Last modified: 1998-01-08

Library Interface Conventions

Overview

Some of the basic Swarm libraries are implemented using object definition conventions established by the defobj library

(../defobj.html)
. This document explains how to read and interpret the header files and interface definitions published by such libraries. It also explains the typical structure of documentation provided for libraries that follow these conventions.

These conventions are documented under the following subsections:

- library header file
- object type definitions
- GridTurtle example
- global object symbols
- interface design conventions
- documentation structure

Library Header File

For a library that adopts these conventions, a library is not merely a collection of source files that are compiled into a library archive under control of a make file. Instead, Objective C source files in the library are processed in a special way to publish their definitions not only as header files, but as generated objects available at runtime to make full use of the library. A library processed in this way is referred to as as a "package." (Note: some of the newer Swarm libraries, such as objectbase and random, don't undergo this special processing and yet still follow all the library interface conventions described in this document.)

The defobj
(../defobj.html)
library documents the full details of special processing performed on a package. For simply using a library, the key fact to keep in mind is that the entire public interface to a library is declared in the one header file having the same name as the package itself, plus a trailing .h suffix. Additionally, the header file of a library normally documents only its public interface, in a way that is completely separated from the implementation of the objects it specifies.

syntax. A key difference from protocols is that object types are published as real external objects that may be used at runtime to create instances of a type. A futher difference is that the object types of the defobj library are divided into separate interfaces that define distinct phases of an object's life cycle.

GridTurtle example

An example will help illustrate the features of type definitions supported by the defobj library. Throughout documentation of the basic Swarm libraries, a series of running examples will be based on a simple type of object belonging to sample Swarm simulations. This type of object is an agent that move around on a two-dimensional grid, always moving in a current direction that it maintains internally. This agent is like a "turtle" of the original Logo system, except that its position is constrained to discrete integer values of its X-Y coordinates, and its direction is always one of the four orthogonal directions north, east, south, or west.

Following is a complete library header file for a library which defines such an object, called GridTurtle:

```
/*
Name:         GridTurtle.h
Description:  object type for Swarm example programs
Library:      grid
*/

#import <defobj.h>

@deftype GridTurtle <Create, Drop, CREATABLE>CREATING
- (void) setXLocMax: (int)xLocMax;
- (void) setYLocMax: (int)yLocMax;
SETTING
- (void) setXLoc: (int)xLoc;
- (void) setYLoc: (int)yLoc;
- (void) setXLoc: xLoc setYLoc: yLoc;

- (void) setDirection: direction;
USING
- (int) getXLoc;
- (int) getYLoc;

- getDirection;

- (void) move: (int)distance;
- (void) turn: (int)angle;  // angle measured in units of pi/2 (90 deg.)
- (void) print;
@end
```

automatically generated.

```
#import "grid.xt"
```

Global Object Symbols

The <grid.h> header file above also contains the declaration:

```
id <Symbol>  North, East, South, West;
```

Lines that declare global id variables of type Symbol, EventType, Warning, or Error (using the angle bracket syntax of id variables conforming to a protocol) are processed somewhat like deftype declarations in that they also produce global id variables initialized to support particular messages defined by defobj. These global variables, however, are not used to define or implement other message interfaces, but only to define certain fixed capabilities referenced through their global object names.

If declared as a Symbol, as in the case here, the generated objects have no particular behavior of their own (other than the character string of their name), but only serve to define unique global id constants which may be used as distinct named values in messages. In this example, the current direction of a GridTurtle object is represented by one of the symbol names North, East, South, or West. These values are like the enum constants of the C language, except that they are defined as full Objective C objects, and may be used with further restrictions. An EventType, or a Warning or Error, defines a further subtype of a Symbol constant with further specialized messages documented in defobj.

Interface Design Conventions

A variety of rules on naming and declaration of object types, symbols, and messages are followed by many of the Swarm libraries. These rules help establish a basic consistency on the library interfaces. Following is a list of such conventions that apply to a public library interface, some but not all of which are derived from standard Smalltalk or Objective C coding practice:

- Names of global object constants are capitalized. In the public interface, such names include types and symbols.
- Recapitalization separates words of a compound name (e.g., GridTurtle). Underscores are generally not used.
- Message names start with a lower-case character, and are named using verbs. Nouns that represent gettable or settable components of object state (e.g., Direction of a GridTurtle), are prefixed by get or set to indicate the action being performed.
- The Smalltalk convention that a message return the receiver of a message if there

also discuss tradeoffs considered along with references to related or supporting work.

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

Last modified: 1998-01-08

The key requirement for Swarm is to be able to make all these operations happen to any object at any time, whenever they're supposed to occur within some simulated world. It stores these operations inside its own data structures, and when you run a simulation in Swarm, the Swarm system itself traverses these data structures to make the necessary operations happen at the proper time.

Swarm needs to be able to make any kind of action happen to any kind of object at any time, and to do this it needs very general-purpose structures that hold some kind of representation of the actions themselves. That's a special requirement that not every object language provides. You already have enough work to define the kinds of objects that can exist inside your model, and Swarm doesn't want you to have to define an even larger number of objects for every kind of event that might occur to your objects. Instead, it wants the object language to provide it with a representation of the operations on objects that you've already defined. Swarm can then store these representations in its own data structures, and make the operations happen whenever it needs to.

In Swarm, the representation of events that happen on objects are just as fundamental to the model as the objects themselves. That's why Swarm is a "discrete-event" simulation system. But if the object system doesn't provide a general-purpose enough representation of events as well as objects, there would be a lot more work to do. The Objective C system, through its special data type called a "message selector," provides a representation of operations that is flexible enough to do this, but the C++ language does not. (In C++, the compiler requires more information about an operation than the event structures in Swarm would be able to provide.)

There's a host of additional reasons why Objective C has also been a good match for the requirements of the Swarm system. Like C++, the operations on objects can be compiled to a very efficient form (though Objective C does require a little more overhead than C++ to get the operation started). This efficiency can be very important for a simulation, since simulations can run for very long periods of time to explore all the behavior that might occur within their simulated worlds. Other languages, such as Lisp, Smalltalk, and Java, also have the "dynamic message dispatch" feature that would make general-purpose event structures possible, but they still carry significant added overhead compared to C.

Objective C is also a very simple extension to the C language. Basic knowledge of C is already widespread, and a few days are typically all that is needed to learn the few additions of Objective C. (Really learning the concepts of objects, however, can take much longer, no matter what object language you try to use.)

Objective C was also a good choice for Swarm because it has a high quality, freely distributable implementation in the GNU C compiler. One of the main concerns about Objective C is that it isn't nearly as widely known or used as other languages like C++, but at least the GNU C compiler assures it will be available on machines where Swarm needs to run. The OpenStep system of the NeXT corporation, and the parallel GNUStep project, also help assure that Objective C is a living language.

There are even more powerful aspects of Objective C that Swarm takes advantage of. Some of these are described in the Advanced Usage Guide
(adv.html)

Some of these rules and guidelines don't even require any specific software to implement them, but are just conventions and recommended style that anyone programming in Objective C can follow when their goals are similar to those of Swarm, and to a large extent often do. These kinds of conventions are the focus of this section; following sections discuss aspects of Swarm Objective C style that depend on specific software support.

Lots more to come... For now, see Library Interface Conventions (library.html) for some of this information, and Swarm tutorials for much of the rest.

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

Last modified: 1998-01-08

Defobj Library: Interface Reference

Dependencies

The defobj library is defined and documented using the library interface conventions

(library.html)
established by the defobj library. It imports the Object superclass and other standard type definitions of the GNU Objective C runtime system.

> The collections library must always be linked along with the defobj library. Even though the interface definitions of defobj do not depend directly on collections, the implementations of these libraries both require the presence of the other. Only the collections library should be initialized directly; initialization of the collections library automatically initializes the defobj library as well.

Contents

Name

DefinedObject – standard support for all objects

Synopsis

```
//
// DefinedObject -- object with defined type and implementation
//
@deftype DefinedObject
- getZone;
- (BOOL) respondsTo: (SEL)aSel;
- getClass;

- (ref_t) addRef: (notify_t)notifyFunction withArgument: (void *)arg;
- (void) removeRef: (ref_t)refVal;

- (int) compare: anObject;

- perform: (SEL)aSel;
- perform: (SEL)aSel with: anObject1;
- perform: (SEL)aSel with: anObject1 with: anObj2;
- perform: (SEL)aSel with: anObject1 with: anObj2 with: anObj3;

- (void) setDisplayName: (const char *)displayName;
- (const char *)getDisplayName;

- (void) describe: outputCharStream;
- (void) xprint;
- (void) xprintid;
@end
```

Description

DefinedObject is the top-level supertype for all objects that follow the object programming conventions of the defobj library. The messages defined by this type are the only messages which should be assumed to be automatically available on objects that follow these conventions. In particular, use of messages defined by the Object superclass of the GNU Objective C runtime should not generally be assumed because future implementations of some objects might not give continued access to them.

from Object unnecessary. They are currently the only messages defined in Object that are recommended for use in Swarm.

```
- (void) setDisplayName: (const char *)displayName;
- (const char *)getDisplayName;
```

These messages assign a character string as a name that identifies an object for display or debug purposes. A global function, xsetname(), is also available to set the name by calling the function from the debugger.

```
- (void) describe: outputCharStream;
- (void) xprint;
- (void) xprintid;
```

The describe: message prints a brief description of the object for debug purposes to the object passed as its argument. The object passed as the outputCharStream argument must accept a catC: message as defined in String and OutputStream in the collections library. Particular object types may generate object description strings with additional information beyond the built-in default, which is just to print the hex value of the object id pointer along with the name of its class, and the display name of the object, if any. xprint prints the standard describe string to standard output; it is also called by the xprint function that may be called from the debugger. xprintid prints just a one-line describe string, consisting of the built-in default.

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

Last modified: 1998-01-08

) for standard ways in which create messages that combine several steps can be combined.

Inherited behavior

The Create type has no inherited behavior beyond the universal messages of the DefinedObject

(DefinedObject.html)
type.

Defined behavior

```
+ create: aZone;
```

The create: message creates a new instance of a type with default options. The zone argument specifies the source of storage for the new object. The receiving object of this message is a previously defined type object. The message is declared as a class message (with a + declaration tag) to indicate that the message is accepted only by the type object itself rather than an already created instance of the type (which a - declaration tag otherwise defines).

The create: message returns the new object just created.

Defined behavior

```
+ create: aZone;
```

The create: message creates a new instance of a type with default options. The zone argument specifies the source of storage for the new object. The receiving object of this message is a previously defined type object. The message is identified as a class message (a + declaration tag) to indicate that the message is accepted only by the type object itself rather than an already created instance of the type, which most messages of a type (those with a - declaration tag) otherwise define.

The create: message returns the new object just created. This object is an instance of some class selected to implement the type. The class which a type selects to implement an object may be obtained by the getClass message, but is not otherwise visible to the calling program. A caller never refers to any class name when creating objects using these messages, only to type names, which are automatically published as global constants from any @deftype declaration.

```
+ createBegin: aZone;
- createEnd;
```

The createBegin: and createEnd messages bracket a series of messages that specify options for an object being created. The intermediate messages can set values defined as parameters of the type, or provide other forms of specification using available messages. A particular object type defines the specific messages that are valid for sending during this interim creation phase.

objects all need the same create-time options, it is often simpler (and can also be faster) to create a customized version of a type first, and then create further instances from that type.

Customizing a type object does not modify the original type object, but instead creates a new type object that has the customizations built-in. A create: message on the new type object creates a new instance as if the same sequence of create-time messages had been sent to the original type object using createBegin: and createEnd. A type is customized by bracketing the sequence of create-time messages not with the createBegin: and createEnd messages used to create a new instance, but with customizeBegin: and customizeEnd messages instead. customizeBegin: returns an interim value for receiving create-time messages much like createBegin:, and customizeEnd returns the new, customized version of the original type. Following is sample code sequence that illustrates the use of these messages:

```
newArrayType = [Array customizeBegin: aZone];
  [newArrayType setCount: 100];
  newArrayType = [newArrayType customizeEnd];
  array1 = [newArrayType create: aZone];
  array2 = [newArrayType create: aZone];
  ... // [array1 getCount] and [array2 getCount] are both 100
```

The customizeCopy: message creates a new copy of the interim object returned by customizeBegin: which may be used for further customizations that do not affect the customization already in progress. It may be used to branch off a path of a customization in progress to create an alternate final customization. Following is a code fragrment which illustrates this usage:

```
newArrayType1 = [Array customizeBegin: aZone];
  [newArrayType1 setCount: 100];
  newArrayType2 = [newArrayType2 customizeCopy: aZone];
  [newArrayType2 setDefaultMember: UnsetMember];

  newArrayType1 = [newArrayType1 customizeEnd];
  newArrayType2 = [newArrayType2 customizeEnd];
  array1 = [newArrayType1 create: aZone];  // no DefaultMember option
  array2 = [newArrayType create: aZone];   // DefaultMember option set
```

customizeCopy may be used only on an interim object returned by customizeBegin: and not yet finalized by customizeEnd. The new version of the interim object being customized may be allocated in the same or different zone as the original version, using the zone argument required by customizeCopy:

The zone passed to customizeBegin: is the same zone from which storage for the new, finalized type object will be taken. This zone need not be the same as any instance later created from that type, since a new zone argument is still passed in any subsequent create message on that type.

Whether a customized version of a type can be created depends on the implementation of the type itself. If a type does not support customization, a customizeBegin: message on the type raises an error. All types defined by an @deftype declaration may be relied on to support at least one cycle of customization to create a new type object. Whether an already customized type object (returned by customizeEnd) supports a further

Name

Drop – end existence of an object

Synopsis

```
//
// Drop -- end existence of an object
//
@deftype Drop
- (void) drop;
@end
```

Description

The Drop supertype defines the drop message, which is a standard message for indicating that an object no longer exists and will never again be referenced. Any future attempt to reference a dropped object is an error. This error may or not produce predictable effects depending on the level of debug checking and other factors.

Immediate effects of the drop message depends on the subtype of Zone
(Zone.html)
used to provide storage for the object. For some zone types, the drop message immediately deallocates storage for the object and makes the freed storage available for other use. Subsequent use could include the allocation of a new object at precisely the same location, resulting in a new object id identical to a previously dropped one. Other zone types (..none implemented yet..) might delay any reclamation of freed storage until a more comprehensive check can be made that other references no longer exist to the dropped object. When running in debug mode (see debug support
(globaldef.html#debug)
) the storage for a dropped object is filled with predictable contents so that an error is raised if there is any immediate attempt to us it again.

The Drop type may be inherited by any type that provides drop support for its instances. In addition to freeing the storage and invalidating the object, a drop message may release other resources acquired or held within the object. Not every object which can be created can also be dropped, and some objects can be dropped which are not directly creatable. Some objects may be created as a side effect of other operations and still be droppable, and some objects may be created with links to other objects and not droppable on their own. A type independently inherits Create or Drop types, or both, to indicate its support of these standard interfaces to define the endpoints

Name

Zone – modular unit of storage allocation

Synopsis

```
//
// Zone -- modular unit of storage allocation
//
@deftype Zone <Create, Drop, GetOwner, CREATABLE>
CREATING
- (void) setReclaimPolicy: reclaimPolicy;
- (void) setStackedSubzones: (BOOL)stackedSubzones;
- (void) setPageSize: (int)pageSize;
USING
- getReclaimPolicy;
- (BOOL) getStackedSubzones;
- (int) getPageSize;

- allocIVars: aClass;
- copyIVars: anObject;
- (void) freeIVars: anObject;

- allocIVarsComponent: aClass;
- copyIVarsComponent: anObject;
- (void) freeIVarsComponent: anObject;

- getComponentZone;

- (void *) alloc: (size_t)size;
- (void) free: (void *)aBlock;

- (void *) allocBlock: (size_t)size;
- (void) freeBlock: (void *)aBlock blockSize: (size_t)size;

-                getPopulation;
- getSubzones;
- (void) mergeWithOwner;

- (BOOL) containsAlloc: (void *)alloc;
- getSubzone: (void *)alloc;

- (void) reclaimStorage;
- (void) releaseStorage;
```

```
@deftype Zone <Create, Drop, CREATABLE>
```

A zone is created using standard create messages just like other objects. This means that a zone must identify another zone from which it obtains its storage. Storage is typically obtained from this other zone in large units called pages, which are then managed by the local zone to support internal allocations. The getZone message of the Drop
(Drop.html)
type returns the zone which provides these base pages.

Since a new zone always requires that an existing zone be identified, no new zones could be created unless there were some zones that already existed. Several such zones are predefined as part of the defobj library; see predefined zones
(globaldef.html#zones)
for a summary.

Create-time behavior

```
- (void) setReclaimPolicy: reclaimPolicy;
- (void) setStackedSubzones: (BOOL)stackedSubzones;
- (void) setPageSize: (int)pageSize;
USING
- getReclaimPolicy;
- (BOOL) getStackedSubzones;
- (int) getPageSize;
```

ReclaimPolicy

(.. Not implemented yet. Will define the various types of zone implementations, all the way to zones that support full automatic storage reclamation. ..)

StackedSubzones

(.. Not implemented yet. Specifies that no further allocations are made in the zone after a subzone is created, until that subzone itself is dropped. Supports an especially efficient of implementation. ..)

PageSize

PageSize specifies the size of pages within which a zone manages its internal allocation. Its default is typically a natural page size (perhaps 4K) for the local machine architecture. The default should be overridden only when tuning storage allocation for very specific situations. Allocations within a zone are not limited to the page size, since any requests that exceed the page size are simply passed up to the owner zone within which the zone was allocated.

Object allocation

```
- (void) freeBlock: (void *)aBlock blockSize: (size_t)size;
```

alloc: allocates a new storage block much like the malloc function of the C library. The storage is aligned according to the most restrictive requirements for any data type on the local machine architecture. The storage is not initialized to any known contents.

free: releases a block of storage previously allocated using alloc:. The size of the block is not required as an argument because alloc: has saved this size as necessary as part of the initial allocation. free: may be used only to free a block allocated by alloc:, and a block allocated by alloc: may be freed only by free:.

allocBlock: allocates a new storage block similar to alloc:, except that the size of the block allocated must be passed as an argument when freeing the block. copyBlock:blockS allocates a new storage with the same requirements on free as allocBlock:, but also fills the new storage with the contents of a block passed as the first argument, up to the size of the newly allocated block. freeBlock:blockSize: must be used to free any block previously allocated by allocBlock: or copyBlock:blockSize.

Subzone structure

```
-              getPopulation;
- getSubzones;
- (void) mergeWithOwner;

- (BOOL) containsAlloc: (void *)alloc;
- getSubzone: (void *)alloc;
```

getPopulation returns a collection all objects allocated in a zone using either allocIVars: or copyIVars: and not yet freed using freeIVars:. getObjects returns nil if the ObjectCollection option is false. The collection returned has the type OrderedSet
(../collections/OrderedSet.html)
as defined in the collections library
(../collections.html)
, with the ReadOnly option set true and the IndexSafety option set to SafeAlways. The members of this collection may change as objects are allocated and freed, but may not added or removed directly within the collection.

getSubzones returns a collection of other zones that all obtain their storage from the local zone. mergeWithOwner drops the local zone, but reassigns all the zone allocations to the zone from which all local storage was obtained. (..neither message currently implemented..)

containsAlloc: tests if a particular allocation was made by the local zone, or any zone which obtains storage from that zone. The pointer argument must point to an allocation previously made by any zone. The message returns true if the allocation was made by either the local zone or any of its subzones. getSubzone is similar, except that it returns the particular zone in which the pointer was allocated, or nil if the pointer was not allocated in the local zone or one of its subzones. (..neither message currently implemented..)

Name

Symbol – object defined as a distinct global id constant

Synopsis

```
//
// Symbol -- object defined as a distinct global id constant
//
@deftype Symbol <Create, GetName, CREATABLE>
CREATING
+ create: aZone setName: (const char *)name;
- (void) setName: (const char *)name;
@end
```

Description

A Symbol is an object created with a fixed name. It has no behavior except to get the name with which it was created. A Symbol is typically used to define unique id values which are assigned to global constant names. These names, capitalized according to the recommended convention for global object constants, are used by some libraries as flags or enumerated value codes in arguments or return values of messages.

Ordinarily, a symbol is created with its character string name matching the global id constant to which it is assigned. These global program constants can then provide a minimal level of self documentation as objects. Subtypes of Symbol can extend the base of a named, global id constant to establish further components of a global, constant definition.

Inherited behavior

A symbol is fully creatable using standard Create messages. A character string name must be supplied for any new symbol; there is no default. Symbol inherits the getName

(misc.html#GetName)
message, which returns the symbol name.

Name

Warning, Error – report of some condition occurring during program execution

```
(..  This section of documentation is only a start.)
```

Synopsis

```
//
// EventType -- report of some condition occurring during program execution
//
@deftype EventType <Symbol, RecordType>
- (void) raiseEvent;
- (void) raiseEvent: (void *)eventData, ...;
@end

//
// Warning -- a condition of possible concern to a program developer
//
@deftype Warning <EventType, CREATABLE>
- (void) setMessageString: (const char *)messageString;
- (const char *)getMessageString;
@end

//
// Error -- a condition which prevents further execution
//
@deftype Error <Warning, CREATABLE>
@end
```

Description

Warning and Event are object types (both subtypes of EventType) that may be used to report conditions detected during program execution. A warning is a condition of possible interest to a developer or user, but one that does not prevent further execution. An error is a condition which prevents further execution of the program.

Warning and Event also inherit from Symbol, so that each type of warning or error may be given a unique global name. Any library may publish its own warning and event names as part of its documented library interface (see module definition conventions

```
  OutOfMemory,             // no more memory available for allocation
  InvalidAllocSize,        // no more memory available for allocation
  InternalError,           // unexpected condition encountered in program
  BlockedObjectAlloc,      // method from Object with invalid allocation
  BlockedObjectUsage;      // method inherited from Object superclass

//
// standard warnings
//
extern id <Warning>
  WarningMessage,          // message in the source defines warning
  LibraryUsage,            // invalid usage of library interface
  DefaultAssumed,          // non-silent use of default
  ObsoleteMessage,         // using feature which could be removed in future
  ObsoleteFeature,         // using feature which could be removed in future
  ResourceAvailability;    // resource from runtime environment not available
```

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

Last modified: 1997-01-08

```
@end
```

Description

The copy message creates a new object that has the same contents and resulting behavior as a starting object, except that an independent copy of the contents of the starting object is created so that further changes to one object do not affect the other. The zone argument specifies the source of storage for the new object. The message returns the id of the new object created.

> An object type that supplies the copy operation defines what it includes as the contents of an object copied. There is no global rule for what is considered "inside" a copied object vs. merely referenced by it. (There is no fixed notion of "shallow" vs. "deep" copy found in some object libraries.) After copying, the new object may still contain some references to other elements also referenced by the starting object, but in general the new object minimizes any dependencies shared with the starting object. Any object type supplying the copy message should also supply documentation on its rules for copied objects.

GetName Interface Description

Synopsis

```
//
// GetName -- get name which identifies object in its context of use
//
@deftype GetName
- (const char *) getName;
@end
```

Description

The getName message returns a null-terminated character string that identifies an object in some context of use. This message is commonly used for objects that are created once in some fixed context where they are also assigned a unique name. Constant objects defined as part of a program or library are examples. This message is intended only for returning a name associated with an object throughout its lifetime. It does not return any data that ever changes.

GetOwner Interface Description

Description

The SetInitialValue type defines a variety of messages relating to an initial or unmodifiable value established as part of an object. This message is typically provided when creation of a new object might be more easily accomplished by copying the value of an existing object rather than establishing a new value from scratch. As with the copy message, precisely what is considered the value of an existing object to copy is defined only by the particular object type that supplies these messages.

If an object has a value which can be established at create time, it is often useful (and can also enable significant optimization) to declare that no further modification will occur to this value during further use of the object. A restriction against modifying a value is referred to as a "read-only" restriction. This type supplies messages to declare a read-only restriction along with any initial value. For some object types, a read-only restriction can also be added or removed after an object has already been created.

The setInitialValue: message requires another object as its argument, from which the value of a newly created object is to be taken. Unlike a copy message, the object used as the source of the new value need not have the identical type as the new object to be created. A particular object type defines the types of initial value objects which it can accept, along with any special conversion or interpretation it might apply to such a value.

The ReadOnly flag specifies that the value of an object may not be modified further by other messages on an object. If such messages are attempted then an error is raised. If the ReadOnly flag is set at create time, the object is first created in read-only mode. Typically an initial value is always provided for an initial read-only object, since otherwise the created object would have only an empty or initial default value. The setReadOnlyValue: message is equivalent to a combination of a setInitialValue: message with an added ReadOnly restriction.

If a read-only restriction is established at create time, then for some object types this restriction may be required to remain for the entire lifetime of an object. For other object types, the read-only restriction may be added or removed any time after initial object creation. If a particular type does not support a change to read-only status, a request to do so raises an error.

Standard forms of create combination messages
(combo.html)
are provided to create an object in one message expression from the value of another object, with or without a read-only restriction.

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

Last modified: 1998-01-08

Even though classes are often thought of as separating the interface of an object from its implementation, this separation is far from complete. Not only do classes typically contain many internal methods not intended for external use, but they also define a particular storage format for an object defined as instance variables.

A library module instead publishes its interface as a set of public object types. These object types may also be supplemented by global object constants called symbols. Both these kinds of definitions normally appear only in the header file of a library. The remaining source files in a library normally contain the classes which implement the object types.

In a library module, the files which implement classes (including class header files), need not ever be referenced simply to make use of the implemented capabilities of the library object types. Documentation for the library is normally expressed entirely in terms of the types and symbols published in the library header file. If a feature does not appear in the library header file, it should not be considered part of a supported public interface.

Individual class header files are required to subclass from existing implementations, but interfaces for subclassing are an entirely separate issue from normal public use of an object library. Class inheritance can be a powerful implementation technique, but extension of an existing class framework is typically safe only if performed in explicitly permitted ways. If a library supports subclassing at all, it must carefully state which classes may be subclassed and in what ways. For a library module, this information is supplied outside the library header file. The library header file specifies only the interfaces by which objects are intended to be used, whether implemented by a local class or an external subclass.

Remaining sections of this document explain the declarations which appear in a library header file, and end with a suggested structure of documentation to be provided for a library module. The library modules of Swarm mostly follow this structure.

Object Type Definitions

The interfaces to objects defined by a library module are specified by object type definitions. An object type defines only a set of messages which may be sent to an object over various phases of its lifetime. An object type makes no commitment to the classes that might be used to implement the object.

Multiple classes may all implement the same messages belonging to a type. The independence of types and classes means that different classes can provide alternate implementations of the same object type. For example, a particular implementing class might be selected to optimize the implementation for a particular case.

Object types are similar to protocols defined by Objective C, and the declarations appearing in a library header file are a minor adaptation of Objective C protocol syntax. A key difference from protocols is that object types are published as real external objects that may be used at runtime to create instances of a type. A futher

```
#import "grid.xt"
```

An object type is defined by an @deftype declaration. The syntax of such declaration is identical to that of an Objective C @protocol definition, except for the entirely uppercase keywords (CREATABLE, CREATING, SETTING, USING) appearing in the GridTurtle example above. All these modifications of Objective C syntax are accomplished by simple preprocessor macros; no extensions to the language compiler are involved.

When this library header file is processed (by a special rule in a make file), an external object id with the name GridTurtle is automatically published. The name of a defined type becomes an ordinary object that accepts specific messages defined by the defobj library. The defobj library
(../defobj.html)
explains the details of such messages; the only purpose here is to explain the basic sections of a deftype declaration.

deftype declarations follow the syntax as Objective C protocols for inheriting messages from each other: a list of names enclosed in angle brackets (e.g., <Create, Drop, ...> above) gives the names of other declared types containing messages to be supported by the new type as well. (These types referenced here are defined by the imported file <defobj.h>.) Like protocols, full multiple inheritance of types is supported. The same messages may be inherited any number of times through any path with no different effect than if inherited or declared just once, so long as no conflicts occur in any of their argument or return types.

The CREATABLE tag appearing in the inherited type list above is a special type which defines no messages of its own, but merely marks the type as one which supports direct creation of instances of the type. Without this tag, the only role of a type is to define messages for inheritance by other types. With this tag, the global type object has a particular implementation that supports object creation using standard messages defined in defobj.

The declared messages of the type may be separated into sections marked by the special uppercase tags such as CREATING, SETTING, and USING above. (Currently, these are the only such tags which may occur.) These sections each define messages belonging to a particular defined "interface" of the object type, which are further combined into distinct "phases" of an object lifecycle supported by defobj messages. Further explanation of the interfaces and phases defined by this example are provided in the Usage Guide
(guide.html)
of the defobj library.

The following line at the end of the <grid.h> header file above is required as part of the standard conventions for coding a module header file: This line adds no information to the header file, but is required as part of the system by which definitions are automatically generated.

```
#import "grid.xt"
```

Documentation Structure

The following standard sections of documentation are suggested for libraries that follow the strict interface vs. implementation separation of library modules, each with purpose and typical contents as given:

- Interface Reference
  Complete, concise summary of all features of a library. Fills the role a Unix "man page" in providing comprehensive definition of library services. Little or no concern to provide path of tutorial introduction. Does not contain extended examples, at most only fragments of examples which serve needs of specification.
- Usage Guide
  Tutorial introduction to a library. Focuses on the most common uses in an order reflecting the needs of a first-time user. Most explanation by means of progressively elaborated examples. Serves as a guided tour of major library capabilities. No attempt at reference-style completeness.
- Advanced Usage Notes
  Continues overview of all significant capabilities of a library, including those which might be needed by advanced users customizing or extending built-in capability. Provides examples of specialized uses extending beyond normal, basic usage, but still using built-in features of the library framework. Helps to simplify the Usage Guide by providing a location for overflow of more advanced features.
- Subclassing Notes
  Documents the rules for writing new classes which subclass from classes that implement the library. Because a library might use complex combinations of classes to implement the range of behaviors defined by its types, the classes in a library are not automatically usable as superclasses of user-defined classes. Each library documents which classes are available for use as superclasses, and the specific rules that must be followed when subclassing from these classes.
- Interface Design Notes
  Explains why choices were made the way they were in the design of the public interface to a library, including other alternatives considered. Serves as a "Rationale" document for the documented interface. May include historical background deliberately left out of the preceding sections. May also include references to other work.
- Implementation Notes
  Explains the structure of classes by which the types of a library are implemented. Summarizes the status of implementation if still incomplete and lists items of possible future work. Provides overview and high-level structure of the implementation in whatever ways would best guide a reader of implementation source code. May also discuss tradeoffs considered along with references to related or supporting work.

Marcus G. Daniels <mgd@santafe.edu>

Defobj Library: Create Combination Messages

The create phase of an object type may contain messages that are defined simply by identifying them as a combination of other defined messages. Such messages are referred to as "create combination messages." Typically, such messages have the following form:

```
-               create: aZone setOption: optionValue;
```

For this message to be a valid create combination message, the object type must already have the create: and setOption: messages defined (either locally or by inheritance). The meaning of this message is then identical to the following sequence (where SomeType is the name of a type implementing the combination message):

```
-   anObject = [SomeType createBegin: aZone];
-   [anObject setOption: optionValue];
-   anObject = [anObject createEnd];
```

Create combination messages are provided as a convenience for commonly used options. Some combination messages also have their own custom implementing methods so that creation is faster when using the combination message. Any such implementation, however, is required to implement the parts of the combination message so that its meaning is identical to the results that would be obtained if the separate parts were requested in separate message.

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

Last modified: 1998-01-08

Defobj Library: Global Definitions

Contents

The following global definitions are documented below:

M() macro

Synopsis

```
//
// M() -- macro to abbreviate @selector()
//
#define M( messageName )  @selector( messageName )
```

The M() macro is currently nothing more than an abbreviation for the Objective C syntax used to obtain a selector for a particular message name. It is provided as a convenience for use in messages that combine a selector with other arguments.

---

func˙t type definition

Synopsis

```
//
// func_t -- function pointer type
//
typedef void (*func_t)( void );
```

Output Files

Synopsis

```
extern FILE *_obj_xerror;     // output file for error messages
extern FILE *_obj_xdebug;     // output file for debugging messages
```

The file pointers _obj_xerror and _obj_xerror define standard output files used for error messages and debugging messages, respectively. If not set otherwise by the time of defobj module initialization they both default to the standard error output defined by the C library.

Debug Support

Synopsis

```
extern BOOL _obj_debug;       // if true then perform all debug error check-
ing

extern void xsetname (id anObject, const char *name);  // debug set dis-
play name

extern void xprint (id anObject);               // debug object print
extern void xprintid (id anObject);             // debug object id print

extern void xfprint (id anObject);              // debug foreach object print
extern void xfprintid (id anObject);            // debug foreach id print

extern void xexec (id anObject, const char *name);  // debug method exec
extern void xfexec (id anObject, const char *name); // debug fore-
ach method exec
```

The _obj_debug variable enables debug-level error checking in libraries that support optional levels of checking. Its default value is true. The default may be overridden either before initialization of the defobj module, or by means of a compiler macro setting (-D_obj_debug=0). If set to zero by a compiler macro setting, optimized compilation of code that refers to it will see only the constant zero and so avoid even generating code to test the variable.

The xsetname function sets the display name for an object using its setDisplayName: message. This string may be used to identify the object in a user-meaningful way; if defined it will appear in the identification string generated by a standard describe: message.

Defobj Library: Subclassing Reference

The defobj library currently includes the CreateDrop and Object_s superclasses used by other libraries. For user objects in Swarm simulations the SwarmObject superclass in the objectbase
(../objectbase.html)
library packages all required behavior of these superclasses into a single, simpler superclass.

The defobj library also provides support for custom-generated classes, such as those which implement separate phases of the standard create protocol. (See generated phase classes
(adv.html#phase-classes)
in the Advanced Usage Guide
(adv.html)
.) Subclassing from custom-generated classes is still not officially supported, because the framework for custom class generation is still being finalized. Custom-generated classes are currently used only in the defobj, collections, and activity libraries; these libraries document the specific classes that provide subclassing support.

In general, classes that implement a library are not automatically available for use as superclasses. Any library must document those specific superclasses that are valid for user classes to subclass from, along all the rules that a user subclass must follow when implementing new behavior.

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

Last modified: 1998-01-08

Defobj Library: Implementation Notes

(This section of documentation is not yet available.)

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

Last modified: 1998-01-08