Introduction to Boopsi

By John Orr

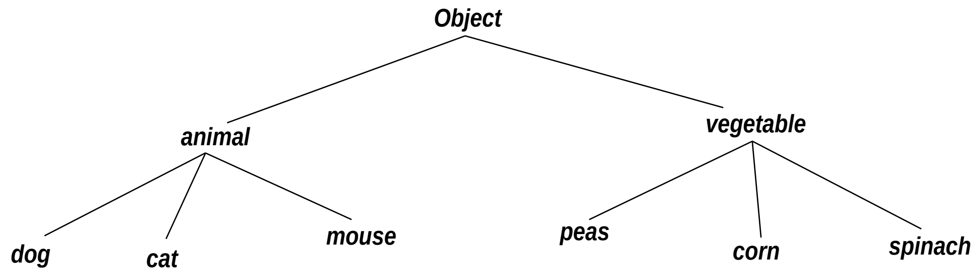
Boopsi is an acronym for Basic Object Oriented Programming System for Intuition. On its simplest level, boopsi allows the application programmer to create Intuition supported gadgets and images with minimal overhead. It allows a program to consolidate gadgets into one entity to make processing and updating easy. On a more sophisticated level, boopsi provides ways to create a wide variety of system supported, extensible custom gadgets.

Understanding boopsi requires an understanding of several of the concepts behind Object Oriented Programming (OOP). This article only briefly covers those concepts. For a more in depth explanation of those concepts, see Timothy Budd's book titled *A Little Smalltalk* (Addison-Wesley Publishing ISBN 0-201-10698-1). A port (by Bill Kinnersley) of Timothy Budd's Little Smalltalk interpreter is on Fish disk #37.

In the boopsi version of the Object Oriented Programming (OOP) model, everything is an *Object*. Each object is a distinct entity. Certain objects have similar characteristics and can be classified into different groups called *classes*. As objects, a dog, a cat, and a mouse are all distinct objects but they all have something in common; they can all be classified as animals. A specific object is an *instance* of a particular class (‘‘Rover’’ is an instance of class ‘‘dog’’).

Classes can be subdivided into *subclasses*. A vegetable object class can have several subclasses such as peas, corn, and spinach. Inversely, the *superclass* of peas, corn, and spinach is ‘‘vegetable’’. In turn, both the ‘‘animal’’ and ‘‘vegetable’’ classes are subclasses. They are subclasses of a universal root category. The OOP language *Smalltalk* calls this class ‘‘Object’’.

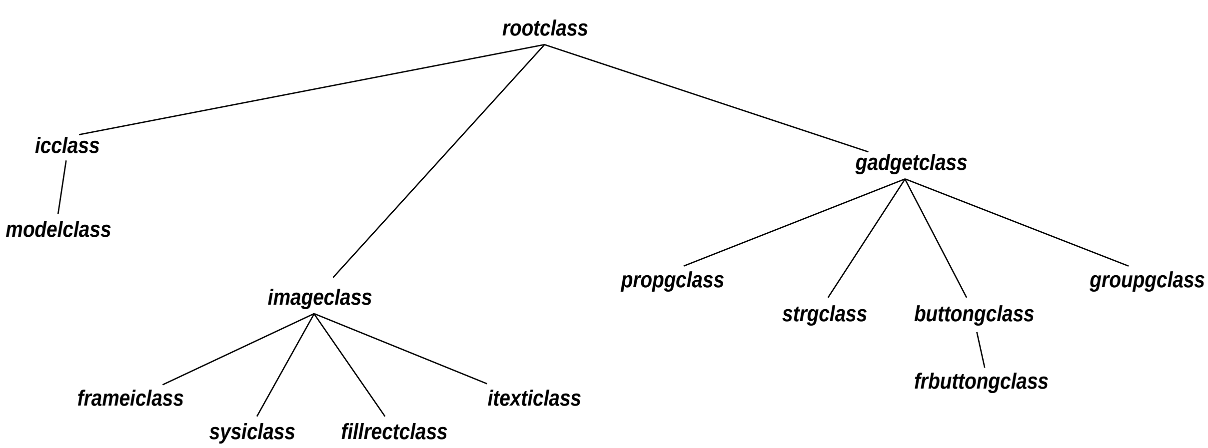
Figure 1 - Classes



In boopsi, the universal root category is called *rootclass*. Intuition supplies three subclasses of *rootclass*: *gadgetclass*, *imageclass*, and *icclass*. All boopsi gadgets in the system are an instance of *gadgetclass*. Likewise, all boopsi images are an instance of *imageclass*. The remaining subclass, *icclass*, is a concept new to Intuition that allow one boopsi object (such as a gadget) to notify another boopsi object when some specific event occurs. For example, a program can offer a user two methods of altering one integer value; one by sliding a proportional gadget, the other by typing in a value at a string gadget. Without boopsi, the program would have to explicitly update one gadget when the other was altered by the user. Using a boopsi *icclass* object, the gadgets can update each other. The gadgets update each other automatically, without the calling program's intervention.

The *rootclass'* subclasses each have their own subclasses. These subclasses are discussed later in this article.

Figure 2 - The Boopsi Classes



It is possible for an application to create its own *custom* boopsi class. When an application creates a class, it can make it either public or private. A public class has a name associated with it (for example *gadgetclass*) so arbitrary applications can access it. A private class has no name associated with it, so unless an application has a pointer to the class, it cannot access it.

Each class has a set of *methods* associated with it Each method of a class is an operation that applies to objects of that class. A example of a method for a class of integers is to add or to subtract "integer'' objects. All boopsi actions are carried out via methods.

In OOP terminology, an application requests an object to perform some method by sending the object a *message* (which is not related to the Exec style message). In boopsi, the *amiga.lib* function DoMethod() accepts a method ID and some method specific parameters (older versions of *amiga.lib* do not have DoMethod(), but the function is available on the Atlanta DevCon disks in *classface.o*). DoMethod() creates a message from these parameters and sends the message to the object:

```
ULONG DoMethod(Object *myobject, ULONG MethodID, ...);
```

where myobject is a pointer to the target object, MethodID specifies which method to perform. Any remaining parameters are passed on to the method in the form of a boopsi message. Each message contains the method ID and the parameters for the method. The parameters are method specific. For example, one way to delete the *imageclass* object *obj2delete* is to call the DoMethod() function like this:

```
DoMethod(obj2delete, OM_DISPOSE);
```

The `OM_DISPOSE` method does not require any arguments, so, in this case, DoMethod() has only two arguments.

One peculiar thing about the above function call is that the `OM_DISPOSE` method is not defined for *imageclass*. It is instead defined in the superclass of *imageclass*, *rootclass*. The `OM_DISPOSE` method works because *imageclass inherits* the methods of its superclass, *rootclass*. If an object is asked to perform a method that its class does not explicitly define, it passes the request onto its superclass for processing. This superclass can in turn pass on the request to its superclass if it does not understand the method requested.

Some of the methods currently defined for Boopsi's *rootclass* are:

`OM_NEW` - Creates a new object. Normally this method is not called directly by the application programmer (a.k.a. using the DoMethod() function). Instead, there is an Intuition function called NewObject() that takes care of creating objects.

```
APTR NewObject( struct IClass *privateclass, UBYTE *publicclassID, unsigned long tag1, ...);
```

The privateclass and publicclassID parameters are used to specify the new object's class. NewObject() only pays attention to one of these. If privateclass is `NULL`, publicclassID points to a string naming the class of the new object. If privateclass is not `NULL`, it contains a pointer to a private class. The remaining arguments make up a series of tag ID/data pairs. These are used to set the object's default attributes. These attributes are specific to the class in question. This function returns a handle to the new object.

Each object (or *instance* of a class) has *instance data* associated with it. The `OM_NEW` method takes care of allocating memory for the instance data for each class. Instance data is class specific, but all subclasses inherit instance data from their superclasses. For example, part of the instance data for a gadgetclass object is a Gadget structure. Any objects that are instances of subclasses of *gadgetclass* have a Gadget structure embedded in them. Subclasses can have their own class specific instance data in addition to the instance data inherited from the superclass.

Boopsi *gadgetclass* and *imageclass* objects are organized so that NewObject() returns a pointer to the corresponding Intuition structure embedded within them (struct Gadget and struct Image, respectively). This makes it possible to use non-boopsi Intuition functions on boopsi objects. Normally, these internal structures should be considered private, and should be accessed through the corresponding attributes, but if it is necessary, an application can look at certain fields in the embedded structure. For both gadgets and images, the Left, Top, Width, and Height fields are legal to look at. The Images NextImage field is also OK for viewing. However, this does not mean that it is OK for an application to go poking around the internals of boopsi objects. All boopsi objects are strictly private and can change without notice. Use only the functions Intuition provides for manipulating boopsi objects.

Another function, NewObjectA(), works exactly like NewObject(), but instead of accepting the tag pairs as arguments, it accepts a single pointer to a TagList. See the Intuition Autodocs for more details.

`OM_DISPOSE` - Deletes an object. Like `OM_NEW` this method is not normally called directly by the application program. Instead there is an Intuition function DisposeObject() that takes care of object disposal.

```
void DisposeObject( APTR object2delete );
```

`OM_SET` - Sets object specific attributes. This method is not normally called by the application program directly. Instead, there are two Intuition functions that set object attributes:

```
ULONG SetAttrs( APTR object, unsigned long tag1, ... );
ULONG SetGadgetAttrs( struct Gadget *mygadgetobject, struct Window *window,
      struct Requester *requester, unsigned long tag1, ... );
```

SetAttrs() accepts as arguments a pointer to the object in question and a series of tag pairs corresponding to the attributes to set. SetGadgetAttrs() is a special version of SetAttrs() that is required to change the attributes of *gadgetclass* objects. SetGadgetAttrs() is similar to SetAttrs(), except it has some extra parameters that a gadget needs to redraw itself in response to the attribute changes. This function can be used on non-gadgetclass objects as the gadget specific parameters are ignored by the other object classes. If the gadget is not yet attached to a window or requester, these arguments should be set to NULL.

Both of these functions have corresponding TagList based arguments, SetAttrsA() and SetGadgetAttrs().

OM_GET - Reads an object specific attribute. The Intuition function GetAttr() provides easy access to this method:

```
ULONG GetAttr( unsigned long attrID, APTR object, ULONG *storagePtr );
```

GetAttr() fills in storagePtr with the value of the object's attribute attrID. The function returns FALSE if there is an error.

OM_ADDMEMBER - Adds a boopsi object to another object's list, if it has one. Certain object classes have an Exec list as part of their instance data. To add the object object2add to the list of the object mainobject use DoMethod() like so:

```
DoMethod(mainobject, OM_ADDMEMBER, object2add);
```

DoMethod() also has a non-varargs form called DM() (also in *amiga.lib*). DM() accepts an object pointer and a pointer to a structure specific to a method. For example, the DM() form of the above DoMethod() call would look like this:

```
DM(mainobject, addmemstruct);
```

where addmemstruct is a pointer to the following structure (defined in <intuition/classusr.h>):

```
struct opMember {
    ULONG        MethodID;      /* in this case MethodID = OM_ADDMEMBER */
    Object       *opam_Object;  /* = addmemstruct */
};
```

OM_REMMEMBER - Removes a boopsi object previously added with OM_ADDMEMBER. The parameters are the same for OM_ADDMEMBER.

OM_UPDATE - Updates attributes of an object. For gadgets, this method is very similar to the OM_SET method. It is normally used only between objects for notifying each other of attribute changes, so simple boopsi users should use the SetAttrs() and SetGadgetAttrs() functions.

OM_NOTIFY - Notifies one object when another object's attribute(s) have changed. It is normally used only between objects for modifying each others attributes, so simple boopsi users should use the SetAttrs() and SetGadgetAttrs() functions.

Attributes

All boopsi objects have attributes associated with them. These attributes reflect various properties of a specific object. For example, an image object has attributes such as IA_Left, IA_Top, IA_Width, and IA_Height, all of which correspond to fields in the Image structure. Five methods deal with an object's attributes:

`OM_NEW` sets attributes at object creation (initialization) time
`OM_SET` changes attributes after the object has been created
`OM_GET` reads an object attribute
`OM_UPDATE` updates an object attribute (for use by objects only)
`OM_NOTIFY` notifies one object of changes to another object's attributes

Not all of these methods apply to all attributes. Some attributes are not ‘‘settable’’ or not ‘‘gettable’’, for example. See the appendix at the end of this article to find out which of these methods apply to specific attributes.

Imageclass Subclasses

An *imageclass* object contains an Image structure that Intuition uses to render objects such as gadgets. Boopsi *imageclass* objects are organized so that when NewObject() returns a pointer to an *imageclass* object, it points to the actual Image structure.

Normally, an application does not create an *imageclass* object. Instead, it will use a subclass of *imageclass*. Currently, there are four subclasses: *frameiclass*, *sysiclass*, *fillrectclass*, and *itexticlass*.

frameiclass - An embossed or recessed rectangular frame, rendered in the proper DrawInfo pens, with enough intelligence to bound or center its contents.

sysiclass - The class of system images. The class includes all the images for the system gadgets, and the Gadtools check mark and button glyphs.

fillrectclass - Rectangle with frame and pattern support.

itexticlass - A specialized image class used for rendering text. Note that you have to calculate *itexticlass* object's width and height yourself.

Gadgetclass Subclasses

A *gadgetclass* object contains an Intuition Gadget structure. Like *imageclass*, applications do not normally create objects of this class, but instead create objects of subclasses of *gadgetclass*. Currently, *gadgetclass* has four subclasses:

propgclass - An easy to use, one-dimensional proportional gadget.
strgclass - A string gadget.
groupgclass - A special gadget class that creates one composite gadget out of several others.
buttongclass - A repeating boolean button gadget.

buttongclass has a subclass of its own:

frbuttongclass - A button that outlines its label with a frame.

The example *boopsi1.c* (at the end of this article) uses *sysiclass* images and several boopsi gadgets. The example takes care of processing and updating the gadgets.

Interconnection

The boopsi gadgets in *boopsi1.c* are not very powerful. The gadgets work independently of each other, forcing the application to unify them. It is possible to make gadget objects update each other without the application's intervention.

Gadgetclass defines two attributes used in this updating process: `ICA_Target` and `ICA_Map`. The `ICA_Target` attribute specifies a pointer to a target object. If an object's attribute changes (and the `OM_NOTIFY` method applies to that attribute), the object sends itself an `OM_NOTIFY` message. If the object has a target object, the target will receive an `OM_UPDATE` message which usually tells the target to set one of its own attributes. For example, when the user slides the knob of a *propgclass* object, its `PGA_Top` attribute changes. If this object has a second *propgclass* object as its target, the second object will receive an `OM_UPDATE` message and will set its corresponding `PGA_Top` attribute to the `PGA_Top` value of the first *propgclass* object.

Because objects of different classes do not always have corresponding attributes, there is a way to map attributes of one object to the attributes of another. The `ICA_Map` accepts a TagItem array of attribute pairs. The first entry in the pair is the source attribute ID and the second is the target object's attribute. For example, an `ICA_Map` that maps a prop gadget's `PGA_Top` attribute to a string gadget's

STRINGA_LongVal attribute would look like this:

```
struct TagItem slidertostring[] = {
    {PGA_Top, STRINGA_LongVal},
    {TAG_END, }
};
```

Note that the OM_UPDATE method has to apply to the target attribute for the change to take place.

Although the gadget attributes ICA_Target and ICA_Map allow boopsi gadgets to update each other, by themselves these attributes do not provide the *application* with any information on changes to the gadgets. Instead of using another gadget as a target object, an object targets an *icclass* object. *Icclass* (or InterConnection) objects are simple information forwarders.

Icclass defines two attributes: ICA_Target and ICA_Map, which are almost identical to the *gadgetclass* attributes. The difference is that *icclass* objects can send the application an IDCMPUPDATE IntuiMessage when it receives an OM_UPDATE. If an *icclass* object receives an OM_UPDATE, it will send the IntuiMessage only if the *icclass* object's ICA_Target is ICTARGET_IDCMP and the updated attribute is mapped to a special dummy attribute, ICSPECIAL_CODE. The IntuiMessage's Code field contains the lower 16 bits of the updated attribute.

Icclass has a more powerful subclass called *modelclass*. A *modelclass* object sends OM_UPDATE messages to an entire list of objects that the *modelclass* object maintains. This makes it possible for gadgets to update each other *and* for the application to find out about the changes. A *modelclass* object ‘‘broadcasts’’ attribute changes to its list of boopsi objects *and* it also lets the application know about attribute changes because it inherits the ICTARGET_IDCMP mechanism from *icclass*. To add objects to a *modelclass* object's list, use the OM_ADDMEMBER method.

The power of *icclass* and *modelclass* lies in using them to create a custom subclass. Consider a group of prop gadgets each of which is used to control one of a color's R, G, B, H, S, and V components. When the user changes one of the color components with a prop gadget, the custom model will be notified of that change and, because it is customized for this specific purpose, it will *recalculate* the values for the remaining components. The model will then let the rest of the prop gadget objects (which are attached to *icclass* objects in the custom model's personal list) know about the changes to their component values so they can move their slider to its new position. All this work is done by the objects themselves; the application does not have to process any of the intermediate input.

In general, boopsi's power lies in its Intuition-supported extensibility. Using the existing boopsi classes as a foundation, you can create entirely new subclasses. This makes it possible to create your own custom gadgetry and have it work perfectly with Intuition, just like any existing gadget. New subclasses of *modelclass* can be used to create gadgets that talk directly to ARexx or the clipboard device. If new classes are general enough, they can be made public so other applications can use them.