Class Exerciser: a Basic CASE Tool for Object-Oriented Development

Chien-Min Wang and Y. S. Kuo
Software Methodology Lab., Institute of Information Science, Academia Sinica
Taipei, Taiwan, ROC

Abstract

Given a set of classes as input, a class exerciser, with its easy-to-use graphical user interface, allows users to create objects for the given classes, invoke methods on specified objects and examine the contents of objects interactively. In other words, one can use a class exerciser to exercise and assess the various functions of classes conveniently. The class exerciser has many applications in object-oriented software development. In particular, it can be used as a tool for demonstration, testing and maintenance all together. We also describe our design and implementation of a class exerciser, RunClass, for the programming language C++.

1. Introduction

Object-oriented programming offers a number of novel features, such as abstraction, encapsulation, inheritance and polymorphism [GR][Mey]. It has emerged as one of the most important software development paradigms. Just as any other software development paradigms, object-oriented programming demands a powerful CASE (computer-aided software engineering) environment in support of software development and maintenance [You]. Then, what CASE tools are required for object-oriented software development and maintenance? A simple answer to this is to build, for each traditional CASE tool, an object-oriented counterpart. For instance, while there are design (analysis) tools supporting structured design (analysis), there should be design (analysis) tools built to support object-oriented design (analysis). Most CASE vendors are taking this approach. However, is this approach adequate? Does object-oriented programming demand any special CASE tools on its own behalf? This has motivated our exploration of new CASE tools for object-oriented development particularly.

In object-oriented programming, a class is a template that describes the behavior of a set of objects. It is the unit for abstraction and encapsulation. An object-oriented application is naturally decomposed into a collection of classes. Thus, the class can serve as a natural unit for software reuse. In fact, object-oriented development is characterized by the development and use of a large set of reusable classes, referred to as class libraries [GR][Boo]. Current object-oriented CASE environments facilitate software reuse by providing tools to inspect and access class libraries [Gol]. Using these CASE tools, one can easily locate desired classes and retrieve information of interest. However, on the other hand, invocation of classes (including instantiation of objects from classes and invocation of methods on objects) can only be made in the form of traditional programming. In other words, one has to write an application program in order to invoke a class, in general. (Other means of programming, such as visual programming and 4GL, are still restricted to special cases.) This is fine if the application program really wants to use the class. However, there are many other situations where people just want to invoke a class independent of any specific application. As an example, one may want to invoke a class to see what it does in the following cases. In one case, the class was developed by a software vendor, but its specification is not thoroughly described in the manual. (This seems the usual case.) One wants to find out what the class does on some undocumented input conditions. In another case, the class was newly developed. Invocation of the class provides a way to verify its correctness. For all such cases, an interactive, easy-to-use environment for "exercising" classes appears more convenient than the traditional edit-and-run programming environment. To meet this requirement, we have thus developed a new CASE tool, the class exerciser.

Given a set of classes as input, a class exerciser allows users to create objects for the given classes, execute methods on specified objects and examine the contents of objects interactively. In other words, a class exerciser allows users to exercise the various functions of classes and see the results. By preparsing the declarations of classes, a class exerciser can present an easy-to-use graphical user interface to users with class names and method names displayed on the workstation screen. Users can then select desired classes and methods with a pointing device without memorizing the names. A class exerciser is also responsible for the management of objects. Whenever an object is created, a user-specified name is associated with the object. For users' convenience, objects are grouped according to classes. Like class names and method names, object names are

0-8186-7171-8/95 $04.00 © 1995 IEEE

displayed on the screen for easy selection by users. With all these features, a class exerciser will then appear as a specialized environment for easy exercising of classes.

CASE tools normally fall within two categories [FM]: Vertical tools serve for one specific software-process stage, e.g. design tools. Horizontal tools are used throughout the entire software process, e.g. project management tools. The class exerciser does not fit into this classification. It is a general tool useful for the back-end process stages, including programming, testing and maintenance, all together. A class exerciser can vividly demonstrate to users (programmers, in this case) the functions of classes. Thus it can serve as a programming aid, supplementing manuals for class libraries. With its ability of executing methods on objects and exhibiting execution results, a class exerciser can be used as a testing tool for unit-test of classes. A class exerciser can help users understand the behavior of previously developed classes. Thus it can serve as a vital tool for object-oriented software maintenance. With all these applications, we will thus treat a class exerciser as a basic CASE tool indispensable to object-oriented development.

This paper is organized as follows: In the next section, some related tools and systems are reviewed. In Section 3, we deliberate on some possible applications of the class exerciser. Then, in Sections 4 and 5, we describe our design and implementation of a prototype class exerciser, RunClass, for the programming language C++ [Str]. Even though being a prototype, RunClass is fairly real with its capability of handling the complex constructs of C++. Finally, we will make some concluding remarks in the last section.

2. Related Work

In this section, we distinguish the class exerciser from other related tools or systems.

A widely-used CASE tool for object-oriented development particularly is the class browser [Gol]. A class browser exhibits the inheritance relationships among classes, which are static in nature. In contrast, a class exerciser demonstrates the run-time behavior of classes. The class exerciser and the class browser are complementary to each other. In a good CASE environment, users should be given the freedom and ease to switch between seeing the static class hierarchies and seeing the run-time behavior of classes.

A class exerciser possesses some characteristics of an interpreter. Like an interpreter for an object-oriented programming language, a class exerciser accepts messages as input and executes associate methods in response to input messages. However, there are two fundamental differences: On the one hand, a class exerciser does not possess the full capacity of an interpreter, e.g. it does not allow users to define new classes at run-time. In fact, a class exerciser can be treated as a specialized interpreter with limited scope. Just targeting for a specialized application domain, a class exerciser can provide users with a customized, easy-to-use graphical user interface while a conventional interpreter is so general that it must accept text commands as input and sacrifice ease of use. On the other hand, while using a class exerciser to invoke methods in a given class, the given class was precompiled rather than being interpreted by the class exerciser. This has the following advantages over complete interpretation: First, execution of the class exerciser does not require the given class's complete source code, but just the declaration part of the source code. This is very important considering that the given class may be developed by a software vendor so that its executable source code is not available. Second, compilation is more efficient than interpretation. Third, when the given class is, later on, used by an application program, it is compiled rather than interpreted (for languages such as C++). Only by invoking the compiled code can the class exerciser demonstrate true invocation of the input class.

An automated test driver[Mye][Pan] is a generic test program that feeds test-case inputs to a module-under-test. A class exerciser can be deemed as a special automated test driver built for testing classes particularly. An automated test driver typically supports a script language for describing test cases and test procedures while a class exerciser accepts test-case inputs in interactive mode. Some automated test drivers can verify the results of test cases while a class exerciser relies on the user to verify the test results. The importance of an automated test driver for classes has been recognized in the area of object-oriented testing [DF][SR]. TOBAC [SN] is a test management environment for Smalltalk programs which includes a test execution tool like RunClass.

3. Possible Applications of the Class Exerciser

A class exerciser has many possible applications in object-oriented development. In particular, one can use a class exerciser for demonstration, testing and maintenance all together.

3.1 Class Exerciser as a Demonstration Tool

As object-oriented programming gets popular, many software vendors provide a wide variety of class libraries for distribution in the public. However, the current situation appears that there have been more class libraries built, yet less in wide distribution. One of the reasons for these class libraries being not in wide distribution resides in the lack of tools for demonstration of class libraries. Traditionally, software vendors provide programming manuals and on-line help for exposition of a software library. These text-based documents tend to be too boring, insufficient, and sometimes even misleading. Some vendors thus provide special application programs for demonstration of their products. However, since these demonstration programs are customized for specific libraries, it is extremely costly for software vendors to build a full set of demonstration programs for all their class libraries. Furthermore, these vendors' demonstration programs are apt to demonstrate the overall functionality of a class library, whereas the class exerciser, being generic for any class library, is able to demonstrate the individual methods of each class separately.

RunClass, the class exerciser developed in our lab., allows users to create objects for input classes, invoke methods on specified objects, and examine the contents of objects interactively. In other words, with its well-designed graphical user interface, RunClass can make a vivid demonstration of the various functions of the input classes under users' command. Playing with such a vivid demonstration is apparently more a fun than reading text-based documentation. Moreover, by invoking methods from RunClass with suitable parameters, one can easily find out what a class does under some undocumented conditions. With these features, a class exerciser can thus serve as a useful supplement to text-based documentation for class libraries.

From the viewpoint of a library developer, a class exerciser can act as a demonstration tool. On the other hand, from the viewpoint of a (potential) user of a class library, a class exerciser can act as an assessment tool. By exercising the various functions of classes, one can assess the functionality and usefulness of a class library. As project supervisors, the authors have used RunClass to assess some class libraries under development in our lab. Like most other project leaders, we are too busy to do extensive programming work. The class exerciser's easy-to-use, nonprogramming environment has help us a lot in monitoring our projects.

3.2 Class Exerciser as a Testing Tool

In traditional procedure-oriented software development, unit test is centered on testing stand-alone modules like subprograms or functions. While in an object-oriented system, the natural unit for testing is a class [DF][SR][HSS]. However, testing a class is more complex than testing a traditional function or subprogram since the methods in a class can interact with one another.

Since a class simply defines the behavior of objects, the only way to observe the operation of a class is to create an instance (a run-time object) from the class, carry out the methods specified within the class on the object, and then observe the state change of the object. As a consequence, research works on class testing have, more or less, yielded a similar test framework [DF][SR][SN] even though they may adopt different test design strategies. The central part of the framework is a class-based test driver which interprets test-case input commands, and carries out object creation, method execution, etc., in response to input commands. A class exerciser, such as RunClass, can create user-defined objects, execute methods on specified objects, and allow users to inspect the contents of objects. All these features of a class exerciser have made it an appropriate class-based test driver.

The class exerciser differs from most other class-based test drivers (the traditional discipline of testing) in that it operates in interactive mode and does not support a script language for describing test cases. The justification for this lies in that unit testing of a class is normally done by the class developer rather than a specialized test engineer, mainly due to costs [Jac]. Providing a class developer with a flexible, easy-to-use test execution tool appears more practical than expecting him to learn a special script language and follow some disciplined test design methods. The lack of support of a script language for a class exerciser can be complemented by a capture and replay function. (Describing test cases with a script language facilitates retention of test cases for later replay.) For the class exerciser, RunClass, the entire interaction process with the user can be captured for replay in a later time. This allows users to retain old test cases. As the class-under-test evolves, the retained test cases can be used for regression testing.

3.3 Class Exerciser as a Maintenance Tool

Maintenance typically costs more than 70 percent of the total software life cycle. The object-oriented paradigm, despite of its promise for quality and productivity, would not let all maintenance problems go away [WMH]. Software maintenance requires understanding of previously developed programs. It was once reported that understanding occupies more than 50 percent of maintenance cost [FH]. A number of reverse engineering tools [Oma] have been built to help understand previously developed programs. The majority of these tools analyze a program's source code statically in an effort to create some more abstract representations of the program, thus are referred to as static reverse engineering tools. Pressman has pointed out the importance of dynamic reverse engineering tools even though they are still relatively rare [Pre].

A class exerciser allows users to execute separate pieces of an object-oriented program, and inspect the outcome of the execution. This helps users collect run-time information and accumulate knowledge about the program. Thus, the class exerciser can act as a dynamic reverse engineering tool. Brooks [Bro] described a complete knowledge about a program as a succession of knowledge domains that bridge between the problem being solved and the program in execution. He also pointed out that a hypothesis-and-verify process is used by a programmer to reconstruct these domains when they seek to understand a program. Just as a class exerciser can act as a test tool for verifying execution results, it can be used for verifying hypotheses made about a program. This also appears as the best way for a maintainer to reconstruct his knowledge domain about the program in execution. In fact, it is well-known that verifying is an important ingredient of understanding.

Zvegintzov [Zve] has drawn an analogy between software understanding and assembling a jigsaw puzzle. While understanding an object-oriented system, the classes correspond to the pieces of the puzzle. A puzzle is not completed in a big bang; rather, it is assembled piece by piece. A class exerciser allows a maintainer to work out a program class by class. If, eventually, it can be proven that maintaining object-oriented software is easier than maintaining traditional software, the natural class structures of object-oriented software must be the key. The class exerciser allows users to fully exploit the class structures.

Software reuse and software maintenance have much in common [Hal]. In particular, understanding how a previously built program works is essential for both software reuse and software maintenance [Sta][BR]. Just as it can be used for software maintenance, the class exerciser is also useful for software reuse.

4. RunClass --- A Class Exerciser for C++

In this section, we describe a class exerciser, RunClass, for the C++ programming language [Str]. RunClass, driven by interactive users, can create objects at run-time, execute class operations on objects, and display the contents of objects. It provides a friendly graphical user interface so that its users can try any sequence of operations conveniently. RunClass is also capable of capturing operations into a log file and replaying these operations later by reading the log file back. With this ability, RunClass is suitable for use as a demonstration, testing or maintenance tool. Its major features are summarized as follows:

- RunClass is a generic tool, being able to take any C++ class as input.
- No extra learning or programming effort is required for users. RunClass uses C++ header files as input.
- It provides an easy-to-use graphical user interface.
- RunClass can handle not only object-oriented constructs, such as class and method, but also other non-object-oriented constructs of C++, such as function, built-in types (including pointer and array), enum, struct, typedef and template, etc.
- Allow users to create and destroy objects at run-time.
- Allow users to invoke methods and functions in arbitrary order at run-time.
- Manipulate type conversion and argument matching automatically.
- Allow users to apply methods to and display the contents of sub-objects.
- Support the ability to capture operations into a log file and to replay these captured operations.

The first challenge the authors faced in developing RunClass is to make it generic. The C++ programming language is strong-typed so that every class or function must be declared before their use. Since the class exerciser will invoke the compiled code of the classes or functions to be exercised, it must know these classes or functions before its execution. This is impossible if the class exerciser is a single executable module. To overcome this problem, RunClass is composed of two parts: the exerciser and the exerciser generator as shown in Fig. 1. The exerciser is what really interacts with the users while the exerciser generator is a preprocessor that parses C++ header files and generates code for inclusion in the exerciser. Because the header files come directly from the class libraries, this approach has the advantage that there is no extra learning or programming effort for the users.

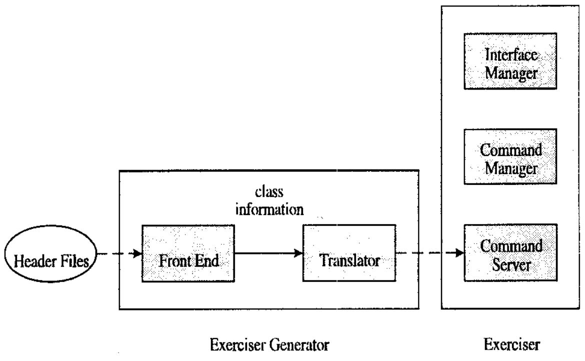


Figure 1 The Components of RunClass

A crucial decision in the design of the exerciser is the differentiation of those modules that are generic from those modules that must be generated by the exerciser generator. Our principle is to minimize the code to be generated, i.e. minimize the coupling between the exerciser and the exerciser generator. After some thought and trial, those modules that must be generated by the exerciser generator are identified. These modules are designated as the command server because most of them are used for executing methods or functions. Another consideration in the design of the exerciser is its portability over different platforms. Our solution is to cluster those input/output modules into an interface manager, which is not portable. The rest of the exerciser is referred to as the command manager. A clean programming interface is defined to separate the command manager from the interface manager. Accordingly, the class exerciser is further divided into the following three components:

- The interface manager, which communicates with the users and controls the progress of the exercising process. It accepts commands from the users or log files and calls the command manager to accomplish these commands.
- The command manager, which interprets and executes user commands. Its main tasks include the management of information about files, classes, and objects; the support of built-in types and functions; the manipulation of argument matching; and the interpretation of commands from the interface manager. The command manager may call the command server for method execution.
- The command server, containing code relevant to the classes declared in the input header files. Its main tasks include the creation and destroy of objects, the invocation of methods of classes with actual arguments, type conversion for arguments and the assignment of objects. The command server may call the actual member functions of classes or access the data members of objects to complete its tasks.

Note that both the interface manager and the command manager are independent of the classes declared in the input header files, i.e. they are generic modules. Only the command server depends on the classes to be exercised and needs to be generated by the exerciser generator.

The exerciser generator is further divided into two parts: the front end and the translator. The front end takes C++ header files as input and extracts from the header files such information as the names of member functions in each class, their parameters, and their types, etc. The information is then passed to the translator. The translator takes the information extracted by the front end as input and generates the command server modules which are then compiled and linked with the generic modules of the exerciser to make the executable code.

5. Implementation of RunClass

We have successfully made a prototype implementation of RunClass. This section describes some of the implementation details, and point out a few less obvious problems in actually getting it to work. In particular, we shall concentrate on the implementation of the exerciser. Fig. 2 shows the structure of the exerciser.

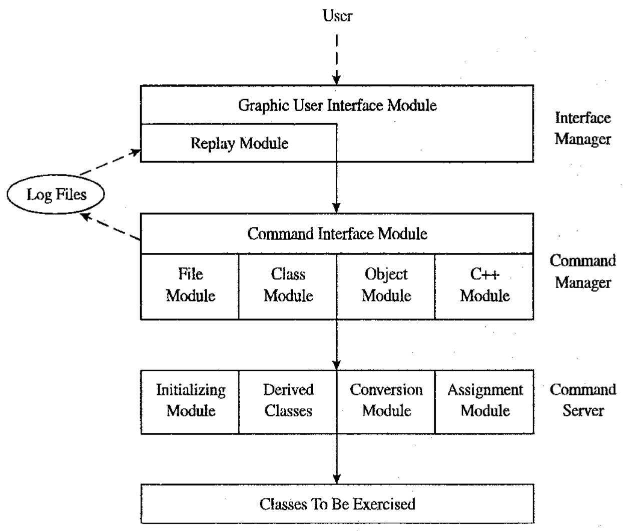
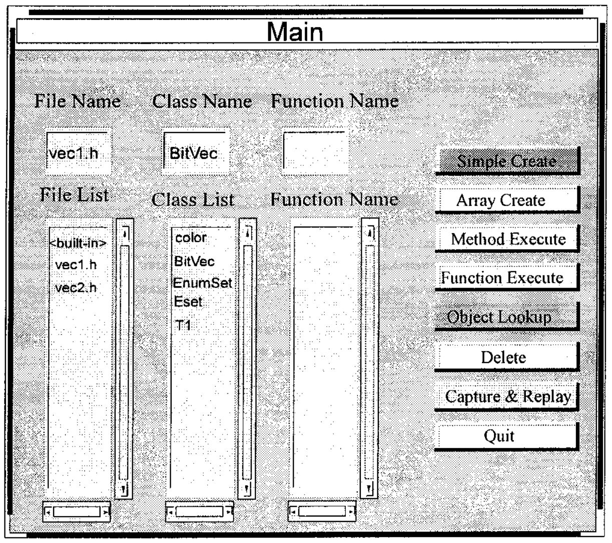


Figure 2: The Structure of RunCLass

5.1 The Interface Manager

Each module in the interface manager is in charge of a certain type of command sources. As shown in Fig. 2, the interface manager currently contains two modules: the graphical user interface (GUI) module and the replay module. The graphical user interface module deals with commands from interactive users. The replay module deals with commands from log files. With this approach, we can take charge of new types of command sources easily by adding more modules to the interface manager. For example, we may have a new module dealing with commands from a pipe coming from some test design tools in the future.

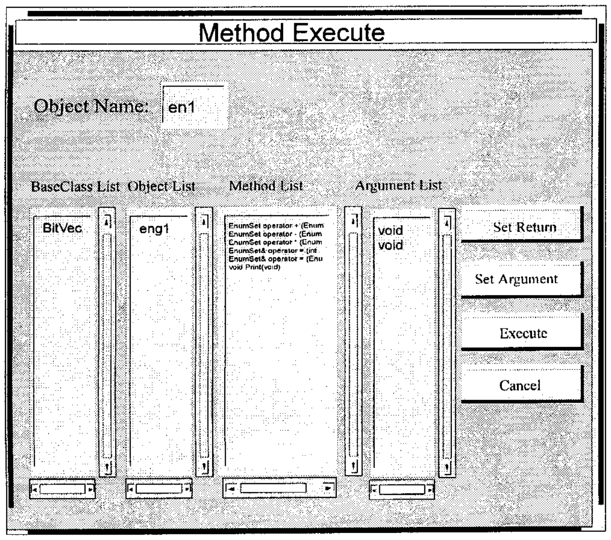
The graphical user interface module is currently implemented on the X-window system. With the graphical user interface, manipulation of information on files, classes, functions, objects, methods, and arguments becomes quite easy and convenient. It also allows users to exercise and inspect pointers, arrays, complex objects and sub-objects. Let's take some examples. Figs. 3 to 5 show three of the menus of the graphical user interface. The first one is the main menu that displays the files, classes and functions declared in the header files. The buttons on the right-hand side of the main menu allow users to control the exercising process. For example, clicking on the button labeled "Method Execute" will pop up the menu in Fig. 4, which allows users to execute member functions of classes. As another example, clicking on the button labeled "Object Lookup" will pop up the menu in Fig. 5, which displays the contents of objects. Users can exercise the various functions of classes simply by clicking the buttons on the screen.



File Name: vec1.h
Class Name: BitVec
Function Name:

File List: <built-in>, vec1.h, vec2.h
Class List: color, BitVec, EnumSet, Eset, T1
Function Name:

Figure 3. The Main Menu



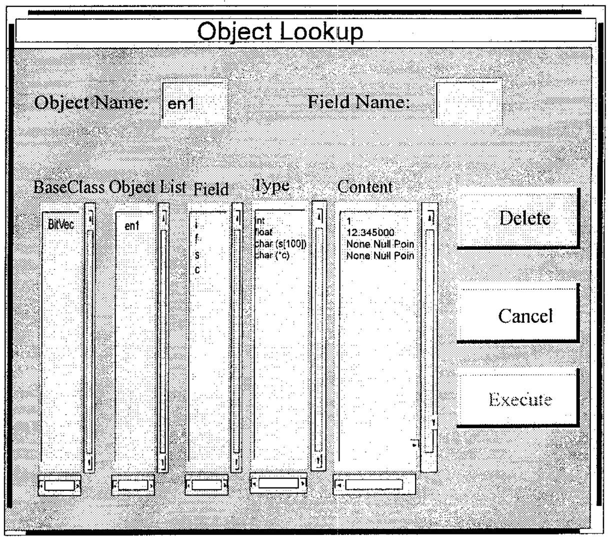
Object Name: en1

BaseClass List: BitVec
Object List: eng1
Method List:
EnumSet operator + (Enum
EnumSet operator - (Enum
EnumSet operator * (Enum
EnumSet& operator *= (int
EnumSet& operator = (Enu
void Print(void)
Argument List:
void
void

Figure 4. Menu For Method Execution

An important feature of the class exerciser is its ability to capture the user operations into a log file and to replay the process automatically in a later time. The replay module of the interface manager carries out the replay function by reading back and interpreting the operations captured on the log file. It then calls the command manager to accomplish the task as usual. On the other hand, the capture function is not carried out by the interface manager; it is carried out by the command manager, instead, which will be discussed later.

We have carefully designed the exerciser so that the interface manager and the command manager interacts through a clean interface. This will enable us to develop different user interfaces for the exerciser while keeping the command manager and the command server unchanged. This property is also desired while porting the class exerciser to different platforms.



Object Name: en1
Field Name:

BaseClass	Object List	Field	Type	Content
BitVec	en1	i	int	1
		f	float	12.345000
		s	char (s[100])	None Null Poin
		c	char (*c)	None Null Poin

Figure 5. Menu for Object Lookup

5.2 Command Manager

As shown in Fig. 2, the command manager contains five modules: the command interface module, the file managing module, the class managing module, the object managing module, and the C++ supporting module. The command interface module is a set of routines to be called by the interface manager. Through calls to these routines, the interface manager requests the command manager to accomplish user commands. For example, a function call to the routine getFileNameList() will return the name list of the input header files. The command interface module may ask other modules in the command manager to accomplish its task. In this example, getFileNameList will call the file managing module to get the filename list. Another task carried out by the command interface module is the capture function mentioned before. While on capture mode, the routines of the command interface module will, in addition to the normal processing of commands, save the commands onto a log file. The capture function, implemented as part of the command manager rather than the interface manager, is independent of the multiple input sources.

The file managing module, the class managing module and the object managing module are in charge of information about files, classes, and objects, respectively. For instance, the file managing module maintains a list of all input header files for use as in the above example. In addition, it also maintains, for each header file, a list of all classes and functions declared in the header file. Users can thus select classes or functions according to the header files where they are declared.

Among all the modules in the exerciser, the class managing module plays a very important role. Its main function is accomplished by class ClassNode. For each input class, there is an associated instance of ClassNode. Information maintained in ClassNode includes the name, parents, data members and member functions of an input class. It also maintains dynamic information, such as the list of objects instantiated from that class. Access to these data is through public member functions as shown in the following class declaration.

```
class ClassNode {
public:  . . .
   int getClassName();
   int getParentList();
   int getDataMemberList();
   int getMemberFunctionList();
   int getObjectList();
   virtual int getObjectContents();
   virtual int executeConstructor();
   virtual int executeDestructor();
   virtual int executeMemberFunction();
};
```

Besides, ClassNode is also responsible for the display of the contents of objects and the execution of member functions. These operations are performed through the last four public virtual functions. However, it should be pointed out that these four functions are not generic in nature since they must call the actual member functions of the input classes. In other words, the implementation of these functions is different for different input classes. We employ inheritance and dynamic binding to overcome this problem. In our implementation, ClassNode is designated as an abstract class. For each input class, there is an associated class derived from ClassNode. The last four functions of ClassNode are redefined in the derived classes. Now function calls to the virtual functions of ClassNode will invoke the associated functions of the derived classes through dynamic binding. Note that both the class managing module and ClassNode are generic. On the other hand, the derived classes of ClassNode must be generated by the exerciser generator as part of the command server. The class managing module also manages inter-class information, such as the inheritance relationships among classes and conversion information among built-in types.

The object managing module maintains information about objects, such as addresses of objects and dimensions and sizes of arrays. The former is used for accessing the actual object while the latter helps the manipulation of pointers and arrays. The object managing module also keeps data structures for finding out the class of a given object. In order to handle more complex expressions, the object managing module is equipped with a parser which can process such C++ expressions as *(a[2].b->c) at run-time.

A critical decision in developing the object managing module is how to store objects. To make the object managing module generic, objects can not be stored in their original types. We thus use void pointers (void *) to access the actual objects. Auxiliary information, such as the original types of objects must be kept so that objects can be restored to their original types whenever necessary.

The C++ supporting module provides several mechanisms in support of the C++ semantics. It performs semantic checks, such as checks for conversion of arguments and checks for assignment of objects. It also supports C++ built-in types and functions. These features make the class exerciser behave consistent with C++.

5.3 Command Server

As shown in Fig. 2, the command server contains four modules: the module for initializing the exerciser, the derived classes of ClassNode, the module for type conversion and the module for object assignment. The initialization module mainly sets up the static information about the files, classes, and objects declared in the header files. It is called as the exerciser starts to execute. These data will then be used by the file managing module, the class managing module, and the object managing module of the command manager.

As mentioned before, the derived classes of ClassNode will invoke the member functions of the input classes. They will also access the data members of objects of the input classes. Now another less obvious problem arises. In C++, the private data members and member functions of a class are not accessible to any other classes. The only exception is for friend classes of that class. However, to allow users to inspect private data members and to invoke private member functions, it is necessary for the derived classes of ClassNode to access the private parts of the input classes. The solution we adopted is to make the derived class associated with an input class a friend of the input class. To accomplish this, some friend declarations are inserted in the original header files. Fortunately, this has no other side effect.

The type conversion module is used for converting objects from their original types to some desired types. The class managing module of the command manager will check whether the desired type conversion is legal or not. If it is legal, the type conversion module will be called to accomplish the conversion. This is desirable since it allows users to supply an actual parameter whose type is not exactly the same as that of the formal argument but can be converted. Care must be taken that type conversion is made between objects pointed to by void pointers (void *). Sometimes, temporary variables may be required in the conversion process. The object assignment module is used for performing similar conversion functions for object assignment.

5.4 Handling Templates

C++ templates are parameterized types. Since they can not be used to instanciate objects, templates themself are not dealt with by RunClass. However, if templates are used to instantiate template classes in the input header files, then the exerciser generator can generate the required server module for the instantiated template classes. As a result, the instantiated template classes can be handled by the exerciser as ordinary classes.

6. Concluding Remarks

The object-oriented paradigm has emerged as one of the most promising software development techniques, with the potential of delivering high-quality software at reduced cost. To fully exploit its power, the object-oriented paradigm demands the support of unconventional CASE tools. The class exerciser is a new CASE tool developed to satisfy this demand.

Object-oriented programming is characterized by the development and use of a large number of class libraries. A class library differs from an application software at least in two fundamental points: First an application software can run by itself while a class library must be invoked by a driver program. Second, unlike an application software which typically accomplishes a single, specific function, a class library provides a large set of functions for use by clients. Due to these two traits of class libraries, it is very hard to demonstrate and assess the functionality of a class library. The class exerciser is developed to overcome this difficulty. By including a generic, easy-to-use driver program, a class exerciser allows users to exercise and assess the various functions of classes. In other words, it somehow converts an intangible class into something which is tangible. With this capability, a class exerciser has many applications in object-oriented software development. In particular, it can be used to aid programming, testing and maintenance all together. Therefore, we would propose the class exerciser as a basic CASE tool indispensable to object-oriented development.

References

[BR] T. Biggerstaff and C. Richter, "Reusability Framework, Assessment, and Directions", IEEE Software, March 1987.

[Boo] G. Booch, Software Components with Ada, Benjamin/Cummings, 1987.

[Bro] R. Brooks, "Using a Behavior Theory of Program Comprehension in Software Engineering", Proc. 3rd International Conference on Software Engineering, 1978.

[DF] R.-K. Doong and P. G. Frankl, "Case Studies on Testing Object-Oriented Programs", Proc. 4th Symp. on Testing, Analysis and Verifacation, New York, 1991.

[FH] R. K. Fjeldstad and W. T. Hamlen, "Application Program Maintenance Study: Report to Our Respondents", Proc. GUIDE 48, Philadelphia, PA, 1979.

[FM] G. Forte and K. McCulley, CASE Outlook: Guide to Products and Services, CASE Consulting Group, Lake Oswego, Ore., 1991.

[Gol] A. Goldberg, Smalltalk-80: The Interactive Programming Environment, Addison-Wesley, 1984.

[GR] A. Goldberg and D. Robson, Smalltalk-80: The Language and its Implementation, Addison-Wesley, 1983.

[Hal] P. A. V. Hall (ed.), Software Reuse and Reverse Engineering in Practice, Chapman & Hall, 1992.

[HSS] D. Hoffman, J. Smillie and P. Strooper, "Automated Class Testing: Methods and Experience", Proc. Asia Pacific Software Engineering Conf., Tokyo, Dec. 1994.

[HC] J. W. Hooper and R. O. Chester, Software Reuse Guide and Methods, Plenum Press, 1991.

[Jac] I. Jacobson, M. Christerson, P. Jonsson and G. Overgaard, Object-Oriented Software Engineering: A Use Case Driven Approach, Addison-Wesley, 1992.

[Mey] B. Meyer, Object-Oriented Software Construction, Prentice Hall, 1988.

[Mye] G. J. Myers, The Art of Software Testing, Wiley, 1979.

[Oma] P. Oman, "Maintenance Tools", IEEE Software, May 1990.

[Pan] D. J. Panzl, "Automatic Software Test Drivers", IEEE

Computer, Vol. 11, No. 4, April, 1978.

[Pre] R. S. Pressman, Software Engineering: A Practitioner' s approach, 3rd edition, McGraw-Hill, 1992.

[SN] E. Siepmann and A. R. Newton, "TOBAC: A Test Case Browser for Testing Object-Oriented Software", Proc. Int. Symp. on Software Testing and Analysis, Seattle, WA, August 1994.

[SR] D. Smith and D. J. Robson, "A Framework for Testing Object-Oriented Programs", J. of Object-Oriented Programming, Vol. 5, No. 3, June 1992.

[Sta] T. A. Standish, "An Essay on Software Reuse", IEEE Transactions on Software Engineering, Vol. SE-10, No. 5, September 1984.

[Str] B. Stroustrups, The C++ Programming Language, 2nd edition, Addition-Wesley, 1991.

[WMH] N. Wilde, P. Matthews and R. Huitt, "Maintaining Object-Oriented Software", IEEE Software, Jan. 1993.

[You] E. Yourdon, Object Oriented System Design: An Integrated Approach, Prentice Hall, 1994.

[Zve] N. Zvegintzov, "Eureka Countdown", Datamation, April 1982.