A system for program component specification and code generation

Robert P. Brazile
University of North Texas
Department of Computer Science

Abstract

This paper describes an interactive system for program specification and generation. It consists of: 1) a methodology for program generation and 2) a special purpose object-oriented database management system(OODBMS). The methodology is based on the object-oriented paradigm and consists of a set of program component classes and the rules for using these classes. The methods defined within the classes are available to the program developer for creating and relating program component objects, as well as generating source programs from the program components. The OODBMS is made up of a user interface, a method interpreter called the generation engine and the object management system.

Introduction

Many ideas for using the object-oriented paradigm to increase programmer productivity have been explored. The concepts have been around in one form or another for a long time, but the understanding and use of these ideas is maturing now. The concepts were implemented first in programming languages[3], and have made their way into data base management systems[5]. An important use of the object-oriented paradigm is in creating tools which assist in the software development process. OPAL[1] is a system designed using object oriented ideas to support application development. A technique for transforming data flow diagrams into object-oriented designs is described in [2]. A design system shell for experimenting with principles of object- oriented design[4] and a database for supporting reusability[6] are other software engineering uses of object-oriented concepts.

We describe a system which can increase the program developer's productivity by 1) assisting in the creation and reuse of program specifications and 2) automatically generating the complete source program from the program specifications. This system is made up of:

1. a fixed number of Program Component Classes,
2. methods associated with those classes to support source code generation,
3. a user interface to assist the program developer in creating instances of the program component classes,
4. a method interpreter to execute the methods and
5. an object management system to maintain the data base.

Each of these modules will be discussed in more detail in the following sections.

Program Component Classes

Program components are pieces of a program. They are not functions or procedures to be called by a main program. They are physical segments of information(code

or data or both) which provide a part of the functionality of a complete program. They would not stand alone and must be combined with other segments to produce a complete program. The program components are a physical decomposition of a program, where each component is generated asyncronously. The various physical components are not merged into a coherent program until late in the generation process. Our criteria for selecting good program components includes creating reusable components as well as simplifying the specification of individual components.

The design of a program includes designing the algorithms and the data structures for the program. Breaking the process down further, the data part of a program may be thought of as internal data and external data. The external data may be input data or output data. The input data may be from a file, from a screen or from some other input device. The output data may be going to a file, a screen, a printer or some other output device. This system has broken the program design problem down to this level and created Program Component Classes for each of these pieces of a program design.

The following six Program Components Classes were identified as sufficient to contain enough information to compose the whole program from its pieces. The six are PROGRAM, BLOCK, DATA, SCREEN, REPORT, and ACCESS(see Fig 1). The DISPLAY class is a superclass of SCREEN and REPORT, present to take advantage of the high degree of commonality in SCREEN and REPORT. These are the only superclasses which are maintained by the system, but each of these may have subclasses. For example, in the PROGRAM class there is a subclass called REPORT PROGRAM. When the program developer creates an instance of the REPORT PROGRAM class, the program object already has knowledge, through its methods, of how to produce a report program. The program developer may provide only the DATA object and the REPORT object in order to generate a complete working program.

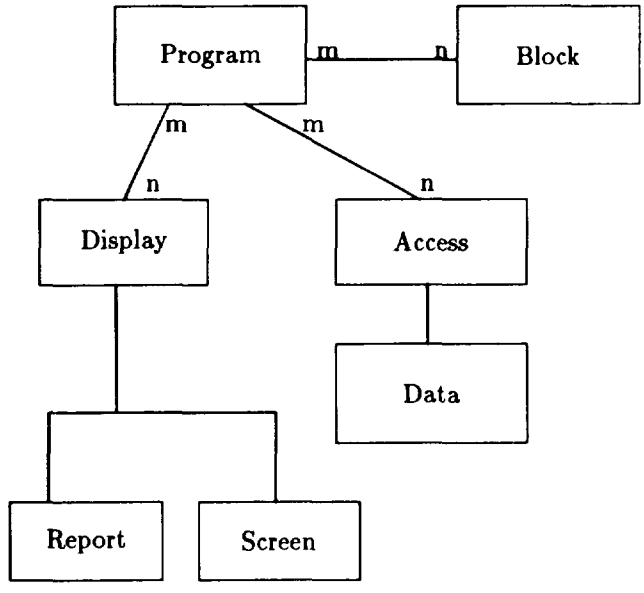

Figure 1: Data Base Diagram

Algorithm Specification

The system currently supports many data processing applications using default methods and existing PROGRAM subclasses. Examples are data entry programs,

file updating programs and report programs. A set of these basic algorithms have been identified and embedded in PROGRAM subclasses. The program developer merely chooses the appropriate PROGRAM subclass to use a particular algorithm. For example, the algorithm for producing a report is to read data from an input source, process the data into the information to be displayed, format the report and send the formatted report to the output destination. As mentioned above, there is a subclass of the PROGRAM class, called REPORT PROGRAM, which already has this algorithm designed inside of it. The PROGRAM class simply allows the program developer to name the program being created, for example "Inventory Report Program" and choose the subclass for the program, for example REPORT PROGRAM. Once the subclass is chosen, which means the algorithm is identified, certain required parameters must be specified. For example, the names of the input source for the data and the name of the report specification for output. Other optional parameters may also be specified to modify the generated programs default behavior.

If there is an algorithm which is not already included in a subclass of the PROGRAM class or if the program developer wants to modify the algorithm of a PROGRAM subclass, the BLOCK class allows definition and inclusion of a block of code into specified points in the generated program. In the above example, if the processing of the input data into information for the report required a special algorithm, such as a formula for the calculation of a report item, then this algorithm could be coded in a BLOCK object and included in the processing section of the generated program at its correct place in the flow of the program execution. Note that standard algorithms can be coded as BLOCK objects once and used in any and all programs that need to perform that algorithm.

Data Structure Specification

Data Structures are specified using the DATA class. Once an object in the DATA class is created, it can be used in any number of programs. There are standard subclasses of DATA such as STRUCTURE, ALPHABETIC and NUMERIC. In addition, it is possible for the user to create a new subclass for a commonly used data item. An example of this might be a CUSTID subclass, which has its format and procedures defined in its subclass. Any program which used it would get its properties already defined.

Other classes closely related to the DATA class are SCREEN, REPORT and ACCESS. SCREEN and REPORT classes are very similar and allow the program developer to layout and define screens and reports, including both format and content. Both screens and reports are created using a screen painter. The code for creating and using the screen or report is generated by the system. Fields on the screen or report are mapped to data items inside a DATA object.

If data is to be input or output on a device other than the screen or printer, then the processing for the device must be specified in an ACCESS object. The ACCESS class allows the program developer to describe the way the data is to be accessed. This may be a file access method such as Sequential or Indexed-sequential, or a data base access such as SQL or some other database management system interface language. If some other way of accessing the data in the program is to be used, such as a remote sensing device, then an ACCESS object must be specified to describe the processing which must be done for that device.

Methods

Methods are the procedures which generate the source code associated with the program component objects. For the current version of the system the *only* function which methods provide is the generation of the source code for the object it is associated with. In the future, methods might do other functions such as produce documentation for the object it is associated with.

Each method uses the data stored with its object(the object's instance variables), as well as the parameters passed to the method, to determine exactly which code to generate. Control of the whole source program generation process is maintained by a *generate* method in the PROGRAM subclass for the program object being generated. This *control* method determines which other program component objects to call upon to generate code for the program and also determines the order of the code in the final program generated.

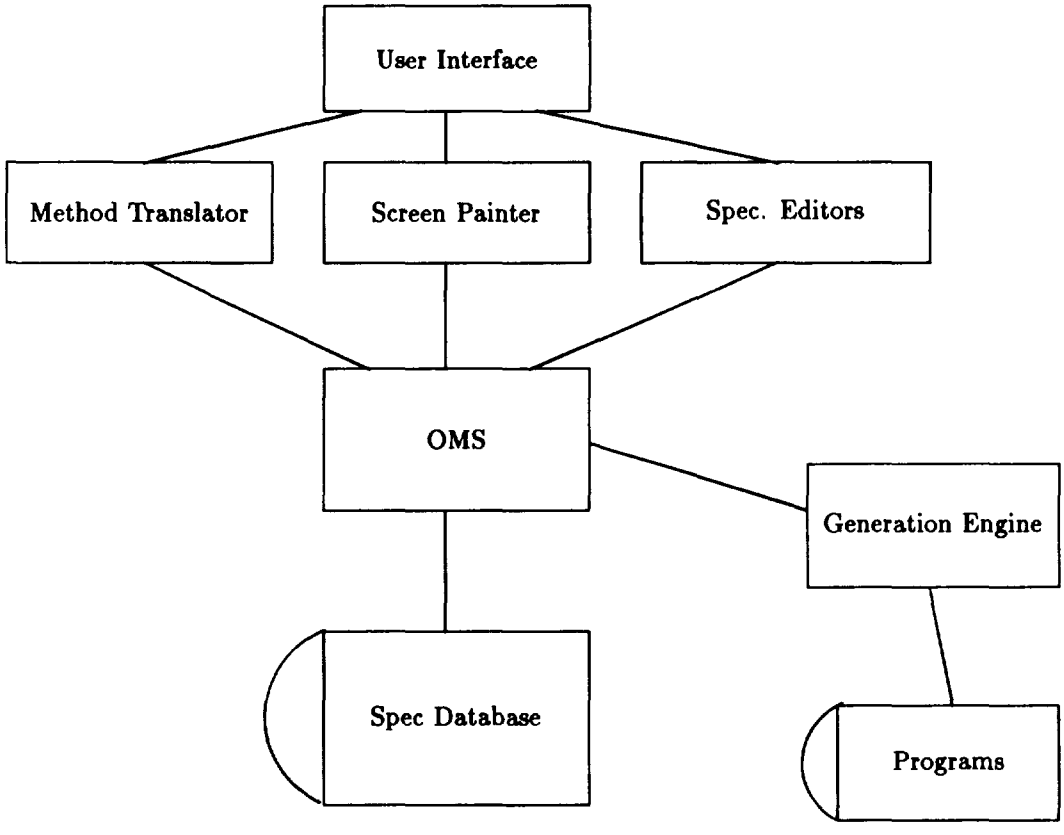

Figure 2: System Flow Diagram

An example of a method for an object is an "open" method for an ACCESS subclass that describes a file access method. The "open" method would contribute source code to "open" the particular file. The methods are written in a *generation* language which is a proprietary high level language similar to the C language, with extensions to support source code generation. It is these extensions which allow efficient generation and placement of code from the methods. The target language, that is the language for which the source code is being generated, is not built into the generation language. At present the generation language is not "aware" of which language it is generating. Therefore, any target language is feasible. Most of the testing of this tool has been done with COBOL as the target language, but C and ADA have also been used. The generation language is interpreted by the generation engine.

User Interface

The user interface assists the program developer both in creating individual objects from the program component classes and in knowing the number and class of objects to create for a particular class of program. The user interface presents the program developer with menus which have the selections to be made for the next level of object creation and with forms to fill in to provide values for the attributes of the objects. When the value to be filled in is the name of an object in the data base, list support provides a list of all objects of that class which already exist for the program developer to select the proper one. Both general help and context help is available from anywhere in the user interface.

In order to assist the program developer in the design process, there is a facility called the "design tree". A natural order for creating the objects which go into a complete program would be to first create the program object and then create any object the program object references and so on. The design tree supports this technique of object creation by displaying a tree structure with the program object at the root and with the tree containing all objects referenced by the program object or any other object in the tree. The program developer may create or modify any object listed in the tree by simply positioning the cursor to that object and pressing the "enter" key. The tree shows, via color coded information, the status(referenced, defined, complete and checked) of all objects displayed in the tree.

Generation Engine

The generation engine is an interpreter. Its architecture includes a lexical analyzer, a parser and a set of action routines. It interprets the programs(methods) written in the generation language which cause source code to be generated. It is invoked by the program developer from the user interface. A *generate* message is sent from the user interface to the PROGRAM subclass corresponding to the program object referenced in the user interface menu. The generation engine starts to interpret the *generate* method and source code generation proceeds from there. Methods are stored in tokenized form and cached in memory for more efficient processing at generation time. A typical program of 500-700 lines will generate in 2-3 minutes on a 386 type of micro-computer.

The generation language, a general purpose high level language with extensions developed specifically for source code generation, is a proprietary language. Its control structures and types are similar to any modern high level language. However it has features for building source statements and creating and composing program segments which facilitate the generation of source programs in almost any computer(and perhaps non-computer) language. The generation language is an essential part of the system.

Object Management System

The Object Management System is responsible for storing and retrieving the classes, methods and objects in the database. The classes, methods and objects are stored in a random access file in a btree structure. A commercially available btree file manager was used for the btree functionality. Since the system is designed to run on a workstation which can be on a network, the object management system automatically maintains both a local and a shared database. The local objects are on the workstation's disk and the shared classes and objects are on the

network disk. When an existing object is referenced it is automatically transferred from the shared database to the local workstation database. However, only someone with project administration permission can move an object from the local database into the shared database. Optionally, an intermediate database, called the project database, can be configured between the shared database and several local workstation databases. This database can hold objects which are being created and shared within one project while they are still in development mode. When the project is complete, the objects can be moved to the network shared database.

The Object Management System is a special purpose Object-Oriented Database System. It is designed to create and maintain a specific set of classes and objects. In general, new classes cannot be added to its capability. However subclasses of existing classes may be added, so the system is not totally rigid. The extent of the system's usability and flexibility is just now being explored.

Problems Identified

The initial generation of a program may take from three to ten or fifteen minutes, depending on the size of the program. Even though this is a relatively long time, it seems to be acceptable for the task being done. However, suppose that once the program is generated and executed, there needs to be a minor change made, say to the format of a report. For example, moving a field over one or two columns. Since the program is generated in source code form, it is very easy and quick to change the generated program. It may take just a few seconds in an editor to change the program and then recompile it. However, now the generated program and the program specified in the tool do not match. To make sure they do match, either 1) the changes must always be made to the program specifications and then the program regenerated or 2) there must be a way to analyze the changed program and "reverse engineer" the changes back into the specifications, or 3) the program developer must update the specifications to match the generated program.

The problem with solution 1) is the time to regenerate the program is the same as the time to generate it in the first place. The second, and subsequent regenerations, are perceived as taking too long for the task involved. Either generation time must be improved or a technique of partial generation must be developed. Partial generation has been analyzed and is a very hard problem.

Solution 2) has not been investigated in this project and there are no current plans to do so.

Solution 3) is the current solution being used, but this solution relies on the program developer to remember to make the changes and to make them accurately. There are too many potential errors in this process for it to be acceptable.

The problem is being solved with a combination of solutions 1) and 3). The generation time is being improved and some partial generation(saving of intermediate forms) is being investigated. In addition, more capability is being developed to assist the program developer in keeping the specifications current and sychronized with the generated program.

Conclusion

The system described above is a special purpose Object Oriented Database Management system for program generation. It is anticipated that experienced program developers may be able to increase their productivity by a significant amount using this system.

The user interface, object management system and generation engine are written in the C language and the methods are written in a high level language designed specifically for source code generation. The system is implemented on microcomputers. Target languages generated include C, ADA and COBOL. Other target languages may be added by changing the methods in the program component classes. The user interface and generation engine do not change.

Plans for future enhancements include more ACCESS classes for different types of database management systems, different hardware environments, different domains(more complex algorithms) and different target languages.

References

1. Ahlsen, Matts, Anders Bjornerstedt and Christer Hulten, OPAL: An object-based system for application development, *Database Engineering*, IEEE Computer Society, Vol. 4, 1985, p267-276.
2. Alabiso, Bruno, Transformation of Data Flow Analysis Models to Object Oriented Design, *Proceedings of OOPSLA'88*, San Diego Ca, Sept 25-30, 1988, p335-354.
3. Dahl, O. and K. Nygaard, SIMULA, an ALGOL based simulation language, CACM vol 9, pp 671-678, 1966.
4. Diederich, Jim and Jack Milton, An Object-Oriented Design System Shell, *Proceedings of OOPSLA'87*, Orlando, Fla, Oct 4-8, 1987, p61-77.
5. Maier, D., A. Otis and A. Purdy, Object-Oriented Database Development at Servio Logic, *Database Engineering*, IEEE Computer Society, vol 4, 1985, p294-301.
6. Nestor, John R., Re-creation and Evolution in a Programming Environment, *Proceedings of 1986 International Workshop on Object-Oriented Database Systems*, Pacific Grove, Ca., Sept 23-26, 1986, p230.