Generalized Message Passing in a Virtual Reality Application

David Flater

February 24, 1997

Abstract

The message passing semantics of an object-oriented programming language is one of the factors that determines how powerful the language is. Whether it is referred to as a "method invocation," a "dispatching call," an "event," a "signal," or something else, each object-oriented programming language supports some operation that is similar in effect to "message passing." What differs from language to language is the semantics of that message passing. These semantics largely determine the degree to which an object-oriented language supports polymorphism, overloading, type checking, and abstraction.

Although it is theoretically possible to implement any given application in any computationally complete language, the flexibility of the language that is used will, in practice, have a great influence over the shape of the final product. Those features that are extremely difficult to implement in the language being used are less likely to be implemented. If the project is one where the choice of programming language is not as important as the quality of the final product, such as in a research environment, then it is wise to investigate the alternatives.

Virtual reality is one research application that can easily be hampered by the limitations inherent in a programming language. We define the concepts of recipient resolution and action resolution, discuss their application to virtual reality, and suggest an approach for extending Objective C to support them.

1 Introduction

It is generally considered to be desirable for a programmer to be able to concentrate on the high-level structure of a program separate from the messy implementation details. Object-oriented programming languages offer ways of structuring programs that help to achieve that goal. By supporting polymorphism, dynamic binding, and dynamic type checking, they give the programmer the ability to state more concisely the high-level behavior of the program.

One may think of routines in a program as having a kind of signal-to-noise ratio that is the number of statements that say what needs to be done divided by the number of statements that say how to do it. Code with a high signal-to-noise ratio is readable and intuitive; code with a low signal-to-noise ratio is cryptic. A statement that is part of the signal of a given subroutine may become noise if it is moved up into the main program, since it "belongs" in the subroutine.

As the flexibility of the programming language decreases, the signal-to-noise ratio of programs written in that language tends to decrease as well. For example, consider the case of translating a C++ virtual function call into C. The programmer must either set up pointers to the correct functions and make the function call via a pointer, or code up a decision tree at the point of call to select among the possible functions. Either way, the added code is pure noise.

Our goal is to improve the signal-to-noise ratio of programs, with particular attention to the virtual reality (VR) class of applications. To do that, we make the programming language more flexible and provide greater support for encapsulating code in the object to which it pertains. However, we must first examine the various levels of flexibility supported by existing programming languages in order to better understand the extensions that can be made.

2 Types of Dispatching

Each programming language comes with its own vocabulary and set of concepts. When one person thinks of sending a message, another thinks of invoking a method. In this paper we will concentrate on the message passing paradigm, since our generalized semantics are most naturally expressed in that paradigm.

The generality of the message passing semantics, which corresponds to how flexible the message passing is when it is used by a programmer, is limited by how the compiler and run-time environment dispatch (deliver) messages to their recipients. Existing programming languages support different kinds of dispatching, but the differences are not always obvious when terms such as "dynamic binding" are used loosely. In order to avoid ambiguities in terminology, we make the following definitions:

1. Non-Dispatching

   Non-dispatching languages have static type checking[1] and static binding[2].[1] The message and the specific class of the recipient (the receiving object) must both be known at compile time. These languages are not very interesting for object-oriented programming since only the most basic forms of polymorphism (e.g. function name overloading) can be supported. C is an example of a non-dispatching language.

2. Subclass Dispatching

   Subclass dispatching is static type checking with dynamic binding. A message can be sent to an object without knowing the specific class of that object, provided that it is known to be derived from or identical to a parent class that is specified at compile time. The message handler (the function that is invoked when the message is received, sometimes called a *method* or a *member function*) is determined at run time since subclasses can override the message handlers of their parents. C++ supports subclass dispatching.

3. Classless Dispatching

   Classless dispatching requires dynamic type checking (if any type checking is done) and dynamic binding. It is possible to operate generically over all objects and all messages. A message can be sent without knowing the name of the message or the identity or class of the recipient at compile time. The compiler can no longer guarantee that the recipient of a message will be able to handle it, so run-time failures are possible. However, there are fewer restrictions on how the programmer must construct the class hierarchy for a given program. Objective C supports classless dispatching.

It is important to understand the difference between supporting an application and merely enabling it. Since all the programming languages mentioned below are computationally complete, they all *enable* the same set of programs to be written. However, writing VR applications in a language that has restrictive message passing semantics is substantially more difficult than writing them in a more flexible language. Where the user of a language with classless dispatching simply performs a single function call, the user of a statically bound language might need to write pages of code to achieve the same effect. Those features that are extremely difficult to implement in the language being used are less likely to be implemented, so it is better to use a language that provides *support* for the desired application.

With its 1995 revision, Ada[3][2] now supports subclass dispatching. Tagged types provide the functionality of subclasses, and a dispatching call is similar in effect to sending a message to an object without knowing its specific class. The parent class must still be specified at compile time. The first Ada standard[3] was non-dispatching.

[1] Dynamic type checking requires each object to have a "tag" that tells the run-time environment what its type is. Statically typed languages determine all types at compile time, so the types are implicit at run time.

[2] Static binding implies that a function invocation in the source code maps to exactly one function. With dynamic binding, there may be several candidate functions, and one of them is selected when the invocation is executed.

[3] Ada is a registered trademark of the U.S. Government, Ada Joint Program Office.

C++[4] supports subclass dispatching that is the same as Ada's; only the syntax and the terminology are different. A virtual function call in C++ is the same as a dispatching call in Ada.

Objective C supports classless dispatching. The data type "id" indicates an object in the generic sense; it is not necessary to specify the class of the recipient of a message at compile time. The "perform" message is used to achieve the effect of sending a message whose name is not known until run time. The following sample, written in the GNU[4] dialect of Objective C, demonstrates the sending of a message that is specified at run time to an object whose class is unspecified (in the function `send_some_message_to_some_object`):

```
#include <stdio.h>
#include <objc/Object.h>

@implementation Thing1: Object
- (void) mess1
{
  puts ("Thing1 instance got mess1");
}
@end

@implementation Thing2: Object
- (void) mess2
{
  puts ("Thing2 instance got mess2");
}
@end

void
send_some_message_to_some_object
(id recipient, char *message)
{
  //  The perform message and the
  //  sel_get_any_uid function are both
  //  defined by the compiler.
  [recipient perform:
    sel_get_any_uid (message)];
}

main ()
{
  char messname[80];
  int choice;
  id something_or_other;
  puts ("Enter name of message.");
  gets (messname);
  puts ("Enter number of object.");
  scanf ("%d", &choice);
  if (choice == 1)
    something_or_other = [Thing1 new];
  else
    something_or_other = [Thing2 new];

  send_some_message_to_some_object
    (something_or_other, messname);

  exit (0);
```

[4] Certain software packages are identified in this paper to foster understanding. Such identification does not imply recommendation or endorsement by the National Institute of Standards and Technology, nor does it imply that the packages identified are necessarily the best available for the purpose.

```
}
```

Here is a sample run of the above program:

```
Enter name of message.
mess2
Enter number of object.
2
Thing2 instance got mess2
```

The program will fail at run time if the specified message cannot be handled by the chosen object:

```
Enter name of message.
mess2
Enter number of object.
1
error: Thing1 (instance)
Thing1 does not recognize mess2
IOT trap/Abort
```

It will also fail if the user types the name of a message that is not declared in the program or in the Objective C library.

Although there have been many object-oriented languages over the years and several of great historical importance that we have not discussed, the languages already mentioned suffice to demonstrate the major forms of dispatching currently supported.

3 Generalized Dispatching

The word "resolution" is most often used to refer to the process of choosing the most appropriate version of an overloaded function, or choosing the correct method for the specific subclass in a subclass dispatching language. There are several possible matches for a message name, and it is up to the compiler or run-time environment to find the "best match."[5] However, it is possible to use best match resolution more extensively. The following three generalized forms of dispatching support best match resolution that is not restricted to name resolution:

4. Recipient Resolution

   A message can be directed at a group of recipients, and received by only the best one. "Best" is determined at run time using evaluation functions. Recipient resolution is routing[5] that is supported by the programming language instead of implemented by the programmer. Recipient resolution provides support for automatically using the best tool for a particular job.

5. Message Resolution

   A group of messages can be directed at a single recipient, and only the best one sent. Message resolution provides support for automatically choosing the best way to use a particular tool. We will not discuss message resolution in detail since it is less interesting than recipient resolution and is a special case of action resolution.

6. Action Resolution

   Action resolution combines recipient resolution with message resolution. Action resolution supports goal-based programming and can abstract out the common task of searching for the best next move.

The selection of the best recipient or best course of action is done by using *evaluation functions.* Evaluation functions are defined by the programmer to compute (or estimate, or specify) the desirability of sending a particular message to a particular recipient relative to the other possible combinations. An evaluation function for generalized dispatching serves somewhat the same purpose as the evaluation function in a best-first search[6].

Subclass and classless dispatching can be viewed as cases of recipient resolution in which the evaluation functions are solely based on the classes and subclasses, the number of parameters, and their data types. True recipient resolution gives the programmer the power to define dispatching priorities explicitly instead of being restricted to a single language-dependent evaluation function. The programmer's evaluation function can be time and situation-dependent and can use any available input.

To clarify the semantics of recipient and action resolution, we define an *action* to be the sending of a specific message to a specific object. The message sent is one of a set of messages, M, and the object is one of a set of possible recipients, R. Recipient resolution requires that M have only one member; action resolution does not.

The selection of a single action to perform out of all the possible combinations of messages and recipients is done by choosing the combination that maximizes the value of an evaluation function f. This value is called the *priority* of the action. f is a function of a message m, an object r, and the global state S. S is meant to incorporate all other possible inputs to the evaluation function, including message parameters, global variables, time-dependent information, and data received from remote sources. If more than one possible action has the maximum priority, then the choice among the winners can be made arbitrarily by the run-time environment[5]. However, one of the winning actions must be taken – issuing an error is not permissible.

Although it is fine in theory to have a single evaluation function that calculates priorities for all actions, in practice it is not very useful. In practice we will want to have a set of evaluation functions F, such that either m or r has been made implicit and the corresponding parameter omitted from each function call. The message or recipient is made implicit by encapsulating the evaluation function inside the message or object to which it relates.

The two approaches (make m implicit or make r implicit) are equivalent in the sense that both can be trivially transformed into the single monolithic evaluation function approach. However, they are vastly different in practice. It would make little sense for messages to encapsulate evaluation functions in an environment that supports only recipient resolution. Since it is the recipient, not the message, that is being resolved, it is only logical for the evaluation functions to be encapsulated by the recipients. That way the evaluation functions can be understood by the programmer as defining the behaviors of the recipients with respect to the messages that might be received.

It is not as obvious what the right approach is when action resolution is fully supported and resolving actions can be a two-dimensional problem. However, since there is more context associated with an object than with a message, it would still make more sense to associate evaluation functions with recipients than with messages. If desired, one can always get back to the single monolithic evaluation function by having each object point to it.

4 Extending Objective C

Although we have not actually implemented a compiler for a language that does recipient or action resolution, we have considered how Objective C might be extended to provide the additional functionality with minimal disruption of its present syntax.

To get started, we define an evaluation function in the top-level Object class, to be inherited by all objects. The name "priority" will henceforth be reserved for evaluation functions. By default, objects will disqualify themselves from action resolution by returning a negative priority regardless of the action:

```
- (float) priority: (char *) message:
                    (int) parmc:
                    (void **) parmv
{
```

[5]In defining the language, we specify neither determinacy nor randomness in this case; the results are implementation-dependent.

```
  return -1.0;
}
```

When action resolution is performed, the dispatcher will never choose an action that has a negative priority. If all possible actions have a negative priority, then a flag is set to indicate that the message could not be sent. Section 5.3 gives an example of why abnormal termination of the program at this point is not an acceptable alternative.

An evaluation function for a regular class of objects could then be done similar to the following example:

```
//  Evaluation function for legal dept.
- (float) priority: (char *) message:
                    (int) parmc:
                    (void **) parmv
{
  //  Messages taking one argument
  if (parmc == 1) {
    if (!strcmp (message, "sue")) {
      // Won't sue other lawyers
      if ([parmv[0] isKindOf:
      [Lawyer class]])
        return -1.0;
      else
        return 10.0;
    }
    if (!strcmp (message, "threaten"))
      return 8.0;
  }

  // Pass the buck for inherited methods
  return [super priority: message:
                          parmc:
                          parmv];
}
```

To group recipients and messages, we assume the existence of a Bag class that acts as a container for objects. Such a class is provided by the GNU Objective C Class Library, libobjects.

```
id recipients = [[[[[Bag new]
   addElement: legal_dept]
   addElement: public_relations_dept]
   addElement: president]
   addElement: spy];
```

So far we have done nothing that violates normal Objective C syntax, but now we come to the turning point. The following syntax is already defined in Objective C to mean that we want to send a message to the Bag object itself, rather than to one of the objects that it contains:

```
[recipients threaten: rival_company];
```

We will therefore invent a new operator to indicate that we want to choose the best recipient out of the Bag:

```
[`recipients threaten: rival_company];
```

In order to do action resolution, we must first decide what it means to have a Bag of messages. Messages in Objective C are not objects; one cannot simply throw a handful of them into a Bag. However, we can achieve the same result using message selectors or message names.

To emphasize that the Bag must contain only messages, we define a MessageBag subclass that allows messages to be added and removed by giving their names:

```
@interface MessageBag: Bag
- addMessage: (char *) message;
- removeMessage: (char *) message;
@end
```

The message names or selectors would be bagged using methods inherited from the parent class. Whether the messages are actually bagged by name or by selector does not matter, although it may be advantageous to verify that the named message does exist before it is bagged.

Now that we have MessageBags, we can do full action resolution:

```
id messages = [[[[[[MessageBag new]
               addMessage: "sue:"]
               addMessage: "threaten:"]
               addMessage: "conciliate:"]
               addMessage: "bash:"]
               addMessage: "observe:"];
[`recipients `messages: rival_company];
```

5 Generalized Dispatching and Virtual Reality

5.1 Recipient Resolution

When building a virtual world, we want to be able to create "robots." A VR robot is an active, intelligent entity within the virtual world that is controlled by the computer. Programming such a robot is often a laborious and difficult task because of the many details that must be handled. The programmer must try to think of every possible situation that the robot could encounter and add code to handle each one.

Consider the example of a robot trying to open a locked door. Without recipient resolution, the programmer must add code to the robot to decide which tool to use, at what time, and for how long:

```
- (BOOL) open_locked_door: door:
 (int) time_available
{
  id tool;
  assert (time_available > 0);
  //  Try to open the door with the key.
  if (tool = [self find_inventory: "key"]) {
    if ([tool open_door: door])
      return TRUE;
    if (!(--time_available))
      return FALSE;
  }
  //  Either didn't have key or it didn't work.
  //  Try to card the door.
  if (tool = [self find_inventory: "card"]) {
    if ([tool open_door: door])
      return TRUE;
    if (!(--time_available))
      return FALSE;
  }
  //  Getting desperate.
  //  If door is wooden, try the axe for a while.
  if ([door isKindOf: [WoodenDoor class]])
    if (tool = [self find_inventory: "axe"]) {
      int a;
      for (a=0;a<30;a++) {
        if ([tool open_door: door])
          return TRUE;
        if (!(--time_available))
          return FALSE;
      }
    }
  //  Try to burn our way in with propane torch.
  if (tool = [self find_inventory: "torch"]) {
    while ([tool has_propane] &&
           [door is_locked]) {
      if ([tool open_door: door])
        return TRUE;
      if (!(--time_available))
        return FALSE;
    }
  }
  //  Unable to open door.
  return FALSE;
}
```

This approach is inherently clumsy because we are attempting to pre-educate the robot with knowledge about every single tool it might encounter. If we later create a new tool, we must also update the robot's program, or else the robot will not know when to use the tool.

With recipient resolution, it is not necessary to pre-educate the robot. The usefulness of a given tool for a given task will be coded into the tool itself, where it belongs. The robot will then be able to pick up a new tool and recognize its usefulness without being re-educated:

```
- (BOOL) open_locked_door: door:
 (int) time_available
{
  assert (time_available > 0);
  for (; time_available &&
  [door is_locked]; time_available--)
    ['inventory open_door: door];
  return ![door is_locked];
}
```

Note that the clumsiness associated with limited-use tools has gone away. When the fuel for the torch is used up, the torch's evaluation function begins returning a negative priority, and the robot stops trying to use the torch. Similarly, the priority of the axe will decrease each time it fails to open the door. (It will always be negative if the door is not wooden.) Depending on how smart we want the robot to seem, the key and the card can either refuse to be used on the same door more than once, or refuse to be used on the wrong door in the first place. Although the total number of lines of code might have increased because of the need for priority functions, the top-level routine is much more straightforward, and the detail work has been moved into the lower levels where it belongs.

In recent years, VR has been redefined by some to refer to the sophisticated graphics that can complement a virtual world, rather than the virtual world itself. Fortunately, recipient resolution can be applied in many different domains, including graphics!

Suppose that we are designing a graphics engine that must render images of many objects in real time. Some of them are constantly moving, rotating, and changing; some do not move unless they are acted upon by another object; and some are simply backdrop. Once an object has been rendered, the rendering can be drawn and redrawn with relative ease; the expensive part is re-rendering objects that have moved or changed so that the picture will look right. At the same time as this rendering is taking place, we must also handle input from the user and compute the behaviors of the objects in real time. In order to keep up with all the computation that needs to be done, we must make intelligent decisions about when to re-render objects. A half second delay in updating the rendering of a slowly moving object will be considerably less noticeable than a half second delay in updating the rendering of an object that is flying around. However, if the fast moving object collides with the slowly moving one, we must devote a large time slice to both of them to make the collision look right.

Writing a main loop for this graphics engine that does not just rely on brute force to keep up with the work load could be a very difficult task. However, with recipient resolution, we can implement a complete task scheduler just by defining the priority function. The main loop could be as simple as this:

```
while (1)
  [‘world render];
```

The evaluation functions would adjust the priority upward as the speed of the object and the age of the rendering increased. Also included in the world would be a special object tasked with processing user input and sending “heartbeat” messages to objects to make them look alive and react to their surroundings. It is an unfortunate kluge that the “render” message is deliberately misused to insure that events unrelated to rendering (user input and heartbeats) will be serviced. In the next section we will see how this inelegance is removed by the more flexible action resolution.

5.2 Action Resolution

Recipient resolution takes for granted that we know which is the best course of action (i.e. which message to send) to achieve our goal. This is not always the case. Let us return to our first example in which we were simulating a company trying to deal with a threat posed by a rival company. With recipient resolution, if we assume that the best response to a threat is a counterthreat, we can delegate the decision of who is best to make the counterthreat. However, if we want to be able to choose between legal action, threats, diplomacy, PR wars, and watchful waiting, we must add code at the top level.

With action resolution, all of the information needed to choose a course of action is encapsulated in the definitions of the objects, so the programmer is free to delegate the decision making to the run time environment. Since lawyers can threaten and conciliate as well as sue, the public relations department can either threaten or conciliate, and the president can threaten, conciliate, or bash, the decision making is not a linear process. It is necessary to weigh the risks and benefits of each course of action, knowing that the effectiveness of each course of action is affected by the tool that is used and the threat that is being dealt with. The risks and benefits may also change over time: if, while observing the threat, the corporate spy discovers evidence of wrongdoing that can be used against the threat, the company might decide to take legal action on the next iteration of the loop.

Of course, the programmer is not at the mercy of the machine any more than he or she wants to be, since decisions can still be hard coded:

```
if ([threat isKindOf:
  [Hostile_takeover class]])
  [company buy_back_stock];
```

If we now return to the graphics example, we find that there is no longer a need for a kluge to poll user input and broadcast heartbeat messages. These actions can simply be resolved against the rendering work:

```
id animate = [[[[MessageBag new]
              //  Render & draw objects
              addMessage: "render"]
              //  Let objects be active
              addMessage: "heartbeat"]
              //  Poll user input
              addMessage: "userinput"];
while (1)
  [‘world ‘animate];
```

Instead of relying on a special object to periodically send heartbeats to every object in the world, we now let objects specify a priority for their own heartbeat. Different objects can now have different heart rates. Although we still need somebody to be responsible for polling user input, we no longer need to bind this unrelated action to the render message. The objects that cannot receive the userinput message simply return a negative priority.

5.3 Experiments with Recipient Resolution

The idea for recipient and action resolution came from Cheezmud[7], an experimental mud written in Objective C. A mud is a virtual world in which any number of users can interact with each other and with characters (robots) run by the computer. Cheezmud is copyrighted software that was developed independently and placed under the GNU Public License.

The virtual world can have any theme; we chose to use the traditional "swords and sorcery" theme in our experiments. Since Objective C does not support recipient resolution or action resolution, we settled for emulating recipient resolution via message routing[5] as shown in the appendix. This was sufficient to discover the many ways in which evaluation functions simplify the task of VR programming. Not only are they useful for abstracting out the complex decision trees that computer-controlled characters might have needed, they also permit user-controlled characters to exhibit a certain level of common sense in following the user's commands.

One major example is the "kill" command. In a swords and sorcery mud, combat is a frequent occurrence, and weapons are plentiful. The emulated recipient resolution of Cheezmud allows the computer-controlled characters to display intelligence in their choice of weaponry. When a character does a command, the message, in this case "kill," is resolved against the character's inventory (the objects that he or she is carrying), the room that the character is in, and the character himself (herself). This is because commands can be related to tools that are being carried (unlock door), to a location (go north), or to the character (grin). The character has an inherent "kill" method that corresponds to unarmed combat; most weapons have "kill" methods with a higher priority.

The default priority of the "kill" method for a weapon is calculated when the speed and maximum damage done by the weapon are set:

```
speed = s;
damage = d;
tspeed = ((float)s / 4.0 - 1.0) * 0.5 + 1.0;
prio = (float)damage / tspeed;
```

The result is a ranking of weapons by their effectiveness, such that the rate at which damage is done and the amount of damage that can be done at one time are both taken into consideration. This is merely a default evaluation function; a more complex function that returns different priorities depending on the nature of the enemy can easily be specified for a specific subclass of weapon. Characters will then use the weapon that is best suited for the specific threat being faced, when different kinds of weapons are best suited for different kinds of enemies.

The power of recipient resolution became particularly obvious when we created a single-use weapon (a bomb) that explodes when thrown at an enemy. Any computer-controlled character that possesses it knows to use the bomb first, then finish off a surviving enemy with some other weapon. It was not necessary to add code to test for the special weapon. Its evaluation function returns a high priority, but it vanishes after one use. The next time the character does the "kill" command, it is resolved against his or her remaining inventory, and the fight goes on without interruption.

Another interesting application of evaluation functions was for containers, such as sacks or bags. A character can carry more items if he or she bags them up. In Cheezmud, a player that has more than one sack can simply "bag" something without specifying which sack to use. Items are trivially distributed among all available sacks by defining evaluation functions that return a variable priority for the "bag" command depending on how full the sack is:

```
if ([contents count] >= capacity)
  return -1;
return 20.0 *
  (float)(capacity - [contents count])
/ (float)capacity;
```

Note that returning -1 when the bag is full prevents anything else from being stuffed into it. If the player attempts to execute a "bag" command with all bags full, an error is returned. This is an example of why abnormal termination of the program should not result if all possible recipients have negative priority. Ideally, all languages would support Ada or SQL-like[8] exception handling, and we could simply write an exception handler for this case. Instead, Cheezmud uses the return code of the message routing routine to indicate success or failure.

From these examples it is possible to see how helpful it would be to have recipient or action resolution for virtual reality programming. The effort of writing evaluation functions is negligible compared to the convenience of abstracting out so many special cases from the main program.

6 Conclusion

Recipient resolution and action resolution provide support for new styles of programming. By moving evaluation criteria away from the main program and abstracting out the search process, we state more concisely the task that is to be accomplished and thus increase the signal-to-noise ratio of the source code.

Future progress in flexible programming languages may occur as the result of "persistent" programming. Persistent languages allow programs to preserve internal state indefinitely and increase support for dynamic linking and dispatching so that previously compiled modules can work with newly compiled modules without themselves having been recompiled. An example is E[9], a persistent version of C++. While languages like E are adding database functions to programming languages, database languages are closing the gap in the other direction. SQL has achieved computational completeness with the addition of Persistent Stored Modules and the control statements that were necessary for PSM to be useful. SQL is now exploring new territory with its function call semantics. Given sufficient metadata, a program can use a SELECT statement to search at run time for modules appropriate to a particular task, and invoke them dynamically. The designers of standard SQL are being careful to preserve the safety and compilability of the language, so certain opportunities for even greater flexibility have been foregone. That leaves open the possibility for experimentation with wilder variants of SQL that do more work at run time. Overload resolution could be done at the last minute so that the best matching function would always be the one that is called, even if the CALL statement were compiled and the best matching function were defined dynamically just now. Exception handlers could recover from failures by calling the next best function from the list of candidate functions.

The most effective programming can only be done if the language permits the programmer to express his or her intentions in a way that seems natural and intuitive. Lower software quality and longer development times result if it is necessary to reformulate ideas in an awkward way to get them implemented. Language designers always run the risk of making a language that seems intuitive and natural to them, but not to anyone else. Hopefully the language extensions that we have made will not fall into that category.

Appendix

The following method resolves the recipient of the message generated when a character tries to do something in Cheezmud. Our definitions of recipient resolution and action resolution were only made after Cheezmud was developed, so the word "action" is used loosely in this example. Note how the character, the character's location, and the character's inventory (contents) are searched for appropriate message handlers. NULL is returned if the recipient cannot be resolved.

```
- resolve_action: (char *) action:
                  (int) numargs
{
  id actor = NULL;
  float prio = -1.0, t_prio;

  void countprio (id whatever)
  {
    if ((t_prio = [whatever priority:
```

```
    action: numargs]) > prio) {
      if (t_prio >= 0.0) {
        prio = t_prio;
        actor = whatever;
      }
    }
  }

  countprio (self);
  if (location)
    countprio (location);
  if ([self isKindOf: [Container class]])
    [[self contents] withObjectsCall:
    countprio];
  return actor;
}
```

The actual sending of the message is done one level up, in a method called "do." The following is the relevant code fragment:

```
if ((t = [self resolve_action:
verb: numargs])) {
  char temp[80];
  sprintf (temp, "%s:", verb);
  if (numargs > 1)
    strcat (temp, ":");
  a = sel_get_any_uid (temp);
  if (numargs == 1)
    [t perform: a with: self];
  else
    [t perform: a with: self with: dobj];
  return 1;
}
```

References

[1] Gordon Blair, John Gallagher, David Hutchison, and Doug Shepherd, editors. *Object-Oriented Languages, Systems and Applications*, chapter 4, page 85. Halsted Press, New York, 1991.

[2] Ada 95 HTML-Hypertext Reference Manual. <URL:http://www.adahome.com/rm95/>.

[3] United States Department of Defense. *Reference Manual for the Ada Programming Language.* U.S. Government Printing Office, Washington, D.C., 1983.

[4] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, Reading, MA, 1990.

[5] Dennis de Champeaux, Douglas Lea, and Penelope Faure. *Object-Oriented System Development*, chapter 21, page 387. Addison-Wesley, Reading, MA, 1993.

[6] Eugene Charniak, Christopher K. Riesbeck, Drew V. McDermott, and James R. Meehan. *Artificial Intelligence Programming*, page 166. Lawrence Erlbaum Associates, Hillsdale, NJ, second edition, 1987.

[7] Cheezmud, an experimental mud implemented in Objective C. <URL:http://www.universe .digex.net/~dave/files/cheezmud-1.0.tgz>.

[8] ISO-ANSI working draft, Database Language SQL (SQL3), March 1995. X3H2-95-083 .. X3H2-95-089.

[9] Murali Vemulapati, Ram D. Sriram, and Amar Gupta. Incremental loading in the persistent C++ language E. *Journal of Object-Oriented Programming*, 8(4):34–42, 1995.

About the author: *David Flater is a Computer Scientist in the Information Management Group of the NIST Computer Systems Laboratory. The National Institute of Standards and Technology is an agency of the Technology Administration, U.S. Department of Commerce.*