Defobj Library

Documentation and Implementation Status

Overview

The defobj library supports the style of object-oriented programming that is used throughout Swarm. It defines a specific style for using the Objective C language that includes its own standard conventions for creating objects and for storage allocation, error handling, and debugging support.

Documentation

The defobj library follows the Swarm `library interface conventions`

`(defobj/library.html)`
. It has following standard sections of documentation:

- `Usage Guide`
`(defobj/guide.html)`
- `Advanced Usage Guide`
`(defobj/adv.html)`
- `Interface Reference`
`(defobj/ref.html)`
- `Subclassing Reference`
`(defobj/subclass.html)`
- `Interface Design Notes`
`(defobj/design.html)`
- `Implementation Notes`
`(defobj/impl.html)`

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

Defobj Library: Documentation and Implementation Status

The current interfaces defined in defobj are expected to remain stable, with the exception of those relating to package structure and custom-generated classes. Those facilities are in process of being totally replaced, and will not be documented further until that replacement is complete. Details of support for defining a customized type will also be provided later.

All the interfaces defined in the header file are currently implemented, except for the various types of zones that would hold all allocated storage in a collection of local pages, and that would provide automatic reclaim of unused storage. The current zone implementation supports all defined allocation messages, and also maintains the population of the zone that consists of all objects created directly within it.

The Usage Guide section is only a start, but it does include subsections that discuss the style of Objective C programming adopted for use throughout Swarm. There is a full set of Interface Reference sections, though some details remain to be filled in.

Throughout the documentation, a parenthesized comment that starts with (.. indicates an editorial comment on the current status of implementation or documentation.

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

Last modified: 1998-01-08

The separation of implementation means that a library publishes its interface entirely without reference to any Objective C classes which implement its objects. 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 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, 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 package, 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. The libraries of Swarm mostly follow this structure.

Object Type Definitions

The interfaces to objects defined by a library 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

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

#import "grid.xt"
```

An object type is defined by an @deftype declaration. (Note: newer libraries, including objectbase and random, now follow the library interface conventions without using this special @deftype tag. Instead they use just an ordinary @protocol declaration, but otherwise they follow all the structure explained in this document.) 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 library header file: This line adds no information to the header file, but is required as part of the system by which definitions are is no other specific return value is generally *not* followed. If there is no specific return value needed from a message the return type is declared (void).

Documentation Structure

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

- 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 Guide
  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.
- 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.
- Subclassing Reference
  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

Defobj Library: Usage Guide

Contents

Why Swarm uses Objective C

Swarm uses object-oriented programming not only because this is a good way to build general-purpose software libraries, but because the very concept of an object is at the base of how you build a model in Swarm. To build a simulation in Swarm, the first step to define the various kinds of objects that can inhabit some real or artificial world, and the second step is to define the kinds of events that can occur to these objects, including all the different ways that the objects can interact with each other.

> The basic concept of an object is that it responds to external events, and that the only thing that really matters about an object are the ways it can be observed responding to these events. In an object-oriented programming system, an object is represented by some uniquely identified chunk of data, and the events are the dynamic operations that can be made to occur on these objects.
>
> Different object-oriented programming systems define the operations that occur on objects in different ways. For example, in C++ the operations are called "member functions" and look much like an ordinary function call in the C language. In Objective C, an operation on an object is called a "message" and has a special syntax by which you call it, but a message also has arguments and behaves in many respects like a call to a function.

of the defobj library. But many others are at the heart of how Swarm uses Objective C, and so are dealt with in the rest of this Usage Guide. The entire purpose of the defobj library is to define a standard style for the use of Objective C in Swarm. This style is backed up by a library of foundation classes that Swarm provides to support this style.

Swarm provides its own foundation classes even for such basic operations as creating an object, and other support that user classes ordinarily receive from a builtin Object superclass. A good understanding of the defobj library is essential for Objective C programming in Swarm. Like any programming language, Objective C requires learning not only the rules of the language itself, but also a standard library of initial capabilities that you build from. Objective C doesn't really have a single official standard library (or language definition either, for that matter), but the NextStep foundation libraries (and the NextStep language definition) come very close to this status.

For a variety of reasons explained here and in the Advanced Usage Guide
(adv.html)
, Swarm doesn't use a standard library based on the NextStep foundation interface. You can still learn the Objective C language from other available sources, but you have to be careful to sort out the language level from the standard object and library support they also discuss.

The index page
(../../index.html)
of the Swarm documentation provides a variety of links to other Objective C resources. In particular, it provides a link to a complete on-line reference for the Objective C language as defined by NextStep and implemented by the GNU C compiler. None of the Swarm documentation attempts to duplicate this coverage of the basic language; Swarm assumes that you learn Objective C from other available sources. The NextStep reference(*Object-Oriented Programming and the Objective C Language*
(http://www.next.com/Pubs/Documents/OPENSTEP/ObjectiveC/objctoc.htm)
) is the single best available source, and it can also be ordered as a hardcopy book from the NeXT publications page
(//www.next.com/Pubs/Documents/OPENSTEP)
. Don't try to program in Objective C without it.

Swarm style of Objective C Programming

No matter where you've learned to program in Objective C, don't try to use Objective C in Swarm without understanding the special ways in which Swarm adapts its use of the language and the language runtime system. Your place to find this information is right here in the defobj library. Defobj not only provides the most basic layer of foundation class libraries for Swarm, but also serves as the place where all the rules and guidelines for the use of Objective C in Swarm are gathered together.

Defobj Library: Advanced Usage Guide

(This section of documentation is not yet available.)

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

Last modified: 1998-01-08

- module definition conventions
(module.html)
- create combination messages
(combo.html)
- global portability assumptions
(globalport.html)
- global definitions
(globaldef.html)

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

Last modified: 1998-01-08

The DefinedObject type defines a minimum of standard messages, and leaves to other types the definition of message that might or might not apply in any general way to particular objects.

Inherited behavior

The DefinedObject type inherits no behavior from any other type. It is the top of the interface type hierarchy for objects that follow the defobj programming conventions. All objects that follow these conventions should inherit from DefinedObject, either directly or indirectly through other types.

Defined behavior

```
- getZone;
- (BOOL) respondsTo: (SEL)aSel;
- getClass;
```

The getZone message returns the zone in which the object was created. The respondsTo: message returns true if the object implements the message identified by the selector argument. To implement a message means only that some method will receive control if the message is sent to the object. (The method could still raise an error.) The respondsTo: message is implemented by direct lookup in a method dispatch table, so is just as fast as a normal message send. It provides a quick way to test whether the type of an object includes a particular message. getClass returns the class that implements the current behavior of an object.

Defined behavior

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

These messages add or remove an external reference to an object that is notified whenever the object is changed or relocated. (More details to follow.)

```
- (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;
```

These messages are have implementations in the Object superclass, excppt for the three-argument form of perform:. They have local implementations in defobj, however, just to make subclassing

Name

Create – create a customized instance of a type

Synopsis

```
//
// Create -- create a customized instance of a type
//
@deftype Create <DefinedObject>
CREATING
+ create: aZone;
+ createBegin: aZone;
- createEnd;

+ customizeBegin: aZone;
- customizeEnd;
- customizeCopy: aZone;
@end
```

Description

The Create supertype defines standard messages that provide a general-purpose protocol for creating new objects. These messages may be used either to create a new instance of a type in one message, or to bracket a series of messages that customize available options for an object to be created. The separation of create-time specifications from later behavior of an object gives substantial flexibility to adapt a generic type to particular needs.

These create messages may be implemented either by a type that hides its implementing classes, or directly by a class that adopts these messages as a uniform interface for creating objects. If implemented directly by a class, then the class object serves as the type object in all message descriptions that follow. Otherwise, a type object might be implemented in a variety of ways that guarantee only that published messages on a type are accepted.

In addition to the create messages defined by Create, an object type may support any other messages of any other names or calling conventions. These messages define only a standard method for creating new objects that other types are free to inherit and implement in conformance with a uniform convention. Further conventions are established elsewhere (create combination messages
(combo.html)

createBegin: returns an interim object intended only for receiving create-time messages. If a type was defined by a @deftype declaration, these messages are those appearing in either the CREATING or SETTING sections. Otherwise, the messages valid as create-time messages are defined by the type without any specific syntactic marker.

The createEnd message completes the process of specifying available options for an object being created. Typically it validates that requested options are valid and consistent with one another, and raises an error if they are not. The standard, predefined error InvalidCombination
(Error.html#stderr)
may be raised by createEnd to indicate an invalid combination of requests, or other, more specific forms of error handling may be used.

If all requests received since the initial createBegin: are valid, both individually and in combination with each other, then createEnd determines a finalized form of object that satisfies all requests received and then returns this object. Any additional storage required for the finalized object is taken from the same zone originally passed to createBegin. The object may have whatever implementation is selected to best satisfy a particular request. Different requests may result in entirely different implementations being returned. The only guarantee is that a returned object supports the messages defined for further use of the finalized object. If a type was defined by a @deftype declaration, these messages are those appearing in either the SETTING or USING sections.

On return from createEnd, the id of the interim object returned by createBegin: is no longer guaranteed to be valid for further use, and should no longer be referenced. A variable which holds this such an id can be reassigned the new id returned by createEnd, so that the same variable holds successive versions of the object being created. Following is a fragment that illustrates typical coding practice using createBegin: and createEnd:

```
newArray = [Array createBegin: aZone];
  [newArray setInitialValue: aList];
  [newArray setDefaultMember: UnsetMember];
  [newArray setCount: [aList getCount] * 2 );
  newArray = [newArray createEnd];   // ! note reassignment of newArray
```

By resetting the variable to the returned value of createEnd, there is no possibility that the id value first returned by createBegin: can ever be used again. Sometimes, the id value returned by createEnd might be the same value first returned by createBegin:, if createBegin: was able to allocate the same object returned later by createEnd. In other cases, however, the object returned by createBegin: might be strictly a temporary object that is thrown away after createEnd determines a final object to be created and returned. To protect against changes in implementation, only the final createEnd return value should ever be used again after createEnd has returned.

```
+ customizeBegin: aZone;
- customizeEnd;
- customizeCopy: aZone;
```

Some types accept create-time messages not only when creating a new instance, but to customize a new version of the type itself. Objects created from a customized type will have all options preset that create-time messages sent to the customized type object have already set. If many cycle of customization (by another sequence of customizeBegin:/customizeEnd) depends on the implementation of the original starting type. A type should not be relied on to support more than one cycle of customization unless it is specifically documented to do so.

Any interim object returned by either createBegin: or customizeBegin: supports the getZone and drop messages that a finalized instance may also support. These messages are defined by the Drop
(Drop.html)
type, which is normally inherited by a type to declare these messages on a finalized instance. This type is not inherited by the Create type because the messages would then apply to the finalized instance, not to the interim object. Even though not declared, the messages are available on the interim objects nonetheless. The drop message on an interim object may be used if it turns out that a finalized version is no longer required after creation or customization has already begun.

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

Last modified: 1998-01-08

of an object lifecycle.

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

Last modified: 1998-01-08

```
- (void) xfprint;
- (void) xfprintid;
@end
```

Description

A zone is a source of storage for objects or other allocated data. Whenever a new object is created, a zone must be identified from which the storage for its instance variables, or other internal data, is obtained. A program may establish multiple zones to ensure that objects with similar lifetime or storage needs are allocated together, and in general to optimize allocation and reuse of storage.

Zones also maintain a collection of all objects allocated within the zone. This collection, referred to as the "population" of a zone, is a set of all objects which have been created but not yet dropped within the zone. Collections maintained automatically by zones can eliminate a need for other, separately maintained collections in applications that need to keep track of entire populations of objects. Collections of allocated objects can provide support for object query, external object storage, and automatic storage reclamation.

A zone may be used to obtain storage not only for objects, but also for raw storage blocks like those provided by the C malloc function. All objects and storage blocks allocated in a zone remain local to that zone. This means that allocation of storage in other zones does not affect the efficiency of storage allocation within a particular zone. For most zone types, individual allocations may still be freed within a zone, and total storage of a zone may grow and shrink according to aggregate needs. In addition to freeing individual allocations, an entire zone may also dropped. Dropping a zone automatically frees all allocations made within it, including final drop processing on any allocated objects that need it. Release of an entire zone can be much faster than individual release of each object within it.

The Zone type is a fully implemented type that provides default storage management support for objects and other allocated storage. It is also a supertype for other zones that implement alternative policies for use in specialized situations.

(..The current Zone implementation is a temporary one based on a simple pass-through of all allocation requests to the C library malloc function. Complete support of the zone population is already provided, however, as is support for internal storage mapping that will eventually be required for automatic storage reclamation Considerable improvements in storage allocation are possible when the current implementation is replaced by more efficient allocation contained in pages local to each zone. ..)

Inherited behavior

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

allocIVars: allocates the instance variable structure for a new object. The initial word of this structure is set to class id passed as its argument. The class also determines the size of the structure allocated. All remaining contents of this structure are initialized to binary zeroes.

copyIVars: creates copies an existing instance variable structure into a new allocation made within the local zone. The existing instance variable structure may be in any zone, but must contain a class pointer in its first word that correctly describes the size of the structure.

freeIVars: releases storage that was previously allocated to hold the instance variable structure of an object. The first word of the object must be a class pointer that correctly describes the size of the structure. Storage allocated by allocIVars: or copyIVars: may be freed only by freeIVars:, and freeIVars: may be used only to free storage allocated by one of these messages.

The object allocation functions are intended for use only within the implementation of an object class. Users of an object type should create a new object only using create: messages or other messages provided for this purpose.

Internal component objects

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

- getComponentZone;
```

These messages allocate, copy, and free the storage that holds the instance variables for an object, but they allocate the object as an internal component of the zone that is not included in the zone population. They are used by classes that allocate additional objects as part of the implementation of another object, and that control the mapping of this storage separately from the zone level objects. getComponentZone returns a specially qualified version of the zone that automatically allocates all its objects with the internal component qualification, even if allocated with allocIVars: or copyIVars:. This qualified zone may be passed as an argument to a create: or createBegin: message so that it will create the new object as an internal component object.

Raw storage block allocation

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

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

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

Last modified: 1998-01-08

Create combination messages

```
CREATING
+ create: aZone setName: (const char *)name;
```

create:setName: is a combination message defined as a caller convenience. See combination messages
(combo.html)
for a summary of conventions on combination messages.

Create-time options

```
CREATING
- (void) setName: (const char *)name;
```

The Name option may be set only at create time. Its value is a null-terminated character string that remains fixed for the life of the object. The inherited getName message returns this name from a created instance.

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

Last modified: 1998-01-08

(module.html)
for standard methods of declaring interface symbols in a library header file.)

These predefined warning and event types are typically used not directly by the messages defined above, but by the following macro defined in defobj.h:

```
#define raiseEvent( eventType, formatString, args... )
```

This macro raises an event of the type identified by the first argument. Default handling of the event is to print a formatted message including standard header and trailer lines, and then either to continue (in the case of a Warning) or to terminate the program in the case of Error). The standard header lines include the file, function, and line number where the macro that raised the event was executed.

If the formatString argument is nil, a default message string already established as part of the warning or event type is used to report the error. Otherwise, the formatString argument must be a null-terminated C character string that overrides the default string.

Either the default or replacement message string may contain printf-style formatted field definitions. For each formatted field definition, an argument value must be supplied in the args... section of the macro. (The support of the GNU C compiler for macros with variable argument lists is used by the macro.)

(.. Methods for defining new event types as part of a library implementation are not currently documented. Instead, just use the raiseEvent macro on one of the predefined Warning or Event Types if you want to raise an event using this standard mechanism. Eventually, this standard mechanism will be used to handle warnings and errors under various forms of logging exception handling support.)

Standard Error and Warning Types

The following standard error and warning types are defined by the defobj library. Thesy may also be used by other libraries for errors or warnings that fit within these standard predefined categories:

```
//
// standard errors
//
extern id <Error>
  SourceMessage,          // message in the source defines error
  NotImplemented,         // requested behavior not implemented by object
  SubclassMustImplement,  // requested behavior must be imple-
mented by subclass
  InvalidCombination,     // invalid combination of set messages for create
  InvalidOperation,       // invalid operation for current state of receiver
  InvalidArgument,        // argument value not valid
  CreateSubclassing,      // improper use of Create subclassing framework
  CreateUsage,            // incorrect sequence of Create protocol messages
```

Miscellaneous Operations

Object Types

The following object types are documented below:

- Copy
- GetName
- GetOwner
- SetInitialValue

Summary of Miscellaneous Operations

An assortment of object types in the defobj library serve just to establish a single declaration of messages that many other object types adopt as part of their declared interface. The centralized definition of the messages in these types avoids their repetition throughout many other types. It also provides a single place where basic rules for these messages can be summarized for all other implementations to follow. This reference documentation provides these summaries along with intended purpose of each message.

> None of these types inherits from any other types; their whole purpose is just to package the definition of one or more messages. The type name is typically the same as a message declared by the type, only with the uppercase convention of a type name. These types may be freely mixed into the inheritance lists of other types to build whatever combinations of messages those other types need to define.

Copy Interface Description

Synopsis

```
//
// Copy -- copy all state defined as part of object
//
@deftype Copy
- copy: aZone;
```

Synopsis

```
//
// GetOwner - get object on which existence of object depends
//
@deftype GetOwner
- getOwner;
@end
```

Description

The getOwner message returns another object which is considered as the owner of an initial object. What is considered as an owner depends on its specific object type, but might be a larger object of which the local object is a part, or an object that has exclusive control over the local object. The principal constraint established by an ownership structure is that a given object can have only a single other object as its unambiguous owner.

Ownership hierarchies arrange themselves in a strict, single-rooted tree. The top-level node of an ownership hierarchy typically returns nil as its owner. If an object is regarded merely as one part of another object defined as its owner, then copying or dropping the owner object should copy or drop the member object as well. Owner and member are neutral terms for a generic relationship sometimes called parent vs. child, but it is up to a particular object type to define specifically what it means by a getOwner relationship.

SetInitialValue Interface Description

Synopsis

```
//
// SetInitialValue - create using initial value from an existing object
//
@deftype SetInitialValue
CREATING
+ create: aZone setInitialValue: initialValue;
+ create: aZone setReadOnlyValue: readOnlyValue;

- (void) setInitialValue: initialValue;
- (void) setReadOnlyValue: readOnlyValue;
SETTING
- (void) setReadOnly: (BOOL)readOnly;
USING
- (BOOL) getReadOnly;
@end
```

Module Definition Conventions

Overview

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

(../defobj.html)
. These conventions include some minor special syntax to declare the public interfaces to Objective C objects. 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 "module."

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

The separation of implementation means that a library module publishes its interface entirely without reference to any Objective C classes which implement its objects.

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 module 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

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

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 is no other specific return value is generally *not* followed. If there is no specific return value needed from a message the return type is declared (void).

(mailto:mgd@santafe.edu)

Last modified: 1998-01-08

Defobj Library: Global Portability Assumptions

In a pure object-oriented system such as Smalltalk, the type of an object reference (an object id in Objective C) is the only fundamental data type used within the system. In a hybrid object system such as Objective C, both object id's and other data types are used together. The Swarm libraries make some basic assumptions about the compatibility of native C data types with the id pointer type of Objective C. These assumptions allow some of the more generic abstractions of the libraries (such as collections and actions) to provide one fundamental abstraction that can still be used under an identical implementation to support most of the other C data types.

The basic assumption is that all data types except double may be cast to and from a stored id type without losing the original value. This assumption is valid on all modern 32-bit or greater machines, including all machines on which Swarm is currently running or planning to run.

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

Last modified: 1998-01-08

func_t provides a type definition for a generic function pointer type (taking no arguments and returning no value). This type name is used in declaring messages that accept or return function pointers of various alternative types.

---

initModule() function

Synopsis

```
//
// initModule() -- module initialization macro
//
#define initModule( module )
```

The initModule function must be called to initialize all the definitions of a library module before any attempt is made to use the library. Initializing a particular library module will also initialize all other library modules on which the library depends. The macro has no effect if the library has already been initialized, so that multiple libraries that all depend on another library can all request initialization of the other library with no ill effect. The argument of the macro is the name of module to be initialized. For example, if just the collections library were being initialized, the macro would be coded:

Global Zones

Synopsis

```
#define globalZone
#define scratchZone
```

The identifiers globalZone and scratchZone refer to two standard zones created by initialization of the defobj library. All other created zones must obtain their storage from one or the other of these two initial zones. (These initial zones are defined as macros only to follow conventions on global external names.)

> globalZone may be used for all storage which must be allocated for the duration of the program, or until explicitly released. scratchZone should be used for storage which is allocated only for use during execution of a single function. It is typically used for objects which are strictly temporary as part of a local computation. (.. Currently, a scratch zone has the identical implementation as the global zone, but eventually a scratch zone may be optimized for temporary allocations, and such allocations may also be freed automatically at various points when functions are guaranteed to be complete.)

The xprint function prints a standard object description to the standard debugging output file. It generates the string to be printed using the describe: message defined by the DefinedObject supertype, but it is defined as a function so that it may be called directly from a debugger. xprintid is similar, but it prints only a one-line description using the built-in default.

The xfprint function is similar to the xprint function, except that it prints a debug description for each member of a collection. The argument of xfprint must support enumeration of collection members using the messages of the Collection
(../collections/Collection.html)
type. xfprintid is similar, but only one line per member is printed.

The xexec function executes a message on an object passed as an argument. The name of message to be executed (the full name including arguments as would be used in naming a selector) must be given as the second argument. This function is equivalent to the "perform:" message of Objective C, but is defined as a function so that it may be called directly from a debugger.

The xfexec function is similar to the xexec function, except that it executes a message on each member of a collection. The first argument of xfprint must support enumeration of collection members using the messages of the Collection
(../collections/Collection.html)
type.

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

Last modified: 1998-01-08

Defobj Library: Interface Design Notes

(This section of documentation is not yet available.)

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

Last modified: 1998-01-08