(54) Title: SYSTEM, METHOD AND ARTICLE OF MANUFACTURE FOR A VARIABLE STORAGE FRAMEWORK

(57) Abstract: A system, method and article of manufacture are provided for managing data associated with a variable. First provided is a variable with an accessor object assigned thereto. Such accessor object includes a code segment dictating a manner in which data associated with the variable is managed in memory. In operation, upon a variable being called, the accessor object assigned to the variable is invoked. Thereafter, the data associated with the variable is handled in memory as prescribed by the accessor object.
SYSTEM, METHOD AND ARTICLE OF MANUFACTURE
FOR A VARIABLE STORAGE FRAMEWORK

FIELD OF THE INVENTION

The present invention relates generally to computer programming systems, and more particularly to data access in object oriented programming.

BACKGROUND OF THE INVENTION

To facilitate discussion, the following terms used in this discussion are defined below:

Class: A template of a program object. It defines the data components of the object and includes the code segments or methods that can be performed using the object.

Object: An instance of a class. The actual data storage defined by the class is encapsulated in the object.

Method: A named code fragment associated with a class. Each method implements a function which may be performed on an object of a specific class.

Accessor: A special class defined for the purpose of storing and retrieving data as described herein.

As imperative programming systems have developed, more and more attention has been paid to high-level representations of the entities making up the system being modeled. One result of this attention is object-oriented programming languages such as Java and C++, in which the entities in the system are modeled as objects. Each object has a class which defines how the object can relate to the other entities in the system and what can be done with the object. The high-level representations employed in the object-oriented programming systems make it possible to create understandable and maintainable models of complex systems.
While computer operating systems and applications are gaining in complexity, it is impossible for systems developers to put every feature that a user might need into the system. As such, programs are often built on top of extendible libraries of high-level classes and objects. However, at the same time these library providers need to retain some control over the amount of modification that can be performed by developer subclassing. Because of function overloading, this type of modification control is often unattainable unless developer subclassing is severely curtailed.

Figure 1 is an illustration showing a prior art data retrieval and storage system 100. The data retrieval and storage system 100 includes a predefined class named VARIABLE_CLASS 102 having variable Value 108, and data retrieval and storage methods GetValue 104 and SetValue 106. Method GetValue 104 retrieves data stored in variable Value 108, while method SetValue 106 stores data in variable Value 108. In addition, method SetValue 106 calls function Check_Value 110 whenever it is executed. In this example, function Check_Value 110 checks to make sure the value being stored in variable Value 108 is within certain bounds, for example a number between 1-200.

The data retrieval and storage system 100 further includes object Age 112, which is of class VARIABLE_CLASS 102, and object A 114 that uses the methods of object Age 112 to store and retrieve data. In use, object A 114 uses method SetValue 106 to store data in variable Value 108, which is located in object Age 112. However, as shown in Figure 1, whenever object A 114 calls SetValue 106 to store data in variable Value 108, the function Check_Value 110 is called. Thus, every time data is stored in variable Value 108, the data is checked to determine if the value of the data is between 1-200. This type of data check is common to prevent system crashes from happening when improper data is sent storage locations.

A problem occurs, however, when other developers are allowed to create subclasses of class VARIABLE_CLASS 102, and redefine the data storage and retrieval methods GetValue 104 and SetValue 110.

For example, a developer creates a subclass of class VARIABLE_CLASS 102 named SUBCLASS_VARIABLE 116. Subclass SUBCLASS_VARIABLE 116 includes variable New_Value 117, and data retrieval and storage methods GetValue 118 and SetValue 120. However, methods GetValue 118 and SetValue 120 overload methods GetValue 104 and SetValue 106 defined in class VARIABLE_CLASS 102. Thus, whenever a program calls
methods GetValue 118 and SetValue 120 from an object of subclass SUBCLASS_VARIABLE 116, GetValue 104 and SetValue 106 from class VARIABLE_CLASS 102 will not be executed. Further, if SetValue 120 from subclass SUBCLASS_VARIABLE 116 is called, method Check_Value 110 will not be called since method Check_Value 110 is only called from method SetValue 106 defined in class VARIABLE_CLASS 102.

For example, object Child_Age 122 is of subclass SUBCLASS_VARIABLE 116, and thus includes methods GetValue 118 and SetValue 120 from SUBCLASS_VARIABLE 116, as well as variable New_Value 117. When object B 124 uses the method SetValue 120 of object Child_Age 122 to store data in variable New_Value 117, the data is no longer checked by method Check_Value 110 before being stored in variable New_Value 117. Thus, improper data may be stored in variable New_Value 117, possibly causing a system wide crash.

Thus, the creator of class VARIABLE_CLASS 102 must either disallow modification of the data retrieval and storage methods, or risk system crashes and other undesirable events if essential operations are not performed during data retrieval and storage routines.

This problem is even further augmented in the case of open source software, where other programmers are permitted to freely generate their own subclasses.
SUMMARY OF THE INVENTION

A system, method and article of manufacture are provided for managing data associated with a variable. First provided is a variable with an accessor object assigned thereto. Such accessor object includes a code segment dictating a manner in which data associated with the variable is managed in memory. In operation, upon a variable being called, the accessor object assigned to the variable is invoked. Thereafter, the data associated with the variable is handled in memory as prescribed by the accessor object.

In one embodiment, the handling of the data associated with the variable includes storing or retrieving the data in the memory. In various other embodiments, such handling may include constraining a value of the data in the memory. Further, it may include mapping an undefined value of the data to a defined value in the memory.

In another embodiment, the accessor object may be programmable. Further, the data associated with the variable may be stored and retrieved to or from memory over a network. As an option, such network may include a wide area network such as the Internet and employ a protocol such as TCP/IP and/or IPX.

The present invention thus includes an access delegation mechanism for data storage. The implementation derives much greater flexibility by delegating the storage process to the accessor object. The accessor object may manage data in any desired way, rather than retrieving it in the standard manner. Unlike the case of sub-classing, however, the overall behavior of the storage mechanism is maintained.

The accessor object may be implemented as an object held by the variable. Completely new accessor objects can be implemented, subclasses of existing accessor objects can be created, or the default accessor object may be utilized. In addition, the accessor object may be changed at run time if necessary. This feature might allow an object to migrate from a local computer to a shared server by delegating where its data was stored.

These and other advantages of the present invention will become apparent to those skilled in the art upon a reading of the following descriptions of the invention and a study of the several figures of the drawing.
**BRIEF DESCRIPTION OF THE DRAWINGS**

Figure 1 illustrates a data storage mechanism of the prior art;

Figure 2 is a schematic diagram illustrating one exemplary hardware implementation of the present invention;

Figure 3 is a flow diagram illustrating the accessor object that facilitates storage of data associated with variables in accordance with one embodiment of the present invention;

Figure 4 is a flowchart illustrating a method of storing data associated with a variable in accordance with one embodiment of the present invention;

Figure 5 is a flowchart illustrating one example of a way of retrieving data associated with a variable in accordance with one embodiment of the present invention;

Figure 6 is a flowchart illustrating one example of a way of setting data associated with a variable in accordance with one embodiment of the present invention;

Figures 7-10 are flowcharts illustrating examples of how various accessor objects are invoked to handle data during the methods of Figures 5 and 6; and

Figures 11-14 illustrate flowcharts showing processes carried out by the accessor objects upon being invoked in the processes of Figures 7-10.
Detailed Description Of The Invention

Figure 1 illustrates a data storage mechanism of the prior art. Figures 2-14 illustrate a technique of the present invention which is capable of overcoming the problems associated with utilizing the data storage mechanism of Figure 1 in an object oriented programming environment. The present invention addresses the issue of subclassing control by the use of modifiable accessor objects to provide low level data control while still allowing predefined classes to enforce essential data storage and retrieval routines.

Figure 2 is a schematic diagram of one possible hardware implementation by which the present invention may be carried out. As shown, the present invention may be practiced in the context of a personal computer such as an IBM compatible personal computer, Apple Macintosh computer or UNIX based workstation.

A representative hardware environment is depicted in Figure 2, which illustrates a typical hardware configuration of a workstation in accordance with one embodiment having a central processing unit 210, such as a microprocessor, and a number of other units interconnected via a system bus 212. The workstation shown in Figure 2 includes a Random Access Memory (RAM) 214, Read Only Memory (ROM) 216, an I/O adapter 218 for connecting peripheral devices such as disk storage units 220 to the bus 212, a user interface adapter 222 for connecting a keyboard 224, a mouse 226, a speaker 228, a microphone 232, and/or other user interface devices such as a touch screen (not shown) to the bus 212, communication adapter 234 for connecting the workstation to a communication network 235 (e.g., a data processing network) and a display adapter 236 for connecting the bus 212 to a display device 238.

The workstation typically has resident thereon an operating system such as the Microsoft Windows NT or Windows/95 Operating System (OS), the IBM OS/2 operating system, the MAC OS, or UNIX operating system. Those skilled in the art will appreciate that the present invention may also be implemented on other platforms and operating systems.

As stated previously, the present invention uses accessor objects to provide low level data control while still allowing predefined classes to enforce essential data storage and retrieval routines.
Figure 3 is an illustration showing a data storage and retrieval system 300 in accordance with one embodiment of the present invention. The data storage and retrieval system 300 includes a predefined class VARIABLE_CLASS 302, and an accessor class ACCESSOR_CLASS 304. Class VARIABLE_CLASS 302 includes data retrieval and storage methods GetValue 306 and SetValue 308. Similarly, class ACCESSOR_CLASS 304 includes data retrieval and storage methods getActualValue 310 and setActualValue 312.

Referring to class VARIABLE_CLASS 302, methods GetValue 306 and SetValue 308 include calls to methods getActualValue 310 and setActualValue 312 of class ACCESSOR_CLASS 304. In addition, methods GetValue 306 and SetValue 308 also include calls to data checking function Data_Check 314. Function Data_Check 314 represents routines that a creator of class VARIABLE_CLASS 302 desires to enforce in later created subclasses of class VARIABLE_CLASS 302, such as data checking routines, or default value routines. Finally, methods GetValue 306 and SetValue 308 are preferably declared “final.” By declaring methods GetValue 306 and SetValue 308 “final,” the compiler will not allow methods GetValue 306 and SetValue 308 to be overloaded by subsequent subclassing of class VARIABLE_CLASS 302. These functions thus remain constant, and therefore allow enforcement of essential routines called within them, in further subclasses of parent class VARIABLE_CLASS 302.

Referring next to class ACCESSOR_CLASS 304, methods getActualValue 310 and setActualValue 312 are data retrieval and storage methods that are capable of being overloaded by creating subclasses of class ACCESSOR_CLASS 304. These methods may be overloaded by developers to suit their data retrieval and storage needs. For example, methods getActualValue 310 and setActualValue 312 may be designed to read and write to an array of data, an internal storage location of an object, or to storage located on a server accessed via the Internet, as described in greater detail subsequently.

In use, objects of class ACCESSOR_CLASS 304 are utilized to actually access and retrieval data. For example, object AGE 316 is an object of type VARIABLE_CLASS 302, and object Accessor 318 is an object of type ACCESSOR_CLASS 304. Object A 320 is an object that uses methods GetValue 306 and SetValue 308 of object AGE 316 to store and retrieve data.

By using methods GetValue 306 and SetValue 308, function Data_Check 314 is always executed whenever data is stored or retrieved by object A 320. As stated previously, function Data_Check 314 represents routines that a creator of class VARIABLE_CLASS 302 desires to
enforce in later created subclasses of class VARIABLE_CLASS 302, such as data checking routines, or default value routines.

In addition to calling function Data_Check 314, whenever a call is made to method GetValue 306, GetValue 306 calls method getActualValue 310 of object Accessor 318. Method getActualValue 310 is the method that actually retrieves data from a physical memory location on the system, for example from data array 322. Since, both method getActualValue 310 and function Data_Check 314 are called from method GetValue 306, function Data_Check 314 will always be executed whenever getActualValue 310 is executed.

Similar to method GetValue 306, whenever a call is made to method SetValue 308, SetValue 308 calls method setActualValue 312 of object Accessor 318. Method setActualValue 312 is the method that actually stores data to a physical memory location on the system, for example to data array 322. Since, both method setActualValue 312 and function Data_Check 314 are called from method SetValue 306, function Data_Check 314 will always be executed whenever setActualValue 310 is executed.

It should be borne in mind that the methods getActualValue 310 and setActualValue 312 are not declared “final,” and therefore may be overloaded by subclassing class ACCESSOR_CLASS 304 and redefining them. Hence, later developers can redefine the way in which data is stored and accessed by simply overloading methods getActualValue 310 and setActualValue 312 with methods that retrieve and store data differently, such as using different data types, or storage locations. Thus, the creator of class VARIABLE_CLASS 302 can enforce necessary routines, such as data checking routines, or default value routines, while still allowing later developers to redefine the way in which data is stored and accessed.

For example object B 324 is another object that uses methods GetValue 306 and SetValue 308 of object AGE 316 to store and retrieve data. However, by creating object Accessor 326 and overloading methods getActualValue 310 and setActualValue 312 with new data retrieval and storage methods, a later developer can have data accessed, for example, from a server 330 across the Internet 328.

Figure 4 is a flowchart illustrating the manner in which the accessor object 300 manages data associated with a variable in accordance with one embodiment of the present invention. As shown, in operation 400, a variable is provided with an accessor object 300 assigned thereto.
Such accessor object 300 includes a code segment dictating a manner in which data associated with the variable is managed in the memory 304.

Next, in operation 402, the accessor object assigned to the variable is invoked upon a variable being called. Thereafter, the data associated with the variable is handled in memory as prescribed by the accessor object. Note operation 404. By utilizing the accessor object, the underlying data exchange in memory may be maintained uniform while various desired methods may be executed during the data exchange. This allows external programmable flexibility without sacrificing stability that results from uniformity while interfacing with the memory storage mechanism.

It should be noted that the data may be handled in any desired manner which may be programmable, or even changed when particular code is executed. For example, the handling of the data associated with the variable may include storing or retrieving the data in the memory. As an option, the data associated with the variable may be stored and retrieved to or from memory over a network. Such network may include a wide area network such as the Internet and employ a protocol such as TCP/IP and/or IPX.

In one embodiment, the variable may be accessed by a graphics-based rule driven application program. One example of such application program, Stagecast Creator, may be found on the Internet at http://www.stagecast.com. Such application program may allow the same to be reprogrammed, thus particularly necessitating the aforementioned accessor object for stability purposes.

In addition to the foregoing examples, the handling may include constraining a value of the data in the memory, or mapping an undefined value of the data to a defined value in the memory. Additional detailed examples of various ways the data may be managed will be set forth during reference to Figures 5-14.

The accessor object may be implemented as an object held by the variable. Completely new accessor objects can be implemented, subclasses of existing accessor objects can be created, or the default accessor object may be utilized. The accessor object may be set once, at the creation of the variable, or dynamically changed during the variable’s lifetime, according to the requirements of the system. Examples of data storage and retrieval methods of accessor objects will now be set forth.
Figure 5 is a flowchart illustrating one example of retrieving data associated with a variable by way of a getValue method 500. This flowchart illustrates the techniques currently in use by Stagecast Creator. As shown, in operation 502, a getActualValue method is invoked for a predetermined owner. Such operation 502 renders a new value in a manner that will be set forth in greater detail in Figure 7.

Next, in decision 504, it is determined whether the new value is unbound. In the present description, whether or not a value is unbound depends on the getDirectValue method of the accessor. If the accessor determines that there is no known value for the variable and owner, it returns the special token UNBOUND. If the value rendered by the getActualValue method 502 is unbound, a mapUnboundValue method is invoked for the present variable and the present owner. Note operation 506. As mentioned earlier, the various methods may be different for each variable and owner.

By invoking the mapUnboundValue method 506, a new value is rendered which overwrites the new value generated in operation 502. Thereafter, the new value rendered by invoking the mapUnboundValue method is returned in operation 508. Further details relating to the manner in which the mapUnboundValue method is invoked will be set forth hereinafter in greater detail with reference to Figure 9.

If, on the other hand, it is determined that the new value generated by the getActualValue method is not unbound, or bound, in decision 504, the new value rendered by invoking the getActualValue method is returned in operation 508.

Figure 6 is a flowchart illustrating an example of a way of storing, or setting, data associated with a variable in accordance with a setValue method 600 of the present invention. As shown, in operation 602, a constrainValue method is invoked for a predetermined owner. Such operation 602 renders a new value that overwrites the value associated with the current variable in a manner that will be set forth in greater detail in Figure 10.

Next, in decision 604, it is determined whether the new value is illegal. Whether or not the new value is illegal depends on the constrainDirect method of the accessor. If the accessor determines that the value is not legal, it returns the special token ILLEGAL. If so, the process terminates in a return operation 606. If, on the other hand, the new value is determined to not be illegal, or legal, in decision 604, another method, setActualValue method, is invoked for the present variable and the present owner in operation 608.
It is then determined in decision 610 whether the value rendered by invoking the setActualValue method 608 is unbound. If so, in operation 612 a getActualValue method is invoked for the current variable, and the appropriate current owner in order to render a value that overwrites the value generated by invoking the setActualValue method in operation 608. Additional information relating to invoking of the getActualValue method of operation 612 will be forth in greater detail in Figure 7.

Thereafter, in decision 614, it is determined whether the current owner has a parent. As mentioned earlier, a parent is an entity that subclasses the owner. If it is determined that a parent exists, a return operation 616 follows.

If no parent exists in decision 614, it is then determined if the variable, actualValue, associated with the owner is unbound in decision 618. If not, the return operation 616 follows. If so, however, the setValue method is again invoked recursively for the current particular parent, owner, and value. Note operation 620.

Figures 7-10 are flowcharts illustrating examples of how various accessor objects are invoked to handle data associated with various other variables required during the methods of Figures 5 and 6. Figure 7 illustrates the method associated with invoking the getActualValue method 700 which includes obtaining the accessor object, Accessor getDirectValue, of the current variable in 702. Next, the accessor object, Accessor getDirectValue, is invoked in operation 704 for the current particular variable and owner. Once invoked, the accessor object, Accessor getDirectValue, returns a new value in operation 706. Additional information on the manner in which the value is rendered by the accessor object, Accessor getDirectValue, will be set forth in greater detail during reference to Figure 12.

Figure 8 illustrates the method associated with invoking the setActualValue method 800 which includes obtaining the accessor object, Accessor setDirectValue, of the current variable in 802. Next, the accessor object, Accessor setDirectValue, is invoked in operation 804 for the current particular variable and owner after which a return operation 806 ensues.

Figure 9 illustrates the method associated with invoking the mapUnboundValue method 900 which includes obtaining the accessor object, Accessor mapUnboundDirect, of the current variable in 902. Next, the accessor object, Accessor mapUnboundDirect, is invoked in operation 903 for the current particular variable, owner, and value. Once invoked, the accessor object, Accessor mapUnboundDirect, returns a new value in operation 904.
Figure 10 illustrates the method associated with invoking the constrainValue method 1002 which includes obtaining the accessor object, Accessor constrainDirect, of the current variable in 1004. Next, the accessor object, Accessor constrainDirect, is invoked in operation 1006 for the current particular variable and owner. Once invoked, the accessor object, Accessor constrainDirect, returns a new value in operation 1008.

Figures 11-14 illustrate flowcharts showing methods carried out by the accessor objects upon being invoked in the processes of Figures 7-10. Figure 11 illustrates the method associated with the accessor object, Accessor setDirectValue, 1100. As shown, it is first decided in decision 1102 whether the current variable has a value table. If not, a value table is created in operation 1104. In the present description, a value table includes a database of owner, value pairs.

If it is decided in decision 1102 that a value table exists, it is then determined in decision 1106 whether the value is unbound. If so, a value table entry is removed for the owner in operation 1108. If not, a value is put into the value table for the owner in operation 1110. The method associated with the accessor object, Accessor setDirectValue, is then terminated in operation 1112.

Figure 12 illustrates the method associated with the accessor object, Accessor getDirectValue, 1200. As shown, it is first decided in decision 1202 whether the current variable has a variable table. In the present description, a variable table includes a database of owner, value pairs. If not, a return unbound operation 1204 is performed.

If it is decided in decision 1202 that a variable table exists, it is then determined in decision 1206 whether an entry exists for the current owner. If not, the return unbound operation 1204 ensues. If an entry in the variable table exists for the current owner, however, the entry for the owner in the value table is assigned as the value, as indicated in operation 1208. The value is then returned in operation 1210.

Figure 13 illustrates the method associated with the accessor object, Accessor mapUnBoundDirect, 1300. As shown, it is first decided in decision 1302 whether the current owner has a parent. If not, a return operation 1304 is performed which returns a default value of the variable.

If it is decided in decision 1302 that the current owner has a parent, the getActualValue method is invoked for the current owner and parent, thus rendering a value in operation 1306. It
is then determined in decision 1308 whether the value generated by invoking the getActualValue method is unbound. If so, the return operation 1304 is executed which returns the default value of the variable. If the value rendered by invoking the getActualValue method is not unbound, or bound, it is returned in operation 1310.

Figure 14 illustrates the method associated with the accessor object, Accessor constrainDirect, 1400. Such method simply includes returning the current value of the variable in operation 1402. The default implementation of constrainDirect never returns the token ILLEGAL. However, other accessors may. For example, an accessor that constrains the value to a positive integer would examine the input value, if it was a positive integer it would return the value passed in, if it were a negative integer, or a string, or some other object, it would return the ILLEGAL token. Note also that the constrainDirect method may also modify the value. For example, instead of the procedure described previously, a positive integer accessor might examine the input value and truncate any floating point numbers to integers, then return 0 for all negative values, and return ILLEGAL only for a non-numeric value.

While various embodiments have been described above, it should be understood that they have been presented by way of example only, and not limitation. Thus, the breadth and scope of a preferred embodiment should not be limited by any of the above described exemplary embodiments, but should be defined only in accordance with the following claims and their equivalents.
What is claimed is:

1. A method for managing data associated with a variable, comprising:
   (a) providing a variable with an accessor object assigned thereto;
   (b) invoking the accessor object assigned to the variable; and
   (c) handling data associated with the variable in memory as prescribed by the accessor object.

2. A method for managing data associated with a variable as recited in claim 1, wherein handling the data associated with the variable includes storing the data in the memory.

3. A method for managing data associated with a variable as recited in claim 1, wherein handling the data associated with the variable includes retrieving the data from the memory.

4. A method for managing data associated with a variable as recited in claim 1, wherein handling the data associated with the variable includes constraining a value of the data in the memory.

5. A method for managing data associated with a variable as recited in claim 1, wherein handling the data associated with the variable includes mapping an undefined value of the data to a defined value in the memory.

6. A method for managing data associated with a variable as recited in claim 1, wherein the accessor object is programmable.

7. A method for managing data associated with a variable as recited in claim 6, wherein the accessor object is programmable for at least one of retrieving, generating, and computing the data.
8. A method for managing data associated with a variable as recited in claim 1, wherein handling the data associated with the variable includes at least one of storing and retrieving the data to or from memory over a network.

9. A method for managing data associated with a variable as recited in claim 1, wherein the variable is accessed by a graphics-based rule driven application program.

10. A computer program embodied on a computer readable medium for managing data associated with a variable, comprising:
(a) a code segment for providing a variable with an accessor object assigned thereto;
(b) a code segment for invoking the accessor object assigned to the variable; and
(c) a code segment for handling data associated with the variable in memory as prescribed by the accessor object.

11. A computer program for managing data associated with a variable as recited in claim 10, wherein the code segment for handling the data associated with the variable includes a code segment for storing the data in the memory.

12. A computer program for managing data associated with a variable as recited in claim 10, wherein the code segment for handling the data associated with the variable includes a code segment for retrieving the data from the memory.

13. A computer program for managing data associated with a variable as recited in claim 10, wherein the code segment for handling the data associated with the variable includes a code segment for constraining a value of the data in the memory.

14. A computer program for managing data associated with a variable as recited in claim 10, wherein the code segment for handling the data associated with the variable includes a code segment for mapping an undefined value of the data to a defined value in the memory.

15. A computer program for managing data associated with a variable as recited in claim 10, wherein the accessor object is programmable.
16. A computer program for managing data associated with a variable as recited in claim 15, wherein the accessor object is programmable for at least one of retrieving, generating, and computing the data.

17. A computer program for managing data associated with a variable as recited in claim 10, wherein handling the data associated with the variable includes at least one of storing and retrieving the data to or from memory over a network.

18. A computer program for managing data associated with a variable as recited in claim 1, wherein the variable is accessed by a graphics-based rule driven application program.

19. A memory system for managing data associated with a variable, comprising:
   a data structure stored in the memory, the data structure including:
   a variable object with an accessor object assigned thereto; and
   an accessor object invoked in response to the variable object being accessed, the accessor object adapted for handling data associated with the variable in memory as prescribed by the accessor object.
START

PROVIDING A VARIABLE WITH AN ACCESSOR OBJECT ASSIGNED THERE TO

INVOKING THE ACCESSOR OBJECT ASSIGNED TO THE VARIABLE

HANDLING DATA ASSOCIATED WITH THE VARIABLE IN MEMORY AS PRESCRIBED BY THE ACCESSOR OBJECT

END

FIG. 4
VARIABLE getValue FOR OWNER

INVOKE getValue FOR OWNER RESULTING IN NEW VALUE

IS NEW VALUE UNBOUND?

YES

INVOKE mapUnboundValue FOR THIS VARIABLE, OWNER, NEW VALUE OVERWRITING NEW VALUE

NO

RETURN NEW VALUE

FIG. 5
**FIG. 6**

- Variable `setValue` for `OWNER, VALUE` (600)
  - Invoke `constrainValue` for `OWNER, VALUE OVERWRITING VALUE` (602)
    - Is `VALUE ILLEGAL`? (604)
      - Yes: Return (606)
      - No: Invoke `setValue` for `THIS VARIABLE, OWNER, VALUE` (608)
  - Is `VALUE UNBOUND`? (610)
    - Yes: Invoke `getActualValue` for `THIS VARIABLE, OWNER OVERWRITING VALUE` (612)
    - No: Does `OWNER HAVE PARENT`? (614)
      - Yes: Return (616)
      - No: Is `actualValue FOR OWNER UNBOUND`? (618)
        - No: Invoke `setValue` for `PARENT, OWNER, VALUE` (620)
        - Yes: Return (616)
GET THIS VARIABLE'S ACCESSOR

INVOKING ACCESSOR
getDirectValue FOR THIS VARIABLE, OWNER RESULTING IN NEW VALUE

RETURN NEW VALUE

FIG. 7

VARIABLE setValue
setActualValue FOR OWNER, VALUE

GET THIS VARIABLE'S ACCESSOR

INVOKING ACCESSOR
setDirectValue FOR THIS VARIABLE, OWNER, VALUE

RETURN

FIG. 8
FIG. 9

VARIABLE mapUnboundValue
FOR OWNER, VALUE

GET THIS VARIABLE'S ACCESSOR

INVOKES ACCESSOR
mapUnboundDirect FOR THIS VARIABLE, OWNER, VALUE
RESULTING IN NEW VALUE

RETURN NEW VALUE

FIG. 10

VARIABLE constrainValue
FOR OWNER, VALUE

GET THIS VARIABLE'S ACCESSOR

INVOKES ACCESSOR
constrainDirect FOR THIS VARIABLE, OWNER, VALUE
RESULTING IN NEW VALUE

RETURN NEW VALUE
StandardDirectAccessor
setDirectValue FOR VARIABLE, OWNER, VALUE

1100

1102

DOES VARIABLE HAVE VALUE TABLE?

YES

1106

IS VALUE UNBOUND?

YES

1108

REMOVE VALUE TABLE ENTRY FOR OWNER

NO

PUT VALUE INTO VALUE TABLE FOR OWNER

RETURN

1104

CREATE VALUE TABLE

1110

1112

FIG. 11
StandardDirectAccessor
getDirectValue FOR VARIABLE, OWNER

1202
DOES VARIABLE TABLE EXIST?

YES

1206
DOES OWNER ENTRY EXIST IN TABLE?

NO

RETURN UNBOUND

YES

PUT ENTRY FOR OWNER IN VALUE TABLE INTO VALUE

RETURN VALUE

FIG. 12
StandardDirectAccessor
mapUnboundDirect FOR VARIABLE, OWNER

1300

DOES OWNER HAVE A PARENT?

1302

NO
RETURN VARIABLE'S DEFAULT VALUE.

1304

YES

INVOKE getActualValue FOR OWNER, PARENT PUTTING RESULT INTO VALUE

1306

IS VALUE UNBOUND?

1308

YES

RETURN VALUE

1310

FIG. 13

StandardDirectAccessor
constrainDirect FOR VARIABLE, OWNER, VALUE

1400

RETURN VALUE

1402

FIG. 14

SUBSTITUTE SHEET (RULE 26)