US 20050125805A1

(19) United States

(12) Patent Application Publication (10) Pub. No.: US 2005/0125805 A1

Hasha (43) Pub. Date: Jun. 9, 2005

(54) METHOD AND SYSTEM FOR IMPLEMENTING VIRTUAL FUNCTIONS OF AN INTERFACE

(75) Inventor: **Richard Hasha**, Seattle, WA (US)

Correspondence Address:
WOODCOCK WASHBURN LLP
ONE LIBERTY PLACE - 46TH FLOOR
PHILADELPHIA, PA 19103 (US)

(73) Assignees: **William H. Gates III**; **Microsoft Corporation**, Redmond, WA

(21) Appl. No.: **11/039,982**

(22) Filed: **Jan. 20, 2005**

Related U.S. Application Data

(63) Continuation of application No. 10/639,108, filed on Aug. 12, 2003, now Pat. No. 6,886,155, which is a continuation of application No. 09/322,965, filed on May 28, 1999, now Pat. No. 6,704,924.

(60) Provisional application No. 60/118,668, filed on Feb. 3, 1999.

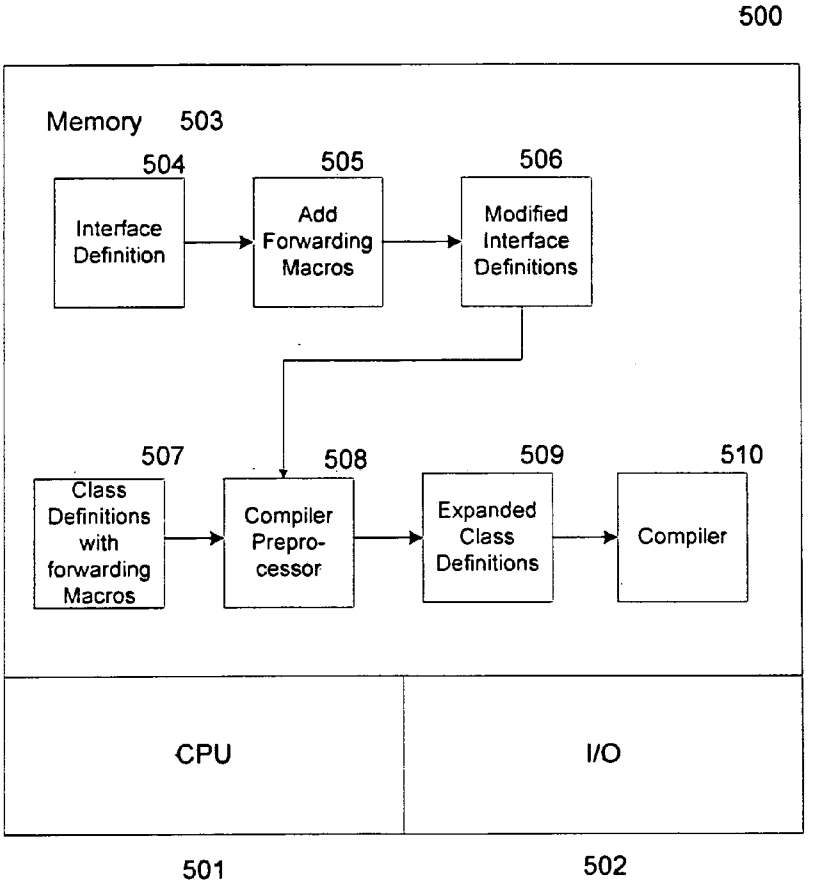
Publication Classification

(51) **Int. Cl.**[7] **G06F 9/46**

(52) **U.S. Cl.** **719/315**

(57) **ABSTRACT**

A method and system for implementing functions in a class that inherits an interface and that inherits an implementing class which implements the interface. A forwarding system adds to the class for each virtual function a forwarding implementation of that virtual function. The forwarding implementation forwards its invocation to the implementation of that the virtual function in the implementing class. The forwarding system implements a special forwarding instruction that specifies the interface and implementing class. A developer of a class that inherits the interface and the implementing class inserts the forwarding instruction into the class definition. When the forwarding system encounters such an instruction during compilation of the class definition, the forwarding system provides an implementation of each virtual function of the interface that forwards its invocation to a corresponding virtual function in the implementing class. The forwarding system also forwards virtual functions of any direct or indirect base interface of the interface to the implementing class.



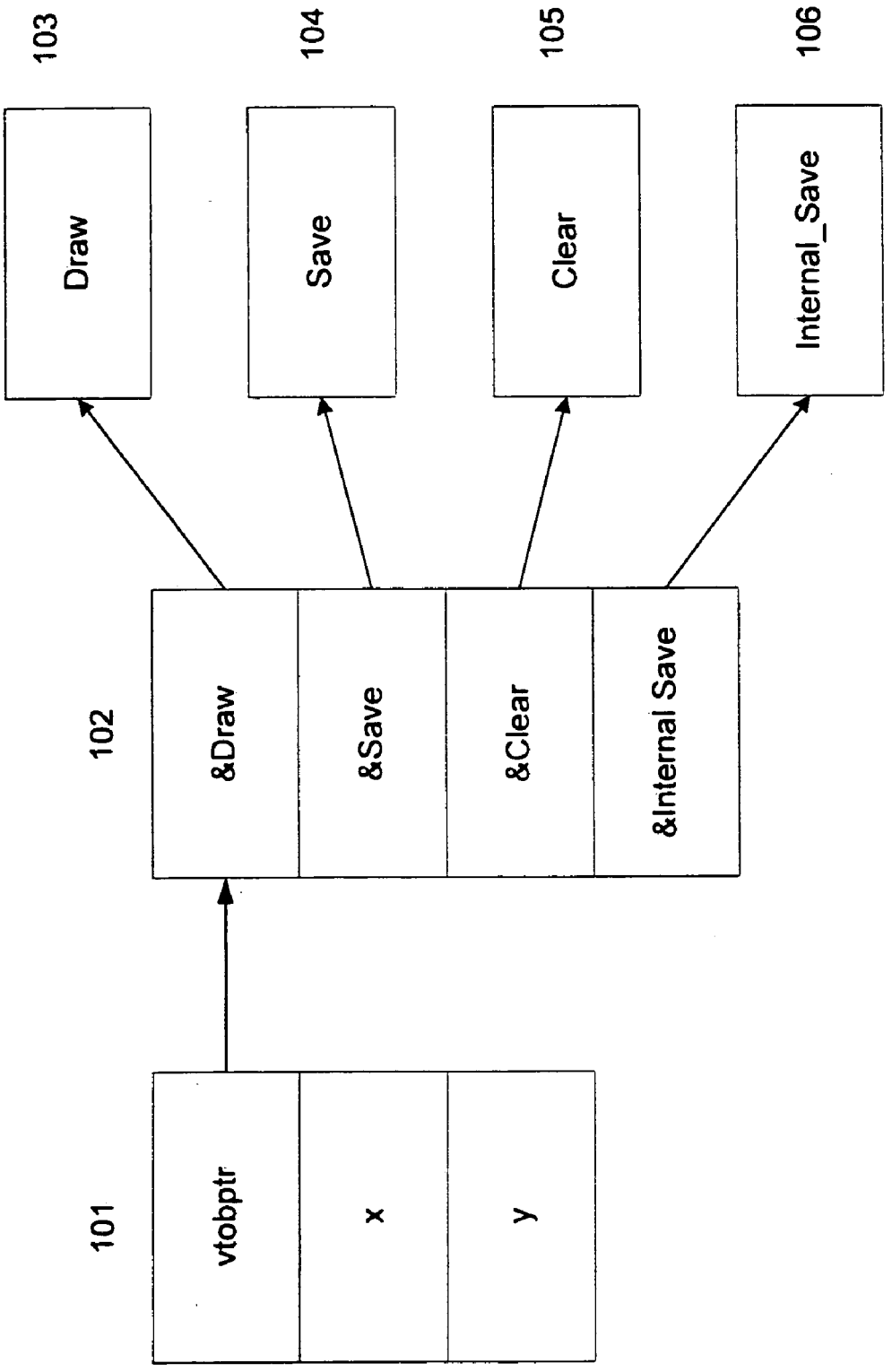

103
104
105
106
Draw
Save
Clear
Internal_Save
102
&Draw
&Save
&Clear
&Internal Save
101
vtobptr
x
y

Figure 1

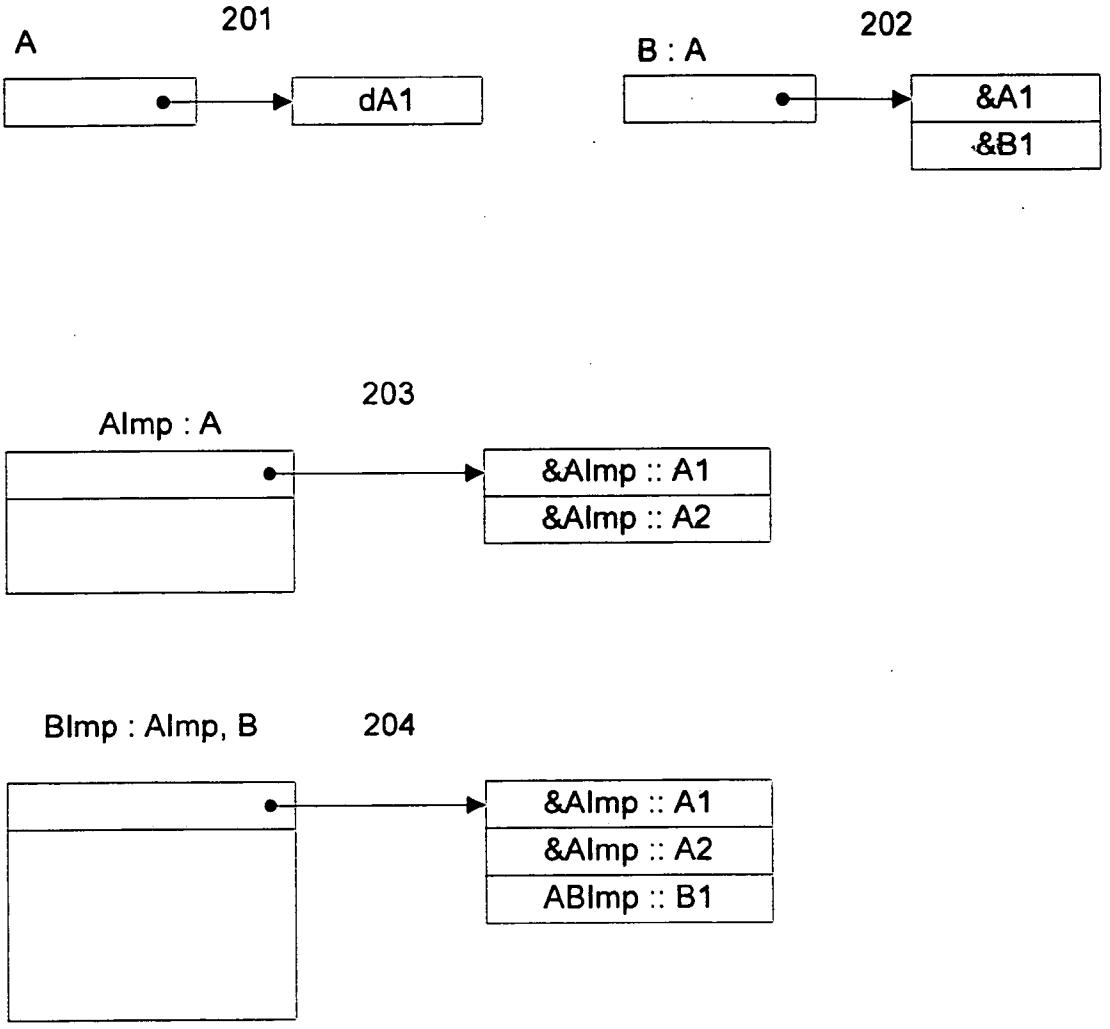

A
201
dA1
B : A
202
&A1
&B1
203
AImp : A
&AImp :: A1
&AImp :: A2
BImp : AImp, B
204
&AImp :: A1
&AImp :: A2
ABImp :: B1

Figure 2

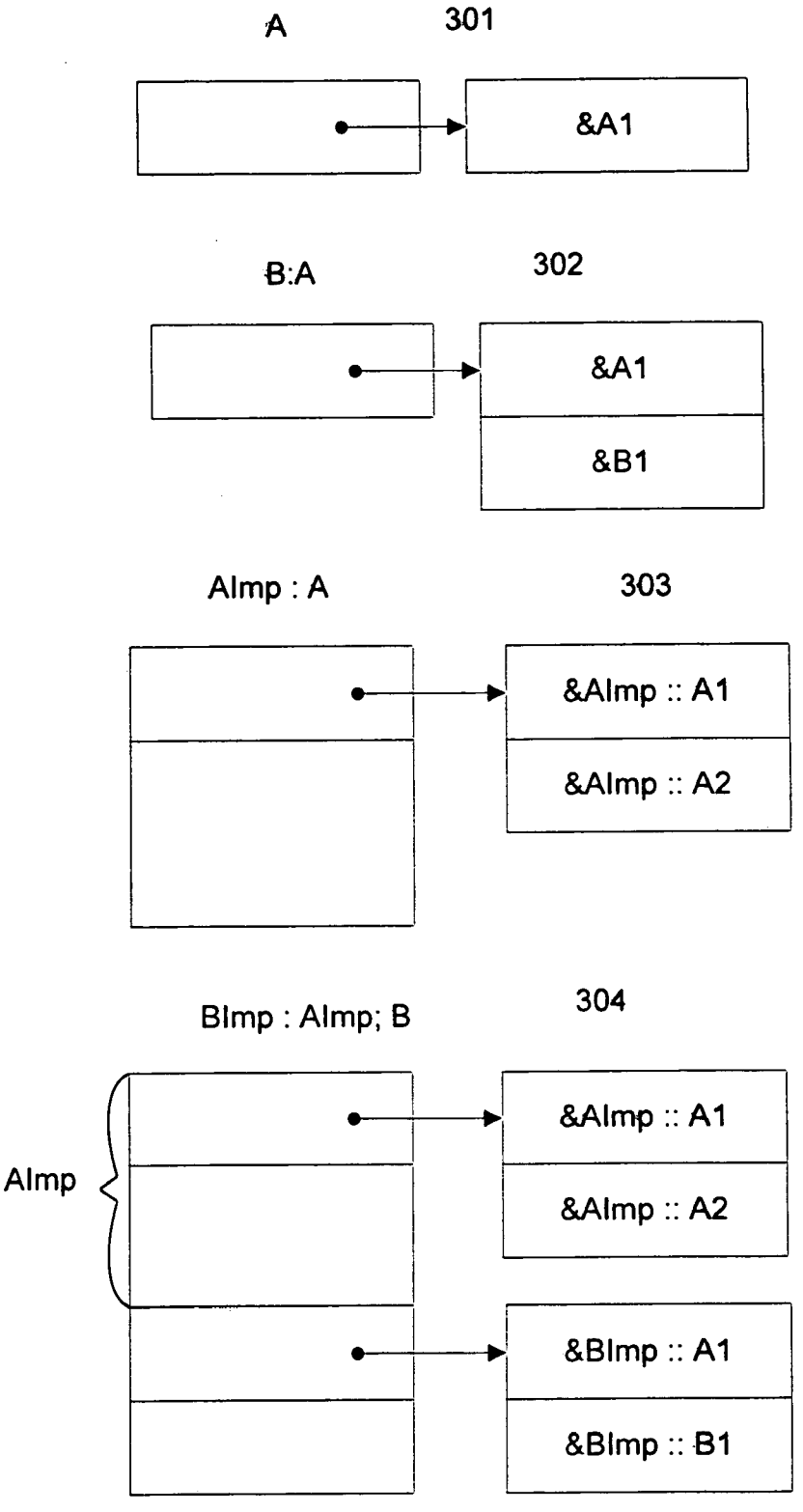

A
301
&A1
B:A
302
&A1
&B1
AImp : A
303
&AImp :: A1
&AImp :: A2
BImp : AImp; B
304
AImp
&AImp :: A1
&AImp :: A2
&BImp :: A1
&BImp :: B1

Figure 3

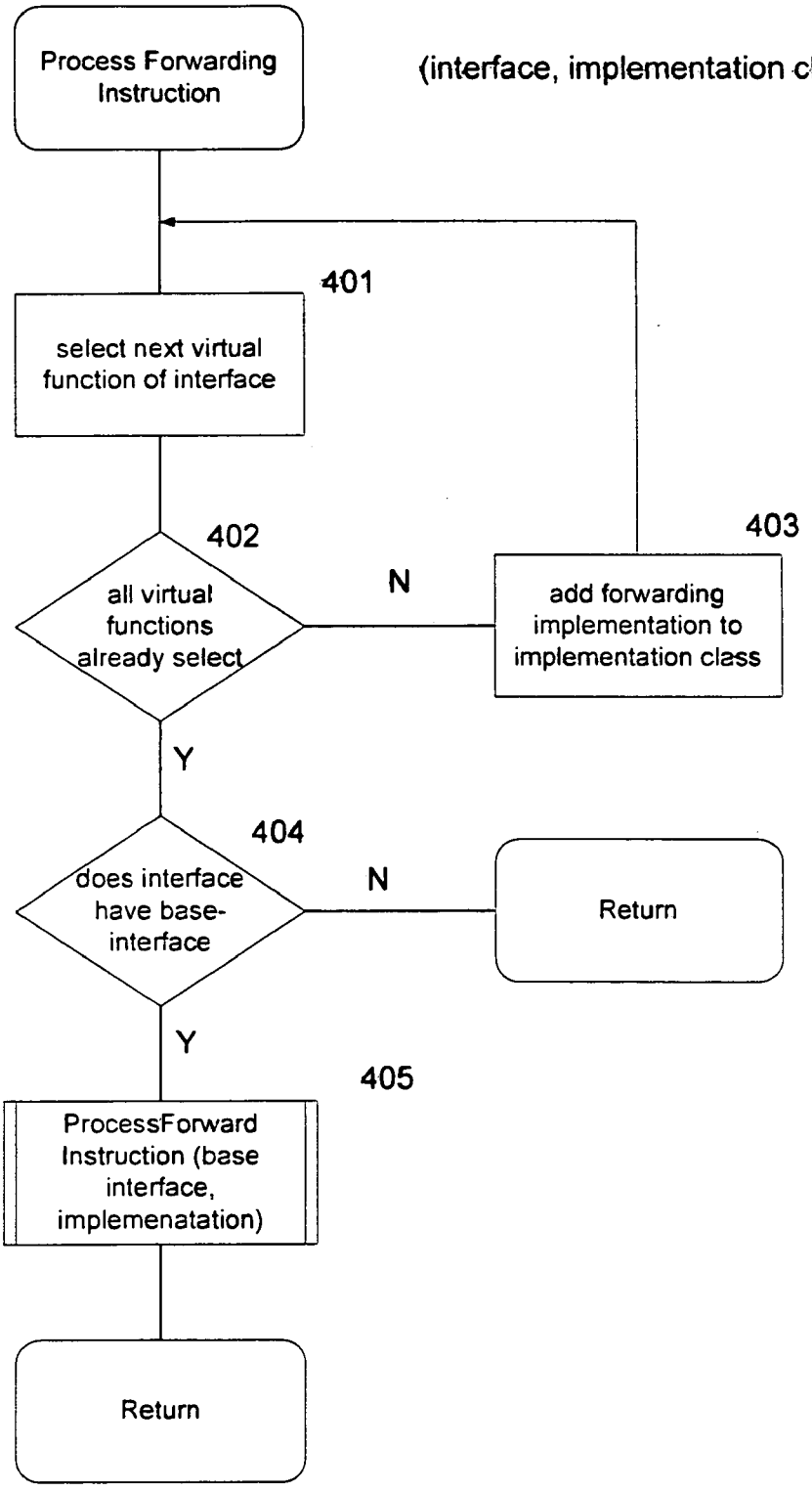

Process Forwarding Instruction
(interface, implementation class)
401
select next virtual function of interface
402
all virtual functions already select
N
403
add forwarding implementation to implementation class
Y
404
does interface have base-interface
N
Return
Y
405
ProcessForward Instruction (base interface, implemenatation)
Return

Figure 4

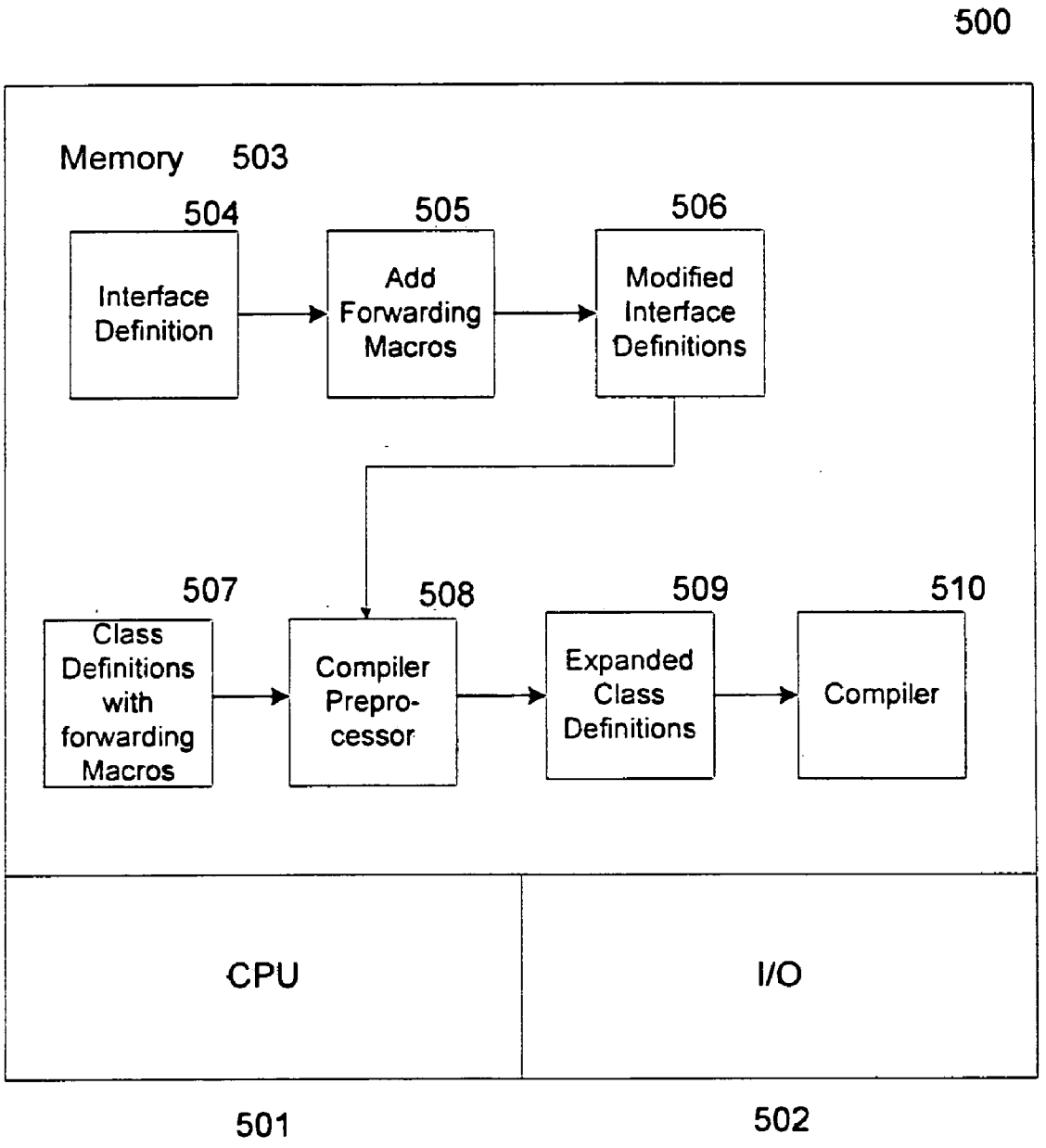

500
Memory 503
504
Interface Definition
505
Add Forwarding Macros
506
Modified Interface Definitions
507
Class Definitions with forwarding Macros
508
Compiler Prepro-cessor
509
Expanded Class Definitions
510
Compiler
CPU
I/O
501
502

Figure 5

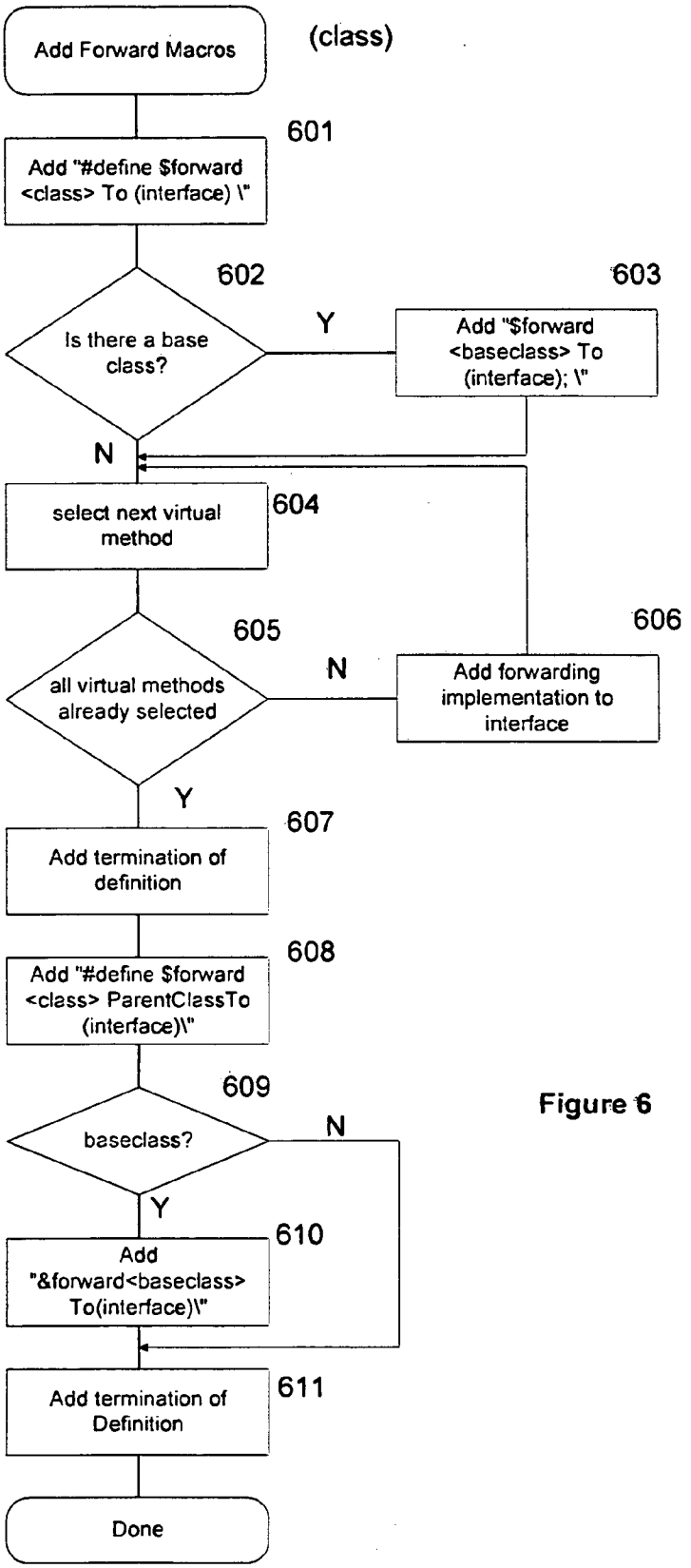

Add Forward Macros
(class)
601
Add "#define $forward <class> To (interface) \"
602
Is there a base class?
Y
603
Add "$forward <baseclass> To (interface); \"
N
select next virtual method
604
605
all virtual methods already selected
N
606
Add forwarding implementation to interface
Y
607
Add termination of definition
608
Add "#define $forward <class> ParentClassTo (interface)\"
609
baseclass?
N
Y
610
Add "&forward<baseclass> To(interface)\"
Add termination of Definition
611
Done

Figure 6

METHOD AND SYSTEM FOR IMPLEMENTING VIRTUAL FUNCTIONS OF AN INTERFACE

[0001] FOR CONTROLLING ENVIRONMENTAL CONDITIONS," filed on May 28, 1999, which has matured into U.S. Pat. No. 6,466,234, the disclosures of which are incorporated herein by reference.

FIELD OF THE INVENTION

[0002] The described technology relates generally to object-oriented programming techniques and more specifically to a system for automatically implementing virtual functions.

BACKGROUND OF THE INVENTION

[0003] Object-oriented programming techniques allow for the defining of "interfaces" by which objects can expose their functionality independently of the implementation of the functionality. In the C++ programming language, an interface is defined by an abstract class whose virtual functions are all pure. A pure virtual function is one that has no implementation in the class. Thus, an interface defines only the order of the virtual functions within the class and the signatures of the virtual functions, but not their implementations. The following is an example of an interface:

```
class IShape
{
    virtual void draw(int x,y)=0;
    virtual void save(char filename)=0;
    virtual void clear(int x,y)=0;
}
```

[0004] This interface, named "IShape," has three virtual functions: draw, save, and clear. The "=0" after the formal parameter list indicates that each virtual function is pure. Concepts of the C++ programming language that support object-oriented programming are described in "The Annotated C++ Reference Manual," by Ellis and Stroustrup, published by Addison-Wesley-Publishing Company in 1990, which is hereby incorporated by reference.

[0005] Once an interface is defined, programmers can write programs to access the functionality independent of the implementation. Thus, an implementation can be changed or replaced without having to modify the programs that use the interface. For example, the save function of the IShape interface may have an implementation that saves the shape information to a file on a local file system. Another implementation may save the shape information to a file server accessible via the Internet.

[0006] To ensure that an implementation provides the proper order and signatures of the functions of an interface, the class that implements the interfaces inherits the interface. The following is an example of a class that implements the IShape interface.

```
class Shape: IShape
{
    virtual void save(char filename){. . .};
    virtual void clear(int x,y){. . .);
    virtual void draw(int x,y) {. . .};
    virtual void internal_save( ){. . .);
    int x;
    int y;
}
```

[0007] The first line of the class definition indicates by the ": IShape" that the Shape class inherits the IShape interface. The ellipses between the braces indicate source code that implements the virtual functions. The Shape class, in addition to providing an implementation of the three virtual functions of the IShape interface, also defines a new virtual function "internal_save," which may be invoked by one of the implementations of the other virtual functions. The Shape class also has defined two integer data members, x and y.

[0008] Typical C++ compilers generate virtual function tables to support the invocation of virtual functions. When an object for a class is instantiated, such a C++ compiler generates a data structure that contains the data members of the object and that contains a pointer to a virtual function table. The virtual function table contains the address of each virtual function defined for the class. **FIG. 1** illustrates a sample object layout for an object of the Shape class. The object data structure **101** contains a pointer to a virtual function table and the data members x and y. The virtual function table **102** contains an entry for each virtual function. Each entry contains the address of the corresponding virtual function. For example, the first entry in the virtual function table contains the address of the draw function **103**. The order of the references in the virtual function table is the same as defined in the inherited interface even though the Shape class specifies these three functions in a different order. In particular, the reference to the draw function is first, followed by the references to the save and clear functions.

[0009] The C++ compiler generates code to invoke virtual functions indirectly through the virtual function table. The following is C++ code for invoking a virtual function:

[0010] Shape*pShape=new Shape;

[0011] pShape-> save{);

[0012] The pointer pShape points to an object in memory that is of the Shape class. When compiling this invocation of the virtual function, the compiler generates code to retrieve the address of the virtual function table from the object, to add to the retrieved address the offset of the entry in the virtual function table that contains the address of the function, and to then invoke the function at that calculated address.

[0013] The inheritance of interfaces allows for references to objects that implement the interfaces to be passed in an implementation independent manner. A routine that uses an implementation may define a formal argument that is a pointer to the IShape interface. The developer of the routine can be unaware that the implementation is actually the Shape class. To pass a reference to an object of the Shape class, a program that invokes the routine would type cast a pointer to the object of the Shape class to a pointer to the IShape interface. So-long as the pointer points to a location

that contains the address of the virtual function table and the virtual function table contains the entries in the specified order, the invoked routine can correctly access the virtual functions defined by the IShape interface:

[0014] Although the use of interfaces facilitates the development of programs that use and implement interfaces, difficulties do arise. The following C++ source code illustrates one of these difficulties.

```
class A
{
    virtual void A1( )=0;
}
class B : A
{
    virtual void B 1( )=0;
}
class AImp : A
{
    virtual void A1( ){. . .};
    virtual void A2( ){. . .};
}
class BImp : AImp, B (
{
    virtual void B1( ){. . .};
}
```

[0015] In this example, classes A and B are interfaces. Interface A defines virtual function A1, and interface B defines virtual function B1. Class AImp is an implementation of interface A and defines a new virtual function A2. Class BImp is an implementation of interface B that inherits the implementation of interface A, which is class AImp. Thus, class BImp inherits multiple classes: class AImp and interface B.

[0016] Some C++ compilers have the option of merging the virtual function tables of multiple base classes into a single virtual function table in the derived class or of generating separate virtual function tables for each base class. One difficulty arises if the virtual function tables of the base classes are merged in the derived class. **FIG. 2** illustrates the difficulty when the virtual function tables are merged. The object layouts **201** and **202** illustrate the order of the virtual functions within the virtual function tables. (Since interfaces A and B are abstract classes, objects of these classes cannot actually be instantiated.) To allow type casting as described above, an implementation of interface B needs to provide a virtual function table with entries in the same order as shown in object layout **202**, that is with the entry for virtual function A1 followed immediately by the entry for virtual function B1. A problem arises when the implementation of interface A, which is class AImp, defines a new virtual function, such as virtual function A2. The virtual function table of class AImp is shown by object layout **203**. Object layout **204** illustrates the virtual function table for class BImp when the virtual function tables are merged. This virtual function table for class BImp illustrates the order of the entries to be virtual functions A1, A2, and B1. If a pointer to an object of class AImp is type cast to a pointer to interface B, then the virtual function table would not have the entries in the order as specified by the interface B. In particular, the entry for virtual function A1 is not immediately followed by the entry for virtual function B1. A C++ compiler could have reordered in entries in the virtual function table within object layout **204** to be virtual functions A1, B1, and A2. In which case, the type casting for interface B would work. However, the type casting from a pointer to a class BImp object to a class AImp object would be incorrect because the class AImp defines that the entry for virtual function A1 is immediately followed by the entry for virtual function A2. Thus, a C++ compiler cannot define a single virtual function table that satisfies the virtual function table ordering of both interface B and class AImp.

[0017] Although the use of the separate option for generating virtual function tables alleviates this difficulty, additional difficulties arise. **FIG. 3** illustrates the object layouts with separate virtual function tables. In this example, the object layout **304** for class BImp has two virtual function tables. The first virtual function table has the order of entries to be virtual functions A1 and A2, which is consistent with the ordering for class AImp. The second virtual function table has the order of entries to be virtual functions A1 and B1, which is consistent with the ordering for interface B. The difficulty arises because the developer of class BImp, who wants to reuse the implementation of interface A, which is class AImp, must provide an implementation for each virtual function of interface A within class BImp. (These multiple implementations can be avoided by virtually inheriting interface A. With such virtual inheritance, a C++ compiler might generate no virtual function tables with the required ordering of the virtual functions.) The developer of class BImp can implement each virtual function of interface A in a straightforward manner. In particular, the developer can implement each virtual function to forward its invocation to the corresponding virtual function in the implementation of interface A, which is class AImp. The following is an example implementation of class BImp:

```
Class BImp : AImp, B
{
    virtual void B 1( ){. . .};
    virtual void A 1( ){ return AImp::A1( )}
}
```

[0018] The implementation of virtual function A1 invokes the implementation of virtual function A1 in the class AImp. Although this implementation is straightforward, problems arise when interface A is changed. For example, if a new virtual function A3 is added to interface A, then the developer of the class BImp would need to change the class definition to include an implementation of the new virtual function A3. Although the changing of the class definition in this example may not be difficult, the changing of class definitions in systems that use complex inheritance hierarchies and many virtual functions can present difficulties. It would be desirable to have a technique which would avoid the difficulties of having to change class definitions when a new virtual function is added to an interface.

SUMMARY OF THE INVENTION

[0019] A method and system for implementing functions in a class that inherits an interface and that inherits an implementing class which implements the interface is provided. In one embodiment, a forwarding system adds to the class for each virtual function a forwarding implementation of that virtual function. The forwarding implementation

forwards its invocation to the implementation of that virtual function in the implementing class. In one embodiment, the forwarding system implements a special forwarding instruction that specifies the interface and implementing class. A developer of a class that inherits the interface and the implementing class inserts the forwarding instruction into the class definition. When the forwarding system encounters such an instruction during compilation of the class definition, the forwarding system provides an implementation of each virtual function of the interface that forwards its invocation to a corresponding virtual function in the implementing class. The forwarding system also forwards virtual functions of any direct or indirect base interface of the interface to the implementing class.

BRIEF DESCRIPTION OF THE DRAWINGS

[0020] **FIG. 1** illustrates a sample object layout for an object of the Shape class.

[0021] **FIG. 2** illustrates the difficulty when the virtual function tables are merged.

[0022] **FIG. 3** illustrates the object layouts with separate virtual function tables.

[0023] **FIG. 4** is a block diagram of an example implementation of the forwarding system.

[0024] **FIG. 5** is a block diagram illustrating a computer system for practicing the forwarding system.

[0025] **FIG. 6** is a flow diagram illustrating an example implementation of a routine to automatically add forwarding macros to interface definitions.

DETAILED DESCRIPTION OF ILLUSTRATIVE EMBODIMENTS

[0026] Embodiments of the present technology provide a method and system for automatically generating implementations of virtual functions of interfaces. The system generates the implementations so that each virtual function forwards its invocation to a virtual function that provides an implementation. In one embodiment, the forwarding system implements a special forwarding instruction that specifies an interface and an implementing class. A developer of a class that inherits the interface and the implementing class inserts the forwarding instruction into the class definition. When the forwarding system encounters such an instruction during compilation of the class definition, the forwarding system provides an implementation of each virtual function of the interface that forwards its invocation to a corresponding virtual function in the implementing class. The forwarding system also forwards virtual functions of any direct or indirect base interface of the interface to the implementing class. Alternatively, the forwarding system automatically detects that no implementation has been provided for the interface and inserts a forwarding implementation to the implementing class.

[0027] In one embodiment, the forwarding system is implemented by adapting a compiler to recognize a special forwarding instruction and to automatically insert and implement the virtual functions. Alternatively, the forwarding system may be implemented using standard macros that are preprocessed by a compiler, such as a C++ compiler. The macro implementation of the forwarding system is referred to as the "macro system." The macro system defines a forwarding macro for each interface that provides an implementation of each virtual function that forwards the invocation of the virtual function to an implementation of that virtual function in a class whose name is passed as a parameter to the macro. The following illustrates the forwarding macro that is defined for class A:

```
class A
{
    virtual void A1( )=0;
    #define $forwardATo(interface)\
    virtual void A 1( ) { return interface##::A1( )};
}
```

[0028] The "#define" indicates that a macro is being defined, the "$forwardATo" is the name of the macro, and the "interface" is the name of a parameter that is passed when the macro is expanded. The body of the macro is "virtual void A1( ){return interface##::A1( )};" where "interface##" indicates that the passed parameter is to be substituted. The macro definition is terminated by a statement that does not end with a "". In other words, a "" at the end of a statement within the macro indicates that the macro continues onto the next line. When this macro is expanded the parameter such as "Imp," the result is:

[0029] virtual void A1( ){return Imp::A1( )};

[0030] which forwards the invocation of virtual function A1 to the implementation of virtual function A1 in class Imp.

[0031] The implementation of interface B, which is class BImp, can call the macro formardATo passing the name of the implementation of interface A, which is class AImp. When the macro is expanded, an implementation of virtual function A1 is provided that forwards its invocation to the implementation of class AImp. The following illustrates the use of that macro:

```
class BImp : AImp, B
{
    virtual void B1( ){. . .};
    $forwardATo(AImp)
}
```

[0032] The following illustrates the definition of class BImp when the macro is expanded:

```
class BImp : AImp, B
{
    virtual void B1( ){. . .};
    virtual void A1( ){return AImp::A1( )};
}
```

[0033] If new virtual functions are added to interface A, the $forwardATo macro is redefined to add statements that are forwarding implementations of the new virtual functions. Because class BImp calls that macro, when the macro is expanded, the definition of class BImp will include

forwarding implementations of the new virtual functions. Thus, once a developer adds a forwarding macro to class BImp modifications to interface A are automatically reflected in class BImp.

[0034] The definition of such a forwarding macro needs to include statements that will generate forwarding implementations of each virtual function in inherited interfaces. The following illustrates the forwarding macro for interface B that inherits interface A:

```
class B : A
{
    virtual void B1( )=0;
#define $forwardBTo(interface)\
    $forwardATo(interface)\
    virtual void B1( ){ return interface##::B1( )};
}
```

[0035] An implementing class that inherits interface B need only expand the forwarding macro for interface B, which automatically expands the forwarding macro for interface A. The following illustrates an implementation of interface C, which is class CImp, that uses the forwarding macro of interface B:

```
class C : B
{
    virtual void C1( )=0;
}
class CImp : BImp, C
{
    virtual void C1( ){. . .} ;
    $forwardBTo(BImp)
}
```

[0036] The following illustrates expansion of the forwarding macro in class CImp:

```
class CImp : BImp, C
{
    virtual void C1( ){. . .};
    virtual void A1( ){return BImp::A1( ));
    virtual void B1( ){return BImp::B1( )};
}
```

[0037] A difficulty with the naming convention of these forwarding macros is that an implementation needs to know of any directly inherited interface of its directly inherited interface. For example, the implementation of interface C, which is class CImp, needs to know that interface B is directly inherited by interface C so that the appropriate forwarding macro, $forwardBTo," can be used. One embodiment of the macro system overcomes this difficulty by defining an additional parent class forwarding macro for each interface that inherits an interface. The parent class forwarding macro has a name that is based solely on the interface for which it is defined, but calls the forwarding macro for its directly inherited interfaces. The following definition of interface B illustrates the parent class forwarding macro:

```
class B : A
{
    virtual void B1( )=0;
#define $forwardBTo(interface)\
    $forwardATo(interface)\
    virtual void B1( ){return interface##::B1( )};
#define $forwardBParentClassTo(interface)\
    $forwardATo(interface)
}
```

[0038] The parent class forwarding macro is "$forwardBParentClassTo." This macro calls the forwarding macro for its inherited interface A. An implementing class can call the parent class forwarding macro of its directly inherited interface rather than calling the forwarding macro of an indirectly inherited interface. Thus, the implementation need not be aware that an inherited interface happens to inherit another interface. The following illustrates the use of the parent class forwarding macro:

```
class BImp : AImp, B
{
    virtual void B1( ){. . .};
    $forwardBParentClassTo(AImp)
}
```

[0039] The expansion of the parent class forwarding macro is illustrated in the following:

```
class BImp : AImp, B
{
    virtual void B1( ){. . .};
    virtual void A1( ){return AImp::A1( ));
}
```

[0040] More generally, any interface regardless of whether it currently inherits another interface can provide an implementation of the parent class forwarding macro. If the interface does not currently inherit another interface, then the parent class forwarding macro would be defined to be empty. Thus, when such a parent class forwarding macro is expanded, no statements are inserted into the calling class definition and the expansion would have no effect. If, however, the interface is eventually changed to inherit another interface, then the parent class forwarding macro would be redefined to call the forwarding macro of the inherited interface. The change in the parent class forwarding macro would automatically be reflected when the parent class forwarding macro is expanded in the calling class definition.

[0041] **FIG. 4** is a block diagram of an example implementation of the forwarding system. In this implementation, the forwarding system may be implemented as part of a compiler or as a preprocessor to a compiler. The process forwarding instruction routine is invoked when the forwarding system encounters a special forwarding instruction, which specifies an interface and an implementing class or automatically detects such a situation. The routine inserts,

into the class definition that inherits the interface and that inherits the implementing class, a forwarding implementation for each virtual function of the interface. The routine then recursively calls itself to insert forwarding implementations for virtual functions of any indirectly inherited interfaces of the interface. In steps **401-403**, the routine loops inserting forwarding implementations for each virtual function of the interface. In step **401**, the routine selects the next virtual function of the interface. In step **402**, if all the virtual functions have already been selected, then the routine continues at step **404**, else the routine continues at step **403**. In step **403**, the routine inserts a forwarding implementation of the selected virtual function that forwards its invocation to the implementing class. In step **404**, if the interface has an inherited interface (i.e., a base interface), then the routine continues at step **405**, else the routine returns. In step **405**, the routine recursively invokes the process forwarding instruction routine passing the base interface and the implementing class. The routine then returns.

[**0042**] **FIG. 5** is a block diagram illustrating a computer system for practicing the forwarding system. The computer system **500** includes a central processing unit **501**, I/O devices **502**, and memory **503**. The I/O devices may include computer readable media, such as a hard disk or a flexible disk, on which components of the forwarding system may be stored. The forwarding system that is stored in memory is the macro system. The forwarding system comprises an add forwarding macro component **505**. The add forwarding macro component inputs interface definitions **504** and outputs modified interface definitions **506**. The add forwarding macro component inserts the forwarding macro and the parent class forwarding macro into the interface definitions. The compiler preprocessor **508** inputs the modified interface definitions **506** and class definitions that have calls to the macros **507**. The compiler preprocessor expands the macros within the class definitions to generate expanded class definitions **509**. The compiler **510** inputs the expanded class definitions to generate the compiled code.

[**0043**] **FIG. 6** is a flow diagram illustrating an example implementation of a routine to automatically add forwarding macros and parent class forwarding macros to interface definitions. This routine is passed an interface definition and inserts macro definitions for the forwarding macro and the parent class forwarding macro. In step **601**, the routine inserts a statement "#define $forward<class>To(interface)\" into the interface definition. This statement starts the definition of the forwarding macro. The "<class>" is replaced by the name of the interface. In step **602**, if the interface inherits a base interface, then the routine continues at step **603**, else the routine continues at step **604**. In step **603**, the routine inserts the statement "$formardcbaseclass>To(interface);". The "<baseclass>" is replaced by the name of the inherited interface. This statement calls the forwarding macro of the inherited interface. In steps **604-606**, the routine loops adding forwarding implementations for each virtual function of the interface. In step **604**, the routine selects the next virtual function. In step **605**, if all the virtual functions have already been selected, then the routine continues at step **607**, else the routine continues at step **606**. In step **606**, the routine adds the forwarding implementation of the selected virtual function and loops to step **604** to select the next virtual function. In step **607**, the routine adds a termination for the macro definition. The termination may be accomplished by inserting a blank line. In the step **608**, the routine adds the statement "#define $forward<class>ParentClassTo(interface)". This statement defines the parent class forwarding macro. In step **609**, if the interface inherits a base interface, then the routine continues its step **610**, else the routine continues that step **611**. In step **610**, the routine adds the statement "Sforward<baseclass>To(interface)". This statement calls the forwarding macro of the base interface. If, however, the interface inherits no base interface, then the parent class forwarding macro is empty. In step **611**, the routine adds a termination of the macro definition and completes.

[**0044**] From the foregoing it will be appreciated that, although specific embodiments of the invention have been described herein for purposes of illustration, various modifications may be made without deviating from the spirit and scope of the invention. Accordingly, the invention is not limited except as by the appended claims.

What is claimed:

1. A method of operating a computer system, comprising:

providing a class object corresponding to a class of an object oriented programming language comprising a virtual function;

defining an implementation of the virtual function; and

automatically inserting an implementation of the virtual function in an second class object wherein the second class object has an inheritance relationship with the class object.

2. The method as recited in claim 1 wherein the class object is an abstract class.

3. The method as recited in claim 1 wherein the automatic insertion of the implementation is performed by a compiler.

4. The method as recited in claim 1 wherein said automatically inserting includes automatically inserting instructions corresponding to the implementation of the virtual function at a location of an indicator within the second object.

5. A computer-readable medium bearing computer executable instructions for carrying out the acts of:

providing a class object comprising a virtual function;

defining an implementation of the virtual function; and

automatically inserting an implementation of the virtual function in a second class object wherein the second class has an inheritance relationship with the class object.

6. The computer-readable medium as recited in claim 5 wherein the class object is an abstract class.

7. The computer-readable medium as recited in claim 5 wherein the automatic insertion of the implementation is performed by a compiler.

8. The computer-readable medium as recited in claim 5 wherein said automatically inserting includes automatically inserting instructions corresponding to the implementation of the virtual function at a location of an indicator included within the second object.

* * * * *