MOOSE: Combining Software Engineering and Database Management Systems*
(Extended Abstract)

Max J. Egenhofer
Surveying Engineering Program
Andrew U. Frank
Computer Science Department
University of Maine
Orono, ME 04469, USA
MAX@MECAN1.bitnet
FRANK@MECAN1.bitnet

Abstract

Software engineering techniques have not been exploiting database management technology in the past. Some of the major deficiencies in the conventional maintenance of medium sized and large software systems can be overcome by storing program code in databases. In order to pursue this goal, data structures for program code have to be developed that can serve as schema descriptions for programs as structured text. MOOSE, the Maine Object-Oriented Specification Editor, is a prototype of such a CASE, tool implemented on top of an object-oriented database.

1 Introduction

Highly modular, object-oriented software systems require a programming environment to support the programmer during encoding, with information about existing modules, routines, and their functionalities. Currently, large software systems are cumbersome to maintain. This deficiency is due mostly to the methods by which programs are developed, and the tools which are used. With the current tools for software engineering, programs are written as text files so that they are suitable for compilation, but not necessarily for human understanding. An effort to restructure the program text for better human understanding was made by Knuth with the WEBB precompiler [Knuth 1984]. In general, batch-oriented concepts and their tools of the 60s and 70s still dominate the process of encoding: code is written into sequential files with a text editor, and afterwards a compiler checks the syntax and translates the code into some machine language. Such systems often require several attempts before a program is free of syntactical errors. More advanced techniques, like on-line syntax checking, identify many syntax errors

*This project is was partially funded by a grant from NSF No IST-8609121 and equipment grants from Digital Equipment Corporation

Egenhofer, M., and A. U. Frank. "Moose: Combining Software Engineering and Database Management Systems." Paper presented at the Second International Workshop on Computer-Aided Software Engineering, Cambridge, Massachusetts, USA, 12-15 July, 1988 1988.

as the programmers type them and reduce the amount of syntax errors tremendously; however, their usage is restricted to the terminal symbols of a language.

While the technology of new tools for software development made at least some progress, maintenance of large software systems has been largely neglected—and most of today's programming effort is directed toward changing existing programs. When modifying a routine, many other routines or modules may have to be adapted to these changes. The list of desired functionality of a software engineering system which is geared to maintenance work must include such important issues as consistent updates throughout an entire system, feedback about incomplete parts, management of modifications such that compilation and linking can be automated, management of the structure of a modular system, etc. These tools must replace the cumbersome and error-prone form of maintenance in which changes are made manually.

The major impediment for better control over the software parts and their dependencies upon each other is the fact that programs are written and stored as text files providing no details about the structure of the modules or their components. We claim that managing code in a database management system will help to overcome many of today's problems with maintaining software systems.

In order to deal efficiently with data, the connections among the data to be stored must be analyzed and formally described. Data structures as formal methods describe common patterns for data of a specific application. The object-oriented data model can be applied for structuring programming code. Abstraction methods built on basic concepts similar to the ones presented in [Brodie 1984] are: classification, generalization, and aggregation.

- *Classification* can be expressed as the mapping of several objects (instances) onto a common class. The word *object* is used for a single occurrence of data describing something that has some individuality and some observable behaviour. The terms *type*, *object type*, *class*, or *abstract data type* refer to types of objects, depending on the context. A type characterizes the behaviour of its instances by describing the operators that can manipulate those objects [O'Brien 1986]. These operations are the only means to manipulate objects.
- *Generalization* establishes an is_a hierarchy among two or several classes [Dahl 1968] [Goldberg 1983].
- *Aggregation* combines several classes to form more complex objects, such that each object part keeps its entire functionality.

Diagrams in the style of the entity-relationship model, which are frequently used in the schema design for data structures of databases, make the links among the individual components of even complex connections visible. Unlike the entity-relationship model, a graphical view of modeling is used which shows the generalization relation between object types as boxes included within boxes, presenting their intuitive order. The general form of the diagrams is summerized in the appendix.

Generally, these abstraction methods are implemented using some techniques of software engineering; here, the techniques are applied to model their own implementation. In particular, classification is mapped directly on software modules, its own implementation; and operations of a software module correspond to the operations which are executable on a class.

This paper investigates how program code can be stored in a database such that details may be quickly retrieved and presented, and that multiple relations among the parts of programs can be made visible. It begins with a discussion about object-oriented programming languages and their suitability for treatement in a database environment. MOOSE-talk, a high-level specification language, is introduced

as a language with a minimal set of functionality. In chapter 3, the data structure of an object-oriented language is analyzed and presented by using a graphical modeling language. The paper closes with results of the implementation MOOSE, a prototype for a code management system implemented in accordance with the structures introduced.

2 Programs as Structured Text

The modular programming concept is closely related to the implementation of abstract data types (ADT) [Goguen 1978] [Guttag 1977], and stresses the paradigm of reusable code in an object-oriented environment. Writing programs as unstructured text in flat files disregards the fact that a program is a complex structure of links between definitions and references. Therefore, programs must be organized as *structured text* such that the references follow the pattern set forth in the language definitions. All related parts are connected with each other. A hypertext-like medium which is suitable for managing unstructured text would not efficiently support the management of programming code. Arbitrary links between any text element could be created, leaving the user without control over the structure. Instead, programs shoule be organized as structured text and stored in databases which are well-suited for managing large amounts of structured data.

2.1 Conventional vs. Object-Oriented Languages

It is feasible to use a mainstream programming language, such as Pascal or Modula-2, for a modern, object-oriented programming style. Encapsulation and modularization can be achieved with a pre-compiler managing import/export of operations between modules and checking types across modules [Egenhofer 1988]; however, the structure of a Pascal-like language leads to code which appears to be verbose if an object-oriented style is adopted. A programming language suitable to be treated as structured text should provide a minimal set of functionality with functions and variables. As the definition will be small, most of the keywords in traditional languages that are responsible for multi-page function definitions are superfluous. Compare the following two segments of Pascal code and a specification-like language which describe Rational number multiplication.

```
FUNCTION ratMult (a, b: ratType): ratType;
VAR num, denom: integer;
BEGIN
  num := ratNumerator (a) * ratNumerator (b);
  denom := ratDenominator (b) * ratDenominator (a);
  ratMult := ratMake (num, denom);
END;
```

```
ratMult (a,b) == ratMake (intMult (ratNumerator (a), ratNumerator (b)),
                          intMult (ratDenominator (a), ratDenominator (b)))
```

2.2 An Object-Oriented Programming Language

Object-oriented programming languages are the tools used in software engineering to implement object-oriented abstraction mechanisms. Object-oriented languages are distinguishable from conventional programming languages because they support [Edelson 1987]:

- modularization by combining type definitions and routines,
- encapsulation of implementation details,
- generic typing [Cardelli 1985],
- inheritance of operations from superclasses to subclasses.

These concepts do not include a message-passing paradigm [Goldberg 1983] that is often cited as necessary for an object-oriented design. It was outlined that message-passing is more of a pedagogical than semantical difference to conventional routine calls, and any procedure call in an Algol-like language could be seen as message-passing [Storm 1986].

The language for which the data structure will be investigated is an experimental, high-level, specification-like language called MOOSE-talk [Frank 1987]. MOOSE-talk was designed to be an implementation independent language, suitable for persistent management by a database system. The structure of the language strictly separates the interface of the object and the implementation of the methods. Abstract implemententations [Frank 1986] [Olthoff 1985] specify what an operation should do. The syntax of MOOSE-talk is concise and has little overhead of syntactical elements in order to be directly mapped upon a data structure. The language has a functional structure and is tailored for very short (most often one-line) abstract implementations of operations. A short example specifying the abstract data type *rational* will help in understanding the concepts. Throughout the analysis of the data structure, this example will be used to clarify the ideas.

```
ADT[1] rat --[2] rational numbers
USING[3] int WITH[4] mult

make[5]:     int[6] x int- rat[7]
num:         rat       - int -- numerator
denom:       rat       - int -- denominator
mult:        rat x rat - rat

AI[8] num (make (i1, i2)) == i1
      denom (make (i1, i2)) == i2
      mult[9] (a, b) == make (mult[9] (num (a), num (b)),
                              mult (denom (a), denom (b)))
```

MOOSE-talk supports object-orientation in the following ways:

- Type definitions and corresponding operations are combined into modules to form object-ADTs,

[1] Upper case commands are keywords.
[2] Text after '--' is comment.
[3] USING describes imported types.
[4] WITH describes used operations of imported ADTs.
[5] Specification of the operations.
[6] Input parameters.
[7] Output parameters (results).
[8] The Abstract Implementation describes how the operations are implemented.
[9] Multiplication is overloaded with different meanings for integers and rationals.

- the specification (i.e., the interface) is separated from the implementation,
- internal parts are encapsulated so that objects are only accessible through their operations in the interface,
- the usage of types and operations, which were defined elsewhere, is encouraged, and
- inheritance of operations is supported.

Currently, MOOSE-talk does not support generic types [Cardelli 1985] and multiple inheritance [Nguyen 1986], but it is planned as a future extension.

3 A Data Structure for Object-Oriented Programs

The components of the language will be analyzed. This analysis leads immediately to a structure that is suitable as a schema design for a code DBMS. The following three major parts in the implementation of an abstract data type will be identified: (1) type definitons, (2) operations upon an ADT, and (3) abstract implementations of these operations.

3.1 The Type Definition

The definition of a type is composed of three parts: (1) the definition of the type name, (2) the reference to other ADTs which are used for the definition or implementation of this ADT, and (3) a listing of the operations used in the ADT implementation. The data structure for the type definition must consider the following issues:

- Each ADT is identified by a unique name, such as *int*.
- Each ADT can use several external ADTs, such as *int USING int*; used types are either sub-parts of the type (as stated in the *make* operation) or occur in the operations as parameters.
- An ADT can provide operations for several other ADTs (by default, any ADT uses itself); *int*, for example, is used by the ADT *int*.
- Operations of other ADTs may be used in the implementation part. The specifications for these operations belong to a class and are explicitly made available. This feature is mainly a control mechanism against undesired usage.

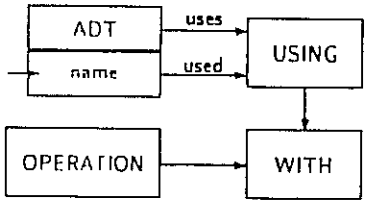


Figure 1: The structure of a type definition.

3.2 The Specification of the Operations

The definition of the operations is composed of: (1) the name of the operation and the link to the corresponding type, (2) the input parameters, (3) the output parameters, and (4) the result of the operation if it is a function. The following issues must be considered for the data structure:

- Each operation has a name which is unique with respect to the ADT, e.g., *make* is unique within *rat*; however, another ADT may have another operation with the same 'local' name. By concatenating the unique ADT name and the locally unique operation name, globally unique operation names are achieved, e.g., combining the ADT name *rat* and the operation *make* to *rat.make*.
- An operation belongs to a single ADT, such as *rat.make* to *rat*, while one ADT may have several operations, e.g., *rat* with *make*, *num*, *denom*, *mult*.
- Each operation can have several parameters.
- The parameters themselves are ADT types which rely on other ADT definitions.
- In its most strict sense, an operation is a function and has a single result; however, for applied programming, it was found necessary to allow the usage of a generalized operation structure, where an operation has a number of input and output parameters, and side effects may change some parameters.

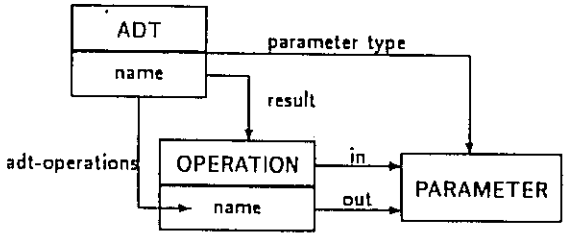


Figure 2: The structure for operations.

3.3 The Abstract Implementation

The abstract implementation is the most complex part of the data structure as it contains both recursive definitions and instances which are linked to previously defined types or operations. An abstract implementation: (1) is connected to the definition of the corresponding operation, (2) contains routine calls to other operations of ADTs, (3) states the connection between the parameters in the definition of the operations and the instances, and (4) includes place holders (variables) which pass values among parameters. The data structure must consider the following details:

- Each specification must have some statement of how it is abstractly implemented.
- A statement in an abstract implementation consists of a left-hand and a right-hand expression.

- An expression is either a function call or an instance of a variable. For the sake of generality, references to constants are implemented as functions with no parameters.
- The instance of a variable refers to a variable.
- A variable may be global with respect to an ADT encapsulating a state.
- Local variables are valid within a single abstract implementation; they are identified by names which must be unique with respect to an abstract implementation.
- Each line of an abstract implementation can have several variables with different names, e.g., *i1* and *i2* in *num*; these variables can be referenced several times within the same abstract implementation.
- Each function call is linked to the definition of an operation; a function can be called several times, while each call belongs to a single definition.
- Functions have arguments which map onto the parameters of their definition. Each parameter can have several instances, while an argument is always linked to a single definition.
- An argument may be an expression, which can be either another function (nested calls) or the instance of a parameter. For example, the argument of the expression *num* is another expression, the function *mult*, while the arguments of the expression are instances of the variables *i1* and *i2*.
- Each expression has a type which is the resulting type of a function or, for variables, the type of the parameter to which it refers.

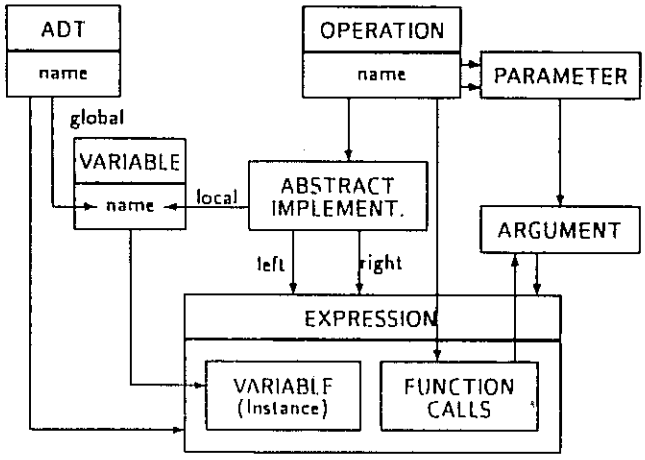


Figure 3: The structure for an abstract implementation.

In addition to the above structure, the type definition, each definition of its operation, and the abstract implementations can have comments which must be stored with the corresponding structure.

4 Conclusion

This study investigated the use of database management techniques for a persistent management system of object-oriented programming code. Current software environments suffer from the demand of linear encoding and the lack of on-line control over used code. As a more efficient approach, program code should be treated as structured text, such that it can be stored in a database. Using standard database management techniques, a code management system can quickly retrieve details, provide cross-references among used program parts, and can guarantee consistency for routine names, types, and parameters within an entire large software system. Updates can be immediately evaluated without leaving it up to the programmer to manually propagate modifications from one module to another.

Additionally, the data structure for an object-oriented programming language is needed to efficiently store programs as strucutred text in a database. Three major parts were identified: (1) the definition of object types, (2) the definition of the corresponding operations, and (3) the abstract implementations of the operations. For each of these three parts, a detailed structure was presented by using a graphical description

MOOSE, a prototype for a DBMS based code management system, was implemented on top of PANDA, an object-oriented database [Frank 1982]. The implementation, including the data definition, contains 70 modules with about 5000 lines of highly modular Pascal code. Some experience with a small set of ADTs was gained. During its design and implementation it was observed that the consistency constraints among the expressions are very complex. High-level, object-oriented operations for deleting and modifying ADTs and operations must perform a large amount of cross-referencing and -checking. This should not surprise anyone with experience in modifying large software packages. The implementation of 'change operations' for each major structural part makes these dependencies visible from an operational viewpoint. Assistance from a program may be an important CASE tool.

A translator is considered to automatically transform this form into Pascal or C. Such a translator can be very powerful if it is combined with a code management system based on a database management system. Two primary advantages are expected:

- Some manual translation of MOOSE code into Pascal showed a reduction in the size of the code by a factor of 3. More compact coding is expected with generic coding. The size of software systems is an important issue, because comprehension of code is faster if the code is spread over fewer lines, but still clearly structured.
- A translator can produce more efficient executable code (e.g., by in-line expansion of functions) without sacrificing the clarity of the format used by human programmers.

5 Acknowledgement

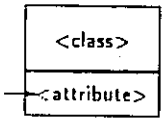
Thanks for the comments of Doug Hudson, David Pullar, and Jeff Jackson who read an earlier version of this paper.

A Diagrams for Abstraction Methods

The diagrams for the abstraction methods have the following structure:

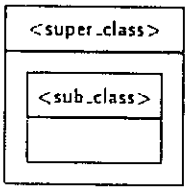
A.1 Classification

A class is represented by a box around the class name. Lower-structured details of a class are written in the lower part of the box of the corresponding class and globally unique key identifiers are marked by an arrow pointing to the corresponding attribute.



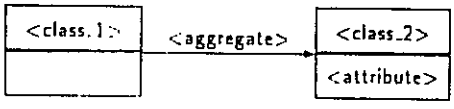
A.2 Generalization

Generalization is drawn as box around a box, such that the superclass contains the subclass(es).



A.3 Aggregation

Aggregation and association among classes are marked by arrows running from one box to another; the direction of the arrow indicates the structure of the aggregate running from the superpart to the subparts (1:m relations).



Unique key identifiers with respect to an aggregate are marked by having the aggregation arrow point inside the box to the corresponding attribute.

References

[Brodie 1984] M.L. Brodie. On the Development of Data Models. In: M.L. Brodie et al., editors, On Conceptual Modelling, Springer Verlag, New York (NY), 1984.

[Cardelli 1985] L. Cardelli and P. Wegner. On Understanding Type, Data Abstraction, and Polymorphism. ACM Computing Surveys, 17(4), April 1985.

[Dahl 1968] O.-J. Dahl et al. SIMULA 67 Common Base Language. Technical Report, Norwegian Computing Center, Oslo, 1968.

[Edelson 1987] D. Edelson. How Objective Mechanisms Facilitate the Development of Large Software Systems in Three Programming Languages. SIGPLAN Notices, 22(9), September 1987.

[Egenhofer 1988] M. Egenhofer and A. Frank. A Precompiler For Modular, Transportable Pascal. SIGPLAN Notices, 23(3), March 1988.

[Frank 1982] A. Frank. PANDA—A Pascal Network Database System. In: G.W. Gorsline, editor, Proceedings of the Fifth Symposium on Small System, Colorado Springs (CO), 1982.

[Frank 1986] A. Frank. The Design of Information Systems, Part 1: Theories for Spatial Information Systems. 1986. classnotes, University of Maine at Orono, Department of Civil Engineering, Surveying Engineering, Orono (ME).

[Frank 1987] A. Frank. Databases are Crucial for Computer-aided Software Engineering. In: E. Chilkofsky, editor, First International Workshop on Computer-Aided Software Engineering, Advance Papers, Cambridge (MA), May 1987.

[Goguen 1978] J.A. Goguen et al. An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types. In: R. Yeh, editor, Current Trends in Programming Methodology, Prentice-Hall, Englewood Cliffs (NJ), 1978.

[Goldberg 1983] A. Goldberg and D. Robson. Smalltalk-80. Addison-Wesley Publishing Company, 1983.

[Guttag 1977] J. Guttag. Abstract Data Types And The Development Of Data Structures. Communications of the ACM, June 1977.

[Knuth 1984] D. E. Knuth. The TEXbook. Addison-Wesley Publishing Company, Reading (MA), 1984.

[Nguyen 1986] V. Nguyen and B. Hailpern. A generalized Object Model. SIGPLAN Notices, 21(10), October 1986.

[O'Brien 1986] P. O'Brien et al. Persistent and Shared Objects in Trellis/Owl. In: International Workshop in Object-Oriented Database Systems, Pacific Grove (CA), 1986.

[Olthoff 1985] W. Olthoff. An Overview on ModPascal. SIGPLAN Notices, 20(10), October 1985.

[Storm 1986] R. Storm. A Comparison of the Objet-Oriented and Process Paradigms. SIGPLAN Notices, 21(10), October 1986.

CASE '88

Second International Workshop on
Computer-Aided Software Engineering

Advance Working Papers

Volume 2

Cambridge, Massachusetts
July 12–15, 1988