Design Issues for Objective-C v?.?
DRAFT of July 3, 1987

s. naroff, a. watt

Research and Development
Productivity Products International
Sandy Hook, CT 06482
(203) 426-1875

ABSTRACT

This document is a general review of the Objective-C language based on the desires of current and prospective customers, and PPI technical staff.

This draft was written to:

- address the engineering staff at PPI.
- communicate research that has been done.
- stimulate discussion on the current state of the technology.
- propose solutions and direction which will be long-lived.

* Objective-C is a Trademark of PPI.

2. Introduction

This document is a general review of the Objective-C language based on the desires of current and prospective customers, and PPI technical staff. Our purpose is to define a language adequate to support PPI's goal of reusable libraries, and attractive in its own right for customer development. Our guiding principles were:

- Conform to well accepted and validated principles of modern programming languages.
- Make Objective-C appear to the experienced user as a graceful extension to C.
- Define changes which will be long-lived and confer lasting benefits.

To accomplish this, we attempt to describe the current state of Objective-C and set a direction for future improvements. We further suggest priorities based on perceived market need and internal technical dependencies. Rather than limit this document to a proposed task list, we invite discussion on the high level direction and priorities outlined.

3. Problems with Objective-C.

Before discussing specific language problems, it is valuable to highlight a few common complaints from the users perspective.

- The system needs to be re-compiled frequently.
- Collisions over selector usage when integrating Software-ic's developed independent of one another.
- Compilation order is highly constrained, and not always clear. Extra effort is required to maintain makefiles.
- Performance problems. Users perceive this to be the fault of dynamic binding.
- Compiler generated interface files must be understood as part of any multi-persion development; this creates confusion.
- The Foundation Library is difficult to "reuse". New users perceive this to be a documentation problem.
- Debugging is inconvenient. Not all systems provide enough support for our documented debugging techniques. In general, C compiler tools (like "lint") can be hard to use/interpret.

The following sections discuss specific language problems that contribute to the current situation.

3.1. Objective-C doesn't produce "relocatable" object files.

Object modules do not contain all the necessary information to be linked with an application. At runtime, each object module currently depends on the existence/integrity of global data that represents the pool of unique selectors used in the entire application. As a result, the object module is not a reliable form of transport between applications that have been compiled separately. This causes two, serious problems:

(1) Previously compiled code can get corrupted for any one of a number of reasons. This causes occasional (or even frequent) need to re-compile the whole system.

(2) Object modules cannot be dynamically linked. This requires that all classes (that a program might use) be linked before the program can begin execution. Since Objective-C does not currently offer dynamic linking, this is only a problem for supporting it on systems that *do* offer it as part of the host environment (like Apollo).

Requiring that source code be the only reliable form of transport is very cumbersome, and falls seriously short of customer needs (it can significantly increase the turn-around time required to make changes to the application).

- Diagnostics are very limited (about possible mis-use).
- Examining object instances using standard C debuggers is inconvenient and very unreliable.
- Makes code harder to read/understand when the type of object is known.

Considering Objective-C is built on top of C (many data types - efficiency), forcing the application writer to use this degree of flexibility (in a hybrid language) seems unnatural. If the developer **knows** the class of object at compiletime, the compiler must allow this to be expressed; this will enable the compiler to manage selectors on a per-class or per-application basis, depending on how much information the developer was able to supply at compiletime.

3.3. The generated files are not well defined, lack flexibility, and are poorly managed.

The class and message declaration files that are generated by Objective-C create confusion among users. A template describing the data model (instance variables) is maintained on "per-class" basis; however, a template describing the procedure model (selectors) for the class is maintained as a "shared pool" of selectors. Managing selectors in this fashion enables Objective-C to provide an *efficient* mechanism for accomplishing dynamic binding.

This approach significantly increases the probability that the **integrity** of a class will be damaged just because another class in the pool has changed; the order in which classes are compiled is sacrosanct. Basically, the integrity of the generated code currently depends on the message declaration file remaining consistent between the moment it is first created and the moment the application is finally built. Because this is unacceptable for developing large, complex software systems, we must revaluate who will create/manage these files by listing the benefits of having Objective-C manage interface files for the user.

3.3.1. Advantages.

- The developer does not have to manage two files, one to declare, another to define. Changes to the definition of a class are automatically reflected in the declaration.
- The developer does not have to manually *#include* the declaration, it is input automatically by Objective-C, based on the class name and include path(s) supplied by the user.

3.3.2. Disadvantages.

- The developer has **no** control over what can be included in the class declaration file; the language is *tailored* for providing information to the Objective-C compiler exclusively. If the developer managed class declarations, they could embody all the information necessary to interface to the class. This has real value for developing large systems and is consistent with the way C programmers are accustomed to working. If the compiler continues to automatically *#include* the declaration, this is not easily possible and is a major departure from the way the C world is accustomed to working.
- The compiler assumes that all method definitions should get published in the interface for a class. If the developer managed this, it would be possible to declare only those methods that are ready for *publication*; this ability to omit declarations (private methods), based on the maturity of a class is especially useful during early stages of the development/learning process.
- The compiler enforces **one** class declaration file per definition. This enables it to automatically *#include* the declaration during the compilation of a consumer of the class. Management of these files can get unwieldy for large systems.
- The "C" community is not accustomed to having descriptions of anything *automatically* created/updated/included when compiling a program. This violates the "one input; one output" model which corresponds to the way people work now. If Objective-C is going to deviate from this model, and manage descriptions automatically, it must do a much better job.
- Super classes must be compiled before sub-classes (minor).

4.1. Summary.

The generated code and runtime support for dynamic binding must change dramatically to implement many of the proposed improvements to Objective-C. As a result, object files will **not** be compatible with previous versions. The constructs that would be eliminated are highly **visible**, current customers will experience significant change in the way they work and perceive the system. New customers should find the system more flexible and easier to learn/use; unusual constructs that currently create confusion will be replaced with support functions that are accomplished in a transparent fashion, during runtime.

5. Extensions to Objective-C.

5.1. Ansi-C constructs.

There are urgent reasons to add to Objective-C some of the features of the not-yet-adopted ANSI C standard.

5.1.1. We Must Work With C Compilers Which Include ANSI Constructs.

Even though the standard has not been adopted, several vendors are introducing certain of the new features already. Microsoft 4.0 on the PC includes function prototypes; The next release of C on VMS (available June '87) will include function prototypes and "const" and "volatile" keywords. The next release for Apollo/DOMAIN (release date not known) will do likewise.

Non-UNIX vendors are moving ahead of the official adoption date for two reasons:

(1) It replaces their own (non-standard) extensions to accomplish the same purposes (VMS and DOMAIN).

(2) It provides desperately needed facilities which ease portability problems (function prototypes for PC, where integers and pointers can be of different sizes).

If we provide Objective-C for an environment whose host C compiler does support ANSI extensions and our compiler does not, then customers cannot use those extensions within Objective-C source files. Because people will often use new features without realizing it (by including vendor-supplied definition files), **this could render Objective-C unusable.**

5.1.2. Providing Some ANSI Constructs May Ease Our Portability Problems

The parser changes necessary to accomodate some new ANSI constructs could be a way to reduce the amount of work necessary to accomodate local non-standard keywords. Examples are "near", "far", and "huge" on the PC, and "globaldef", "globalref", "globalvalue", "readonly", and "noshare" on VMS. Our current parser design requires explicit productions for these be added to the parser, and possibly additional classes to manage these non-standard declarations. Some of these local extensions will unfortunately remain even after everyone adopts ANSI - there is no equivalent in the standard for "global*", "near", "far", "huge".

Adapting our parser to local non-standard language extensions is a major headache from technical, documentation, and configuration control perspectives.

5.1.3. Partial Task List.

- Function Prototypes; support new-style function declarations.
- New keywords "const", "signed", "volatile".
- New types "long double".
- Remove old types "long float".
- New constant type "\Xddd" (hex digits).
- New constant qualifiers "U" (unsigned) and "F" (float).
- ANSI-spec preprocessor (?) - some of the proposed changes will break existing code. We might want to wait until the ANSI draft is blessed to implement this.

5.2.5. Disadavantages

There are a number of constrants the programmer must observe, most assisted by compiler warnings. Certain straight-forward ways to perform common operations can incur unacceptable overhead, and must be done instead in a way which is fully aware of the garbage collection system and its operation. Basically, a project must decide early whether to use garbage collection, and design accordingly. Additional programmer **training** is also required.

The space overhead totals between 12 and 16 bytes per object (in addition to the space penalty already imposed by dynamic allocation). This is prohibitive for seriously memory-poor environments like the PC.

All parts of a system which uses garbage collection must be compiled with the options which enable it. There is at present no reliable way to detect a violation of this requirement; the effect is to cause needed objects to be freed.

5.2.6. Applications

This garbage collection system will probably be of no use to people who are running out of memory because there simply isn't enough (e.g. PC). It would be useful for long-running applications which create objects frequently and cannot know when to free them. The performance penalty should be evenly spread through the application, so it is suitable for interactive and other realtime-constrained systems.

5.2.7. Partial Task List.

- Port to Sun.
- Test on sample applications.
- Marketing strategy (?) - sold separately or part of the standard product.
- Research which machines we might support short/long term.
- Consult with HP on possible changes/enhancements.

5.3. Static/Late Binding.

The claim Objective-C makes that fully dynamic binding is **necessary for some** purposes does not justify supplying a language assuming such binding is **sufficient for all** purposes. Instead of considering only the polar opposites of completely static vs. fully dynamic binding, we should provide a range of binding styles and allow the user to choose the most appropriate. "Binding" here does not just mean messages vs. function calls – it means the degree of intimacy the user and the compiler have with the private details of an object, and all the consequences which follow.

Enforcing the "dynamic only" binding on object operations means developers are forced to handle all objects anonymously. Several consequences of this are:

- All implementations for a selector must return the same type.
- Compile time error and type checking are difficult or impossible.
- Increased code complexity and less efficiency in many situations.
- Objects can only be allocated dynamically off the heap; not defined statically or as procedure locals.
- Currently, intra-object (local) messages are just as expensive as inter-object messages; unacceptable for developing large systems, where performence is crucial.

If the developer knows the class of a particular object at compiletime, we should allow him to communicate that to the compiler in a convenient fashion. Declaring an instance of a particular class allows the compiler to provide the following facilities and benefits that are not possible if declared anonymously:

- Better compiler diagnostics about possible mis-use.

implementation tips and techniques.

- Convenient interface to host-supplied tools – so that the user sees class and method names instead of funny function names.
- A detailed guide for each system on how to use the tools and interpret the results.

It isn't sufficient simply to make messaging faster or allow people to do less of it – their applications will still run too slow and they will still blame us. We need to give them the tools to truly understand their application's behavior and guidance on how to improve it.

5.5. Multiple Inheritance

In its most general form, **multiple inheritance** is the ability to define classes which inherit both data and procedures from more than one superclass. In a C-based language, the full generality is very difficult to provide, but a class' ability to acquire functionality from more than just its superclass clearly has value. A more restricted form of multiple inheritance, called **mix-in**, allows users to define packages of useful functionality which can be "relocated" to different places in a single inheritance hierarchy.

Mixin packages would be sufficient to support functionality such as linked lists, name labels, display dependencies, position in a coordinate system, etc. They might also provide an implementation for the abstraction of a *protocol*, or a group of methods which are uniformly implemented in several possibly unrelated classes.

5.6. Blocks

Blocks in Smalltalk-80 are like procedure variables in Pascal, but with none of the restrictions. A block carries with it the local context (arguments and local variables) of the method which created it, and can also be supplied with additional arguments when invoked. The difference between blocks and Pascal procedure variables is that the block remains "live" *even after the context which created it exits*. The difference between blocks and C function pointers is that the C function has no access to local variables or arguments from the context which created the pointer.

A block therefore is an executable statement which retains access to its creation context regardless of the state of program execution since creation. Blocks can be assigned to variables, passed as arguments, and in general used in any way an expression can be used. Blocks can be activated by sending them a message, possibly with arguments, which causes the statement part of the block to execute.

Blocks provide a convenient way for users to *extend* the functionality of existing clases by defining new procedures which those clases can execute. An example is the **select:** method in the Smalltalk-80 collection classes. The argument is a block and the effect is to create a new collection which contains all those members of the original collection for which the block evaluates true. There is a complementary method **reject:**. This is an obviously powerful facility to have as part of generic collection classes.

Another very handy use of blocks is as part of exception handling. Several Smalltalk-80 methods take as an argument a block to execute if the requested operation fails

There are significant implementation difficulties in providing blocks as described here.

5.7. Dynamic Linking

Dynamic linking is the ability to load relocatable object files into a running program and resolve symbolic references. Dynamic linking allows more flexibility in building applications, as well as keeping program images on disk smaller by excluding seldom executed code. This tool would be part of the Objective-C runtime support environment, not part of the language.