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.

Objectbase Library: Documentation and Implementation Status

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

Last modified: 1998-01-08

```
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

```
 (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:

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;

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

Last modified: 1998-01-10

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

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

(Interface.html)

(UsageGuide.html)

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

Last modified: 1998-01-10

(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

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-field.. 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.

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

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 messageprobes 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

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)
.

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

```
  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];
```

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

```
                        "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,

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