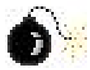
Objectbase Library

Documentation and Implementation Status



Compatibility

- 1.0.4 => 1.0.5
  No changes.
- 1.0.3 => 1.0.4
  The name of this library is now **objectbase**, it is has been renamed from **swarmobject** largely to reflect the more generic nature of the library and also motivated by the impending port of Swarm to Windows NT (to avoid filename conflicts with the SwarmObject class). There should be little, or no effect on the user, the only visible change is the fact that the actual library (.a) or (.so) file will now have a different name and the header file name has changed. A symbolic link from objectbase.h to swarmobject.h has been provided in the distribution, to ensure backwards compatibility, however, users should not continue to rely on this being so. Users should port references to swarmobject.h in their code to objectbase.h, because this symlink will be removed in a future release.
  Note: this is no way affects the SwarmObject class which remains the same as in all previous releases.
- 1.0.0 => 1.0.1
  The interface has changed again! *EmptyProbeMap* is now a subclass of *CustomProbeMap*, which is subclassed from *ProbeMap*. And a shortcut create: method was added to that branch.
  Also, a new method was added to *ProbeLibrary* called isProbeMapDefinedFor that serves to non-invasively test for the existence of a *ProbeMap* for a given class.
- *Beta* => 1.0.0
  The new interface for the *swarmobject* library might cause some problems for apps that worked under the Beta release of Swarm. To get the whole scoop, read the Library Interface Conventions
  (defobj/library.html)
  . To get a few hints at explicit changes you might need to make, check out: Porting apps to use the new *swarmobject* interface
  (objectbase/PortingAppsToNewInterface.html)
  .

Overview

The *objectbase* library encapsulates various fundamental aspects of the `Swarm` object model and defines the probing machinery used to take data from `Swarm` objects. Most of the underlying functionality of the classes defined here is contained in the *defobj*

(defobj.html)
and *activity*
(activity.html)
libraries.

```
The main classes defined here that are intended to be subclassed by users are Swarm
and SwarmObject.  The probing functionality provided here is mainly for use within
the Swarm kernel.  However, this probe machinery should be used when designing any
interface between Swarm and any other agent or device.
```

SwarmObject

SwarmObject is the base class for all interactive agents in `Swarm`. It defines the standard behavior for Swarmstyle agents, which includes hooks for creation, probing, zoned memory allocation, and destruction.

Swarm

The *Swarm* class encapsulates all of the abstract execution machinery in the *activity* library, as well as the notion of a group of related objects. Metaphorically, a "Swarm" is a combination of a collection of objects and a schedule of activity over those objects. Hence, the *Swarm* class provides the behavior necessary for creating such an object and starting up the schedule.

Probes

Probes are idealized entities that are intended to allow the user to monitor and modify the innards of objects without explicitly providing the functionality to do so at compile time. Hence, they allow dynamic interaction with an objects instance variables and methods. Most of the functionality of probes is implemented here; but, at present, they are intimately linked to the *simtools*

(simtools.html)
library, which contains all the widgetry needed to use probes from a GUI. Despite this intimacy, probes are intended to be a general purpose mechanism for any agent or device to interact dynamically with `Swarm` objects. This `Introduction to Probes`
`(probing.html)`
will explain more about the reasoning and purpose behind probes.

Documentation

The *objectbase* library is defined and documented following the basic structure of the `Library Interface Conventions`

`(defobj/library.html)`
. The following standard sections of documentation are available:

- `Interface Reference`
  `(objectbase/Interface.html)`
- `Usage Guide`
  `(objectbase/UsageGuide.html)`
- `Advanced Usage Notes`
  `(objectbase/AdvancedUsage.html)`
- `Interface Design Notes`
  `(objectbase/InterfaceDesign.html)`
- `Implementation Notes`
  `(objectbase/Implementation.html)`

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-10

Objectbase Library: Documentation and Implementation Status

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-08

Swarmobject Library: Porting Apps to the New Interface

The problems that might arise because of this new interface should have to do with the placement or existence of C preprocessor `#import`s. In the beta release, the `objectbase.h` file simply included all the header files for each class defined in the objectbase library. These files contain the `@interface` declarations for each respective class. Hence, before, if an application file included `objectbase.h`, it was including the interface for every class in the library.

The Swarm standard module interface uses Objective-C protocols to establish which messages may be sent to objects of a class. Doing this isolates the classes in the library more completely from the rest of the code. So, the problems that are likely to arise from this new interface are ones of access to objects that reside in the objectbase library. And they are of two types:

1. Messages that are no longer available on one or more classes where they were available before. I.e. you see warnings like the following when you compile your app:

```
warning: `ClassName' does not respond to `messageName'
```

What this means is that since `objectbase.h` simply imported all the class header files, somewhere in that chaotic mess of an import tree, the header for the `ClassName` was imported. Now, since `objectbase.h` only defines protocols for the classes it contains, it no longer imported that header file.

Solution #1: If `ClassName` is a member of the objectbase library, then all you need to do is import `objectbase.h` in the file where that class (or an instance of it) is used. If, however, the class in question is a member of some other library, then you should find out which library it is a member of and include the appropriate header file. (This latter case is actually part of the justification for making these libraries into "modules" like this.... to avoid incestuous include trees.)

Another case where this occurs is when using objects that have been statically typed with a class name. In using the protocols instead of direct class interfaces, we are depriving the compiler of the ability to analyze the type of an object and decide if that object can accept a given message. This is a bit counter intuitive for those of us reared with strongly typed languages and styles. But, Swarm takes advantage of the freedom this allows and it is part of Swarm-style programming

(../defobj/style.html)

.

Solution #2: If this is what's happening to generate the above warning, there are two further degradations of typing available. The next step down is to take away the specific class type and replace it with a protocol qualification on a pure `id` type. For example:

```
ProbeMap * myProbeMap;
```

can be changed to

```
id <ProbeMap> myProbeMap;
```

This both allows the freedom from the static typing and adds to the readibility of the code. However, this still results in compile time checking for messages sent to that object with the protocol designated. So, any message you intend to send to that object must be defined or inherited by the given protocol. So, in some cases, one may want to go with the ultimate freedom in

```
id myProbeMap;
```

which, obviously, means the compiler will not second guess the programmer in what messages can be sent to that object. And if your variables are named appropriately, this doesn't really detract from the readibility of the code.

2. When subclassing from a class defined in the Swarm Kernel, it is necessary to import the `@interface` of that class. Before, when `objectbase.h` included all those interfaces, it was the only file needed if you wanted to subclass from a class in objectbase. Now, however, simply importing `objectbase.h` doesn't help at all. So, in order to subclass from a class that's a member of objectbase, you need to import the specific interface file to that class.

   The solution: If you had an app that worked before when you subclassed from something (say, Swarm) in objectbase but doesn't work now, simply replace the

```
#import <objectbase.h>
```

in the header file for the subclass with

```
#import <objectbase/Swarm.h>
```

Note: In order to reduce the number of broken apps due to this, we've explicitly imported `objectbase/SwarmObject.h` and `objectbase/Swarm.h`at the end of the `objectbase.h` file. This allows the subclassing of SwarmObject or Swarm by an import of `objectbase.h` without the explicit import of the interface files. But, this was done for backward compatibility and may go away in the future. So please subclass by explicit importation of the interface file of the class.

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-08

Objectbase Library: Interface Reference

The *objectbase* library uses the module definition conventions
(../defobj/module.html)
established by the *defobj* library
(../defobj.html)
. It also depends on standard supertypes and classes defined by this library.

Dependencies

Following are the other header files imported by <objectbase.h>:

```
#import <defobj.h>
#import <objectbase/SwarmObject.h>;
```

The *defobj* library interface is included to provide the basic object support. The *SwarmObject* class header file was provided for backward compatibility since the *objectbase* library has been brought into agreement with the library standards.

Object Types (with links to reference pages)

```
SwarmObject
 (SwarmObject.html)
                  /     |     \
                 /      |      \                  ProbeLibrary
 (ProbeLibrary.html)
                /       |       \
  ActivityControl
 (ActivityControl.html)
      Probe
 (Probe.html)
      ProbeMap
 (ProbeMap.html)
                     /   |        |  \   \
                   /     |        |   \    \
                 /       |        |    \     \
               /         |        |     \      \
        VarProbe
 (VarProbe.html)
       Message
 (MessageProbe.html)
   Custom
```

```
(CustomProbeMap.html)
   \      Complete
(CompleteVarMap.html)
                              Probe
(MessageProbe.html)
   Probe
(CustomProbeMap.html)
    \       Var
(CompleteVarMap.html)
                                          Map
(CustomProbeMap.html)
      \       Map
(CompleteVarMap.html)
                                                     Complete
(CompleteProbeMap.html)
                                                      Probe
(CompleteProbeMap.html)
                                                       Map
(CompleteProbeMap.html)
```

(don't want a graphic with image map to avoid excessive download times!)

Contents:

(CompleteProbeMap.html)

(../objectbase.html)

(../objectbase.html)

(UsageGuide.html)

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-10

SwarmObject Class

```
//
// SwarmObject --
//   Superclass of most objects in a Swarm simulation.
//   Provides support for probing.
//
@protocol SwarmObject <Create, Drop>
USING
- (const char*)getInstanceName;
- getProbeMap;
- getCompleteProbeMap;
- getProbeForVariable: (const char *)aVariable;
- getProbeForMessage: (const char *)aMessage;
@end
```

A SwarmObject is an object that is intended to be a member of a Swarm. That is, its behavior will be perpetuated by messages sent from the schedule of events defined in the context of a Swarm object.

The SwarmObject is where the models of all the agents of a simulation will reside. Hence, most of the burden on defining the messages that can be sent to any agent lies with the user. SwarmObject inherits its basic functionality from the Create and Drop object types defined in the defobj library. It is useful to mention these messages here to show the complete span of messages that can dictate the behavior of an instance of a SwarmObject; but it is not useful to explain their function in detail. The messages SwarmObject inherits from defobj are:

- DefinedObject
  - \- getType;
  - \- getClass;
  - \- (BOOL)respondsTo: (SEL)aSel;
  - \- describe: anOutputStream;
  - \- (BOOL)setDisplayNameTo: (const char *)aName;
  - \- (const char *)getDisplayName;
  - \- createIDString: aZone;
- Create
  - \+ create: aZone;

  - + createBegin: aZone;
  - - createEnd;
  - + customizeBegin: aZone;
  - - customizeEnd;
  - - customizeCopy: aZone;
- Drop
  - - getZone;
  - - (void)drop;

In addition to these, SwarmObject is augmented by the messages that provide the probe support.

getInstanceName is a message used by the probe machinery to get a meaningful word or phrase to indicate the object being probed (and, in the case of a probe display, place it in a widget on the screen). Support for this has recently been added to the DefinedObject object type, thereby making it more generally available. So, this message will likely become obsolete soon.

getProbeMap returns a pointer to the ProbeMap for the object if there has been one created for that object's class. If it hasn't, then it create's a default one.

getCompleteProbeMap returns a newly created CompleteProbeMap for the object.

getProbeForVariable: (const char *)aVariable returns the VarProbe indexed in the ProbeMap by the string aVariable.

getProbeForMessage: (const char *)aMessage returns the MessageProbe indexed in the ProbeMap by the string aMessage.

(Interface.html)

(../objectbase.html)

(ProbeLibrary.html)

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-10

ProbeLibrary Class

```
//
// ProbeLibrary --
// A (singleton) Class, whose instance is used as a container for aglobal
// mapping between classnames and their 'default' ProbeMaps. These defaults
// can be changed by the user, thus allowing him/her to customize the default
// contents of the ProbeDisplays generated when probing objects.
//
@protocol ProbeLibrary <Create, Drop, ProbeConfig>
USING
- setDisplayPrecision: (int) nSigDisplay;
- (int)getDisplayPrecision;
- setSavedPrecision: (int) nSigSaved;
- (int)getSavedPrecision;
- (BOOL)isProbeMapDefinedFor: (Class)aClass;
- getProbeMapFor: (Class)aClass;
- getCompleteProbeMapFor: (Class)aClass;
- getCompleteVarMapFor: (Class)aClass;
- getProbeForVariable: (const char *)aVar inClass: (Class)aClass;
- getProbeForMessage: (const char *)aMessage inClass: (Class)aClass;
- setProbeMap: aMap For: (Class)aClass;
@end

extern id <ProbeLibrary> probeLibrary;
```

The normal Swarm simulation will probably only ever contain one instance of this class, namely the *probeLibrary* object. This object is used for Library Generation of *Probe*s and *ProbeMaps*: its role is to cache one unique "official" *ProbeMap* for every Class ever probed during a run of Swarm. These *ProbeMap*s are generated lazily as they are requested.

- setDisplayPrecision: (int)nSigDisplay

  Sets the number of significant digits for floating point and double floating point numbers displayed on GUI widgets. Currently only implemented for `VarProbes` not `MessageProbes` as yet. This allows *all* probes checked out the global ProbeLibrary instance to access this displayed precision. However, individual probes can vary from this global default, by using the `setFloatFormat` method on a exisiting probe. See probing precision (../probing.html#precision) for more details on how to do this.

- (int)getDisplayPrecision

Gets the current display precision set in the ProbeLibrary instance. Mostly used internally.

- setSavedPrecision: (int)nSigSaved

  Sets the number of significant digits saved for floating-point and double floating-point numbers through ObjectSaver. This function sets the global default precision for all floating (and double floating) point numbers. This floating point precision affects all numbers saved via the ObjectSaver class. There is currently no way to override this global default for an individual probe.

-(int) getSavedPrecision

  Gets the current saved precision set in the ProbeLibrary instance. Mostly used internally.

Note: if neither of these methods are used to set the defaults, the precision in both cases defaults to six (6) significant figures, which should simulate the default precision before the introduction on this feature *on most platforms*. Previously, the sprintf "%g" default format was used to print out floats and doubles, which is not considered to have entirely portable behavoir.

- (BOOL) isProbeMapDefinedFor: (Class) aClass returns True

  If there is a non-nil value in the *ProbeLibrary* for that class and False otherwise.

- getProbeMapFor: (Class)aClass

  returns a *ProbeMap* for the *aClass* class. If a specific *ProbeMap* has been designed and installed in the *ProbeLibrary* for that class, then that *ProbeMap* is returned. If not, then a *CompleteProbeMap* is created and returned.

- getCompleteProbeMapFor: (Class)aClass

  This method returns a *ProbeMap* containing *Probe*s for all the instance variables and messages of the given Class (including inherited variables and messages). Since we are assuming that normal, single-level *ProbeMap*s will be more widely used, the current implementation of *ProbeLibrary* does not cache *CompleteProbeMap*s. However, if we find that there is a demand for *CompleteProbeMap*s, we will cache them as well.

- getCompleteVarMapFor: (Class)aClass

  This method returns a *ProbeMap* containing Probes for all the instance variables of the given Class (including inherited variables) but does not include any *MessageProbe*s.

- getProbeForVariable: (const char *)aVariable inClass: (Class)aClass

- getProbeForMessage: (const char *)aMessage inClass: (Class)aClass

  These methods are used to "check out" the appropriate Probes from the probe library. Note that the returned probes will be cached, so you must make sure to clone them if you want to make modifications to the probes without affecting the result of future requests for those same probes.

- setProbeMap: aMap For: (Class)aClass

In those cases where the user wants to set the standard probe map directly, she can provide the probe map using this method, and it will be cached as though it were produced by the library itself.

(SwarmObject.html)

(Interface.html)

(Probe.html)

Marcus G. Daniels <mgd@santafe.edu> (mailto:mgd@santafe.edu)

Last modified: 1998-01-10

ActivityControl Class

```
//
// ActivityControl -- This class provides an object interface to
//                    an activity.  It contains a subset of the messages
//                    that any activity responds to and provides explicit
//                    control over that activity.
//
@protocol ActivityControl <SwarmObject>
CREATING
- attachToActivity: anActivity;
USING
- run;
- stop;
- next;
- step;
- stepUntil: (timeval_t)stopTime;
- (void)terminate;
- updateStateVar;
- getStatus;
- (const char *)getInstanceName;
@end
```

The ActivityControl class specifies an object that can be attached to an activity (regardless of how or where that activity is created) for the purpose of explicitly controlling the execution of actions on that activity's action plan. There is nothing that available through this class that is not already available through the variables or messages available via the activity itself. However, it packages up the main control messages and makes them available to other objects that may need control over an activity, thereby shielding the activity from directly receiving messages from outside objects and saving the user from having to parse the more complex interface to the activity.

Create Phase Protocols

attachToActivity: anActivity sets an instance variable inside the ActivityControl object that points to the Activity to be controlled. It then creates a Schedule upon which it places a message to itself to update its own variables.

Usage Phase Protocols

run sends a run message to the activity if the conditions are appropriate. This message causes the activity to continue executing the actions on it's schedule until either no other actions are waiting, at which point it returns Completed, or until it is stopped by a subactivity or a stop message to the activity.

stop sends a stop message to the activity if the conditions are appropriate. This message causes the control to move back up the run-stack and resume at the place in the code where the run was first executed, which doesn't mean much in a parallel execution context; but, in a serial context, it means that the next action on the super-activity will begin without finishing the rest of the current activity's actions.

next sends a next message to the activity if the conditions are appropriate. It runs an activity forward through as many actions as necessary until it hits a breakFunction, at which point it walks back up the tree of activities and returns Stopped. In most cases, this means that an entire action or action group on the activity under control will be executed, including completion of all subactivities.

step sends a step message to the activity if the conditions are appropriate. It causes the execution of a single action.

stepUntil: (timeval_t)stopTime sends a stepUntil: message to the activity if conditions are appropriate. This causes all actions on the activity's schedule, including any actions on subactivities' schedules, to be executed until the activity's relative time is equal to stopTime - 1.

(void)terminate sends the terminate message to the activity, which causes all actions and action groups to be removed from its schedule. This can cause some problems. If terminate is sent to an activity and, subsequently, a run is attempted on that activity, the program will exit with an error. A way to avoid this might be to eliminate the ActivityControl attached to the terminated activity; but, that would preclude the reuse of ActivityControllers on other activities. Ideally, a controller could be adjusted to monitor any of several activities and it could leap from one activity to another. So, this behaviour is advertised as a feature.

updateStateVar updates the ActivityControl instance variables and tests for the continued existence of the activity that's being controlled. This message is sent on each cycle of the schedule for the activity being controlled.

getStatus returns the status of the activity.

(const char *)getInstanceName This method is provided for backwards compatibility to the probe machinery. It echos the displayName resident in the object name database. When the probe machinery is upgraded to use this database, this method will become obsolete.

(CompleteVarMap.html)

(Interface.html)

(UsageGuide.html)

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-10

Probe Class

```
//
// Probe --
//   Abstract superclass of both VarProbe and MessageProbe.
//
@protocol Probe <SwarmObject>
CREATING
- setProbedClass;
SETTING
- setSafety;     // There should be a setSafety: BOOL
- unsetSafety;   // and getSafety methods instead...
USING
- clone: aZone;
- (Class)getProbedClass;
- (const char *)getProbedType;
@end
```

A Probe is simply an object that contains pointers to an element (instance variable or message description) of another object. The Probe contains instance variables that describe the referent's class and type. It's actually an abstract class that is further subdivided into VarProbe and MessageProbe, which represent the two basic types of elements of any object. The Probes are collected into a ProbeMap and subsequently installed in the ProbeLibrary.

Create Phase Protocol

setProbedClass: (Class)aClass sets the class of the referent and must be called at create time.

Setting Phase Protocol

setSafety turns on the option of checking the compatibility of the class of the object before any actions are performed on the referent. This is only used in the VarProbes and should probably be moved.

unsetSafety turns the safety option off.

Regular Protocol

(Class)getProbedClass returns the class of the referent as a Class pointer.

`(const char *)getProbedType` returns the typing of the probed variable/message. The typing is represented using the string-format provided by the Objective-C runtime system ('c' for char, '@' for id, and so on). In general the programmer will know the typing of the variable so this method will probably only be used internally by Swarm kernel code.

`clone: aZone` returns a clone of the probe. It is useful in case you got the probe by Library Generation or by the default version of Object Generation. In both cases you will have been given a unique shared instance of the probe - so if you wanted to make changes to the safety status of the probe etc. without affecting the other potential users of the probe, you would first clone it and then make the required changes.

(ProbeLibrary.html)

(SwarmObject.html)

(VarProbe.html)

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-10

VarProbe Class

```
//
// VarProbe --
// Allows the user to inspect a given variable in any candidate
// which is an instance of, or inherits from, a given class
//
@protocol VarProbe <Probe>
CREATING
- setProbedVariable: (const char *)aVariable;
SETTING
- setStringReturnType: returnType;
USING
- (const char *)getProbedVariable;
- setFloatFormat: (const char *)format;
// Currently, setNonInteractive must be used *after* create phase
// In any case these should also become setInteractive: BOOL etc.

- setNonInteractive;
- (int)isInteractive;

- (void *)probeRaw: anObject;
- (void *)probeAsPointer: anObject;
- (int)probeAsInt: anObject;
- (double)probeAsDouble: anObject;
- (const char *)probeAsString: anObject Buffer: (char *)buffer;
- (const char *)probeAsString: (id) anObject Buffer: (char *)buf withFullPre-
cision: (int)precision;
- setData: anObject To: (void *) newValue; // pass by reference.
- (int)setData: anObject ToString: (const char *)s;
@end
```

This is a specialized subclass of the abstract class *Probe*. It completes the specification of a probe whose referent is an instance variable element of an object.

Create Phase Protocol

- setProbedVariable: (const char *)aVariable sets the variable being probed (elsewhere called the referent). The *aVariable* identifier is simply a character string consisting of the identifier of the variable referent. For example, if we were probing a variable called *unhappiness*, then the string would be "unhappiness". This method must be called during the create phase.

Set Phase Protocol

- setStringReturnType: returnType sets the format that will be used to print the variable.

Usage Protocol

- setFloatFormat: (const char *)
  Given an sprintf style formatting string (eg. "%.3f") to a floating-point (or double-floating-point) probe will use this formatting *only* when probing on a GUI display widget. See section on probing precision (../probing.html#precision) when probing floating point number in the documentation on probes.
- setSafety
- unsetSafety
  In order to ensure efficiency, the default behaviour of the *VarProbe* is to "trust" that the object it is being presented with is of an appropriate class (the one specified in *setProbedClass*, or a subclass of it). However, by calling *setSafety*, we force the probe to inspect the class of the target object before probing it.
- setNonInteractive
- (int)isInteractive
  By setting a *VarProbe* to be non-interactive we ensure that the user will not be able to change the value of the probe, but only observe it. The default is for variables of type (char *), char, unsigned char, int, unsigned int, floatand double to be interactive and for all the others to be non-interactive (with the exception of the id type this follows because there is no simple way to set such things as structs or arrays by typing values into an entry field). For the time being, setting the VarProbe to be non-interactive will not interfere with the drag&drop capability of objects into the *VarProbe* entry-fiel This is considered an omission and will be corrected by the next release. Also, for the time being there is no way to disable a *VarProbe* once it has been displayed (i.e. after a *ProbeDisplay* containing the *VarProbe* has been created). The ability to do this will also be included in the next release.
- (const char *)getProbedVariable
  returns a string matching the identifier of the referent.
- (int)getDataOffset
  This is another example of a method used(?) exclusively by the Swarm kernel, it returns the offset used to extract the variable from the target object.

- (void *)probeRaw: anObject

- (void *)probeAsPointer: anObject

- (int)probeAsInt: anObject

- (double)probeAsDouble: anObject

- (const char *)probeAsString: anObject Buffer: (char *) buffer

- (const char *)probeAsString: anObject Buffer: (char *) buffer withPrecision: (int) precision

  These are the methods which actually use the probe. *probeRaw* simply returns a pointer to the probed variable, whereas the other methods return it as a particular type. *probeAsString* simply prints the value of the variable into the buffer (in more or less the same format you would expect from a *sprintf()*). The buffer should be pre-allocated. The second form of *probeAsString* is only used internally by ObjectSaver

  (../simtools/ObjectSaver.html)

  to use the "saved as" precision form which can be different from the "displayed" precision. See the section on setting the probing precision

  (../probing.html#precision)

  for floating-point numbers. Note: The *VarProbe* will complain if you try to do weird type conversions: for example, if the probed variable is of type pointer and you try to use the *probeAsDouble* method then the return value will be undefined (and you will get an error message on *stderr*).

- setStringReturnType: returnType

  When the *probedVariable* is of type unsigned char or char, the method *probeAsString* will, by default, return a string of the format: "'%c' %d". This is meant to reflect the commonplace use of an unsigned char as a small int. However, the user can tailor the format returned by the probe to use only one of the two possible representations.

The legal values for returnType are:

- DefaultString ("'%c' %i")
- CharString ("'%c'")
- IntString ("%i").

  Note: When setting the value of an unsigned char or a char using the *setData:ToStr* (method described below), the expected format of the string is always "%i" unless *CharString* was chosen (in which case the format should be "'%c'").

- setData: anObject To: (void *)newValue

- setData: anObject ToString: (const char *)s

  The programmer can set the *probedVariable* via these methods, either by providing a pointer to the new value or a string which the probe will read and convert appropriately. Note: For the time being, the kernel itself is the only software which uses probes in this fashion, so we can get away with expecting either a string or a pointer. However, if any user should require more types (such as the choice provided in the *probeAs* series), please email (mailto:swarm@santafe.edu) us and I will include them in the following release!

(Probe.html)

(SwarmObject.html)

(MessageProbe.html)

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-10

MessageProbe Class

```
//
// MessageProbe --
//   Allows the user to call a given message on any candidate
//   which is an instance of, or inherits from, a given class
//
@protocol MessageProbe <Probe>
CREATING
- setProbedSelector: (SEL) aSel;
- setProbedMessage: (const char *)aMessage;
- setHideResult: (int) val;
USING
-(int)isResultId;                  // I doubt that a user will
-(int)isArgumentId: (int)which;    // ever need these.

- (const char *)getProbedMessage;
- (int)getArgNum;
- (const char *)getArg: (int)which;
- (const char *)getArgName: (int)which;
- (int)getHideResult;

- setArg: (int)which To: (const char *)what;

- dynamicCallOn: target resultStorage: (const char **)result;
- dynamicCallOn: target;

- updateMethodCache: anObject;
- (int)intDynamicCallOn: target;
- (float)floatDynamicCallOn: target;
- (double)doubleDynamicCallOn: target;
@end
```

This is a specialized subclass of the abstract class Probe. It completes the specification of a probe whose referent is a message element of an object.

Create Phase Protocol

setProbedSelector: (SEL)aSelector

setProbedMessage: (const char *)aMessage

sets the message to be probed. At least one of these methods must be called.

Note that due to a bug in gcc-2.7.2 it is unadvisable (though we have not disallowed it) to make a message probe using setProbedMessage. The next release of gcc will contain a patch for this bug. If you absolutely require the ability to create messageprobe using setProbedMessage, please email
(mailto: swarm@santafe.edu)
us.

Usage Protocol

(const char *)getProbedMessage
: returns the string matching the identifier of the message being probed.

(int)getHideResult
: setHideResult: (int)val By setting hideResult to 1, the user is indicating that the result field, in a graphical representation of the message probe, should not be shown. In other words, if a probe display attempts to represent the message probe, it should hide the result window. This is useful, particularly when the return value is defined to be self.

(const char *)getArg: (int)which
: returns a string representation of the n'th argument.

(int)getArgNum
: returns the number of arguments the message takes.

(const char *)getArgName: (int)which
: returns a string representation of the argument with the given name.

setArg: (int)which To: (const char *)what
: sets the n'th argument of the message - the argument must be provided in string form.

(int)isResultId
: This method is used by the Swarm internals to find out whether the return value of the message is of type object.

(int)isArgumentId: (int)which
: This method is used by the Swarm internals to find out whether a given argument of the message is of type object.

updateMethodCache: anObject
: The message probe will always attempt to optimise the way in which it dispatches dynamic calls to the probed objects. In practice, this means that when the probed message takes no arguments, then the message probe will cache the actual implementation of the method and generate direct C function calls (as opposed to calling perform, for example) if the user promises that only one class of objects is being probed. The user does this by calling the updateMethodCache: with an example object of the appropriate target class. This method can be called as often as necessary throughout the lifetime of the message probe.

dynamicCallOn: target

dynamicCallOn: target resultStorage: (const char **)result

(int)intDynamicCallOn: target

(double)doubleDynamicCallOn: target

These are the methods used to actually generate a dynamic message call on the target object. The first method does not return a result, the second will return a dynamically malloc'ed string based representation of the result, whereas the third and fourth methods assume the user knows the exact typing of the method and would like a direct translation into the appropriate datatype.

(VarProbe.html)

(SwarmObject.html)

(ProbeMap.html)

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-10

CustomProbeMap Class

```
//
// CustomProbeMap --
//   A subclass of ProbeMap, whose initial state is empty unlike the default
//   probeMap initial state which contains all the VarProbes
//   of the requested target class.
//
@protocol CustomProbeMap <ProbeMap, CREATABLE>
+ create: aZone forClass: (Class)aClass withIdenti-
fiers: (const char *)vars, ...;
@end
```

This subclass of the ProbeMap is used to create probe maps which are initialised in an emtpy state or with the VarProbes and MessageProbes intended. In other words, the probed class is set, as is the case with the normal ProbeMap class but upon createEnd no VarProbes or MessageProbes will be present within it. This feature is useful when creating a probe map from scratch (e.g. to be used in conjunction with the setProbeMap:For message of the ProbeLibrary).

An additional convenience method was added to help compress the amount of code needed to create a custom probe.

**+ create: aZone forClass: (Class)aClass withIdenti-
fiers: (const char *)vars, ...;**

This is a variable argument message which takes a Zone, a Class name, and a colon-delimited list of instance variable names and message names valid for that class. The instance variable identifiers are listed first followed by a string consisting solely of a ":" followed by the message identifiers. The general format would be:

```
[CustomProbeMap create: aZone
              forClass: aClass
       withIdentifiers: "variable1", "variable2", ...,
                        ":",
                        "message1", "message2", ...,
                        NULL];
```

All strings up to the ":" are read as instance variable names and placed in the probe map and all strings after the ":" are read as message names and placed in the probe map. The NULL string terminates the list. An example of this message being used is given in the Advanced Usage Guide
(./AdvancedUsage.html)
.

(ProbeMap.html)

(Interface.html)

(CompleteProbeMap.html)

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-10

CompleteProbeMap Class

```
//
// CompleteProbeMap --
//   A subclass of ProbeMap, whose initial state contains the VarProbes and
//   MessageProbes of the requested target class but also those of all its
//   superclasses.
//
@protocol CompleteProbeMap <ProbeMap> @end
```

Upon creation, this subclass of the ProbeMap will contain all the variables and all the messages of a given class (including the inherited ones).

(EmptyProbeMap.html)

(Interface.html)

(CompleteVarMap.html)

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-10

Objectbase Library: Usage Guide

Overview

The objectbase library contains the most basic objects users need to design their agents and swarms. It also serves, at present, as a repository for the probe machinery, which is provided for every SwarmObject. The way the classes in this library are to be used is varied. But, basically, it is provided so that the user will have something to subclass from for her own objects and Swarms.

Example Usage of SwarmObject

The best way to explain how the library should be used is to walk through an example. So, using Heatbugs, we'll walk through the ways objectbase is used and discuss them. Since more documentation is usually better than less, I'm going to explain things at a low level so that those not familiar with Objective C will understand the discussion. If you already are familiar with Objective C, then you should skip this part.

First off, the basic elements of the Heatbugs simulation are the heatbugs, the model swarm (which bundles the heatbugs), and the observer swarm (which bundles the displays of the probes poking into the model swarm and the heatbugs). The interface files for each show what must be imported and the declaration syntax needed to subclass from SwarmObject.

We'll use `Heatbug.h` for our discussion here. The first part of the file shows the C-preprocessor imports needed:

```
#import <objectbase/SwarmObject.h>
#import <space.h>
#import "HeatSpace.h"
#import <tkobjc/Raster.h>
```

The `#import <objectbase/SwarmObject.h>`; is included in order to subclass from SwarmObject. However, to provide backwards compatibility, we've placed this import in the library interface file `objectbase.h` as well, which means one could subclass from SwarmObject by simply importing the `objectbase.h` file. This is discouraged in order to make the library interfaces as standard as possible.

The next objectbase relevant piece of code in this file is:

```
@interface Heatbug: SwarmObject
{
  double unhappiness;
  int x, y;
  HeatValue idealTemperature;
  HeatValue outputHeat;
```

```
  float randomMoveProbability;

  Grid2d * world;
  int worldXSize, worldYSize;
  HeatSpace * heat;
  Color bugColor;
}
```

The @interface keyword indicates that you are beginning the definition of the part of an object (a Heatbug in this case) that will be visible to other objects. The Heatbug: SwarmObject indicates that you are calling this object Heatbug and it is a subclass of SwarmObject. What follows between the curly braces ({}) are the instance variables defined for the Heatbug class above and beyond those inherited from the SwarmObject class.

Inside this "agent," we have defined several parameters associated with either the agent, itself, or the space in which it sits. Any data that will need to be present throughout all the behavior and lifetime of the agent should be declared here. Also, anything declared here will be accessible to the probe machinery, and so will be capable of being manipulated and viewed from outside the agent.

Next come the message prototypes to which this agent will respond. And it is worth noting again that these are in addition to those declared in the SwarmObject superclass. So, not only will other objects be able to send messages to this agent that are declared here, but other objects will be able to send all the messages declared in the objectbase/SwarmObject.h imported previously. The messages prototyped here will dictate what the compiler thinks this object can respond to. Hence, if any part of any of these prototypes differs from the corresponding function definition in the Heatbug.m file, then the compiler will say something like Object: aHeatbug does not respond to xyz, where "xyz" is the name of the message that is being sent to the Heatbug. A script is provided with the Swarm distribution that fixes header file prototypes to match the message declarations in the corresponding ".m" file. This script should be in the $SWARMHOME/bin directory and is called m2h.

One more thing to notice about these prototypes is that some of them are duplicates of what appears in the objectbase/SwarmObject.h file. This means that when the message is called on a Heatbug object, it will execute the method defined here and not the one in the SwarmObject class. In the objectbase library, the following messages are intended to be overridden, as necessry: create:, createBegin:, createEnd, customizeBegin:, customizeEnd, customizeCopy:, describe:, and getInstanceName. Each of these messages do specific things that may change from subclass to subclass of SwarmObject. In this case, however, we're only overriding createEnd. The differences between we implement it in Heatbugs and the default is not that significant. But, it should be pointed out that when overriding certain messages, like createBegin: and createEnd, the new method should call the superclass' version of the message, as well. This is done using the default pointer to the superclass, designated super. The syntax in the Heatbugs case is:

```
[super createEnd];
```

The reasons for doing this are related to the object phase protocols used by defobj. If you would like more info on that, see Objective C - Swarm Style
(../../swarmObjC.html)
in the section entitled *The Create Phase In Principle*.

Finally, the `@end` keyword signifies the end of the interface definition. GNU Objective C allows one to leave this off; but, it is not good practice.

And that's it. Of course, there're a few tricky aspects to using the objectbase library that weren't mentioned here. Some of them will be mentioned in the Advanced Usage Notes
(AdvancedUsage.html)
and the Implementation Notes
(Implementation.html)
; but, the best way to learn is to examine the way the demo applications do it and try to make some changes yourself.

Subclassing from Swarm

Subclassing from the Swarm class works very similar to subclassing from SwarmObject.

ActivityControl

The ActivityControl object provides much more finely grained control over the execution of an interactive simulation. It addresses both the problems of not being able to stop the simulation at any given point in any given activity and provides an initial step towards a Swarm debugger.

An activity controller can be attached to any activity that is created in a Swarm simulation, including those that are created for use only by the Swarm kernel. The controller then provides the basic activity manipulation messages on that activity, which are: `run`, `stop`, `next`, `step`, `stepUntil`, and `terminate`.

The presence of the ActivityControl object might cause some confusion about what role the ControlPanel should play in the controlled execution of the various schedules. The ControlPanel should still be used for the top-level control of any simulation that is running in a context where random interference is expected (like under a GUISwarm where the user may click a button at any time). The reason this is true is because the ControlPanel sends pseudo-interrupts to the infinite loop we use to perpetuate execution of the top level Swarm (which can only be seen in the form of the `go` message on a GUISwarm at present). This type of control may change in the future! But, for now, it is how we monitor the control state of the simulation.

Now, having said that, the ControlPanel should no longer be used to run the simulation.

It should only be used to instantiate the control context and quit the entire simulation. That means that sometime in the future, the `Go` and the `Stop` buttons will be removed from the ControlPanel display. They have been left in for backwards compatibility so that applications that do not use the new ActivityControl will retain their (albeit handicapped) controllability. Also, the current `Time Step` button will be renamed to `Start` to be consistent with it's new purpose.

In order to use the new control mechanism, you must place code like the following in the top-level Swarm. (This code was taken from a modified mousetrap demo app.)

```
observerActCont = [ActivityControl createBegin: [self getZone]];
  observerActCont = [observerActCont createEnd];
  [observerActCont attachToActivity: [self getSwarmActivity]];
  [probeDisplayManager createProbeDisplayFor: observerActCont];
```

This creates an ActivityControl and attaches it to the top-level activity (in this case an `observerSwarm`). It also creates a display for the controller. (The probe map for the ActivityControl class is designed within the ActivityControl, itself. This is done because all of these objects are expected to look the same to any outside object.) With this activity controller, you will then be able to `run`, `stop`, `next`, `step`, `stepUntil`, and `terminate` that activity.

There are some tricky aspects to successfully using an ActivityControl object that the Advanced Usage Notes
(AdvancedUsage.html)
will cover.

(Interface.html)

(../objectbase.html)

(AdvancedUsage.html)

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-10

Objectbase Library: Advanced Usage Notes

1. ProbeMap design: When designing a ProbeMap for a given (subclass of) SwarmObject, inclusion of instance variables or messages defined in the super class might be desirable. The normal ProbeMap design code might look like (this code was taken from the tutorial app called "hello-world"):

```
probeMap = [CustomProbeMap createBegin: [self getZone]];
      [probeMap setProbedClass: [person class]];
      probeMap = [probeMap createEnd];
      [probeMap addProbe: [probeLibrary getProbeForVariable: "room"
 inClass: [person class]]];
      [probeMap addProbe: [probeLibrary getProbeForVariable: "party"
 inClass: [person class]]];
      [probeMap addProbe: [probeLibrary getProbeForVariable: "name"
 inClass: [person class]]];
      [probeMap addProbe: [probeLibrary getProbeForVariable: "stillhere"
 inClass: [person class]]];
      [probeMap addProbe: [probeLibrary getProbeForVariable: "listOfFriends"
 inClass: [person class]]];
      [probeMap addProbe: [probeLibrary getProbeForVariable: "myColor"
 inClass: [person class]]];
      [probeLibrary setProbeMap: probeMap For: [person class]];
      [probeDisplayManager createProbeDisplayFor: person];
```

where `room`, `party`, `name`, `stillhere`, `listOfFriends`, and `myColor` are instance variables declared in the interface to the Person subclass. And Person is a subclass of Agent2d, which is a subclass of SwarmObject.

Now let's say that there are two variables declared in Agent2d that you want to put into this custom probe in addition to the ones you've picked out of Person. Call them `x` and `y`. The way to add them to the `probeMap` is to add the following two lines of code to the above.

```
[probeMap addProbe: [probeLibrary getProbeForVariable: "x"
 inClass: [Agent2d class]]];
      [probeMap addProbe: [probeLibrary getProbeForVariable: "y"
 inClass: [Agent2d class]]];
```

And that's it! The two superclass-declared variables, which are, in fact, instance variables of the instance of the subclass, are now included in the probe.

In addition, a convenience message has been added to the CustomProbeMap interface to compress the above rather cluttered mechanism into one message. This convenience message can be used in the usual case where a ProbeMap will consist of variables and messages from the same class. For example, the first part of the custom probe creation above can be shortened to:

```
[probeLibrary create: [self getZone] forClass: [person class]
      withIdentifiers: "room", "party", "name", "stillhere",
```

```
                    "listOfFriends", "myColor", NULL];
```

And if the user wanted messages in the probe as well, it could be extended to:

```
probeMap = [CustomProbeMap create: [self getZone]
                         forClass: [person class]
                  withIdentifiers: "room", "party", "name",
                      "stillhere", "listOfFriends", "myColor",
                      ":",
                      "setWorld:Room:Party:",
                      "setPerson:Topic_array:ShowSpeech:",
                      NULL];
```

At present, this message doesn't search the superclasses for the message names listed here. But, that will soon be rectified.

2. ActivityControl Issues: It is completely reasonable to assume that explicit control can be had over all the activities in a simulation. However, at present, this control is limited because the context in which an activity runs determines how it behaves. To understand how an ActivityControl is to be used, we will have to explore the behavior of activities in context. (For a more complete explanation of the behavior of activities, see the activity library
(../activity.html)
documentation.)

There are two ways to get an activity started, one can activate the activity in `nil` or in some other activity. So called "top-level" activities are activated in `nil` and are intended to be independent of the sophisticated scheduling activity that dictates the execution of actions in any other context in the simulation. I.e. the only activities that should be activated in `nil` are those sets of events that are expected to preserve the same behavior no matter what goes on in any other part of the simulation.

The other type of activity, those activated in some other activity, is intended to be an integral part of its owner activity. However, this doesn't mean that it must depend on the outcome of actions happening in the owner activity. In fact, an ActionPlan can be designated as containing actions that are capable of being processed in parallel, via code like the following:

```
[anActionPlan setDefaultOrder: Concurrent];
```

But these activities are still intended to be meshed with their owner activities. In other words, they are part and parcel of the same model or simulation.

Now, the operational effect of activating an activity in `nil` is that it will not be meshed with the rest of the Swarm activity structure. This gives the user (or process) complete control over the execution of that activity. A `run` on that activity will run either to completion or until a stop flag is set by a sequence of events purely internal to that activity. Or, one can stop it from the outside with a message from something like an ActivityControl.

What all this means is that, while one can attach an ActivityControl to any activity, only the "top-level" activities (those having been activated in `nil`) are going to respond well to it. Any sub-activity will respond half-heartedly, if at all. For example, in the Mousetrap demo distributed with Swarm, an ActivityControl has been placed on both the ObserverSwarm and the ModelSwarm activities. Now, if one sends a `run` message to the ActivityControl that is attached to the `observerSwarm`'s activity, the entire model continues to run to completion, unless the user sends a `stop` message. However, if the sim is stopped at some point, a `run` message to the `modelSwarm`'s activity will have no effect at all. (Note: If you do this via the activity controllers, you see the `currentTime` variable get updated; but, the actual `run` message to the activity, itself, has no effect.)

So, the rule of thumb, for the present, is to attach ActivityControl objects only to "top-level" activities, like the ObserverSwarm.

(UsageGuide.html)

(../objectbase.html)

(InterfaceDesign.html)

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-10

Swarmobject Library: Interface Design Notes

Unfortunately, this interface has not undergone a rigorous design review. As such it is subject to change in the future. However, there are rumours that this library will be integrated into the defobj library
(../defobj.html)
anyway. So, even though little thought was given to the design of this interface (and it is not likely to be worthwhile designing a robust interface at this time), it was implemented in order to provide a first step towards bringing all the various libraries in line with the standard set by defobj.

Along these lines, a few notes are relevant.

1. Probes may become an inherent part of any object.
2. ActivityControls will become a part of a larger set of tools used for debugging Swarm models.
3. The explicit importation of the SwarmObject interface should be removed. This is only provided for backward compatibility and when all the libraries follow the interface conventions, then those conventions should be encouraged.

(AdvancedUsage.html)

(../objectbase.html)

(Implementation.html)

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-10

Swarmobject Library: Implemenation Notes

---

1. Right now, Probe's rely on a special method, `getInstanceName`, that has to be implemented in any probe-able object in order to get anything other than the class name of that object into the object designation widget. However, a more general capability has been added to defobj to give a meaningful name to any object. Probes will be changed to take advantage of this new capability.
2. In the ActivityControl the frequency of the message `updateStateVar` is very high. It is sent at least once every cycle and every time a message is sent to the ActivityCont This is unsatisfactory. Some of these messages can be pruned out of the object.
3. The `probeMap` designed for use with an ActivityControl was chosen fairly arbitrarily. Right now, it serves as a default for the class. A user can override it by designing a new one and inserting it into the `probeLibrary`.
4. Errors specific to objects in the objectbase library need to be gathered and initialized like those in the defobj library.

(InterfaceDesign.html)

(../objectbase.html)

(../objectbase.html)

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-10