The Extension of Data Abstraction to Database Management

Anthony I. Wasserman

Medical Information Science
University of California, San Francisco
San Francisco, CA 94143

Introduction

The long-term goal of the User Software Engineering (USE) project at the University of California, San Francisco, is to provide an integrated homogeneous programming environment for the design and development of interactive information systems. Realization of this goal involves the development of new software tools, their integration with existing tools, and the creation of an information system development methodology in which these tools are systematically used [1,2].

The successful construction of interactive information systems requires the utilization of principles of user-centered design [3,4,5], combined with features traditionally associated with the separate areas of programming languages, operating systems, and data base management [6]. It has become increasingly clear that the key to being able to provide such a unified view lies in providing a unified view of data [7]. The potential benefits of such a unification are considerable, including:

1) conceptual simplification of the system structure permitting, for example, joint design of data structures and data bases
2) the elimination of duplication or inconsistencies among diverse software components
3) the ability to achieve greater reliability in systems because of reduced dependence upon multiple software systems

The Programming Language PLAIN

The programming language PLAIN plays a key role in trying to effect this unification. PLAIN treats relation as a data type and includes a procedural (algebra-like) set of operations on relations and data base objects, including a simple technique for associative access to tuples and to individual elements [8]. This combination of programming language and data base features is augmented by the PLAIN facilities for abstraction and modularity, which build upon the concepts of Pascal and provide support for a systematic approach to programming [9]. A complete specification of PLAIN is given in [10].

Unlike many of the other approaches that combine programming languages with data base management, PLAIN provides a data abstraction facility (termed a **module**) in which data base objects can be accessed. Operations upon these data objects can be encapsulated within the module, thereby making it possible to restrict access to a set of predefined procedures and functions upon a set of relations, providing a form of database abstraction that is particularly useful for transaction-based systems.

The link to the features of the operating system is provided through the **external** declaration, which permits access to objects within the execution environment, including existing databases, previously compiled program units, and other objects, such as special variables storing environment-specific information.

The PLAIN **module** permits both the introduction of new data types based upon existing types and the unification of related operations in an encapsulated object, providing support for the key concepts of abstraction and modularization.

Database Abstraction in PLAIN

The combination of the **module** facility with features for data base definition and manipulation is particularly powerful. Not only does it provide type checking that can be used to constrain operations upon databases, but it also provides an extra level of "information hiding" that makes it possible to make large parts of an information system independent of the database design.

Consider the example of a small banking system. Among the classes of users that can be identified are tellers. We can define informally several activities that are performed by tellers; for purposes of this example, let us permit the teller to process an account holder's deposit or withdrawal and to inform an account holder of the current balance in that account.

We may then construct an abstract specification [11] of these operations, stating preconditions and postconditions that must be satisfied by these operations. Using PLAIN and the relational model of data, it becomes possible to define a set of relations to store the necessary banking information and a module teller that contains the implementation of the operations deposit, withdraw, and balance.

This module *exports* those operations, along with two exceptional conditions that may be raised by those operations (badacct for a bad account number; overdrawn if a withdrawal would cause the account to become improperly overdrawn). The user of the module can use these operations and exceptions in a calling program, but has no access to the realization of the operations.

```
type money = 0 .. 999999.99; idno = 0. .. 99999;
   { idno is a transaction identification number }
type teller = module
  exports
      procedure deposit (acct: integer; amt: money -> tid: idno);
      procedure withdraw (acct: integer; amt: money -> tid: idno);
      function bal (acct: integer): money;
      exception badacct, overdrawn;
  imports checktrans, check: modified; info: readonly;

  ops
      procedure deposit (acct: integer; amt: money -> tid: idno);
      imports checktrans, check: modified; bal: invoked;
      begin
          check%.balance := bal(acct) + amt;
          checktrans :+ <[acct,tid,today,1,amt]>;
           { 1 denotes the transaction type of deposit }
          tid := (tid + 1) mod 100000
      end deposit;

      procedure withdraw (acct: integer; amt: money -> tid: idno);
      imports checktrans, check: modified; info: readonly;
      exception badacct, overdrawn;
      begin
          if bal(acct) < amt & ~info[acct].overdraft then
              signal overdrawn
          end if
          check%.balance := check%.balance - amt;
          checktrans :+ <[acct,tid,today,2,amt]>;
           { 2 denotes the transaction type of withdrawal }
          tid := (tid + 1) mod 100000
      end withdraw;

      function bal (acct: integer): money;
      imports check: modified;
      exception badacct;
      begin
          check% := check [acct];
          if check% = 0! then
              signal badacct
          else
              bal := check%.balance
          end if
      end bal
  end module
```

It can be seen that the **teller** module **imports** relations checktrans, check, and **info,** and therefore has access to the attributes and data elements of these relations.

This module is then used, **for** example, in a main program, here called banker. While banker has access to relations check, checktrans, and info through the **external** declaration, banker serves only to process user input and to invoke the appropriate subprograms and modules, including operations included within teller. Note that the body of banker does little more than repeatedly read lines of input, then dispatch them to the appropriate case label.

This approach serves to **achieve** a number of important goals with respect to the design and implementation of information **systems,** and support of data abstraction, including the **following:**

(1) The specification and **implementation** of the modules can be performed independently of the specification and implementation of the program. Similarly, the database design can be carried out independently of the program design. The database design can simply be included in an external declaration when the **program** is ready for integration testing and **subsequent** release. One achieves an integration **of** database design and program structure, while **maintaining** a systematic approach to both program **and** data design.

```
program banker;
{ This program is an initial design for a simple banking system }
external
        var tid: idno; today: integer;
        type teller: module;
            inforec = record
                        acct: integer;
                        name: string;
                        address: string;
                        phone: char [10];
                        mmnname: string;
                        accttype: 1..6;
                        overdraft: boolean;
                         { true if account can be overdrawn }
                        primary: boolean
                        end record;
        checktransrec = record
                            acct: integer;
                            ident: 0..9999;
                            date: integer;
                            ttype: 1..4;
                            amt: money
                        end record;
        checkrec = record
                       acct: integer;
                       balance: money
                   end record;

var inline: string; { holds the current input line }
    info: relation [key acct, name] of inforec;
        { info stores information on account holders }
    checktrans: relation [key acct, ident] of checktransrec;
        { checktrans stores information on all
          transactions for all accounts }
    check: relation [key acct] of checkrec;
        { check holds the current balance for all accounts }
    helptext: relation [key lineno] of
                lineno: integer;
                text: string
              end relation;
end external;

pattern commands = [cr, deb, balance, help, quit];
        { set of legal commands for teller }
        cr = ('cr', *G);   { the credit command }
        deb = ('db', *G);   { the debit command }
        balance = ('bal', *X, I);   { the balance command }
        help = ('help', *G);   { the help command }
        quit = ('quit', *G);   { the quit command }


    handler trouble;
    begin
       write 'fatal input/output error on reading command\n';
       signal fail { terminates program on terminal I/O error }
    end trouble;



begin { main program starts here }
   loop
       read inline ![ioerr: trouble];
       case inline match commands of
           when cr: credit(inline);
           { credit requests an account number and amount,
             then calls teller.deposit }
           when deb: debit(inline);
           { debit requests an account number and amount,
             then calls teller.withdraw }
           when balance: teller.bal(acct);
           when help: foreach i in helptext
                      loop write text, '\n'
                      repeat;
           { help information is stored in relation helptext }
           else write 'illegal command'
       end case
   repeat;
   write 'goodbye';
end banker.
```

(2) It is straightforward to add new database objects, new modules, and new programs to the information system environment. For example, other programs could be designed to support the new account department, which can open new accounts, and the operations department, which can close accounts and produce monthly statements. In each case, a new module could be created for the new user class and a new program could be written to provide a suitable interface for that user class. Each new program would invoke the appropriate module operations, e.g., newaccount.opennew, and could have access to the same database, in this case the relations info, check, and checktrans, as needed.

(3) The abstract system view for each user class is different, since each user class sees only a predefined set of operations. Nonetheless, all of the distinct views work in harmony with one another. Taken together, they form a complete, possibly distributed, information system.

(4) This approach allows effective enforcement of access rights and integrity checking. Note that the operations provided in module teller do not permit a teller to make any modifications to relation info, since only readonly access is granted. By contrast, module newaccount would be able to modify relation info in order to make new entries, but could not have access to relations check or checktrans. Each module serves to limit the permissible operations for each user class and to make certain that all updates to the database are made through these "structured" operations or, alternatively, by a *very small* number of highly trusted system managers. Each operation checks the validity of the input and of any database modification, making certain that the operation conforms to the pre and post conditions in the abstract specification of that operation.

In such an environment, query languages are used for retrieval purposes only and cannot be used to make unconstrained updates. In that way, the abstract specification and abstraction operations on the data can be preserved.

(5) Use of this design methodology makes it possible to separate the database model and database design from the main program. Indeed, a view of the database could be provided to the module rather than presenting the database itself. Such a step would provide maximum separation among the various aspects of system design. Indeed, the user of the data need know little or nothing about its representation, which could be a collection of files or even program data structures as well as a database.

The USE Programming Environment

The goal of the USE project is to provide a set of tools and a system development methodology that facilitates the task of producing interactive information systems. This methodology and the accompanying tools (of which PLAIN is one), constitute a programming environment intended to enhance the productivity of individual software developers and programming groups, as well as to lead to higher quality systems.

In addition to PLAIN, a number of other tools are being designed and implemented for the USE environment, including:

(1) TROLL, a small relational database handler [12], that can be used both in conjunction with the PLAIN runtime system and as a standalone system;

(2) TDI, a tool for the automation of transition diagrams [13], developed in support of a specification method for interactive information systems [14]; this tool makes it possible to build rapid prototypes of user/program dialogue-based systems;

(3) the Module Control System, a tool to provide assistance in the management of medium-to-large software projects, including version control, configuration control, and documentation support; although it is independent of specific development methodologies, it permits the inclusion of "policies" that can enforce certain disciplines.

All of these tools are being implemented in conjunction with use of the Unix™ operating system. Taken together upon completion, they will provide support for a systematic approach to the specification, design, and development of interactive information systems based strongly upon concepts of modularization and data abstraction.

Acknowledgements

This work was supported in part by National Science Foundation grant MCS78-26287. Computing support for text preparation was provided by National Institutes of Health grant RR-1081 to the University of California, San Francisco, Computer Graphics Laboratory, Principal Investigator: Robert Langridge.

References

(1) Wasserman, A.I., "USE: a Methodology for the Design and Development of Interactive Information Systems," in *Formal Models and Practical Tools for Information System Design*, ed. H.-J. Schneider. Amsterdam: North-Holland, 1979, pp. 31-50.

(2) Wasserman, A.I., "Software Tools and the USE Project," in *Software Development Tools*, ed. W. Riddle and R.E. Fairley. Amsterdam: North-Holland, 1980, in press.

(3) Wasserman, A.I., "User Software Engineering and the Design of Interactive Systems," in *The Non-Expert User*. Maidenhead, England: Infotech, 1981, in press.

(4) Shneiderman, B. *Software Psychology*. Cambridge, MA: Winthrop, 1980.

(5) Martin, J. *Design of Man-Computer Dialogues*. Englewood Cliffs, NJ: Prentice-Hall, 1973.

(6) Wasserman, A.I. and C.J. Prenner, "Toward a Unified View of Database Management, Programming Languages, and Operating Systems," *Information Systems*, vol. 4, no. 2 (1979), pp. 119-126.

(7) Wasserman, A.I., "Toward a Unified View of Data," in *Data Design*, ed. M. Atkinson. Maidenhead, England: Infotech, 1980, pp. 1-28.

(8) Wasserman, A.I., "The Data Management Facilities of PLAIN," *Proc. ACM 1979 SIGMOD Conference*, pp. 60-70.

(9) Wasserman, A.I., "The Design of PLAIN -- Support for Systematic Programming," *Proc. AFIPS 1980 NCC*, vol. 49, pp. 731-740.

(10) Wasserman, A.I., D.D. Sherertz, M.L. Kersten, R.P. van de Riet, and M.D. Dippé, "Revised Report on the Programming Language PLAIN," Technical Report #42, University of California, San Francisco, Laboratory of Medical Information Science, 1980.

(11) Leveson, N., "Applying Behavioral Abstraction to Information System Design and Integrity," Ph.D. Dissertation, University of California, Los Angeles, 1980.

(12) Kersten, M. and A.I. Wasserman, "The Architecture of the PLAIN Data Base Handler," *Software -- Practice and Experience*, to appear.

(13) Wasserman, A.I. and D. Shewmake, "Automating the Development and Evolution of User Dialogue in an Interactive Information System," submitted for publication, 1980.

(14) Wasserman, A.I. and S.K. Stinson, "A Specification Technique for Interactive Information Systems," *Proc. IEEE Computer Society Conference on Specifications of Reliable Software*, 1979, pp. 68-79.