SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 17(9), 623–645 (SEPTEMBER 1987)

An Object-oriented Approach to Application Generation

FRANS VAN HOEVE AND ROLF ENGMANN
Department of Informatics, University of Twente, Postbus 217, 7500 AE Enschede, Netherlands

SUMMARY

The TUBA system consists of a set of integrated tools for the generation of business-oriented applications. Tools and applications have a modular structure, represented by class objects. The article describes the architecture of the environments for file processing, screen handling and report writing.

KEY WORDS Application generation Automatic programming Object-oriented programming

INTRODUCTION

In 1980 the research project TUBA (Tools for User-friendly Business Applications) was started, which yielded the TUBA system, an integrated set of tools for the generation of business-oriented software. Several aspects of the TUBA system have been reported in the literature.[1,2] The present article discusses the architecture of the TUBA system. A more extensive report[3] on the principles of its technical and implementation backgrounds is obtainable from the authors.

The TUBA project has the following objectives:

1. Development of software tools for the implementation of data-processing programs, especially for administrative applications. In this context tools include facilities for (e.g.) file organization and file manipulation, reporting, user/program dialogue, etc.
2. The tools should support the implementer in producing reliable software in a relatively short time and with reduced programming effort.
3. The tools should support the maintenance of consistency between data description and software.
4. Special attention should be devoted to the user-friendliness of software tools.
5. From scientific and educational points of view the project should yield insight into principles and basic techniques in relation to application generation.

TUBA facilities provide an environment for defining, implementing and executing application programs. Comparable systems have been described in the literature.[4–6] The TUBA environment contains facilities for

(a) *Files*: sequential and indexed sequential file organizations, generation of file declarations

0038–0644/87/090623–23$11.50

Received 11 November 1986
Revised 14 May 1987

(b) *Screens*: generation of screens, screen-handling functions
(c) *Reports*: generation of reports.

Earlier favourable experiences with the programming language Simula induced us to use that language for implementing the TUBA system. Its main impact on the project is a style of design and programming that has recently become known as object-oriented programming.[7] The benefits of object-oriented programming are indicated in some detail in subsequent parts of this paper.

PRINCIPLES OF DESIGN AND IMPLEMENTATION

Classification of application programs

For technical reasons, and from the user's viewpoint, TUBA applications can be classified into the following types:

1. *Batch* programs, which have no user interaction; however, execution may be initiated from a user terminal.
2. *Interactive* programs, which maintain a dialogue with the user during execution.
3. *Report* programs, a special kind of batch programs, which produce reports using data from one or more input files.

Modular structure of TUBA applications

In order to meet the objectives, application programs produced by the TUBA tools are compounded from a set of software modules which share only very restricted interfaces. Figure 1 shows the architecture of a TUBA application program. As most of these modules are either available as TUBA standard software or can be generated automatically, the amount of hand-coding is reduced considerably compared with conventional implementation techniques.

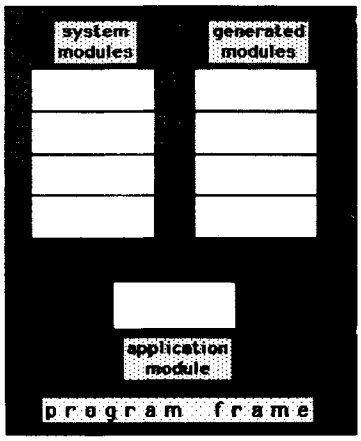

Figure 1. Modular architecture of a TUBA application program

With respect to the software implementation process the following three types of TUBA modules may be distinguished:

1. *System* modules contain a set of related procedures to support specific functions, e.g. reporting, screen communication, file organization, etc. System modules are available in a library and may be linked to any application program as they are application-independent.
2. *Generated* modules are produced automatically by one of the TUBA module-generating programs. Examples of generated modules are file modules and screen set modules, to be discussed further below. Generally these modules are generated on behalf of a specific application program, which uses the applicable files and/or screen sets. The parameters of the generated modules are either specified in a dialogue between the implementer and the module-generating program or retrieved from the dictionary.
3. *Hand-coded* modules contain the statements and local data declarations, which are specific for each application program. Each application program contains at most one hand-coded module, which constitutes the body of the program, the so-called *Application* module. Since many functions of the application program are contained within the other modules mentioned previously the size of an application module is usually small compared with the total program size.

For specific types of programs, e.g. programs which enable the user to update and browse through indexed sequential files via a visual display, all composing modules are either system or generated modules. In such cases the program is generated without any hand-coding.

The modular architecture as it has been described above is also applicable for the TUBA system programs themselves. The TUBA software tools have been constructed in exactly the same way as application programs.

Basic technical background

The implementation of TUBA software has two corner-stones:

(i) the programming language *Simula*
(ii) a dictionary containing meta-data of files and the layout of screens.

Simula

The programming language Simula[8,9] is used for the implementation of tools as well as application programs. Simula provides a very powerful construct for the implementation of abstract data types, namely the *class* construct. The class construct is also especially suited to the implementation of software modules. Features of the Simula class are:

1. A class unites several properties of a procedure, a program block (in languages such as Algol60) and a Pascal record type.
2. As in the case of a block, a class may contain data and procedures as well as executable statements. In addition, a class may have parameters.
3. Internal attributes of a class (data, procedures) may be protected against external access.
4. Each class may have subclasses established by means of the prefixing mechanism; a subclass inherits the attributes of all its ancestor classes.

5. As in the case of Pascal records, multiple objects (instances, occurrences) of a class may be created.
6. Class objects are not deleted automatically after termination of their execution, but their existence is controlled by the program.
7. Class objects may be referenced by other class objects and program blocks via reference variables (pointers) or by prefixing, such that a set of class objects (of one or more types) may be interconnected in a network structure.
8. Classes and procedures can be compiled separately from the main program module if they are specified as external.

As class objects may be connected to other class objects and programs, they are very suitable to implement modules. For this reason, they are frequently used in the TUBA software.

The versatility of the class concept, in particular the feature of prefixing of classes, makes Simula a very suitable language for object-oriented or object-based programming.[7,10,11] In the object-oriented programming technique, species of abstract or real entities are represented by levelled types, and individual entities are represented by objects of such types. Examples of the role of objects are given below.

In addition to the class concept Simula has the following facilites:

(a) the standard class SIMSET, which provides the data structures and procedures for list processing
(b) the standard data type TEXT, with utility procedures for handling text strings
(c) the lower and upper bounds of arrays may be arithmetic expressions, to be evaluated when the array is created
(d) co-routines, which are used in the TUBA command interpreter modules.

Unlike application generators based on APL or BASIC, for example, TUBA does not use code interpretation. All software modules are written or generated in Simula source code and have to be compiled before they can be executed. This feature makes the TUBA system complex, but it is favourable for the performance at execution time.

The dictionary

The dictionary consists of two files, the data dictionary and the screen dictionary. A brief description of the dictionary files follows.

The data dictionary (DD) is a repository of all meta-data of the applications to be developed, e.g. descriptions of file types, record types and item types. The DD has the following functions:

1. Automatic generation of file modules: using the contents of the DD, so-called file modules may be generated. A file module provides the interface between an application and a file.
2. Generation of documentation: from the contents of the DD, surveys of file structures may be generated. Such a survey includes (e.g.) the record types and item types in a file structure.

The screen dictionary (SD) contains complete descriptions of the layouts of screens used by interactive applications. The SD had the following functions:

1. Automatic generation of screen set modules: from the contents of the SD, so-called screen set modules may be generated. A screen set module links a set of screens to the application program.

2. During the execution of an application, the layouts of the screens to be displayed are read from the SD. Therefore the SD is connected to all interactive TUBA programs.
3. Generation of documentation: from the contents of the SD, surveys of screen sets and the layout of screens may be generated.

The catalogue function of the dictionary

The dictionary contains a catalogue of file system names and mnemonic names (catalogue names) of data sets, programs, file modules and set modules. File system names (containing up to six characters) are only specified when the data set, module or program is specified or created; afterwards file system names are only used internally by the TUBA system. Catalogue names may contain up to fifteen characters, in order for the user to have more freedom to select a name which conveys significant information.

When a user has to specify a file, module or program the TUBA system generates a selection screen with the appropriate catalogue names from which the user may select one or more items. In this way only a passive knowledge of catalogue names is demanded from the user. This selection technique is also used for entities such as names of screens, records and data set items: the user never has to enter a name of an entity which is known in the dictionary.

The implementation of modules

Modules in TUBA applications are represented by Simula class objects; a class object is an instance of a Simula class. An application may contain more than one module of the same class. A module is created during the execution of a program either explicitly by the NEW operator, or implicitly when an object of one of its subclasses is created.

The source texts of file modules and screen set modules are produced by generating programs. These generating programs retrieve the main specifications of the modules to be generated from the dictionary; a few of the specifications may be requested in a dialogue with the user. After compilation, an object of such a generated class may be created and linked to an application module.

System classes are common to many or all applications of a specific type and therefore they are not application-dependent. When a system module is needed for an application, an object of the corresponding system class which has been coded and compiled once and for all is created and linked to the application module.

Computer system environment

The TUBA system has been implemented on a DECsystem-20 mainframe computer under the TOPS10 operating system. Screen handling functions operate in GIGI and VT100-compatible visual display terminals, but the terminal interface can easily be adapted to other types of terminals.

THE FILE ENVIRONMENT AND BATCH APPLICATION PROGRAMS

Introduction

The TUBA system provides all applications with a specific file environment, which offers the following facilities:

(a) a sequential file organization (SEQ)
(b) an indexed sequential file organization (TIS : TUBA indexed sequential)
(c) files may contain records of several types and formats
(d) access to attributes of records and data items at execution time.

The file environment is described within the context of batch programs, because batch programs have the least complicated configuration of TUBA application programs. A very simplified configuration for a batch application program is illustrated in Figure 2, which shows the modular structure of a batch application. The batch application module is connected to each of its physical files (in IBM jargon: data sets) through a file module. A file module contains all declarations of records and items in the corresponding data set. The VARDAT module is the common module for all TUBA application programs; it is connected to all file modules in the application. The combination of the VARDAT module and the set of file modules provides a high degree of data independence. The structure and functions of the modules in a TUBA application are discussed in greater detail in the following paragraphs; the user interfaces of some major system classes are described in the Appendix.

The modular structure of the file environment

The TUBA file environment consists of a set of Simula classes. A complete schema of all classes involved in a batch application is shown in Figure 3. In the schema the rectangular blocks denote classes. When two blocks share (a part of) a compound box

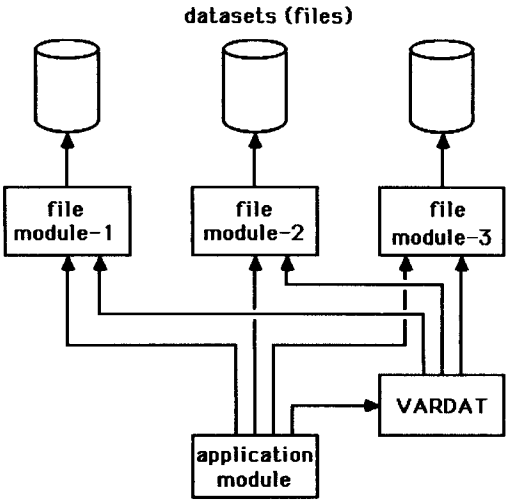

Figure 2. Simplified configuration of a batch application

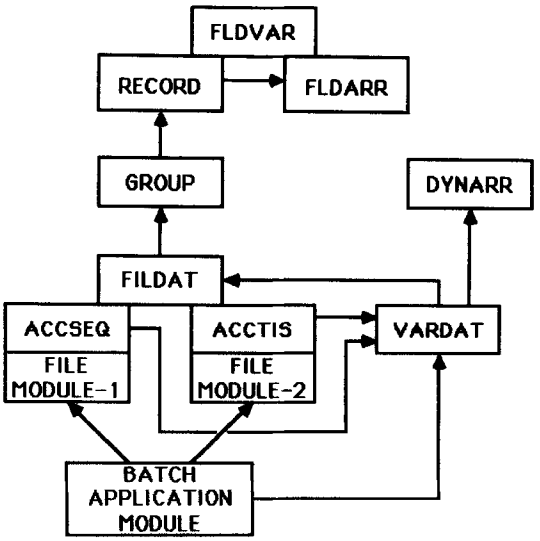

Figure 3. Schema of classes in a batch application

structure, the class denoted by the lower block is a subclass of the class denoted by the upper block; e.g. the class RECORD is a subclass of the class FLDVAR. The arrows denote external referencing; e.g. the class FILDAT is connected to the class VARDAT by a reference variable which has been declared in the class VARDAT.

As at present the class GROUP is used exclusively in report programs, it will be neglected in the current section. The individual classes will be discussed in the following subsections.

The representation of data

All types of items and records within a specific TUBA application program are represented by individual objects; the relevant classes FLDVAR, FLDARR, RECORD and DYNARR are described in the following paragraphs.

The class FLDVAR. An object of the class FLDVAR (FieLDVARiable) represents a specific item in a record type of a data set. A FLDVAR object provides access to the following attributes of an item:

(a) the name of the item
(b) the data type (boolean, text, integer, real, text set, integer set, real set, compound)
(c) the format: length and number of decimals
(d) the current value TXT, which is represented as a text string
(e) a validation expression (optional)
(f) a column heading for reporting (optional)
(g) a narrative description (optional).

All attributes of objects of the class FLDVAR and its subclasses are accessible at execution time, so it is possible to display on request such entities as the description of any item during the execution of an application program.

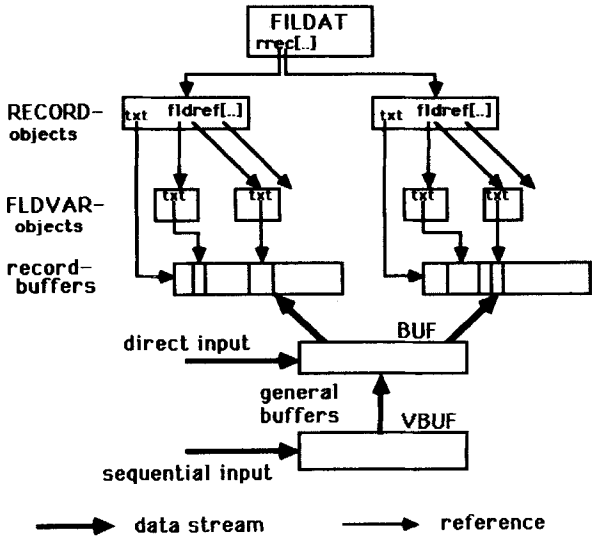

Figure 4. Buffers and text representations of data in the file environment

The class FLDARR. An object of the class FLDARR (FieLDARRay) represents a 1-, 2- or 3-dimensional array of items in a record type. As the class FLDARR is a subclass of the class FLDARR, it has the same attributes as FLDVAR. In addition the class FLDARR contains attributes, which represent the index range(s).

The class RECORD. An object of the class RECORD represents a specific record type of a data set. A RECORD object provides access to the following attributes of a record type:

(a) the name of the record type
(b) the current value TXT of the record, represented as a text string; the text string is the buffer for record occurrences of the corresponding record type
(c) an array FLDREF (FieLDREFerences) of references to all FLDVAR objects, representing the items of the record type.

The TXT-attributes of the FLDVAR and FLDARR objects are substrings of the TXT-attribute of the corresponding RECORD object; see Figure 4. In this way changes in the current values of FLDVAR and FLDARR objects are reflected in the current value of the corresponding RECORD object.

The class DYNARR. The class DYNARR (DYNamic ARRay) has been introduced for storage efficiency. An object of the class DYNARR contains a text array for storage of names, value sets, validation expressions, column headings and descriptions of all unique items and records in an application. Each FLDVAR, FLDARR and RECORD object contains an index into the array, to the entry where the corresponding text values have been stored. This representation is especially effective for items which occur in more than one record type and records which occur in more than one data set in the same application.

The representation of the file environment

The file environment is represented by the classes VARDAT, FILDAT, ACCSEQ, ACCTIS and the file modules. The configuration of modules in a batch application with one sequential (SEQ) and one indexed sequential (TIS) data set is shown in Figure 5, in which the boxes denote modules (class objects) and the arrows denote references between modules. Note the difference from Figure 3, which shows classes and external declarations. For each data set there is an object consisting of a file module, an occurrence of either the class ACCSEQ (SEQuential ACCess) or ACCTIS (Tuba Indexed Sequential ACCess), and an occurrence of the class FILDAT. For each application there is just one object of the class VARDAT. The application module has direct references to the VARDAT module and to the file modules.

File modules. A file module is a class object, which contains declarations of references to all FLDVAR, FLDARR and RECORD objects corresponding to the connected data set. In addition it contains procedure calls, which give rise to the creation of those objects. In this way all FLDVAR, FLDARR and RECORD objects are directly accessible through the file module. A file module also contains assignments to a number of system variables and provisions for creation of a new (empty) file or for connecting an existing file to the application.

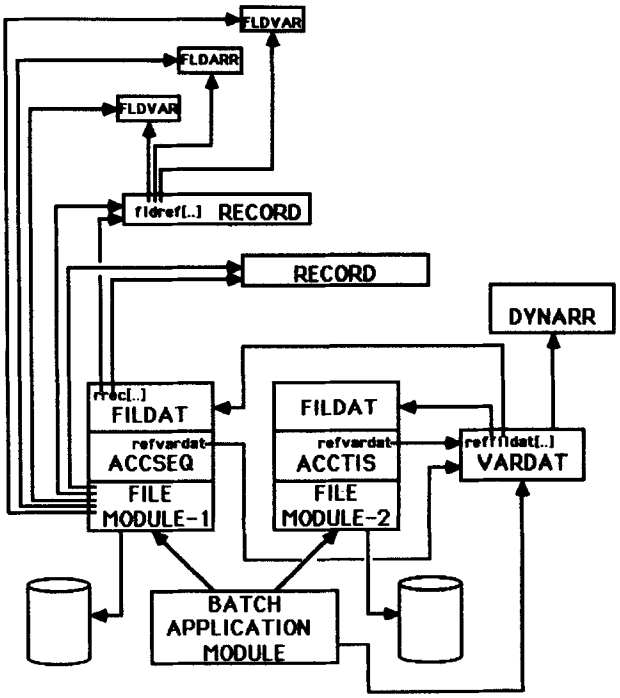

Figure 5. Configuration of modules in a batch application

The classes ACCSEQ and ACCTIS. The classes ACCSEQ and ACCTIS contain
(a) procedures for processing the SEQ and TIS data sets respectively
(b) a reference to the data set
(c) a reference to the VARDAT module in the application.

The user interfaces of ACCSEQ, ACCTIS and other system classes are described in more detail in the Appendix.

The class FILDAT. The class FILDAT (FILe DATa) contains
(a) an array RREC with references to the RECORD objects representing the record types of the connected data set (see also Figure 4)
(b) a buffer BUF for the value of the current record
(c) a buffer VBUF for the value of the next record in sequential reading
(d) status variables.

The class FILDAT is the common superclass of ACCSEQ, ACCTIS and the file modules. In this way the compound modules consisting of objects of FILDAT, ACCSEQ/ACCTIS and a file module can be reference from the VARDAT module.

The class VARDAT. The class VARDAT (VARiable DATa) contains
(a) an array REFFILDAT with references to all FILDAT objects in the application
(b) a set of procedures which are common to ACCSEQ, ACCTIS and all file modules
(c) a reference to the DYNARR object, which contains the text values of several text attributes of FLDVAR, FLDARR and RECORD objects.

The batch application module has access to the VARDAT module. By means of the REFFILDAT array the VARDAT module provides access to the connected FILDAT modules. As has been described previously, a FILDAT module has access to the set of RECORD objects, which in turn have access to the FLDVAR and FLDARR objects. This means that, except for the references to the file modules, the application module can be written in a completely data-independent way: the same application module may be connected to any set of files, because an application module normally has access to all objects representing files, records and items.

For most of the practical cases, however, this way of access is too cumbersome. In cases where complete data independence is not necessary, the file module provides a short cut from the application module to the FLDVAR, FLDARR and RECORD objects.

Generation of file modules

The Simula source text of a file module is generated automatically from meta-data contained in the DD. A schema of the operations and programs that are involved in the generation of a file module is shown in Figure 6. Descriptions of file types are entered into the DD by way of the interactive program EDITDD (EDIT Data Dictionary). Such a description in the DD is not specific for a data set, but describes only the structure of a certain type of file, namely the record types, the key item and the other items in the record types. The program EDITDD requests in a screen dialogue with the user all specifications of a specific file type.

The file module generating program GENFIL (GENerate FILe module) is also an interactive program; the user enters specific data on the data set, e.g. its name, the name of the file type and the file organization. The description of the file type is retrieved from the DD. GENFIL produces the Simula source code of the desired file module and calls the Simula compiler, which compiles the file module code.

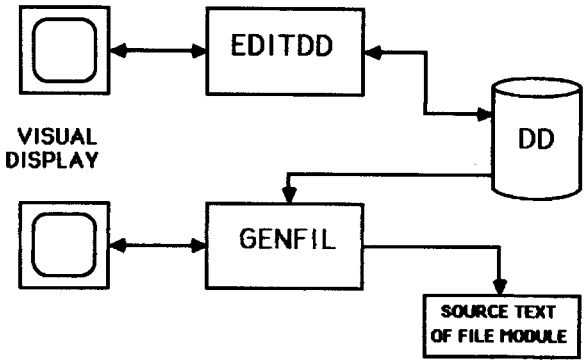

Figure 6. Schema for the generation of a file module

Generation of a batch application program

The modules VARDAT, FILDAT, ACCSEQ, ACCTIS and DYNARR are TUBA system modules; they are available as standard classes in the TUBA system library. FLDVAR, FLDARR and RECORD objects are relatively small modules, which are created by the corresponding file module.

The generation of file modules has been described in the previous subsection. The batch application module is a hand-coded module. A frame for a batch application module can be generated by an interactive system program. A frame contains a comment block, provisions for connecting the appropriate file modules and statements for opening and closing the data sets. The rest of the code has to be inserted into the frame; usually the size of the hand-code is modest in comparison with the total size of a generated application.

The whole set of modules representing an application is created and linked together by the Simula run-time system at the beginning of the execution of the application program.

Except for the possible addition of some Simula code to the program frame, TUBA applications are specified completely interactively; i.e. there is no meta-language in which a user has to describe his application.

SCREEN ENVIRONMENT AND INTERACTIVE PROGRAMS

Introduction

For the construction of interactive application programs the TUBA screen environment is available. The most important modules in the screen environment are shown in Figure 7, which represents a rough outline of the modular composition of an interactive program. The different types of modules will now be discussed briefly:

1. SCRNWA (SCReeN Work Area) is the central module of the screen environment. It contains procedures for screen handling, which may be called from the application module (see the Appendix). Additionally it owns several peripheral modules

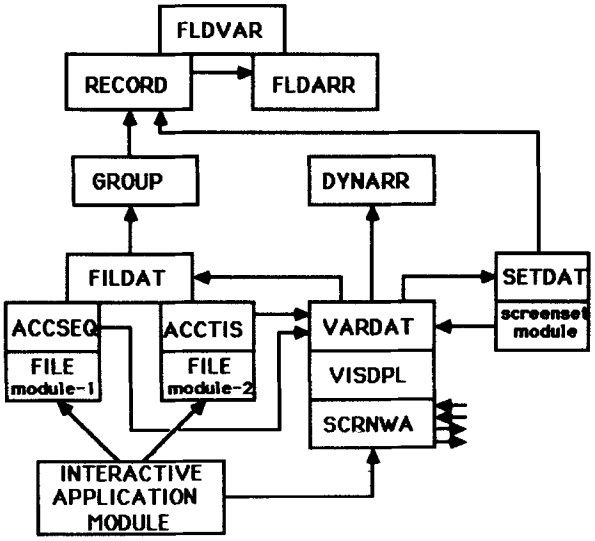

Figure 7. Schema of classes in an interactive application

for the interpretation of key commands; those modules are not shown and will not be discussed here.

2. VISDPL (VISual DisPLay) is the prefix module of SCRNWA. It contains procedures which implement operations on the visual display, such as cursor movement and screen painting. As is shown in Figure 7 VISDPL is prefixed by VARDAT. In this way each type of module belonging to the TUBA file environment is included within the scope of the module SCRNWA, such that the screen environment is an extension of the file environment.
3. A *Screen set* module is a type of module related to a specific set of screens. Such a screen set module can be generated automatically. It will be discussed in more detail under the 'screen sets' heading below.

Screens

Screen items

In TUBA a screen consists of a set of screen items. The following types of screen items may be distinguished:

(a) text constants
(b) variables: data items represented by objects of the class FLDVAR, which are known within the scope of the application module. Using them in a screen offers the facility for entering and reading data via the visual display.

In order to keep screens program-independent, screen variables are declared and generated outside the application module. Two different types of variables occur in a screen:

(i) file variables: items of a particular record type of a particular file for which a FLDVAR object has been declared and generated in the corresponding file module.
(ii) screen set variables: items for which a FLDVAR object has been declared in the corresponding screen set module.

Screen types

The screen environment supports screens of three different types:

1. *Message* screens are only suited to be displayed.
2. From a *selection* screen the user may select one or more screen items.
3. In an *update* screen the user may update one or more screen items of the variable type.

Figure 8 shows an example of an update screen, which contains five text constants, four file variables and one screen set variable (of the type boolean). The Figure illustrates the direct connection between items in the file buffers and a screen.

Screen data structure

All relevant data about the layout of a screen are stored in the SD at creation time. The screen environment loads a screen from the SD into main memory at execution time, when it is needed by the application program. For fast manipulation of screen attributes, loaded screens are represented by class objects of a specific class (not shown in the Figures). Such class objects are accessible by references from the SCRNWA module.

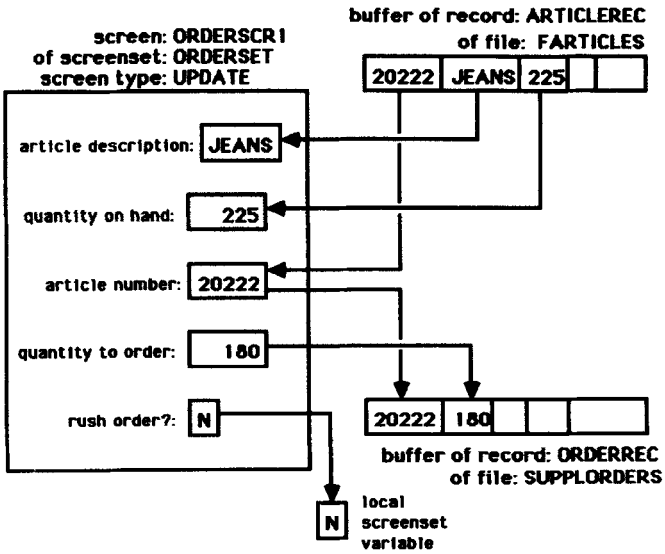

Figure 8. Example of an update screen

Screens can easily be created and modified via so-called screen manipulation programs. For each screen set such a screen manipulation program is generated automatically.

Screen sets

A screen set contains a set of related screens, e.g. all the screens which are used in an application program. For each screen set a screen set module may be generated. A screen set module has much similarity with a file module. It contains the declarations and generating instructions for the screen set variables of the screen set.

Analogous to a file module, each screen set module is prefixed with a system module of type SETDAT (SET DATa). Such a common superclass is necessary to arrange that each screen set module can be referenced via its prefix class by the VARDAT module.

Generation of an interactive application program

Interactive programs are generated in the same way as batch programs. The modules VISDPL, SCRNWA and SETDAT are system modules. A screen set module is generated in the same way as a file module. For the application module a frame can be generated; such a frame contains provisions for connecting the appropriate file and screen set modules.

REPORT ENVIRONMENT AND REPORT PROGRAMS

Introduction

The TUBA system offers facilities to simplify considerably the design and implementation of report programs. The report facilities are incorporated in the module REPWRI (REPort WRIting). REPWRI contains many procedures and data structures which may be used both by the application programmer and the TUBA system (see the Appendix). Figure 9 shows the modular structure of a report program. This structure differs slightly from the structure of other types of programs, e.g. interactive programs. The application module is prefixed with REPWRI, which in turn is prefixed with the Simula system class SIMSET for processing list structures. Simula allows SIMSET only to be connected to a class by prefixing, i.e. external referencing is illegal.

Each report program processes a specific file (the so-called main file) in sequential order. One or more secondary files may be accessed for retrieval of additional data needed for the report; e.g. when the primary file contains an item part number the name and price of the part may be retrieved from a directly-accessible master file containing records describing parts. In Figure 9 file modules of a main SEQ file and one secondary TIS file are shown.

The structure of a report program

The application module of a report program has a fixed structure. It may be divided into the following three parts:

1. The local declarations. In this part local report variables of type FLDVAR may be declared in order to keep values, which are used in the report. These values may

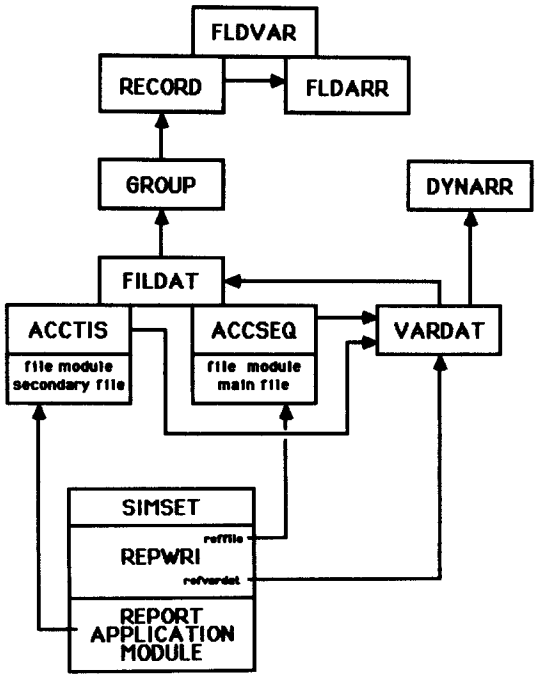

Figure 9. Schema of classes in a report program

be either results of numerical functions or values which are calculated in the processing part of the program.

2. The report description. In this part a static description of the layout and the contents of the report is given. The description includes parameters for page size, page layout and specifications for the different types of print lines.
3. The processing part. In this part the main file is processed sequentially and the report is generated; this part may also contain statements for accessing secondary files.

Group structure of files

In order to simplify the definition of reports, the TUBA system offers the facility to describe the structure of a particular file in terms of the JSP programming technique.[12] In such a description the contents of the data set are divided into a number of groups of consecutive records. Each of these groups may be subdivided into subgroups, etc. In this way a tree of groups of different types may be constructed, which reflects the structure of the data set. Figure 10 shows an example of a file structure, which contains transactions of two different types: orders and returns. Each transaction consists of one header record followed by an iteration of one or more detail records.

With the program EDITDD the user may specify one or more different group descriptions for each particular file type. When a file module is generated for a particular data set one of these group descriptions may be incorporated in the file module.

During the execution of a report program the report environment keeps track of the processing of the main file and maintains the trace of the current groups in the related

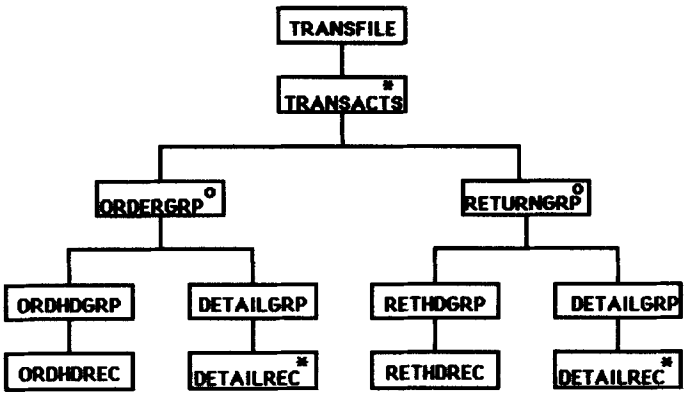

Figure 10. Example of a JSP group structure

group structure. The group structure of the data set is represented in main storage by way of objects of the class GROUP, which is part of the file environment of the TUBA system. The relevant set of group objects is created by appropriate statements in the file module, analogously to the creation of objects of the classes FLDVAR, FLDARR and RECORD.

A number of different types of print lines may be connected to each group type. Print lines are distinguished into heading, detail and footing lines, depending upon the status of the corresponding group of records.

The report environment of the TUBA system

The generation of a report

The generation of a report is controlled by the procedure GENERATEREPORT, which should be called after each read operation on the main data set. This procedure performs the following operations in the indicated order:

1. By comparing the next trace and the current trace it is determined which groups have just become current; for those groups the related heading lines are written.
2. The current trace is made equal to the next trace and a 'fresh next trace' is determined.
3. All numerical functions that are specified in the report specification and that are related to groups of the current trace are updated.
4. The detail lines of the current trace are written, including any necessary page heading/footing lines, when a new print page starts.
5. For those groups which do not remain current in the next trace, footing lines are written.

The implementation of print lines

As is shown in Figure 11, a print line is implemented as a chain of class objects of the class ELEMENT, headed by a class object of the class PRINTLINE. The prefix classes

HEAD and LINK are members of the Simula system class SIMSET. The classes ELEMENT and PRINTLINE are internal classes of the class REPWRI. A PRINTLINE object contains information about the positioning of the print line on the print page. An ELEMENT object contains information about an item of a print line, such as its type (text constant or variable), and references to the variable and attributes which control the position and format of the item in the report.

The report writer also includes facilities for numerical functions; more information about them is given in References 1 and 3.

Generation of a report program

Report programs are generated analogously to batch and interactive programs. A frame for a report program can be generated automatically. The generating program asks the user to indicate the main file and the secondary files, in order to include the related file modules in the frame of the report application module.

At present effort is being devoted to extending the TUBA report writer so that the layout of even complicated reports including tables and arithmetic functions can be specified completely in an interactive dialogue with the user. In this way it should be possible to generate report-writing programs without any hand-coding.

THE ROLE OF THE CLASS CONCEPT IN TUBA

Representation of objects

User-known entities are represented by objects in TUBA system programs as well as in generated application programs. This applies to entities such as items, records, groups, data sets, screens, screen sets, print lines and print pages. Each type of object

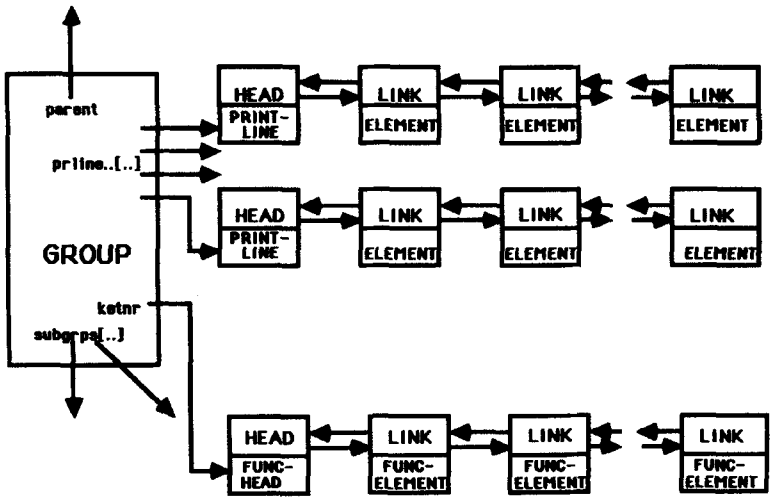

Figure 11. Data structures of print lines and functions in the report writer environment

is represented by a Simula class. Benefits of this object-based programming technique are the following:

1. Data and procedures are located in relatively small modules (objects), which can easily be designed and modified.
2. Modifications in object descriptions cause only local changes in the software.
3. Objects have well-defined interfaces with the environment.
4. Objects which are conceptually related may share attributes by inheritance.
5. Compiled versions of object modules may be used in various programs, so that for the construction of any application only a limited amount of code need be compiled.
6. Attributes of objects are available during execution of the program and may be inspected by the software as well as by the user. This also holds true for the generated application programs.

Inheritance

A class B which is logically related to another class A because B is an extension of A can be described as a subclass of A by the prefix mechanism. Prefixing is the Simula implementation of the inheritance mechanism between types of objects. Examples of prefixing in TUBA are the classes FLDARR and RECORD, which are subclasses of FLDVAR. In this way the attributes of FLDVAR are inherited by the classes FLDARR and RECORD.

Virtual procedures

When two or more subclasses of the same parent class share a procedure with a common name, but with different implementations in those subclasses, then the construct of the *virtual procedure* should be used.

As an example we consider the classes ACCSEQ and ACCTIS, with the common parent class FILDAT (see the earlier subsection on 'The representation of the file environment'). The two classes share several procedures, e.g. the procedure READNEXT for reading the next record. A program may read sequentially a SEQ as well as a TIS data set, because READNEXT has been declared as a *virtual procedure* in FILDAT. The implementations of the READNEXT procedures in ACCSEQ and ACCTIS are different, but on the FILDAT level they share the same name.

Mutual referencing of classes

In several cases it is desirable for modules of different types to be able to reference each other. However, the following construct is not acceptable in Simula:

```
EXTERNAL CLASS B;          EXTERNAL CLASS A;
CLASS A;                   CLASS B;
⟨body of class A⟩          ⟨body of class B⟩
```

The reason for this construct not being acceptable is that when class A is compiled class B should exist (have been compiled); however, in order for class B to be compilable, class A should exist. In such a case the following construct should be used:

```
CLASS C;
⟨body of class C⟩;

EXTERNAL CLASS B;
C CLASS A;
⟨body of class A⟩;
```

```
EXTERNAL CLASS C;
CLASS B;
⟨body of class B⟩;
```

In this construct a module of class B may reference a module of class C, and a module of class A may reference a module of class B. The classes should be compiled in the order C, B, A. This construct also permits class A to stand for a set of classes, if all those classes share the same prefix class C. An example of the application of the construct is the case where the class FILDAT is the prefix class of ACCSEQ and ACCTIS. ACCSEQ and ACCTIS modules have a reference to VARDAT; a VARDAT module has an array REFFILDAT of references to the connected FILDAT modules (see Figure 5).

Coroutines

Two coroutines which call each other may be implemented as a pair of class objects with mutual references. A class object may suspend its execution by calling the standard Simula procedure DETACH or by activating another object with the standard procedure RESUME. A class object may be activated from the module by which it has been created by means of the standard procedure CALL, or by a RESUME procedure call from another module. This feature has been applied to implement the command interpreter modules in the screen interface. After creation by the SCRNWA module a command interpreter module detaches itself and may be activated again by a CALL procedure call from the SCRNWA module.

Object-based implementation languages

Many constructions used for implementing the TUBA system can be applied in languages other than Simula. It is desirable that such a language contains a mechanism for representing objects or abstract data types. From this point of view languages such as Modula-2 and Ada may be acceptable. However, most of the languages which support the representation of abstract data types lack the inheritance mechanism. C++[10] is an extension of the language C, which supports classes with about the same functionality as Simula. In Smalltalk-80[11] the ideas of object-oriented programming have been extended more rigorously to all data types and control structures.[7] At present we are investigating the merits of C++ and Smalltalk-80 in relation to application generation.

CONSISTENCY IN THE TUBA SYSTEM

The programs manipulating the dictionary files, such as the program EDITDD and the screen manipulation programs, contain specific tests to maintain consistency within the dictionary, e.g. a specific item is described only once though it may occur in many files and screens.

Inconsistency may arise when the contents of the dictionary are modified, e.g. a file description is changed and a new file module is generated, compiled and linked to an application, but a data set with an old format still exists. In such a case the system generates a user warning when the application starts execution; the TUBA system provides utility programs to perform data set conversion. The TUBA system does not contain provisions to check directly the consistency between generated modules and the dictionary. In our experience this has not been a great problem.

CONCLUSION

The technique of object-oriented programming allows transparent structures for data and programs. It has been used in TUBA for the application generation system as well as the generated software. The modular construction of the software yields the benefit that for an application usually only a limited number of small modules have to be compiled, so that for the development of an application only minor compiler operations are needed.

The integrated dictionary, containing meta-data on files, layout of screens and descriptions of programs, promotes the consistency of specifications and the generated applications. The dictionary is also used as a tool in the implementation of user-friendly features of the TUBA system.

APPENDIX: USER INTERFACES OF SYSTEM CLASSES

This appendix describes the most significant attributes (procedures, functions and variables) which are available for the user of the TUBA system. The attributes are ordered according to the classes in which they have been declared. The parameter refrecord used below is a reference to a record-object. In cases where procedures and functions have a large number of parameters the parameter list is indicated as (...).

Class ACCSEQ (sequential file access)

openf — opens the file for processing;
closef — closes the file for processing;
readnext — reads the next record from the file;
writenext (refrecord) — writes the next record of the type indicated by refrecord in the file;
beof — boolean value indicating the end of file status.

Class ACCTIS (direct file access)

openf — opens the file for processing;
closef — closes the file for processing;
readnext — reads the next record from the file;
readprev — reads the previous record with the next-lower key value from the file;
readdirect (keyvalue) — reads the record with key=keyvalue;
readapprox (keyvalue) — reads the record with key=keyvalue if such a record exists,

otherwise a record with the next-higher key value is read; if the record with key=keyvalue exists the boolean function readapprox returns **true** else **false**;
readifpresent (keyvalue) — reads the record with key=keyvalue if it exists; if the search succeeds the boolean function readifpresent returns **true** else **false;**
search_placekey (keyvalue) — searches for the record with key=keyvalue without reading; if the record exists the boolean function search_placekey returns **true** else **false;**
writedirect (refrecord) — writes the current record of the type indicated by refrecord into the file; this implies the addition of a new key value to the file;
rewrite (refrecord) — rewrites the current record of the type indicated by refrecord into the file; this operation is used for modification of existing records;
delete (keyvalue) — deletes the record with key=keyvalue from the file;
delete_range (keyvalue1, keyvalue2) — deletes all records with keys in the interval keyvalue1:keyvalue2.

Class VARDAT

The class VARDAT contains several procedures which are called by file modules:

cr_fldvar(...) — creates an object of the class FLDVAR;
cr_fldarr(...) — creates an object of the class FLDARR;
cr_record(...) — creates an object of the class RECORD;
cr_group(...) — creates an object of the class GROUP.

Class SCRNWA (screen operations)

dplscr (screenname) — (*displ*ay *scr*een) makes the screen with identifier screenname current and displays it on the visual display;
dplcurscr — (*displ*ay *curr*ent *scr*een) displays the current screen;
dplflds (screenname, fieldnbr1, fieldnbr2) — (*displ*ay *fields*) makes the screen with identifier screenname current and displays the items with ordinal numbers fieldnbr1: fieldnbr2;
dplcurflds (fieldnbr1, fieldnbr2) — (*displ*ay *curr*ent *fields*) displays the items with ordinal number fieldnbr1: fieldnbr2 of the current screen;
rstscr (screenname) — (*reset scr*een) resets the screen with identifier screenname, i.e. initializes the items with current or default values.
Analogous to the display functions there are the following additional reset functions: rstcurscr (*reset curr*ent screen), rstflds (*reset fields*) and rstcurflds (*reset curr*ent *fields*).
updscr (screenname) — (*upd*ate *scr*een) makes the screen with identifier screenname current and lets the user fill in or change the items in the screen.
Analogous to the display functions there are the additional update functions: updcurscr (*upd*ate *curr*ent *scr*een), updflds (*upd*ate *fields*) and updcurflds (*upd*ate *curr*ent *fields*).
selscr (screenname) — (*sel*ect *scr*een) makes the selection screen with identifier screenname current and lets the user select one or more items (depending on the specifications of the selection screen).

Analogous to the display functions there are the additional select functions: selcurscr (*sel*ect *cur*rent *scr*een), selflds (*sel*ect *fields*) and selcurflds (*sel*ect *cur*rent *fields*).
setprotection (fieldnbr) — protects the item with ordinal number fieldnbr against update or selection operations;
rstprotection (fieldnbr) — reset (clear) the protection of the item with ordinal number fieldnbr;
selected (fieldnbr) — returns **true** if the item with ordinal number fieldnbr has been selected in the current selection screen, else **false;**
clrscr (screenname) — (*clear scr*een) deletes the display of the screen with identifier screenname from the visual display;
Analogous to the display functions there are the additional clear functions: clrcurscr (*clear cur*rent *scr*een), clrflds (*clear fields*) and clrcurflds (*clear cur*rent *fields*);
message (messagetext) — displays a message denoted by messagetext on the visual display;
wait — waits for response from user, e.g. after display of a message.

Class REPWRI (report writer functions)

data_phpage(...) — specifies the physical page size, e.g. the size of the paper in the line printer;
data_lgpage(...) — specifies the size of the logical page, e.g. an invoice; a logical page may be larger or smaller than the physical page;
lgpage_map(...) — describes the mapping of logical pages to physical pages;
fixedcolumn (columnnbr, start, width) — specifies a column in a table of a report; the ordinal number of the column is specified by columnnbr, the leftmost position of the column is start, width is the size of the column measured in character positions;
autocolumns (leftmargin, rightmargin, gap) — specifies a table in which the positions of the columns are calculated by the system; only two of the three parameters may be specified in a specific parameter call;
createlines (refgroup, linetype) — specifies a type of report line; refgroup denotes one of the groups specified in the file module of the main input file; linetype may have one of the values: headings, footings or details, specifying the creation of a report line at the beginning of the group, at the end of the group or for each input record in the group.
linecontrol(...) — specifies the position of a report line on the logical page;
textstr(...) — specifies a text string constant to be printed in a report line;
variable(...) — specifies a variable to be printed in a report line;
funcvariable(...) — (*func*tion *variable*) specifies a variable to be calculated from other variables; the expression is specified in the parameters;
generatereport — controls format and position of the next line to be printed.

REFERENCES

1. F. A. van Hoeve, 'Design and implementation of a report program generator using abstract data types', in H. J. Schneider (ed.), *Proc. Int. Computer Symposium 1983 on Application Systems Development*, Nürnberg, Teubner, Stuttgart 1983, pp. 382–401.
2. F. A. van Hoeve and R. Engmann, 'The TUBA-project : a set of tools for application development and prototyping', in R. Budde, K. Kuhlenkamp, L. Mathiassen and H. Züllighoven, (eds), *Approaches*

to Protyping, Springer-Verlag, Berlin, 1984, pp. 202–213.
3. F. A. van Hoeve and R. Engmann, 'Design and implementation of the TUBA application generation system', *Memorandum INF-85-23,* University of Twente, 1985.
4. N. A. Rin, 'An interactive applications development system and support environment', in H.-J. Schneider and A. I. Wasserman (eds), *Automated Tools for Information Systems Design*, North-Holland, Amsterdam, 1982, pp. 177–213.
5. J. Martin, *Application Development Without Programmers,* Prentice-Hall, Englewood Cliffs, N.J., 1982.
6. D. Martland, S. Holloway and L. Bhabuta, *Fourth Generation Languages and Application Generators,* The Technical Press, Unicom, 1986.
7. S. Cook, 'Languages and object-oriented programming', *Software Engineering J.***1**, 73–80 (1986).
8. G. M. Birtwistle, O.-J. Dahl, B. Myhrhaug and K. Nygaard, *SIMULA BEGIN*, Studenlitteratur, Lund, 1973.
9. G. M. Birtwistle, L. Enderin, M. Ohlin and J. Palme, *DECsystem-10/20 SIMULA Language Handbook,* Stockholm University Computing Center, Stockholm.
10. B. Stroustrup, *The C++ Programming Language,* Addison-Wesley, Reading, Mass., 1986.
11. A. Goldberg and D. Robson, *Smalltalk-80, The Language and its Implementation,* Addison-Wesley, Reading, Mass., 1983.
12. M. A. Jackson, *Principles of Program Design,* Academic Press, London, 1975.