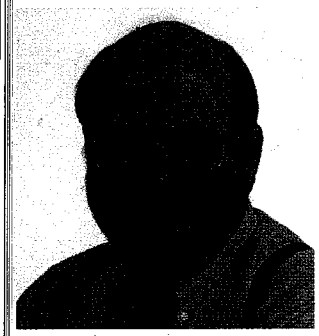

My Girl Scouts Are Badder Than Your Girl Scouts

BY ROGER SESSIONS

Some of the best programming students I've taught have been 15- and 16-year old Girl Scouts. At this age they are old enough to deal with intellectual issues, but still young enough to have wonder. This is the general age of my daughter Emily's troop. When the troop leader (who also happens to be my wife, Alice) asked me to help them get their computer badge, I was delighted.

I had two meetings. The first was easy. We spent an hour on the Web learning to do searches and discussing censorship and an hour at our local computer store learning to configure systems. But this still left me a second meeting to fill.

I decided that I would use that meeting to teach them C++ and how to work with object frameworks. Now I know most adults spend years learning these topics, but I figured these girls had three advantages. First, they liked computers. Second, they had nothing to unlearn. Third, nobody had ever told them these topics were hard.

I set up a simple game framework. The Girl Scouts then split into teams and programmed game players. The players then plugged into the game framework and played against each other.

The game was very successful. The girls developed some fascinating strategies, learned (with the help of some knowledgeable C++ programmers) to turn these strategies into algorithms, and then to turn those algorithms into working C++ objects. They also learned what it means to work within an object framework.

It occurred to me that this project would be even more interesting if opened up to a larger audience. So that is what I'm doing in this article. I invite all of my readers to program players for this framework, and we will have a large playoff.

I especially urge readers who are working with youth groups, such as Girl Scouts and Boy Scouts, to involve their youth and help them to design strategies and submit players. Many of these groups offer computer badges, for which this project will probably make them eligible. To sweeten the pot, I am offering a first prize ($100), second prize ($50), and third prize ($25) to the best submissions from youth groups. These prizes have been donated by Object-Watch Inc.

Here is the basic game setup. The name of the game is "Dog Meets Dog." The goal is to program various types of dogs. The framework will create one instance each of the various dog types, and then set up a series of rounds. In each round, two dogs will be chosen to play against each other. Each dog is told which dog it is playing against in that round. The dog then has to decide whether it will share or steal from the other dog. Both dogs make the decision. Both dogs are then informed of what the other dog decided.

In each round, the framework assigns payoffs to each dog. The unit of payoffs is dog biscuits. If both dogs decide to steal, each dog is awarded one dog biscuit. If both dogs decide to share, each dog is awarded three dog biscuits. If one dog steals and the other shares, the stealer gets five dog biscuits and the sharer none.

A full game consists of many rounds, many more rounds than there are dog objects. In the course of a game, each dog will meet each other dog many times.

I did not originate the idea for this game. I first ran into it in a book that was published several years ago,

Listing 1: Pseudo-code for the game framework.

```
initialize scores;
instantiate dogCollection;
fill dogCollection with all dogs playing;
repeat until (game-over) {
  get two dogs from dogCollection;
  get unique id from first dog;
  get unique id from second dog;
  ask first dog what it wants to do with second dog;
  ask second dog what it wants to do with first dog;
  compute scores;
  tell first dog what second dog did;
  tell second dog what first dog did; }
display scores;
```

and one I consider highly influential in shaping my own philosophy. I am not going to give the reference now, because I want to encourage readers to develop their own game strategies rather than turn this into a research project. I will give the reference in a later article when I discuss the results of this contest.

Although I did not originate this basic game, I have added several features I believe to be novel. First, I have turned this into an object-oriented framework. Second, I have set up the players as instances of C++ classes and have programmed both the overall framework and the dog players in VisualAge C++ running on OS/2. Third, I have made the players dogs, which for some reason has never before been done.

The game framework is typical of many frameworks: it defines and implements an architectural framework in which objects operate and interact. The framework itself is a program. It instantiates specialized objects and coordinates their interactions. The pseudo-code for the game framework program is shown in Listing 1.

Two related interactions occur in each round of the game. In the beginning of the round, the dogs are asked what they want to do with their opponents. At the end of the round, the dogs are informed of what their opponents decided to do with them.

Each dog is assigned a unique, stable ID, which is a long integer. This ID is assigned by the framework at the time the dog is instantiated, and it never changes. The dogs are identified to each other by this ID. So when a dog is asked how it wants to interact with dog 14, it can base its decision on its history of previous interactions with dog 14. When the dog is told what dog 14 decided to do, it can make a note of that information to be used in future interactions when it meets dog 14 again.

The dogs are not told about interactions in which they did not participate; for example, dog 11 is not told

Listing 2: C++ Definition of a dog player.

```
typedef int dogBiscuits;
typedef enum resultType {share, steal} result;

class dog {
public:

   dog();
// dog constructor.

   virtual char *IAmA() = 0;
// Returns the class of the dog.

   virtual result meetOtherDog(
     int dogNumber) = 0;
// Called by the framework to ask how this dog will
// interact with the opponent in this round.

   virtual void thisIsWhatHappened(
     int dogNumber,
     dogBiscuits yourScore,
     result youDidThis,
     result otherDogDidThis);
// Called by the framework to tell this dog what the
// result of this round is.

   void yourNumberIs(int dogNumber);
// Used by the framework to assign a unique id.

   int whatIsYourNumber();
// Used by the framework to determine the unique id.

protected:
   virtual int howManyMeetingsWithThisDog(
     int dogNumber);
// Returns how many meetings this dog had with a given
// opponent.

   virtual void whatHappenedInPreviousMeetingWithThisDog(
     int dogNumber,
     int meetingNumber,
     result* youDidThis,
     result* otherDogDidThis);
// Returns information about a particular meeting with a
// given opponent.

   virtual dogBiscuits yourScore();
// Returns the current score for the dog.
// (Not used by the framework)

private:
   /* ... */
};
```

of the outcome between dog 12 and dog 3.

In an object-oriented framework, base classes are typically provided. These base classes contain both concrete methods (ones that have been fully implemented) and abstract methods (ones that have been defined but not implemented). The abstract methods are the hooks by which programmers provide specialized objects.

As an example, let's look at the C++ definition of dog, shown in Listing 2. I have simplified the definition by showing only protected and public areas.

First of all, let's consider the purpose of the different protections. C++ provides three different protection types within a class definition. The *public* region defines information about the class that anybody can use. The *protected* region defines information that can be used only by methods in this class or derived classes. The *private* section defines information that can only be used by methods of this class (**dog**).

The dog's public region contains both virtual and nonvirtual methods. A virtual method can be overridden in a derived class. A nonvirtual method cannot be overridden. If you are not familiar with the concept of overriding methods and polymorphism, see the February issue of *OS/2 Magazine* ("littleDogs, Polymorphism, and Frameworks," p. 46).

Two of the virtual methods are declared using the peculiar C++ syntax **=0** (for example, **IAmA()**). This syntax is used to declare an abstract virtual method.

So the dog methods have the following characteristics:

- They may be public or protected.
- They may be virtual or nonvirtual.
- They may be abstract or concrete.

Let's see how we make these choices.

Methods that will be called by the framework are public. An example of this is the method **meetOtherDog**, the method used by the framework to ask the dog how it wants to interact in this round. Methods that are used only within **dog** are private, and those intended for derived dog types are protected. An example of a protected method is **whatHappenedInPreviousMeetingWithThisDog**, a method intended to be used by the override of **meetOtherDog**.

Methods that the dog types either may or must override are virtual.

Methods that the dog types *may* override are concrete and virtual. An example of a method that the dog types may override is **thisIsWhatHappened**, the method used by the framework to tell the dog the result of the round. The base class provides a perfectly acceptable implementation of this but also allows the dog types to override it, if the programmer has a better idea.

Methods that the dog types *must* override are abstract and virtual. An example of such a method is **meetOtherDog**. The whole point of this game is for the dog to provide a different implementation of this method, so no default is provided.

Methods that the dog may not override are nonvirtual. Dogs are not allowed to change their IDs, so the methods that deal with IDs, such as **whatIsYourNumber** are nonvirtual.

A typical dog type will be relatively simple, overriding exactly two methods: **IAmA** and **meetOtherDog**.

Listing 3: Definition of niceDog.

```
class niceDog : public dog {
  char *IAmA();
  result meetOtherDog(
  int dogNumber); };
```

Listing 4: Implementation of niceDog.

```
char *niceDog::IAmA() {
  static char *myTypeIs =
  "niceDog";
  return myTypeIs; }
result niceDog::meetOtherDog(
  int dogNumber) {
    return share; }
```

Listing 5: Implementation of sneakyDog's meetOtherDog.

```
result sneakyDog::meetOtherDog(int dogNumber) {
  int totalMeetings;
  result IDid, otherDid;
  totalMeetings = howManyMeetingsWithThisDog(dogNumber);
  if (totalMeetings == 0) {
    return share; }
  whatHappenedInPreviousMeetingWithThisDog(
    dogNumber, totalMeetings, &IDid, &otherDid);
  if (IDid == share) {
    return steal; }
  else {
    return share; } }
```

Listing 6: Implementation of unforgivingDog's meetOtherDog.

```
result unforgivingDog::meetOtherDog(int dogNumber) {
  int totalMeetings, n;
  result IDid, otherDid;
  totalMeetings = howManyMeetingsWithThisDog(dogNumber);
  if (totalMeetings == 0) {
    return share; }
  for (n=1; n<=totalMeetings; n++) {
    whatHappenedInPreviousMeetingWithThisDog(
      dogNumber, n, &IDid, &otherDid);
    if (otherDid == steal) {
      return steal; } }
  return share; }
```

IAmA is used by the framework to determine the class of the dog. **meetOtherDog** is used to determine the dog's decision on the round.

A typical example of a dog type is **niceDog** (Listing 3). Like all players, **niceDog** is derived from our framework-provided dog. As you can see, the actual work involved with defining a new dog type is much less than you would expect from the dog discussion. Notice that while **niceDog** doesn't say anything about overriding the **meetOtherDog** method, this is implied by the fact that the base class declared the method virtual.

niceDog uses one of the simplest possible algorithms in its implementation of **meetOtherDog**. Its code is shown in Listing 4. It always shares. We might rename **niceDog** to be **patsyDog**.

I have also implemented a **badDog**, a dog that always steals.

Most dogs will base their decision on whether to share or steal with a particular dog on information about previous encounters with that dog. The protected virtual methods defined for **dog** are provided for that purpose. One example of such a dog is **sneakyDog**, who always does the reverse of what he did last time he met that dog. If last time he shared, this time he steals. **sneakyDog**'s implementation of **meetOtherDog** is shown in Listing 5. **sneakyDog** is one of the dogs the Girl Scouts invented.

sneakyDog bases his share/steal decision only on the last interaction with the other dog. **unforgivingDog** looks through the entire previous history of interactions with the other dog. If the other dog ever stole, then **unforgivingDog** steals. She never gives you another chance. Her implementation is shown in Listing 6.

Notice that both **sneakyDog** and **unforgivingDog** make extensive use of the protected dog methods. None of the dog implementations shown here take advantage of the opportunity to override these methods. I don't see why anybody would want to override these, but I also don't want to eliminate the possibility.

Let's look at a few rounds of this game. Figure 1 shows six rounds and the decisions each dog makes based on their own implementations of **meetOtherDog**. Notice that the framework instantiates only one of each dog type.

Figure 1: Six rounds of dog meets dog.

```
1.  nice dog meets unforgiving dog
    nice dog shares; unforgiving dog shares
    nice dog +3; unforgiving dog +3;
    (The only possible outcome between these two dogs)

2.  nice dog meets bad dog
    nice dog shares; bad dog steals
    nice dog +0; bad dog + 5
    (The only possible outcome between these two dogs)

3.  unforgiving dog meets bad dog
    unforgiving dog shares; bad dog steals
    unforgiving dog +0; bad dog + 5
    (Unforgiving dog hasn't learned about bad dog yet)

4.  unforgiving dog meets nice dog
    nice dog shares; unforgiving dog shares
    nice dog +3; unforgiving dog +3
    (The only possible outcome between these two dogs)

5.  bad dog meets unforgiving dog
    bad dog steals; unforgiving dog steals
    bad dog +1; unforgiving dog + 1
    (unforgiving dog will never forget round 3)

6.  unforgiving dog meets sneaky dog
    unforgiving dog shares; sneaky dog shares
    unforgiving dog +3; sneaky dog +3
    (sneaky dog starts out nice, but it won't last)

Scores at the end of round 6:
                nice dog:       6
                unforgiving dog: 10
                bad dog:        11
                sneaky dog:     3
(of course, the game is young)
```

This should give you an idea of the game. Now it's your turn. Can you beat **puddingHeadDog**, another of Troop 161's inventions? Or how about **jeanBobDog**? (This is a Texas troop!)

Fine Print

OK. Here is the deal:

1. All entries must be e-mailed to *roger@fc.net* before September 1st. All entries must have a return e-mail address.

2. All entries must compile on VisualAge C++ on OS/2 without warnings or errors. I will not fix errors or warnings.

3. In the event that I am inundated with entries, I reserve the right to limit the contest to the first 50 entries from youth groups.

4. One instance of each dog will be instantiated and played against each other dog. Each dog will play against each other dog at least five times, perhaps much more.

5. An entry consists of a header file (similar to Listing 3) and an implementation file consisting of code for **IAmA**, modeled after the

The World Of Objects, *cont'd on p. 63*

The World of Objects, *cont'd from p. 47*

code in Listing 4, and **meetOtherDog**, using the algorithm of your choice.

6. Youth groups are allowed to have help with the coding, but you must promise that they developed the algorithms on their own.

7. One entry per e-mail address. If a youth group is involved, it must be identified.

8. No cheating by overriding methods you shouldn't override or updating private data you shouldn't be accessing.

So? Are you in?

Your first stop is the SOMobjects Home Page where I will maintain a full set of the official rules and a *.zip* file containing all the files you need to compile and test this framework using VisualAge C++. I will include several sample dog types. On the SOMobjects Home Page, look for a link to the *OS/2 Magazine* Dog Meets Dog Contest. The URL for the SOMobjects Home Page is *http://www.fc.net/~roger/owatch.htm*.

Good luck. You'll need it. My Girl Scouts are bad. OS/2

Acknowledgements

I am grateful to Girl Scout Troop 161 (Amy, Emily, Meg, Catie, Erin, Zer, and Chandra) and its intrepid leader (and my wife) Alice Sessions for their inspiration for this article.

Roger Sessions is president of ObjectWatch Inc., a company specializing in training and consulting in the use of SOM and DSOM. He has spoken at over 30 conferences and has written extensively. His books include Object Persistence: Beyond Object-Oriented Databases*;* Class Construction in C and C++: Object-Oriented Fundamentals*; and* Reusable Data Structures for C. *Roger also publishes an Internet newsletter called* ObjectWatch on SOM *and can be contacted on the Internet at roger@fc.net.*

The REXX Column, *cont'd from p. 56*

By using a trace instruction that gets its settings from your own environment variable, you can eliminate this hassle. I use an environment variable of **TRACE** in my own programs and have a group of instructions, all written as a single line with the instructions separated with semicolons, which I move around in the program as I need it. When I want it out of the way, I move the entire line to a position in the program where program flow will never reach it. I call this instruction my "magic" trace instruction, and it lies dormant in *skeleton.cmd* until I need it. I can then move it to the point where I want it and run the program with an environment variable of **SET TRACE=?R** to cause interactive tracing. I then reset the environment variable with **SET TRACE=** to negate the trace. OS/2

Dick Goran is the author of The REXX Reference Summary Handbook*, an OS/2 reference. He has been in the computer industry since 1961 and was in the IBM mainframe systems software development business until 1987. He teaches OS/2 REXX to corporate clients and lectures on OS/2. His free REXX utility programs can be downloaded from OS2DF1, Lib 6 on CompuServe or FTPed from ftp://ftp.cfsrexx.com/pub. He can be reached via e-mail at 71154.2002@compuserve.com.*

Resources:

REXXLIB ($50)
Quercus Systems
(408) 867-7399
WWW *http://www.quercus-sys.com*
READER SERVICE NO. 120

Rexx SuperSet/2 ($79.00)
GammaTech
(405) 947-8080
E-mail *72274.102@compuserve.com*
READER SERVICE NO. 121

Dave Boll's RXU (free for download)
FTP *ftp.cfsrexx.com/pub/rxu19.zip*