Homework 3: Introductory Smalltalk Programming

Due: problems 1–2, Thursday Nov 4; problems 5–6, Thursday Nov 11; problems 7–9, Thursday Nov 18.

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 Smalltalk Express (i.e., Smalltalk/V). 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 ...". For code you add to an existing class, just file out the new or changed methods (from the Methods menu in the class browser, choose "File Out ..."). This is also a good way to save your work.

For testing, test your programs in a workspace; if your program has textual output, use "Show 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: Chapters 2-3, 5, 6-7, and 9 of *Discovering Smalltalk* by Wilf LaLonde (Benjamin-Cummings, 1994). Alternatively, you can skim chapters 2-10 of *On to Smalltalk* by Patrick Winston (Addison-Wesley, 1998), and then read more carefully chapters 11-13, 16-18, 20-21, and 27. For background on OO ideas and OO design, see chapter 8 of *Discovering Smalltalk*, chapters 10 and 14-15 of *On to Smalltalk*. More comprehensive background is found in chapters 1 and 2 of the *Design Patterns* book by Gamma, *et al.* (See also the *Introduction to the Literature* handout for more readings.)

1. (15 points) In *Discovering Smalltalk*, section 3.9, the classes Point and Pen are described; these are also described in chapters 31 and 32 of *On to Smalltalk*.

   Add a method `ngon:withSide:` to the class `Pen`. 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. For example

   ```
   | myPen |
   myPen := (GraphPane openWindow: 'Testing') pen.
   myPen up; go: (200@200); down.
   myPen ngon: 3 withSide: 50.
   ```

   should draw an equilateral triangle at the point 200@200. (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.

2. (30 points) Make a subclass `TestRandom` of the class `Random` in the following file.

   `http://www.cs.iastate.edu/~leavens/ComS541/homework/lottery/random.cls`

   The subclass should support an instance method `visualize` that graphs (using `GraphPanes` and `Pens`) the function $f(i)$ = "the result of the ith call to the `next` method" (for some fixed instance of `TestRandom`). That is, draw the x and y axes, label them, and draw a dot at the graph coordinate $(i, f(i))$. Graph points with $i = 1$ to $i = 400$.

3. (10 points extra credit) From the picture you draw, can you tell how good the random number generator is? If it's bad can you fix it?

4. (30 points extra credit) Make a class to draw graphs of functions. Demonstrate its use on various problems.

5. (40 points) In this problem you will define a class `BinaryRelation` which models binary relations. The abstract value of a `BinaryRelation` object is a set of pairs of objects. For example, a binary relation with abstract value $\{(a,b),(c,d),(e,f)\}$ 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 an empty binary relation. However, you don't need to override **`new`** if what you inherit works correctly.

   Instance methods should include the following. These are specified using structured English, as described in ISU Com S TR #91-22a [3]. An excerpt from this appears in the appendix at the end of this homework.

   The following is the specification, in the notation described in the appendix, of some of the operations of `BinaryRelation` that you should include.

   relate:to: *self: BinaryRelation, key: Object, value: Object*
   Modifies at most: *self.*
   Effect: make **post**(*self*) be the union of **pre**(*self*) with $\{(key, value)\}$.

   relates:to: *self: BinaryRelation, key: Object, value: Object* $\rightarrow$ *b: Boolean*
   Ensures: b is true just when *self* contains the pair $(key, value)$.

   do: *self: BinaryRelation, aBlock: Block*
   Requires: *aBlock* takes one argument, which has type `Association`, and does not modify *self.*
   Modifies at most: the objects modified 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* $\rightarrow$ *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, $\mathbf{post}(s) = \{v \mid (key, v) \in self\}$.

   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.

6. (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*
   Modifies at most: *self.*
   Effect: make **post**(*self*) be the union of **pre**(*self*) with $\{(key, value), (key, key), (value, value), \}$.

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. Note that the `relates:to:` method will not necessarily return true if $key == value$.

7. This problem discusses binary methods. (See *On to Smalltalk* chapter 27 and 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 `GraphPane` 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?

8. (100 points) Write a Smalltalk program to do something interesting worth 100 points. This program should have at least 3 classes. For example, you could do something related to something in your creative component, or write a simple game, or animation system. The program should not be directly related to one of the problems above. It's best if what you do reuses some of the built in classes in Smalltalk/V to do something interesting, by adding new classes to the system.

   Don't change any existing code in the system, unless you need to fix a bug. Instead add subclasses that override methods you want to change. Besides being good OO coding practice, this prevents you from crashing the system by accidentally modifying some crucial part.

   If you can't think of ideas, I have some.

9. 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 also 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. I'd prefer that you did not get an article that is a survey article, or one that deals with language implementation primarily (compilers, optimizations, etc.) 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:

   - (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 learned 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 passages from the paper, be sure to mark the quotations and give the page number(s).

Appendix

This appendix describes the structured English specification notation described in ISU Com S TR #91-22a [3]. The following is an excerpt from that document.

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* $\rightarrow$ *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 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 *modifies clause* is included to state which objects are allowed to be modified. Most 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 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.

[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.

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

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

Modifies at most: *self*.

Effect: makes o the element of **post**(*self*) at index i.

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