Proceedings of the 37th Annual Simulation Symposium, April 18-22, 2004, Arlington, Virginia, pp. 178-186

# Object-Oriented SIMSCRIPT

Stephen V. Rice[1], Ana Marjanski[2], Harry M. Markowitz, and Stephen M. Bailey
[1]*The University of Mississippi, rice@cs.olemiss.edu*
[2]*CACI Products Company, amarjanski@caci.com*

## Abstract

*A new object-oriented version of the SIMSCRIPT programming language is under development. Incorporating modules and object classes, the new language is designed for modular, object-oriented simulation programming.*

## 1. Introduction

SIMULA, the first object-oriented programming language, was developed in Norway in the 1960s. Its inventors, Kristen Nygaard and Ole-Johan Dahl, received the ACM Turing Award in 2001 for their innovation. The influence of SIMULA became widespread in the 1980s when many object-oriented programming languages were developed. The annual OOPSLA Conference (on Object-Oriented Programming Systems, Languages, and Applications) was first held in 1986. The 1990s saw rapid adoption of the object-oriented paradigm in software engineering.

SIMULA is also a simulation programming language. The combination of simulation programming and object-oriented programming is natural: simulation programs need to represent real-world objects and the object-oriented approach provides a powerful and elegant technique for representing them. In fact, simulation programming inspired the development of object-oriented programming [1]. Bjarne Stroustrup's experience with SIMULA in building a large simulation program inspired him to add object-oriented features to the C programming language to create C++ [2].

Despite the popularity of object-oriented programming and the recognized value of object orientation to simulation programming, there are remarkably few object-oriented simulation programming languages. In the early 1980s, David McArthur, Philip Klahr, and Sanjai Narain at the RAND Corporation developed an object-oriented simulation language for the U.S. Air Force based on Lisp named ROSS [3]. In the late 1980s, Joel West, Alasdar Mullarney, Steve Rice, and Ron Belanger at CACI developed an object-oriented simulation language for the U.S. Army based on Modula-2 named ModSim. It was later marketed by CACI under the names MODSIM II and MODSIM III [4]. For an object-oriented language that does not provide simulation features, a software library may be available that facilitates the development of simulation programs, for example, Sim++™ for C++ [5] and Silk™ for Java [6].

In 1962, Harry Markowitz, Bernard Hausner, and Herb Karr at the RAND Corporation developed the SIMSCRIPT I simulation programming language for the U.S. Air Force [7]. Philip Kiviat, Richard Villanueva, and George Benedict later collaborated with Markowitz and Hausner to create SIMSCRIPT II, also at RAND [8]. A commercial version known as SIMSCRIPT II.5 has been marketed and improved by CACI since the 1970s [9].

The SIMSCRIPT I language had "considerable impact" on the design of SIMULA "through its list processing and time scheduling mechanisms" and its "random drawing and other utility routines" [1]. Hausner spent the summer of 1963 in Norway, which increased the communication between the SIMSCRIPT and SIMULA teams. The influence was reciprocal: Nygaard is given credit in the SIMSCRIPT II manual for his design suggestions, and in the mid-1970s, SIMSCRIPT II.5 incorporated SIMULA's "quasi-parallel" time-elapsing processes.

SIMSCRIPT II.5 has been widely used in the programming of large and small simulation models, at least in part because of language features that greatly facilitate simulation programming. The language includes support for discrete-event simulation, combined discrete-event/continuous simulation, interactive graphics and animation, database access, and an integrated development environment.

This article is the first report of a new version of SIMSCRIPT under development named SIMSCRIPT O-O (for Object-Oriented). SIMSCRIPT O-O is a strongly-typed object-oriented simulation programming language. The design of its object-oriented features has been influenced by SIMULA and four other strongly-typed object-oriented languages: C++, Eiffel, Java, and ModSim.

SIMSCRIPT O-O is a superset of SIMSCRIPT II.5. The new features in SIMSCRIPT O-O share a similar syntax and interoperate with SIMSCRIPT II.5 statements so that new and old combine to form a coherent language. In this article, we focus on the following new features: modules; classes and objects and their attributes,

methods, and sets; inheritance and accessibility; and a special kind of subroutine for simulation programming called a process method. Existing SIMSCRIPT II.5 programs run without modification or performance penalty in SIMSCRIPT O-O.

## 2. Modules

A SIMSCRIPT II.5 program consists of a preamble, a main routine, and zero or more subordinate routines. The preamble contains all global data definitions. Through the introduction of modules, SIMSCRIPT O-O facilitates modular design, promotes better source code organization, and makes it easier for a team of programmers to collaborate on large programs and for program components to be shared by multiple programs.

A SIMSCRIPT O-O program consists of a main module and zero or more subordinate modules known as *subsystems*. A main module is like a SIMSCRIPT II.5 program: it consists of a preamble, a main routine, and zero or more subordinate routines. In fact, every SIMSCRIPT II.5 program can be viewed as a SIMSCRIPT O-O main module. A subsystem is a named module consisting of a public preamble, an optional private preamble, and zero or more routines. A subsystem may be imported by the preamble of main modules and by the public and private preambles of other subsystems. A preamble may import any number of subsystems.

The names defined and imported in the preamble of a main module are accessible to the routines of the main module. The names defined and imported in the public preamble of a subsystem are accessible to the private preamble and routines of the subsystem, and to every module that imports this subsystem. The names defined and imported in the private preamble of a subsystem are accessible only to the routines of the subsystem. A public preamble is analogous to a definition module in ModSim (and Modula-2) and is stored in its own file. The private preamble and subsystem routines correspond to a ModSim implementation module, but may span multiple files.

"System attributes," which are variables that are global to a main module, are defined in the preamble of a main module by **the system** statement. "Subsystem attributes" are global to a subsystem and are defined in the public and private preambles of the subsystem using **the subsystem** statement. A subsystem may have a special **initialize** routine which is called once during program initialization and may be used to initialize subsystem attributes.

If the name LOAD is defined in the public preamble of an imported subsystem named PORT, it may be referred to simply as LOAD in an importing module if the name is unambiguous in the module; otherwise, it must be qualified by the subsystem name and specified as PORT**:**LOAD.

The syntax permits **module** and **package** to be used as synonyms of **subsystem**. In the following example, a main module named SHIPPING imports a subsystem named PORT:

```
preamble for the SHIPPING system ''main module
importing the PORT module ''imported subsystem
   ...
end ''preamble

routines for the SHIPPING system

main
   ''may refer to LOAD or PORT:LOAD,
   ''but may not refer to CAPACITY
   ...
end ''main
...

public preamble for the PORT subsystem
   ''definitions here are visible to
   ''importing modules
   ...
   ''the following statement specifies the
   ''public interface of a routine
   define LOAD as a routine
      given an integer argument
         and a real argument
      yielding 2 real values
end ''public preamble

private preamble for the PORT subsystem
   ''definitions here are not accessible to
   ''importing modules
   ...
   ''define a private subsystem attribute
   the subsystem has a CAPACITY
   define CAPACITY as an integer variable
end ''private preamble

routines for the PORT subsystem

initialize
   ''this routine gets executed automatically
   ''before the main routine begins execution
   ...
   let CAPACITY = 50
end ''initialize

routine LOAD given IVAL, RVAL yielding R1, R2
   ''private implementation of the routine
   ...
end ''LOAD
...
```

## 3. Classes and objects

A SIMSCRIPT II.5 "temporary entity" is a dynamically-allocated block of memory containing variables known as attributes. An entity's attributes are accessed using an untyped reference variable containing the address of the entity. The attributes are globally accessible. A temporary entity is akin to a dynamically-allocated C **struct** or Pascal **record**.

In C++, Stroustrup greatly elaborates the C **struct** to make it an object [10]. In SIMSCRIPT O-O, we chose

not to provide a complex extension of the SIMSCRIPT II.5 temporary entity. Instead we introduce a new kind of entity type called a class.

A **begin class** block is specified in a preamble to define a class. Definitions of attributes, methods, and sets are placed within this block. The following block defines a class named SHIP:

```
begin class SHIP
   ''definitions for class SHIP go here
   ...
end ''SHIP
```

A **begin class** block defines a mode, i.e., a data type, with the same name as the class, and reference variables may be defined to be of this mode. For example:

```
define TANKER as a SHIP variable
```

An instance of a class is called an object and is created by a **create** statement. The following statement allocates an object of class SHIP, initializes its attributes to zero, and assigns its address to the reference variable named TANKER:

```
create TANKER
```

With a few exceptions, dynamic memory explicitly allocated by a SIMSCRIPT II.5 program must be explicitly de-allocated by the program to avoid a memory leak. SIMSCRIPT O-O provides a garbage-collection mechanism which automatically reclaims dynamically-allocated memory that is no longer in use. Also a **destroy** statement may be specified to explicitly de-allocate an object:

```
destroy TANKER
```

C++ supports both static and dynamic objects, which introduces considerable complexity into the language. Like Java, SIMSCRIPT O-O supports only dynamic objects accessible by reference variables.

## 4. Attributes

Object attributes are defined by **every** and **define**...**variable** statements within a **begin class** block. For example:

```
begin class SHIP
   ...
   every SHIP
      has a NAME,
          a NUMBER.OF.ENGINES,
      and a MAXIMUM.SPEED

   define NAME as a text variable
   define NUMBER.OF.ENGINES
      as an integer variable
   define MAXIMUM.SPEED as a real variable
   ...
end ''SHIP
```

Attribute names have class scope, that is, they are local to the class. These names may also be defined in the global scope or local to another class without implying SIMSCRIPT II.5's notion of "common attributes." Where there is ambiguity, it is necessary to qualify an attribute name with the name of its class, for example, SHIP'NAME, or PORT**:**SHIP'NAME if SHIP is defined within the PORT subsystem.

To access an object attribute, a reference value expression evaluating to the address of an object is specified in parentheses immediately following the attribute name. This is the same notation used to access temporary entity attributes in SIMSCRIPT II.5. Each attribute access is type-checked at compile time. For example:

```
let NAME(TANKER) = "Emerald Glory"
let NUMBER.OF.ENGINES(TANKER) = 4
let MAXIMUM.SPEED(TANKER) = 25.0
```

Object attributes may be arrays. For example:

```
begin class SHIP
   ...
   every SHIP
      has a MAXIMUM.RPM

   define MAXIMUM.RPM as a 1-dim real array
   ...
end ''SHIP
```

An array must be reserved (i.e., allocated) before it can be accessed. In this example, the array is given four elements, one for each engine of the tanker. Each element contains the maximum number of revolutions per minute (rpm) for one engine.

```
reserve MAXIMUM.RPM(TANKER) as 4
let MAXIMUM.RPM(TANKER)(1) = 182.5
let MAXIMUM.RPM(TANKER)(2) = 175.0
let MAXIMUM.RPM(TANKER)(3) = 180.0
let MAXIMUM.RPM(TANKER)(4) = 167.5
```

A *class attribute* is associated with the class rather than an object of the class. Whereas each object has its own copy of each object attribute, the program has only one copy of each class attribute. In C++, class attributes are known as "static data members."

Class attributes are defined by **the class** and **define**...**variable** statements within a **begin class** block. **The class** statement is similar to **the system** and **the subsystem** statements. For example:

```
begin class SHIP
   ...
   the class has a NUMBER.OF.SHIPS

   define NUMBER.OF.SHIPS
      as an integer variable
   ...
end ''SHIP
```

Because a class attribute is not associated with an object, it is accessed without a reference value expression:

```
write NUMBER.OF.SHIPS
   as "There are ", i 3, " ships.", /
```

# 5. Methods

An *object method* is a routine that is invoked on behalf of an object. If the method returns a function result, it is a *function method*; otherwise, it is a *subroutine method*.

Object methods are defined by **every** and **define**...**method** statements within a **begin class** block. The optional **define**...**method** statement indicates the number and mode of the method's arguments, and the mode of the function result for function methods. If this statement is omitted, the method is assumed to be a subroutine method with no arguments.

In the following example, the SHIP class defines two object methods. DESCRIBE is a subroutine method that writes attributes of a SHIP object to the current output unit. COMPARE.RPM is a function method that returns the number of engines with a maximum rpm exceeding a given rpm.

```
begin class SHIP
   ...
   every SHIP
      has a DESCRIBE method
      and a COMPARE.RPM method

   define COMPARE.RPM as an integer method
      given a real argument
   ...
end ''SHIP
```

Like attribute names, method names have class scope. Where there is ambiguity, it is necessary to qualify a method name, for example, SHIP'DESCRIBE or PORT**:**SHIP'DESCRIBE.

A subroutine method is invoked by a **call** statement like a SIMSCRIPT II.5 subroutine, and a function method is invoked like a SIMSCRIPT II.5 function. However, a reference value expression is specified in parentheses immediately following the method name and identifies the object for which the method is invoked. For example:

```
call DESCRIBE(TANKER)
let RESULT = COMPARE.RPM(TANKER)(172.0)
```

Method calls are type-checked at compile time. The reference value expression is passed implicitly by value to the object method and is not considered to be a "given" argument. In this example, DESCRIBE has no given arguments and COMPARE.RPM has one: 172.0.

## 5.1. Method implementations

A method implementation is a routine that starts with **method** instead of **routine**, **function**, or **subroutine**. The modes of method arguments need not be defined within the implementation because they are already known from the **define**...**method** statement in the preamble. The method implementations for a class must appear within the module in which the class is defined.

Within a method implementation, the reference value of the object for which the method is invoked is available in an implicitly-defined local reference variable with the same name as the class. In other object-oriented languages, this variable is named "self," "this," or "current." In our example, this variable is implicitly defined by:

```
define SHIP as a SHIP variable
```

The following statements specify implementations for the DESCRIBE and COMPARE.RPM methods:

```
method SHIP'DESCRIBE
   write NAME(SHIP), NUMBER.OF.ENGINES(SHIP),
      MAXIMUM.SPEED(SHIP) as "The ship ",
      "named ", t *, " has ", i 2, " engines",
      " and a maximum speed of ", d(5,1), /
end ''SHIP'DESCRIBE

method SHIP'COMPARE.RPM(RPM)
   define I, COUNT as integer variables
   for I = 1 to NUMBER.OF.ENGINES(SHIP)
   with MAXIMUM.RPM(SHIP)(I) > RPM
      add 1 to COUNT
   return with COUNT
end ''SHIP'COMPARE.RPM
```

If a method name is unqualified in the heading of a method implementation, the method is assumed to belong to the class identified by the preceding **methods for** statement. Within an object method, the parenthesized reference value expression may be omitted when accessing an object attribute or invoking an object method; in this case, the implicitly-defined local reference variable is implied. With these changes, the method implementations of our example look like this:

```
methods for the SHIP class

method DESCRIBE
   write NAME, NUMBER.OF.ENGINES,
      MAXIMUM.SPEED as "The ship ",
      "named ", t *, " has ", i 2, " engines",
      " and a maximum speed of ", d(5,1), /
end ''DESCRIBE

method COMPARE.RPM(RPM)
   define I, COUNT as integer variables
   for I = 1 to NUMBER.OF.ENGINES
   with MAXIMUM.RPM(I) > RPM
      add 1 to COUNT
   return with COUNT
end ''COMPARE.RPM
```

## 5.2. After creating and before destroying

An **after creating** statement within a **begin class** block specifies an object method to be called implicitly after each object of the class is created. A **before destroying** statement within the block identifies an object method to be called implicitly before each object of the class is destroyed. These methods must be subroutine methods with no arguments. It is often desirable to define another object method that accepts arguments and uses them to initialize a newly-created object. However, this method must be called explicitly. For example:

```
begin class SHIP
   ...
   every SHIP
      has a  CONSTRUCTOR method,
          a  DESTRUCTOR method,
      and an INITIALIZE method

   after creating a SHIP, call CONSTRUCTOR
   before destroying a SHIP, call DESTRUCTOR

   define INITIALIZE as a method
      given a  text argument,
            an integer argument,
        and a  real argument
   ...
end ''SHIP
...

methods for the SHIP class

method CONSTRUCTOR
   write as "A ship has been created", /
   add 1 to NUMBER.OF.SHIPS
end ''CONSTRUCTOR

method DESTRUCTOR
   call DESCRIBE
   write as "This ship is destroyed", /
   subtract 1 from NUMBER.OF.SHIPS
end ''DESTRUCTOR

method INITIALIZE given TVAL, IVAL, RVAL
   let NAME = TVAL
   let NUMBER.OF.ENGINES = IVAL
   let MAXIMUM.SPEED = RVAL
end ''INITIALIZE
```

Now when "**create** TANKER" is executed, an object of class SHIP is allocated, its attributes are initialized to zero, its address is assigned to TANKER, and the **after creating** method is called implicitly:

```
call CONSTRUCTOR(TANKER)
```

After the **create** statement, the program should call the INITIALIZE method explicitly to initialize the object:

```
call INITIALIZE(TANKER)
   given "Emerald Glory", 4, 25.0
```

When the object is destroyed, whether explicitly by a **destroy** statement or implicitly by the garbage collector, the **before destroying** method is called implicitly before the object is de-allocated.

## 5.3. Class methods

A *class method* is a routine associated with a class, and unlike an object method, it is invoked without an object of the class. Hence, the invocation of a class method lacks the reference value expression, and the implementation lacks the implicitly-defined local reference variable. Class methods are defined by **the class has a**...**method** and **define**...**method** statements within a **begin class** block. In C++, class methods are known as "static member functions."

## 5.4. Right/left methods and monitored attributes

Through a technique known as "overloading," C++ and Java permit a method to have multiple implementations differentiated by the number and data type of their arguments. Method overloading provides notational convenience; however, the rules are complex for determining which implementation will be invoked by a method call. It can be a difficult bug to find if the implementation invoked is not the one intended. We chose not to allow this form of method overloading in SIMSCRIPT O-O.

In SIMSCRIPT II.5, each function may have one or two implementations: a "right" implementation invoked when the function is called to return a value, and/or a "left" implementation invoked when the function is assigned a value. If the latter is provided, a function call may appear on the left-hand side of an assignment statement (an "lvalue" in C terminology). A left implementation does not **return with** a value; instead, an **enter with** statement brings the assigned value into the routine for processing.

SIMSCRIPT II.5 also supports "monitored attributes." An attribute monitored on the right and/or the left implicitly defines a function with the same name and mode as the attribute, with right and/or left implementations. If monitored on the right, every right-

hand access to get the value of the attribute invokes the right implementation of the function. If monitored on the left, every left-hand access to set the value of the attribute invokes the left implementation.

An attribute access and a function call have the same syntax; therefore, an attribute can be changed to a function, and monitoring can be turned on and off, without modifying the code for each attribute access. Bertrand Meyer, the inventor of the Eiffel programming language, calls this the "uniform access property" [11]. Like SIMSCRIPT II.5, Eiffel uses the same notation to access an attribute and call a function. However, Eiffel's uniform access is limited to scalar attributes and functions with no arguments. In SIMSCRIPT II.5, an *n*-dimensional array may be monitored by a function with *n* integer arguments. The same syntax is used for accessing an element of the array and calling the function; *n* integer expressions are interpreted as either array subscripts or function arguments.

In SIMSCRIPT O-O, a function method may have right and/or left implementations. The right implementation starts with **method** or **right method**, and the left implementation begins with **left method**. Furthermore, an attribute may be monitored on the right and/or the left by a function method. An object method monitors an object attribute, and a class method monitors a class attribute.

Accessing an attribute and calling a function method share the same syntax. In our example, the NUMBER.OF.ENGINES attribute could be monitored on the left to prevent the assignment of an invalid value and to ensure that the MAXIMUM.RPM array has one element for each engine. The MAXIMUM.SPEED attribute can easily be replaced by a function method of the same name. The right implementation of the method might compute the maximum speed of the ship based on the number and maximum rpm of its engines. The left implementation might flag attempts to set the maximum speed now that its value is computed and not stored. For example:

```
begin class SHIP
   ...
   every SHIP
      has a NAME,
          a NUMBER.OF.ENGINES,
      and a MAXIMUM.SPEED method

   define NAME as a text variable
   define NUMBER.OF.ENGINES
      as an integer variable
      monitored on the left
   define MAXIMUM.SPEED
      as a real method with no arguments
   ...
end ''SHIP
...

methods for the SHIP class

left method NUMBER.OF.ENGINES
   define N as an integer variable
   enter with N ''obtain the assigned value
   if N is positive
      ''assign N to the
      ''NUMBER.OF.ENGINES attribute
      move from N
      ''if MAXIMUM.RPM doesn't have N elements
      if DIM.F(MAXIMUM.RPM) is not equal to N
         ''discard the old array and
         ''create a new one
         release MAXIMUM.RPM
         reserve MAXIMUM.RPM as N
      always
   else ''N is negative or zero
      write as "Invalid number of engines", /
   always
end ''left NUMBER.OF.ENGINES

right method MAXIMUM.SPEED
   define RESULT as a real variable
   ''compute maximum speed and
   ''store it in RESULT
   ...
   return with RESULT
end ''right MAXIMUM.SPEED

left method MAXIMUM.SPEED
   write as "Cannot set maximum speed", /
end ''left MAXIMUM.SPEED
```

## 5.5. Statistics

SIMSCRIPT II.5 facilitates the computation of statistics. The number, sum, mean, sum of squares, mean square, variance, standard deviation, maximum, minimum, and histogram of values assigned to a variable are computed automatically by specifying a **tally** statement, or an **accumulate** statement for statistics weighted by simulation time. The assigned values are acquired and processed by monitoring the variable on the left.

In SIMSCRIPT O-O, a **tally** or **accumulate** statement in a **begin class** block specifies statistics to collect on the values assigned to an object attribute or class attribute. The attribute may be a scalar or an array. In the following example, SPEED is an object attribute implicitly monitored on the left, and AVG.SPEED is a system-generated object method with a right implementation that returns the time-weighted mean of the values stored in SPEED.

```
begin class SHIP
   ...
   every SHIP
      has a SPEED

   define SPEED as a real variable
   accumulate AVG.SPEED as the mean of SPEED
   ...
end ''SHIP
```

## 6. Sets

A SIMSCRIPT II.5 "set" is an ordered collection of entities which is implemented as a doubly-linked list. The entities in a set are known as "members," and each set has an "owner" which is either an entity or "the system." An entity that may belong to a set has implicitly-defined "member attributes," including pointers to the predecessor and successor entities in the set. The owner possesses implicitly-defined "owner attributes" containing the number of members in the set and pointers to the first and last members. The "set discipline" specifying the order of members may be FIFO, LIFO, or ranked based on attribute values of members. An entity may own, and may belong to, any number of sets.

Likewise, a set of objects may be defined in SIMSCRIPT O-O. It may be owned by an object or a class, or by the system or subsystem, or by a SIMSCRIPT II.5 temporary, permanent, or compound entity. An object or class may own any number of sets and arrays of sets. A set contains either SIMSCRIPT O-O objects or SIMSCRIPT II.5 entities, but not a mixture of the two.

In the following example, a set of SHIP objects named FLEET is owned by the SHIP class:

```
begin class SHIP
   ...
   every SHIP
      belongs to a FLEET

   the class
      owns a FLEET
   ...
end ''SHIP
```

The following statements create and initialize a SHIP object and add it to the FLEET:

```
create SHIP
call INITIALIZE(SHIP)
   given "Queen Mary", 8, 35.0
file SHIP in FLEET
```

The entire FLEET is described by:

```
for each SHIP of FLEET
   call DESCRIBE(SHIP)
```

## 7. Inheritance

A SIMSCRIPT O-O class may be derived from one base class (*single inheritance*) or from two or more base classes (*multiple inheritance*), and it may be a base class of one or more derived classes. A derived class inherits all of the attributes, methods, and sets defined and inherited by its base classes, and may define attributes, methods, and sets of its own. Furthermore, it may "override" an inherited object method by providing its own implementation(s) of the method.

In the following example, a class named TUGBOAT is defined. It is derived from the SHIP class as indicated by the **is a** clause of the **every** statement. It thereby inherits the object attributes and object methods of SHIP, namely NAME, NUMBER.OF.ENGINES, MAXIMUM.SPEED, MAXIMUM.RPM, DESCRIBE, COMPARE.RPM, CONSTRUCTOR, DESTRUCTOR, INITIALIZE, SPEED, and AVG.SPEED.

An object attribute named TUGGED.SHIP is introduced, which refers to the ship that is currently being tugged by the tugboat, or contains zero if no ship is being tugged. New methods named BEGIN.TUGGING and END.TUGGING are also defined. The DESCRIBE method inherited from SHIP is overridden and a new implementation is provided that invokes the old implementation to extend its behavior.

```
begin class TUGBOAT

   every TUGBOAT
      is  a  SHIP,
      has a  TUGGED.SHIP,
          a  BEGIN.TUGGING method, and
          an END.TUGGING method, and
      overrides the DESCRIBE method

   define TUGGED.SHIP as a SHIP variable

   define BEGIN.TUGGING
      as a method given a SHIP argument

end ''TUGBOAT
...

methods for the TUGBOAT class

method BEGIN.TUGGING given SHIP
   if TUGGED.SHIP is not zero
      call END.TUGGING
   always
   let TUGGED.SHIP = SHIP
   write NAME, NAME(TUGGED.SHIP)
      as t *, " begins tugging ", t *, /
end ''BEGIN.TUGGING

method END.TUGGING
   if TUGGED.SHIP is not zero
      write NAME, NAME(TUGGED.SHIP)
         as t *, " finishes tugging ", t *, /
      let TUGGED.SHIP = 0
   always
end ''END.TUGGING

method DESCRIBE
   ''invoke the overridden implementation
   call SHIP'DESCRIBE
   if TUGGED.SHIP is zero
      write as "It is an idle tugboat", /
   else
      write NAME(TUGGED.SHIP)
         as "It is currently tugging ",
            "the ship named ", t *, /
   always
end ''DESCRIBE
```

A TUGBOAT object inherits from SHIP the ability to belong to a FLEET set. The following statements create and initialize a TUGBOAT object, add it to the FLEET, and ask it to tug the TANKER:

```
create TUGBOAT
call INITIALIZE(TUGBOAT)
   given "Mighty Mouse", 1, 20.0
file TUGBOAT in FLEET
call BEGIN.TUGGING(TUGBOAT) given TANKER
```

Using the object-oriented technique of *polymorphism*, the implementation invoked by the **call** statement below depends on the type of the object: SHIP'DESCRIBE is called for each SHIP object and TUGBOAT'DESCRIBE is called for each TUGBOAT object in the FLEET.

```
for each SHIP of FLEET
   call DESCRIBE(SHIP)
```

The reference value of a TUGBOAT object may be assigned to a SHIP reference variable because every TUGBOAT "is a" SHIP. Thus, TUGGED.SHIP may refer to a TUGBOAT object, which allows one tugboat to tug another. The reference value of a SHIP object may not be assigned to a TUGBOAT reference variable; such assignments are flagged at compile time.

## 8. Accessibility

A *private class* is defined by a **begin class** block appearing in the preamble of a main module or in the private preamble of a subsystem. The class is accessible only within the defining module.

On the other hand, a *public class* is defined by a **begin class** block in the public preamble of a subsystem and by an optional, second **begin class** block in the private preamble of that subsystem. Only those attributes, methods, and sets defined and inherited in the first block are accessible to importing modules; the rest are accessible only within the subsystem. This allows some characteristics of a public class to be private. Typically, attributes of a public class are defined within the private preamble, and methods defined in the public preamble provide the interface to the class. For example:

```
public preamble for the PORT subsystem
   ...
   begin class SHIP
      ''public part of a public class,
      ''i.e., the class interface
      ...
   end ''SHIP
   ...
end ''public preamble

private preamble for the PORT subsystem
   ...
   begin class SHIP
      ''private part of a public class,
      ''accessible only within the subsystem
      ...
   end ''SHIP
   ...
end ''private preamble
```

Each class is accessible to the other classes defined within the same module. These cohabitating classes are "friends" in C++ terminology.

## 9. Process methods

SIMSCRIPT II.5 provides instantaneous events and time-elapsing processes for discrete-event simulation. Unlike an event routine, the execution of a process routine may be suspended and later resumed, and while it is suspended, simulation time may elapse. In SIMSCRIPT II.5, a process routine may **wait**, thereby suspending itself and scheduling its resumption, or it may **suspend** itself and allow another routine to **reactivate** it. Other routines may **interrupt** and **resume** a **wait**ing process.

In SIMSCRIPT O-O, a *process method* is a subroutine method with no yielded arguments that is both a method *and* a process. It may be invoked directly by a **call** statement like any SIMSCRIPT O-O subroutine method, and it may be scheduled for execution at a simulation time like any SIMSCRIPT II.5 process. Each scheduled invocation is represented by a "process notice" (a temporary entity with some special attributes) that is filed in the future-events set. A direct invocation is performed without a process notice.

A process method may be either an object method or a class method. If it is an object method, then it is invoked on behalf of an object and can be thought of as an activity of the object. Concurrent activities of an object are conveniently modelled by concurrent process methods invoked on behalf of the same object.

In the following example, we define a process method named VOYAGE in the SHIP class. We also define an object attribute named AT.SEA.FLAG that is set equal to one when the ship is on a voyage and is zero otherwise.

```
begin class SHIP
   ...
   every SHIP
      has a  VOYAGE process method
      and an AT.SEA.FLAG

   define VOYAGE as a process method
      given a real argument

   define AT.SEA.FLAG as an integer variable
   ...
end ''SHIP
...
```

```
methods for the SHIP class

process method VOYAGE given DURATION
   if AT.SEA.FLAG = 1 ''already on a voyage
      write NAME as "The ship named ", t *,
         " is already travelling", /
      return
   otherwise
   write NAME as "The ship named ", t *,
      " begins a voyage", /
   let AT.SEA.FLAG = 1
   wait DURATION days ''go on the voyage
   let AT.SEA.FLAG = 0
   write NAME as "The ship named ", t *,
      " ends a voyage", /
end ''VOYAGE
```

In the following statement, the caller directly invokes the VOYAGE process method to initiate a three-day voyage of the tanker. The caller then waits for the method to return and the completion of the voyage before executing the next statement in sequence.

```
call VOYAGE(TANKER) given 3.0
''the next statement is executed after
''the voyage has ended
```

Alternatively, the caller may schedule the voyage and then proceed without delay to the next statement:

```
schedule a VOYAGE(TANKER) given 3.0 now
''the next statement is executed before
''the voyage has begun
```

The voyage may be scheduled to begin sometime in the future:

```
schedule a VOYAGE(TANKER)
   given 3.0 in 7.0 days
```

These **schedule** statements create a process notice and file it in the future-events set. The reference value of the process notice is stored in an implicitly-defined object attribute named VOYAGE. This reference value may be used to **cancel**, **reschedule**, **interrupt**, and **resume** the voyage. For example:

```
cancel the VOYAGE(TANKER)
reschedule the VOYAGE(TANKER) in 8.0 days
...
interrupt the VOYAGE(TANKER)
wait 2.0 days
resume the VOYAGE(TANKER)
```

## 10. Conclusion

The SIMSCRIPT O-O programming language adds modules and object classes to the SIMSCRIPT II.5 language. SIMSCRIPT II.5's view of the world in terms of entities and their attributes and sets has been extended to incorporate the object-oriented view. This extension includes object and class attributes; object and class methods; sets containing objects; sets owned by objects and classes; single and multiple inheritance; and polymorphism. Modular software development and accessibility control are achieved through the introduction of subsystems. Garbage collection simplifies the management of dynamic memory.

The descriptive, English-like syntax of SIMSCRIPT II.5 facilitates the communication and verification of simulation models and is continued in SIMSCRIPT O-O. SIMSCRIPT II.5's processes are elaborated as process methods, and its monitoring and statistics-gathering mechanisms can be applied to object and class attributes.

SIMSCRIPT O-O is a general-purpose language suitable for diverse programming tasks. Its many features for simulation make it one of a small number of object-oriented simulation programming languages.

## 11. Acknowledgement

The development of SIMSCRIPT O-O is sponsored by CACI.

## 12. References

[1] K. Nygaard and O.-J. Dahl, "The Development of the SIMULA Languages," *ACM SIGPLAN Notices*, 13(8), 1978, pp. 245-272.

[2] B. Stroustrup, "A History of C++: 1979-1991," in T.J. Bergin and R.G. Gibson (Eds.), *History of Programming Languages*, ACM Press, New York, 1996, pp. 699-769.

[3] D. McArthur, P. Klahr, and S. Narain, "ROSS: An Object-Oriented Language for Constructing Simulations," Technical Report R-3160-AF, RAND Corporation, Santa Monica, CA, 1984.

[4] CACI Products Company, *MODSIM III: The Language for Object-Oriented Programming*, CACI Products Company, La Jolla, CA, 1996.

[5] G. Lomow and D. Baezner, "A Tutorial Introduction to Object-Oriented Simulation and Sim++™," in *Proceedings of the 1990 Winter Simulation Conference*, IEEE, Piscataway, NJ, 1990, pp. 149-153.

[6] K.J. Healy and R.A. Kilgore, "Introduction to Silk™ and Java-Based Simulation," in *Proceedings of the 1998 Winter Simulation Conference*, IEEE, Piscataway, NJ, 1998, pp. 327-334.

[7] H.M. Markowitz, B. Hausner, and H.W. Karr, *SIMSCRIPT: A Simulation Programming Language*, Prentice-Hall, Englewood Cliffs, NJ, 1963.

[8] P.J. Kiviat, R. Villanueva, and H.M. Markowitz, *The SIMSCRIPT II Programming Language*, Prentice-Hall, Englewood Cliffs, NJ, 1968.

[9] CACI Products Company, *SIMSCRIPT II.5 Reference Handbook*, CACI Products Company, La Jolla, CA, 1997.

[10] B. Stroustrup, *The C++ Programming Language*, Third Edition, Addison-Wesley, Reading, MA, 1997.

[11] B. Meyer, *Eiffel: The Language*, Prentice Hall, New York, 1992.