1 General

1.1 Introduction

The desktop follows a document centered approach in contrast to other task driven implementations. This means the user is working with documents to be found on the system rather then starting some kind of application and working inside of that. For example to modify a text document the user just clicks the document and doesn't has to care which application is able to handle that kind of document because the system will find an appropriate one. In task driven environments the user first starts an application, for example a word processor, and opens documents from inside the application using a file chose dialog.

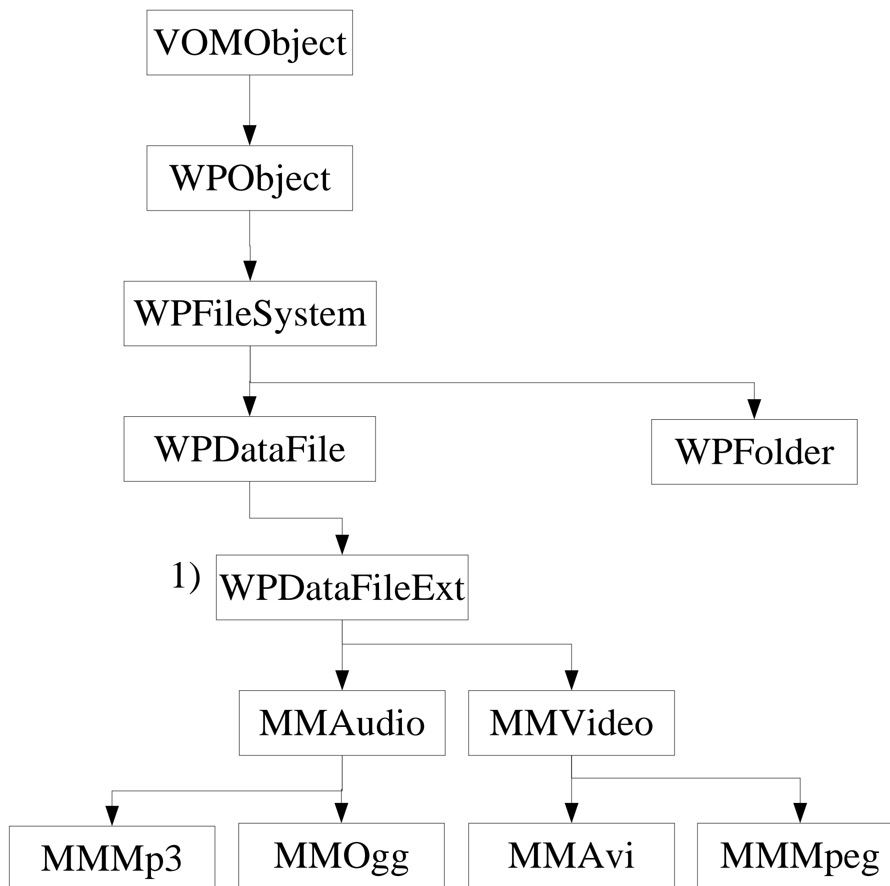
All documents, files etc. the user interacts with are objects because the desktop is based on an object model described elsewhere in this document. This means the desktop is fully object orientated with all the benefits coming from that. So it's possible to have data encapsulated in the objects and subclassing of document or file classes is not even allowed but encouraged by the underlying system.

1.2 Classes and objects

The user of the desktop is dealing with objects. Files for example are not classic files like on other systems but are real objects in the sense that they might carry additional information which is encapsulated and come with object methods to work with them. For example one doesn't copy a file by using a filemanager or the desktop application and calling some copy() API of the operating system. Instead a method on the file object is invoked and the file object handles the copying on itself.

Every object is an instance of a class which is registered with the desktop. Classes may be registered and deregistered at any time (a restart of the desktop may be necessary). They are implemented as shared libraries and link dynamically to the desktop. Basing the whole desktop on an object model which allows subclassing and class replacement without recompiling it's possible for independend developers to create new classes without access to the desktop source code. In fact developers are encouraged not to modify sources to extend the desktop but to create new classes to add features to the desktop. By replacing classes with new versions overriding inherited methods it's possible to modify the behaviour of any class of the desktop.

The root class of the whole desktop is VOMObject which is also the root class of the underlying object system (see picture below). WPObject implements basic desktop methods like context menus or property notebooks. Derived from the basic desktop class is the filesystem class for objects representing files or folders. A subtree handles all types of folders which isn't covered here. All kinds of files can be seen on the left part of the tree. A basic datafile class WPDataFile implements methods common to all files like querying the filesize or getting the date of creation. A subtree starting with MMAudio implements audio multimedia files while a class MMVideo handles digital video files. The WPDataFileExt class will be covered later.



1) The WPDataFileExt class replaces WPDataFile as the base class for MMAudio and MMVideo.

The benefits of using such a class tree in the desktop are easily spotted. The MMAudio class implements basic features like playing audio files or displaying audio information like the playtime of the file in question using operating system features. It may also provide a method for accessing tags in the audio file containing the artists name or the trackname. One may implement the latter by adding code to handle ID3 tags of MP3 files and code to handle the same for OGG files. In the end MMAudio would contain code for every tag format known. If a developer ever came up with a new audio codec and a different tag architecture the maintainer of the desktop or the MMAudio class had to be asked to add additional parsing code. Even after the code addition users can't immediately benefit from it because the next release may only be available at some point in the future.

Subclassing

Using derived classes said developer may just create a new class which inherits everything from MMAudio but overrides the method which deals with tag parsing to implement the new parser. Using a binary compatible object system this class can be added without recompiling the desktop. Even more the developer doesn't need the desktop sources at all and doesn't have to care about the rest of the class hierarchy or changes to MMAudio made by others (as long as the others don't break the class by changing the binary interface by purpose). Users may install the new class without changing anything of their current system and may immediately use the new features implemented by the class seamlessly.

So instead of adding more and more code to MMAudio it's better to move specialized code like tag parsing to specialized classes as done in the example with MMMp3 and MMOgg. This also helps to keep the source maintainable because possible execution path are clear and well contained.

Class replacement

To be written...

1.3 Folder views

Each folder object opens in it's own window (spatial view). The characteristics of this window like size, position, used font are saved in the instance data of the folder object. This means the window reopens exactly the way the user left it before.

Three different kind of views are supported:

- Icon view: All objects are shown with an icon and a name
- Details view: Objects are shown in a list with each row containing the icon, the name and details about the object in question like filesize or creation date.
- Tree view: A tree of the folder and the subfolder is shown. Single objects are not shown in this view. If any of the subfolders contains additional folders this is depicted by a plus sign next to it. Clicking on this plus shows the subfolder tree.

Browser view is not supported because this kind of traversing the filesystem breaks the object metaphor of the desktop. Changing the folder one is in has to also change position, size and shape of the browser window because in the object world the view should represent the underlying object. Not taking into account the object settings confuses the user because he can't rely on a consistant behaviour of folder views with respect to the folders window settings.

1.4 Templates

New documents, files or other objects are created by dragging an object template somwhere into the filesystem. Templates normally are located in the central Templates folder but may also be created in other places. Beside the system or application provided templates the user may create his own templates. This is done by checking the Templates checkbox in the settings of an object.

> Voyager-desktop compliant applications handling documents should create the appropriate templates for the user.

2 Implementation

This Chapter describes the implementation of the desktop. It's a technical document intended merely for developers.

2.1 Memory management

Using wpAllocMem() and wpFreeMem() allocated memory is tracked in an **inuse list**. Whenever an object is deleted or goes dormant this list is scanned by the system and memory still allocated will be freed automatically. Nevertheless memory not in use anymore should be freed by the programmer as usual because the desktop does not include a garbage collector. The inuse list should be considered as a kind of safety net.

> While the usual memory function like malloc(), g_alloc(), VOMMalloc() are available to desktop developers it is strongly adviced to only use wpAllocMem() and wpFreeMem().

a)Instance Variables

Any desktop object may or may not have instance variables. There's no overhead with using them and the count of them is only limited by system memory. Be aware that access to these variables is only possible using dedicated object methods. Don't assume any specific layout for this data.

Instance variable space is allocated during creation of an object and freed on object destruction. Instance variables are initialzed to zero.

See the documentation about the object model for further information.

b)Memory allocation with wpAllocMem()

A programmer asks for memory using

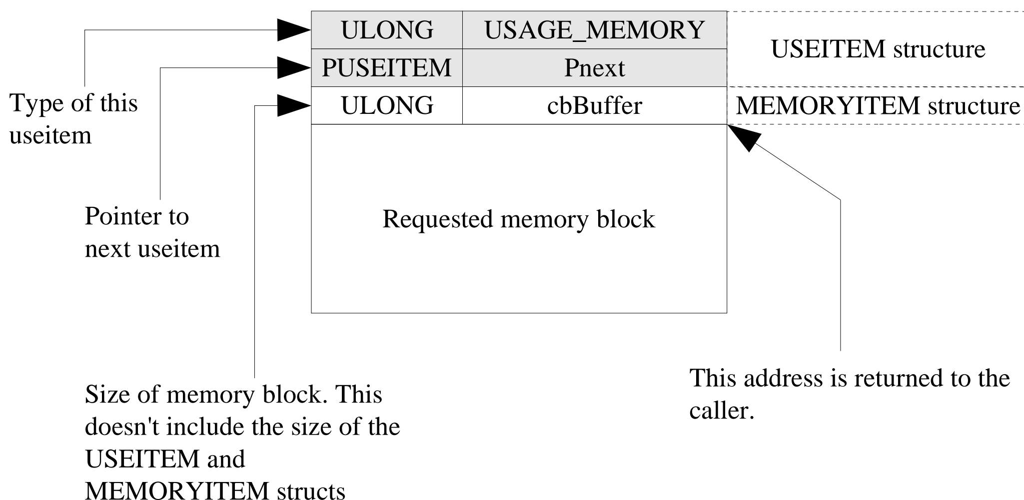
wpAllocMem(WPObject vSelf, ULONG cbBytes, ...)

This method is introduced by the WPObject class.

wpAllocMem() does the following:

- Call VOMCalloc() to allocate a memory block of size cbBytes+sizeof(USEITEM)+sizeof (MEMORYITEM)
- Fill information area of the USEITEM part of the memory block
- Fill information area of the MEMORYITEM part of the memory block
- Call wpAddToObjUseList(vomSelf, ...) to add the item to the objects inuse list
- Return a pointer to the memory block following USEITEM and MEMORYITEM to the caller

The layout of the memory is like this.



To ensure the desktop is thread safe each object has a semaphor to serialize access to critical object instance variables. The inuse list is such a critical resource so wpAddToObjUseList() tries to aquire

the objects semaphor using wpRequestObjectMutexSem().

Calling wpAllocMem() while holding the object semaphor will cause a deadlock.

c)Freeing memory with wpFreeMem()

Memory is freed using **wpFreeMem()**.

This method introduced by WPObject takes the memory pointer and travels back to the USEITEM part of the allocated memory block (see the section about memory allocation). Using a pointer to the USEITEM the memory is removed from the objects inuse list using wpDeleteFromObjUseList(). Be aware that this method tries to aquire the object semaphor so a deadlock may occur when the semaphor is already held.

Calling wpFreeMem() while holding the object semaphor will cause a deadlock.

After removing the item from the inuse list a call to VOMFree() deallocates the memory block.

3 Annex

3.1 Methods using the object semaphor

- wpAllocMem();
- wpFreeMem();