clause. Otherwise, the operation does not need to behave as specified in the ensures clause, and may instead do something else, such as printing an error message. The ensures clause says that the value returned is the i^{th} element of the receiver (*self*). This is a termination semantics, that is to say, the operation must not loop forever or encounter an error when the requires clause is satisfied. If the operation does not return, those cases will be made explicitly described.

"A slightly different format is used for operations that mutate their arguments and for those that have no return value. For operations that mutate their arguments, a *modifiable clause* is included to state [the parts of] objects [that] are allowed to be modified. Many operation specifications omit this clause, which means no objects are mutated by that operation. For some operations, especially those that mutate their arguments, the return value is irrelevant. Therefore for such operations, the arrow ($\rightarrow$) and the information about the result is omitted from the header; this is interpreted as follows: if the requires clause is satisfied, the receiving object itself will be returned. For such operations the post-condition is stated in an *effect clause* instead of an ensures clause. Logically the meaning is the same, however such a post-condition only states side effects, because there is no return object to refer to.

"In the specification of an operation that mutates its arguments, it is sometimes necessary to refer the value of [part of] an object in two different states; the states before and after the call. It is also necessary to refer the identity of the object (i.e., its address), that is to say, the object itself not its value. These distinctions are made by qualifying formal argument names. Consider, for example, an object named *self*. The value of this object before the call is denoted by **pre**(*self*) while the value after the call is represented by **post**(*self*). The object's identity is denoted by **obj**(*self*). Qualifications are often redundant. This leads us to adopt certain defaults depending on the context in which a name appears. In the modifies clause, one always refers to object identities, so the object qualification is the default. An unqualified formal argument name *arg* is, by default, qualified as **pre**(*arg*). A formal result name *res* is, by default, qualified as **post**(*res*). For example, consider the following description of the at:put: operation for arrays, which stores an object at a given index in the receiver.

at:put: *self: Array*, *i: Integer*, *o: Object*

Requires: i is greater than 0, and less than or equal to the size of *self*.

Modifiable: the $i^t h$ element of *self*.

Effect: makes **obj**(o) the $i^t h$ element of **post**(*self*).

The input formal i appearing in the requires and effect clauses, is a short form of **pre**(i). The receiver *self* in the modifies clause is short for **obj**(*self*).

"The major conceptual difference between this notation and Liskov and Guttag's is that we omit from a type's specification specifications for operations defined in the type's supertypes. For instance, since all types are subtypes of the type Object, all have a method class that returns the class of the receiving object. However this method is not repeatedly specified in each type, since repeating it would be redundant and uninformative. It is only specified in the type Object once and for all. Therefore one can think of subtyping as inheritance of *specifications* (as opposed to code inheritance, which is subclassing in Smalltalk).

"A set of method specifications are collected together to give a specification of an abstract data type. An abstract data type, which will be called a *type* for short, is an abstraction of a set of Smalltalk classes characterized by their behavior. A type is usually implemented by a single Smalltalk class. However it may be implemented by several classes, as in the case of type Boolean which is implemented by three Smalltalk classes, Boolean, True, and False."

References

[1] K. Bruce, L. Cardelli, G. Castagna, T. H. O. Group, G. T. Leavens, and B. Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):221–242, 1995.

[2] G. T. Leavens. Modular specification and verification of object-oriented programs. *IEEE Software*, 8(4):72–80, July 1991.

[3] G. T. Leavens and Y. Cheon. Overview and specification of the built-in types in Little Smalltalk. Technical Report 91-22a, Iowa State University, Department of Computer Science, 226 Atanasoff Hall, Ames IA 50011, Feb. 1994. Available by anonymous ftp from ftp.cs.iastate.edu, and by e-mail from almanac@cs.iastate.edu.

[4] G. T. Leavens and W. E. Weihl. Reasoning about object-oriented programs that use subtypes (extended abstract). In N. Meyrowitz, editor, *OOPSLA ECOOP '90 Proceedings*, volume 25(10) of *ACM SIGPLAN Notices*, pages 212–223. ACM, Oct. 1990.

[5] B. Liskov and J. Guttag. *Abstraction and Specification in Program Development.* The MIT Press, Cambridge, Mass., 1986.

[6] B. Liskov and J. Wing. A behavioral notion of subtyping. *ACM Trans. Prog. Lang. Syst.*, 16(6):1811–1841, Nov. 1994.