Standards for Modular Programming in 4D

created 10/13/99, last modified 10/13/99

Lincoln Stoller, Ph.D.

Introduction

The Purpose of Standards

The first purpose of this document is to encourage 4D programmers to follow general standards in the design of modular code libraries written in 4th Dimension. This effort parallels current developments in the 4D language, which will soon supply methods for creating code libraries. It also follows increasing interest in the market place in the use of preexisting code.

Modular coding, and especially object oriented code methodologies, are an increasingly active topic in the software engineering industry. While the 4th Dimension language is not an object oriented language, future releases of the language now on the drawing board are moving toward becoming more object oriented.

The second purpose of this document is to start the discussion of what elements should be included in, and what disciplines should be followed by, code that is created for reuse. This is especially pertinent to code that is packaged and sold as a code library.

The 4D community is still at an early stage in the establishment of modular design standards. With this article I hope to help codify some basic concepts and expectations. I hope this will be useful to those people who are creating and using 4D source code libraries.

The Use of Modules

Modules are sets of methods, forms, tables, and other objects native to the 4th Dimension programming environment that can be installed in a 4D database in order to add functionality. 4D modules can consist of any set of 4D objects that can be moved into, or recreated in another 4D database.

In the future modules may be implemented as “libraries”. This means they will be more encapsulated and have more independence that is currently provided within the 4D language. Future versions of the 4D language are scheduled to provide new features of this kind.

I have put together the following standards I the hope that they are general enough to apply to the design of modular code in the future implementations of 4D.

Module Interface

The two main problems in the creation of modular code correspond to the two aspects of all computer programs. These are the internal structure and the interface. A module must have the internal structure that solves the problem for which it was created. This structure must be robust, tractable, and able to evolve.

A module’s internal structure comprises the means by which it provides a solution to some computer problem. If a module’s actions are completely internal to the module, then they are not subject to modular design standards as envisioned here. The hidden means by which a module effects its solution is not subject to any conformity with standards.

A module must also interact with an outside code environment, as well as with other programmers working in an outside code environment. This interaction constitutes the module’s “interface”. It is the module’s general interface that is subject to design standards. The interface is that aspect of the module that is “seen” by the programmer, the user, or both. In a general sense a module’s interface consists of all the ways in which the module interacts with outside data, both intentionally and unintentionally.

This interface can be as general as shared block of memory or a set of common tables, or as specific as a graphical form through which the developer communicates with the module much as an end-user would communicate with a compiled application.

In most cases modules support a variety of “windows” consisting of both shared tables, variables, methods, and forms. A module can also have its own developer interface though which the developer can communicate with or control the actions of the module.

Basic Module Structure

A module consists of three types of private, shared, and public code. Different guidelines apply to each type of code.

Private Code

This is the modules internal structure consisting of programming elements that are never seen or accessed by the user of the module. In current versions of 4D there is no way to hide internal structure except through the use of a compiled plug-in. This will change in the future when it will be possible to hide parts of code libraries that are written entirely in native 4D.

Private code consists of any group of 4D objects (such as variables, lists, forms, methods, pointers, tables, processes, semaphores, etc.) that are isolated within the module and appear nowhere outside of the module. The module's internal actions have complete control of its private objects and they, in turn, should have no effect on anything outside of the module.

In practice it is not possible to have anything that is completely private in its scope since all the code is executing within the same system environment. No matter how effectively one isolates a method's private code, for example, if the method aborts abnormally it will probably effect other operations.

In practice "private" is understood to mean programming elements that are not normally shared and whose scope is limited. Private code usually consists of global variables given reserved names, or tables that are internal to the operation of the module.

Shared Code

Shared code consists of code and data used by the module internally and by the programmer or user who interacts with the module. Shared code provides the interface to the internal functions of the module. Shared code provides the recognized elements used for passing information into and out of a module.

Any data the module acts upon that is assigned outside a module, or assigned by the module for later use, is shared code in a general sense. Any method contained in the module that is called from outside the module, or code written outside the module to be called from within the module, is also shared code. Shared code typically consists of specially named global variables, tables, methods, and files on disk.

Programming standards apply primarily to shared code. The objective of applying standards is to make the area of the code regular in its structure and syntax, predictable in its behavior, and reliable in its operation.

Public Code

Public, like shared code, consists of code and data used by the module and by those interacting with the module. Unlike shared code, however, public code is not used for communication between the module and the outside world. If necessary information must be passed across the modules boundary then it is done through shared code, not public code.

Public code generally consists of temporary data such as global variables used in printing or for the calculation and storage of intermediate values. It may be possible for some use of tables and methods to satisfy the definition of "public code", thought I cannot think of any examples. A familiar example of public code is the space of generic global variables used for data entry, import/export and report printing.

Modularity Guidelines

1.0 Private Code

1.1 Communication

1.1.1 All variables, tables, files or other structures used internally and that are not to be used outside the module are identified as private.

1.1.2 No shared information is passed through private structures.

1.2 Compatibility with Other Code

1.2.1 All variables, tables, files or other structures used as private code are created especially for this purpose.

1.2.2 If private code is visible or accessible outside the module, then it is identified to enable the user to avoid using, modifying or referring to it.

1.2.3 Private code memory should be released after it is no longer needed.

This means that any objects which used memory dynamically should be cleared when they are no longer needed. This includes:

- arrays
- sets

- named selections
- blobs
- pictures
- large fields loaded in memory (text or picture)

1.3 Contention

1.3.1 Private code should not subject to contention due to any operation outside of the module.

1.4 Encapsulation

1.4.1 Private code is isolated from shared and public code. Private code should remain private under all conditions.

1.5 Handling Errors and Exceptions

1.5.1 Errors that occur in private code should be reported as such.

1.6 Memory Management

1.6.1 Private code should guard against memory related errors.

1.6.2 Memory used in the execution of private code should be released when exiting the module.

1.7 Module Access

1.7.1 There should be no direct outside access to private code. While private code may execute within the module to generate intermediate results, these intermediate results should not be visible or accessible from outside the module.

2.0 Shared Code

2.1 Completeness

2.1.1 The module must initialize itself or, if initialization is required before entering the module, then initialization conditions should be tested upon entering the module.

2.1.2 Document all outside initialization that is required before entering the module.

2.1.3 The module should have an exit method that performs clean-up operations. This includes such actions as releasing of unused memory, closing windows, killing processes, enabling menus, etc.

2.2 Communication

2.2.1 Error conditions should be monitored and when an error occurs some indication of the error should be returned by the module.

2.2.2 There should be some reserved code or area for the reporting of errors. This could a return parameter, a global variable, a flag, or some other shared code.

2.2.3 All errors that occur within the module should be reported to the code that called the module.

2.2.4 All error messages are either displayed to the user from within the module, or placed in a predefined location for display to the user once outside the module.

2.2.5 If the module relies on interprocess message passing, then there should be a consistent message passing method. Messages can be passed using interprocess variables, message handling routines, or passed parameters, semaphores, and other means.

2.2.6 A consistent parameter passing syntax is used for calls into the module = and for the return of information from the module.

2.2.7 A consistent method of concatenating information is used. For example, if the forward slash (“/”) is used as a delimiter, then it is used consistently. Exceptions should be documented and kept to a minimum.

2.3 Compatibility with Other Code

2.3.1 If a 4D transaction has been started before entering the module, then the module will not pause for interaction with the user as long as the 4D transaction is running.

Nonfatal error messages are passed back to the calling methods for display outside of the 4D transaction.

2.3.2 The module should check for any currently executing 4D transactions before starting or stopping any transactions.

2.3.3 The module should test for and document and required preconditions.

2.3.4 The module should avoid making changes to shared data that is under the control of code that is outside the module.

For example, a Customer module should not make changes to Vendor tables, rather the Customer module should call the Vendor to make such changes.

2.4 Consistency

2.4.1 Side effects on code outside of the module should be minimized.

2.4.2 Predicable side effects should be documented.

For example, if the module changes the current selection and unloads records from certain tables, then this should be clearly noted.

2.4.3 The syntax of calls to module methods should be consistent.

2.4.4 The structure of module tables should be regular and predictable as possible.

For example, if ID fields are longints, then make them longints everywhere. If field names are abbreviated, then abbreviate them the same name everywhere.

2.4.5 The structure of module forms and their supporting code should be consistent.

For example, if form methods or object methods call project methods to handle execution, then project methods should be used in all form and object methods.

2.4.6 The level of understanding that the module assumes of the user should be consistent.

If you are assuming a beginners level of familiarity with coding, then continue to use this assumption in all areas of shared code. In this case you should either avoid complex strategies, or provide sufficiently detailed documentation.

2.5 Contention

2.5.1 Attempt to secure access to all needed data before beginning related modifications. This is generally done using some sort of flag or semaphore method to ensure that only one user is performing a particular process before letting another user begin the same process.

2.5.2 If contention leads to an abnormal condition (user or system abort) then it should be reported back to that method that called the module.

2.5.3 Time-out or user-abort exits should be placed in all areas of processing bottleneck areas. These are areas where “deadly embrace” or other perpetual wait conditions can develop.

2.6 Encapsulation

2.6.1 Shared code is segregated from other types of code. Code elements identified as shared must remain subject to the conditions of shared code under all conditions.

2.7 Handling Errors and Exceptions

2.7.1 The type of error encountered should be reported to the user of calling methods. Types of errors can include:

•shared code error

•private code error

•incorrect default setting

•incorrect parameters

•incorrect data

•user abort or termination

•programming error

•system condition (preventative flag set, memory allocation failure, etc.)

2.7.2 Errors should ideally report the problems

•nature

•origin

•location

•consequences

•remedial action

Not all of this information is available or appropriate, but whatever is available and useful should be provided to the user.

2.8 Memory Management

2.8.1 Shared code should test for memory related errors.

2.8.2 Shared data should only be modified where necessary.

That is, shared data should not be erased or released unless necessary. This also applies to the modules exit processing and “garbage collection” functions.

2.9 Module Access

2.9.1 Access to shared code should be documented.

2.9.2 The module should employ a consistent message passing syntax.

2.9.3 The module should test all input parameters for type and content.

2.9.4 The module should test all required values (variables, data, selections, etc.).

2.9.5 Document all required settings. Describe in detail all data that needs to be supplied or preset for the module to operate correctly.

2.10 Race Conditions

2.10.1 Control access to all bottlenecks using reserved variables, semaphores, or other means.

2.10.2 Use semaphores or other means to ensure that shared values are not overwritten while they are in use.

2.11 Reentrance

Reentrant code is code that calls itself. For example a method that performs many possible functions might be used as a handler method that calls itself to perform various related actions on the same set of data.

Recursion is a special kind of reentrance in which a given piece of code repeatedly calls itself to perform the same action on different data.

2.11.1 Prevent infinite recursion by limiting the recursive calls to a method to that method itself. That is, avoiding having related functions that call each other recursively.

2.11.2 Test for infinite recursion. Use such testing methods as monitoring:

- time spent in recursion
- number of iterations within the recursion
- occurrence of identical input and output over each recursion cycle

2.11.3 Provide a means of automatically or manually escaping from infinite recursive situations.

2.12 Segmentation

2.12.1 When closely related actions performed they should be performed from closely related areas.

Here are some examples:

- The creation, modification, and deletion of records in a table call all be managed by a single controlling method.

•The creation of different varieties or versions of records in a given table could be performed by passing different indications to a single controlling method.

2.12.2 Document the locations which modify related data.

For example:

•The control of tables (addition, modification and deletion of record) could be centralized.

•If different rules are applied under different circumstances, then all the locations that assign or affirm values according to such rules should cross reference each other.

2.12.3 A given action performed upon a table is only performed from one location.

For example:

•Create, initialize and save a record in a table in only one location. Do not duplicate similar code and scatter it in various locations.

•Use triggers or trigger-like code to centralize rule enforcement.

•Write controlling methods to centralize related actions are performed on related information.

3.0 Public Code

3.1 Communication

3.1.1 No information should be passed in or out of the module through public code.

3.2 Compatibility with Other Code

3.2.1 All public code used or called from outside the module (variables or methods) should not have any affect on code inside the module.

3.2.2 All public code used inside the module should not affect anything outside the module.

3.2.3 Public code memory should be released after it is no longer needed.

3.3 Consistency

3.3.1 Public code should be clearly identified.