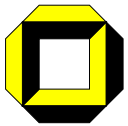

Universität Karlsruhe (TH)
Fakultät für Informatik
Institut für Mikrorechner und Automation
Prof. Dr.rer.nat U. Brinkschulte

Fluids, Poem

—

Application Programming Interface

Version 1.6 — June 2, 1997

Holger Vogelsang

```
public:

    WW( boolean created, const char *abc1, long d1, long d2 ){
        if( created ) {
            strcpy( abc, abc1 );
            d[ 0 ] = d1;
            d[ 1 ] = d2;
            y = 'a';
            }
        }
    ~WW(){}

    static long Metatyp();

    void Dump();
};


class XX : public Persistent, WW {

    short   y;
    long    z;
    char    xy[ 5 ];
    long    dr;

public:

    XX( long id ) : Persistent( Address(), id, Metatyp()),
        WW( Created(), "blubb", 31415, 27879 ){ ..... }
    XX() : Persistent( Address(), Metatyp() ),
        WW( Created(), "blubb", 31415, 27879 ){ ..... }

    ~XX(){ Update(); }

    static long  Metatyp();
    void         Address(){ return( this ); }
void         Dump();
};
```


Table of contents

```
        }
    WW( long id ) : Persistent( Address, id, Metatyp() ){}
    ~WW(){}

    void        PrintStatus(){ cout << Text << endl; }
    static long Metatyp();
    void        *Address(){ return( this ); }
};


class XX : public Persistent{

    DoubleLinkedList<WW>    ww_list;

public:

    XX( long id ) : Persistent( Address(), id, Metatyp() ){}
    XX() : Persistent( Address(), Metatyp() ){}

    ~XX(){ ww_list.Update(); Update(); }

    static long Metatyp();
    void        *Address(){ return( this ); }

        /*
            list operations
        */
    void    PrintStatus(){
        WW  *ww;
        DLL_FORALL( ww_list, ww )
            ww->PrintStatus();
        }

    void    Insert( WW *obj ){ ww_list.InsertFirst( obj ); }
    void    Remove( WW *obj ){ ww_list.Remove( obj ); }
};
```

2.4 Example

The following section of code shows a small example of persistent objects.

```
class WW {

    char    abc[ 7 ];
    long    d[ 2 ];
    char    y;
```

```
        type *l = first.GetPtr(), *ll;
        while( l ) {
            ll = l->GetNext();
            if( db_remove )
                l->Delete();
            delete l;
            l = ll;
            }

        last  = (type*)NULL;
        first = (type*)NULL;
        }

        /*
            destructor: remove list only from memory
        */
    ~DoubleLinkedList() {
        RemoveAll();
        }

        /*
            Update entire list in database
        */
    void Update() {
        type *elem;
        for( elem = GetFirst(); elem; elem = GetNext( elem ))
            elem->Update();
        }
};

/*
    iterator for double-linked list
*/
#define DLL_FORALL(list,elem) \
    for( elem = (list).GetFirst(); elem; \
         elem = (list).GetNext( elem ))
```

2.3.2 Example

The foolowing piece of code contains a small application example for double linked lists.

```
class WW : public Persistent, ListElement<WW> {

    char    Text[ 10 ];

public:
    WW( char *s ) : Persistent( Address(), Metatyp() ){
        strncpy( Text, s, sizeof( Text ));
```

```
        return last element
    */
inline type *GetLast() { return( last.GetPtr() ); }

    /*
        return next list element of a given element
    */
inline type *GetNext( type *obj ) { return( obj->GetNext() ); }

    /*
        return previous list element of a given element
    */
inline type *GetPrev( type *obj ) { return( obj->GetPrev() ); }

    /*
        return number of elements in list
    */
inline short GetCount(){ return( count ); }

    /*
        dequeue element from list, don't remove
        it from memory or database
    */
void Disconnect( type *obj ){
    if( first == obj )
        first = obj->GetNext();
    if( last == obj )
        last = obj->GetPrev();

    obj->RemoveElement();
    }

    /*
        dequeue element, delete it from memory and
        ( if db_remove set ) from database
    */
void Remove( type *obj, boolean db_remove = FALSE ) {
    if( db_remove )
        obj->Delete();
    Disconnect( obj );
    delete obj;
    }

    /*
        remove entire list from memory and
        (if db_remove set ) from database
    */
void RemoveAll( boolean db_remove = FALSE ){
```

```
                            obj->GetId() );
    if( !first )
        first = last;
    count++;
    }

    /*
        insert a new element after a given list element
    */
void InsertAfter( type *obj, type *after ) {
    if( after->GetNext())
        obj->AddElement( after, after->GetNext(), obj,
                         obj->GetId() );
    else
        last = obj->AddElement( after, after->GetNext(), obj,
                                obj->GetId() );
    count++;
    }

    /*
        insert object at a given position,
        0  = as first item
        -1 = as last item
    */
void InsertAt( type *obj, short pos = 0 ) {
    if( !pos || !count )
        InsertFirst( obj );
    else {
        if(( pos == -1 ) || ( pos >= count ))
            InsertLast( obj );
        else {
            short i       = 0;
            type  *search = first.GetPtr();
            do {
                i++;
                search = search->GetNext();
            } while( i < pos );
            InsertAfter( obj, search );
            }
        }
    }

    /*
        return first element
    */
inline type *GetFirst() { return( first.GetPtr() ); }

    /*
```

```
*/
template <class type>
class DoubleLinkedList {

public:

    AutoPersistentPtr<type>   first,   /* first element in list */
                              last;    /* last element in list  */
    short                     count;   /* number of elements    */
    static long               mt;      /* own metatyp           */

    DoubleLinkedList(){}

        /*
            create metatyp-definition
        */
    static long Metatyp() {
        if( !mt ) {
            mt = mt_OpenMetaRecord( sizeof( DoubleLinkedList<type> )
                                    "DoubleLinkedList" );
            DoubleLinkedList<type> *xptr = NULL;
            mt_PushMetaVar( mt, xptr, &xptr->first,
                AutoPersistentPtr<type>::Metatyp(), 1, "first" );
            mt_PushMetaVar( mt, xptr, &xptr->last,
                AutoPersistentPtr<type>::Metatyp(), 1, "last" );

            mt_PushMetaVar( mt, xptr, &xptr->count, MT_SHORT, 1,
                            "count" );
            }
        return( mt );
        }

        /*
            insert element at list head
        */
    void InsertFirst( type *obj ){
        first = obj->AddElement( NULL, first.GetPtr(), obj,
                                 obj->GetId());
        if( !last )
            last = first;
            count++;
            }

        /*
            insert element at last position
        */
    void InsertLast( type *obj ){
        last = obj->AddElement( last.GetPtr(), NULL, obj,
```

```
                            prev;      /* prev. entry    */
    static long             mt;        /* metatyp-handle */

    ListElement(){}
    ~ListElement(){}

    static long  Metatyp(){             /* define metatyp */

        if( !mt ) {
            mt = mt_OpenMetaRecord( sizeof( ListElement<type> ),
                                    "ListElement" );
            ListElement<type> *xptr = NULL;
            mt_PushMetaVar( mt, xptr, &xptr->next,
                AutoPersistentPtr<type>::Metatyp(), 1, "next" );
            mt_PushMetaVar( mt, xptr, &xptr->prev,
                AutoPersistentPtr<type>::Metatyp(), 1, "prev" );
            }
        return( mt );
        }

    type *AddElement( type *last_el, type *next_el, type *this_el,
                      OBJECT_ID this_id ) {
            next = next_el;
            prev = last_el;

            if( next != NULL )
                next->prev.Set( this_el, this_id );

            if( prev != NULL )
                prev->next.Set( this_el, this_id );
            return( (type*)this );
            }

    void RemoveElement() {
            if( next != NULL )
                next->prev.Set( prev.GetPtr(), prev.GetId() );
            if( prev != NULL )
                prev->next.Set( next.GetPtr(), next.GetId() );
            }

    inline type *GetNext() { return( next.GetPtr() ); }
    inline type *GetPrev() { return( prev.GetPtr() ); }
};

class DoubleLinkedList

/*
    can be used only in persistent objects !
```


1

Man machine service

1.1 An overview

1.1.1 Symbols

The basic idea of the man machine service is the introduction of symbols as state-picture structured objects of an application, e.g. process variables of a control unit. The user can de symbols and their behavior on events very flexible with an interactive tool, the *symbol-ed* Symbols are composed of base-symbols, such as lines, circles … and other user-defined symb As a result, symbols may contain a hierarchy of components. These are stored in a configura database for usage within the application. After the configuration or construction, the symbols available in the application. To use them, they have to be connected with an object. Changi value of this object leads to a different graphical representation. Changing the graphical repre tation (e.g. the user moves a symbol interactive) leads to a different object value. The relati between object values and the resulting images can be defined. This relation is either continu where linear or logarithmic functions are provided, or discrete.

All symbols are arranged and positioned in *planes*. Each plane defines a unit of measuren respectively a scale. One symbol can only be assigned to one plane. Planes with all their syml are displayed in windows, using a user defined scaling. It is possible to display multiple plane one window at the same time, using a stack of planes. On the other hand it is possible to visua a plane in more than one window at the same time.

1.1.2 Presentation objects

Symbols are used to visualize a big amount of user defined data types. The *presentation ob* is introduced to offer the developer the facility to group symbols together and to create image complex data type with a special semantic.

1.1.2.1 Predefined presentation objects

There are different types of presentation objects predefined:

- A *picture* is a set of symbols. This is used as an image for a set of application objects. There are no restrictions concerning the object types. The picture is the basis for all other presentation objects.
- A *menu* is an image for an object component of an enumeration type, each button shows a selectable value. Any kind of symbol with a boolean state value is usable as a button.
- A *mask* is an image for an object or a structure of an application: Modifiable components of the object can be changed by the manipulation of the corresponding symbols (sliders, buttons, text fields, ...).
- A *table* is an image of an array of objects or structures.
- A *text-document* is a picture with a special semantic and behavior.
- A *help-document* is a set of text-documents, connected together.
- A *hierarchical graph* is a set of pictures with a predefined behavior.

Some presentation objects can be build automatically by the service if the type of the corresponding object is known at runtime.

1.1.3 Events and bindings for distributed systems

Presentations objects themself are useful for the manipulation and visualization of objects. But to allow the user a communication with an application, he must be able to interact with the entire system by creating events. In traditional user interfaces, the application needs an event-loop to recognize such events. The system is build "around" this loop. This design is not very practicable in service-oriented applications due to the fact, that a service can be composed of many light-weight processes. To overcome this problem, a new mechanism, the *binding*, is introduced. A binding is defined as a connection between an operation and an event on a component of the user interface. The execution of the bounded operation is triggered by the event.
The main properties of this approach are:

- Many internal operations of the man machine service can be bound to events, so that typical user interactions with the system are definable by an interactive GUI tool (see below) without writing any line of code in the application.
- Presentation objects can be bound together to create hierarchical menus, masks and tables.
- User defined operations can be connected to events to create callback functions or methods (in C++). An application is able to catch an event using this technique.
- The interaction with the application is event-driven without the need of an event-loop. The mechanism is not limited to a local computer, it is available system-wide. This contains global call-back functions (or methods in C++) to remote systems.

2.2.3.8 PersistentPtr<type>::GetPtr

```
type *PersistentPtr<type>::GetPtr();
```

GetPtr returns the pointer to the refered object, if the object is in memory.

2.2.3.9 PersistentPtr<type>::Set

```
void PersistentPtr<type>::Set( type *ptr, OBJECT_ID id );
```

Set sets the references object.

- **ptr**
  address of the refered object, or *NULL*
- **id**
  identification of the refered object

2.2.3.10 PersistentPtr<type>::operators

```
type *PersistentPtr<type>::operator->();
bool  PersistentPtr<type>::operator!();
bool  PersistentPtr<type>::operator==( void *ptr );
bool  PersistentPtr<type>::operator==( PersistentPtr ptr );
type *PersistentPtr<type>::operator=( PersistentPtr ptr );
type *PersistentPtr<type>::operator=( void *ptr );
bool  PersistentPtr<type>::operator!=( void *ptr );
bool  PersistentPtr<type>::operator!=( PersistentPtr ptr );
```

2.3 Double Linked List

The file *list.h* contains the definition of a double linked list with persistent objects. The lis
implemented as a template class and must be located in a persistent object. An object, whic
placed into a double linked list, must be derived from the class *ListElement* !

2.3.1 Class definition

```
class ListElement

template <class type>
class ListElement {

public:

    AutoPersistentPtr<type>   next,      /* next entry     */
```

2.2.3.2 PersistentPtr<type>::~PersistentPtr

```
PersistentPtr<type>::~PersistentPtr();
```

The pointer object is removed from memory, the referred object is kept.

2.2.3.3 PersistentPtr<type>::Metatyp

```
long PersistentPtr<type>::Metatyp();
```

Metatyp returns the metatyp handle for the class itself.

2.2.3.4 PersistentPtr<type>::Delete

```
void PersistentPtr<type>::Delete();
```

Delete removes the referred object from memory, not database.

2.2.3.5 PersistentPtr<type>::Rescan

```
LOAD_STATE PersistentPtr<type>::Rescan();
```

Rescan loads the referred object to memory.

- **Return**
  The return-value contains the load state of the object:
  - **NO_OBJECT**: There is no object refered.
  - **OBJECT_LOADED**: Object is loaded to memory.
  - **OBJECT_FOUND**: Object was allready loaded to memory. Only the reference count was increased.
  - **OBJECT_CREATED**: not used here

2.2.3.6 PersistentPtr<type>::ExistsNonPresent

```
boolean PersistentPtr<type>::ExistsNonPresent();
```

ExistsNonPresent determins, if on object is refered but not in memory.

2.2.3.7 PersistentPtr<type>::GetId

```
OBJECT_ID PersistentPtr<type>::GetId();
```

GetId returns the identification code of the referred object.

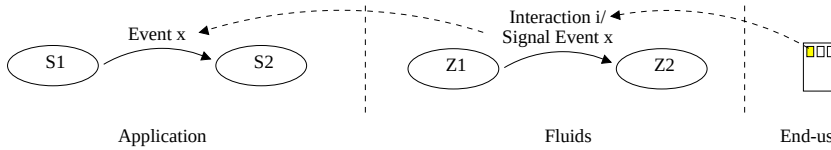

Figure 1.1: User interaction

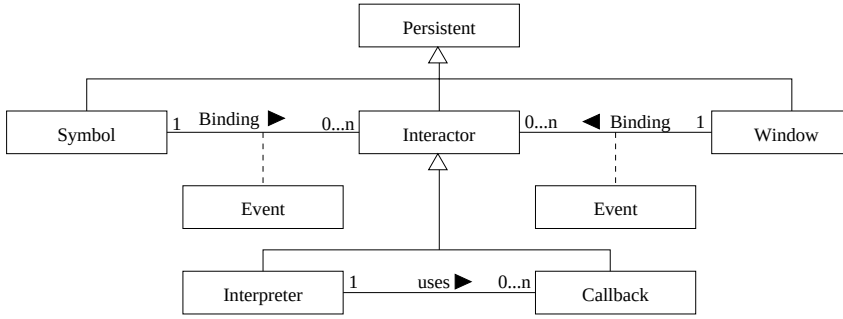

Figure 1.2: Behavior model

- The bindings are changeable during runtime with the goal to allow permanent runti modifications of the GUI behavior.
- The dynamic behavior model is a portably described for different hardware architect without the need of recompilation.
- The handling of all unbound events is simplified by applying default-bindings for these ki of events. These means, that bindings are definable on more than one type of event or on ery unbound event. Furthermore, unbound components of the user interface can be conne by bindings to implement a default-handler.

Presentation objects with a predefined behavior on events are implemented using bindings. jects of a higher level, which are using presentation objects like pictures, must supply their c ponents with task-specific binding functions to have control over the event responding.
An interactive GUI editor allows the developer to create user interfaces, consisting of presenta objects and windows together with bindings in a comfortable way. For runtime or applica defined interfaces a GUI creation or modification by the service API is possible.

1.1.4 Interpreted functions as event handler

Picture 1.2 shows the simplified system structure of the behavior model using UML notation.

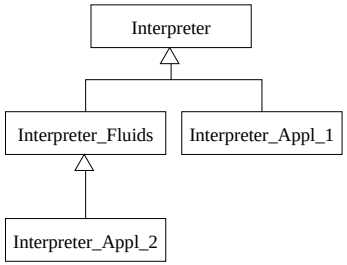

Figure 1.3: Interpreter usage

The user causes events by symbol or window manipulation. If an interactor object is bound to this symbol and the binding condition concerning the event type is true, the specified interactor is called. This function is either coded in a Pascal-like syntax and executed by a built-in interpreter, or a precompiled callback function of the application or man-machine service Fluids. The interpreter normally “knows” a basic set of built-functions, which are now extended by the methods defined on the user interface components. Since every application needs some kind of communication with its user interface, applications can register callback-methods to their own objects to allow asynchronous event notification. These callback-methods are available in the interpreter as ordinary functions. The class-hierarchy makes a distinction between callback- and interpreter interactors to allow an uninterpreted fast call of time-critical callbacks. Special application functionality, which could be required for text editors, is added using this technique. Picture 1.3 shows the scheme:

The basic interpreter has only a core functionality. Derived interpreter classes add more functions for special tasks. The user interface management service builds its own extended interpreter, while special application-dependent interpreters are either derived from the user interface interpreter, if this functionality must be available to the end user too, or directly from the basic interpreter. Because this work is based on the distributed environment *CORBA*, callback-functions are not limited to platform boundaries. For this reason, the interpreter is also able to call remote functions or methods, if their callbacks are registered.

The interpreter and its Pascal-like language are not discussed in detail in this paper, because it is based on the *LUA* interpreter of the the *TeCGraf-Grupo de Tecnologia em Computacao Grafica* in Rio de Janeiro. This interpreter is freely available for commercial and non-commercial applications. More information is also available in the LUA-manual.

1.1.5 Realization

MMS is written in C++ language for manipulating symbols, planes and windows. The platform dependent parts of the MMS are based upon an uniform interface provided by a window service. The functions of this service are grouped in two parts: window manipulations and graphical drawing primitives in windows. This is implemented using distributed objects. The new defined standard for distributed objects "Corba" is not used due to its requirement for a TCP/IP layer. This

know whether the referred object is in main memory, only in database or not present (NULL). important to know this, because the mechanism is completely under the control of the programm He has to request a persistent pointer to reload the referred object or to preempt it. This could very complex task for large data structures, so that there is a slightly modified pointers class, ca *AutoPersistentPtr*. An object of this class reloads the referred object during its own (re-)creat This implies, that data structures, connected by this pointer, are reloading themselves. Only s kinds of base or root objects have to be reloaded on demand. The following sequence of acti starts after reloading the root object:

- If the loaded object contains an *AutoPersistentPtr* as a component, this object is created the language’s runtime control and initialized by the persistent object system.
- The constructor of the created pointer object recreates the referred object if it exists. sequence continues at step 1 until there is no referred object left.

To stop this mechanism, a large data structure should be divided into self loading sub-structu which are connected together by *PersistentPtr* as a breakpoint. There are two major strategy r for large persistent data structures:

- Objects, which are used together, should be kept in self- loading structures.
- Breakpoints separate such groups to keep the memory utilization small.

2.2.2 Multiple references

Introducing relations between persistent objects leads to one problem: How are multiple re ences to one object handled ? Assume, there are two objects A and B referring an object C. W happens when both of them are reloading or preempting C ? The first creation attempt restore in main memory. All other creation-accesses only determine the memory address of C and re it. C is not multiply loaded. This requires a reference counter, which is used to free an object if no references by *PersistentPtr* exist. This is not a common solution since there can be ma copy of a *PersistentPtr* without informing the persistent object system.

2.2.3 PersistentPtr<type>

Persistent references between different databases are not supported in the current version. persistent object system is not able to determine such illegal constructions !

2.2.3.1 PersistentPtr<type>::PersistentPtr

```
PersistentPtr<type>::PersistentPtr( DatabaseObject *db = NULL );
```

The pointer refers an oject of the given database object.

- **db**
  Database handle or *NULL*, if the referred object is placed in the default database.

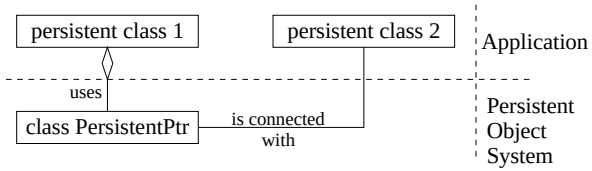

Figure 2.2: Persistent pointer

2.1.4.8 Persistent::GetDataObject

```
void *Persistent::GetDataObject();
```

GetDataObject return the address of the passive data object, assigned to this persistent object, if this data object exists. The memory can be overwritten or read.

2.1.4.9 Persistent::GetDatabaseObject

```
DatabaseObject *Persistent::GetDatabaseObject();
```

GetDatabaseObject return the database handle.

- **Return**
  Database handle or *NULL*, if the object is assigned to the default database.

2.2 Persistent pointers

In most object-oriented applications it is necessary not only to keep objects persistent, but to hold the relations between objects. A very simple example is a doubly-linked list of persistent objects, which has to be saved as a totality. Relations between objects sometimes lead to very large data structures, of which not every part is used in any situation. To ensure an efficient main memory utilization a mechanism for a dynamic load and preemption of structure compo- nents should be available. An example is the above mentioned man machine service: It holds only the visible user interfaces in main memory.

2.2.1 Characteristics and design

Relations between non-persistent objects are normally realized by pointers. The main idea behind the construction of persistent relations is the introduction of a special pointer class, called *PersistentPtr*. In addition to the pointer, which is only valid in main memory, it contains the destination object's identification code too. Therefore this pointer class can be used only for persistent objects. The next picture shows the class structure in OMT notation for persistent pointers.
The use of a real pointer together with the identification number is important for an additional facility: The mechanism for a dynamical reload and preemption of persistent objects needs to

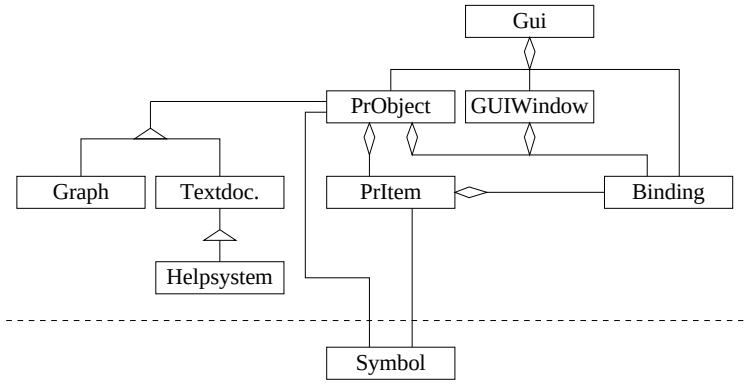

Figure 1.4: Realized class hierarchie

is not possible in some kind of automation applications.
GUI's are stored in a database, using persistent objects, to ensure reuseability in different proj
and to allow the unmodified use on other hardware platforms.
Figure 1.4 presents the internal structure of the service without the persistent object layer.

1.2 GUI and presentation objects

The following sections describe the C++ API for the man machine service.

1.2.1 Initialization: System class

It is important to do some initialization before using the MMS API. The *System*-Class han
most initializations.

1.2.1.1 System::System

```
System::System( const char *name,
                boolean network = FALSE,
                boolean lwp = TRUE );
```

This operation starts the internal system, including task switching and networking.

- **name**
  Name of this computer.

- **network**
  TRUE, if networking is needed

- **lwp**
  If *lwp* is *TRUE*, internal processes are created as lightweight processes. If this flas is not set, heavy-weight processes are used. In the current version, this flag is only needed for UNIX-systems. On all other systems it should be set to *TRUE*.

1.2.1.2 System::~System

```
System::~System()
```

This operation stops the internal system, including task switching and networking.

1.2.1.3 System::InitPersistentObjectSystem

```
void System::InitPersistentObjectSystem( const char *def_database );
```

InitPersistentObjectSystem is invoked after the system creation. This operation starts the persistent object system and loads or creates the default database. There is no database handle returned, since all persistent object operations provide a default parameter for the default database. This database can't be switched during the lifetime of the application.

- **def_database**
  Name of the default database

1.2.1.4 System::ExitPersistentObjectSystem

```
void System::ExitPersistentObjectSystem();
```

ExitPersistentObjectSystem terminates the persistent object system and closes all open databases. This operation mulst be called deleting the system-object.

1.2.1.5 System::OpenDatabase

```
DatabaseObject *System::OpenDatabase( const char *name );
```

OpenDatabase opens an addtional database, which can be used for persistent objects. A total of 4 databases can be kept open at the same time. If no database with the given exists, a new will be created.

- **name**
  database name

- **Return**
  database handle, which is used as a parameter for persistent objects

2.1.4.2 Persistent::~Persistent

```
Persistent::~Persistent();
```

The object is removed from memory, not from database.

2.1.4.3 Persistent::Update

```
void Persistent::Update();
```

Update writes the object's contents into the database, using the local stub to convert the ob
into a unique database format.

2.1.4.4 Persistent::Undo

```
void Persistent::Undo();
```

Undo overwrites the object's contents with the value in the database.

2.1.4.5 Persistent::Delete

```
void Persistent::Delete();
```

Delete removes the object only from the database, but not from the memory. The applicatio
Undo and *Update* on this object is disabled after a removal.

2.1.4.6 Persistent::GetId

```
OBJECT_ID Persistent::GetId();
```

GetId returns the unique identification code.

- **Return**
  unique object handle

2.1.4.7 Persistent::Created

```
boolean Persistent::Created();
```

Created is used to determine whether this object is newly created or loaded from the datab
This method is useful in a constructor of a persistent class to do some initializations only
newly created objects.

- an optional identification number and
- a reference to a database stub.

The type-id is necessary to enable the object system to access the metatype information. If no identification is given during the first creation, a unique one will be determined. The con nection with a database server is managed through the referenced stub object, which has to be created first. The behavior of persistent objects is controllable through the interface of the class *Persistent*. The initialization and database access for persistent objects is described in 1.2.1.3.

2.1.4.1 Persistent::Persistent

```
Persistent::Persistent( void *addr, OBJECT_ID id, long metatyp,
                        DatabaseObject *db = NULL );
Persistent::Persistent( void *addr, long metatyp,
                        DatabaseObject *db = NULL );
```

Only for compatibility with older version, the old constructors are still active. Note: Using the old constructors, virtual functions are not allowed in persistent classes !

```
Persistent::Persistent( OBJECT_ID id, long metatyp,
                        DatabaseObject *db = NULL );
Persistent::Persistent( long metatyp,
                        DatabaseObject *db = NULL );
```

The first constructor is used to create a new persistent object, its identification code is determined automatically. The second one creates a local copy of an existing object and reloads its contents or creates a new one with the given id, if no object with this id exists. It is possible to omit a stub, if one of the stubs is marked as default.

- **addr**
  Object address has to be set, because in classes with virtual function, the address cannot be determined by the persistent class itself.
- **id**
  object identification
- **metatyp**
  Metatyp information for the corresponding class: This parameter is not required, it can be replaced by a macro, which determins the size of the object in main memory:
  - **MT_EMPTY_OBJECT(x)**: The object does not have metatyp information.
  - **MT_EMPTY_DATA**: An object is created from *Persistent*, only a passive data object is bound to the persistent object. No metatyp information for the data object is available.
  - **MT_DATA**: An object is created from *Persistent*, only a passive data object is bound to the persistent object. Metatyp information for the data object is available.
- **db**
  Database handle, if the object is not stored in the default database, or *NULL*, if the default database is used.

1.2.1.6 System::CloseDatabase

```
void System::OpenDatabase( DatabaseObject *db );
```

CloseDatabase closes an addtional database. The default database cannot be closed, until persistent object system is ended.

- **db**
  database handle

1.2.1.7 System::InitMms

```
MMS *System::InitMms( const char *esa, short x, short y );
```

InitMms starts the man machine service. This method must be called after creating the sys object and starting the persistent object system.

- **esa**
  base name of the file, containing the symbol descriptions
- **x,y**
  These paramaters are used in the current version to set the default size of the main win of the application.
- **Return**
  handle to the man machine service

1.2.1.8 System::ExitMms

```
void System::ExitMms();
```

ExitMms terminates the man machine service. Since there can be only one service active in current version, no handle has to be supplied.

1.2.1.9 MMS::MaxWidth

```
coord MMS::MaxWidth();
```

MaxWidth determins the maximum width of the main window. This method does not make m sense in window systems like X-Windows since the window size may change. If there is underlying window system (e.g. in DOS), this method returns the screen resolution.

- **Return**
  x-coord, given in *System::InitMms* call.

1.2.1.10 MMS::MaxHeight

```
coord MMS::MaxHeight();
```

MaxHeight determins the maximum height of the main window. This method does not make much sense in window systems like X-Windows since the window size may change. If there is no underlying window system (e.g. in DOS), this method returns the screen resolution.

- **Return**
  y-coord, given in *System::InitMms* call.

1.2.1.11 MMS::FirstSymbolType

```
short MMS::FirstSymbolType( char *type );
```

FirstSymbolType finds the first defined symbol type.

- **type**
  type returns the name of the first symbol type, or an empty string, if no symbol is found.
- **Return**
  Index number, used for calls to *NextSymbolType*.

1.2.1.12 MMS::NextSymbolType

```
short MMS::NextSymbolType( short index, char *type );
```

NextSymbolType finds the next defined symbol type.

- **index**
  index contains the index of the previously found symbol type.
- **type**
  type returns the name of the found symbol type or an empty string, if no more symbols are predefined.
- **Return**
  Index number, used for more calls to *NextSymbolType*.

1.2.1.13 MMS::GetSymbolMetatype

```
long MMS::GetSymbolMetatype( const char *symbol_type );
```

GetSymbolMetatype returns the metatyp handle for the given symbol type.

- **symbol_type**
  name of the symbol type
- **Return**
  Metatype handle or *MT_UNDEF*, if the symbol type is unknown. The return value may not the same in different runs of the application.

- **db**
  database handle: *NULL* for the default database, or the handle returned from *OpenDatab*

The following sequence of lines does all necessary initialization. *PersistentObjectSystem* is an ternal global variable which must be set in order to ensure a correct handling of persistent obje

```
MetatypContainer  *MetatypServer[ MAX_OPEN_DATABASES ];

void InitPersistentObjectSystem( char *def_database ) {

    for( short i = 0; i < MAX_OPEN_DATABASES; i++ )
        MetatypServer[ i ] = NULL;

    PersistentObjectSystem = new DatabaseContainer( def_database );
    MetatypServer[ 0 ]     = new MetatypContainer();
}
```

2.1.3 Exitialize Persistent Object System

The following sequence of lines does all necessary exitialization after updating the databases.

```
void ExitPersistentObjectSystem() {

    for( short i = 0; i < MAX_OPEN_DATABASES; i++ ) {
        if( MetatypServer[ i ] ) {
            MetatypServer[ i ]->Update();
            delete MetatypServer[ i ];
            MetatypServer[ i ] = NULL;
            }
        }

    if( PersistentObjectSystem ) {
        delete PersistentObjectSystem;
        PersistentObjectSystem = NULL;
        }
}
```

2.1.4 Persistent

The described approach for persistent objects is totally embedded into C++. As written ab every persistent class is derived from the internal class "Persistent". Since version 1.2 vir functions in persistent classes are allowed.
The constructor is provided with

- the address of the corresponding class,
- the type-id of the corresponding class,

each program execution cycle. As a solution, every persistent object is provided with a unique identification code. This is either granted by the internal system during the first dynamical creation of the object or given out by the programmer for static objects. This second mechanism for static objects is necessary to allow the storage of an object's reference in the program code.

- **Runtime type information**
  The next problem is the desired platform independence for persistent objects. Since the selected programming language C++ does not provide full runtime type information, a more mighty mechanism has to be created. A persistent class has an internal method called "Metatyp" that is first internally called by the object itself to specify its own type information and second used by other objects to determine the metatyp of this object. This method uses a local metatype server to store its type information. First, it creates a new metatype with its own class-name. Within the next steps it describes each class component using

  - its name,
  - its position relative to the object together with its size and
  - its type.

  "Metatyp" is automatically created for persistent classes by a precompiler to reduce the programming overhead and avoid errors. If the persistence is limited to one hardware plat form, the type information can be omitted.

  The database stub uses this method to get the full type information for a persistent object when the object is stored or loaded. Therefore the object can be stored in a unique data base format.

2.1.2 Initialize Persistent Object System

```
DatabaseContainer::DatabaseContainer( char *def_database );
```

Creating a database container object initializes the persistent object system and opens or creates the given database as the default database for all other accesses. More databases can be accessed by using *OpenDatabase*. Only one database container should be activ.

- **def_database**
  name of the default database
- **Return**
  handle to a database container, used to create additional databases

```
MetatypConatiner::MetatypContainer( DatabaseObject *db = NULL );
```

A new metatype server is created in or reloaded from a database. If no database is given, the metatype server is built for the default database.

1.2.1.14 MMS::CreatePlane

```
plane MMS::CreatePlane( const char *plane_name );
```

CreatePlane creates a new plane with a given name. A unique handle is returned. If a plane v
this name exists already, only the handle is returned and the internal reference count is increa
This method is not used in normal application programs.
The following constants represent the characteristics of planes:

- **MAX_SIMU_PLANE**: maximum number of simultaneously used planes
- **MAX_PLANE_NAME**: maximum plane name length

The method parameters are:

- **plane_name**
  unique plane name
- **Return**
  plane handle or one of the followeing error-codes:
  - **MMS_TOO_MANY_PLANES**: The number of simultaneously used planes exce
    MAX_SIMU_PLANE.
  - **MMS_PLANE_NOT_FOUND**: This return code is only used in the find-methods
    is set, if a plane can not be found.

1.2.1.15 MMS::DeletePlane

```
void MMS::DeletePlane( plane handle );
```

DeletePlane decrements the reference count and removes the plane, if no more references exi

- **handle**
  unique plane handle, return by *CreatePlane*

1.2.1.16 MMS::FindPlane

```
plane MMS::FindPlane( const char *name );
char *MMS::FindPlane( plane handle );
```

FindPlane searches for a plane with the given name or handle.

- **handle**
  unique plane handle, return by *CreatePlane*
- **name**
  unqiue plane name
- **Return**
  name or handle

1.2.1.17 GuiList::GuiList

```
GuiList::GuiList( DatabaseObject *db = NULL );
```

A GuiList contains a directory of every gui in the given database or the main database. If no GuiList object is found in the database, an empty object is created. There can be only one of such directories in the same database.

- **db**
  database handle

1.2.1.18 GuiList::~GuiList

```
GuiList::~GuiList();
```

Deleting such object removes the directory from main memory. The contents is not written to the database.

1.2.1.19 GuiList::Metatyp

```
static long GuiList::Metatyp();
```

Metatyp gets the metatype handle for the gui list itself.

- **Return**
  metatype handle

1.2.1.20 GuiList::FirstGui

```
OBJECT_ID GuiList::FirstGui();
```

FirstGui returns the unique identification code of the first gui in the directory. The gui is not loaded to memory.

- **Return**
  unique gui handle

1.2.1.21 GuiList::NextGui

```
OBJECT_ID GuiList::NextGui( OBJECT_ID last_gui );
```

NextGui returns the unique identification code of the next gui in the directory. The gui is not loaded to memory.

- **last_id**
  previous gui id
- **Return**
  unique gui handle of the next gui

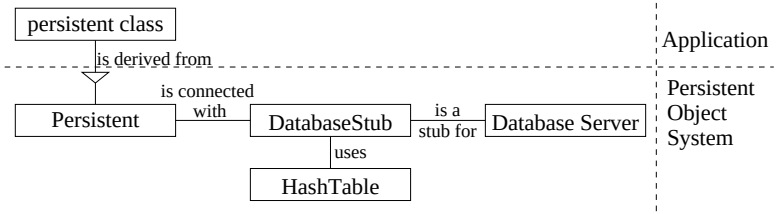

Figure 2.1: Internal data structure

of full type information at runtime, which could not be derived from RTTI in C++. Ev object can provide the persistent object system with its own type description.

- **Realtime access**
  This solution allows realtime access to the database if the underlying operating system realtime capabilities.

2.1 Concept of persistent objects

As mentioned above, the main goal in the introduction of persistent objects is a "natural" emb ding into a given object-oriented programming language (here C++). The intention is to hide n of the additional functionality from the programmer by applying a clear object-oriented des Persistent objects should be usable like any other non- persistent object.

2.1.1 Characteristics and design

Persistent objects are realized by declaring the corresponding class as persistent. This is done deriving these classes from the internal class "Persistent" on the applications side. On the syst internal side every persistent object is connected to one (of several possible) database objects. T acts as a stub for the real local or remote database server in a heterogeneous net of computers. connection to a stub is dynamically changeable so that an object's content can be loaded fron stored to different databases. The advantage of this design is that a persistent object can be pla anywhere on a net. No object has to know its own storage place. Only the reference to the lo stub is kept. All database stubs share the same machine-dependent hash table for two purpo First, to determine whether an object is already in memory and second, to find the memory add for a given object identification. The following picture shows the internal structure together w the application interface. The diagram uses the OMT notation for data structures.
The following two attributes are characteristics of persistent objects:

- **Unique identification**
  The first problem in persistence that has to be solved is the unambiguous identificatio objects in external and internal memory, because internal memory addresses can diffe

2

Persistent objects

In the above mentioned man machine service every component of the user interface is represented by an object. Their dependencies and links are expressed by relations between objects. The major problem that arises from such a design is, that there is no standard way to keep the contents of objects together with their relations persistent without giving up a clear object oriented design. Persistent objects are objects in a programming language, which are able to survive a program execution cycle. This means, that their content is not lost after a program crash or normal termination because it is saved in application defined periods of time in an external memory. To assure this functionality in traditional systems, often an application dependent store-and- load mechanism is used to write and restore the object's contents. This can be done by placing persistent objects into a special container object, which itself is responsible for the object management. This has one major disadvantage: persistent objects cannot be treated like other non-persistent objects. A much better approach is to request an object to save its own state or to restore it, because of the following main advantages:

- **Embedding**
  The access to persistent objects is identical to non- persistent objects. There are only additional functions to control the load/save mechanism.

- **Ease of use**
  The programmer can reuse this mechanism in every application without any change. The object interface is identical.

- **State dependence**
  The states of critical or other important objects must be accessible even after a program crash either to find the problem by examining the last state values or to reinitialize the program during a new-start. An object is able to request itself to save its own state after major or important changes.

- **Platform independence**
  A major problem in the above mentioned man machine service was the reuse of user interfaces on different hardware platforms like Sparc-Workstations and Intel-PC's with a variety of compilers. As a result, there was no guarantee that saved objects on one system are usable on other systems due to alignment and byte-order problems. This requires the availability

1.2.1.22 GuiList::FindGui

```
OBJECT_ID GuiList::FindGui( const char *name );
```

FindGui returns the unique identification code of the gui with the given name or 0, if no such exists.

- **name**
  gui name

- **Return**
  Unique gui handle or 0, if no gui was found with this name.

1.2.1.23 GuiList::FindGui

```
const char *GuiList::FindGui( OBJECT_ID id );
```

FindGui returns the name of gui, for which the unique identification code was given.

- **id**
  gui handle

- **Return**
  Gui name, or an empty string, if the mentioned gui is not in the directory.

1.2.1.24 GuiList::InsertGui

```
void GuiList::InsertGui( Gui *gui );
```

InsertGui inserts the given gui into the directory.

- **gui**
  gui to insert

1.2.1.25 GuiList::RemoveGui

```
void GuiList::RemoveGui( Gui *gui );
```

RemoveGui removes the given gui from the directory.

- **gui**
  gui to remove

1.2.1.26 GuiList::Update

```
void GuiList::Update();
```

Update writes the directory to the database.

1.2.2 Gui

The container class for all other user interface objects is the *Gui* [1]. It groups the interface objects for a special task of the application.

1.2.2.1 Gui::Gui

```
Gui::Gui( const char *name, DatabaseObject *db = NULL );
Gui::Gui( Gui *copy_gui, DatabaseObject *db = NULL );
Gui::Gui( long db_id, DatabaseObject *db = NULL );
```

The gui-object is created with a given unique database identification to reload an existing gui from the database or to create one with a special id. To create a new, empty gui with an automaticly assigned id, the constructor should be used in its third form. The second constructor is needed to build a new gui as a copy from an existing one. All Bindings and Components are copied too !

- **db**
  If given, an other as the default database can be specified.
- **name**
  (unique) name for the gui
- **db_id**
  unique database id
- **copy_gui** Existing gui, which is used as a template for the new one

1.2.2.2 Gui::~Gui

```
Gui::~Gui();
```

The gui object is removed from memory, but not from the database.

1.2.2.3 Gui::Metatyp

```
static long Gui::Metatyp();
```

This operation returns a unique metatyp handle for the gui class itself. The handle is only valid for a single run of an application. **In an other run, the value can be different**.

[1] Gui = graphical user interface

1.3.14.16 LABEL

LABEL is a predefined symbol for a structure of type *SymbolStringAttr*. The global object *SymbolLabelAttr* can be used to initialize a label with standard fonts and font attributes.

1.3.14.17 TITEL

TITEL is a predefined symbol for a structure of type *SymbolStringAttr*. The global object *SymbolTitelAttr* can be used to initialize a label with standard fonts and font attributes.

1.3.14.18 A-EDIT

A-EDIT is a predefined symbol for a structure of type *SymbolStringAttr*. The global object *SymbolEditAttr* can be used to initialize a label with standard fonts and font attributes.

1.3.14.19 S-EDIT

S-EDIT is a predefined symbol for a structure of type *SymbolShortAttr*.

1.3.14.20 L-EDIT

L-EDIT is a predefined symbol for a structure of type *SymbolLongAttr*.

1.3.14.21 F-EDIT

F-EDIT is a predefined symbol for a structure of type *SymbolFloatAttr*.

1.3.14.22 D-EDIT

D-EDIT is a predefined symbol for a structure of type *SymbolDoubleAttr*.

Figure 1.5: Button symbol, released and pressed state

Figure 1.6: Switch symbol, passive and active state

- **Selected**
  Selection state (*TRUE* or *FALSE*)

- **Name**
  Name of the BMP-File, representing the symbol

1.3.14.10 Symbols

1.3.14.11 BUTTON

BUTTON is a predefined symbol for a structure of type *SymbolStringAttr.*

1.3.14.12 TOGGLE

TOGGLE is a predefined symbol for a structure of type *boolean.* It looks like a button, used to switch between the two possible values.

1.3.14.13 BITMAP

BITMAP is a predefined symbol for a structure of type *SymbolBitmapAttr.* It is displayed using a BMP-File.

1.3.14.14 HSLIDER

HSLIDER is a predefined symbol for a structure of type *SymbolSliderAttr.* It looks like a horizontal slider, used to select a float value between two boundaries.

1.3.14.15 VSLIDER

VSLIDER is a predefined symbol for a structure of type *SymbolSliderAttr.* It looks like a vertical slider, used to select a float value between two boundaries.

1.2.2.4 Gui::SimulateEvent

```
void Gui::SimulateEvent( const PrObject *object,
                         const PrItem *item,
                         const GuiWindow *gui_win,
                         EVENT_TYPE event, long value,
                         WeltKoord mouse_x, WeltKoord mouse_y );
```

SimulateEvent initiates an event on an user interface object. During normal operation, events only generated by the gui thread (event handler) and transmitted to the destination object. In so rare cases — internal: bindings — it can be necessary for an object to simulate events on o objects.

- **object**
  the destination object, which receives the event

- **item**
  the item in the destination object, which receives the event (this parameter can be *NULL*

- **gui_win**
  the given window receives the event

- **event**
  the type of the event, refer 1.3.2 for details.

- **value**
  The value of an event depends on the event type: For example, a mouse-click event ke the value of the pressed mouse button. See 1.3.3 for details.

- **mouse_x, mouse_y**
  The position of the mouse pointer during the event or -1, if not required.

1.2.2.5 Gui::ChangeFocus

```
void Gui::ChangeFocus( PrObject *object, PrItem *item,
                       WeltKoord x_pos = -1,
                       WeltKoord y_pos = -1 );
```

In the current version, *ChangeFocus* is only defined for mask, menu and table objects to cha the keyboard focus between different items. The gui initiates first an *EVENT_LEAVE_FOC* event on the object and item, which will loose the focus and second an *EVENT_ENTER_FOC* on the new object and item. These events are normally interpreted in the user interface and ca discarded in normal applications when not required for other purposes.

- **object**
  address of the object, which receives the focus

- **item**
  address of the item of *object*, which receives the focus. Both — *object* and *item* — must be non zero values to set a new focus. If *object* or *item* are *NULL*, the focus is only removed.

- **x_pos, y_pos**
  In addition, the position of the mouse pointer can be supplied to allow the recalculation of the cursor position for textfield items. If the position is not given, the cursor is set on the first item. On other input symbols, the behaviour is similar or the position is not interpreted.

1.2.2.6 Gui::Execute

```
void Gui::Execute( GUI_EXEC_FLAG flag = GUI_EXEC_EXECUTE );
```

This operation is used to switch the event generation for a given gui on or off. After the (re-)creation, the gui is not generating events, until this method is called.

- **flag**
  flag specifies the execution level, refer 1.3.10 for details.

1.2.2.7 Gui::Event

```
boolean Gui::Event( EVENT *ev, boolean wait = TRUE );
```

Event determines, whether a new event for the gui is available or not.

- **ev**
  If a new event is available, this parameter is filled with the event specification. See 1.3.3 for details.

- **wait**
  If *wait* is set, the execution is suspended, until an event has occured.

- **Return**
  The return value is *TRUE*, when an event is available, or *FALSE*, if not.

1.2.2.8 Gui::Show

```
void Gui::Show( boolean flag = TRUE );
```

Show shows or hides a user interface.

- **flag**
  Specifies, whether the gui should be shown or hidden. If the gui is not shown, the event generation for this gui is switched off. It will not be switched on automaticly after a redisplay.


```
    float           value,
                    minimum,
                    maximum;
public:

    SymbolSliderAttr( float val  =   0.0,
                      float mini =   0.0,
                      float maxi = 100.0 );
    SymbolSliderAttr( double val,
                      double mini =   0.0,
                      double maxi = 100.0 );
    ~SymbolSliderAttr(){}

    float   Value();
    void    Value( float val );

    float   Minimum();
    void    Minimum( float mini );

    float   Maximum();
    void    Maximum( float maxi );
};
```

- **Value**
  Value contains the displayed value.

- **Minimum, Maximum**
  Minimum and maximum values with the following condition:
  Minimum $\leq$ Value $\leq$ Maximum

1.3.14.9 SymbolBitmapAttr

This type represents a bitmap symbol, no editing is possible (only selection).

```
class SymbolBitmapAttr {

    short           selected;
    char            name[ MAX_INPUT_LEN ];
public:

    SymbolBitmapAttr( const char *name = NULL );
    ~SymbolBitmapAttr(){}

    short   Selected();
    void    Selected( short sel );

    char    *Name();
    void    Name( const char *nn );
};
```

```
    ~SymbolDoubleAttr(){}

    short   Selected();
    void    Selected( short sel );

    float   Float();
    void    Float( float f );

    short   HighlightSymbol();
    void    HighlightSymbol( short hls );

    ushort  EditAttributes();
    void    EditAttributes( ushort ea );

    ushort  MaxLen();
    void    MaxLen( ushort ml );

    short   CrsrPos();
    void    CrsrPos( short cp );

    short   RelXCrsrPos();
    void    RelXCrsrPos( short rel );

    short   Font();
    void    Font( short f );

    short   FontSize();
    void    FontSize( short s );

    short   FontColor();
    void    FontColor( short c );

    short   FontAttributes();
    void    FontAttributes( short a );
};
```

Most components are explained in 1.3.14.3. Individual attributes for every character are not supported. The font attributes are explained in 1.3.14.2.

- **Double**
  Double contains the displayed value.

1.3.14.8 SymbolSliderAttr

This type represents a symbol for a float value, displayed and edited in slider representation.

```
class SymbolSliderAttr {
```


1.2.2.9 Gui::CreatePrObject

```
PrObject *Gui::CreatePrObject( PR_TYPE pr_type,
                               const char *name,
                               DESIGN *design,
                               const char *plane_name,
                               long data_type_mt,
                               boolean empty,
                               const char *back_symbol );
PrObject *Gui::CreatePrObject( const PrObject copy_po );
```

CreatePrObject creates a new user interface object in the given gui. The second constructor cre
a new object as a copy of an existing object. All Bindings and items are copied too.

- **pr_type**
  the type of the object (mask, menu etc.), refer 1.3.5 for details.

- **name**
  optional name for the object through which the object can be searched. An empty strin
  also valid.

- **design**
  position, size and orientation of the object, refer 1.3.7 for details. Menus, masks and tab
  which are created using the metatype, return the calculated position and size of the objec

- **plane_name**
  The plane name, in which the object is shown. An object can be created in its own win
  using the macro *OWN_WINDOW*. In this case, the object is placed in the lower left co
  of the window. All sizing commands modify the window position. The plane gets the n
  of the PrObject (see parameter: *name*). In this case, it is required to set *name*.

- **data_type_mt**
  the metatyp handle for the corresponding object of the application or -1, if the inter
  object is not a picture for an application object (like PR_PICTURE f.e.).

- **empty**
  If *empty* is set, the object is not generated with all items, using the type information *da*
  type_mt. If *empty* is set, the object is build with a standard design and behaviour from
  metatyp description.

- **back_symbol**
  The name of an optional background symbol for this object. The symbol should not con
  any state variable, since these are not set.

- **copy_po**
  Original object, which is not modified

1.2.2.10 Gui::DeletePrObject

```
void Gui::DeletePrObject( PrObject *object );
void Gui::DeletePrObject( F_MODIFIER fmod );
```

DeletePrObject removes an object from the gui and database. This object cannot be accessed any longer. The second form deletes all objects with a given attribute (see section 1.3.4).

- **object**
  address of the object or *NULL*, if all objects of this gui are to be deleted

- **fmod**
  Unique identifier (e.g.: marked)

1.2.2.11 Gui::FirstPrObject

```
PrObject *Gui::FirstPrObject( F_MODIFIER fmod = F_ALL );
```

FirstPrObject returns the first object out of the list of objects in the gui. The selection can be bound to objects of a given type (see section 1.3.4).

- **Return**
  address, if such an object exists, or *NULL*

- **fmod**
  attribute, default: ignore attribute (F_ALL)

1.2.2.12 Gui::NextPrObject

```
PrObject *Gui::NextPrObject( PrObject *object,
                             F_MODIFIER fmod = F_ALL );
```

NextPrObject returns the next object out of the list of objects in the gui. The selection can be bound to objects of a given type (see section 1.3.4).

- **object**
  previous object

- **Return**
  address, if a next object exists, or *NULL*

- **fmod**
  attribute, default: ignore attribute (F_ALL)


```
    void    EditAttributes( ushort ea );

    ushort  MaxLen();
    void    MaxLen( ushort ml );

    short   CrsrPos();
    void    CrsrPos( short cp );

    short   RelXCrsrPos();
    void    RelXCrsrPos( short rel );

    short   Font();
    void    Font( short f );

    short   FontSize();
    void    FontSize( short s );

    short   FontColor();
    void    FontColor( short c );

    short   FontAttributes();
    void    FontAttributes( short a );
};
```

Most components are explained in 1.3.14.3. Individual attributes for every character are not s
ported. The font attributes are explained in 1.3.14.2.

- **Float**
  Float contains the displayed value.

1.3.14.7 SymbolDoubleAttr

This type represents a symbol for a double value, displayed and edited in string representatio
created object is initialized with reasonable values as font attributes.

```
class SymbolDoubleAttr {

    double          _double;
    short           selected;
    CharAttr        cattr;
    unsigned short  edit_attributes,
                    max_len;
    signed   short  crsr_pos,
                    highlight_symbol,
                    rel_x_crsr_pos;
public:

    SymbolDoubleAttr( double val = (double)0.0 );
```

```
    void    Font( short f );

    short   FontSize();
    void    FontSize( short s );

    short   FontColor();
    void    FontColor( short c );

    short   FontAttributes();
    void    FontAttributes( short a );
};
```

Most components are explained in 1.3.14.3. Individual attributes for every character are not supported. The font attributes are explained in 1.3.14.2.

- **Long**
  Long contains the displayed value.

1.3.14.6 SymbolFloatAttr

This type represents a symbol for a float value, displayed and edited in string representation. A created object is initialized with reasonable values as font attributes.

```
class SymbolFloatAttr {

    float           _float;
    short           selected;
    CharAttr        cattr;
    unsigned short  edit_attributes,
                    max_len;
    signed   short  crsr_pos,
                    highlight_symbol,
                    rel_x_crsr_pos;
public:

    SymbolFloatAttr( float val = 0.0 );
    ~SymbolFloatAttr(){}

    short   Selected();
    void    Selected( short sel );

    float   Float();
    void    Float( float f );

    short   HighlightSymbol();
    void    HighlightSymbol( short hls );

    ushort  EditAttributes();
```


1.2.2.13 Gui::FindPrObject

```
PrObject *Gui::FindPrObject( const char *name );
```

FindPrObject searches for an object with the given name in the gui.

- **name**
  name of the object

- **Return**
  address, if such an object exists, or *NULL*

1.2.2.14 Gui::CreateGuiWindow

```
GuiWindow *Gui::CreateGuiWindow( const char *name,
                                 const WIN_ATTRIB *wa );
GuiWindow *Gui::CreateGuiWindow( GuiWindow *copy_guiw );
```

CreateGuiWindow creates a window in the given gui. A copy of an existing window is crea
using the second form.

- **name**
  optional name for the window through which the window can be searched. An empty st
  is also valid.

- **wa**
  position, size and other window attributes, refer 1.3.12 for details.

- **copy_guiw**
  template for the new window

1.2.2.15 Gui::DeleteGuiWindow

```
void Gui::DeleteGuiWindow( GuiWindow *win );
```

DeleteGuiWindow removes a window from the gui and database. This window cannot be acces
any longer.

- **win**
  address of the window

1.2.2.16 Gui::FirstGuiWindow

```
GuiWindow *Gui::FirstGuiWindow( F_MODIFIER fmod = F_ALL );
```

FirstGuiWindow returns the first window out of the list of windows in the gui. The selection can be bound to windows of a given type (see section 1.3.4).

- **Return**
  address, if such a window exists, or *NULL*

- **fmod**
  attribute

1.2.2.17 Gui::NextGuiWindow

```
GuiWindow *Gui::NextGuiWindow( GuiWindow *win,
                               F_MODIFIER fmod = F_ALL );
```

NextGuiWindow returns the next window out of the list of windows in the gui. The selection can be bound to windows of a given type (see section 1.3.4).

- **win**
  previous window

- **Return**
  address, if a next window exists, or *NULL*

- **fmod**
  attribute

1.2.2.18 Gui::FindGuiWindow

```
GuiWindow *Gui::FindGuiWindow( const char *name );
```

FindGuiWindow searches for a window with the given name in the gui.

- **name**
  window name

- **Return**
  address, if such a window exists, or *NULL*

Most components are explained in 1.3.14.3. Individual attributes for every character are not s
ported. The font attributes are explained in 1.3.14.2.

- **Short**
  Short contains the displayed value.

1.3.14.5 SymbolLongAttr

This type represents a symbol for a long value, displayed and edited in string representation
created object is initialized with reasonable values as font attributes.

```
class SymbolLongAttr {

    long            _long;
    short           selected;
    CharAttr        cattr;
    unsigned short  edit_attributes,
                    max_len;
    signed   short  crsr_pos,
                    highlight_symbol,
                    rel_x_crsr_pos;
public:

    SymbolLongAttr( long val = (long)0 );
    ~SymbolLongAttr(){}

    short   Selected();
    void    Selected( short sel );

    long    Long();
    void    Long( long l );

    short   HighlightSymbol();
    void    HighlightSymbol( short hls );

    ushort  EditAttributes();
    void    EditAttributes( ushort ea );

    ushort  MaxLen();
    void    MaxLen( ushort ml );

    short   CrsrPos();
    void    CrsrPos( short cp );

    short   RelXCrsrPos();
    void    RelXCrsrPos( short rel );

    short   Font();
```

```
class SymbolShortAttr {

    short           _short,
                    selected;
    CharAttr        cattr;
    unsigned short  edit_attributes,
                    max_len;
    signed   short  crsr_pos,
                    highlight_symbol,
                    rel_x_crsr_pos;
public:

    SymbolShortAttr( short val = 0 );
    ~SymbolShortAttr(){}

    short   Selected();
    void    Selected( short sel );

    short   Short();
    void    Short( short s );

    short   HighlightSymbol();
    void    HighlightSymbol( short hls );

    ushort  EditAttributes();
    void    EditAttributes( ushort ea );

    ushort  MaxLen();
    void    MaxLen( ushort ml );

    short   CrsrPos();
    void    CrsrPos( short cp );

    short   RelXCrsrPos();
    void    RelXCrsrPos( short rel );

    short   Font();
    void    Font( short f );

    short   FontSize();
    void    FontSize( short s );

    short   FontColor();
    void    FontColor( short c );

    short   FontAttributes();
    void    FontAttributes( short a );
};
```


1.2.2.19 Gui::CreateBinding

Three types of bindings can be distinguished: Bindings to interpreter functions, bindings to bu
in functions and bindings to external callback function. A binding is created and assigned to
object. It is triggered by events on the object, that contains the binding.
Define a binding to an interpreter function:

```
Binding *Gui::CreateBinding( const char *name,
                             const char *inter_func,
                             const PrObject *po,
                             const PrItem *pi,
                             const GuiWindow *gui_win,
                             EVENT_TYPE ev, long ev_value,
                             DatabaseObject *db = NULL );
```

Define a binding to an external callback function:

```
Binding *Gui::CreateBinding( const char *name,
                             const char *callback_obj,
                             const char *callback_func,
                             EVENT_TYPE ev, long ev_value,
                             DatabaseObject *db = NULL );
```

Define a binding to an internal function:

```
Binding *Gui::CreateBinding( const char *name,
                             BINDING_FUNCTION bf,
                             const PrObject *po,
                             const PrItem *pi,
                             const GuiWindow *gui_win,
                             EVENT_TYPE ev, long ev_value,
                             DatabaseObject *db = NULL );
```

Create a binding as a copy of an existing binding:

```
Binding *Gui::CreateBinding( Binding *copy_bi,
                             DatabaseObject *db = NULL );
```

- **name**
  A binding can have a (unique) name to allow an identification.

- **inter_func**
  Name of the interpreter function.

- **callback_obj**
  Name of the external callback object. This mechanism is discussed in detail in section 1.

- **callback_func**
  Name of the external callback method of *callback_obj*. This mechanism is discussed in de
  in section 1.3.8.

- **po**
  Destination object, on which the function is executed. *po* contains its reference. Only one of the parameters *po* or *pi* can be set to *NULL*, when using internal functions. External and most interpreter functions do not need both parameters.

- **pi**
  Destination item, on which the function is executed. *pi* contains this reference.

- **gui_win**
  Destination window, if the function is executed on a window.

- **ev_type**
  ev_type contains the type of the event, on which the corresponding function is activated. Refer 1.3.2 for details.

- **value**
  Most events have additional values, specifying the events in detail (e.g., a mouse event sets the value to the pressed mouse button). *value* defines the requiered value to start the function or contains -1, if the value should be ignored.

- **bind_func**
  This parameter contains the identification for an internal function. These functions are called when the trigger conditions are fulfilled. Refer 1.3.9 for details.

- **db**
  Database handle, if the object is not stored in the default database.

- **copy_bi**
  template binding as copy source

1.2.2.20 Gui::DeleteBinding

```
void Gui::DeleteBinding( Binding *bind = NULL );
```

This operation removes a binding from this gui. The binding object is removed from memory and database.

- **bind**
  pointer to binding object, *NULL*, if all bindings should be removed

1.2.2.21 Gui::FirstBinding

```
Binding *Gui::FirstBinding( BINDING_FUNCTION bf = -1 );
```

FirstBinding returns the first binding of the object.


```
    short   FontAttributes();
    void    FontAttributes( short a );
};
```

- **String**
  string value, displayed in the symbol

- **Selected**
  Selected controls the additional state value of the corresponding symbol: If it is not zero, symbol is shown in selected state. Not every symbol interprets this component.

- **Cattr**
  font attributes, see above for details

- **StringAttributes**
  If no default font is selected in *Cattr*, this component contains a font attribute for every si character. The attribute at index *x* matches the character in *String* at the same position. not possible to select different fonts or sizes for characters of the same string.

- **EditAttributes**
  Attributes, used for interactively inserted characters. This component is examined, onl no default attributes are specified.

- **MaxLen**
  max. input size for string

- **CrsrPos**
  Input- and output value: index of the current cursor position

- **HighlightSymbol**
  One symbol in a string can be highlighted, using the underline-attribute even if all cha ters use the same attributes. The intention of this is to allow the highlightning of hotk characters in buttons. If no such symbol exists, this value should be -1.

- **RelXCrsrPos**
  Offset of cursor position, relativ to symbol (given in *WeltKoord.*

The font attributes are explained in 1.3.14.2.

1.3.14.4 SymbolShortAttr

This class represents a symbol for a short value, displayed and edited in string representatior created object is initialized with reasonable values as font attributes.

```
class SymbolStringAttr {

    char            string[ MAX_INPUT_LEN ];
    short           selected;
    CharAttr        cattr;
    unsigned short  string_attributes[ MAX_INPUT_LEN ],
                    edit_attributes,
                    max_len;
    signed   short  crsr_pos,
                    highlight_symbol,
                    rel_x_crsr_pos;
public:

    SymbolStringAttr( const char *string = NULL, short hls = -1 );
    ~SymbolStringAttr(){}

    short   Selected();
    void    Selected( short sel );

    char    *String();
    void    String( const char *str );

    ushort  *StringAttributes();
    void    StringAttributes( ushort *attr );

    short   HighlightSymbol();
    void    HighlightSymbol( short hls );

    ushort  EditAttributes();
    void    EditAttributes( ushort ea );

    ushort  MaxLen();
    void    MaxLen( ushort ml );

    short   CrsrPos();
    void    CrsrPos( short cp );

    short   RelXCrsrPos();
    void    RelXCrsrPos( short rel );

    short   Font();
    void    Font( short f );

    short   FontSize();
    void    FontSize( short s );

    short   FontColor();
    void    FontColor( short c );
```


- **bf**
  bf specifies the binding type: If $bf \neq -1$, all bindings are examines, else only binding the given type are used.

- **Return**
  pointer to binding object, or *NULL*, if no binding (of the given type *bf*) is assigned.

1.2.2.22 Gui::NextBinding

```
Binding *Gui::NextBinding( Binding *bind,
                           BINDING_FUNCTION bf = -1 );
```

NextBinding returns the next binding of this object.

- **bind**
  previous binding

- **bf**
  bf specifies the binding type: If $bf \neq -1$, all bindings are examines, else only binding the given type are used.

- **Return**
  pointer to the next binding object, or *NULL*, if no binding is assigned

1.2.2.23 Gui::FindBinding

```
Binding *Gui::FindBinding( const char *name );
```

FindBinding returns the binding with the given name or *NULL*, if no such binding exists on object.

- **name**
  name of the binding

- **Return**
  pointer to the binding object, or *NULL*, if no binding found

1.2.2.24 Gui::SetName

```
void Gui::SetName( const char *name );
```

SetName sets a new name to the gui.

- **name**
  new gui name

1.2.2.25 Gui::GetName

```
char *Gui::GetName( char *name = NULL );
```

GetName returns the gui's name.

1.2.2.26 Gui::Update

```
void Gui::Update();
```

Update updates or creates the complete gui with all components in the database.

1.2.2.27 Iterators

For some operations of an application it is necessary to access every component of the gui. For this purpose three iterator macros are defined. *GUI_FORALL_PO* is a for-loop, which determines every presentation object in the gui. *GUI_FORALL_WIN* determines every window of the gui. The last macro examins all bindings of the gui object.

```
#define GUI_FORALL_PO(gui,po)                \
    for( po = (gui)->FirstPrObject(); po;    \
         po = (gui)->NextPrObject( po ))

#define GUI_FORALL_GUIWIN(gui,win)           \
    for( win = (gui)->FirstGuiWindow(); win; \
         win = (gui)->NextGuiWindow( win ))

#define GUI_FORALL_BINDING(gui,bi)           \
    for( bi = (gui)->FirstBinding(); bi;     \
    bi = (gui)->NextBinding( bi ))
```



```
    void    Color( short c );

    short   Attributes();
    void    Attributes( short a );
};
```

- **Font** The following fonts are available:
  - **F_FIXED_SYSTEM**: fast system font for menues etc., probably not sizable
  - **F_PROP_SYSTEM**: same thing but proportional
  - **F_BOOK**: bookstyle for long text, sizable proportional with serifes
  - **F_TYPEWRITER**: a typewriter like style
  - **F_NOTE**: sizable proportional font without serifes
- **Size**
  Font size, given in points (current version uses pixel size for DJGPP).
- **Attributes**
  Or-combination of the listed font attributes:
  - **FONT_BOLD**: boldface font
  - **FONT_ITALIC**: italic style font
  - **FONT_UNDERLINED**: underlined font
  - **FONT_STRIKEOUT**: not available for DJGPP in the current version

  Some attributes are predefined for special solutions:
  - **FONT_NO_DEFAULT**: This is used only for string symbols: There is no default
    for all characters, instead, every character has it's own style. See 1.3.14.18 for detai
  - **FONT_DEFAULT**: no additional attributes
  - **FONT_MENU_DEFAULT**: no additional attributes
- **Color**
  Font color, see 1.3.13

1.3.14.3 SymbolStringAttr

This class represents a string symbol with individual attributes for every character. A crea
object is initialized with reasonable values as font attributes.

1.3.13 Colours

```
typedef enum{   WHITE           = WEISS,
                YELLOW          = GELB,
                VIOLET          = VIOLETT,
                RED             = ROT,
                CYAN            = TUERKIS,
                GREEN           = GRUEN,
                BLUE            = BLAU,
                BLACK           = SCHWARZ,
                LIGHT_WHITE     = HELLWEISS,
                LIGHT_YELLOW    = HELLGELB,
                LIGHT_VIOLET    = HELLVIOLETT,
                LIGHT_RED       = HELLROT,
                LIGHT_CYAN      = HELLTUERKIS,
                LIGHT_GREEN     = HELLGRUEN,
                LIGHT_BLUE      = HELLBLAU,
                GRAY            = GRAU } MMS_COLOURS;
```

1.3.14 Predefined Symbols

Since the basic service does not allow the interactive symbol manipulation in the current version, some modifyable symbols are predefined. This section describes the state values of all predefined symbols. First, some common structures for fonts are explained.

1.3.14.1 Predefined Symbol Attributes

1.3.14.2 CharAttr

This class describes font, size and attributes of a string symbol.

```
class CharAttr {
    unsigned short  font,
                    size,
                    color,
                    attributes;
public:
    CharAttr( short f, short s, short c, short a );
    CharAttr();
    ~CharAttr(){}

    short   Font();
    void    Font( short f );

    short   Size();
    void    Size( short s );

    short   Color();
```


1.2.3 GuiWindow

The *GuiWindow* is the window class. It is used to keep the attributes and methods for all M windows. Windows are created using the gui method *CreateWindow*.

1.2.3.1 GuiWindow::GuiWindow

```
GuiWindow::GuiWindow( const char *name, const WIN_ATTRIB *wa,
                      DatabaseObject *db = NULL );
GuiWindow::GuiWindow( long db_id,
                      DatabaseObject *db = NULL );
GuiWindow::GuiWindow( GuiWindow *copy_guiw,
                      DatabaseObject *db = NULL );
```

The gui window is created with a given unique database identification to reload an existing from the database or to create one with a special id. To create a new window with an automat assigned id, the first constructor should be used. To assure, that windows are assigned to g windows must created using the gui method *CreateWindow*.

- **name**
  optional window name or empty string
- **wa**
  initial window attributes, for details see 1.3.12.
- **db_id**
  unique database id
- **db**
  database handle, if object is not created in the default database
- **copy_guiw**
  template window

1.2.3.2 GuiWindow::~GuiWindow

```
GuiWindow::~GuiWindow();
```

The constructor removes the window from memory, not from database. **A window object sho only be destroyed using the gui method *DeleteWindow***. This ensures, that no pending re ences to this window object are left in a gui object.

1.2.3.3 GuiWindow::Metatyp

```
static long GuiWindow::Metatyp();
```

This operation returns a unique metatyp handle for the window class itself. The handle is valid for a single run of an application. **In an other run, the value can be different**.

1.2.3.4 GuiWindow::CreateBinding

Three types of bindings can be distinguished: Bindings to interpreter functions, bindings to built-in functions and bindings to external callback function. A binding is created and assigned to the object. It is triggered by events on the object, that contains the binding.
Define a binding to an interpreter function:

```
Binding *GuiWindow::CreateBinding( const char *name,
                            const char *inter_func,
                            const PrObject *po,
                            const PrItem *pi,
                            const GuiWindow *gui_win,
                            EVENT_TYPE ev, long ev_value,
                            DatabaseObject *db = NULL );
```

Define a binding to an external callback function:

```
Binding *GuiWindow::CreateBinding( const char *name,
                            const char *callback_obj,
                            const char *callback_func,
                            EVENT_TYPE ev, long ev_value,
                            DatabaseObject *db = NULL );
```

Define a binding to an internal function:

```
Binding *GuiWindow::CreateBinding( const char *name,
                            BINDING_FUNCTION bf,
                            const PrObject *po,
                            const PrItem *pi,
                            const GuiWindow *gui_win,
                            EVENT_TYPE ev, long ev_value,
                            DatabaseObject *db = NULL );
```

Create a binding as a copy of an existing binding:

```
Binding *Gui::CreateBinding( Binding *copy_bi,
                            DatabaseObject *db = NULL );
```

- **name**
  A binding can have a (unique) name to allow an identification.
- **inter_func**
  Name of the interpreter function.
- **callback_obj**
  Name of the external callback object. This mechanism is discussed in detail in section 1.3.8.
- **callback_func**
  Name of the external callback method of *callback_obj*. This mechanism is discussed in detail in section 1.3.8.

- **pos**
  window position, given in screen coordinates
- **type**
  window type:
  - **WINDOW_TYPE_EMPTY**: window frame and title are not visible
  - **WINDOW_TYPE_BORDER**: window has a small frame, but not title
  - **WINDOW_TYPE_SMALL_FRAME**: window has a small frame and a title
  - **WINDOW_TYPE_BIG_FRAME**: window has a big frame and a title
- **frame_c**
  frame color, if window has a frame
- **back_c**
  background color
- **title_c**
  color of the title backgound
- **text_c**
  color of the title text string, if the window has a title
- **text**
  text string of the window title, if the window contains a title
- **vis**
  visibility of the window: If *vis* not zero, the window is visible.

The following values are only return values. They can't be set by an application.

- **used_dx,used_dy**
  size of the window, available for an application (window size minus frame and title)
- **match_left, match_bottom**
  offset of a plane, assigned to the window, in the lower left window corner
- **match_right, match_top**
  visible size of a plane

- **GUI_EXEC_EXECUTE**
  Events are generated, the symbol manipulation is controlled by the design of the corresponding symbol. E.g., text fields can be edited, etc. This mode is used during the execution of a gui.

- **GUI_EXEC_MODAL**
  same as GUI_EXEC_EXECUTE, but the object is handled in a modal way (no other objects receive events until this flag is reset). This flag is not implemented now.

1.3.11 RPOSITION

RPOSITION is a base type, containing position and size of a window.

```
typedef struct {
    short        left;
    short        top;
    short        right;
    short        bottom;
} RPOSITION;
```

- **left, top**
  upper left corner of the window relative to the main window

- **right, bottom**
  lower right corner of the window relative to the main window

1.3.12 WIN_ATTRIB

WIN_ATTRIB contains all attributes of a window.

```
typedef struct {
    RPOSITION   pos;
    short       type;
    short       frame_c;
    short       back_c;
    short       title_c;
    short       text_c;
    char        title[128];
    boolean     vis;
    short       used_dx;
    short       used_dy;
    WeltKoord   match_left;
    WeltKoord   match_bottom;
    WeltKoord   match_right;
    WeltKoord   match_top;
    } WIN_ATTRIB;
```


- **po**
  Destination object, on which the function is executed. *po* contains its reference. Only
  of the parameters *po* or *pi* can be set to *NULL*, when using internal functions. External
  most interpreter functions do not need both parameters.

- **pi**
  Destination item, on which the function is executed. *pi* contains this reference.

- **gui_win**
  Destination window, if the function is executed on a window.

- **ev_type**
  ev_type contains the type of the event, on which the corresponding function is activa
  Refer 1.3.2 for details.

- **value**
  Most events have additional values, specifying the events in detail (e.g., a mouse event
  the value to the pressed mouse button). *value* defines the requiered value to start the func
  or contains −1, if the value should be ignored.

- **bind_func**
  This parameter contains the identification for an internal function. These functions are ca
  when the trigger conditions are fulfilled. Refer 1.3.9 for details.

- **db**
  Database handle, if the object is not stored in the default database.

- **copy_bi**
  template binding as copy source

1.2.3.5 GuiWindow::DeleteBinding

```
void GuiWindow::DeleteBinding( Binding *bind = NULL );
```

This operation removes a binding from this gui window. The binding object is removed f
memory and database.

- **bind**
  pointer to binding object, or *NULL*, if all bindings to this object should be removed.

1.2.3.6 GuiWindow::FirstBinding

```
Binding *GuiWindow::FirstBinding( BINDING_FUNCTION bf = -1 );
```

FirstBinding returns the first binding of the object.

- **bf**
  bf specifies the binding type: If $bf \neq -1$, all bindings are examines, else only bindings of the given type are used.

- **Return**
  pointer to binding object, or *NULL*, if no binding (of the given type *bf*) is assigned.

1.2.3.7 GuiWindow::NextBinding

```
Binding *GuiWindow::NextBinding( Binding *bind,
                                 BINDING_FUNCTION bf = -1 );
```

NextBinding returns the next binding of this object.

- **bind**
  previous binding

- **bf**
  bf specifies the binding type: If $bf \neq -1$, all bindings are examines, else only bindings of the given type are used.

- **Return**
  pointer to the next binding object, or *NULL*, if no binding is assigned

1.2.3.8 GuiWindow::FindBinding

```
Binding *GuiWindow::FindBinding( const char *name );
```

FindBinding returns the binding with the given name or *NULL*, if no such binding exists on this object.

- **name**
  name of the binding

- **Return**
  pointer to the binding object, or *NULL*, if no binding found

1.2.3.9 GuiWindow::Execute

```
boolean GuiWindow::Execute( GUI_EXEC_FLAG flag = GUI_EXEC_EXECUTE );
```

This operation is used to switch the event generation for a given window on or off. After the (re-)creation, the window is not supplied with events, until this method is called.

- **flag**
  Specifies, which kind of event generation is used, refer 1.3.10 for details.

- **BIND_INTERPRETER**
  binding is linked to an interpreted function

- **BIND_SUBOBJECT**
  The destination object is treated as a subobject of the caller (e.g.: submasks). A bind linked to such a function, has to be created with the presentation object parameter not e to *NULL*. When executing this function, first, the state of the subobject is saved and sec the subobject is shown and monitored.

- **BIND_HIDE_SUBOBJECT**
  Hide subobject, bound with *BIND_SUBOBJECT*: The subobject uses this binding to sw back to the main object.

- **BIND_HIDE_SUBOBJECT_AND_RESTORE**
  Hide subobject, bound with *BIND_SUBOBJECT* and restore original state of the subob which is saved during the execution of *BIND_SUBOBJECT*.

- **BIND_CHANGE_FOCUS**
  Change the keyboard focus to the destination item.

- **BIND_DEFAULT_EVENT**
  Delivers all events to the standard event handler of the PrObject. This binding should be on all presentation objects.

- **BIND_HIDE_MENU_SUBOBJECT**
  Only used for menu objects: The same functionality as *BIND_HIDE_SUBOBJECT*.

- **BIND_EMBED**
  The refered object is a embedded into the referring PrObject. It's position is defined rela to the container object. BIND_EMBED is a special version of BIND_SUBOBJECT.

- **BIND_SIMULATE_EVENT_OK**
  BIND_SIMULATE_EVENT_CANCEL
  These bindings can be linked to buttons to simulate EVENT_OK or EVENT_CANC events. This mechanism allows the easy recognition of events on ok or cancel buttons.

1.3.10 GUI_EXEC_FLAG

GUI_EXEC_FLAG controls the generation of events on the specified object.

- **GUI_EXEC_NONE**
  No events are generated for this object.

- **GUI_EXEC_MONITOR**
  Events are generated for the object, but not executed. This means, symbols are not modi on user interactions. This mode is intended for a construction of user interfaces or a comp application control on events.

```
            SetCallbackStatus( "EventHandler", CBS_INACTIVE );
            }

    boolean EventHandler( const EVENT *ev );
};

/*
    event catching
*/
boolean HandleCallback::EventHandler( const EVENT *ev ) {

    // do some event handling ....
    return( TRUE );
}

/*
    create callback handler
*/
HandleCallback *hcb = new HandleCallback( "Callback1" );
```

To use this object as a callback, a binding object with the name of the GuiCallback-object has to be created on a gui-object:

```
    pi->CreateBinding( "Binding1", "Callback1", "EventHandler",
                       EVENT_NEW_VALUE, -1 );
```

These bindings are restored after a reload of the gui. Callbacks are executed, when the linked application object (here: hcb) is restored in memory. This is under the control of the application. It is possible to use the gui, even if the callback objects are not reloaded. It is important to disable the callback method before destroying the callback object (see destructor).

1.3.9 BINDING_FUNCTION

BINDING_FUNCTION is an enumeration type, describing an internal MMS function.

- **BIND_SHOW**
  The binding is linked to an internal function, which shows the destination object when executed.
- **BIND_HIDE**
  The binding is linked to an internal function, which hides the destination object when executed.
- **BIND_DELETE**
  not implemented in the current version
- **BIND_CALLBACK**
  binding is linked to an external function

1.2.3.10 GuiWindow::Show

```
void GuiWindow::Show( boolean flag = TRUE );
```

Show shows or hides a window.

- **flag**
  Specifies, whether the window should be shown or hidden. If the window is not shown, event generation for this window is switched off. It will **not** be switched on automaticly a a redisplay.

1.2.3.11 GuiWindow::SetPosition

```
void GuiWindow::SetPosition( const POSITION *pos );
```

SetPosition moves or resizes a window.

- **pos**
  new position or size and mode, refer 1.3.6 for details.

1.2.3.12 GuiWindow::GetPosition

```
void GuiWindow::GetPosition( POSITION *pos );
```

GetPosition reads the window size and position.

- **pos**
  position and size of the window, refer 1.3.6 for details.

1.2.3.13 GuiWindow::Set

```
void GuiWindow::Set( const WIN_ATTRIB *wa );
```

Set overwrites all window attributes.

- **wa**
  new attributes, refer 1.3.12 for details.

1.2.3.14 GuiWindow::Get

```
void GuiWindow::Get( WIN_ATTRIB *wa );
```

Get reads all window attributes.

- **wa**
  attributes, refer 1.3.12 for details.

1.2.3.15 GuiWindow::SetName

```
void GuiWindow::SetName( const char *name );
```

SetName sets a new name to the window.

- **name**
  new window name

1.2.3.16 GuiWindow::GetName

```
char *GuiWindow::GetName( char *name = NULL );
```

GetName returns the window name.

- **name**
  returned window name, can be a *NULL* reference

- **Return**
  returned window name

1.2.3.17 GuiWindow::SetPlane

```
void GuiWindow::SetPlane( const char *plane_name,
                          WeltKoord x, WeltKoord y,
                          WeltKoord w, short detail = 10 );
```

SetPlane assignes a plane to a window. The scaling factor is calculated. In the current version of the base service, only one plane can be assigned to one window. But one plane can be assigned to multiple (different) windows.

- **plane_name**
  plane to assign

- **x, y**
  coordinates of the origin in the plane, which is placed in the lower left corner of the window.

- **w**
  w is the width of the visible plane section, which should fit into the window. This value is used to calculate the scaling factor between window and plane.

- **detail**
  A plane can consist of symbols with a different level of detail. This parameter determines, which of these symbols are shown in the window. Since these parameter is not (now) used in the high-level MMS functions, the value should be high (>10) or omitted.

1.3.8.5 Macros

To ease the registration process, four macros are defined:

```
#define REGISTER_FUNCTION_PLAIN( funcName, status ) \
    RegisterCallback( #funcName, (status), CB_F_PLAIN, \
                      (TCbAdress)& funcName )
#define REGISTER_FUNCTION_EVENT( funcName, status ) \
    RegisterCallback( #funcName, (status), CB_F_EVENT, \
                      (TCbAdress)& funcName )
#define REGISTER_MEMBER_PLAIN( funcClass, funcName, status ) \
    RegisterCallback( #funcName, (status), CB_M_PLAIN, \
                      (TCbAdress)(TMCbPlain)& funcClass :: funcName
#define REGISTER_MEMBER_EVENT( funcClass, funcName, status ) \
    RegisterCallback( #funcName, (status), CB_M_EVENT, \
                      (TCbAdress)(TMCbEvent)& funcClass :: funcName
```

- **REGISTER_FUNCTION_PLAIN**
  Register pure C-function as callback handler, the function is not provided with the ev
  value

- **REGISTER_FUNCTION_EVENT**
  Register pure C-function as callback handler, the function is provided with the event valu

- **REGISTER_MEMBER_PLAIN**
  Register C++-member as callback handler, the function is not provided with the event va

- **REGISTER_MEMBER_EVENT**
  Register C++-member as callback handler, the function is provided with the event value

1.3.8.6 Example

Callbacks are used in an application, using the class *GuiCallback*. Events are directed to appl
tion defined classes, which are derived from the class *GuiCallback*. The following piece of c
shows an example application:

```
class HandleCallback : public GuiCallback {

    char    name[ 30 ];
public:

    HandleCallback( char *nn ) : GuiCallback( nn ) {
              strncpy( name, nn, sizeof( name ));
  REGISTER_MEMBER_EVENT( HandleCallback,
                                      EventHandler,
                                      CBS_ACTIVE );
  }
    ~HandleCallback(){
```

1.3.8.1 GuiCallback::GuiCallback

```
GuiCallback::GuiCallback( const char *name );
```

- **name**
  The name must be unique for all external callback handler objects, since it is used in the event handler (Binding class) to determine the callback receiving object.

1.3.8.2 GuiCallback::RegisterCallback

RegisterCallback is called to register either a function or member callback. This method is not explained, because four macros should be used to ease the access (see: 1.3.8.5).

```
short GuiCallback::RegisterCallback( TCB_CPNAME name,
                                     TCB_STATUS status,
                                     TCB_TYPE type,
                                     TCbAdress adress );
```

1.3.8.3 GuiCallback::GetCallbackStatus

Determine the current status of a callback handler.

```
TCB_STATUS GuiCallback::GetCallbackStatus( TCB_CPNAME name );
```

- **name**
  Unique callback function or member name
- **Return**
  Status:
  - **CBS_NOTINLIST**: *name* is not a registered callback
  - **CBS_INACTIVE**: The callback is not active.
  - **CBS_ACTIVE**: The callback is active and able to process events.

1.3.8.4 GuiCallback::SetCallbackStatus

Modify the current status of a callback handler.

```
void GuiCallback::SetCallbackStatus( TCB_CPNAME name,
                                     TCB_STATUS status );
```

- **name**
  Unique callback function or member name
- **status**
  New status value:
  - **CBS_INACTIVE**: The callback is not longer active.
  - **CBS_ACTIVE**: The callback is set in an active state, it able to process events.

1.2.3.18 GuiWindow::GetPlane

```
void GuiWindow::GetPlane( char *plane_name,
                          WeltKoord *x, WeltKoord *y,
                          WeltKoord *w, short *detail );
```

GetPlane reads the plane parameters for this window. Please refer 1.2.3.17 for a description of parameters.

1.2.3.19 GuiWindow::Update

```
void GuiWindow::Update();
```

Update updates or creates the window attributes in the database. This method is called b *Gui::Update*.

1.2.3.20 Iterators

For some operations of an application it is necessary to access every component of the wind For this purpose one iterator macro is defined. *GUIWINDOW_FORALL_BINDING* is a for-lo which determines every binding object in the window.

```
#define GUIWINDOW_FORALL_BINDING(guiw,bi) \
    for( bi = (guiw)->FirstBinding(); bi; \
         bi = (guiw)->NextBinding( bi ))
```

1.2.4 PrObject

The presentation object — or in it's short form the *PrObject* — is the main container class. It is used to define the behaviour of gui objects. A *PrObject* is created using the gui method *CreatePrObject.*

1.2.4.1 PrObject::PrObject

```
PrObject::PrObject( const Gui *gui, PR_TYPE pr_t,
                    const char *name,
                    DESIGN *design,
                    const char *plane_name,
                    long data_type_mt, boolean emtpy,
                    const char *back_symbol,
                    DatabaseObject *db = NULL );
PrObject::PrObject( long id,
                    DatabaseObject *db = NULL );
PrObject::PrObject( PrObject *copy_po,
                    DatabaseObject *db = NULL );
```

The object is created with a given unique database identification to reload an existing object from the database. To create a new object with an automaticly assigned id, the first constructor should be used. To assure, that presentation objects are assigned to guis, they must be created using the gui method *CreatePrObject.* The paramaters are explained in 1.2.2.9. **A presentation object should not be created without the gui methods.**

1.2.4.2 PrObject::~PrObject

```
PrObject::~PrObject();
```

This destructor removes the object from memory. To assure a correct reference handling, presentation objects should only be removed using the gui method *DeletePrObject.*

1.2.4.3 PrObject::Metatyp

```
static long PrObject::Metatyp();
```

This operation returns a unique metatyp handle for this class itself. The handle is only valid for a single run of an application. **In an other run, the value can be different**.

1.2.4.4 PrObject::CreatePrItem

```
PrItem *PrObject::CreatePrItem( const char *name,
                                DESIGN *design,
                                const char *symbol_name,
                                const void *def_attr,
```


- **pos**
  position, size and interaction mode, refer 1.3.6 for details

- **orient**
  orientation of a presentation object:

  - **O_LEFT**: The items in the object are placed from right to left (f.e. menu).
  - **O_RIGHT**: The items in the object are placed from left to right.
  - **O_UP**: The items in the object are placed from down to up.
  - **O_DOWN**: The items in the object are placed from up to down.
  - **O_CENTER**: An item is centered (only for internal use).
  - **O_ALT_START_DOWN**: Hierarchical menus are created alternating: The first hie
    chical level with *O_DOWN*, the next with *O_RIGHT*, ...
  - **O_ALT_START_RIGHT**: Hierarchical menus are created alternating: The first hie
    chical level with *O_RIGHT*, the next with *O_DOWN*, ...

1.3.8 GuiCallback

Callback functions can be bound to events to catch gui-events in an application. This sec
describes the technique.

```
typedef enum { CBS_NOTINLIST, CBS_INACTIVE,
               CBS_ACTIVE, CBS_MAXNUM } TCB_STATUS;
typedef enum { CB_F_PLAIN, CB_F_EVENT, CB_M_PLAIN,
               CB_M_EVENT, CB_MAXNUM } TCB_TYPE;
extern "C"{
typedef short (*TCbPlain)( void );
typedef short (*TCbEvent)( const EVENT *);
}

typedef short (GuiCallback::*TMCbPlain)( void );
typedef short (GuiCallback::*TMCbEvent)( const EVENT * );

class GuiCallback {

public:
    GuiCallback( const char *name );
    ~GuiCallback();

    short      RegisterCallback( TCB_CPNAME name, TCB_STATUS status,
                                 TCB_TYPE type,   TCbAdress adress )
    TCB_STATUS GetCallbackStatus ( TCB_CPNAME Name ) const;
    TCB_STATUS SetCallbackStatus ( TCB_CPNAME Name,
                                   TCB_STATUS status );
};
```

1.3.6 POSITION

POSITION is a base type, containing position and size of a symbol.

```
typedef struct {
    WeltKoord    left;
    WeltKoord    top;
    WeltKoord    right;
    WeltKoord    bottom;
    short        mode;
} POSITION;
```

- **left, top**
  upper left corner of the symbol relative to the plane
- **right, bottom**
  lower right corner of the symbol relative to the plane
- **mode**
  mode specifies the interaction method with the service:
  - **MMS_NOT_IA**: symbol is placed non-interactive
  - **MMS_LEFT_IA**: position *left* is placed interactively, all other coordinates are taken from the structure.
  - **MMS_UP_IA**: position *up* is placed interactively, all other coordinates are taken from the structure.
  - **MMS_RIGHT_IA**: position *right* is placed interactively, all other coordinates are taken from the structure.
  - **MMS_DOWN_IA**: position *down* is placed interactively, all other coordinates are taken from the structure.
  - **MMS_LEFT_UP_IA**: position *left* and *up* are placed interactively, all other coordinates are taken from the structure.
  - **MMS_RIGHT_DOWN_IA**: position *right* and *down* are placed interactively, all other coordinates are taken from the structure.
  - **MMS_ALL_IA**: all positions are placed interactively, the structure contains only the start position

1.3.7 DESIGN

DESIGN is a structure, defining position, size and orientation of a man machine object.

```
typedef struct _DESIGN  DESIGN;

struct _DESIGN {
    POSITION        pos;
    short           orient;
    };
```



```
                                    short offset, short length,
                                    GUI_EXEC_FLAG flag );
PrItem *PrObject::CreatePrItem( PrItem *copy_pi );
```

This operation creates a new PrItem object and assignes it to this presentation object.

- **name**
  item name
- **design**
  position, size and creation mode for the item. See 1.3.7 for details.
- **symbol_name**
  name of the symbol, representing the item in the plane (screen)
- **def_attr**
  default attributes (values), used to display the corresponding symbol
- **offset,length**
  If the created item visualizes a component of a data structure of the presentation object, *o*
  contains the offset of the component from the startaddress of the structure and *length* cont
  the size of the component. If the presentation object does not visualize a data structure
  for simple pictures) or if the item is not part of the structure (f.e. a button), *offset* mus
  −1. In all cases, *length* contains the default size of the attributes, used for the *Set* and
  methods.
- **flag**
  flag describes the behaviour of the symbol, because the basic man machine service can
  provide this information in the current version. Refer 1.3.10 for details.
- **copy_pi**
  template item

1.2.4.5 PrObject::DeletePrItem

```
void PrObject::ItemPrDelete( PrItem *item );
```

This operation removes an item from this presentation object. The item is removed from mem
and database.

- **item**
  pointer to the item object, or *NULL* to remove all items

1.2.4.6 PrObject::FirstPrItem

```
PrItem *PrObject::FirstPrItem( F_MODIFIER fmod = F_ALL );
```

FirstPrItem returns the first item of the object. The item can be specified with a given attribute (see 1.3.4).

- **fmod**
  attribute
- **Return**
  pointer to the item object, or *NULL*, if no item is assigned

1.2.4.7 PrObject::NextPrItem

```
PrItem *PrObject::NextPrItem( PrItem *item,
                              F_MODIFIER f_mod = F_ALL );
```

NextPrItem returns the next item of this object. The item can be specified with a given attribute (see 1.3.4).

- **item**
  previous item
- **fmod**
  attribute
- **Return**
  pointer to the item object, or *NULL*, if no item is assigned

1.2.4.8 PrObject::CreateBinding

Three types of bindings can be distinguished: Bindings to interpreter functions, bindings to built-in functions and bindings to external callback function. A binding is created and assigned to the object. It is triggered by events on the object, that contains the binding.
Define a binding to an interpreter function:

```
Binding *PrObject::CreateBinding( const char *name,
                              const char *inter_func,
                              const PrObject *po,
                              const PrItem *pi,
                              const GuiWindow *gui_win,
                              EVENT_TYPE ev, long ev_value,
                              DatabaseObject *db = NULL );
```

Define a binding to an external callback function:

- **item**
  PrItem object, on which an event occured or *NULL*, if the event does not belong to an PrIt
- **object**
  PrObject object, on which an event occured or *NULL*, if the event does not belong to PrObject.
- **gui**
  Gui object, on which an event occured. This component is set for every event.
- **gui_win**
  GUIWindow object, on which an event occured. This component is set for every event.

1.3.4 F_MODIFIER

F_MODIFIER spezifies the behaviour of several find operations. It has one of the follwing val

- **F_ALL**: Perform operation on all objects.
- **F_MARKED**: Limit operation to marked objects.
- **F_NOT_MARKED**: Limit operation to not marked objects.
- **F_VISIBLE**: Limit operation to visible objects.
- **F_NOT_VISIBLE**: Limit operation to invisible objects.
- **F_EXECUTED**: Limit operation to executed objects.
- **F_NOT_EXECUTED**: Limit operation to not executed objects.

1.3.5 PR_TYPE

PR_TYPE is an enumeration type, specifying the type ob a presentation object. The follow types are supported:

- **PR_MENU**: menu object, consists of a column or row of items
- **PR_MASK**: mask object, consists of a number of items, representing a data structur class of the application
- **PR_TABLE**:table object, representing a set of objects or variables of a single class (typ
- **PR_PICTURE**: picture object, a set of items with no relations

- **EVENT_HOTKEY**: a hotkey was pressed (normal key plus ALT-key)
- **EVENT_NEW_VALUE**: An item has changed its value.
- **EVENT_OK, EVENT_CANCEL**: Ok or cancel button was pressed, this events can only be executed using a simulate event call or the apropriate bindings.

There are several other events defined but not used in the current version.

1.3.3 EVENT

This structure contains all information, send to an application after an event.

```
typedef struct _EVENT EVENT;

struct _EVENT {
    long        type;
    long        value;
    WeltKoord   mouse_x,
                mouse_y;
    PrItem      *item;
    PrObject    *object;
    Gui         *gui;
    GUIWindow   *gui_win;
    };
```

- **type**
  type of the event, this component contains a result of type EVENT_TYPE, but is coded as a long
- **value**
  value specifys the event more detailed, possible values depend on the event type:
  - **EVENT_KEYBOARD**: *value* contains the code of the pressed key.
  - **EVENT_HOTKEY**: *value* contains the code of the pressed key without any flag for the ALT-key.
  - **any mouse event**: *value* contains the code for the pressed mouse button:
    MOUSE_KEY_LEFT,
    MOUSE_KEY_MID and
    MOUSE_KEY_RIGHT.
  - **all other event**: *value* is undefined
- **mouse_x,mouse_y**
  position of the mouse pointer, given in coordintates relative to the window, in which the event occured


```
Binding *PrObject::CreateBinding( const char *name,
                                  const char *callback_obj,
                                  const char *callback_func,
                                  EVENT_TYPE ev, long ev_value,
                                  DatabaseObject *db = NULL );
```

Define a binding to an internal function:

```
Binding *PrObject::CreateBinding( const char *name,
                                  BINDING_FUNCTION bf,
                                  const PrObject *po,
                                  const PrItem *pi,
                                  const GuiWindow *gui_win,
                                  EVENT_TYPE ev, long ev_value,
                                  DatabaseObject *db = NULL );
```

Create a binding as a copy of an existing binding:

```
Binding *Gui::CreateBinding( Binding *copy_bi,
                             DatabaseObject *db = NULL );
```

- **name**
  A binding can have a (unique) name to allow an identification.
- **inter_func**
  Name of the interpreter function.
- **callback_obj**
  Name of the external callback object. This mechanism is discussed in detail in section 1.
- **callback_func**
  Name of the external callback method of *callback_obj*. This mechanism is discussed in de
  in section 1.3.8.
- **po**
  Destination object, on which the function is executed. *po* contains its reference. Only
  of the parameters *po* or *pi* can be set to *NULL*, when using internal functions. External
  most interpreter functions do not need both parameters.
- **pi**
  Destination item, on which the function is executed. *pi* contains this reference.
- **gui_win**
  Destination window, if the function is executed on a window.
- **ev_type**
  ev_type contains the type of the event, on which the corresponding function is activa
  Refer 1.3.2 for details.

- **value**
  Most events have additional values, specifying the events in detail (e.g., a mouse event sets the value to the pressed mouse button). *value* defines the requiered value to start the function or contains -1, if the value should be ignored.

- **bind_func**
  This parameter contains the identification for an internal function. These functions are called when the trigger conditions are fulfilled. Refer 1.3.9 for details.

- **db**
  Database handle, if the object is not stored in the default database.

- **copy_bi**
  template binding as copy source

1.2.4.9 PrObject::DeleteBinding

```
void PrObject::DeleteBinding( Binding *bind = NULL );
```

This operation removes a binding from this presentation object. The binding object is removed from memory and database.

- **bind**
  pointer to binding object, or *NULL*, if all bindings should be removed

1.2.4.10 PrObject::FirstBinding

```
Binding *PrObject::FirstBinding( BINDING_FUNCTION bf = -1 );
```

FirstBinding returns the first binding of the object.

- **bf**
  bf specifies the binding type: If $bf \neq -1$, all bindings are examines, else only bindings of the given type are used.

- **Return**
  pointer to binding object, or *NULL*, if no binding (of the given type *bf*) is assigned.

1.2.4.11 PrObject::NextBinding

```
Binding *PrObject::NextBinding( const Binding *bind,
                                BINDING_FUNCTION bf = -1 );
```

NextBinding returns the next binding of this object.

- **bind**
  previous binding

1.3 Predefined strucures and values

1.3.1 mms_sys_param

mms_sys_param is a small data structure, containing the needed settings for an initilization of basic man machine service.

```
typedef struct {
    coord    max_x;
    coord    max_y;
    coord    char_height;
    coord    char_width;
    boolean  debug;
    short    back_color;
    } mms_sys_param;
```

- **max_x, max_y**
  resolution (size) in pixel coordinates of the main application window

- **char_height, char_width**
  ignored

- **debug**
  If debugging is on (*debug* not 0), the basic service works in a verbose mode, dumping formation about the internal work on the main window (in the current version not work under Microsoft Windows).

- **back_color**
  Background color for main window

1.3.2 EVENT_TYPE

EVENT_TYPE contains the occured event. The follwing values are possible:

- **EVENT_NO_EVENT**: no event occured

- **EVENT_MOUSE_SINGLE**: single click on a mouse button

- **EVENT_MOUSE_DOUBLE**: double click on a mouse button

- **EVENT_KEYBOARD**: single key on the keyboard pressed

- **EVENT_MOUSE_RELEASE**: mouse button has been released, this events occures aft *EVENT_MOUSE_SINGLE* or *EVENT_MOUSE_DOUBLE*

- **EVENT_LEAVE_FOCUS**: an item object has lost the keyboard focus

- **EVENT_ENTER_FOCUS**: an item object has received the keyboard focus

1.2.7 Methods — an overview

In this table, columns represent different MMS components, while rows show possible methods on an object. If the interaction of both is not blank, the combination of both is allowed and the number shown there contains the chapter, where the operation is discussed.

	Gui		GuiWindow		PrObject		PrItem		Binding	
	Ext.	Ref.	Ext.	Ref.	Ext.	Ref.	Ext.	Ref.	Ext.	Ref.
Constructor		1.2.2.1		1.2.3.1		1.2.4.1		1.2.5.1		1.2.6.1
Destructor		1.2.2.2		1.2.3.2		1.2.4.2		1.2.5.2		1.2.6.2
Metatyp		1.2.2.3		1.2.3.3		1.2.4.3		1.2.5.3		1.2.6.3
SimulateEvent		1.2.2.4								
ChangeFocus		1.2.2.5								
Execute		1.2.2.6		1.2.3.9		1.2.4.13				1.2.6.4
Event		1.2.2.7								
Show		1.2.2.8		1.2.3.10		1.2.4.14		1.2.5.9		
Create	PrObject	1.2.2.9			PrItem	1.2.4.4				
	GuiWindow	1.2.2.14								
	Binding	1.2.2.19	Binding	1.2.3.4	Binding	1.2.4.8	Binding	1.2.5.4		
Delete	PrObject	1.2.2.10			PrItem	1.2.4.5				
	GuiWindow	1.2.2.15								
	Binding	1.2.2.20	Binding	1.2.3.5	Binding	1.2.4.9	Binding	1.2.5.5		
First	PrObject	1.2.2.11			PrItem	1.2.4.6				
	GuiWindow	1.2.2.16								
	Binding	1.2.2.21	Binding	1.2.3.6	Binding	1.2.4.10	Binding	1.2.5.6		
Next	PrObject	1.2.2.12			PrItem	1.2.4.7				
	GuiWindow	1.2.2.16								
	Binding	1.2.2.22	Binding	1.2.3.7	Binding	1.2.4.11	Binding	1.2.5.7		
Find	PrObject	1.2.2.13								
	GuiWindow	1.2.2.18								
	Binding	1.2.2.23	Binding	1.2.3.8	Binding	1.2.4.12	Binding	1.2.5.8		
Set				1.2.3.13		1.2.4.17		1.2.5.12		
			Position	1.2.3.11	Position	1.2.4.15	Position	1.2.5.10		
	Name	1.2.2.24	Name	1.2.3.15	Name	1.2.4.19	Name	1.2.5.14	Name	1.2.6.5
									PrObject	1.2.6.10
									PrItem	1.2.6.11
									GuiWindow	1.2.6.12
									Function	1.2.6.14
			Plane	1.2.3.17						
Get				1.2.3.14		1.2.4.18		1.2.5.13		
			Position	1.2.3.12	Position	1.2.4.16	Position	1.2.5.11		
	Name	1.2.2.25	Name	1.2.3.16	Name	1.2.4.20			Name	1.2.6.6
									PrObject	1.2.6.7
					Type	1.2.4.23			PrItem	1.2.6.8
					Gui	1.2.4.24			GuiWindow	1.2.6.9
					Focus	1.2.4.25			Function	1.2.6.13
					Metatyp	1.2.4.26	Metatyp	1.2.5.18		
			Plane	1.2.3.18						
Update		1.2.2.26		1.2.3.19		1.2.4.27		1.2.5.19		
Mark						1.2.4.21		1.2.5.16		
Marked						1.2.4.22		1.2.5.17		
Iterators		1.2.2.27		1.2.3.20		1.2.4.28		1.2.5.20		

- **bf**
  bf specifies the binding type: If $bf \neq -1$, all bindings are examines, else only binding the given type are used.

- **Return**
  pointer to the next binding object, or *NULL*, if no binding is assigned

1.2.4.12 PrObject::FindBinding

```
Binding *PrObject::FindBinding( const char *name );
```

FindBinding returns the binding with the given name or *NULL*, if no such binding exists on object.

- **name**
  name of the binding

- **Return**
  pointer to the binding object, or *NULL*, if no binding found

1.2.4.13 PrObject::Execute

```
void PrObject::Execute( GUI_EXEC_FLAG flag = GUI_EXEC_EXECUTE );
```

This operation is used to switch the event generation for a given presentation object on or After the (re-)creation, the object is not supplied with events, until this method is called.

- **flag**
  Specifies the event generation type for this object, refer 1.3.10 for details.

1.2.4.14 PrObject::Show

```
void PrObject::Show( boolean flag = TRUE );
```

Show shows or hides a presentation object.

- **flag**
  Specifies, whether the object should be shown or hidden. If the object is not shown, event generation for this object is switched off. It will **not** be switched on automaticly a a redisplay.

1.2.4.15 PrObject::SetPosition

```
void PrObject::SetPosition( const POSITION *pos );
```

SetPosition moves or resizes a presentation object.

- **pos**
  new position or size and mode, refer 1.3.6 for details.

1.2.4.16 PrObject::GetPosition

```
void PrObject::GetPosition( POSITION *pos );
```

GetPosition reads the object size and position.

- **pos**
  position and size of the object, refer 1.3.6 for details.

1.2.4.17 PrObject::Set

```
void PrObject::Set( const void *attributes );
```

Set overwrites the attributes for the object. This method is only avaluated for presentation objects of type PR_MENU, PR_MASK and PR_TABLE. All other objects don't represent a single data structure.

- **attributes**
  new attributes: every item, which represents a component of the entire data structure, is supplied with the corresponding part of the attributes using the method *PrItem::Set.*

1.2.4.18 PrObject::Get

```
void PrObject::Get( void *attributes );
```

Get reads the attributes for the entire object. This method is only avaluated for presentation objects of type PR_MENU, PR_MASK and PR_TABLE. All other objects don't represent a single data structure.

- **attributes**
  Every item, which represents a component of the entire data structure, fills out the corresponding part of the attributes using the method *PrItem::Get.*

1.2.4.19 PrObject::SetName

```
void PrObject::SetName( const char *name );
```

SetName sets a new name to the object.

- **name**
  new object name

1.2.6.13 Binding::GetFunction

```
BINDING_FUNCTION Binding::GetFunction( GuiCallback **cb );
```

- **cb**
  If the binding is linked to an external function, the address of the corresponding event c
  is returned. This mechanism is under construction and therefor not available now.

- **Return**
  If the binding is linked to an internal function, their code is returned On external functi
  BIND_FUNCTION is returned.

1.2.6.14 Binding::SetFunction

```
void Binding::SetFunction( BINDING_FUNCTION bf );
void Binding::SetFunction( const char *inter_func );
void Binding::SetFunction( const GuiCallback *cb );
```

- **bf**
  Function code for internal binding

- **inter_func**
  Name of an (existing) interpreted function

- **cb**
  If the binding is linked to an external function, the address of the corresponding event c
  is returned.

1.2.6.7 Binding::GetPrObject

```
PrObject *Binding::GetPrObject();
```

- **Return**
  a pointer to the presentation object, which is passed as a parameter to the executing function or *NULL*, if no presentation object specified

1.2.6.8 Binding::GetPrItem

```
PrItem *Binding::GetPrItem();
```

- **Return**
  a pointer to the item object, which is passed as a parameter to the executing function or *NULL*, if no item specified

1.2.6.9 Binding::GetGuiWindow

```
GuiWindow *Binding::GetGuiWindow();
```

- **Return**
  a pointer to the window object, which is passed as a parameter to the executing function or *NULL*, if no window specified

1.2.6.10 Binding::SetPrObject

```
void Binding::SetPrObject( const PrObject *po );
```

- **po**
  a pointer to the presentation object, which is passed as a parameter to the executing function or *NULL*, if no presentation object specified

1.2.6.11 Binding::SetPrItem

```
void Binding::SetPrItem( const PrItem *pi );
```

- **pi**
  a pointer to the item object, which is passed as a parameter to the executing function or *NULL*, if no item specified

1.2.6.12 Binding::SetGuiWindow

```
void Binding::SetGuiWindow( const GuiWindow *win );
```

- **win**
  a pointer to the window object, which is passed as a parameter to the executing function or *NULL*, if no window specified

1.2.4.20 PrObject::GetName

```
char *PrObject::GetName( char *name = NULL );
```

GetName returns the objects name.

- **name**
  returned object name, can be a *NULL* reference

- **Return**
  returned object name

1.2.4.21 PrObject::Mark

```
void PrObject::Mark( boolean flag = TRUE );
```

Mark places a mark symbol around the object.

- **flag**
  determines, whether the mark is shown or removed

1.2.4.22 PrObject::Marked

```
boolean PrObject::Marked();
```

Marked returns *true*, if the object is marked.

1.2.4.23 PrObject::GetType

```
PR_TYPE PrObject::GetType();
```

GetType returns the type of the object.

- **Return**
  the type of the object (mask, menu etc.), refer 1.3.5 for details.

1.2.4.24 PrObject::GetGui

```
Gui *PrObject::GetGui();
```

GetGui returns the reference of the gui to which this object is assigned.

- **Return**
  reference to the gui

1.2.4.25 PrObject::GetFocus

```
PrItem *PrObject::GetFocus();
```

GetFocus returns the reference to the PrItem, which has the keyboard focus.

- **Return**
  Item with keyboard focus or *NULL*, if no item of this object has the focus. There can be only one item in all guis with keyboard focus at one point of time.

1.2.4.26 PrObject::GetMetatyp

```
long PrObject::GetMetatyp();
```

GetMetatyp returns the metatyp of of the correspond data structure or class, which is represented using this object. *GetMetatyp* is -1 for pictures.

- **Return**
  Unique metatyp handle or -1, if this object is not representing one data structure or class.

1.2.4.27 PrObject::Update

```
void PrObject::Update();
```

Update updates or creates the objects attributes in the database. This method is called by a *Gui::Update*.

1.2.4.28 Iterators

For some operations of an application it is necessary to access every component of the presentation object. For this purpose two iterator macros are defined. *PR_OBJECT_FORALL_PR_ITEM* is a for-loop, which determines every item object in the PrObject. *PR_OBJECT_FORALL_BINDING* determines every binding of the object.

```
#define PR_OBJECT_FORALL_PR_ITEM(po,pi) \
    for( pi = (po)->FirstPrItem(); pi;    \
         pi = (po)->NextPrItem( pi ))

#define PR_OBJECT_FORALL_BINDING(po,bi) \
    for( bi = (po)->FirstBinding(); bi; \
         bi = (po)->NextBinding( bi ))
```


1.2.6.3 Binding::Metatyp

```
static long Binding::Metatyp();
```

This operation returns a unique metatyp handle for this class itself. The handle is only valid f
single run of an application. **In an other run, the value can be different**.

1.2.6.4 Binding::Execute

```
boolean Binding::Execute( const EVENT *event );
```

The *Execute* method is in most cases internaly called from the event manager to ask the binc
object to execute the linked function if the conditions fit with the given event. The binding ob
examins, whether these conditions are fulfiled and executes the function (or not). This met
should only be called by an application to force the execution for a single binding. A be
solution is to simulate an event (*Gui::SimulateEvent*). In this case, the gui (or exactly: the ev
manager) calles the *Execute* method on all bindings of all components of the own gui.

- **event**
  reference to the real or simulated event, refer 1.3.3 for details.

- **Return**
  TRUE, if other bindings on the corresponding object should be started, or *FALSE*, if
  execution of binding should be stoped for this event. Using *FALSE* as a return-value, a
  defined callback is able to stop binding execution after an error.

1.2.6.5 Binding::SetName

```
void Binding::SetName( const char *name );
```

SetName sets a new name to the binding.

- **name**
  new binding name

1.2.6.6 Binding::GetName

```
char *Binding::GetName( char *name = NULL );
```

GetName returns the bindings name.

- **name**
  returned binding name, can be a *NULL* reference

- **Return**
  returned binding name

1.2.6 Binding

Bindings define the behaviour of a gui on user interactions. They are used to link gui-events to functions, where links to internal functions, links to external callback functions and links to interpreter functions are supported.

1.2.6.1 Binding::Binding

Binding to interpreter functions:

```
Binding::Binding( const char *nn, const char *inter_func,
                  const PrObject *po, const PrItem *pi,
                  const GuiWindow *gui_win,
                  EVENT_TYPE ev, long ev_value,
                  DatabaseObject *db = NULL );
Binding::Binding( Binding *copy_bi, DatabaseObject *db = NULL );
```

Binding to external callback functions:

```
Binding::Binding( const char *nn, const char *callback_obj,
                  const char *callback_func,
                  EVENT_TYPE ev, long ev_value,
                  DatabaseObject *db = NULL );
```

Binding to internal functions:

```
Binding::Binding( const char *nn, BINDING_FUNCTION bf,
                  const PrObject *po, const PrItem *pi,
                  const GuiWindow *gui_win,
                  EVENT_TYPE ev, long ev_value,
                  DatabaseObject *db = NULL );
```

Reload binding from database:

```
Binding::Binding( long id, DatabaseObject *db = NULL );
```

Create a binding as a copy of an existing binding:

```
Binding::Binding( Binding *copy_bi,
                  DatabaseObject *db = NULL );
```

No binding should be created directly. Instead, the *CreateBinding* method on the corresponding object should be called. The parameters are described there.

1.2.6.2 Binding::~Binding

```
Binding::~Binding();
```

The destructor removes a binding from memory. An application should not delete a binding object other than using the appropriate *DeleteBinding* method of the object, which contains the reference to this binding.

1.2.5 PrItem

A presentation object contains a set of items, which are displayd using symbols.

1.2.5.1 PrItem::PrItem

```
PrItem::PrItem( const PrObject *po, const char *name,
                DESIGN *design,
                const char *symbol_type, const void *attr,
                short offset, short len,
                GUI_EXEC_FLAG flag,
                DatabaseObject *db = NULL );
PrItem::PrItem( long id ,
                DatabaseObject *db = NULL );
```

The item is created with a given unique database identification to reload an existing gui from database. To create a new item with an automaticly assigned id, the first constructor should used. To assure, that items are assigned to presentation objects, they must be created using PrObject method *CreateItem*. The paramaters are explained in 1.2.4.4.

1.2.5.2 PrItem::~PrItem

```
PrItem::~PrItem();
```

This destructor removes the item from memory. To assure a correct reference handling, it should only be removed using the PrObject method *DeleteItem*.

1.2.5.3 PrItem::Metatyp

```
static long PrItem::Metatyp();
```

This operation returns a unique metatyp handle for this class itself. The handle is only valid f single run of an application. **In an other run, the value can be different**.

1.2.5.4 PrItem::CreateBinding

Three types of bindings can be distinguished: Bindings to interpreter functions, bindings to bu in functions and bindings to external callback function. A binding is created and assigned to object. It is triggered by events on the object, that contains the binding.
Define a binding to an interpreter function:

```
Binding *PrItem::CreateBinding( const char *name,
                                const char *inter_func,
                                const PrObject *po,
                                const PrItem *pi,
                                const GuiWindow *gui_win,
                                EVENT_TYPE ev, long ev_value,
                                DatabaseObject *db = NULL );
```

Define a binding to an external callback function:

```
Binding *PrItem::CreateBinding( const char *name,
                                const char *callback_obj,
                                const char *callback_func,
                                EVENT_TYPE ev, long ev_value,
                                DatabaseObject *db = NULL );
```

Define a binding to an internal function:

```
Binding *PrItem::CreateBinding( const char *name,
                                BINDING_FUNCTION bf,
                                const PrObject *po,
                                const PrItem *pi,
                                const GuiWindow *gui_win,
                                EVENT_TYPE ev, long ev_value,
                                DatabaseObject *db = NULL );
```

Create a binding as a copy of an existing binding:

```
Binding *Gui::CreateBinding( Binding *copy_bi,
                             DatabaseObject *db = NULL );
```

- **name**
  A binding can have a (unique) name to allow an identification.
- **inter_func**
  Name of the interpreter function.
- **callback_obj**
  Name of the external callback object. This mechanism is discussed in detail in section 1.3.8.
- **callback_func**
  Name of the external callback method of *callback_obj*. This mechanism is discussed in detail in section 1.3.8.
- **po**
  Destination object, on which the function is executed. *po* contains its reference. Only one of the parameters *po* or *pi* can be set to *NULL*, when using internal functions. External and most interpreter functions do not need both parameters.
- **pi**
  Destination item, on which the function is executed. *pi* contains this reference.
- **gui_win**
  Destination window, if the function is executed on a window.
- **ev_type**
  ev_type contains the type of the event, on which the corresponding function is activated. Refer 1.3.2 for details.

1.2.5.20 Iterators

For some operations of an application it is necessary to access every component of the it
For this purpose one iterator macro is defined. *PR_OBJECT_FORALL_PR_ITEM* is a for-lo
determines every binding of the item.

```
#define PR_ITEM_FORALL_BINDING(pi,bi)   \
    for( bi = (po)->FirstBinding(); bi; \
         bi = (po)->NextBinding( bi ))
```

1.2.5.15 PrItem::GetName

```
char *PrItem::GetName( char *name = NULL );
```

GetName returns the items name.

- **name**
  returned item name, can be a *NULL* reference

- **Return**
  returned item name

1.2.5.16 PrItem::Mark

```
void PrItem::Mark( boolean flag = TRUE );
```

Mark places a mark symbol around the item.

- **flag**
  determines, whether the mark is shown or removed

1.2.5.17 PrItem::Marked

```
boolean PrItem::Marked();
```

Marked returns *true*, if the item is marked.

1.2.5.18 PrItem::GetMetatyp

```
long PrItem::GetMetatyp();
```

GetMetatyp returns the metatyp handle of of the corresponding symbol.

- **Return**
  unique metatyp handle

1.2.5.19 PrItem::Update

```
void PrItem::Update();
```

Update updates or creates the item's attributes in the database. This method is called by a *PrObject::Update*.

- **value**
  Most events have additional values, specifying the events in detail (e.g., a mouse event
  the value to the pressed mouse button). *value* defines the requiered value to start the func
  or contains -1, if the value should be ignored.

- **bind_func**
  This parameter contains the identification for an internal function. These functions are ca
  when the trigger conditions are fulfilled. Refer 1.3.9 for details.

- **db**
  Database handle, if the object is not stored in the default database.

- **copy_bi**
  template binding as copy source

1.2.5.5 PrItem::DeleteBinding

```
void PrItem::DeleteBinding( Binding *bind );
```

This operation removes a binding from this item object. The binding object is removed f
memory and database.

- **bind**
  pointer to binding object

1.2.5.6 PrItem::FirstBinding

```
Binding *PrItem::FirstBinding( BINDING_FUNCTION bf = -1 );
```

FirstBinding returns the first binding of the object.

- **bf**
  bf specifies the binding type: If $bf \neq -1$, all bindings are examines, else only binding
  the given type are used.

- **Return**
  pointer to binding object, or *NULL*, if no binding (of the given type *bf*) is assigned.

1.2.5.7 PrItem::NextBinding

```
Binding *PrItem::NextBinding( const Binding *bind,
                              BINDING_FUNCTION bf = -1 );
```

NextBinding returns the next binding of this object.

- **bind**
  previous binding

- **bf**
  bf specifies the binding type: If $bf \neq -1$, all bindings are examines, else only bindings of the given type are used.

- **Return**
  pointer to the next binding object, or *NULL*, if no binding is assigned

1.2.5.8 PrItem::FindBinding

```
Binding *PrItem::FindBinding( const char *name );
```

FindBinding returns the binding with the given name or *NULL*, if no such binding exists on this object.

- **name**
  name of the binding

- **Return**
  pointer to the binding object, or *NULL*, if no binding found

1.2.5.9 PrItem::Show

```
void PrItem::Show( boolean flag = TRUE );
```

Show shows or hides an item. The show method on a presentation object overwrites the effects of a show on an item.

- **flag**
  Specifies, whether the item should be shown or hidden. If the item is not shown, the event generation for this item is switched off. It will be switched on automaticly after a redisplay, if the corresponding presentation object is monitored.

1.2.5.10 PrItem::SetPosition

```
void PrItem::SetPosition( const POSITION *pos );
```

SetPosition moves or resizes an item.

- **pos**
  new position or size and mode, refer 1.3.6 for details.

1.2.5.11 PrItem::GetPosition

```
void PrItem::GetPosition( POSITION *pos );
```

GetPosition reads the items size and position.

- **pos**
  position and size of the item, refer 1.3.6 for details.

1.2.5.12 PrItem::Set

```
void PrItem::Set( const void *attributes,
                  short len = PR_ITEM_DEF_SIZE  );
```

Set overwrites the attributes for the item. This method is defined for all types of items.

- **attributes**
  attributes contain the new values for the item. Their type has to match the symbol typ
  a subset of it. This means: *attributes* can contain only parts of the symbol states to ease
  manipulation. E.g., when using text fields to enter string values, in normal operation ther
  only a need to set a new string. It is not necessary to set all attributes, including font-na
  font-size, color, ... This allows the application developer to include only parts of the sym
  state into his own data-type. These are the steps to create a simple mask:

  - **Design phase**: Create a mask with all items. Set the default values on all items, us
    the entire symbol state.
  - **Application phase**: Using the mask does not require to set the entire state value. C
    these part are interesting which contain application specific data.

- **len**
  The length parameter is only given, if the size of the attributes differs from the given ler
  during the item's creation. The following values are possible:

  - **PR_ITEM_DEF_SIZE**: Set item value with length given in constructor. This is
    default behaviour, if no parameter is specified).
  - **PR_ITEM_DEF_SIZE_REC**: Set item value with default length **and** set presenta
    objects, appended with a binding as a subobject, to their values (recursive setting, e
    submask etc.).
  - **other non zero**: Set item value with attributes of desired length.

1.2.5.13 PrItem::Get

```
void PrItem::Get( void *attributes,
                  short len = PR_ITEM_DEF_SIZE );
```

Get reads the attributes for the given item. The parameters are described in 1.2.5.12 (previ
section).

1.2.5.14 PrItem::SetName

```
void PrItem::SetName( const char *name );
```

SetName sets a new name to the item.

- **name**
  new item name