IntuiObject

COLLABORATORS			
	TITLE : IntuiObject		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		September 19, 2022	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

IntuiObject

1.1 Implementation notes

```
                              The IntuiObject class
                    ----------------------------
                    ($Date: 1994/05/09 21:25:01 $)
```

The IntuiObject is the abstract root class that fulfills the task of making each encapsulated Intuition® object accessable for A++ classes in a uniform way.

All IntuiObject-derived classes have in common:

1. Their current state is reflected by means of Attribute Tags, which resemble Intuition® Taglists. Attributes can be read and written by the class user.

2. Each object is placed in the application's IntuiObjects tree where its position states its interdependencies to other IntuiObjects regarding lifetime and some special things that are introduced by derived classes.

3. They are capable of maintaining notification streams that transmit Attribute Tag changes to other IntuiObjects.

It provides methods to read and change these attributes. Each attribute change, may it be caused by the programmer calling these methods or by the user manipulating the Intuition® element, triggers a notification.


```
  ----------------------------------------------------------
```

-> Back to the root menu..

1.2 intuiobject_1

All IntuiObjects that are created during an application are added to a tree of IntuiObjects.
At the start of the application only the root of the tree exists, representing the default public screen, on which all windows will be opened that are not opened on a custom screen. This is explained in much more detail in the documentation of the ScreenC and WindowCV classes.

What is common for all IntuiObjects is, that they have one parent IntuiObject and may have as many child objects as necessary.
An IntuiObject is assigned to its parent at creation time and can never be reassigned later.

The constructor demands a parent IntuiObject and an AttrList object that will be copied to the IntuiObject Attribute Taglist.

Any IntuiObject derived class should have default values for all of its attributes that allow creating an object also with an empty tag list.
After creation, an IntuiObject is supposed to have all its attributes initialised to proper values so that a getAttribute() call results in a definite way.

IMPORTANT: IntuiObjects and derived class objects MUST be created with 'new'.
Never create IntuiObjects on the stack since they are deleted automatically!

A constructor looks like this:

```
IntuiObject *parent_intuiobject;
ITransponder *itransponder;
...
IntuiObject *myIntuiObject = new IntuiObject(
   IntuiObject *parent_intuiobject,
   AttrList( PGA_Top,1,PGA_Visible,2,TAG_END ) );

if (myIntuiObject && myIntuiObject->Ok())    // check object validity
{
   // work on the object

}
else delete myIntuiObject;
```

The destruction of an IntuiObject causes the deletion of all its child IntuiObjects. All IntuiObjects are assumed to be created on the runtime heap with the new operator and therefore must not be created on the stack!

At the end of an application the destruction of the tree root IntuiObject causes automatic deletion of all still not deleted IntuiObjects thus providing an elegant way of freeing resources.

To create an IntuiObject that immediately has the tree root as parent use OWNER_ROOT as first parameter.

1.3 intuiobject_2

```
There are two methods to read and write attribute tags for class users:

To read one attribute tag value use the method:
   IntuiObject *intuiobject;
   ULONG dataStore;
   ULONG returnValue = intuiobject->getAttribute(TAG_SPECIFIER,dataStore);

If the addressed object 'understood' the TAG_SPECIFIER, its value is written
to 'dataStore' and also being returned. In case the TAG_SPECIFIER was not
recognized by the object, both 'dataStore' and return value become zero.
Note, that a result of zero can also be the current tag value.

To set one or more attribute tags to new values use:

   intuiobject->setAttributes( AttrList& attrs );

Setting attribute tags always involves the notification mechanism.
```

1.4 intuiobject_3

```
                        The IntuiObject Notification Mechanisms
        ------------------------------------------

A considerable amount of most application's code manages how any
GUI element changes depending on another GUI element that has been changed
by the user.

Between IntuiObjects these dependencies can easily be expressed by Attribute
dependencies: you can tell an object's attribute that it shall adopt its
value from another object's attribute value. Each time the second object's
attribute value changes, this change extends to the first object.
Or, you listen in on one object's attribute changes and select these changes
you want having an effect on certain other objects.
The first method is called
             Attribute Constraints
             , the second one is named

             ITransponders
             .
```

1.5 intuiobject_3.1

```
                        Notification via Attribute Constraints
        ------------------------------------------

A very easy and elegant way to make an IntuiObject's attribute value change
depending on another IntuiObject' attribute is the usage of Constraints.

Instead of initialising the attribute value with a constant given in an
Attribute Tag List at the IntuiObject's creation, you can have it initialised
```

by another IntuiObject's attribute using the following scheme:

```
DerivedIntuiObject *source = new DerivedIntuiObject(owner,
      AttrList(GTSC_Top, 10, TAG_END) );

DerivedIntuiObject *dest = new DerivedIntuiObject(owner,
      AttrList( CONSTRAINT(PGA_Top, source, GTSC_Top), TAG_END) );
```

In this example the PGA_Top attribute of the second object will always follow the GTSC_Top attribute value of the first object.
Each time the GTSC_Top value changes the second object will receive the call

```
dest->setAttributes( AttrList(PGA_Top, gtsc_top_value, TAG_END) )
```

Note that the second object assumes the first one's GTSC_Top attribute to be initialised to a proper value either by naming it in the AttrList or default value.
You can use constraints together with ITransponders. But each attribute should only be refreshed one way or the other.

Circular constraint dependencies are allowed; the loop is broken before the 'setAttributes' call is invoked on the IntuiObject that triggered the notification.

If the first object' GTSC_Top attribute from the scheme above should also become depending on the the PGA_Top attribute only add the following:

```
source->setAttributes( AttrList(
   CONSTRAINT(GTSC_Top,dest,PGA_Top),
   TAG_END
   ) );
```

The first object's 'GTSC_Top' attribute is being initialized with the second one's 'PGA_Top'. Since 'PGA_Top has been initialized by 'GTSC_Top' before, the 'GTSC_Top' value still keeps the value from the first object's AttrList. Now, both IntuiObjects notify each other on each GTSC_Top/PGA_Top attribute change.

The IntuiObject Constraints Mechanism is by far one of the most powerful features in A++ since it can be aplied to virtually every IntuiObject-derived class.
The

ITransponder

serves the same task but is customizable to a higher degree.

1.6 intuiobject_3.2

The ITransponder Notification Mechanism

The ITranponder class serves as base class for notification forwarders that establish notification streams between IntuiObjects.
This notification simply is the virtual method call of an ITransponder object that basically forwards the notification to other IntuiObjects, which may adopt the new values to their own attribute values with eventual translation.

The class user can define specialized ITransponder classes to connect certain IntuiObjects by overloading the virtual method

```
virtual void ITransponder::sendNotification( AttrList& changedAttrs);
```

Every IntuiObject calls this method on its assigned ITransponder with the changed attribute tags in the attrlist each time a change has been made to the internal attributes. Attribute changes may have been caused by the class user using the setAttributes() method or by the application user acting on the Intuition® objects, but each class is free to use it whenever it needs.

Note that sendNotification() is only called from the IntuiObject class and never by the class user!

Within sendNotification() the class user can change attributes of other IntuiObjects he wants to notify with setAttributes().

For an ITransponder example
click here.

1.7 intuiobject_4

Here is an ITransponder example that shows you how to customarize the ITransponder class to your needs. If you are not quite aware of how ITransponder works together with the IntuiObject class
click here
.

```
class Boopsi2GT_Scroller : public ITransponder
{
   void sendNotification(attrs)  // called by BoopsiGadget
   {
      // explanation for mapAttrs() see below
      if (attrs.mapAttrs( PGA_Top, GTSC_Top, TAG_END))
      {
         //attrs only consists of GTSC_Top if PGA_Top was present before.
         if (OK(receiver1))   // check for valid receiver IntuiObject
            receiver1->setAttributes(attrs);  // change receiver attributes
      }
   }
};
```

The ITransponder usually modifies the notification attributes list 'attrs' to match attributes of the receiver IntuiObject. The AttrList class provides useful methods to map tags and to filter tags. For detailed information click AttrList.

It is not necessary for an ITranponder to have only one receiver object. But be aware of the fact that mapping the 'attrs' Attribute List alters the list.
So, if you wanted to notify several IntuiObjects each would need to get a copy of the attribute list you prepared from the received Notification Stream Attribute List. You can obtain a copy by using the copy constructor:

```
void sendNotification(attrs)
{
   // copy constructor : 'copy = AttrList(const AttrList& attrs);'
   rec1->setAttributes( AttrList(attrs) );
   // copy and work on the copy before sending..
   rec2->setAttributes( AttrList(attrs).filterAttrs(PGA_Top,TAG_END) );
   rec3->setAttributes( attrs ); // the last can have the original list
}
```

1.8 intuiobject_5

```
                Class Implementor Notes
              ----------------------------
```

Every derived class has to implement specialized getAttribute/setAttributes methods to get the full support of IntuiObject class features.
Multiple and virtual inheritance is not possible for IntuiObjects!

Within this methods the class has to reveal its public attributes on request (getAttribute), and react to new settings (setAttributes)..

There are a few points that are to be considered:

- the getAttribute() call requests the internal value of a specified Attribute Tag. If the getAttribute method recognizes the Tag as one of its own it returns the current corresponding internal value. Otherwise, if the Tag could not be recognized it has to invoke the base class' getAttributes method.

- the setAttributes() call must be propagated to the base class at one point within the overwritten setAttributes() method. Changes on base class attributes and notifications become effective when the base class' setAttributes method is invoked.

- the Attribute Taglist that is given as parameter to setAttributes() must not be altered except for one case: each class must check the attributes that belong to this class for valid values before adopting them. Values that are not conform with class specifications must then be overwritten with the current internal values.

- if the object changes an internal attribute that corresponds to a public Attribute Tag of oneself that is, without having received the request to do so via 'setAttributes', this change is to be propagated directly to IntuiObject by invoking

  ```
  setAttrs( AttrList(TAG_SPECIFIER,value,..,TAG_END);
  ```

  .

Attribute changes are propagated to the base classes until they arrive at the root base class, the IntuiObject class. From there, they are passed to the notification streams.
This is necessary since the parameter list usually contains various tags from all base classes of the object class.

For an example look at the setAttributes() method of the BoopsiGadget class:

```
...
ULONG BoopsiGadget::setAttributes(AttrList& attrs)
{
  if (notificationLoop()) return 0L;    // loop detected

   if (gadgetPtr())
   {
      // change becomes effective to the BoopsiGadget
      SetGadgetAttrsA(gadgetPtr(),(Window*)getHomeWindow()->windowPtr(),NULL ↩
         ,attrs);
   }

   return GadgetCV::setAttributes(attrs); // invoke base class method
}

void BoopsiGadget::callback(const IntuiMessageC *imsg)
{
   if (imsg && imsg->getIAddress())
   {

      switch(imsg->getClass())
      {
         case CLASS_IDCMPUPDATE :
            // msg->IAddress holds the address of a taglist with changed tag ↩
                values.
            // so, internal attributes have changed, spread the news in  ↩
               telling
            // the IntuiObject base class about it.
            setAttrs( AttrList((struct TagItem*)imsg->getIAddress()) );
            break;
      }
   }
}
```

The IntuiObject Notification Mechanism is capable of loop inhibition.
But the loop will be detected not before IntuiObject::setAttributes() is
being invoked.So, to prevent each class' setAttributes() from being executed
check for 'notificationLoop()' being TRUE and then return immediately with 0.