The Elate® Object Based Programming Guide

Version 1.20

1. Introduction

Elate® allows applications to be designed and written in an object-based style. Object Based programming within Elate utilises the philosophy of small tools held in store, dynamically loaded only when required. (See the manual '*VP Programming with Elate.*')

A number of design issues need to be addressed before coding can begin. The programmer needs to consider:

a) The definition of the instance data structure for the object and where it is to be defined, how it is to be allocated and initialised and how it is to be de-initialised and de-allocated when no longer required. In traditional OOP terms this would be comparable to declaring class member variables (aka class properties). The initialisation and de-initialisation code would be comparable to code typically found in constructor and destructor methods of traditional OOP classes.

b) Whether all the method code should be written within the class or whether it would be more beneficial to dynamically load a tool from store when the method is actually called, thereby minimising the amount of memory required when first loading the class into memory. The tool is called using the VIRTUAL or VIRTUAL+FIXUP options to achieve this.

It is normally the responsibility of the application using the object to allocate and initialise the instance in line with the policy defined by the class programmer. If the object has already been allocated and initialised then the application is not responsible for this. E.g. a device driver object may have been allocated and initialised when the system was first booted.

2. Memory Structure

The definition of the instance data structure for the class instance can be defined either within the class itself or as an include file. If the structure is defined within the class then it will be difficult to inherit from that class. It is therefore recommended that the structure be defined in an include file which is then referenced at the beginning of the class source file. A baseclass structure's first offset must always be defined as **OB_SIZE**.

A subclass structure must begin at the size of the baseclass from which it is inheriting.

It is also necessary to check whether an include file has already been included elsewhere. This is to avoid redefinition errors if multiple source files are included in the same structures.

In the examples below, the class properties include parameters to hold the location and size of a buffer area.

Example Code of baseclass include file:

```
.if ~?def(BASECLASS_SIZE); if not defined then define BASECLASS_SIZE
      structure OB_SIZE ; must always use OB_SIZE for base class
      int32 PROPERTY1   ; properties of class
      int32 PROPERTY2
      int32 PROPERTY3
      size BASECLASS_SIZE     ; size of this base class structure
.endif
```

Example Code of subclass include file:

```
.if ~?def(SUBCLASS_SIZE)
; include properties of baseclass
.include 'demo/example/baseclass/class'
      structure BASECLASS_SIZE      ; offset is baseclass structure size
      int32  SUBCLASS_PROPERTY1
      size SUBCLASS_SIZE            ; size of subclass structure
.endif
```

3. Allocation and de-allocation of Memory

The allocation and de-allocation of an instance of a class is the responsibility of the application. Objects are usually allocated memory via calls to an appropriate library function, e.g. *sys/kn/mem/allocdata*. Typically, the programmer writes an allocator tool called *_new*, whose main task is to allocate memory for the object. There will also be a corresponding de-allocator tool, called *_delete*, which frees up memory allocated to the object. These special tools are normally coded in the same source file as the class methods.

In summary, the programmer will code the class properties in a *class.inc* file. The class methods will be coded in a file called *class.asm*, which will also include the special allocator and de-allocator tools, *_new* and *_delete* respectively. Note that because of this naming convention each class will need its own sub-directory.

4. Defining a Class and Method Coding

Within Elate, a class is a distinct type of tool with particular macros provided to build the contents. The name of the class is defined immediately after the first macro *class*. The name is the full pathname of the class tool. By convention, the name of the class tool is '*class*' within the same directory as the *'_new'* and *'_delete'* tools.

The *classend* macro defines the end of the class. All the methods for the object's services including the initialisation, de-initialisation methods and defaultmethod are coded within these two macros.

method is a macro, which defines a named service of the class. The macro is immediately followed by a subroutine containing code for the service. This may be all the code required or it may be a call to an external tool. If the method code is held as a tool, it can be coded in the same source file as the class, although it will be held separately on disk once assembled.

Note that a subroutine does not have to end with a *ret*; it has to end with something which does not permit execution to fall through it, i.e. one of *ret, go, endloop* (for a loop which does not contain a *break[if]* in it), *parentclass, chain*, or *chainclass*. Also note that *noret* is allowed; this is an indication that the call just before it is guaranteed not to return (e.g. when it throws an error and does not return). For more information on these terms, please see the '*VP Reference Manual.'*

Example baseclass method code framework:

```
; demo/example/baseclass/class.asm
.include 'tao'

.include 'demo/example/baseclass/class'

class 'demo/example/baseclass/class',VP

      method _init
            ent p0:i0
            tracef "baseclass : _init\n"
            clr [p0+BASECLASS_PROPERTY1]
            clr [p0+BASECLASS_PROPERTY2]
            clr [p0+BASECLASS_PROPERTY3]
            clr i0            ; return 0 if OK
            ret

      method _deinit
            ent p0:i0
            tracef "baseclass : _deinit\n"
            cpy 1,i0    ; drivers use <0 error, 0= in use, 1 as OK
            ret

      defaultmethod
            entd
                  tracef "baseclass : default method called.\n"
            ret
classend
```

Two methods, *_init* and *_deinit,* must be defined in the class code as well as a *defaultmethod*. The *_init* method initialises the object instance and the *_deinit* method performs the opposite of *_init*. All other methods are private to the particular class.

The *defaultmethod* defines the code to be executed if a method called is not provided. The method name called by the application must match exactly the method name defined within the class, otherwise the *defaultmethod* will be called instead. The *defaultmethod* can code for an error.

```
defaultmethod
      entd
```

```
      ; method name not recognised handler code
      ret
```

If the default method of a subclass is called, control is transferred to the baseclass default method. Note that the baseclass is also known as the parent class.

```
defaultmethod
      entd
      parentclass       ; call baseclass default method
```

Note that *defaultmethod* is not always necessary when its parent class is the only requirement. If it is included, a *ret* will not be required, as above. When defaultmethod is not included, the system will provide one. If this happens to be a subclass, the system version will provide a parentclass. If it is a baseclass, the system will execute a *noret*, will produce undefined behaviour, except when running with pentiumt, when it will produce a breakpoint trap.

For more information upon this, please see the documentation on *entd* provided within the Elate build.

Another method type that may be defined within the class file is xmethod, e.g.

```
      :
xmethod char
      ent p0 p1 i0 i1 i2 i3 i4 i5 i6:i0 i1
qcall lib/grf/fnt/chardraw,(p0,p1,i0,i1,i2,i3,i4,i5,i6:i0,i1),VIRTUAL|FIXUP
            ret
      :
```

This allows the programmer to make a call of the type :

```
            ncall p0,@char,(p0:p3)
```

A call of this type (with @) returns a pointer to the method code, in this case in p3. The method code can thus be invoked subsequently without the overhead of a normal *ncall* by using a *gos* instruction, e.g.

```
            gos p3,(p0,p1,i0,i1,i2,i3,i4,i5,i6:i5,i9)
```

It is also still possible to use a normal *ncall* to invoke the method code.

5. Allocation and De-allocation of Memory

For the methods of an object to be accessed by an application, an instance of the class of the object must be in memory. An allocation and de-allocation policy must be decided upon by the application programmer but conventionally the class programmer will provide a tool to allocate memory, called *_new,* and a tool to deallocate, called *_delete*.

There are several Elate functions available for memory allocation, but for this example we will use *sys/kn/mem/allocdata*. For memory allocation in device drivers it is preferable to use *sys/kn/mem/allocdef*, as this allocates memory from a pool that can be shared by multiple processes and can also remain once the allocating process has closed. The ANSI library functions are not usually used in this situation as they may not be available to device drivers.

5.1 Allocating memory using the *_new* tool

Memory allocation is carried out by a tool because an instance of the object has not yet been created. However, the source of the *_new* tool can be coded within the same source file as the class but is held separately once assembled, and which is stored in the same directory as that of the object.

The *_new* tool allocates the memory, and it must also reference the class and return the instance pointer in *p0*. To allocate a memory block of the correct size, we need to allocate the size of the instance data structure as defined in the instance include file.

```
cpy BASECLASS_SIZE,i0
qcall sys/kn/mem/allocdata,(i0 : p0,i0)
```

To initialise the object header and reference the class, the macro *refclass* is used with the full name of the class tool. *refclass* searches for the class tool, bringing it into memory from store, and translating it, only if it is not already present on the tool list.

```
refclass p0,demo/example/baseclass/class
```

Example code:

```
; allocator tool in demo/example/baseclass/class.asm

tool 'demo/example/baseclass/_new'
      ent - : p0        ; return instance pointer
      cpy.i BASECLASS_SIZE,i0
      qcall sys/kn/mem/allocdata,(i0:p0,i0)
      if.p p0 != NULL
            refclass p0,demo/example/baseclass/class
      endif
      ret
toolend
```

5.2 De-allocating memory using the *_delete* tool

The *_delete* tool reverses the *_new* tool provided by the class programmer. The *_delete* tool can be coded within the same source file as the class but it is held separately once assembled. Note it is stored in the same directory as that of the object.

The *_delete* tool de-references the class, using the macro *derefclass,* and then frees the memory.

Example code:

```
; de-allocator tool in class.asm
tool 'demo/example/baseclass/_delete'
      ent p0 : -
      derefclass p0
      qcall sys/kn/mem/free,(p0 : -)
      ret
toolend
```

6. Initialising and de-initialising the instance data structure

Once an object has been allocated memory and referenced, the application has access to the methods within the class. The instance data structure should be initialised by calling the *_init* method, before any of the other methods will be available for use. Otherwise, member variables in the instance data structure will not be in an initialised state.

The coding of the *_init* method of a class will depend on the type of object that is to be initialised. It may take the form of private memory allocation or setting up values within all or part of the instance data structure.

If initialisation should fail, a tidy up routine must be carried out by the application to delete the object.

If the instance being initialised is a subclass, the baseclass section of the instance data structure must be initialised first. This is effected by calling the baseclass *_init* method. If this is successful, the subclass section can then be initialised.

If the subclass *_init* method was unsuccessful, but the baseclass *_init* was successful, the baseclass *_deinit* method must be called before returning failure to the application.

Once initialisation has been successful, the instance of the class i.e. the object, is available for use by the application.

Once an object is no longer required, the instance must be de-initialised before it can be deleted. The *_deinit* method must be a complete reversal of the *_init* method and is again private to the object. Once de-initialisation has been successful, the instance may then be deleted.

7. Defining a Baseclass

The definition of a class, be it base or sub, requires that the correct include files are defined at the beginning of the class source file. These are normally the standard Elate include file, *'tao,'* and any private include files specifying the structure of the instance data.

If the instance data structure is not to be defined within an include file it can be defined within the class source file, before the *class* macro. However, it should be remembered that if the instance data structure is defined inside the class source file, it will not be possible to inherit from the class easily.

Once the include files and instance data structure have been defined, these must be immediately followed by the *class* macro as described earlier.

Once the include files and instance data structure have been defined, these must be immediately followed by the *class* macro as described earlier.

The two compulsory methods, *_init* and *_deinit,* are defined next. Like all methods, the *_init* and *_deinit* methods can make external calls to tools, for greater memory efficiency.

```
      method _init
            ent p0:i0
            tracef "baseclass : _init\n"
            clr [p0+BASECLASS_PROPERTY1]
            clr [p0+BASECLASS_PROPERTY2]
            clr [p0+BASECLASS_PROPERTY3]
            clr i0            ; return 0 if OK
            ret
```

The *defaultmethod* is coded after all the other methods. The *defaultmethod* must come after all other methods in order for the class to be created correctly. The code within the *defaultmethod* is specific to the class, as are all other methods, but it will normally be coded to log an error or throw an exception when in a baseclass. Sometimes the baseclass method will be written simply to return and perform no other action.

To close the definition of the class tool, the *classend* macro is used.

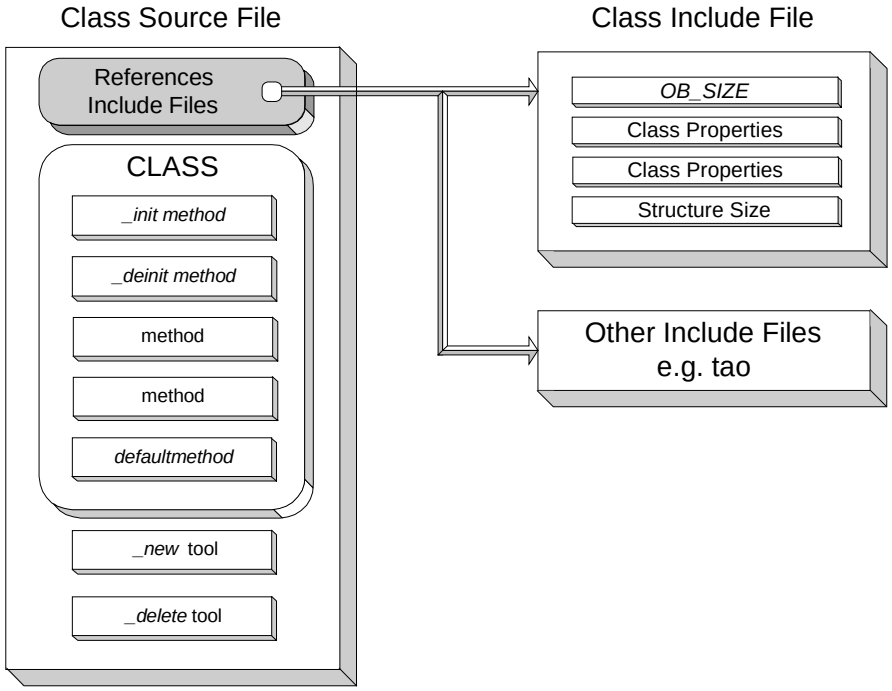

Figure 1: The Baseclass

Example code for baseclass :

```
; class.asm       --    baseclass

.include 'tao'

.include 'demo/example/baseclass/class'

class 'demo/example/baseclass/class',VP

      method _init
            ent p0:i0
            tracef "baseclass : _init\n"
            clr [p0+BASECLASS_PROPERTY1]
            clr [p0+BASECLASS_PROPERTY2]
            clr [p0+BASECLASS_PROPERTY3]
            clr i0            ; return 0 if OK
            ret

      method _deinit
            ent p0:i0
            tracef "baseclass : _deinit\n"
            cpy 1,i0    ; drivers use <0 error, 0= in use, 1 as OK
            ret

      method getval
            ent p0:i0
            tracef "baseclass : getVal\n"
            cpy [p0+BASECLASS_PROPERTY1],i0
            ret

      method setval
            ent p0 i0 : -
            tracef "baseclass : setVal\n"
            cpy i0,[p0+BASECLASS_PROPERTY1]
            ret

      defaultmethod
            entd
                  tracef "baseclass : default method called.\n"
            ret
classend


tool 'demo/example/baseclass/_new'
      ent - : p0                        ; return instance pointer
      tracef "baseclass : _new\n"
      cpy.i BASECLASS_SIZE,i0
      qcall sys/kn/mem/allocdata,(i0:p0,i0)
      if.p p0 != NULL
            tracef "referencing base class\n"
            refclass p0,demo/example/baseclass/class
      endif
      ret
toolend


tool 'demo/example/baseclass/_delete'
      ent p0 : -
      tracef "baseclass : _delete\n"
      derefclass p0
      qcall sys/kn/mem/free,(p0 : -)
      ret
toolend
.end
```

8. Defining a Subclass

A subclass is defined in a similar manner to a baseclass with some alterations (see Figure 2).

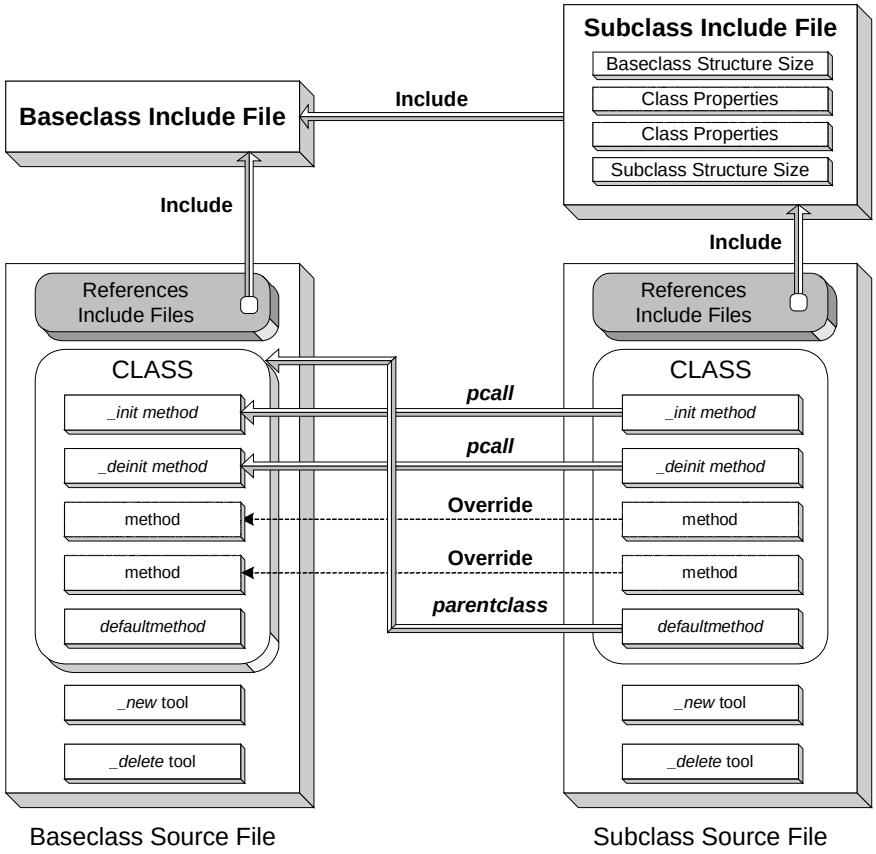

Figure 2: The Subclass

The subclass name is defined after the *class* macro, with the name of the baseclass added before the language definition. Note there are no quotes around the baseclass name.

```
class 'demo/example/subclass/class',demo/example/baseclass/class,VP
```

The *_init* method is still required, but the subclass *_init* method must now include a call to the baseclass *_init* method to initialise the baseclass section of the instance data structure so that its methods are also available for use. This is achieved by using *pcall* (parent call) within the subclass *_init* method. Note that a *pcall* can only be made from within a class tool, unless a large method's code is moved into a separate tool, in which case the programmer may use the *__extends* macro at the start of that tool to enable you the usage of the *pcall* macro in that tool.

```
pcall p0,_init,(p0:i0)
```

The *_deinit* method of the subclass must also call the baseclass *_deinit* method by using *pcall* after de-initialising the subclass section of the instance data structure.

```
pcall p0,_deinit,(p0:i0)
```

The *defaultmethod* must pass responsibility to the baseclass as a method called and not found may be within the baseclass. The command for this is *parentclass*.

```
defaultmethod
entd
      ; additional code here if required
      parentclass
ret
```

All other methods are private to the subclass. Any methods defined in the subclass, with the same name as methods defined in the baseclass, will override such baseclass methods. To close the definition of the subclass code, the *classend* macro is used as before.

Example code for a subclass:

```
; class.asm --   subclass

.include 'tao'
.include 'demo/example/baseclass/subclass/class'

class
'demo/example/baseclass/subclass/class',demo/example/baseclass/class,VP

      method _init
            ent p0:i0
            tracef "subclass : _init\n"
            pcall p0,_init,(p0:i0)        ; init baseclass
            ifnoterrno i0
                  ; baseclass init was OK so init subclass instance data
                  cpy 456,[p0+SUBCLASS_PROPERTY1]
            endif
            ret

      method _deinit
            ent p0:i0
            tracef "subclass : _deinit\n"
            pcall p0,_deinit,(p0:i0)      ; base class deinit
            ret

      method scmethod
            ent p0 i0:-
                  tracef "subclass : subclassmethod.\n"
                  cpy i0,[p0+SUBCLASS_PROPERTY1]
                  tracef "[p0+SUBCLASS_PROPERTY1] =
%d\n",[p0+SUBCLASS_PROPERTY1]
            ret

      defaultmethod
            entd
                  parentclass
            ret
classend


tool 'demo/example/baseclass/subclass/_new'
      ent - : p0                          ; return instance pointer
      cpy.i SUBCLASS_SIZE,i0
      qcall sys/kn/mem/allocdata,(i0:p0,i0)
      if.p p0 != NULL
            refclass p0,demo/example/baseclass/subclass/class
      endif
      ret
toolend


tool 'demo/example/baseclass/subclass/_delete'
      ent p0 : -
      derefclass p0
      qcall sys/kn/mem/free,(p0:-)
      ret
toolend
.end
```

9. Calling an object from within an application

For an application to be able to call a method of an object, it is necessary for an instance of the class to be in memory. This can be achieved by the application calling the *_new* tool of the class required.

```
qcall demo/example/baseclass/_new,(-:p0)  ; object reference returned
```

In some circumstances, memory for the object will be allocated from within the application, in which case the application must reference the class by using the *refclass* macro and create the instance, in the same manner as the *_new* tool.

If the object was successfully created p0 will be the instance pointer, and the object can then be initialised.

To call a method we use the Elate named call, *ncall.*

```
ncall p0,_init,(p0:i0)
```

If the initialisation was successful, all of the methods of the object will be available to be called. The application has no knowledge of whether the object instance is a subclass or a baseclass and has no interest in how the services are provided by the object. All the methods can be called by using *ncall* and the specific method name. If the method name is not called correctly or is not available, the *defaultmethod* will be invoked.

Once the object is no longer required, it is the job of the application to de-initialise and delete the instance in line with the policy defined, calling the *_deinit* method, and then either calling the *_delete* tool or de-referencing the class and freeing the instance memory.

Example code to test baseclass :

```
; test.asm

.include 'tao'

tool 'demo/example/baseclass/test',VP,TF_MAIN,8192,0

      ent - : -

    cpy.p 0,p1

      ; new instance
      qcall demo/example/baseclass/_new,(-:p0)  ; p0 object ref
    if p0==0
        tracef "out of memory creating first baseclass object\n"
        go tidyup
    endif
    ncall p0,_init,(p0:i0)
    iferrno i0
        tracef "error initialising first baseclass object\n"
      qcall demo/example/baseclass/_delete,(p0:-)
        cpy.p 0,p0
        go tidyup
    endif

      ; another new instance
      qcall demo/example/baseclass/_new,(-:p1)  ; p1 object ref
    if p1==0
        tracef "out of memory creating second baseclass object\n"
        go tidyup
    endif
    ncall p1,_init,(p1:i0)
    iferrno i0
        tracef "error initialising second baseclass object\n"
      qcall demo/example/baseclass/_delete,(p1:-)
        cpy.p 0,p1
        go tidyup
    endif

    cpy.i 4,i0
    ncall p0,setval,(p0 i0:-)

    cpy.i 123,i0
    ncall p1,setval,(p1 i0:-)

    clr i1
    ncall p0,getval,(p0 : i1)
    tracef "getval returned : %d \n",i1

    clr i1
    ncall p1,getval,(p1 : i1)
    tracef "getval returned : %d \n",i1

    ncall p0,fred,(-:i0)

tidyup:
    if p0!=0
        ; Deinitialise and destroy first object
        ncall p0,_deinit,(p0:i0)
      qcall demo/example/baseclass/_delete,(p0:-)
    endif
    if p1!=0
        ; Deinitialise and destroy first object
```

```
        ncall p1,_deinit,(p1:i0)
      qcall demo/example/baseclass/_delete,(p1:-)
    endif

      ; shutdown
      qcall lib/exit,(0:-)
      ret
toolend
.end
```

Example code to test subclass :

```
; test.asm

.include 'tao'

tool 'demo/example/baseclass/subclass/test',VP,TF_MAIN,8192,0

      ent - : -

      ; new instance
      qcall demo/example/baseclass/subclass/_new,(-:p0)     ; p0 object ref
    if p0==0
        tracef "out of memory creating subclass object\n"
        go tidyup
    endif
      ncall p0,_init,(p0:i0)
    iferrno i0
        tracef "error initialising subclass object\n"
        qcall demo/example/baseclass/subclass/_delete,(p0:-)
        cpy.p 0,p0
        go tidyup
    endif

      cpy 4,i0
      ncall p0,setval,(p0,i0:-)

      clr i1
      ncall p0,getval,(p0:i1)
      tracef "returned by method : %d \n",i1

      ; this method not defined
      ncall p0,fred,(-:-)

      ; defined subclass method
      ncall p0,scmethod,(p0,i0:-)


tidyup:
    if p0!=0
          ncall p0,_deinit,(p0:i0)
      qcall demo/example/baseclass/subclass/_delete,(p0:-)
    endif

      ; shutdown
      qcall lib/exit,(0:-)
      ret
toolend
.end
```

10. A note on the Example Code

Generally in the Elate system the class hierarchy is reflected in the directory hierarchy. Subclasses should be located in directories below the parent class. For the example used in this manual the directory structures should be:

demo/example/baseclass/class.asm

demo/example/baseclass/subclass/class.asm

If this structure was used a call to create a subclass object would be:

qcall demo/example/baseclass/subclass/_new,(-:p0)

In other words, the directory structure should be reflected throughout the class definitions. This approach was not used in this manual to help keep the names more manageable, but this may change for future versions of this document.

For the very latest version of the code referred to in this manual please see *demo/example/*.*

11. Glossary of Terms

Object based programming
Provides encapsulation (data hiding, bundling together data and access procedures), inheritance (the ability to include previously defined attributes in your new object) and polymorphism (the ability to run code with the same name and interface on different data types).

Data hiding
The internal functionality within an object is often 'hidden' from the application programmer. The programmer normally manipulates the object through a set API of methods. These methods may manipulate the internal data (properties) of the object. It is safer to manipulate this data through the API as the methods can perform error checking on the request to manipulate properties.

Object
Objects provide services and are classified according to the services they provide. An application requesting services has no knowledge of the way in which the object implements those services. An object is an instantiation of a class. Each object is manipulated via the methods and properties of that class of object.

Class
A Class defines the functionality of objects of that class and the services that such objects provide. A class can be thought of as an object framework. It is possible to instantiate multiple objects of the same class, each object having unique properties.

Instance
An Object is an Instance of a Class. The Instance is the data structure that holds the private data of the object. There can be more than one instance of a class in memory at one time with each instance being unique.

Method
A Method defines a service that an object can perform. Methods can be inherited or overridden by subclasses. This list of methods of a class defines the API for that class.

Defaultmethod
If the named method called by an application is not present in a class, the defaultmethod will automatically be executed instead. The defaultmethod can code for an error or if coded in a subclass, it can pass responsibility for handling the method to the parent class.

Inheritance
Every object is or has a baseclass (parent class). A similar object may reference within its class code, a baseclass thereby inheriting its methods (services), which have already been coded. In this way, subclasses have access to the methods of the baseclass without being aware of their implementation. Each subclass is only aware of its own parent which may itself be a subclass of another class.

Ncall

An *ncall* is a named method call on an object. It is the means by which an object's method code is invoked.

Classname
Normally the class methods are coded in a file *class.asm*. Each class will therefore have it's own subdirectory e.g. */path/myclass/class.asm*. In addition to the class.asm file there will be a class.inc file, which defines the data structure (properties) for that class, e.g. */path/myclass/class.inc*

12. Example code

The following simple example code demonstrates the principles that have been discussed in this manual.

In this example, four classes are used: *node*, *job*, *list* and *queue*. There is also a simple test program.

Code for *node* class include file:

```
; demo/example/node/class.inc
;
.if ~?def(NODE_SZ)
      structure OB_SIZE
            int32 ITEM
            pointer NEXT
            pointer PREV
      size NODE_SZ
.endif
```

Code for *node* class file :

```
; demo/example/node/class.asm

.include 'tao'
.include 'demo/example/node/class'

class 'demo/example/node/class',VP

      method _init
      ent p0 i0: -
      cpy i0,[p0+ITEM]              ; init node
      cpy.p NULL,[p0+NEXT]
      cpy.p NULL,[p0+PREV]
      ret

      method _deinit
      ent p0 : -
      cpy 0,[p0+ITEM]                     ; deinit node
      cpy.p NULL,[p0+NEXT]
      cpy.p NULL,[p0+PREV]
      ret


      method get_node_item
      ent p0 : i0
      cpy [p0+ITEM],i0
      ret

      method get_node_next
      ent p0 : p0
      cpy.p [p0+NEXT],p0
      ret

      method set_node_next
      ent p0 p1: -
      cpy.p p1,[p0+NEXT]
      ret

      method set_node_item
      ent p0 i0 : -
      cpy i0,[p0+ITEM]
      ret

      defaultmethod
```

```
      entd
            tracef “node class default method.\n”
      ret
classend


tool 'demo/example/node/_new'
      ent - : p0
;tracef "debug: creating new node ...\n"
      cpy NODE_SZ,i0
;tracef "debug: NODE_SZ = %d\n",i0
      qcall sys/kn/mem/allocdata,(i0 : p0 i~)
      refclass p0,demo/example/node/class
      ret
toolend

tool 'demo/example/node/_delete'
      ent p0 : -
      derefclass p0
      qcall sys/kn/mem/free,(p0 : -)
      ret
toolend
.end
```

Code for *job* class include :

```
; demo/example/job/class.inc
; inherits from node
.if ~?def(JOB_SZ)
.include 'demo/example/node/class'
      structure NODE_SZ
            int32 PRIORITY
      size JOB_SZ
.endif
```

Code for *job* class :

```
; demo/example/job/class.asm -- sub-class of node

.include 'tao'
.include 'demo/example/job/class'
.include 'demo/example/node/class'

class 'demo/example/job/class',demo/example/node/class,VP

      method _init
      ent p0 i0: i0
      pcall p0,_init,(p0 i0 : -)
      cpy 0,[p0+PRIORITY]
      ret

      method _deinit
      ent p0 : -
      pcall p0,_deinit,(p0 : -)
      cpy 0,[p0+PRIORITY]
      ret


      method get_job_pri
      ent p0 : i0
      cpy [p0+PRIORITY],i0
      ret


      method set_job_pri
      ent p0 i0 : -
```

```
      cpy i0,[p0+PRIORITY]
      ret

      defaultmethod
      entd
            parentclass
      ret
classend


tool 'demo/example/job/_new'
      ent - : p0
      cpy JOB_SZ,i0
      qcall sys/kn/mem/allocdata,(i0 : p0 i0)
      refclass p0,demo/example/job/class
      ret
toolend

tool 'demo/example/job/_delete'
      ent p0 : -
      derefclass p0
      qcall sys/kn/mem/free,(p0 : -)
      ret
toolend
.end
```

Code for *list* class include file :

```
; demo/example/list/class.inc

.if ~?def (LIST_SZ)
      ; list header
      structure OB_SIZE
            pointer HEAD
            pointer TAIL
            int32 COUNT
      size LIST_SZ

.endif
```

Code for *list* class :

```

; demo/example/list/class.asm

.include 'tao'
.include 'demo/example/list/class'

class 'demo/example/list/class',VP

; methods

      method _init
      ent p0 : -
      ; init list structure
      cpy.p NULL,[p0+HEAD]
      cpy.p NULL,[p0+TAIL]
      cpy.i 0,[p0+COUNT]
      ret

      method _deinit
      ent p0 : -
      ncall p0,zap_list,(p0 : -)
      ret
```

```
      method add_node
      ; adds node object to end of list
      ; p0 is list object
      ; p1 is node object to add
      ent p0 p1 : -
      if.p [p0+HEAD] = NULL
            cpy.p p1,[p0+HEAD]      ; update head
            cpy.p p1,[p0+TAIL]      ; tail and head point to same node
      endif
      cpy.p [p0+TAIL],p2                  ; p2 is tail
      ncall p2,set_node_next,(p2 p1 : -)
      cpy.p p1,[p0+TAIL]
      cpy.p NULL,p2
      ncall p1,set_node_next,(p1 p2 : -)
      inc [p0+COUNT]
      ret


      method print_list
      ; p0 is list object to print
      ent p0 : -
      clr i0
      clr i1
      cpy.p [p0+HEAD],p1
      tracef "list contains\n"
      while.p p1 != NULL
            ncall p1,get_node_item,(p1:i1)
            tracef "item %d:%d  ",i0,i1
            ncall p1,get_node_next,(p1:p1)
            inc i0
      endwhile
tracef "\ndebug: count is %d items\n",[p0+COUNT]
tracef "debug: count is %d items\n",i0
      ret

      method zap_list
      ; p0 is list object to zap
      ent p0 : -
      clr i0
      clr i1
      cpy.p [p0+HEAD],p1
      tracef "zapping list...\n"
      while.p p1 != NULL
            ncall p1,get_node_next,(p1:p2)  ; save p1->next
            ncall p1,get_node_item,(p1:i1)
tracef "debug: deleting node containing %d\n",i1
            ncall p1,_deinit,(p1:-)
            ncall p1,_delete,(p1:-)
            cpy.p p2,p1
            inc i0
      endwhile
tracef "debug: %d nodes deleted.\n",i0
      ret

      defaultmethod
      entd
            tracef “List class default method.\n”
      ret
classend

tool 'demo/example/list/_new'
      ent - : p0
      cpy LIST_SZ,i0
      qcall sys/kn/mem/allocdata,(i0 : p0)
      refclass p0,demo/example/list/class
```

```
      ret
toolend

tool 'demo/example/list/_delete'
      ent p0 : -
      derefclass p0
      qcall sys/kn/mem/free,(p0 : -)
      ret
toolend
.end
```

Code for *queue* class include :

```
; demo/example/queue/class.inc
; inherits from list
;
.if ~?def (QUEUE_SZ)
.include 'demo/example/list/class'

      structure LIST_SZ
            ; same as list at present
      size QUEUE_SZ

.endif
```

Code for *queue* class :

```
; demo/example/queue/class.asm
; Code TPB 98

.include 'tao'
.include 'demo/example/queue/class'

class 'demo/example/queue/class',demo/example/list/class,VP

; methods

      method _init
      ent p0 : -
      pcall p0,_init,(p0 : -)
      ret

      method _deinit
      ent p0 : -
      pcall p0,zap_list,(p0 : -)
      ret

      method add_job
      ent p0 p1 : -
      ; p0 queue
      ; p1 job to add
      clr i0
      clr i1
      cpy.p [p0+TAIL],p4                  ; save tail
      ncall p1,get_job_pri,(p1 : i1)
      if.p [p0+HEAD]=NULL                 ; list empty add to head
            ncall p0,add_node,(p0 p1:-)
      else
            if.p p4 != NULL
                  ncall p4,get_job_pri,(p4:i0)
                  if i0 >= i1
                        ncall p0,add_node,(p0 p1:-)   ; add on end
                  else
                        cpy.p [p0+HEAD],p2
                        ncall p2,get_job_pri,(p2:i0)
                        if i0 < i1
                              ; add before current head
                              ncall p1,set_node_next,(p1 p2:-)
                              cpy.p p1,[p0+HEAD]
                              inc [p0+COUNT]
                        else
                              while.p p2 != NULL
                                    ncall p2,get_job_pri,(p2:i0)
                                    if i0 < i1
                                          ; insert job
                                          ncall p1,set_node_next,(p1 p2:-)
                                          ncall p3,set_node_next,(p3 p1:-)
                                          inc [p0+COUNT]
                                    endif
                                    cpy.p p2,p3       ; save
                                    ncall p2,get_node_next,(p2:p2)
                              endwhile
                        endif
                  endif
            endif
      endif
      ret

      method despatch_job
      ent p0 : -
```

```
      ; not coded
      ret

      defaultmethod
      entd
            parentclass
      ret
classend

tool 'demo/example/queue/_new'
      ent - : p0
      cpy QUEUE_SZ,i0
      qcall sys/kn/mem/allocdata,(i0 : p0)
      refclass p0,demo/example/queue/class
      ret
toolend


tool 'demo/example/queue/_delete'
      ent p0 : -
      derefclass p0
      qcall sys/kn/mem/free,(p0 : -)
      ret
toolend
.end
```