A Smalltalk/V Example

Gary T. Leavens
Department of Computer Science, 229 Atanasoff Hall
Iowa State University, Ames, Iowa 50011-1040 USA
leavens@cs.iastate.edu

November 15, 1999

Abstract

An example of a Smalltalk/V program, and how to run it.

1 The Problem

Ada and Haskell Bliss were computer programmers, and in love. At least at first. They spent the early days of their marriage designing Smalltalk programs together, and becoming wealthy. However, as time went on, Ada discovered that Haskell was becoming addicted to gambling. Soon they were forced to move from state to state, chased by their creditors. It was ruining their marriage.

Try as she might to argue with Haskell, Ada couldn't convince him to see that the lotteries in each state, although different, all held the same bleak picture. Arguing the probabilities with Haskell didn't seem to make any difference. Somehow, he couldn't see the big picture. Then one day, Ada hit on the idea of writing a Smalltalk program that could simulate the various state lotteries, and show Haskell what he could except. Since Haskell would understand this program, and could adjust it to various situations, she hopes that it will help cure him.

What would that program look like?

Because of the variations between states, and over time, the program should be able have various parameters that may be varied in different runs of the simulation.

- The range of numbers the players pick from (e.g., 1 to 42).
- How many numbers a player picks for each ticket (e.g., seven).
- Whether or not the player can choose the same number more than once on a ticket.
- How much each ticket costs (e.g., one dollar).
- A function that gives the payoffs for ticket, according to how many numbers on the ticket match the state's winning numbers.
- The probability distribution used by players to choose their numbers. (It seems clear that the state should use a uniform distribution, but players might tend to favor certain numbers, such as 7 and 11.)

The main thing in the program is to simulate a customer or customers buying tickets and playing the lottery. Statistics of interest are at first, the net gain or loss by each customer.

2 The Design

Ada began by making some simplifying assumptions; that is, she decided that, instead of doing the whole thing at once, she would evolve the program from simpler versions of it. The first simplification was that players choose numbers uniformly. Second, she assumed that all numbers chosen for a ticket must be distinct.

Actions in the program should therefore allow the user to define the range of numbers, define the payoff table, create customers, tell them to place a number of bets, and report their losses (or winnings).

2.1 What are the Objects?

The concepts and things in this problem are lottery tickets, the winning numbers, choosing distinct numbers from a uniform distribution, money, the lottery agency (which chooses the range of numbers, and the payoff table), and the customer (who picks the numbers on tickets).

It seems useful to represent the winning numbers as a special kind of ticket. Distinguishing the winning number selection from the customer selection allows customers to pick from a skewed distribution (eventually). Making the discrete uniform probability distribution an object will help keep down the size of the code and ensure that there is only one random number generator in use. On the other hand, having a separate class for money seems like overkill, so we just use numbers to represent money.

The class `Ticket` will have as its abstract values, a set of distinct integers. It provides the operations of the class `Collection` and an initialization operation.

The class `WinningTicket` will be a subtype of `Ticket` and have an extra operation that compares a ticket and returns the number of matches.

It is possible to specify probability distributions in a very general framework [1, Chapter 21]. For the sake of brevity we settle for something simplier: a class `UniformDiscrete`, whose abstract value is a stream of random integers from 1 to a given size. We will have an operation to return a random integer from the stream and a special operation to return a set of n such numbers without duplicates.

The class `Lottery` has as its abstract value, the range of numbers that can be chosen on tickets, how many numbers are chosen per ticket, a payoff table, and a set of winning numbers (i.e., a `WinningTicket`). It has operations that allow the winnings of a ticket to be calculated.

Finally, the class `LottoCustomer` will have as its abstract value a bag of tickets, and money representing its loss or gain. It will have an operation to report its winnings (or loss).

The implementations of these classes follows.

3 Implementation

The following code is written for Smalltalk/V [3] (also known as Smalltalk Express). The code shown is directly taken from the files that create the system; hence it is in the Smalltalk "file out" format. This format consists of commands terminated by exclamation points (`!`) with two consecutive exclamation points marking the ends of sections of methods.

3.1 Ticket

The class `Ticket` is presented in Figure 1. It is a subclass of `Collection`. Instances contain a `Set` of integers, stored in the instance variable `rep`. The class method `new:` takes a set and creates a

```
Collection subclass: #Ticket
  instanceVariableNames: 'rep '
  classVariableNames: '' poolDictionaries: '' !

!Ticket class methods !

new
    ^ self new: (Set new)!

new: aSet
    "ENSURES: result is a new Ticket with the elements of aSet
                        as its set of numbers"
     ^super new initialize: aSet! !

!Ticket methods !

copy
    "ENSURES: result is a ticket with the same abstract value as self"
    ^ self class new: rep!

do: aBlock
    ^ rep do: aBlock!

initialize: aSet
    "EFFECT: this ticket's abstract value becomes the same as aSet"
    rep := Set new.
    rep addAll: aSet.! !
```

Figure 1: The Smalltalk/V class `Ticket`.

new instance and initializes it with the set. The class method `new` is redefined to ensure that the instance variable `rep` is never nil.

In the instance methods, note the method `initialize:` extracts the elements of the argument set so that the value of the instance variable is not pointing to an object that is outside the control of this class. (Not doing this would be an instance of the problem of "exposing the representation".)

3.2 Winning Ticket

The class `WinningTicket` is presented in Figure 2. It is a subclass of `Ticket`. It adds to its superclass the method `matches`, which allows other tickets to be compared with the set of winning numbers.

The method `matches` is a time sink, since it takes n^2 comparisons to determine the matches of each ticket. Faster algorithms are left to the reader.

3.3 Probability Distribution

The class `ProbabilityDistribution` is presented in Figure 3. It is a place-holder for future expansion of probability distribution classes. Although this class isn't needed directly in this

```
Ticket subclass: #WinningTicket
  instanceVariableNames: ''
  classVariableNames: '' poolDictionaries: ''    !

!WinningTicket methods !

matches: aTicket
        "ENSURES: result is the number of matches between self and aTicket"
    ^ aTicket inject: 0 into: [ :n :elem | n + (self occurrencesOf: elem) ]! !
```

Figure 2: The class `WinningTicket`.

```
Object subclass: #ProbabilityDistribution
  instanceVariableNames: ''
  classVariableNames: '' poolDictionaries: ''  !

!ProbabilityDistribution methods !

next
    ^ self implementedBySubclass! !
```

Figure 3: The class `ProbabilityDistribution`.

program, it makes sense to add it because we already imagine having other probability distributions, and because it is painful to change the superclass of a class in Smalltalk.

(See Chapter 21 of Goldberg and Robson's book [1] for a comprehensive treatment of probability distributions in Smalltalk.)

3.4 Uniform Discrete

The class `UniformDiscrete` is presented in Figure 4. It is a subclass of `ProbabilityDistribution`. It has an class variable `rand`, which is an instance of class `Random`. A class variable is used so that there is only one source of random numbers for the simulation. It also has instance variables `range` and size, the first of which which determine the interval it selects from.

3.5 Random

There is no class `Random` in Smalltalk/V. So I put together a version of it for myself as shown in Figure 5. This uses the linear congruential method, as described by Knuth [2, Section 3.2].

3.6 Lottery

The class `Lottery` is presented in Figure 6. It is a subclass of `Object`. One might consider making it a subclass of some of the other classes above, but it seems better to make it contain instances of these other classes, as there is no clear behavioral continuity.

Instances of the class `Lottery` have the following instance variables:

- `winningTicket`, which is a `WinningTicket` that holds the winning numbers,

```
ProbabilityDistribution subclass: #UniformDiscrete
  instanceVariableNames: 'range size '
  classVariableNames: 'Rand' poolDictionaries: ''  !

!UniformDiscrete class methods !

new: anInterval
    (Rand isNil) ifTrue: [Rand := Random new].
    ^ super new initialize: anInterval! !

!UniformDiscrete methods !

initialize: anInterval
    range := anInterval.
    size := anInterval size!

next
    "ENSURES: result is a random integer in the interval"
    ^ (range first + ((Rand next) * size) \\ size) asInteger!

pick: anInteger
    "REQUIRES: anInteger <= size
     ENSURES: result is a new set that contains anInteger
              distinct random integers"
    | s |
    s := Set new.
    [ (s size) < anInteger ]
        whileTrue: [ s add: self next ].
    ^s! !
```

Figure 4: The class `UniformDiscrete`.

```
Object subclass: #Random
  instanceVariableNames: 'seed '
  classVariableNames: 'M C A ' poolDictionaries: ''    !

!Random class methods !

new
    (A isNil)
        ifTrue: [ M := 2 raisedToInteger: 35.
                  C := 541. "should be relatively prime to M"
                  A := 315159621].
    ^ super new initialize! !

!Random methods !

initialize
    self seed: 0.5714159!

next
    "ENSURES: result is a random floating point number between 0 and 1"
    seed := (A * seed + C) rem: M.
    ^ (seed /  M) asFloat!

randInteger: size
    self error: 'this message deleted, use Uniform Discrete instead'!

seed: anInteger
    seed := (1 max: anInteger).! !
```

Figure 5: The class `Random`.

```
Object subclass: #Lottery
  instanceVariableNames: 'numsPerTicket winningTicket prizes range probDist '
  classVariableNames: ''  poolDictionaries: ''    !

!Lottery class methods !

picking: aNat from: anInterval paying: anArray
    ^ super new picking: aNat from: anInterval paying: anArray!

example
    ^ self picking: 7 from: (1 to: 42)
           paying: #(0.0 0.0 0.0 1.0 10.0 1000.0 10000000.0)! !

!Lottery methods !

picking: aNat from: anInterval paying: anArray
    (anArray size = aNat)
        ifFalse: [ self error: 'bad array size' ]
        ifTrue: [ numsPerTicket := aNat.
                  prizes := anArray copy.
                  range := anInterval.
                  probDist := UniformDiscrete new: anInterval.
                  winningTicket := WinningTicket new ]!

numsPerTicket
    ^ numsPerTicket!

range
    ^ range!
payoffs
    ^ prizes copy!
ticketCost
    ^ 1.0!
winningTicket
    ^ winningTicket copy!

pickWinner
    winningTicket initialize: (probDist pick: numsPerTicket)!

winnings: aTicket
    "EFFECT: return the amount of money won by aTicket"
    | m |
    m := winningTicket matches: aTicket.
    (m = 0) ifTrue: [ ^0.0 ] ifFalse: [^prizes at: m]! !
```

Figure 6: The class `Lottery`.

- prizes, which is an array of numbers giving the payoffs (in dollars) for each number of matches,
- range, which is an interval giving the range of numbers from which customers can choose,
- and probDist, which an instance of UniformDiscrete, used to pick the winning numbers.

The class method picking:from:paying: method of class Lottery creates a new lottery object and sends it a message to initialize its instance variables. The winning ticket, although created, is not yet initializd. That is handled by the method pickWinner. Note that in the instance method picking:from:paying:, the array is copied, so that it is not shared with the caller (which would cause "rep exposure" otherwise). However, the interval is not copied, because it is immutable.

The class method ticketCost gives the cost of a ticket.

The methods range and winningTicket are typical of methods that give clients access to instance variables of a class. Note how the array, as it is a mutable object, is copied before being released to callers of payoffs. The method ticketCost gives the cost of each ticket; for the moment this is a constant.

In the final two methods of Figure 6, the method pickWinner picks a winning ticket. The method winnings: tells how much money was won by the given ticket.

3.7 Lotto Customer

The class LottoCustomer is presented in Figure 7. It is also a subclass of Object. It has instance variables tickets (a bag of tickets), money (how much money the customer has), won (how much money has been won), loss (how much has been lost), and lottery (a Lottery instance). Having a Lottery instance contained in a customer allows several customers to share the same lottery.

3.8 Tracing Lotto Customer

The class TracingLottoCustomer, is a subclass of LottoCustomer, and is presented in Figure 8. It is an example of an object-oriented "trick:" writing a subclass for debugging instead of modifying the code of the original class.

TracingLottoCustomer is just like LottoCustomer, except that its playTicket: method sends debugging information to the Transcript window. By using super, the method playTicket: it is able to reuse the code of the superclass LottoCustomer without change; hence there is no question that when the debugging is finished, the method of class LottoCustomer will work as expected.

4 Running the Program

It is convenient when building a program composed of several classes, to have a script that will read in the class definitions from files. I put this script in the file loadit.st, as shown in Figure 9. One can open file this in or open the file in a file browser and execute 'do it' to read in all the files.

To run the simulation, one first creates an instance of the class Lottery and initializes the range and payoffs. One then creates one or more customers. The customer can then be asked to play the lottery and report winnings. Again, I put this in a script file, such as example.st shown in Figure 10. You might enjoy developing a script that prompts you for the various parameters.

In summary, to run the system, select first the code in Figure 9, select 'do it' from the Smalltalk menu, then select the code in Figure 10, and select 'do it' from the Smalltalk menu. You should see lots of information appearing on the Transcript window from the trace, followed by the final message.

```
Object subclass: #LottoCustomer
  instanceVariableNames: 'tickets money won cost lottery'
  classVariableNames: '' poolDictionaries: ''    !

!LottoCustomer class methods !

new: aNumber
    ^ super new initialize: aNumber!

example
    ^ (self new: 5000.0) lottery: (Lottery example)! !

!LottoCustomer methods !

initialize: aNumber
    tickets := Bag new.  money := aNumber.  won := 0.0.  cost := 0.0.!

tickets
    ^Bag new addAll: tickets!

winnings
    ^won - cost!

lottery: aLottery
    lottery := aLottery!

lottery
    ^ lottery!

balance
    ^ money - cost + won!

play: numTimes
    | ud t |
    ud := UniformDiscrete new: lottery range.
    numTimes timesRepeat:
          [ (self balance < lottery ticketCost)
                ifTrue: [^self].
            self playTicket:
                ((Ticket new) initialize: (ud pick: lottery numsPerTicket))]!

playTicket: aTicket
    cost := cost + lottery ticketCost.
    tickets add: aTicket.
    won := won + (lottery winnings: aTicket)! !
```

Figure 7: The class LottoCustomer.

```
LottoCustomer subclass: #TracingLottoCustomer
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''    !


!TracingLottoCustomer methods !

playTicket: aTicket
   super playTicket: aTicket.
   Transcript cr;
       nextPutAll: 'picked ticket: ';
       nextPutAll: aTicket printString;
       cr;
       nextPutAll: self winnings printString! !
```

Figure 8: The class `TracingLottoCustomer`.

```
"Load the lottery files, in the correct order.
 To load the lottery files, either file this in, or select and do it.
"
    |dirString d|
    dirString := Prompter prompt: 'Load lottery classes from this directory?'
                            default: 'd:\classes\cs541\etc\lottery'.
    dirString isNil
        ifTrue: [MessageBox message: 'Not loading anything'.
                 ^self].
    d := Directory pathName: dirString.
    #('ticket.cls' 'wnnngtck.cls' 'random.cls' 'prbbltyd.cls' 'unfrmdsc.cls'
          'lottery.cls' 'lttcstmr.cls' 'trcngltt.cls')
        do: [ :fn | (File pathNameReadOnly: fn in: d)
                        fileIn;
                        close. ]
```

Figure 9: A script to load all the class files, found in `loadit.st`.

```
"Example of running the lottery system."

    | lotto cust |
    lotto := Lottery picking: 7 from: (1 to: 50)
                     paying: #(0.0 1.0 10.0 100.0 1000.0 10000.0 1000000.0).
    "cust := (LottoCustomer new: 5000.0) lottery: lotto."
    cust := (TracingLottoCustomer new: 600.0) lottery: lotto.
    lotto pickWinner.
    cust play: 500.
    Transcript cr;
        nextPutAll: 'Final winnings: ';
        nextPutAll: cust winnings printString;
        cr.
```

Figure 10: A script to load all the class files, found in `example.st`.

References

[1] A. Goldberg and D. Robson. *Smalltalk-80, The Language and its Implementation.* Addison-Wesley Publishing Co., Reading, Mass., 1983.

[2] D. E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming.* Addison-Wesley Publishing Co., Reading, Mass., 1969.

[3] W. R. LaLonde. *Discovering Smalltalk.* The Benjamin/Cummings Publishing Company, Inc, Redwood City, California, 94065, 1994.