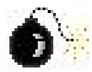
Objectbase Library

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)
  .

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

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:

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
```

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

Last modified: 1998-01-10

  - + 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)

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

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

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)

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.

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

- (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

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.

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

(ProbeMap.html)

(Interface.html)

(CompleteProbeMap.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;
```

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.

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",
```

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: 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