A Guide to Natural Naming

Daniel Keller
ETH, Projekt-Zentrum IDA
CH-8092 Zürich, Switzerland

Summary

The naming scheme described in this paper is a set of hints and guidelines on how to select names in a procedural programming language. The analysis of the structure of identifiers shows that type identifiers are the core of most names. Therefore the starting point for a meaningful naming is the set of **names for the types** within a program. Short class names should be chosen for these. The other identifiers can now be derived from the type names. Additional hints are given to complete the naming scheme. This naming convention is perceived as a real aid in finding good names, not as a restricting set of rules. It has been taught and used in projects with success in the past two years. The resulting programs are highly readable.

Introduction

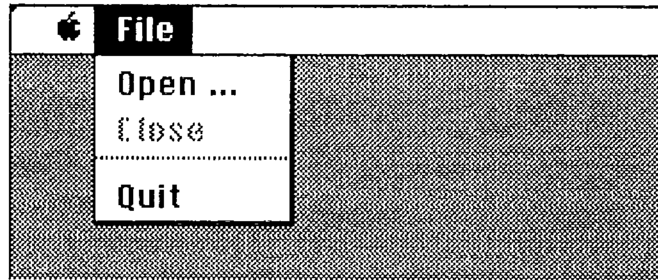
Every programmer faces the problem of having to select meaningful names for identifiers in a program, yet hardly a text book on programming discusses this subject. A lot of research has gone into the structure of programs, but not into the structure of names.
Over the years it has been realized that programs are being read primarily by people and that satisfying the compiler's requirements on syntactical correctness is only a necessary but not a sufficient criterion. Programs must also be readable in order to be easily understood by humans. Consistent indenting, commenting, and naming is an economical necessity since unreadable programs quickly become too expensive to maintain.

So far, finding good names has been a matter of luck and intuition – if not restricted by a rigid naming convention of the kind that defines three-letter prefixes, a five-letter free naming space and a dollar sign suffix for constants. Such a naming convention inevitably leads to endless battles against its restrictions yielding unreadable variable names like `PRS_XCNVT`. On the other hand, without naming convention one is tempted to use the letters of the alphabet as one-character identifiers (`i, x, p, r, s`), and, after running out of letters, using two-character identifiers like `ii` or `xx`.

This paper proposes a way out of this dilemma: a naming convention which helps in choosing good names without imposing any restriction. In the true sense it is not a naming convention but a name structuring guide.

Please note that the naming scheme described here only applies to strongly typed languages which allow arbitrarily long identifiers, e.g. Pascal, Ada, C, Modula-2, and even some BASIC dialects, but not to standard FORTRAN or Minimal BASIC. The convention is only partially applicable to declarative languages like PROLOG or to a language like LISP where only functions and no procedures are known.

As an introductory example, two code fragments are presented, the first as the "normal" example, the second as the more readable one.
The purpose of one procedure in these fragments is to build up a Macintosh-like menu bar with one menu having three command entries (see picture below). The other procedure asks the user for a file name with the file selector box and opens the file for reading.



Fragment 1 (the "normal"):

```
VAR
   f:    file;
   fnam: ARRAY[0..31]OF CHAR;
   fl:   mnu;         (* the "File" menu *)
   op, cl, q: cmd;    (* the Open, Close, and Quit commands *)


PROCEDURE newmnu;              (* build up the menu bar *)
BEGIN
   mnuclear;                 (* makes an empty menu bar *)
   (* make one menu with three commands in it *)
   newM( fl, "File", sel );      (* sel = selectable *)
   newC( fl, "Open ...", sel );
   newC( fl, "Close", nosel );
   sep( fl );     (* adds a separator line in the menu *)
   newC( fl, "Quit", sel );
END newmnu;


PROCEDURE fopen;         (* open the file for reading *)
VAR
   ok: BOOLEAN;
BEGIN
   fselinput( fnam, ok );       (* ask for file name *)
   IF ok THEN
      open( f, fnam, rd );
      cmdsel( fl, cl );       (* enable close command *)
      cmdnosel( fl, op );     (* disable open command *)
      fread( f );
   END;
END fopen;
```

The above example looks like ordinary, well documented code, doesn't it? If you do not fully understand what this code is supposed to do, have a look at the second fragment - which does exactly the same - where the names have been chosen with more care:

Fragment 2 (the "readable"):

```
VAR
   InFile:     File;
   InFileName: ARRAY[ 0..31 ] OF CHAR;
   FileMenu:   Menu;
   OpenComm:   Command;
   CloseComm:  Command;
   QuitComm:   Command;


PROCEDURE BuildUpMenuBar;
BEGIN
   MakeEmptyMenuBar;
   AddMenu( FileMenu, "File", Enabled );
   AddCommand( FileMenu, OpenComm, "Open ...", Enabled );
   AddCommand( FileMenu, CloseComm, "Close", Disabled );
   AddSeparator( FileMenu );
   AddCommand( FileMenu, QuitComm, "Quit", Enabled );
END BuildUpMenuBar;


PROCEDURE OpenAndReadFile;
VAR
   ok: BOOLEAN;
BEGIN
   ShowFileSelectorBox( InFileName, ok );
   IF ok THEN
      Open( InFile, InFileName, ReadOnly );
      EnableCommand( FileMenu, CloseComm );
      DisableCommand( FileMenu, OpenComm );
      ReadData( InFile );
   END;
END OpenAndReadFile;
```

This second example is clearly easier to read, even without the comments. The commands are more obvious - provided that the referenced and imported procedures do what their names imply. Note that the structure of both fragments is identical, only the names have been replaced.

The remainder of the paper explains *why* the second code fragment is easier to read. The structure of some good names is analyzed and rules and guidelines for selecting names are presented.

The Structure of Names

In a strongly typed procedural programming language names must be given to identifiers of programs, modules, constants, types, variables, and procedures. Usually the sequence in which the names of the various objects are chosen is random, but it should not be. There is a clear advantage in choosing the names in a certain order. The following section discusses the structure of names for the different objects. After that it will become clear that the proposed sequence in naming is preferable.

Type Names

The simplest and shortest names are the names for type identifiers. Therefore they must be chosen before any other name, especially before the names for variables of this type. Type names should be short generic class names. They may also be composed of two such class names.

Examples: `Table, Error, State, Word, Name, File, Address, Graph, Title, Menu, StateTable, FileName, TableIndex.`

Procedure Names

A procedure is (literally) called by its name which stands for a group of statements to be executed. Therefore the name should express the implied action ("DoThis!") by including a verb. Since procedures usually operate on a specific type, the structure "verb + type name" is best suited for a procedure name.
(A special note for non-English programmers: the verb should be in its imperative form - which in most languages is different from the infinitive form).

Examples: `StoreWord, DisplayError, ShowPrinterStatus, PrintPage, InvertMenuTitle, OpenWindow, DrawLine, PrintAddress, GetFirstElement, CheckMachineState, FindName`

Variable and Function Names

In strongly typed languages a variable is of a particular type. Therefore the structure "adjective + type name" for variable names is an obvious suggestion. It does not have to be an adjective, it can also take on the more general form "qualified type name". This convention also applies to function procedures (subroutines returning a value) because they are used like variables within expressions.

Examples: `FirstState, NextState, LastElement, BigWindow, HomeAddress, RunningTitle, HeadPointer, TitlePage, EndOfList, MaxLength, CurrentSymbol, OptimalLevel, ScreenWidth`

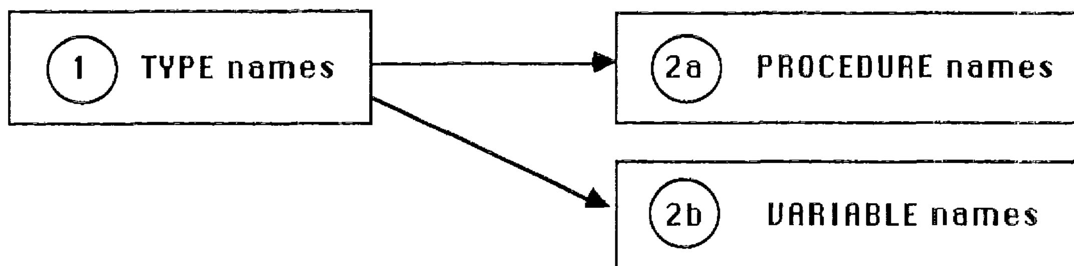
Constants

They often describe a limit within a program. In these cases it is appropriate to use the prefix "Max" in connection with the type name. Otherwise treat the names for constants like variable names.

Examples: `MaxLineLength, MaxLinesPerPage, MaxNoOfEntries, MaxBitmapSize, MaxOpenWindows, MinWindowWidth, DefaultRepeatRate, FastClick, SlowClick`

The Naming Order

It is a logical consequence of the naming structure described above that the **type names must be chosen first**. This is no surprise when looking at object oriented programming or the technique of abstract data types.
After having chosen the type names one can start to name the procedures, variables, and constants. Finally, the program and module names can be chosen; they are the least important for readability and conflicting names.



This sequence in naming the programming objects is important: if one has named the variables first (which is usually the case) then the best names for the types are already gone. Example:

> A variable `State` has been declared of type `StateType`. This is fine as long as only one variable of this type is used. The naming problem starts when a second variable of the same type is needed. First shot: `State2`. This is certainly better than naming it `'S'`, but why not pick really good names like `CurrentState, LastState, NormalState, ErrorState`, etc. for the variables and name the type `State` as described above?

General Hints for Naming

The following is a collection of hints and guidelines. Some of these can be found in Ledgard´s excellent book (1) about professional programming, some of the hints are "folklore" with unknown sources.

The most important criterion when choosing a name is: how easily can *another programmer* understand the program (and not only yourself). If understanding a name wasn't important we could name the variables `v1, v2, v3, v4 ... v568` couldn't we?

o **Names must be pronounceable.**

Make long names, do not truncate as if you still had to program in FORTRAN. As a "rule of mouth": *If you can´t read a name out loud, it´s not a good name.*
Do not hesitate to use long names, even if you are not a top typist. Use a good editor, it can help you with an easy cut and paste for identifiers.
Sometimes it is argued that long names make the lines too long. This may be true, but I have yet to see a program where the names are really too long. The usual case is that the names are too short (who has not pondered for hours about the meaning of names like `x, ir2q, n, p0, p1, xx, grp, cfv`?)

Examples: `GroupID` instead of `grpID, NameLength` and not `namln, PowersOfTwo` and not `pwrsOf2, ResetPrinter` and not `RSTPRT.`

o **Use capitalizing (or underscores) to mark the beginning of a new word within a name.**

Capitalizing (or using underscores) makes the names easier to read. I prefer capitalizing because it does not lengthen the names unnecessarily.

Examples: `LatestEntry, NextState, TopOfStack`
or: `latest_entry, next_state, top_of_stack`

o **Abbreviate with care.**

Abbreviations always carry the risk of being misunderstood (does `TermProcess` mean `TerminateProcess` or `TerminalProcess`). Usually they are also hard to pronounce ("`NxtGrp`").
Only use commonly known abbreviations, like the "ID" in `ProcessID`, or abbreviations which are known and agreed upon within a company or group. In the

latter case they should be documented for each project so that all programmers use the same abbreviations consistently.

Abbreviate a name only if it saves more than three characters. Take the famous `JMP` and `MOV`: one letter has been saved, what for?

Examples: `error` and not `err`, `next` and not `nxt`, `name` and not `nam`,
but: `MaxLineLength` is probably better than `MaximumLineLength`.

- **Do not use digits within a name**

  Numbers within a name are easy to misread: `0 and O, 1 and l, 2 and Z, S and 5.`
  If a program really requires three pointers `p1, p2, p3` for instance, should they not better be declared as an `ARRAY[1..3] OF POINTER TO ...` instead (or named `PreviosPtr, CurrentPtr, NextPtr` if this is what they mean)?

- **Boolean variable and function names should state a fact that can be true or false.**

  This is easy to achieve with the inclusion of "is" in the name.

  Examples: `PrinterIsReady, QueueIsEmpty,` or simply: `done, IOfailed`
  (note how naturally this reads: `IF PrinterIsReady THEN ...`)

Conclusions

This naming scheme has been used in several programming courses and in two projects so far. It has proven to be very useful. The programs became easier to read and understand.

As an interesting side-effect the good names make many comments superfluous. Comments appear only in the code to flag something unusual or not obvious (apart from the module header and the comments to each declaration of a constant or a variable). This is quite contrary to the common belief "the more comments the better".

It is also interesting to note that people reading such programs usually find them easy to read, but cannot see *why* they were easy to read. The naming scheme goes unnoticed for the reader, a clear advantage over a convention which needs an explanation.

Best of all, it is not a pain to use (like so many other programming guidelines, which have to be *enforced*); this one is a real help in finding good names.

Limitations

The naming scheme does not help very much for naming record fields and formal parameters of procedures. One idea can serve as a guideline: the scope of a name influences its length. Global variables, procedures, or constants have a bigger scope than local variables or formal parameters. A bigger scope requires that the name be longer in order to be uniquely identifiable. Therefore local variables, formal parameters and record fields can have shorter names.

Additional conventions - like using a small first letter for variables and types, a capital first letter for procedures and constants - can be used on top of this one. However, when the names are chosen carefully and using the above described suggestions, one does not need such artificial conventions, the names clearly express what they stand for.

Literature

(1) Henry Ledgard with John Tauer: "Professional Software, Vol II, Programming Practice", Addison-Wesley, 1987