

The Origins of Objective-C at PPI/Stepstone and Its Evolution at NeXT

BRAD J. COX, Retired, USA

STEVE NAROFF, Retired, USA

HANSEN HSU, Computer History Museum, USA

Shepherd: Shigeru Chiba, University of Tokyo, Japan

The roots of Objective-C began at ITT in the early 1980s in a research group led by Tom Love investigating improving programmer productivity by an order of magnitude, a concern motivated by the perceived “software crisis” articulated in the late 1960s. In 1981, Brad Cox, a member of this group, began to investigate Smalltalk and object-oriented programming for this purpose, but needed a language compatible with the Unix and C environments used by ITT. That year, Cox quickly wrote up the Object-Oriented Pre-Compiler (OOPC) that would translate a Smalltalk-like syntax into C.

Love felt there was a market for object-oriented solutions that could coexist with legacy languages and platforms, and after a brief stint at Schlumberger-Doll, co-founded with Cox Productivity Products International (PPI), later renamed as Stepstone, to pursue this. At PPI, Cox developed OOPC into Objective-C. Cox saw Objective-C as a crucial link in his larger vision of creating a market for “pre-fabricated” software components (“software-ICs”), which could be bought off the shelf and which, Cox believed, would unleash a “software industrial revolution.”

Steve Naroff joined Stepstone in 1986 as Steve Jobs’ NeXT Computer became an important customer for Objective-C, as it was being used in its NeXTSTEP operating system. Naroff became the primary Stepstone developer addressing NeXT’s issues with Objective-C, solving a key fragility problem preventing NeXT from deploying forwards-compatible object libraries. Impressed with NeXT, Naroff left Stepstone for NeXT in 1988, and once there, added Objective-C support to Richard Stallman’s GNU GCC compiler, which NeXT was using as its C compiler, removing the need to use Stepstone’s ObjC to C translator. Over the next several years, Naroff and others would add significant new features to Objective-C, such as “categories,” “protocols,” and the ability to mix in C++ code. When Stepstone folded in 1994, all rights to Objective-C were acquired by NeXT. This eventually transferred to Apple when NeXT was acquired by Apple in 1997. Objective-C became the basis for Apple’s Mac OS X and then iOS platforms, and Naroff and others at Apple added additional features to the language in the late 2000s as the iPhone App Store greatly expanded Objective-C’s user base.

CCS Concepts: • **Software and its engineering** → **General programming languages; Object oriented languages; Classes and objects; Inheritance; Polymorphism; Abstract data types; Compilers; Runtime environments; Object oriented frameworks;** • **Social and professional topics** → **History of programming languages; History of software.**

Additional Key Words and Phrases: Objective-C, OOPC, PPI, Stepstone, ITT, NeXT, Apple, message passing, dynamic binding, Smalltalk, C++, software-ICs, software crisis, categories, protocols

Authors’ addresses: Brad J. Cox, Retired, USA, bradcox@gmail.com; Steve Naroff, Retired, USA, naroff@me.com; Hansen Hsu, Software History Center, Computer History Museum, 1401 N. Shoreline Blvd., Mountain View, CA, 94043, USA, hhsu@computerhistory.org.



This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2020 Copyright held by the owner/author(s).

2475-1421/2020/6-ART82

<https://doi.org/10.1145/3386332>

ACM Reference Format:

Brad J. Cox, Steve Naroff, and Hansen Hsu. 2020. The Origins of Objective-C at PPI/Stepstone and Its Evolution at NeXT. *Proc. ACM Program. Lang.* 4, HOPL, Article 82 (June 2020), 74 pages. <https://doi.org/10.1145/3386332>

CONTENTS

Abstract	1
Contents	2
1 Introduction	2
2 The Context of the “Software Crisis”	3
3 The Origins of OOPC at ITT	4
4 Productivity Products International and Objective-C (c. 1984–1985)	10
5 Developing Objective-C	12
6 Stepstone and Software-ICs (c. 1985–1990)	14
7 Steve Naroff Joins PPI/Stepstone (1986)	16
8 Early Adopter Feedback	17
9 Solving the Fragility Problem — Dynamic Selector Resolution	19
10 Adopting a Declarative Programming Model	20
11 Stepstone Visit to NeXT (1988)	21
12 Stepstone’s Demise (1990–1994)	21
13 Objective-C and the Free Software Movement (1988)	22
14 Categories (1989)	23
15 Objective-C++ (1989–1990)	24
16 Method Forwarding, Signatures, and Protocols (1990)	25
17 Stasis and Evolution at Apple (1997–2012)	27
18 Conclusion	30
Acknowledgments	30
A Objective-C Timeline	30
B Historical Document: Design Issues for Objective-C DRAFT of July 3, 1987, S. Naroff, A. Watt, PPI	34
C Historical Document: Objective-C v.4.0 charts, Stepstone, 1988	40
D Historical Document: Spec.Language, S. Naroff, c.1987	53
E Historical Document: Spec.Runtime, S. Naroff, c.1987	59
References	71
Non-archival References	73

1 INTRODUCTION

In 2020, most software developers know of Objective-C as the language used to write software for Apple’s iPhone and Macintosh computing platforms. What they may not know is that Objective-C predates the iPhone by two decades, and was not originated by Apple. Objective-C was chosen as the main programming language for NeXTSTEP, the operating system of the NeXT Computer, produced by Steve Jobs’ startup NeXT after he left Apple in 1985. It was Apple’s purchase of NeXT in 1997, and its development of NeXTSTEP into Mac OS X and later iOS, that made Objective-C central to Apple’s technology stack. Yet Objective-C predates even NeXT. Objective-C was created by Brad Cox and Tom Love, the co-founders of a small Connecticut company, Productivity Products International, later renamed Stepstone, to spur the computing industry to adopt object-oriented

programming and create a market for software components, which they believed could revolutionize software production by improving programmer productivity by an order of magnitude.

2 THE CONTEXT OF THE “SOFTWARE CRISIS”

Objective-C had its origins in a technology research group concerned with improving the productivity of software programmers, immersed in the discourse of the “software crisis.” Beginning in the 1960s, a growing body of computing literature began to articulate a sense of immanent crisis in the production of software. This literature has been studied by a number of historians of computing, with some arguing that the perceived “crisis” was primarily a rhetorical construction [Abbate 2012; Ensmenger 2010; Ensmenger and Aspray 2002; MacKenzie 2001; Mahoney 1990, 2002, 2004; Slayton 2013; Tomayko 2002, NA Haigh 2010]. Regardless of its reality, the discourse surrounding the software crisis focused on a number of issues faced by practitioners. The cost of producing software was projected to begin outstripping the cost of hardware, according to a widely copied graph by Barry Boehm [Boehm 1973, 49; MacKenzie 2001, 33; Mahoney 2002, 91; Slayton 2013, 155–57]. Demand for programmers was apparently causing a labor shortage [Abbate 2012, 90–91], driving up costs, and programmers were acquiring a reputation for becoming unmanageable [Abbate 2012, 93–97], due to programming acquiring the reputation of becoming a “black art” requiring virtuoso craft skill [Abbate 2012, 93–97; Ensmenger 2010, 19]. The increasing complexity (which increased both risk and cost) of software systems, and the inadequacy of existing technical and organizational solutions to manage such complexity, became a central feature of this crisis discourse [Brooks 1987, 11–12; Slayton 2013, 63–84]. A number of large scale software projects had very publicly failed, most notably IBM’s OS/360 effort, led by Fred Brooks [Brooks 1975, vii–viii, 47–48; MacKenzie 2001, 31–32; Slayton 2013, 112–15]. Brooks’ influential book, *The Mythical Man-Month* [Brooks 1975], was written as a post-mortem of the OS/360 project and focused primarily on the failures of human organization as the primary reason for the failure, recommending a new organizational structure of the programming team as a solution. Brooks was part of an emerging discourse of “software engineering,” coalescing around the NATO Conferences of 1968 in Garmisch and 1969 in Rome, organized to discuss how to address the crisis [Abbate 2012, 97; MacKenzie 2001, 34–37; Slayton 2013, 115].

Three aspects of this discourse are particularly salient to the creation of Objective-C. Boehm noted one of the ways to address cost issues was to improve individual programmer productivity [Boehm 1973, 49–54]. Numerous studies had shown that some virtuoso programmers were an order of magnitude more productive than their peers. Depending on the study, the productivity difference factor cited could be anywhere from 5x all the way up to 100x, with a 10x number often becoming a shorthand for this phenomenon [Boehm 1973, 52; Brooks 1975, 30; Ensmenger 2010, 18–19; Ensmenger and Aspray 2002, 6]. This often justified such programmers’ high salaries and contributed to the sense of their unmanageability.

The flip side of this discourse, however, was the hope for solutions to boost all programmers’ productivity by an order of magnitude. While Brooks and IBM’s Harlan Mills focused on managerial and organization solutions [Boehm 1973, 54; Brooks 1975, 30–37], many looked to technological or disciplinary solutions rooted in making programming more rigorous and mathematical. Program verification, structured programming, modular programming, code reuse, rapid prototyping, graphical programming environments, and object-oriented programming had all been put forth as “silver bullets” to solve the software crisis [Boehm 1973, 54; Brooks 1987, 13–18]. Many had begun as disciplined practices that programmers were exhorted to follow, the most famous being Edsger Dijkstra’s exhortation to avoid the use of the “harmful” GOTO statement [Dijkstra 1968]. However over time, systems, particularly programming languages and compilers, began to be developed that would guide or force their users to think in these new ways and make harmful practices or

certain categories of errors impossible. Boehm cited studies showing at least a 2x productivity increase simply by switching languages [Boehm 1973, 52, 54]. Pascal, Mesa, Modula, and Ada were all created to accomplish these goals. Object-oriented programming (OOP) languages, though conceived for different reasons (for simulation, and to empower children) [Kay 1993], incorporated many of these earlier ideas into a single digestible package. By the 1980s, OOP had begun to be seen as a leading contender amongst “silver bullet” solutions, and as we will see, the creators of Objective-C would play a leading role in making that case. Revolutionary hype surrounding new software paradigms would focus on whether they served up this 10x boost in productivity.

A third feature of the software productivity discourse, especially salient in “software engineering,” was the concern that programming was a craft skill mired in pre-industrial practices, predicated on an analogy with hardware production that came to be seen not as metaphor but as fact. The most famous articulation of this was put forth at the 1968 NATO Conference by Douglas McIlroy of Bell Labs: “We undoubtedly produce software by backward techniques. We undoubtedly get the short end of the stick in confrontations with hardware people because they are the industrialists and we are the crofters. Software production today appears in the scale of industrialization somewhere below the more backward construction industries.” [Mahoney 2004, 11] The very selection of the word “engineering” for this new concern brought with it all the connotations of the engineering of physical systems, which Janet Abbate argues was a deliberate rhetorical move [Abbate 2012, 97–105]. The analogy with the Industrial Revolution, with its interchangeable parts, mass production and assembly lines, became a particular feature of this software engineering discourse. This was reinforced by software’s unflattering comparison to progress in computer hardware, which due to Moore’s Law was improving exponentially. As we will see, this industrial metaphor, and more specifically, the computer hardware metaphor, would become the theoretical foundation for the creation of Objective-C.

3 THE ORIGINS OF OOPC AT ITT

It was within this context that the precursor to Objective-C, the Object-Oriented Pre-Compiler (OOPC), was born at International Telephone and Telegraph (ITT) in the early 1980s. During Harold Geneen’s tenure as ITT CEO, ITT had grown into a complex international conglomerate of more than 100,000 employees and 250 profit centers (Hartford Insurance, Sheraton Hotels, Continental “Wonder Bread” Baking). These holdings included the national telecommunications companies of many countries around the globe, plus numerous smaller ventures. While ITT controlled the international telecommunications market, AT&T controlled the North American market. Rand Araskog became CEO in 1979 as ITT was transitioning from an analog to digital telephone switching system, the System 1240, to compete with AT&T.

To support the System 1240 effort, ITT decided in 1980 to create a research group to rival Bell Labs, called the Programming Technology Center (PTC), located in Stratford, Connecticut. This lab would support the nearby Advanced Technology Center in Shelton (relocated from Stamford), CT, which was modifying the System 1240 switch’s operating system for the North American market. Jim Frame, who had worked at IBM on OS/360 and had been the head of an IBM laboratory in California, was hired to lead the new PTC. Dr. Tom Love, a cognitive psychologist who’d written his dissertation on the characteristics of successful programmers and had done human factors work for the Software Psychology Research Group at General Electric Aerospace, was the second employee hired into the group, after software engineering guru Capers Jones. As the PTC had decided to use Unix as a base for the development environment for ITT, Love discovered Dr. Brad Cox, a Unix expert who had completed his Ph.D. in Mathematical Biology from the University of Chicago simulating neural networks on minicomputers. Other key members of the team included Ted Biggerstaff, Rudy Ramsey, Anatol Holt and Alan Watt [Love 2019, 5]. Watt had been a classmate

of Bill Joy's at Reed College [Cox 2016, 15–16], and had worked with Cox at a previous employer in New Hampshire. After a visit by Watt and Cox to Bill Joy at Berkeley, Cox decided to use a suitcase-sized Unix-based Onyx computer as the host for his demonstration of ITT's desktop future, which could support up to eight users with glass teletypes on the desktop or remotely via 1200 baud modems and UUCP connections to other systems. Cox and Anatol Holt conducted research in coordination technologies to improve the productivity of teams rather than individuals. Cox's coordination project involved creating a better tool for expense travel vouchers, using a program generator that compiled a forms description language into programs that could display expense reports. However, Cox soon realized that program generation would not scale to the thousands of data types necessary to affect the productivity of a large corporation.

From the beginning, the Programming Technology Center was immersed in the discourse of the software crisis. Through Jim Frame's IBM connections, Fred Brooks became a consultant to the group. Another outside consultant was Gerald Weinberg, author of *The Psychology of Computer Programming*. The influence of Brooks and the goal of improving productivity by an order of magnitude can be seen throughout both Love and Cox's publications on OOPC and later Objective-C, starting as early as 1983. In his 1983 paper formally introducing OOPC, Cox made clear the purpose of his new language: "The ITT Programming Environment is being developed as part of a company-wide effort to accomplish an order of magnitude increase in programming productivity during the 1980's. [sic]" [Cox 1983b, 15] Cox's paper "The Message/Object Programming Model," originally written for the 1983 Softfair Conference, and later reworked for publication in *IEEE Software* [Cox 1983a, 51; Cox 1984, 50], cited Brooks' *Mythical Man-Month* [Brooks 1975] in its second paragraph: "books like *The Mythical Manmonth* [sic] have achieved great popularity by emphasizing the need for tools that increase both organizational, as well as individual, productivity." The problem of rising software costs is noted in the opening sentence: "Powerful tools to optimize programmer productivity have become increasingly important as programming costs continue to rise." The order of magnitude productivity improvement metric is mentioned several times in the paper: "I know of no quantitative studies of how much inheritance can help in reducing code size and increasing reuse of pre-existing code, but anecdotal evidence suggests that it can be very substantial, possibly as much as ten-fold." [Cox 1983a, 56] "In 1981 I [Cox] was in a research team [at ITT] responsible for designing and building an advanced programming environment. We worked under the *Buy the Best and Build the Rest* [emphasis in original] motto, concentrating our efforts on new tools that might deliver order-of-magnitude impacts..." [Cox 1983a, 57] Similarly, in a paper given at the same 1983 Softfair conference, Love explained: "Smalltalk-80 has surpassed our initial expectations. Order of magnitude reductions in the bulk of code required for certain applications are achievable." [Love 1983, 61] Love, interviewed for Datamation in 1987, similarly said, "Structured programming... provided only a 10% to 15% improvement in productivity when people were really looking for improvements of 10 to 15 times [emphasis in original]." [Verity 1987] In Cox's later publications, references to the software crisis and software engineering, including the NATO Conferences, become more explicit over time [Cox 1986, 3; Cox 1990a, 25; Cox 1990b, 209; Cox and Novobilski 1991, iii–iv, 3]. For example, a 1985 *Byte* article co-authored by Cox began, "The Software World has run headlong into the Software Crisis—ambitious software projects are hard to manage, too expensive, of mediocre quality, and hard to schedule reliably." [Ledbetter and Cox 1985, 307]

A common feature of software engineering discourse is the metaphor of hardware manufacturing applied to software, which became a dominant theme in Cox's vision for Objective-C. A key was the notion of reusable components as interchangeable parts, which hearkened back to colonial firearms manufacture as organized by Eli Whitney [Cox 1990a, 26; Cox 1990b, 214; Cox 1986, 1; Cox and Novobilski 1991, 1]. According to Cox, as interchangeable parts enabled the conditions

for the original Industrial Revolution, so would reusable software components unleash a new Software Industrial Revolution [Cox 1990a; Cox 1990b, 214]. In 1990, Cox wrote, “I use a separate term—software industrial revolution—to mean what ‘object-oriented’ has always meant to me: transforming programming from a solitary cut-to-fit craft into an organizational enterprise like manufacturing.” [Cox 1990a, 27] Tom Love wrote in 1993, “In the early 1990s, software engineering remains more a statement of desire than reality. The technologies most commonly available to software developers do not allow software to be engineered. Instead, it has been handcrafted by extraordinary craftsmen.... Software engineering as a discipline has failed due to inadequate technology. If we can’t effectively reuse software components, we can’t engineer.” [Love 1993, 21] Cox also compared different layers of abstraction and encapsulation in software to the different levels of organization of computer hardware: gates, blocks, chips, cards, and racks [Cox 1990a, 29; Cox 1990b, 212; Cox and Novobilski 1991, 50]. For Cox, the key level in this hierarchy was the chip, or integrated circuit (IC). To reinforce this metaphor, Cox and Love would trademark the term “Software-IC” to refer to reusable software components created with object-oriented techniques [Cox 1988, 166; Cox 1989, 331; Cox 1986, 2, 9, 19, 26, 53, 68, 70, 78, 88, 91, 96, 104, 156, 215; Cox and Novobilski 1991, 19–20, 26–28, 69, 74–75, 90–92, 118, 174, 217; Cox and Hunt 1986; Ledbetter and Cox 1985; Love 1988, 240–41]. Their component object libraries would eventually be marketed as “ICpaks.” [Cox 1988, 166, 168; Cox and Novobilski 1991, 174, 179, 181, 185, 188, 194; Love 1988, 239–41] Most prominently, Cox’s 1990 *Byte* article, “There is a Silver Bullet,” [Cox 1990b] was an explicit response to Fred Brooks’ 1987 article, “No Silver Bullet,” in which Brooks argued that recent techniques and tools, including object-oriented programming, had only addressed the Aristotelian “accidents” of the problems of software development, without attacking their essence. Cox argued that object-oriented Software-ICs, which could be bought off-the-shelf, would create a new market and new economic patterns, radically disrupting existing software production practices and initiating the “Software Industrial Revolution.” [Cox 1990b, 209–14]

Love and Cox were highly influenced by Xerox PARC’s work on Smalltalk. The Programming Technology Center had been trying to model itself on PARC, and had contact with PARC’s Human Factors group and its Learning Research Group (which created Smalltalk), headed at the time by Adele Goldberg [Love 2019, 7–9]. Although Cox had known of Goldberg as a fellow graduate student at the University of Chicago, he had not had much interaction with her at the time [Cox 2016, 9–10]. Rather, it was the release of August 1981’s special issue of *Byte Magazine* [Xerox Learning Research Group 1981] that was the impetus for Cox’s creation of the precursor to Objective-C, the Object-Oriented Pre-Compiler (OOPC). Smalltalk-80 was the first public release of Smalltalk outside of Xerox, and the *Byte* special issue brought the ideas of object-oriented programming to many in the microcomputer industry for the first time. This led to a big interest in object-oriented programming that would gain momentum in the computer industry throughout the 1980s. Within a year of the January 1984 release of Objective-C, a number of other object-oriented languages would also be introduced commercially, including Clascal (precursor to Object Pascal), Neon (precursor to Actor), Methods (precursor to Smalltalk-V), and C++, the successor to Bjarne Stroustrup’s C with Classes [Love 1993, 41]. Love and Cox would also play outsized roles in promoting object-oriented programming: starting on July 17, 1985, Love was involved in several lunch meetings in Palo Alto to plan the first ACM Object-Oriented Programming, Systems, Languages and Applications (OOPSLA) conference [Love 1993, 24; Love 2019, 22–23]. *Byte* would devote another special issue to object-oriented programming in August 1986, with articles by Cox as well as Larry Tesler and Ted Kaehler, who had worked on Smalltalk at PARC, both employed by Apple in 1986 [Cox and Hunt 1986; Kaehler and Patterson 1986; Tesler 1986]. By 1991, object-oriented programming had generated enough hype to be profiled in the cover story of *BusinessWeek*, with quotes from Cox, Adele Goldberg and Bjarne Stroustrup interspersed with Philippe Kahn, Bill Gates, and Steve Jobs

[Verity and Schwartz 1991]. Cox and Love’s publications and participation in the OOP community helped raise the profile of object-oriented programming among the larger computing industry. Even Fred Brooks in 1987, though arguing that no single solution would be a silver bullet, thought that object-oriented programming held the most promise for order of magnitude gains [Brooks 1987, 14].

Object-oriented programming also gained in popularity due to its association with graphical user interfaces. Steve Jobs’ famous visit to PARC had involved him seeing overlapping windows and popup menus for the first time, which had been implemented in Smalltalk, but he had been so impressed with the GUI that he had not noticed OOP [Hiltzik 1999, 330; NA Cringely 1996]. Nevertheless, in the Smalltalk environment the GUI and OOP went hand-in-hand. When Apple created the Lisa, Larry Tesler, who had come to Apple from PARC, helped create the Lisa’s object-oriented class library, Lisa Toolkit, alongside Clascal, an object-oriented extension to Pascal. This later provided the basis for a similar framework on the Macintosh, MacApp, based upon Object Pascal, a collaboration between Tesler and Niklaus Wirth [Schmucker 1986a,b, Tesler 2016, 42–45]. The Macintosh itself, however, did not require object-oriented programming methods, primarily for performance reasons. Nevertheless, the association between GUIs and OOP was very close, sometimes becoming conflated with each other. “The microcomputer that is most closely related to the original Smalltalk philosophy and object-oriented programming in general is the Apple Macintosh,” wrote the editors of the 1986 *Byte* special issue on object-oriented programming [White and Malloy 1986, 137]. Cox in that issue actually referred to GUIs as “object-oriented user interfaces”: “Making computers easier to use has been an enduring dream since the dawn of computing. This is one of the reasons for the current interest in *iconic* or *object-oriented* [emphasis in original] user interfaces—interfaces that present information as pictures instead of text and numbers.... They must determine the user’s needs from a graphical input device like a mouse rather than the usual commands from a keyboard.” Indeed, Cox notes that GUIs actually increase the difficulty of programming, a task for which object-oriented programming is especially well suited to solve: “...iconic programs can be excruciatingly difficult to build.... Unless tools can reduce this complexity, iconic user interfaces will remain costly and comparatively rare.” [Cox and Hunt 1986, 161] Because the GUI mapped neatly onto OO concepts, using OOP for GUI programming could greatly improve programmers’ productivity when developing such graphical applications [Schmucker 1986a,b; Tesler 1986]. This would become a key selling point for NeXT’s development environment.

The August 1981 *Byte* issue’s description of Smalltalk-80 excited Cox, who thought it might be better than C for programming graphical interfaces as well as provide encapsulation. However Smalltalk’s performance issues and lack of collaboration facilities prevented it from being deployed in production telecommunications applications, in which speed, group coordination, and compatibility with existing software was important. The PTC had committed to Unix and C for these benefits and because it was cross-platform, not locking them in to solely IBM or DEC hardware. However Cox believed it possible to extend C with Smalltalk-like object-oriented extensions, and thus get the best of both worlds. With Love’s blessing, Cox immediately set out to create a Smalltalk-like object-oriented language that would be compatible with C. Over two weeks after receiving the *Byte* issue, Cox wrote the “Object-oriented Pre-Compiler” (OOPC) that translated Smalltalk-style message passing expressions into C [Cox 1983b].

Smalltalk syntax and semantics are based on the notion of message passing. As described by Xerox’s Learning Research Group in *Byte*, “The Smalltalk-80 system is composed of objects that interact only by sending and receiving messages.... Messages are described by *expressions*, which are sequences of characters that conform to the syntax of the Smalltalk-80 programming language. A message-sending expression describes the *receiver*, *selector*, and *arguments* of the message. When

an expression is *evaluated*, the message it describes is transmitted to its receiver [emphasis in original].” [Xerox Learning Research Group 1981, 36] In the following example, taken from this article:

```
frame center
```

frame is the *receiver*, the object receiving the message. Center is the *selector* which tells the receiver which method to invoke upon receiving the message. A method “describes a sequence of actions to be taken when a message with a particular selector is received by an instance of a particular class.” [Xerox Learning Research Group 1981, 39] For more complicated message expressions, Smalltalk-80 syntax provides a keyword message: “...one or more arguments and a selector that is made up of a series of *keywords*, one preceding each argument [emphasis in original]. A keyword is an identifier with a trailing colon.” [Xerox Learning Research Group 1981, 36–37] See the following examples from page 36 of the *Byte* article:

```
frame moveTo: newLocation
list at:index put:element
```

In the first example, frame is the receiver, moveTo: is the selector, and the single argument is newLocation. The second example “is a two-argument keyword message whose selector is made up of the keywords at: and put: and whose arguments are index and element. To talk about the selector of a multiple-argument keyword message, the keywords are concatenated. So, the selector of the fourth example is at:put:.” [Xerox Learning Research Group 1981, 37]

A key conceptual contribution of Love was to maintain a clear separation between the Smalltalk-style messaging syntax (which would be delimited by easy-to-recognize character pairs, square brackets “[]”) and standard C syntax [Biancuzzi and Warden 2009, 245; Love 2019, 18–19]. (OOPC originally used the { | and | } tokens instead of [] to delineate message expressions, but otherwise the usage was similar [Cox 1983b, 16–21].) The idea was motivated by the metaphor of reusable software components (i.e. objects) as interchangeable parts, which would allow for easy division of labor among programmers. When programmers saw brackets, they could instantly recognize that an object was being sent a message. Developers responsible for the larger design of an application would work mostly “within the brackets,” in Smalltalk-like style, connecting components together. More detailed, lower-level programmers would implement the objects themselves using standard procedural C syntax. The hybrid language easily allowed these two styles to coexist side-by-side, but individual programmers could specialize on the portions they were best at. Said Love, “It was a deliberate decision to design a language that essentially had two levels—once you had built up enough capability, you could operate at the higher level.” [Biancuzzi and Warden 2009, 245] The above Smalltalk examples, translated into Objective-C, would be rendered as:

```
[frame center];
[frame moveTo: newLocation];
[list at:index put:element];
```

Once inside brackets, these Smalltalk-style message expressions could be easily mixed into standard C expressions, like so:

```
p = [Container new:10];
```

This example comes from Cox’s article, *Message/object programming: An evolutionary change in programming technology* [Cox 1984, 57].

Contrast this message passing syntax with C++ and Java’s use of dot notation for method invocation. The following might be Java/C++ equivalents of the first two expressions, for comparison:

```
frame.center();
frame.moveTo(newLocation);
```

The assignment expression would no longer be a mixed C/Smalltalk-like expression but would simply be:

```
p = Container.new(10);
```

Because Java and C++ do not have keyword labeled arguments, there is no simple equivalent for the expression with multiple arguments. Rather, a modified method name might awkwardly incorporate both keywords, but would not indicate which argument was which:

```
list.atPut(index, element);
```

The precompiler automatically inserted `#include obj.h` at the beginning of each generated file to include a file that defined widely used types, particularly `id` (object reference) like this:

```
typedef struct Object *id;
```

so that `id` could be used in source code like this:

```
id x, y, z;
```

These examples above are slightly anachronistic, being given in Objective-C syntax (1983 and later) rather than the original OOPC syntax of 1981 to better facilitate the conceptual comparison to Smalltalk. The original 1981 version of OOPC simply used Unix text-processing utilities (*sed*, *awk*, etc.) and shell scripts to copy text to the output intact while transforming “message expressions” into calls on a subroutine that implemented Smalltalk-style messaging. The “message expressions” in OOPC were actually comma-separated lists of variables and string literals. For example, the precompiler would convert the OOPC statement:

```
z = {|x, "doThis:", y|};
```

to C like this:

```
z = _msg("doThis:", x, y);
```

Thus in original OOPC syntax, the simpler code examples above would be:

```
{|frame, "center"|};
{|frame, "moveTo:", newLocation|};
p = {|Container, "new:", 10|};
```

True Smalltalk-style message expressions between the brackets did not appear until the first version of Objective-C in 1983. The most significant difference can be seen in keyword expressions with multiple arguments. Original OOPC syntax simply took a selector (which contained all keywords concatenated together) as a string in the second argument after the receiver, which was the first argument. All subsequent arguments were simply listed after. As the above examples show, for unary expressions or expressions with a single keyword argument, this did a reasonable job of mimicking a Smalltalk-style message expression. However for multiple arguments, this was awkward, similar to how it would look in dot notation:

```
{|list, "at:put:", index, element|};
```

Examples of this can be found in the 1983 ACM SIGPLAN paper Cox published about OOPC [Cox 1983b, 16–18, 21]. Compare this again to the Objective-C version, which implements a true Smalltalk-style keyword expression:

```
[list at:index put:element];
```

Other minor syntactic differences between OOPC and Objective-C exist, though conceptually, the differences are not significant. Different tokens (`<+>` and `<->` instead of `+` and `-`) are used to identify class (i.e. factory) and instance method definitions, and the entire class definition is enclosed by `<{>` and `<>`, whereas in early Objective-C, these tokens are `=` and `:=`: [Cox 1983b, 17; Cox 1984, 55; Cox 1986, 84]. This was later changed by Steve Naroff (see section 10) to the current `@implementation`

and @end, which is reflected in the second edition of Cox's book *Object-Oriented Programming: An Evolutionary Approach* [Cox and Novobilski 1991, 88]. In addition, while the token for end-of-line terminated comments (/#) remained the same between OOPC and 1983 Objective-C, by 1986 it had changed to // to mirror C++, which derived it from BCPL [Cox 1983b, 17; Cox 1984, 55; Cox 1986, 60–61; Stroustrup 1993, 21].

Once messaging was working, Cox extended the precompiler with syntax for defining classes. These were transformed into cross-linked class and metaclass structure definitions to maintain Smalltalk's instance-class-metaclass and class-superclass relationships to structures accessible at run time [Goldberg and Robson 1983]. Because the 1981 *Byte* article wasn't clear about how this linkage worked, Cox obtained help from Adele Goldberg at Xerox PARC, who introduced him to Stoney Ballard, who with Stephen Shirron was developing a Smalltalk-80 implementation for the VAX at DEC [Krasner 1984]. Being familiar with both C and Smalltalk, Ballard helped Cox flesh out a solution over an extended telephone call.

This version of OOPC was ultimately adopted and used throughout the Programming Technology Center for several research applications in spite of its many shortcomings. For example, nested message expressions were not supported because of the lack of a real parser, and performance was poor because message selectors were represented as C strings. Since C string addresses are not globally unique, this required a full string comparison in the innermost message dispatch lookup.

Although Love's group was committed to Unix, other parts of the organization decided to deploy to proprietary IBM and DEC systems, causing friction within the team. Moreover, it soon became clear to Love that ITT was not going to beat AT&T in the U.S. telecommunications business, and he began to see that the Programming Technology Center would eventually be defunded. In 1983, Love left ITT for Schlumberger-Doll Research Labs in Ridgefield, Connecticut, with the aim of ultimately reassembling his former PTC team at Schlumberger. Cox joined him soon after. At Schlumberger, Love became a beta tester for Smalltalk-80 and thus among the first commercial users of Smalltalk, using it on Xerox Dolphin workstations within a Lisp-dominated AI group for oil field services. Cox established a Unix bridgehead in Schlumberger's VMS/Fortran dominated production operations. Love continued to believe, however, that there was a market for simpler object-oriented solutions that could coexist with legacy languages and applications within conventional operating systems and computer platforms, as opposed to Smalltalk which was a unique standalone system developed on custom Xerox hardware. Later in 1983, a friend of one of Love's former ITT colleagues then at Philips Labs in Eindhoven asked Love and Cox to conduct a six month study of their purchased computer-aided design (CAD) tools and determine whether it would be practical to build their own in-house tools to replace them. With this productivity support contract with Phillips and several of its subsidiaries in hand, Love and Cox resigned from Schlumberger, took out second mortgages, and founded Productivity Products International (PPI). PPI was essentially the reconstitution of the former ITT Programming Technology Center, as much of Love's former group joined the new startup [Love 2019, 10, 13].

4 PRODUCTIVITY PRODUCTS INTERNATIONAL AND OBJECTIVE-C (C. 1984–1985)

PPI began as a productivity consulting company, with plans to build software products that would help customers improve productivity. Tom Love worked on the consulting business side, acquiring contracts with clients including Philips and a Swedish bank. Cox worked on the technical products side, with his first task to recreate his Object-Oriented Pre-Compiler from a clean slate to avoid legal entanglements with ITT or Schlumberger. Love felt that to attract customers and investors, the language needed a better name than "OOPC," so Cox renamed the language "Objective-C."

As discussed earlier, Cox published *Object-Oriented Programming: An Evolutionary Approach* in 1986, which was his book-length treatment of the vision behind Objective-C: a tool to unleash

the “Software Industrial Revolution” through object-oriented, reusable software components or “Software-ICs.” [Cox 1986] Paradoxically, the revolutionary potential of Software-ICs would be made possible by a merely evolutionary addition to existing languages, grafting the Smalltalk-style message/object model onto C to create the hybrid Objective-C. It was in this context that Cox thought of Objective-C as a “soldering iron,” used to glue Software-ICs together [Biancuzzi and Warden 2009, 260]. While the language was sold as a product in its own right (at \$2,500 per compiler license), the real purpose was to use it to produce and glue together the component libraries (marketed as “ICpaks,”) that would constitute PPI’s main product line.

Objective-C versus C++. Cox contrasted this lightweight approach explicitly from C++. If Objective-C was a simple soldering iron, Cox likened C++ to a “fabrication plant.” [Cox 2016, 29–30, Biancuzzi and Warden 2009, 263] Both are hybrid languages that extend procedural C with object-oriented features, both began as precursor languages implemented via the C macro preprocessor (C++ evolved from C with Classes, Objective-C from OOPC), and both were developed in roughly the same period (C++ from 1979–1985, Objective-C from 1981–1984). Objective-C and C++ both became commercially available a year apart from each other (1984 and 1985 respectively). Yet C++ and Objective-C are very different languages, with different philosophies and goals.

As explained by Bjarne Stroustrup, C++’s overriding goal was to add abstract data types to C without compromising C’s performance, especially run-time efficiency. “I was very concerned that improved program structure was not achieved at the expense of run-time overheads compared to C,” wrote Stroustrup [1993, 5]. C++ patterned itself on Simula rather than Smalltalk, including its use of static or “strong” type checking, to improve safety and avoid run-time type checking [Stroustrup 1993, 30–31]. Lastly, C++’s focus on performance and C compatibility led it to favor static or “early” binding over dynamic or “late” binding, in other words, when “binding of a message to its target routine is done at run time.” [Cox 1983b, 15] As Love explained in his book *Object Lessons* [Love 1993, 29], with dynamic binding, “the particular method [or function] executed in response to a message is not known when the program is compiled. Instead, objects determine which method to execute at runtime; that is, as the code executes.” With static binding, the implementations for functions are chosen at compile time. C with Classes had support only for early binding. C++ added “virtual functions,” which enabled some of the semantic behavior of dynamically bound polymorphic methods as in Smalltalk, but precomputed the dispatch tables at compile and link-time, reducing run-time overhead at the cost of not being able to fully duplicate the semantics of Smalltalk’s dynamic dispatch. Unlike in Smalltalk and Objective-C, where all methods are virtual, virtual functions in C++ were not seen as an important feature initially and Stroustrup received some pushback to adding them [Stroustrup 1993, 23].

Contrast this with Objective-C, whose goal was to implement the Smalltalk message passing model on top of C. Rather than optimize for run-time efficiency, Objective-C’s goal was to add the programmer productivity advantages of Smalltalk’s dynamically typed and bound system to C. This was rooted in very different notions of what “object-oriented” meant. Although Stroustrup stated that the addition of virtual functions differentiated C++ from the merely “object-based” C with Classes [Stroustrup 1993, 22], elsewhere he wrote “object-oriented programming is programming using inheritance.” [Stroustrup 1993, 30] By contrast, Alan Kay, who coined the term “object-oriented programming,” has stated that “OOP to me means only messaging, local retention and protection and hiding of state-process, and extreme late-binding of all things.” [NA Ram 2003] Inheritance was not even a feature of the original version of Smalltalk, Smalltalk-72, only being added in Smalltalk-76 [Kay 1993, 84]. Additionally, C++ supported multiple inheritance, while Objective-C, like Smalltalk-76, supported only single inheritance. (NeXT later added “protocols” which allowed multiple inheritance of interface, but not implementation.)

Similarly, for Cox, dynamic binding was essential. “Smalltalk’s dynamically bound message/object paradigm solves key problems that can prevent us from building highly malleable, reusable software. Static binding requires all data types to be known at compile time. This makes environmental code explicitly dependent on the type of contents known when the code was developed—when new data types must be introduced, the change ripples throughout the environment. The late binding provided by messaging weakens this dependence so that environmental code can often be reused without change as new data types are added over time.” [Cox 1984, 61] Cox’s concern here was for the flexibility and maintainability of the program in the face of future changes in requirements. In 1986, Cox similarly wrote, “statically bound languages are extremely poor for building loosely coupled collections... When every data type is known when the code is compiled, static binding works. Otherwise binding must be done dynamically, period.” [Cox 1986, 24] Tightly-coupled data structures such as this impede malleability of the code, preventing reuse and making the creation of “Software-ICs” difficult. Similarly, Love in 1993 wrote, “Deferring until runtime the binding of procedure names to procedure invocations allows software to be malleable in the face of changing requirements.... Because messaging specifies only the desired response, dependencies of type cannot spread through a system. Instead they remain within objects. Therefore, programmers can install new data types in working systems without altering, and possibly disrupting, the rest of the system.” [Love 1993, 30] These benefits to programmer productivity were deemed by Cox and Love to be more important than raw efficiency. “Message/object programming seeks to remove work from the user and give it to the machine, so it trades machine cycles to gain the benefits described in this article.” [Cox 1984, 60] Indeed, optimization was not a concern when Cox was developing the original OOPC: “OOPC used full-length strings for command names, so the dispatch tables are searched via string comparisons with the command name — a substantial piece of overhead that I never moved to compile-time only because speed was sufficient as it stood.” [Cox 1983a, 60] (This was later changed for efficiency, see section 5 “Developing Objective-C” below.) Raw performance was not seen as much of an issue due to the hybrid nature of the language. When efficiency was needed, a programmer could use procedural C and static binding; the flexibility to mix and match styles tailored to the nature of the task was a benefit of this hybridity: “Objective-C is a hybrid of two languages of opposite philosophies, each designed for very different jobs. Programmers have considerable freedom to apply each tool to the job at which it excels. For example, a compiler developer would normally favor C constructs over message expressions within the lexical analyzer because this code requires maximum efficiency—C’s forte—and little bulk and complexity. In the rest of the compiler, these factors flip in importance, and the Smalltalk-80 coding style, with heavy use of message expressions, would be favored.” [Cox 1984, 60]

Thus we can see that C++ and Objective-C made very different trade-offs in design. Modeled on Simula, C++ was designed to be as fast as C itself, requiring no additional overhead at run-time, and also more type-safe than Smalltalk, though with necessary compromises for C compatibility and performance. Objective-C, modeled on Smalltalk, was designed to provide flexibility, promote code reuse, and enable software componentization in the name of improving programmer productivity, at some cost to raw performance, through the use of a dynamic runtime library. C++ favors early/static binding while Objective-C favors late/dynamic binding. This difference would be crucial to NeXT’s adoption of Objective-C over C++. (See section 8 “Early Adopter Feedback” below.)

5 DEVELOPING OBJECTIVE-C

Without a background in compiler development, Cox had to learn as he went, beginning by building a lexical analyzer and parser using the tools *lex* and *yacc* and using Unix source code as a test case. This initial work was done from home and then from a rented dentist’s office in Southbury, CT, on

a Fortune Unix workstation with 5.25" floppies for storage. Once the analyzer was recognizing C source correctly, Cox added support for message expressions, like this:

```
z = [x doThis]; /* unary expression */
z = [x doThis:y]; /* binary expression */
z = [x doThis:y with:z]; /* keyword expression */
```

C's variable-length argument lists were supported as follows:

```
z = [x doThis:y, a, b, c];
z = [x doThis:y with:z, a, b, c];
```

The full “compiler” was a script that inserted the parser between the first and second stages of the conventional C precompile-compile-assemble-link-execute tool chain. The parser detected and reported syntactic errors, built a syntax tree for valid programs, and walked this tree to recreate the original program (minus comments) while converting message expressions to invocations of the messaging subroutine, `_msg`. For example, the last example would be converted as follows:

```
z = _msg(_selectorTable[23] /* doThis:with: */, x, y, z, a, b, c);
```

The first argument supports a critical performance optimization to remove string compares from the speed-critical messaging operation. The original OOPC design generated message expressions as `{_msg("doThis:with:", x, y, z, a, b, c)}` which required the messaging routine to compare strings during message dispatch. This was extremely slow. When Cox first redesigned OOPC into Objective-C, the compiler managed a global database of unique message selectors (see `_selectorTable` above) entirely at compile time, leading to significant performance gains over the OOPC approach. However, this optimization eliminated some of the flexibility of the string comparison approach, creating a “fragile selector” issue that was problematic for multi-person (and cross-organization) development and deployment. This would be changed later by Steve Naroff (see section 9 “Solving the Fragility Problem” below) after PPI became Stepstone to have the compiler manage a unique selector table for each class, which would be scanned and rewritten at program startup-time to give each selector a unique address.

A less obvious optimization arises from the fact that messaging actually will not work as implied by the above C examples. C pushes arguments onto the stack before calling a subroutine and pops the stack when that routine returns. The messaging routine searches for the message's selector in the class structure of the message's receiver (and if required, its superclasses) to find the C function for that selector. The problem is how to call that function without disrupting the stack, which is already exactly as the C runtime expects it to be. After some experimentation with other alternatives, Cox settled on compiling the message routine to assembler and manually changing the call instruction (JSR) to a goto (JMP) instruction. The advantage was that messaging performance was extremely high, only about 30% less efficient than an ordinary C function call. However over time, the market seemed to value portability over performance. The compiler was then changed to emit two subroutine calls per message; one to lookup the target function based on the selector and the second to actually call that function, thus eliminating the more efficient but less portable hand-tweaked messaging routine.

Aside from the above syntax for message expressions, the early compiler also supported constructs for defining classes, instance variables, and class or instance methods. To avoid conflicts with ordinary C names, single-character tokens (`=`, `+` and `-`) served as delimiters in all cases. For example:

```
// Define the SinglyLinkedListNode class as a subclass of Object, with
// an instance variable, successor.
= SinglyLinkedListNode : Object { id successor; }
```

```

// The new method creates new instances with successor containing nil
+ new {
    id newObject = allocNilBlock(self->clsSizInstance, 0);
    newObject -> isa = self;
    return newObject;
}
// Return the contents of the successor variable
- successor { return successor; }

// Set the successor to the argument and return successor's prior contents
- successor: aSinglyLinkedListNode {
    id tmp = successor;
    successor = aSinglyLinkedListNode;
    return tmp;
}

// Return the last item in the linked list
- tail { return successor == nil ? self : [self successor]; }

```

This example demonstrates the “soldering iron” philosophy that governed the language’s early evolution. Of the three special characters (in the first column in this example), ‘=’ signals a new class definition, ‘+’ a class (i.e. factory) method, and ‘-’ an instance method. Everything else is ordinary C code except for the message expression in the last line delimited by ‘[’ and ‘]’. It also adopted the ‘//’ commenting convention from C++. Low-level classes such as this one consisted mainly of C, much as low-level computer components are fabricated on silicon fabrication lines. High-level classes would typically be assembled from these lowest-level classes by means of Objective-C message expressions, during which the language serves as a “soldering iron.” At this stage, the full “compiler” was a source-to-source translator that operated within the traditional multi-stage C compilation chain like this:

```
Preprocessor -> Translator -> Compiler -> Assembler -> Linker
```

The translator was written in Objective-C. It used a number of compiler-specific classes for each node in the syntax tree which were in turn based on a general-purpose foundation library. This was modeled after the Smalltalk Object, Array and Collection classes. Thus structural changes to the language often required bootstrapping the support classes, either by using the last working version of the language to compile them or by editing their generated C source code by hand.

In Smalltalk, all classes inherit from the Object class, which is therefore a natural place to support broadly useful features like reflection and the object serialization mechanism. This was modeled after Smalltalk’s `saveAs/readFrom` approach and similar to the serialization mechanism used later in Java. It provided an automated, ever-present way to convert an arbitrary graph of objects into a serial stream from which the graph can be reconstructed later. These features, present in Smalltalk and absent in C, facilitated inheritance which Cox had originally been fascinated by when he created Objective-C. (In later years he came to the opinion that inheritance was much less important than he had originally thought. [Biancuzzi and Warden 2009, 259; Cox 2016, 50–51])

6 STEPSTONE AND SOFTWARE-ICS (C. 1985–1990)

By 1985, PPI was profitable with several major contracts with world-class companies and a popular line of software training courses. It had leveraged its services profits into a first product (the

Objective-C translator or “compiler”) thereby justifying the emphasis on productivity products explicit in the company’s name. It was time for stage two.

Tom Love’s focus shifted to contacting major venture capitalists for the funding to market these products more aggressively than the services revenue could support. The main problem was that the only product PPI had to show was the Objective-C “compiler,” bundled with a rudimentary set of foundation class libraries. Love’s pitch would shift PPI’s focus to selling a more feature-rich set of class libraries separately, positioning the company not as a purveyor of a new programming language, but as a purveyor of quality software components that would save the customer time and money.

The modest objectives of the early language actually worked in PPI’s favor. Positioning Objective-C as a modest tool (“soldering iron”) for assembling existing libraries more effectively actually helped to focus attention on the components, not the language. In other words, the compiler would be a loss leader. The real focus was to be on the libraries (ICpaks) that the compiler made possible, plus other productivity-enhancing products that could be developed over time. The venture capital community strongly resonated with the notion of launching a software components company that might play the role that Intel (and other silicon chip vendors) played for the hardware industry.

The company moved from its original quarters in a renovated dentist’s offices to a rustic fire-hose factory in Sandy Hook, CT. This was a beautiful historic site with antique exposed beams and an industrial waterfall right outside Cox’s window. This drove a hydraulic electrical generator being renovated in the basement.

The first round of venture capital brought in a new board of directors. The board elected a new CEO, Dennis Sisco, who immediately added considerable staff in development, testing, sales and marketing. Most of the new development staff were former colleagues from ITT, including Alan Watt, Donn Combelle (an ITT executive who’d decided to become a C programmer when he retired, who led PPI’s Quality Assurance [Love 2019, 13–14]), and Ken Hamer-Hodges (one of System 1240’s lead designers). One exception was Steve Naroff, whose work will be discussed in more detail later (see section 7). The board also insisted on hiring a firm to find the company a new name, which suggested “Syzygy,” but this was rejected by Tom Love [2019, 18]. Sometime between November 1987 and July of 1988, the company’s name would be officially changed from Productivity Products International, Inc. to The Stepstone Corporation.

Cox turned the compiler and foundation library over to the development team (managed by Hamer-Hodges) and focused his attention on new product research. Naroff and Watt took over responsibility for the Objective-C language, while Cox and others worked on the ICpak libraries, including a library of graphical user interface components modeled after Smalltalk’s. This was ultimately marketed as ICpak 201, alongside the foundation library, ICpak 101, which was bundled with the compiler. Creating a cross-platform graphics library was time-consuming and resource intensive, as every platform had its own proprietary windowing system. MIT’s X Window system was not made public until years later. Stepstone developed a primitive graphics portability layer, based initially on Sun’s windowing system, and used that to build the user-accessible functionality of ICpak 201. Around this time Cox developed Producer, the Smalltalk to Objective-C converter, to keep the graphics API as close as possible to Smalltalk’s [Cox and Schmucker 1987].

At its height, Stepstone had hundreds of customers for Objective-C, including HP, Apple, IBM, Siemens, Philips, US West, Accuray (acquired by ABB Process Automation), Enator Functional Systems, Applied Intelligent Systems, Inc., the Port of Singapore, GE Medical, Prime Computer, Lawrence Livermore Labs, Artecon, Sun, Tektronix, NeXT, and Ford Motor Company [Love 1993, 84–94; NA Love 2019]. All purchased the compiler and later ICpaks, signing PPI and later Stepstone purchase contracts. These contracts were product purchases and associated contracts but generally not development contracts. NeXT and HP had corporate-wide licenses. NeXT had negotiated the

best deal, paying only \$5 per device running Objective-C. Apple had purchased a copy of Objective-C as early as 1983 for use in the Lisa, but ended up not using it for much other than getting PPI to test Apple's C compiler. HP was by far the largest customer for Objective-C, with multiple groups using the language, including HP Labs in Palo Alto, groups in Fort Collins, Colorado, the Lake Stevens Instrument Division, which built VISTA, a "window-based, interactive measurement" and the Instrumentation Group, which released a product called Visual Engineering Environment (VEE), a graphical development environment that allowed engineers to create iconic representations of laboratory instruments that could be used to control the real instruments [Love 1993, 86,93]. In 1989, IBM, HP, and NeXT were negotiating a joint press release committing to the bundling of Objective-C on their companies' workstations, but unfortunately the deal fell apart.

Although the graphics library was successful as a product, Cox believes that it proved to be a terrible business decision overall. Except for that one-line assembly language patch in the messaging routine, the compiler itself was quite portable, a few hours work at most. But porting the graphics library required a monumental effort to learn each vendor's graphics library and to build a portability layer to match it. But ensuring that customers' existing applications wouldn't break when they upgraded their native code or installed a new release of Stepstone's product was even more costly. Productivity plummeted as the testing team became overloaded and the development team was drawn in to build ever more elaborate test cases. Cox got involved too, building the automated unit test system used for this work.

Meanwhile, the Objective-C compiler's customers made extensive feature requests, which clashed with Cox's initial notion of a simple "soldering iron" glue language. Two prominent customers drove the development of Objective-C at Stepstone. The first was HP Labs, for research purposes. HP wanted Stepstone to add automatic garbage collection. Work along these lines was begun, but adding garbage collection to a C-compatible language was a difficult research problem, one that Stepstone was not equipped to solve. Another feature requested was Smalltalk's blocks (which are equivalent to closures or lambdas). Another early adopter wanted an Objective-C interpreter, and actually built one and asked for permission to market it. It turned out that he had started from an early version of Stepstone's proprietary compiler source code, which set off the investors' alarms. They eventually struck a deal in which he turned over rights to the interpreter in exchange for remuneration. However, this committed Stepstone to maintaining the interpreter, an effort which further drained its resources.

Cox soon inherited the source to the interpreter with the request to make it "fully compatible" with the compiler. As it turned out, the interpreter was built exactly like the compiler, around an expression tree that could be executed in place, with call-outs to whatever compiled code was linked into the interpreter when it was built. Cox got it working and turned it over to the development and testing groups for deployment. Although the interpreter was useful for demonstrating the graphics library, Cox felt that it was fragile, and seemed like too radical a departure from C's modest roots. The addition of this independent build environment was a further burden to the already overloaded testing group.

7 STEVE NAROFF JOINS PPI/STEPSTONE (1986)

As mentioned earlier, after PPI received venture capital funding and hired a programming staff, Cox turned over development of the language and compiler to Steve Naroff and Alan Watt. Prior to his hiring at PPI in November 1986, Naroff had been at HHB-Softtron working on CAD systems for digital circuit design on VAX and Unix workstations, and then at Cerikor, a Utah-based company making graphical CAD tools implemented in MAINSAIL, a MACHINE INdependent version of Stanford Artificial Intelligence Language (SAIL) created by the Stanford AI Lab. MAINSAIL was a powerful language designed to simplify the portability and maintenance of large programming projects. It

supported a rich declarative programming model, dynamic loading, runtime class identification, persistence, and garbage collection. Although not object-oriented per se, MAINSAIL's dynamic runtime model of programming lent itself well to an object-oriented style, and Cerikor built an object-oriented library on top of MAINSAIL. Nevertheless, Cerikor had been acquired by Hewlett-Packard. Despite its technical excellence, given MAINSAIL's niche position in the marketplace, HP ordered the newly acquired Cerikor to rewrite its product in the more mainstream C++. Given C++'s static model of programming, Naroff believed this to be a losing proposition and left HP.

Steve Naroff met Brad Cox and Tom Love at the first OOPSLA conference in 1986, and soon after interviewed at PPI's Sandy Hook, CT headquarters. Naroff had been initially taken with Cox's vision of object-oriented software components presented at OOPSLA, and was attracted to PPI's more practical bent, to make software for actual production environments rather than research. Naroff also liked the pragmatism of Objective-C, of bringing the benefits of object-orientation to C programming without all the overhead of research languages like Smalltalk. Having been burned by the failure of MAINSAIL, Naroff felt that trying to improve C would be the more successful approach. From PPI's perspective, Naroff's MAINSAIL and C/Unix experience would be extremely beneficial.

Naroff's first six months at PPI were spent porting the Objective-C compiler to various platforms for its customers, some of which were very niche. There was no prior documentation so Naroff had to document both the architecture and implementation so decisions could be made moving forward. Naroff became intimately familiar with the compiler's inadequacies as originally written by Cox. First of all, the "compiler" was at this stage still merely a translator that converted object-oriented message expressions into procedural C code, which was then handed off to a true C compiler. Worse, the translator could not detect errors, such as misspellings, in message names. The reason for this was in the language definition: there was no explicit interface declaration. Therefore a misspelled message expression received by an object was treated as simply a different message than the one intended, which would only be detected at runtime. In addition, Objective-C in its original incarnation was, like Smalltalk, completely dynamically typed (all objects were typed id). Naroff felt that Smalltalk's imperative model caused problems in a statically compiled, C-based platform, and pushed for adding static types (what he called "classified types," from MAINSAIL usage) so that the compiler could do rudimentary type-checking at compile time.

At a strategic level, the PPI engineering staff was torn between adding high-end features from Smalltalk (e.g., blocks) and improving C language integration. To provide leadership and stimulate discussion, Naroff and Alan Watt drafted an internal white paper (July '87) describing problems and feature requests [Appendix B: Naroff and Watt, Design Issues for Objective-C DRAFT July 1987, also available at [Naroff and Watt 1987](#)]. (See section 8 below.) Naroff tackled integration problems with C and Watt focused on new features. Naroff was specifically interested in adding explicit declaration constructs and solving fragility problems. After weeks of discussion, it became apparent to Naroff that his concerns were not gaining momentum at PPI.

8 EARLY ADOPTER FEEDBACK

In 1987, both Objective-C and C++ were in their infancy. NeXT and Hewlett-Packard were highly visible early adopters of Objective-C, but their reasons for adoption were very different. As mentioned earlier in section 6, HP was using Objective-C in a research setting, for "pilot" projects and prototyping, and pushed for research features like automatic garbage collection.

NeXT's interest in Objective-C came from a more pragmatic, production angle. NeXT was developing Steve Jobs' next big product, a Unix-based workstation with a graphical user interface and object-oriented development tools aimed initially at higher education. In the academic market, object-oriented programming would allow students and faculty to more easily write their own

custom applications. Making object-oriented programming a major feature of NeXTSTEP (the NeXT operating system) was championed by William Parkhurst, who wrote the first version of the Application Kit, or AppKit, an object-oriented library of graphical interface objects, akin to PPI's ICpak 201. Languages such as Object Pascal and even Display PostScript (which was being used for NeXTSTEP's graphics) were considered. However, Objective-C was ultimately chosen because C was the base language of Unix, and the NeXTSTEP OS was based on a hybrid of BSD Unix and Mach from Carnegie Mellon University.

NeXT did not choose C++. Where for many companies C++ represented the path of least resistance, NeXT preferred Objective-C's simplicity but especially dynamic binding, which was a critical feature for the implementation of the AppKit and NeXT's Interface Builder development tool. C++'s static binding prevented NeXT from updating its frameworks without breaking compatibility. (See following section 9 below.) Moreover, NeXT wanted to evolve Objective-C to harmonize with their new software platform. They wanted the language to conform to needs of the platform (not vice-versa). From their perspective, C++ was a complex language and controlled by AT&T, a much larger company than PPI. NeXT was unlikely to influence the direction of AT&T/C++, but it had much more leverage over the direction of PPI/Stepstone and Objective-C.

Thus, despite significant early problems with Objective-C, NeXT's decision would ultimately pay off as it gained an advocate (Naroff) for its needs at PPI. This began when Naroff and Alan Watt drafted the white paper of July 1987 outlining key issues with Objective-C. The following excerpt from the white paper in Appendix B, page 35, (also available at [Naroff and Watt 1987](#), 2) describes such problems:

- The system needs to be re-compiled frequently.
- Collisions over selector usage when integrating Software-ic's [sic] 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-person development; this creates confusion.
- The [PPI] 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.

In addition, the paper outlined possible new features, many requested by customers, some of which would break binary compatibility and require recompilation. Some of these were indeed, as described earlier in section 6, attempted. These included:

- ANSI C compatibility.
- Garbage collection.
- Optional Static/Late binding.
- Multiple Inheritance.
- Blocks.
- Dynamic Linking. [Appendix B, 37–39, also available at [Naroff and Watt 1987](#), 5–9]

Subsequently Naroff organized a key meeting with NeXT engineers Steve Stone (who was working on the Objective-C compiler at NeXT) and Trey Matteson (on the AppKit team), focused on solving the problems Naroff and Watt had identified in the white paper. The meeting was both

timely and productive; it helped validate Naroff's thinking and establish priorities. NeXT was pleased to see Naroff anticipate their issues, but they wanted PPI to commit to solutions.

Naroff believed that collaborating with NeXT was mission critical to PPI and became a passionate advocate for NeXT inside PPI. NeXT was building critical pieces of their operating system on top of Objective-C, while HP was only playing with it in the lab and would likely (thought Naroff) never ship a product with it. PPI management did not share this opinion, believing that the big name HP, and not Steve Jobs' tiny startup, was the more important client. To rally support, Naroff requested that NeXT draft a letter (11/'87) outlining "showstopper" issues blocking NeXT's use of Objective-C (see section 9 below). NeXT's letter finally convinced PPI/Stepstone to pursue changes Naroff had lobbied for. As a result, Naroff became the primary engineering resource focused on solving NeXT issues.

9 SOLVING THE FRAGILITY PROBLEM – DYNAMIC SELECTOR RESOLUTION

As mentioned previously, the AppKit, implemented in Objective-C, was a central component of the NeXTSTEP operating system, greatly simplifying the creation of applications with high quality graphical user interfaces. While NeXT utilized AppKit for their bundled applications, they realized it was impractical to copy AppKit into every application (the target computer only had 4 megabytes of RAM). As a result, NeXT scrambled to add support for shared libraries, a feature unavailable on Unix 4.3BSD.

In addition to reducing memory usage, NeXT needed to dynamically update system shared libraries without re-compiling dependent applications. The ability to add/move selectors (method names) without breaking clients was essential. As discussed above in section 5, Cox's original implementation of Objective-C stored message selectors in a single global table of unique strings at compile time. This approach improved efficiency over OOPC's previous system requiring expensive string comparisons during runtime message dispatch, but led to the "fragile selector" problem—selectors for new methods could not be added to a class without potentially breaking clients. A global table could not be made robust in the face of changes with the use of the development tools of the day such as make. Frequent recompilation and complicated build procedures thwarted day-to-day development, especially for multi-person and cross-organization development. For deployment, this problem was more than a time sink—it had implications on third party applications (i.e., source code NeXT didn't control). NeXT would not be able to update its libraries without breaking binary compatibility and requiring applications to recompile. Although many early adopters tolerated fragility, NeXT considered this to be a "showstopper" for a commercial software product—it needed to be fixed or they would have to abandon the language.

To solve these selector fragility problems, Naroff modified the parser to output a table of selectors into each compilation unit. At program startup, these tables were scanned and rewritten such that each selector was represented by a unique address. In other words, each object module had its own copy of the strings in its own selector table, and the launch time initialization routine would go across object modules to make sure each string was unique. These changes are described in Appendix E, "Design Notebook: Selector Mapping in the Objective-C Run-Time Support System" or spec.runtime. This approach worked well—the only downside was initialization overhead and memory footprint. For most applications, the overhead was acceptable (1-2 seconds on the NeXT Computer's 25Mhz Motorola 68030 processor). This was the best that Stepstone could do at the time, given that it only controlled the compiler. To further optimize selector resolution would require NeXT to later extend its static linker to unique selector tables across module boundaries. With this work done by the linker instead of the compiler, the only string pools that needed to be unique would be in shared libraries. This optimization would result in a dramatic 50% savings, but was restricted to NeXT's version of the language and tool chain.

Solving this fragility problem was mission critical for NeXT. It completed Objective-C's support for dynamic binding (and would make possible dynamic linking) and inspired confidence in Stepstone's ability to support NeXT's needs.

As of 2006, C++ still suffered from a "fragile method" problem based on its use of virtual method tables for method dispatch. According to Naroff, virtual table dispatch is fragile by design, depending on offsets computed at compile time for efficiency, but this means that the size of compiled base classes in shared libraries cannot be added to by clients later at runtime, complicating shared library deployment. Many companies during the 1990s (including IBM, Taligent, and Be) attempted to fix the C++ fragility problem. However the static nature of the C++ language makes solving this problem more difficult, says Naroff. For example, to support inlining and templates, C++ encourages code to be distributed in header files and copied. Fortunately for NeXT, Objective-C's dynamic object model and modest language definition contributed to solving this problem in a timely manner.

10 ADOPTING A DECLARATIVE PROGRAMMING MODEL

Coincident with solving the fragility problem, Naroff added explicit declaration constructs for class interfaces, method prototypes, and classified (static) typing. Here is a code fragment:

```
@interface Person : Object
{
    String *name;
    Date *birthdate;
}
+ new; // default return type is "id" (unclassified i.e. dynamic)
- (String *) name; // return type is String (classified i.e. static)
- (Date *) birthdate; // return type is Date (classified i.e. static)
@end;
```

@interface declares a new class:superclass pair, the curly braces declare the class's instance variables, and the +/- introduce each class/instance method prototype (respectively). Naroff was not a fan of the +/- syntax, however, as it was already in use for method definitions, but all the alternatives seemed too verbose.

This Person class could then be imported as follows:

```
#import "Person.h"

Person *employee = [Person new];
```

The #import directive was a minor, but useful extension to ensure that a header file is included once and only once (without using C-style #ifndef/#endif markers).

Naroff chose to implement static typing, *not* static binding or allocation. His focus was on diagnosing errors and documenting intention. Improving the performance of binding/allocation was not a priority. All messages remained dynamically bound and all objects were heap allocated. There was no desire to complicate the object model or introduce fragility. Although this disappointed some C++ customers familiar with static binding/allocation, Naroff felt it offered the right balance for a hybrid language with Smalltalk roots.

The splitting of the class interface declaration from the class implementation definition into separate .h and .m files dates from this change, as does the shift from the definition start and end indicators = and =: to what eventually would be @implementation and @end. (There may have been a period in between where the indicators were @module and @.) These changes are described in Appendix D, "Functional Specification: Class Declaration Syntax and Run-Time Selector Mapping in the Objective-C Language," or spec.language, pages 55–58.

11 STEPSTONE VISIT TO NEXT (1988)

To help validate the NeXT version of the Objective-C language, Naroff visited NeXT Palo Alto headquarters (3/'88) and worked closely with Steve Stone. NeXT's product plans were still a closely guarded secret; engineering offices were strictly off limits. Instead, Naroff was set up in a business office with a Sun workstation and sample source code. After only a couple of days, the code was converted and compiled without error. For Naroff, the work was exhilarating—he felt an immediate kinship with NeXT engineers.

Once the code was working, Bud Tribble (NeXT's Vice President of Software and former manager of the original Apple Macintosh Software team) arranged a briefing between Naroff, NeXT CEO Steve Jobs and Tom Love, Stepstone's co-founder. Jobs thanked Naroff for his efforts and mentioned his engineers were thrilled with recent progress. Jobs then advised Love that Stepstone's development efforts were too scattered. He urged Love to focus on making the core language "great" and stop "wasting time" on ICpaks. Jobs expressed dissatisfaction with error diagnostics, compilation time, and debug support. His message was crystal clear—if Stepstone did not focus on polishing Objective-C's implementation, NeXT would rethink its language strategy and development efforts.

The Steve Jobs meeting had a profound impact on Naroff—his ability to articulate problems with Objective-C was impressive. It was obvious to Naroff that Jobs held at least two roles—CEO and what Naroff calls "CEA" (Chief Engineering Advocate). His hands-on leadership of engineering issues was energizing for Naroff, who was more accustomed with CEOs who avoided technical details. Before returning home, Steve Jobs asked Naroff to interview with his team. Four months later, Naroff moved his family cross-country to begin employment with NeXT.

12 STEPSTONE'S DEMISE (1990–1994)

Stepstone's resources were stretched too thin and its products were not bringing in enough revenue to cover expenses. Compiler sales were steady but \$2,500 per compiler sale did not go far enough. According to Cox, the compiler license to NeXT did wonders for morale but very little for the bottom line. Sales of the graphics library and interpreter brought in far more revenue per sale (\$30,000 for ICpak2 according to Love [2019, 16–17, 31]), but the cost of porting and testing them on the ever-accelerating flood of new and incompatible computer platforms and windowing environments had become crippling. Meanwhile, Tom Love became involved in a dispute with the Stepstone board and CEO. Love wanted to lower the price of the compiler to \$100 while keeping ICpak pricing at \$30,000 per license, but after failing to convince the board, resigned. CEO Dennis Sisco was himself replaced only a year or so later. Naroff's departure triggered a lawsuit between Stepstone and NeXT, which was resolved quickly so that Naroff could join NeXT. By 1990 most staff had been laid off until only six remained, at which point Cox, the only remaining co-founder, also left the company. In August of 1994, Tom Love, now a managing director at Morgan Stanley, was approached by Steve Jobs about purchasing NeXT workstations, and Love started a trial comparing Morgan Stanley's existing development tools with NeXT's. However Love told Jobs that he would not do business with NeXT unless Jobs resolved an outstanding contract dispute with Stepstone, in which NeXT had not paid Stepstone the agreed upon royalties owed for Objective-C. This led to a negotiation between NeXT and Stepstone (conducted by third Stepstone CEO K.K. Tan and the VC-controlled board), resulting in the complete sale of all rights to Objective-C to NeXT for a fixed price, a deal Love would never have agreed to had he still been with the company. Stepstone folded soon after.

From Naroff's perspective, Stepstone was not a well-managed company, lacking leadership and focus. Neither founder involved himself much with the day to day management of the company, he says. Tom Love was busy traveling, selling consulting contracts and compiler and ICpak licenses,

and eventually was forced out of the company. Cox concerned himself with doing his own research but collaborated only minimally with the development staff, treating his job somewhat like an academic appointment. The company's actual management had come from ITT, with a lack of entrepreneurial hunger or focus. Management pushed to maximize sales regardless of the workload on engineering, advocating for more ICpak products when the existing ICpaks (and the compiler) were still buggy and inadequate. Naroff's porting work on the compiler was a similar case. The compiler was ported to some computer platforms whose user bases were too small to financially support the engineering effort to port it, yet the sales team only saw a win in the sale column, rather than the losses the engineering effort would incur.

Looking back on it today, Naroff feels that, technically, the ICpak products were poorly executed. (Love disagrees, citing happy customers.) Their failure in the marketplace, Naroff believes, was not due to the advent of the open source model, which in the late 1980s was only just beginning. True, it was a difficult sell for clients to base their own software products on object libraries for which they did not have source code; bugs could not be fixed, and potential design flaws could not be corrected for. This, however, magnified a larger issue. No one at Stepstone had developed any applications using the ICpaks. Without this experience, the ICpaks' designs were purely speculative and did not well anticipate actual usage. Well-built code libraries are iterated over time in the context of actual use by multiple client applications, whose different needs signal to the library what functionality should be made general and what should remain specific (and thus not part of the library). To design a library prior to building multiple proof of concept applications along with it is to doom the library to a prematurely designed architecture inadequate for real world needs. And without access to source code, clients would not be able to modify the library to correct such design flaws, making it unlikely for any company to want to base its future on such a risk. Today Naroff feels that no one at Stepstone, not even himself at the time, understood this [Naroff 2018, 24–25]. This is why, from Naroff's perspective, despite NeXT's AppKit being proprietary, because it was carefully designed and then tested by both NeXT's own applications and eventually third party ones, it successfully weathered competition from open source alternatives to survive today in Apple's macOS.

Love, Cox, and Naroff thus all have different perspectives on the reason for Stepstone's failure. For Naroff, it was due not to the concept of software components, but the execution, on both the business and technology sides. Without strong direction from the founders, the company became ravaged by managers put in place by the board. Love contends that the management brought in by the investors was inexperienced with both the technology and the market for software, and therefore could not make the necessary long-term decisions (lowering the price of the compiler to drive adoption) for the survival of the company. Cox, meanwhile, feels that the engineering team got bogged down by porting the graphics libraries to increasingly smaller and less consequential platforms, and adding complex features, such as garbage collection, blocks, and an interpreter, that only one or two customers requested.

13 OBJECTIVE-C AND THE FREE SOFTWARE MOVEMENT (1988)

As mentioned earlier, Naroff's hiring at NeXT triggered a brief lawsuit between Stepstone and NeXT, due to a non-compete agreement barring Naroff from working on Objective-C at NeXT. Jobs had NeXT's counsel Gary Moore handle the suit. Because NeXT had a source code license for Objective-C from Stepstone, and because Naroff had no knowledge of Stepstone's ICpak implementation, NeXT believed there would be no misuse of trade secrets. Nevertheless, NeXT agreed that any improvements Naroff made to Objective-C at NeXT for 12 months would be licensed back to Stepstone royalty-free, and that the existing license agreement between NeXT and Stepstone would be extended to the end of 1989. The suit was resolved, allowing Naroff to disclose to NeXT only

Stepstone proprietary information regarding Objective-C, and Naroff began work at NeXT in August of 1988.

Naroff immediately joined the NeXT operating system group to take over the compiler effort from Steve Stone, initially fixing bugs in the Stepstone Objective-C translator. (Eventually Naroff would become the head of a four-person Tools team.) Naroff also wrote a new Objective-C runtime that was better optimized for the Mach kernel, replacing the Stepstone runtime, and helped Kevin Enderby to improve the linker. His most important early task, however, would be to improve the translator’s integration with the GNU C Compiler (GCC), which NeXT had chosen for its C compiler. GCC had been developed by Free Software pioneer Richard Stallman. Naroff would be responsible for the Stepstone translator, Objective-C runtime, and the performance and correctness of GCC, a complex, native compiler. The NeXT platform was only three months from introduction and plenty of compiler bugs nagged developers.

GCC was first released in 1987. Although it was far from complete, GCC appeared well crafted. Nevertheless, the idea of free software collaboration seemed radical and unsettling at the time. The decision to use GCC was risky. Naroff immediately established a working relationship with Richard Stallman. Stallman turned out to be one of the most prolific programmers Naroff had ever worked with, often turning around bug fixes within an hour or so, much more agile than the average commercial company. Within a few months, Naroff was convinced that GCC and the free software development model was a huge competitive advantage for NeXT.

NeXTSTEP 0.8 pre-release shipped with Stepstone’s translator and GCC. For the 1.0 release Naroff added native Objective-C support to GCC (generating parse trees directly in GCC, removing the need for a translator). The benefits for the developer were faster compilation, unified error diagnostics, and better debugging support. In addition, having one parser greatly reduced NeXT’s maintenance effort. To avoid fragmenting the language, Naroff asked Stepstone to collaborate with NeXT on GCC. The free software “pitch” didn’t resonate with Stepstone—they stuck with the separate translator and eventually decided to track NeXT’s feature additions. Nevertheless, this would lead to a fork between NeXT and Stepstone’s versions of the language.

NeXT contributed the Objective-C front-end back to Stallman’s Free Software Foundation. According to Naroff, NeXTSTEP was the first commercial operating system to embrace GCC, helping validate what would in later years come to be known as “open source” development for commercial use.

14 CATEGORIES (1989)

NeXT’s InterfaceBuilder (“IB”), developed by Jean-Marie Hullot, was a development tool that pioneered designing sophisticated graphical interfaces interactively, without traditional programming, providing immediate productivity benefits. IB allowed “mere mortals” (Steve Jobs’ characterization) to instantiate user interface elements, make connections between objects graphically, and test the result immediately. No sub-classing or recompilation was necessary. It showcased Objective-C’s class metadata, reflection, and serialization—the “magic” that enabled IB’s groundbreaking features.

InterfaceBuilder’s feature set and implementation influenced the language in many interesting ways. For example, IB “outlets” required metadata/API extensions to dynamically get and set the values of instance variables. A more visible outgrowth was Objective-C’s support for class *categories*. Hullot had originally implemented a prototype of IB in Common Lisp Object System (CLOS), which freely allowed the addition of methods to existing classes at runtime. Without this feature in Objective-C, IB’s initial implementation in Objective-C used the “poseAs” feature (the ability for one class to take another’s place in the inheritance hierarchy, which was implemented in the root Object class) to augment AppKit classes with various behaviors (such as custom inspectors and editors).

Although this feature borrowed the name “categories” from Smalltalk, it in fact meant something very different. In Smalltalk, a “category” was not a language feature but a user interface feature of the development environment, to group a set of related classes (and methods too) together in the Smalltalk browser to help reduce scrolling [Tesler 1981, 122, 124; Tesler 2013, 18]. The original PPI version of Objective-C contained an unrelated notion of “categories” that also grouped together related classes, but was used by the compiler to generate the global table of unique selectors. This second notion of “categories” was also not a formal part of the language definition but was an implementation detail of the compiler. Later PPI releases of Objective-C renamed the concept to “phylum” [Cox 1986, 84–88, see also Appendix E: spec.runtime, 67]. This “phylum” concept was eliminated when Naroff rewrote the runtime selector mapping system described above in section 9 (see Appendix D: spec.language, 58, and Appendix E: spec.runtime, 62, 67), and the section describing it was eliminated in the second edition of Cox’s book *Object-Oriented Programming: An Evolutionary Approach* [Cox and Novobilski 1991, 88–89]. Conceptually, “categories” in NeXT’s version of Objective-C were not directly related to either of these precursors but instead were inspired by the notion of “mixins” from CLOS.

The idea from CLOS of dynamically adding methods to an existing class/framework was very powerful. To better support the idiom used by Hullot for IB, Naroff added explicit support for class categories. This was a major new feature addition to the language at the compiler level, not merely in the runtime. Unlike “poseAs”, categories were far simpler to use (and did not alter the runtime class hierarchy). Here is an example category declaration:

```
@interface String (Draw_Methods)

- drawIn: (Rect)r;
- drawAt: (Point)p;

@end
```

Categories are then defined using “@implementation ClassName (CategoryName)”, so that the definition of the Draw_Methods category of String would be “@implementation String (Draw_Methods)”. Note that a category could only add behavior—it could not declare any new instance variables for the class. If a category required state, developers could use a hash table to associate the original instance with additional storage. In practice, this was uncommon.

Categories promoted an “open” class system. Extending classes defined by other implementers (*without* subclassing) was uncommon in class-based languages. Grouping methods by functionality resulted in a modular, more loosely-coupled system. By reducing subclass proliferation, categories fit well with NeXT’s “less is more” engineering culture. Categories became one of two major, critical additions to the Objective-C language idiom that became widely used by the AppKit and other frameworks at NeXT and later at Apple. In Apple’s 2014 Swift language, *categories* are known as *extensions* [NA Apple Inc. 2020a]. Microsoft’s C# includes this feature also as *extension methods*, implemented as a special kind of static method [NA Microsoft 2015], but C++ does not have a similar construct as of this writing.

15 OBJECTIVE-C++ (1989-1990)

In the summer of 1989, the Lotus “BackBay” team visited NeXT. The team was working on Improv, an advanced spreadsheet application with an innovative user interface. After struggling with an OS/2 implementation, the team decided to port the spreadsheet “engine” and rewrite the user interface using NeXTSTEP. There was only one problem—the spreadsheet engine was written

in C++, and NeXT didn't support C++. Since Improv was a potential "killer" application, Naroff believed supporting C++ in the compiler was becoming mission critical.

Fortunately, Michael Tiemann was developing G++ (a GCC-based C++ front-end). Naroff visited his newly formed open source company Cygnus Support. Although G++ was still a work in progress, Michael was confident it would stabilize quickly and meet NeXT's needs, and appeared eager to work with NeXT. Both NeXT and Cygnus wanted to prove the viability of open source development. Within 6-8 weeks, G++ compiled the Lotus spreadsheet engine, a huge milestone. This made it possible to build Lotus Improv on NeXTSTEP.

Lotus loved the InterfaceBuilder/AppKit/Objective-C triumvirate, but they quickly grew tired of writing glue functions (stubs in C) to bridge Objective-C and C++ code. Lotus requested seamless integration—they wanted to use both languages within the same source file. After much deliberation, Naroff became convinced that "Objective-C++" was both sensible and inevitable. Like it or not, C++ was quickly becoming the next generation of C in the wider marketplace. Here is a brief example of the language integration:

```
class ImprovEngine
{
public:
    void computeSomething(float f);
};

@interface ImprovTableView : View
{
    View *v;
    ImprovEngine *s;
}
- (ImprovEngine *) engine;
- (void) setEngine: (ImprovEngine *)e;
@end;
```

Here is an expression that illustrates the seamless integration:

```
[self engine]->computeSomething([textfield floatValue]);
```

Sending self the "engine" message returns a C++ object that implements the "computeSomething" member function. No bridging is necessary.

The Objective-C++ design point can be characterized as "peaceful coexistence." Both runtime systems were unaltered and mutually oblivious. Changing the C++ object model was a very tricky business with subtle semantic and performance implications. As a result, Naroff chose the most pragmatic and quickly useful course, to avoid subclassing between the object models. Rather, the inheritance hierarchies of the two languages would be orthogonal to each other, rather than merged, a much harder problem. The integration "magic" was isolated to the compiler front-end. The Lotus BackBay team was so impressed with the result that they awarded Naroff a special T-shirt for accomplishing a feat no one thought could be done. The resulting Objective-C++ integration was soon used by Pixar and by Adobe for Photoshop, and since by many others.

16 METHOD FORWARDING, SIGNATURES, AND PROTOCOLS (1990)

In 1990, the NeXT Workspace Manager (the equivalent of the Macintosh Finder) team was exploring distributed programming using remote objects. As the name implies, remote objects allow message sending between processes and machines. To enable this, NeXT engineer Bertrand Serlet (later Senior Vice-President of Apple Software) prototyped *forwarding*, an extension to the message

dispatch routine. Forwarding provided a low-level hook to enable proxy objects. Here is a simple delegate proxy that demonstrates forwarding:

```
@implementation DelegateProxy
{
    id delegate;
}
// the message dispatcher sends forward:: for unrecognized messages

- forward: (SEL)selector :(marg_list)method_arguments
{
    if ([delegate respondsToSelector: selector])
        return [delegate performv: selector : method_arguments];
    else
        return [self doesNotRecognize: selector];
}
```

Implementing a remote object proxy is far more complex. For example, parameter marshalling required type information to dynamically interpret arguments on the stack. To support parameter marshalling, Naroff modified GCC to emit method type *signatures*. The static linker reduced space overhead by removing duplicate signatures across modules.

After forwarding and signatures were in place, Bertrand Serlet and Blaine Garst drafted a proposal for adding explicit *protocol* support to Objective-C. The initial motivation was improving the runtime efficiency of remote method invocation. Protocols enabled client-side method signatures, eliminating the need to fetch method signatures “over the wire”.

A protocol is an abstract type represented by a list of method declarations. No instance variables are specified. Here is an example “Dragging” protocol:

```
@protocol Dragging

- (id)draggingSource;
- (Point)draggingLocation;
- (Window *)draggingDestinationWindow;

@end
```

Blaine Garst remembers protocols being inspired by modules in Modula [Garst 2016, 22–25, 54]. Naroff, however, believes that the notion of declaring an abstract interface for others to implement was inspired by C++ “pure virtual” functions. Instead of adopting the C++ idiom verbatim, however, the team distinguished concrete types (i.e., classes) from abstract types (i.e., protocols). In turn, protocols in Objective-C inspired interfaces in the Java language, according to James Gosling [Gosling 2019, 13–14] and Patrick Naughton [NA Naughton 1997]. Unlike Objective-C, Java interfaces and classes are in the same namespace. Naroff likes Java’s simplification for local programming tasks, but doesn’t think it fits well with forwarding/proxies.

Class interfaces were extended for protocol *adoption*. The angle brackets contain a list of adopted protocols (e.g., Dragging, Spelling, and Encoding). The following syntax requires the Text class implement methods for the adopted protocols.

```
@interface Text : View < Dragging, Spelling, Encoding >
...
@end
```

Object declarations were extended for protocol *conformance*. Protocols resulted in better type checking without compromising the flexibility of “id”. Declarations are permitted to specify a class as well as multiple protocols. Here are a couple of examples:

```
// any object conforming to the Dragging protocol
id <Dragging> local_or_remote;

// a View object conforming to the Dragging and Spelling protocols
View <Dragging, Spelling> *local;
```

The reflection API was extended to check for conformance.

```
if ([object conformsToProtocol:@protocol(Dragging)])
...

```

Protocols provided a way to augment Objective-C’s single inheritance model with *multiple interface inheritance*, but not inheritance of implementation. This design point was far simpler than adding C++ style multiple inheritance, which included implementation. NeXT had a strong desire to keep the core object model simple. The focus was on sharing interfaces and expressing intentions, not traditional code reuse. Blaine Garst immediately began to use protocols to implement Portable Distributed Objects, and eventually also for archiving in the Foundation framework [Garst 2016, 54–55]. Along with categories, protocols became a breakthrough feature addition that would be used throughout the AppKit and other frameworks to describe all kinds of reusable behavior, including implementation of the delegation design pattern. In 1995, Serlet and Garst added the ability to add *categories on protocols*. However, soon after, NeXT signed a deal with Sun Microsystems in which Sun bought a source code license to NeXTSTEP, freezing the previous codebase, causing NeXT to abandon this new feature [Garst 2016, 60–61]. Though never shipped in Objective-C, this capability would be released as part of Swift in 2014, in the form of *protocol extensions* [NA Apple Inc. 2020b], which enable a new idiom of protocol-oriented programming in which the use of subclassing is bypassed completely.

17 STASIS AND EVOLUTION AT APPLE (1997-2012)

By 1993, it was clear that NeXT’s computer sales were not succeeding in the marketplace, and the company executed a hard and painful pivot into becoming a software company. The entire hardware business, including its automated factory, was either laid off or sold to Canon. The NeXTSTEP software was ported to the Intel x86 architecture, HP PA-RISC, and Sun Solaris. In this environment, Naroff decided to seek other opportunities, interviewing with First Person (the Sun spin-off that developed Java), Apple’s Advanced Technology Group in Cambridge (where he would have worked with Ike Nassi’s team developing the dynamic object-oriented language Dylan), and IBM. Naroff moved to North Carolina in 1994 to join IBM’s Architecture Design Counsel, investigating the prospects of various software initiatives IBM had invested in (Taligent, OS/2, OpenDoc). Dissatisfied with his work at IBM, Naroff returned to NeXT in 1995, working remotely to coordinate a Java strategy for the company, creating an Objective-C/Java bridge that would be important for NeXT’s new WebObjects product. After Apple acquired NeXT in late 1996/early 1997, Steve Jobs asked Naroff to move back to Silicon Valley to become the Director of a developer tools team that merged the groups from Apple and NeXT, which was also responsible for deploying Java on both the classic Macintosh (Mac OS 8/9) platform and the new Mac OS X platform. By 2002, tired of management, Naroff returned to technical work to develop Xcode, Apple’s new Integrated Development Environment. Naroff hired Chris Lattner to the compiler team, and Apple would use Lattner’s open source LLVM compiler architecture to replace its use of the GCC compiler. Naroff

led the effort on Clang, a new Objective-C compiler frontend for LLVM, in 2007, replacing the old GCC frontend.

By 2005, Objective-C had been unchanged for over a decade. Just before Stepstone's closing, NeXT had acquired all rights to Objective-C from Stepstone and those rights had transferred to Apple in 1997. However the language was effectively frozen as it had been at NeXT. For a time under Bertrand Serlet's leadership of the OS X division, both C and Java received more attention. Mac OS X needed to bring legacy C-based Macintosh applications to the new system, and thus needed C-based libraries such as Carbon and CoreFoundation. Java was the buzz-worthy language of the dot.com boom, and Apple needed to support it. Moreover, there were enough similarities between Java and Objective-C that the object-models could be mapped relatively easily. Naroff's Java bridge allowed the old NeXT-based frameworks (AppKit and Foundation, now rebranded "Cocoa") to easily express their APIs in Java, allowing developers to write Cocoa applications using Java. This "Cocoa Java" was seen as a way to persuade third party developers unfamiliar with Objective-C to write Mac OS X applications. "Cocoa Java" never got much traction among developers, however. Moreover, Sun's control over Java prevented Apple from contributing features to the language that better supported Cocoa. Sun's view of Java was not just as a language, but a platform-independent platform of its own, which conflicted with Apple's needs as Apple was itself in the platform business.

The only significant work on Objective-C during the period when NeXTSTEP was developed into Mac OS X was ultimately abortive. In 1997, Naroff's work on the Java bridge had inspired a brief flirtation with changing Objective-C's syntax to use Java and C++ style dot notation for invoking methods. Naroff had written a tool that could have effectively changed all of Apple's Objective-C APIs, including those in AppKit and Foundation, to use the new "modern syntax." Apple Executive Vice President of Software Avie Tevanian was extremely supportive of the effort, but it met significant resistance from the AppKit team led by Ali Ozer. Naroff ultimately rescinded his support for "modern syntax" on the basis that it would make Objective-C++ code much less readable. Under modern syntax, Objective-C message passing would be difficult to distinguish from C++ member function calls in mixed code. Due to the failure of the modern syntax proposal, and with engineering efforts focused on other languages to support shipping the initial versions of Mac OS X, no new work on Objective-C was undertaken until 2005, except for the introduction of a new exception-handling syntax to better support Cocoa Java.

By this time, Bertrand Serlet had concluded that the company needed to recommit and reinvest in Objective-C, while finding a way to incorporate modern features that developers were requesting. At one point, a proposal was made to create a whole new language (internally dubbed "C*"), but these plans were scaled back to adding evolutionary improvements to Objective-C, an effort which became "Objective-C 2.0." Released in 2007 with Mac OS X 10.5 Leopard, Objective-C 2.0 introduced a conservative, generational garbage collector, a new 64-bit runtime that allowed adding new instance variables to superclasses, class extensions (nameless categories, which provided a private way to add methods without advertising them in the public header file), and properties, which automatically generated accessor methods (setters and getters) for instance variables. These and later additions were collaborations between Blaine Garst and the compiler team led by Chris Lattner, to whom Naroff now reported.

The new 64-bit runtime created an opportunity for breaking binary compatibility as new applications would have to be recompiled for 64-bit support anyway, and this allowed the 64-bit runtime to solve a remaining longstanding fragility issue, the "fragile instance variable problem"—classes with subclasses couldn't add new instance variables, which would change the size of classes in a compiled binary executable. (This was not the same issue as the fragile selector problem that

had been fixed by Naroff while still at Stepstone in the late 1980s.) The 32-bit runtime remained unchanged and backward compatible.

The properties feature engendered some controversy among Apple developers, as the synthesized getters and setters used Java-style dot notation to access the underlying fields, masking the fact that a method invocation was taking place. Nevertheless, properties ultimately proved to be a popular feature addition, greatly cutting down on boiler-plate code. Less successful was the addition of garbage collection (GC), the most ambitious and technically difficult feature of Objective-C 2.0. The requirements of compatibility with reference-counted memory management and manually managed C code created constraints for the new garbage collector. Applications, and the libraries they loaded in, had to run either completely in garbage collected mode or not. GC did not get much buy-in from groups outside of Apple’s developer tools. Other than Apple’s Xcode IDE, no other internal applications were ever converted to run with GC. Most glaringly, the new iPhone did not use garbage collection as its developers considered the non-deterministic pauses of a collection-in-progress to be unacceptable in a mobile environment.

After Objective-C 2.0, the next major feature to be added was blocks, known as closures or lambdas in other languages. These are unnamed higher order functions that can be stored in variables and passed in arguments and return values. This was actually an addition in Clang’s implementation of C itself, not only Objective-C, as it was deemed important to support the new concurrency feature in Mac OS X 10.6 Snow Leopard named Grand Central Dispatch, which provided C APIs.

The refusal of the iPhone to use garbage collection had effectively doomed the feature. With the advent of the App Store in 2008, waves of new iOS developers were learning Objective-C for the first time, but without garbage collection, were having to do manual reference counting to manage memory. 2011’s Automatic Reference Counting (ARC) replaced GC and removed most of the burden. ARC relied on the fact that most idiomatic Objective-C code followed predictable conventions regarding retain and release. The reaction to ARC was overwhelmingly positive, and GC was soon deprecated.

2012 brought a new literal syntax for working with boxed arrays, dictionaries, and numbers [NA Clang Team 2020; Dalrymple 2012]. Because of the original separation between corporate ownership of Objective-C and the NeXTSTEP frameworks that became OS X’s Cocoa frameworks, the fundamental data types in NeXT’s Foundation framework, NSArray, NSDictionary, and NSNumber, were not part of the Objective-C language. Stepstone had intended for its ICPak101 to be Objective-C’s standard library, but NeXT had replaced it with Foundation. Use of arrays, dictionaries and boxed numbers had required verbose message passing syntax, such as

```
foo = [myArray objectAtIndex:2];
```

With the new literals syntax, this became

```
foo = myArray[2];
```

making it effectively the same as the syntax for C arrays.

From 2005 through 2012, then, significant evolutionary improvements to Objective-C, large and small, were proposed, debated, and implemented by Blaine Garst, Bertrand Serlet, Steve Naroff, Chris Lattner, and others. Naroff retired from Apple in 2010. Garst and Serlet both left in 2011, after Steve Jobs’ death. Future language development at Apple would be directed by Chris Lattner, who in 2014 unveiled Swift, a new language initially billed as “Objective-C without the C.” Reactions to Swift, especially among iOS developers, have been extremely positive. Swift introduces new features such as generics and better support for functional programming, in addition to the previously mentioned protocol extensions. Due to these modern features, third party developers have shifted in large numbers from Objective-C to Swift. Yet Swift, despite its interoperability features, is not

Objective-C. Swift is statically typed, making dynamic design patterns such as key-value coding and observing, widely used in Apple's Cocoa frameworks, unnatural in Swift. Idiomatic Swift and idiomatic Objective-C do not mesh easily. Rather, new native Swift design patterns and idioms are gradually replacing the old dynamic Objective-C ones. Moreover, Objective-C still allows existing C and C++ code to be easily mixed in with Objective-C code in the same file, while Swift does not. Legacy frameworks at Apple are still implemented in Objective-C, and it does not appear that Swift will replace it inside Apple any time soon. Nevertheless, with the release of new native Swift frameworks such as SwiftUI in 2019, the transition may have begun.

18 CONCLUSION

The roots of Objective-C originated at the international telecommunications firm ITT in the early 1980s in the context of a software engineering community concerned with improving programmer productivity by an order of magnitude to address the perceived “software crisis” that had first been articulated in the late 1960s. Brad Cox and Tom Love, seeing great promise in the Smalltalk-80 message/object model of object-oriented programming, but also a practical need for compatibility with Unix and existing C programs, and better performance on workstations, decided to graft a Smalltalk-style layer on top of C, first with OOPC at ITT and later Objective-C. Cox and Love founded PPI (later Stepstone) in 1983 to commercialize Objective-C and libraries of software components, or “Software-ICs” written in Objective-C, to help usher in what they hoped would be a transformative new way to produce and market software, in which developers would acquire ready-made components to construct their programs rather than write everything anew from scratch. Cox believed this would usher in a “Software Industrial Revolution.” Cox and Love were very active participants in the growing advocacy of object-oriented programming in the 1980s and early 1990s, helping to start the ACM OOPSLA conference.

Stepstone sold licenses to Objective-C to many customers, most notably Hewlett-Packard and NeXT, and was successful for a time. However, due to ineffective management, Stepstone went out of business by 1994, selling Objective-C to NeXT. NeXT, with its focus on making software development easier and more accessible, akin to the way the Macintosh had made computer use easier, shared in Cox and Love's concern with improving developer productivity through object-oriented technology, and marketed its software on these benefits. Ultimately, it would be NeXT, the company that bet its entire technology stack on Objective-C, that would carry the language forward. Under the stewardship of Steve Naroff, engineers at NeXT added many improvements to the language, most notably categories and protocols. When NeXT was acquired by Apple in 1997, Objective-C became the basis for Apple's most important new platforms, Mac OS X, and later, iOS, powering Apple's resurgence and the mobile app revolution. The iPhone App Store would bring Objective-C into the mainstream of software development for the first time in its history. Despite being eclipsed by a new Apple language, Swift, that may eventually replace it, Objective-C today continues to serve as a critical foundation for Apple's platforms.

ACKNOWLEDGMENTS

Thanks to Tom Love for providing additional information for the paper and much of the timeline.

A OBJECTIVE-C TIMELINE

1976 April	Apple founded as partnership by Steve Jobs and Steve Wozniak
1977	ITT begins work on System 1240 digital telephone exchange, designed at ITT's Advanced Technology Center in Stamford, CT and later Shelton, CT.
1979	Bell Labs' Bjarne Stroustrup begins work on C with Classes

1979	Rand Araskog replaces Lyman Hamilton as ITT CEO, Harold Geneen remains as Chairman of the Board
1979 Nov	Tom Love hired as Director of Advanced Programming Technology at ITT from GE Information Systems
1980 April	C with Classes published in Bell Labs CS Technical Report (republished in Jan 1982 SIGPLAN Notices)
1980	ITT creates Programming Technology Center in Stratford, CT. Tom Love hires Anatol Holt, Rudy Ramsey, Brad Cox, Ted Biggerstaff, Alan Watt, Maude Sawyer to staff the Advanced Technology Department within the PTC to develop tools to support telecom developers at ITT
1981 Aug	Brad Cox receives <i>Byte</i> magazine issue devoted to Smalltalk Cox asks for permission to take an Onyx computer home for a week to develop an “object-oriented pre-compiler” (OOPC) for C language
1981	Cox writes paper describing OOPC, published in Jan 1983
1982	OOPC released at ITT
1982	Stroustrup begins work on C++, successor to C with Classes
1982 June	Love hired by Schlumberger-Doll, with intent of moving his Advanced Technology Department to Schlumberger
1982 c. Sept	Love gets first Smalltalk development environment from Xerox without user manual or training materials, with only a Xerox help line staffed by Evelyn van Orden in Los Angeles
1982	Cox spends most of his time at ITT improving and extending OOPC, for the purpose of improving programming tools for telecom developers
1983 Jan	OOPC paper published in ACM SIGPLAN Notices
1983 Jan	Apple releases the Lisa
1983 c. Jan	Brad Cox hired by Tom Love at Schlumberger to build expert systems for petroleum engineers
1983 c. Mar	Tom Love introduced to technical director at Philips Research Lab in Eindhoven
1983 June 6	Tom Love and Brad Cox resign from Schlumberger to work on the Philips contract and to develop a commercial version of the object-oriented extension to C, founding new company Productivity Products International (PPI)
1983 July	Cox and Love present their work on OOPC and Smalltalk at Softfair: A Conference on Software Development Tools, Techniques, and Alternatives in Arlington, VA
1983 Nov	First copy of Objective-C sold and shipped to Tom Lubinski in California
1984 Jan	Love, Cox, and two initial PPI employees occupy small former dentist office in Newtown, CT
1984 Jan	Stroustrup publishes C++ Technical report in Bell Labs CS Technical Journal (written summer 1983)
1984 Jan	Apple releases the Macintosh
1984	Love organizes angel financing with friends and family to provide necessary working capital to grow PPI
1985	Love contracts with K.C. Branscomb to help PPI get venture capital investors. This financing was led by Oak Management in Westport, CT with caveat that they bring in a full time/experienced CEO. Headquarters moved to small renovated mill near center of Sandy Hook, CT, on Pootatuck River.

- 1985 Love and Cox develop training course in object-oriented programming with the goal of convincing participants to buy copies of the Objective-C compiler and supporting class libraries (called ICpaks). More than 90% of class participants purchased both the compiler and ICpaks.
- 1985 C++ 1.0 becomes commercially available
- 1985 Love unsuccessfully attempts to hire Bjarne Stroustrup from Bell Labs to PPI
- 1985 June Steve Jobs leaves Apple, founds NeXT with five other former Apple employees
- c. 1986 Oak Management partner Dennis Sisco hired as CEO.
- 1986 Nov Steve Naroff joins PPI
- 1987 July Steve Naroff and Alan Watt draft white paper describing needed improvements in Objective-C
- 1987 Nov NeXT engineers Steve Stone and Trey Matteson visit PPI to discuss changes in Objective-C needed for NeXT to deploy shared libraries
- c. 1987-1988 Company name changed from PPI to Stepstone.
- 1988 Stepstone negotiating deal with IBM, HP, NeXT to include ObjC on all their workstations
- 1988 June Tom Love resigns from Stepstone after heated discussion of pricing strategy at board meeting entirely staffed with VC investors.
- 1988 June Tom Love starts Orgware, one-person consulting company based in Roxbury, CT
- 1988 July/Aug Steve Naroff leaves Stepstone, joins NeXT
- 1988 July Love/Orgware signs contract with Ascent Logic to spend a week a month in offices of a Silicon Valley startup run by a former colleague from ITT
- 1988 Oct NeXT Computer released with NeXTSTEP operating system 0.8. NeXTSTEP uses Objective-C for its AppKit application framework and InterfaceBuilder graphical development tool.
- 1989 Naroff adds native Objective-C support to GNU C Compiler (GCC), removing the need for NeXT to use Stepstone's Obj-C to C translator.
- 1989 Categories added to Objective-C at NeXT
- 1989 Sept NeXTSTEP 1.0 released.
- 1989-1990 C++ compatibility (Objective-C++) added by Naroff to Obj-C compiler
- 1990 Brad Cox leaves Stepstone
- 1990 Protocols added to Objective-C at NeXT
- 1990 Sept NeXTSTEP 2.0 released
- 1993 June Love hired as VP Object Technology Practice within IBM Consulting, working for Bob Howe
- 1992 Sept NeXTSTEP 3.0 released
- 1993 May NeXTSTEP 3.1 for Intel i386 architecture released
- 1994 July Steve Naroff briefly leaves NeXT for IBM
- 1994 Aug Love acquired by Morgan Stanley with an offer several times larger than his package at IBM
- 1994 Fall Steve Jobs arrives at Love's office at Morgan Stanley attempting to sell NeXT workstations. Love tells Jobs that he will only do business with NeXT if Jobs resolves dispute with Stepstone over Objective-C license contract.
Love funds a trading system developer in San Francisco to compare NeXT machines and the development tools already at use at Morgan Stanley. Results are not compelling enough for Love to recommend purchase of NeXT machines.

	Jobs contacts third Stepstone CEO KK Tan, writes Stepstone check for overdue royalties owed, and acquires full rights to Objective-C from Stepstone for a fixed amount.
c. 1994	Stepstone folds
1995 Feb	NeXTSTEP 3.3 released, includes ports for HP PA-RISC and Sun SPARC architectures
1995 June	Steve Naroff rejoins NeXT
1996 Dec	NeXT begins acquisition talks with Apple
1997 Feb	Apple finalizes acquisition of NeXT, Jobs returns as advisor. NeXT employees, including Naroff, become Apple employees
1997	Steve Naroff becomes Senior Director, Java Technologies and Core Tools at Apple, builds relationships with Metrowerks, Genentech and Oracle
1997	Naroff proposes Java-like “modern syntax” for Objective-C. The change is ultimately rejected.
1997 July	Gil Amelio resigns as Apple CEO, Steve Jobs becomes interim CEO
1999 March	Mac OS X Server 1.0, based on NeXTSTEP, is released
2001 March	First consumer version of Mac OS X, 10.0 “Cheetah,” with new Aqua interface, released. OS X comes with two parallel application frameworks, “Carbon,” based on the classic Macintosh Toolbox, and “Cocoa,” based on NeXTSTEP and Objective-C. AppKit, carried over from NeXTSTEP, is the central component of Cocoa.
2002-2005	Naroff becomes Chief Technologist, Apple Tools Group, working on first version of Apple’s Xcode IDE. Naroff hires Chris Lattner; Apple begins to transition from GCC-backend to LLVM.
2005	Naroff and others begin work on “C*”, which will ultimately lead to Objective-C 2.0. Objective-C 2.0 will include the first new features added to the language since the 1990s. Features include garbage collection, properties, class extensions, fast enumeration, and a 64-bit runtime with non-fragile instance variables.
2007 June	Apple ships first iPhone
2007 Oct	Objective-C 2.0 ships with Mac OS X 10.5 Leopard.
2007 Oct	Steve Jobs reverses course, announcing iPhone will have a native SDK for 3rd party developers
2007-2010	Naroff leads effort on developing Clang, a new Objective-C/C/C++ front-end for LLVM, to replace GCC frontend
2008 March	iPhone SDK, branded “Cocoa Touch,” released. “Cocoa Touch,” whose central component is UIKit, is designed along similar principles as AppKit on OS X, and requires the use of Objective-C. Developers begin working on apps using Cocoa Touch, spurring Objective-C adoption.
2008 July	iPhone App Store opens with 500 apps, all developed with Objective-C.
2009	Mac OS X 10.6 Snow Leopard ships with support for blocks (closures) in Clang compiler for Objective-C and C
2010	Steve Naroff retires from Apple.
2011	Automatic Reference Counting introduced in Mac OS X 10.7 Lion
2012	Literal syntax for boxed numbers and literal and subscripting syntax for arrays and dictionaries added to Apple LLVM compiler 4.0.
2014	Apple announces Swift, designed by Chris Lattner, a new language compatible with, but ultimately intended to supersede, Objective-C.

B HISTORICAL DOCUMENT: DESIGN ISSUES FOR OBJECTIVE-C DRAFT OF JULY 3, 1987, S. NAROFF, A. WATT, PPI

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-person 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 reevaluate 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 accommodate some new ANSI constructs could be a way to reduce the amount of work necessary to accommodate 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 to 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. Disadvantages

There are a number of constraints 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 performance 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 classes by defining new procedures which those classes 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.

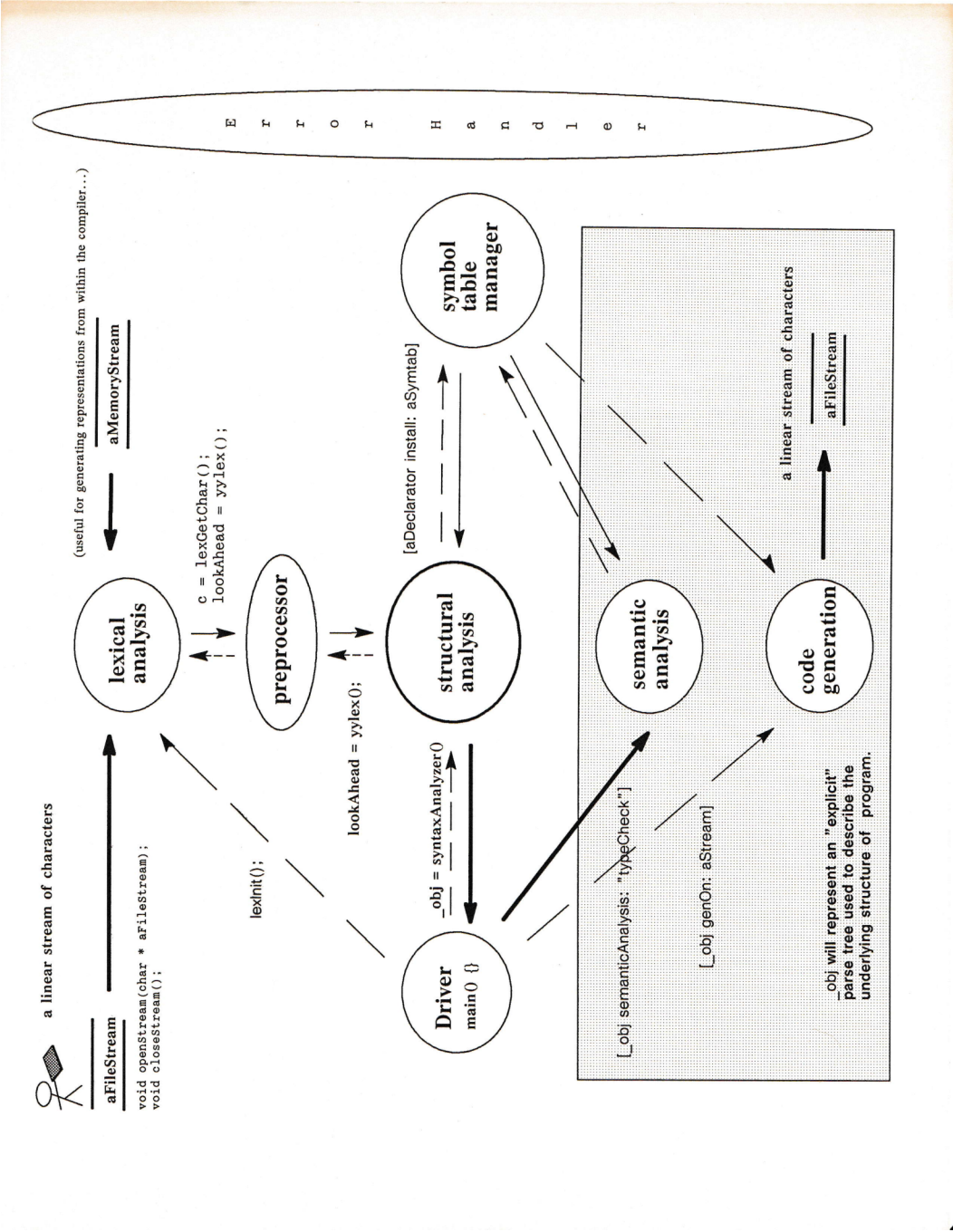
Another very handy use of blocks is as part of exception handling. Several Smalltalk-80 methods take as an argument a block to execute if the requested operation fails

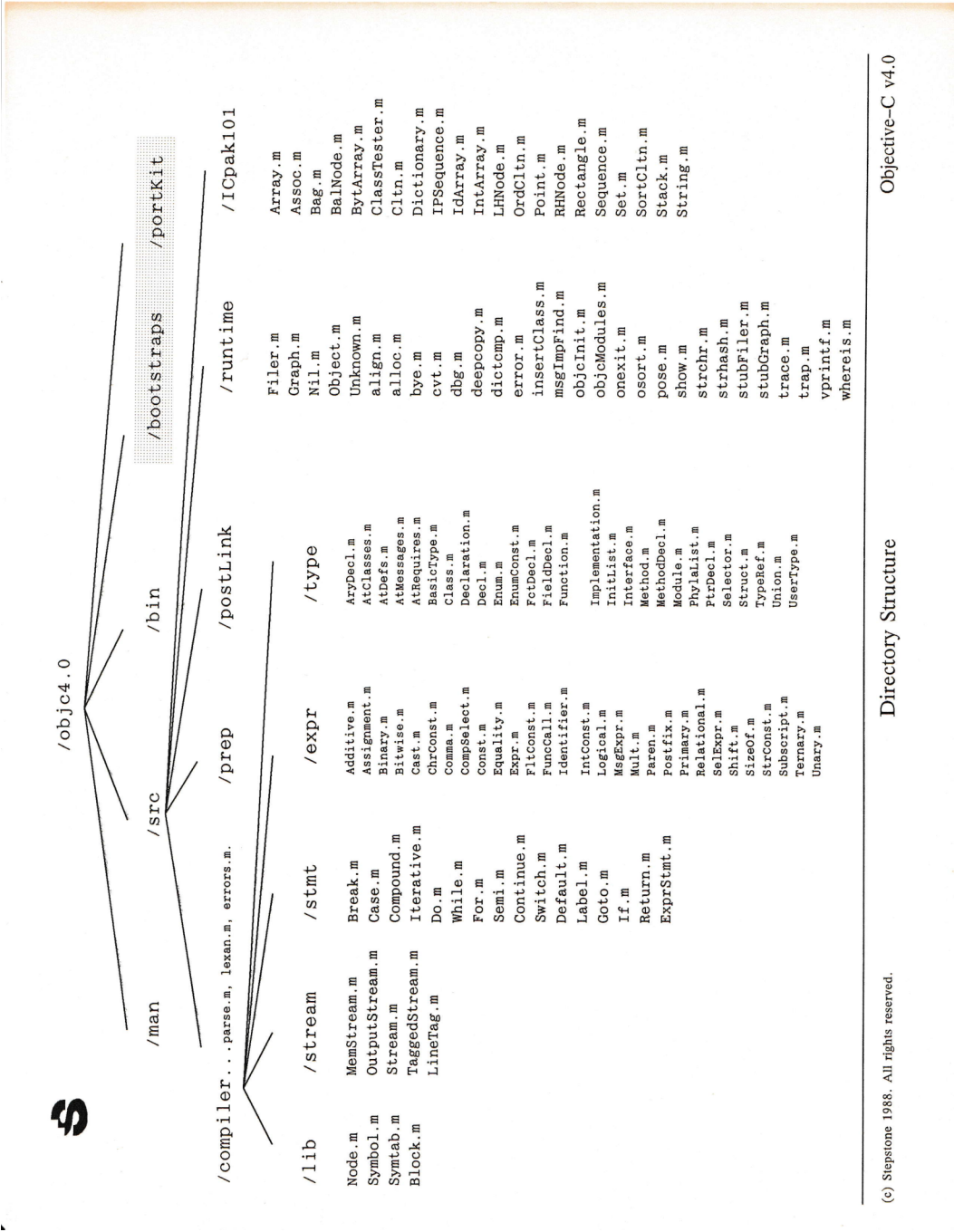
There are significant implementation difficulties in providing blocks as described here.

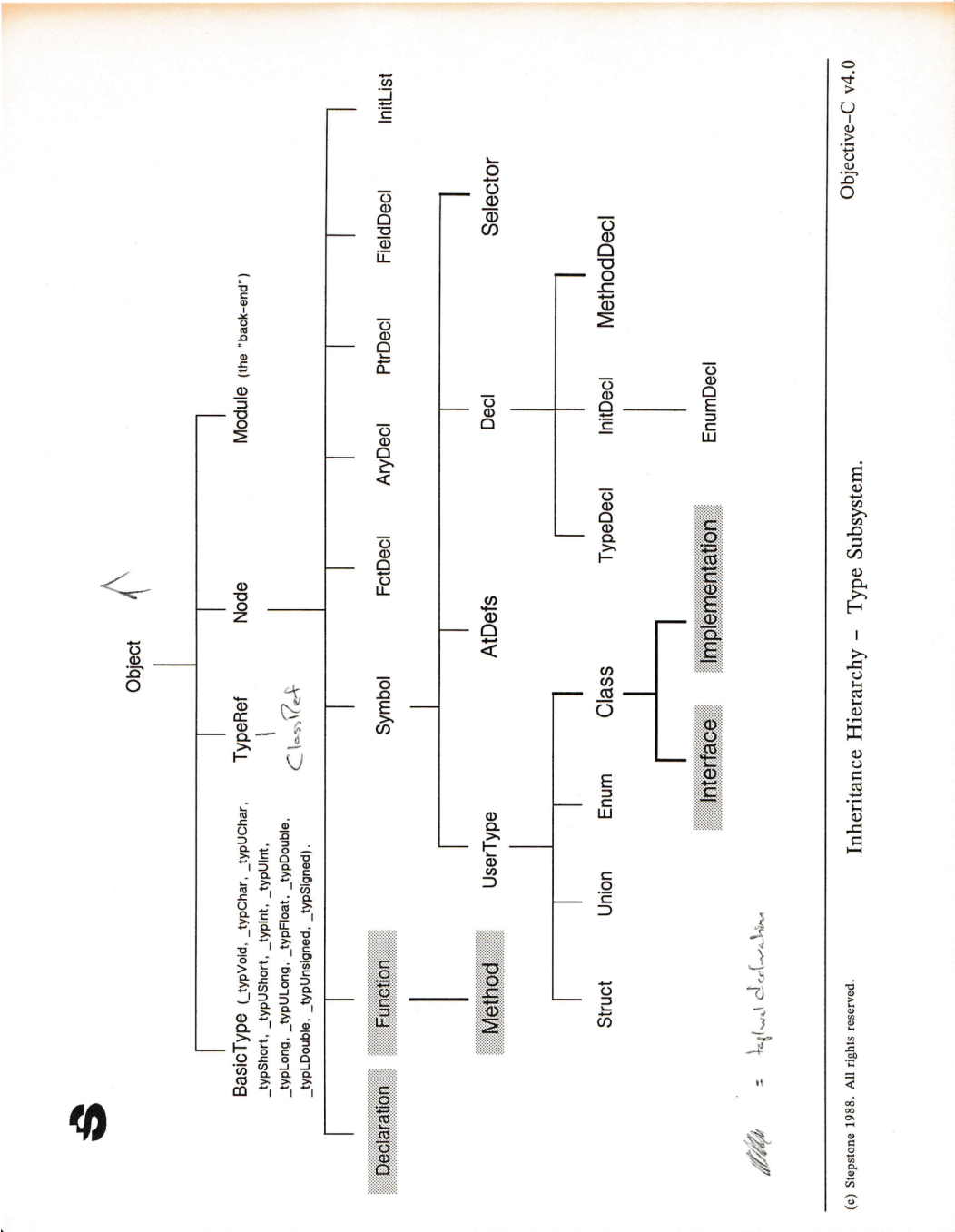
5.7. Dynamic Linking

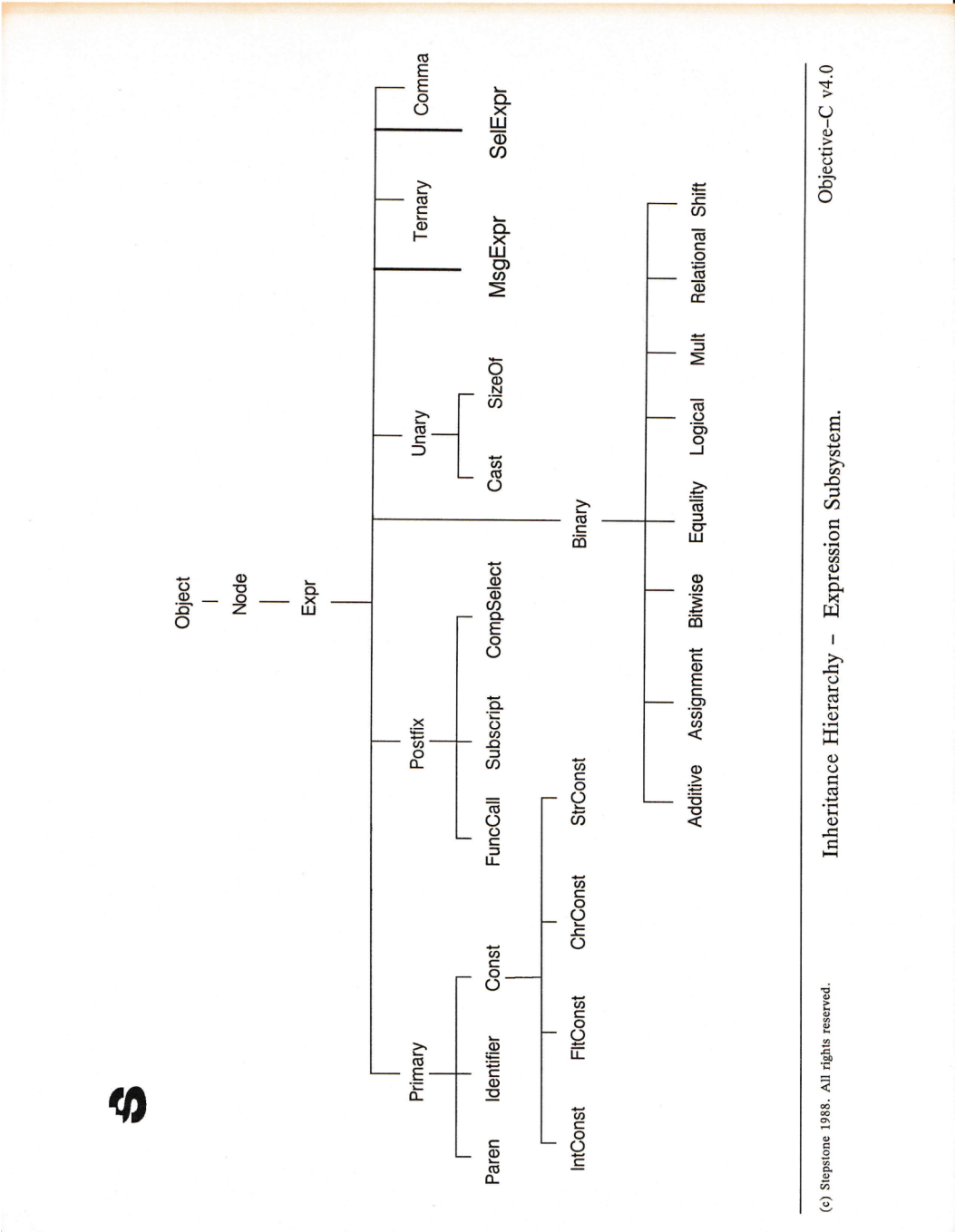
Dynamic linking is the ability to load relocatable object files into a running program and resolve symbolic references. Dynamic linking allows more flexibility in building applications, as well as keeping program images on disk smaller by excluding seldom executed code. This tool would be part of the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

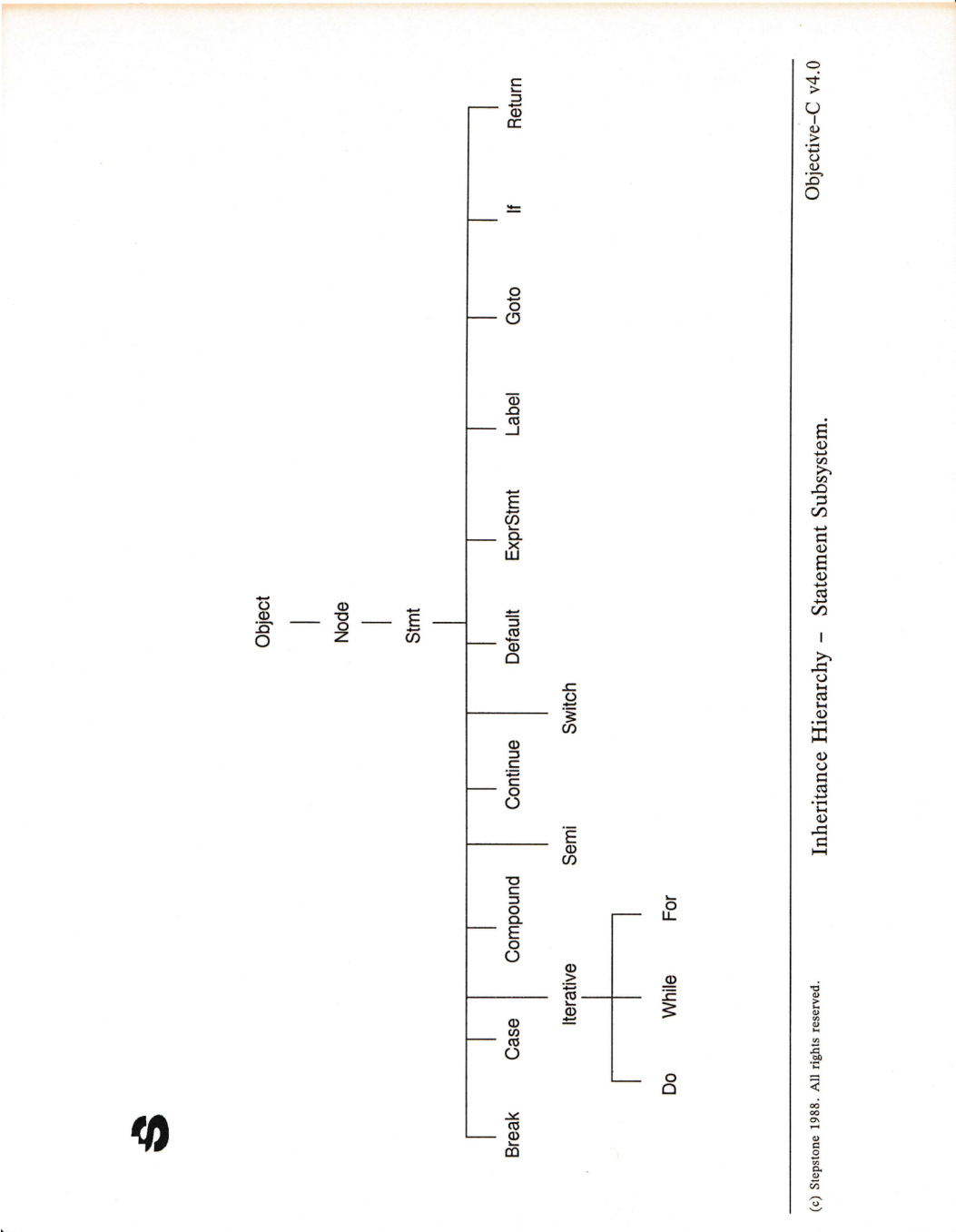
C HISTORICAL DOCUMENT: OBJECTIVE-C V.4.0 CHARTS, STEPSTONE, 1988

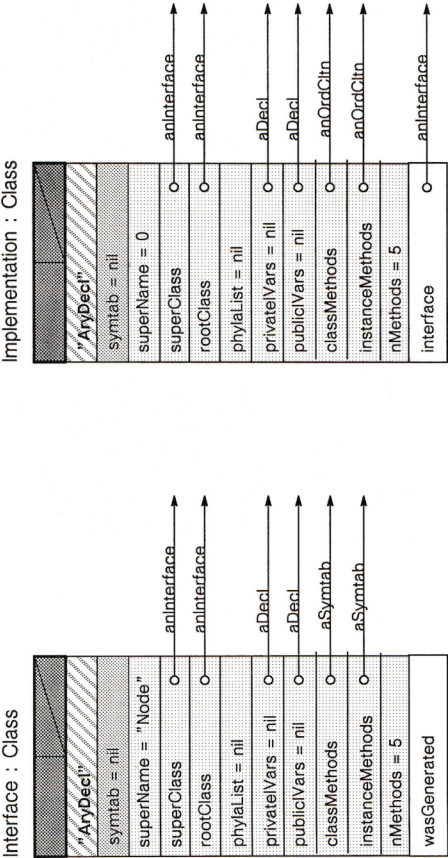








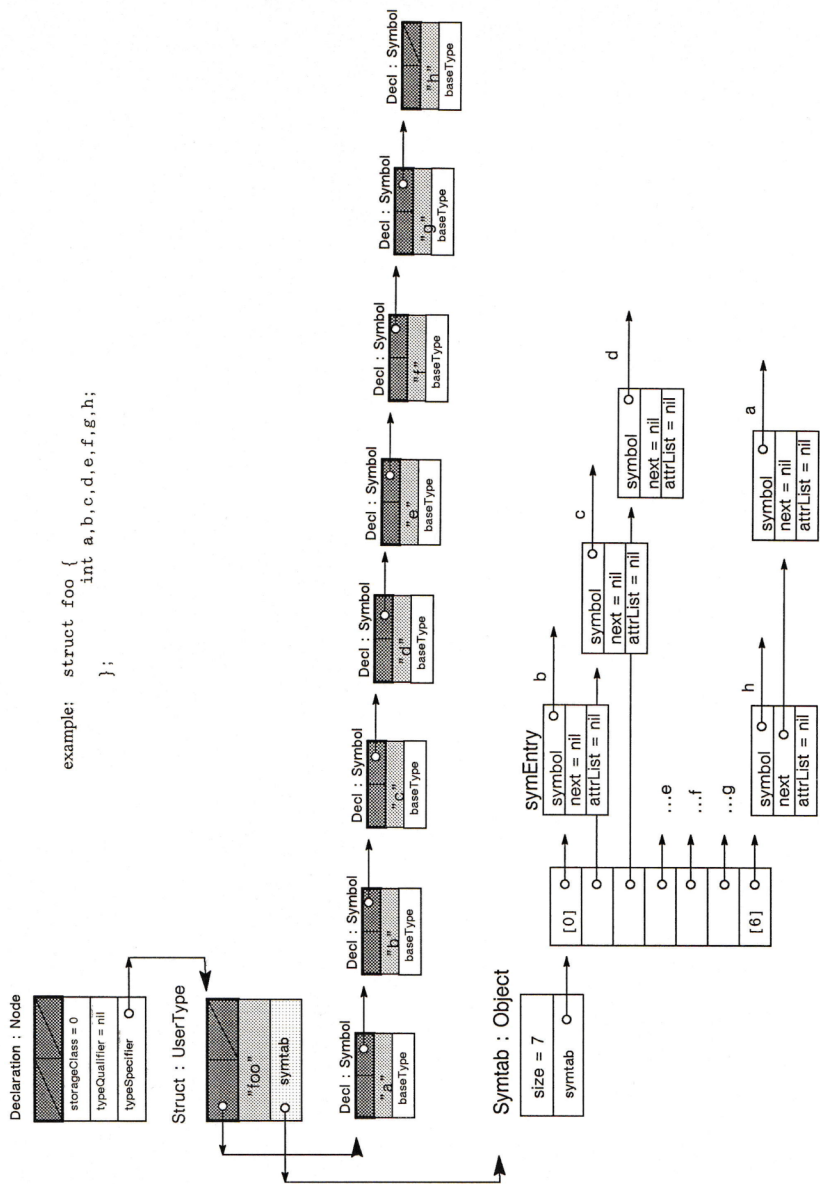




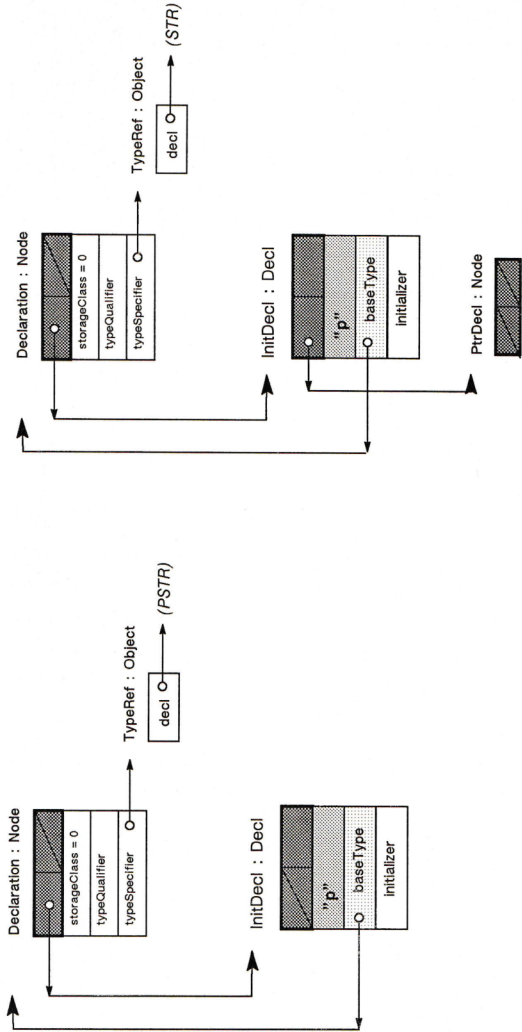
```
@interface AryDecl : Node
- (BOOL) isSameType:anId;
- encode:aBuf typeSpecifier:tSpec;
- (int) sizeOf:aTypeSpecifier;
- assemble:aComplexDeclarator;
- explainOn:astrm;
@end

@implementation AryDecl
- (BOOL) isSameType:anId { ... };
- encode:aBuf typeSpecifier:tSpec { ... };
- (int) sizeOf:aTypeSpecifier { ... };
- assemble:aComplexDeclarator { ... };
- explainOn:astrm { ... };
@end
```

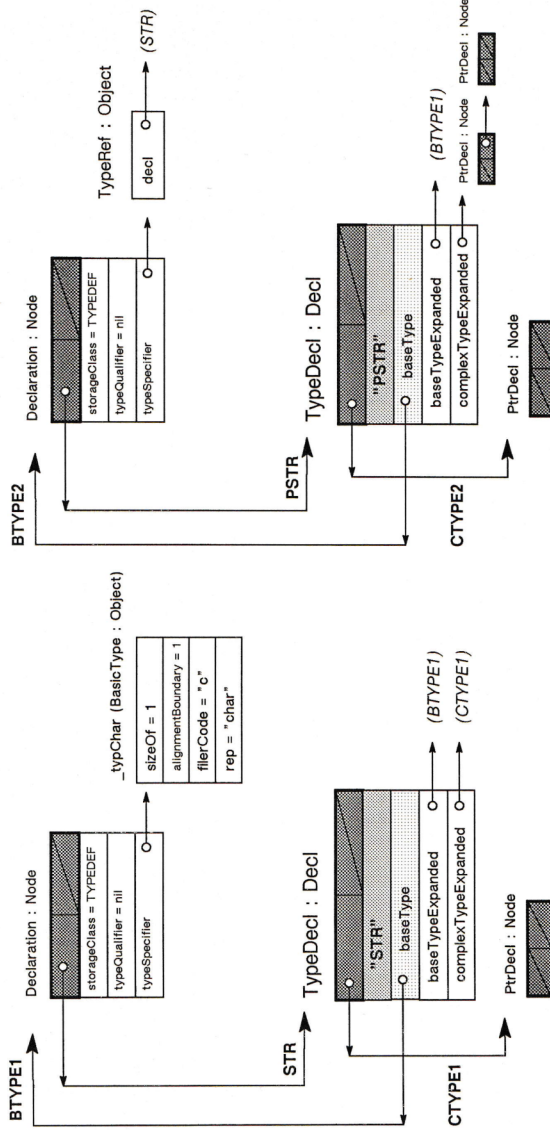


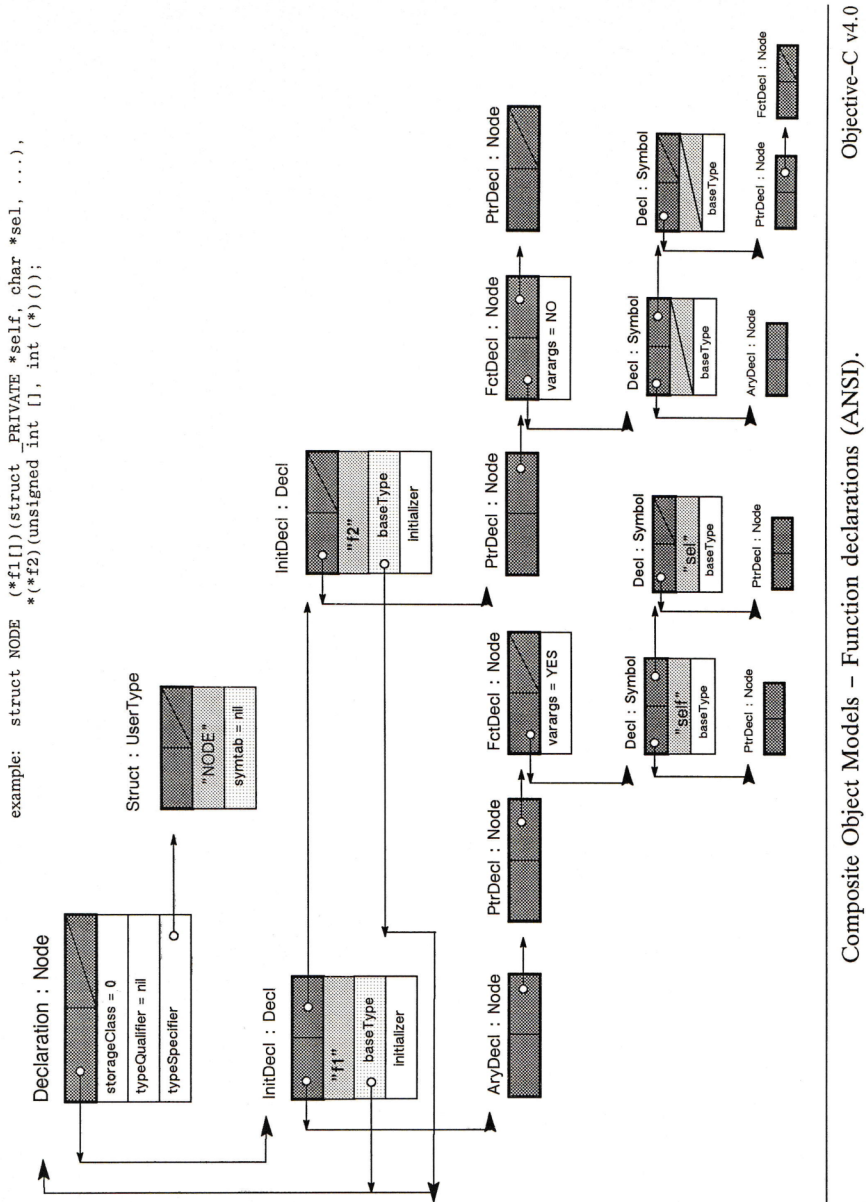


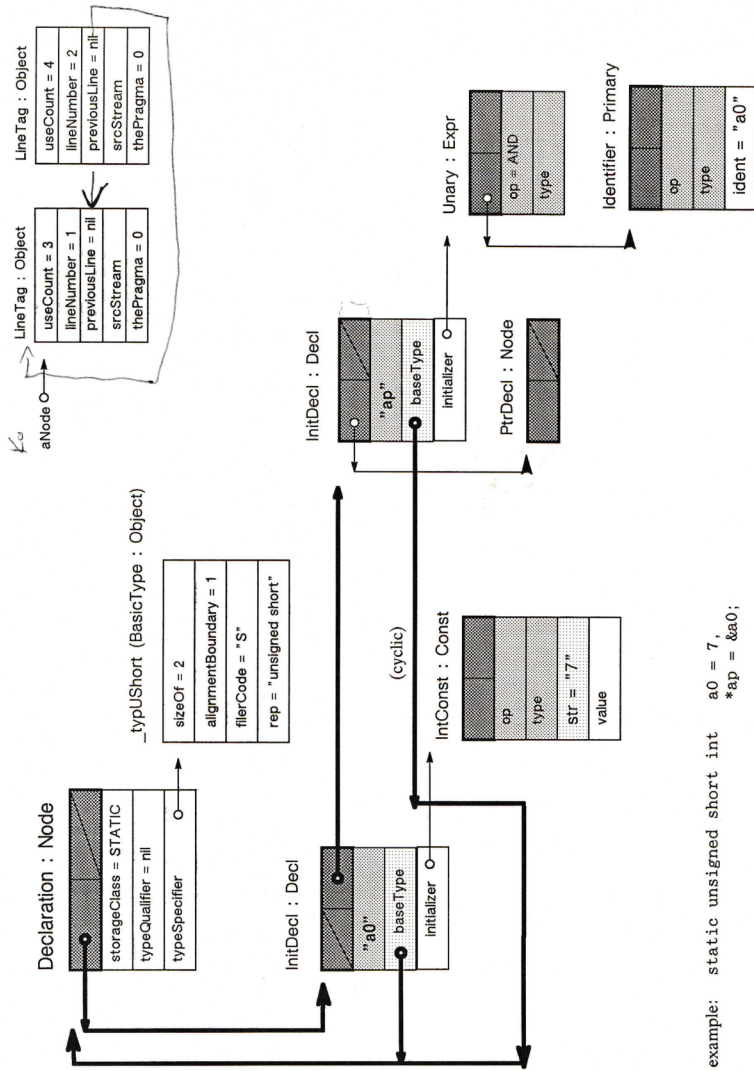
example: PSTR p;
 STR *p;



example: `typedef char *STR; // simple type definition...`
`typedef STR *PSTR; // nested type definition...`







```
example: static unsigned short int a0 = 7,
        *ap = &a0;
```

Composite Object Models – Data declarations/definitions.

Objective-C v4.0

D HISTORICAL DOCUMENT: SPEC.LANGUAGE, S. NAROFF, C.1987

Functional Specification

Class Declaration Syntax and Run-Time Selector Mapping in the Objective-C Language

ppi confidential

s. naroff

spec.language-1

Requirements...based on perceived market need and technical dependencies.

- fix bugs related to forward references, compilation order, and incomplete class declaration files (or002, or017, or028, or042, or044, or058, or071, or072, or075, etc) .
- insure the integrity of the object module.
- allow classes to be dynamically linked (on systems like Apollo/NeXT/HP).
- eliminate language constructs that impede the development/management of applications involving many programmers, the language must be predictable and easy to use.
- provide a solid foundation for offering future enhancements to the language (e.g. static binding) and environment.
- ease integration of IC-paks developed independent of one another.

ppi confidential

s. naroff

spec.language-2

Specification

This section describes changes (from the users perspective) to the Objective-C language and runtime support system which meet the above requirements. The proposed strategy adds only one new construct to the language, a class declaration. It does, however, modify or replace several existing constructs. Note that the examples shown are for illustration purposes only.

(1a) Class Declaration Syntax

Changes:

- the developer would be required to provide the compiler with an interface declaration for the class. Private methods that are only used locally do not have to be listed.
- the developer would be required to "*#import*" the declaration for all of the classes it consumes. This facility will allow a more robust interface; the file that contains the class interface can also embody all the C constructs (pre-processor macros, user-defined types, typedefs) required to interface to the class.
- this replaces the current C_ and P_ files.

Example:

```
/* Expr.h: Objective-C compiler interface specification */
#import "Node.h"
#import "ConstValue.h"

#define declarePTR      id *_PTR = (id *)IV(self)

@interface Expr : Node
{
    id anExpr;
}
- decl;
- constValue;
@

@interface StropExpr : Expr
{
    id primary;
    id aToken;          /* "->", ".", " */
    id componentName;
}
+ initialize;
- decl;
- constValue;
@
```

ppi confidential

s. naroff

spec.language-3

(1b) Importing class interfaces

Changes:

- `#import` is a pre-processor extension defined by Objective-C. It differs from `#include` in two ways:
 - (1) it will allow arbitrary nesting.
 - (2) it will **not** `#import` a file more than once. This will free the user from having to use the following idiom (which is error prone):


```
/* Fruit.h */
#ifndef FRUIT.H
#define FRUIT.H
...
#endif FRUIT.H
```
- the "`= (Primitive, Demo)`" and "`= ClassA : ClassB(Primitive, Demo)`" constructs would be replaced by `#import`. They currently make selectors from the named message groups available during the translation of the current non-class/class source files.
- the "`@requires`" construct would also be replaced by `#import`. Currently, it helps the compiler manage the `C_` and `P_` file dependencies.

Example:

```
#import "Demo.h"
#import "Primitive.h"

main( argc, argv )
int argc;
char *argv[];
{
    id aFruit, anApple;

    aFruit = [Fruit create];
    [aFruit grow];
    anApple = [Apple create];
    [[anApple color:"red"] flavor:"Macintosh"] diameter:7;
}
```

(1c) Class Definition Syntax

Changes (minor):

- specification of the superClass and/or instance variables is optional. If present, they must match the `@interface` declaration for the class. If they do not match, the compiler will issue a warning, indicating a module interface inconsistency.

ppi confidential

s. naroff

spec.language-4

- "@module" will replace the current start indicator ("=").
- "@" will replace the current finish ("=").

Example

```
#import "Expr.h"

@module StropExpr : Expr
{
    id primary;
    id aToken; /* "->", ".", " */
    id componentName;
}
+ initialize { ... };
- decl { ... };
- constValue { ... };
@
```

(1d) Conversion

To ease the conversion to this approach, the compiler will provide a `-genDecl:"aFile"` option. This option will create an interface declaration from the current class definition files.

Given the command:

```
objc StropExpr.m -genDecl:"StropExpr.h"
```

Input:

```
/* StropExpr.m: Objective-C compiler source */

@requires UserType, ConstValue, Decl;

= StropExpr : Expr(ObjectiveC, Collection, Primitive)
{
    id primary;
    id aToken; /* "->", ".", " */
    id componentName;
}
+ initialize { ... };
- decl { ... };
- constValue { ... };
=;
```

ppi confidential

s. naroff

spec.language-5

Output:

```

/* StropExpr.h: Interface declaration for "StropExpr.m"
 * Produced by Objective-C on 1/17/87
 */
#import "UserType.h"
#import "ConstValue.h"
#import "Decl.h"
#import "Expr.h"
#import "ObjectiveC.h" /* a group of related classes */
#import "Collection.h" /* a group of related classes */
#import "Primitive.h" /* a group of related classes */

@interface StropExpr : Expr
{
    id primary;
    id aToken; /* "->", ".", " */
    id componentName;
}
+ initialize;
- decl;
- constValue;
@

```

(2) Runtime Selector Mapping

The additional support required for mapping selectors at runtime will have little impact on users of Objective-C that do not rely on the exact format of the structures generated by the compiler. Users that do rely on many of the internal details (e.g. id **_Classes) will have to convert to the new support structures that are detailed in the design notes on the technique the compiler will employ to accomplish this mapping.

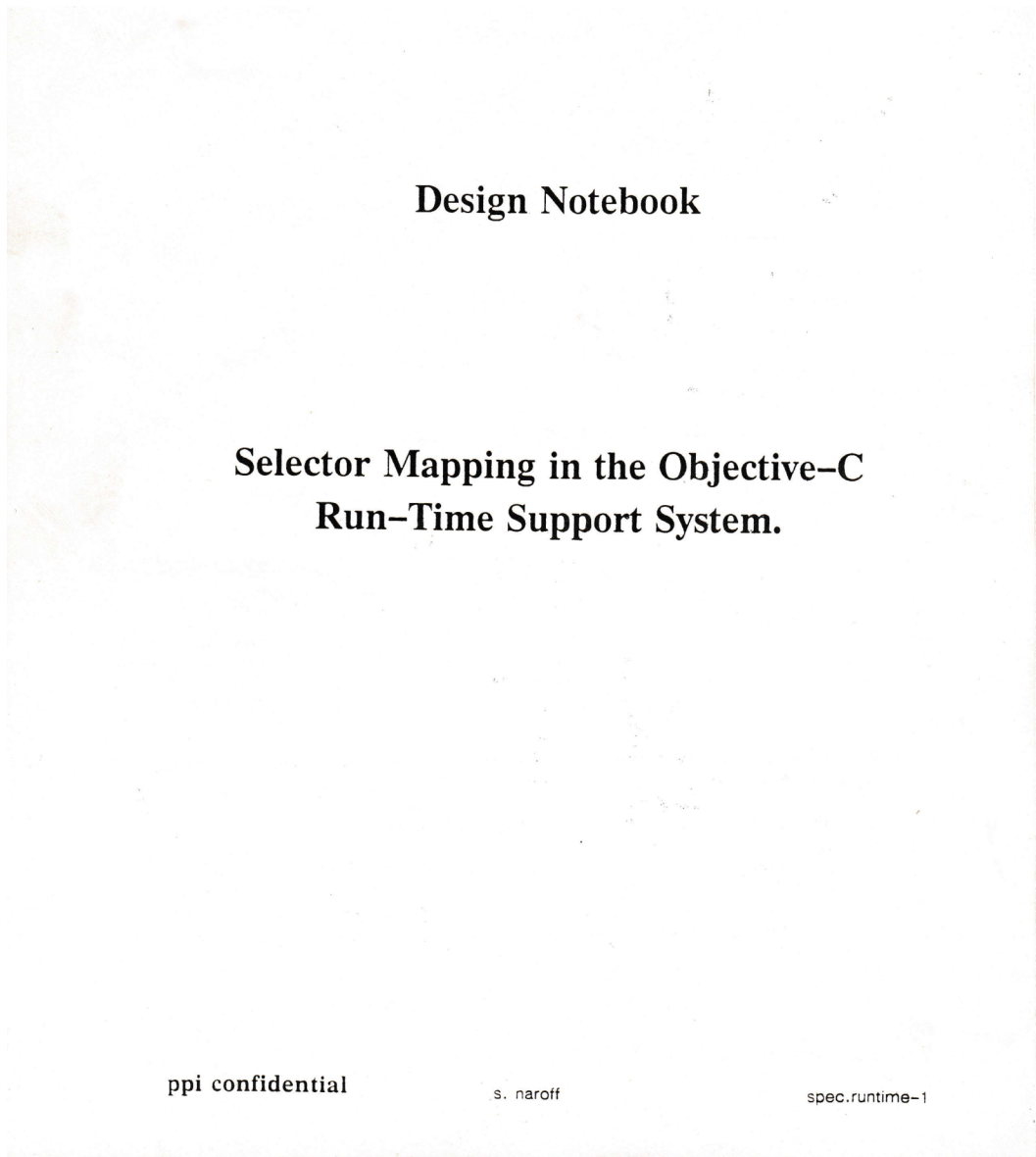
- `@messages(<aPhylaList>)` becomes obsolete. It currently tells the Objective-C compiler to generate a global table of unique selector codes.
- `@classes(<aClassList>)` becomes obsolete. To prevent confusion, the facility for determining what order the classes are initialized must be preserved someplace else.

ppi confidential

s. naroff

spec.language-6

E HISTORICAL DOCUMENT: SPEC.RUNTIME, S. NAROFF, C.1987



Design Notes (evolving)...Implementation details on mapping selectors at runtime.

Selector Tables

Fruit.c

```
static char __msgSelectors[] = {
    'c', 'r', 'e', 'a', 't', 'e', 0,
    'c', 'o', 'l', 'o', 'r', 0,
    'd', 'i', 'a', 'm', 'e', 't', 'e', 'r', 0,
    'g', 'r', 'o', 'w', 0,
    'n', 'e', 'w', 0,
};
```

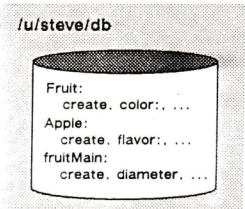
fruitMain.c

```
static char __msgSelectors[] = {
    'c', 'r', 'e', 'a', 't', 'e', 0,
    'd', 'i', 'a', 'm', 'e', 't', 'e', 'r', 0,
    'c', 'o', 'l', 'o', 'r', 0,
    'g', 'r', 'o', 'w', 0,
    'f', 'l', 'a', 'v', 'o', 'r', 0,
    'c', 'o', 'l', 'o', 'r', 0,
    'f', 'l', 'a', 'v', 'o', 'r', 0,
    'd', 'i', 'a', 'm', 'e', 't', 'e', 'r', 0,
};
```

Apple.c

```
static char __msgSelectors[] = {
    'c', 'r', 'e', 'a', 't', 'e', 0,
    'f', 'l', 'a', 'v', 'o', 'r', 0,
    'f', 'l', 'a', 'v', 'o', 'r', 0,
    'f', 'l', 'a', 'v', 'o', 'r', 0,
    'g', 'r', 'o', 'w', 0,
    'n', 'e', 'w', 0,
    'd', 'i', 'a', 'm', 'e', 't', 'e', 'r', 0,
    'c', 'o', 'l', 'o', 'r', 0,
};
```

futures?



- * each compilation unit (source/object file) that defines or references any selectors will contain its own local selector table.
- * the combination of selector tables will be deferred until runtime.
- * store type information (argument and return type) for each selector (not shown). Why?
 - the interpreter requires this information...it currently must read the P_* files.
 - the compiler can offer more sophisticated runtime support.

ppi confidential

s. naroff

spec.runtime-2

Dispatch Tables

Fruit.c

```
static struct _SLT _clsFruit[1]={
    (SEL)&__msgSelectors[0], (id (*)())_1_Fruit, /* create */
};
static struct _SLT _nstFruit[5]={
    (SEL)&__msgSelectors[7], (id (*)())_2_Fruit, /* color: */
    (SEL)&__msgSelectors[14], (id (*)())_3_Fruit, /* color */
    (SEL)&__msgSelectors[20], (id (*)())_4_Fruit, /* diameter: */
    (SEL)&__msgSelectors[30], (id (*)())_5_Fruit, /* diameter */
    (SEL)&__msgSelectors[39], (id (*)())_6_Fruit, /* grow */
};
```

Apple.c

```
static struct _SLT _clsApple[1]={
    (SEL)&__msgSelectors[0], (id (*)())_1_Apple, /* create */
};
static struct _SLT _nstApple[4]={
    (SEL)&__msgSelectors[7], (id (*)())_2_Apple, /* flavor: */
    (SEL)&__msgSelectors[15], (id (*)())_3_Apple, /* flavor */
    (SEL)&__msgSelectors[22], (id (*)())_4_Apple, /* flavor:diameter:color: */
    (SEL)&__msgSelectors[45], (id (*)())_5_Apple, /* grow */
};
```

- each entry will be mapped to a unique code at runtime.
- no need for the compiler to use an extra level of indirection when initializing the dispatch tables. Eliminate `msgImpFind.fixAllDispatchTables(id * msgFirstId)` and `fixDispatchTable(SHR cls)`. The structure template "struct __SLT { char **_cmd; ... }" is therefore obsolete.
- the structure tag used to identify the class and meta-class dispatch tables should include the class name – so that we can move towards lifting the one class per source file restriction.

ppi confidential

s. naroff

spec.runtime-3

Reference Tables

Fruit.c

```
static SEL _selRefs[] = {
    (SEL)&__msgSelectors[44], /* new */
    (SEL)&__msgSelectors[7], /* color: */
    (SEL)&__msgSelectors[20], /* diameter: */
};
```

Apple.c

```
static SEL _selRefs[] = {
    (SEL)&__msgSelectors[50], /* new */
    (SEL)&__msgSelectors[22], /* flavor:diameter:color: */
    (SEL)&__msgSelectors[7], /* flavor: */
    (SEL)&__msgSelectors[54], /* diameter: */
    (SEL)&__msgSelectors[64], /* color: */
};
```

fruitMain.c

```
static SEL _selRefs[] = {
    (SEL)&__msgSelectors[0], /* create */
    (SEL)&__msgSelectors[7], /* diameter */
    (SEL)&__msgSelectors[16], /* color */
    (SEL)&__msgSelectors[22], /* grow */
    (SEL)&__msgSelectors[27], /* flavor */
    (SEL)&__msgSelectors[34], /* color: */
    (SEL)&__msgSelectors[41], /* flavor: */
    (SEL)&__msgSelectors[49], /* diameter: */
};
```

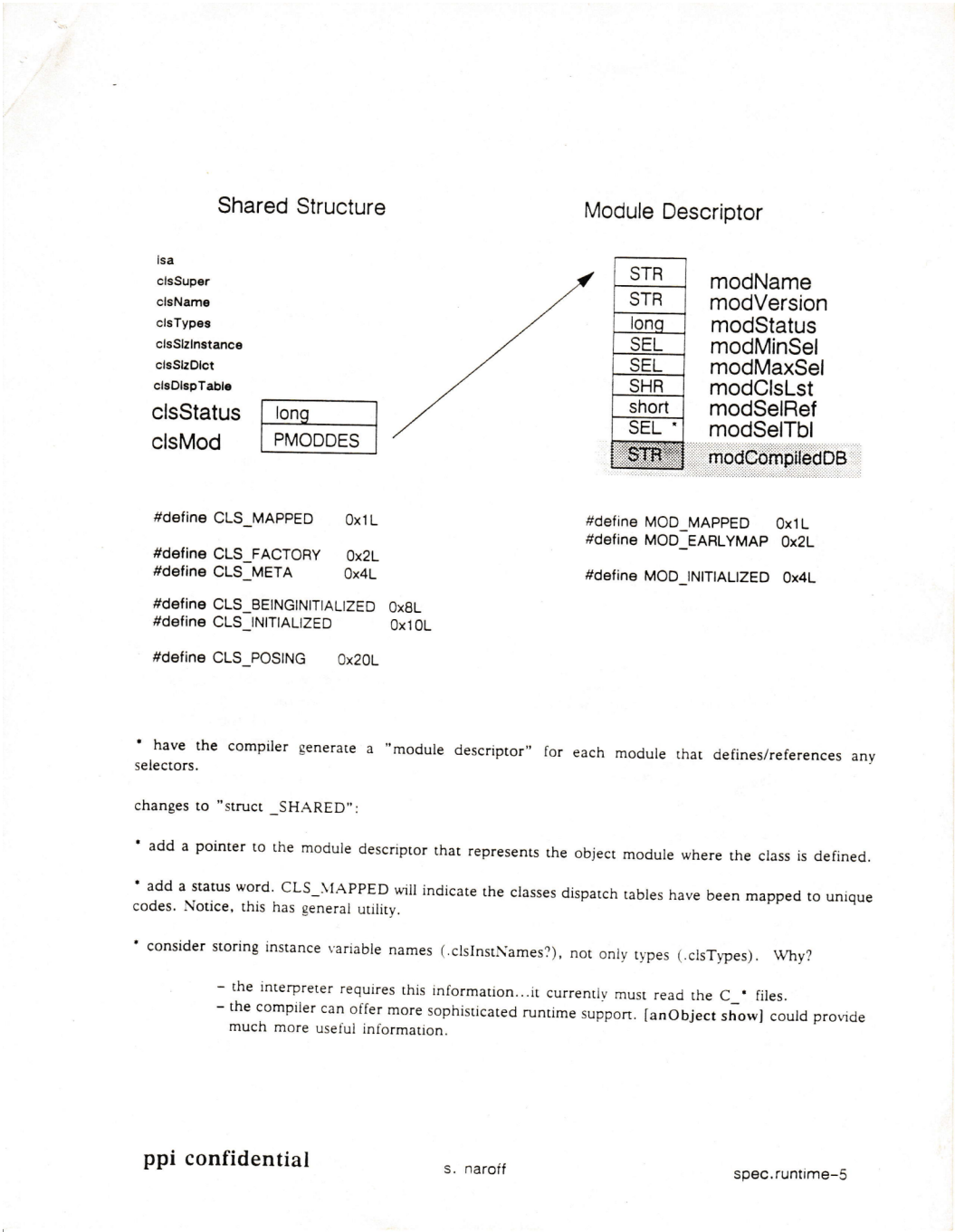
```
_msg(anId, _selRefs[0 <= n < _nSelRefs]);
```

- * each compilation unit that sends messages will contain its own local reference table, this replaces the current Phyla tables which are published to the entire application.
- * each entry will be mapped to a unique code at runtime.
- * the message dispatcher does not need to be modified.

ppi confidential

s. naroff

spec.runtime-4



CompileTime

Fruit.c

```
static struct modDescriptor _modDesc = {
    "Fruit",
    "objc v3.7",
    0L,
    (SEL)&__msgSelectors[0],
    (SEL)&__msgSelectors[44],
    &_Fruit,
    3,
    _selRefs
};
struct modDescriptor *_BIND$Fruit()
{
    return &_amp;modDesc;
}
```

Apple.c

```
static struct modDescriptor _modDesc = {
    "Apple",
    "objc v3.7",
    MOD_EARLYMAP,
    (SEL)&__msgSelectors[0],
    (SEL)&__msgSelectors[64],
    &_Apple,
    5,
    _selRefs
};
struct modDescriptor *_BIND$Apple()
{
    return &_amp;modDesc;
}
```

fruitMain.c

```
static struct modDescriptor _modDesc = {
    "fruitMain",
    "objc v4.0",
    0L,
    (SEL)&__msgSelectors[0],
    (SEL)&__msgSelectors[49],
    (SHR)0,
    8,
    _selRefs
};
struct modDescriptor *_BIND$fruitMain()
{
    return &_amp;modDesc;
}
```

LinkTime

__linkVector.c

```
#include "module.h"
PMODES _BIND$Fruit();
PMODES _BIND$Apple();
PMODES _BIND$fruitMain();

static struct modEntry {
    PMODES (*modLink)();
    PMODES modInfo;
} _modControl[] = {
    _BIND$Fruit,    0,
    _BIND$Apple,    0,
    _BIND$fruitMain, 0,
    0, 0
};

PMOD _objcModules = _modControl;
```

This file would be produced by a "post-linker" that would be part of the Objective-C compiler tools chain.

It would then be included as part of the application.

* the compiler must be taught to generate a `_BIND$` entry point for any source file that defines or references any selectors. This entry point will be used to describe information about the module to the Objective-C runtime support system.

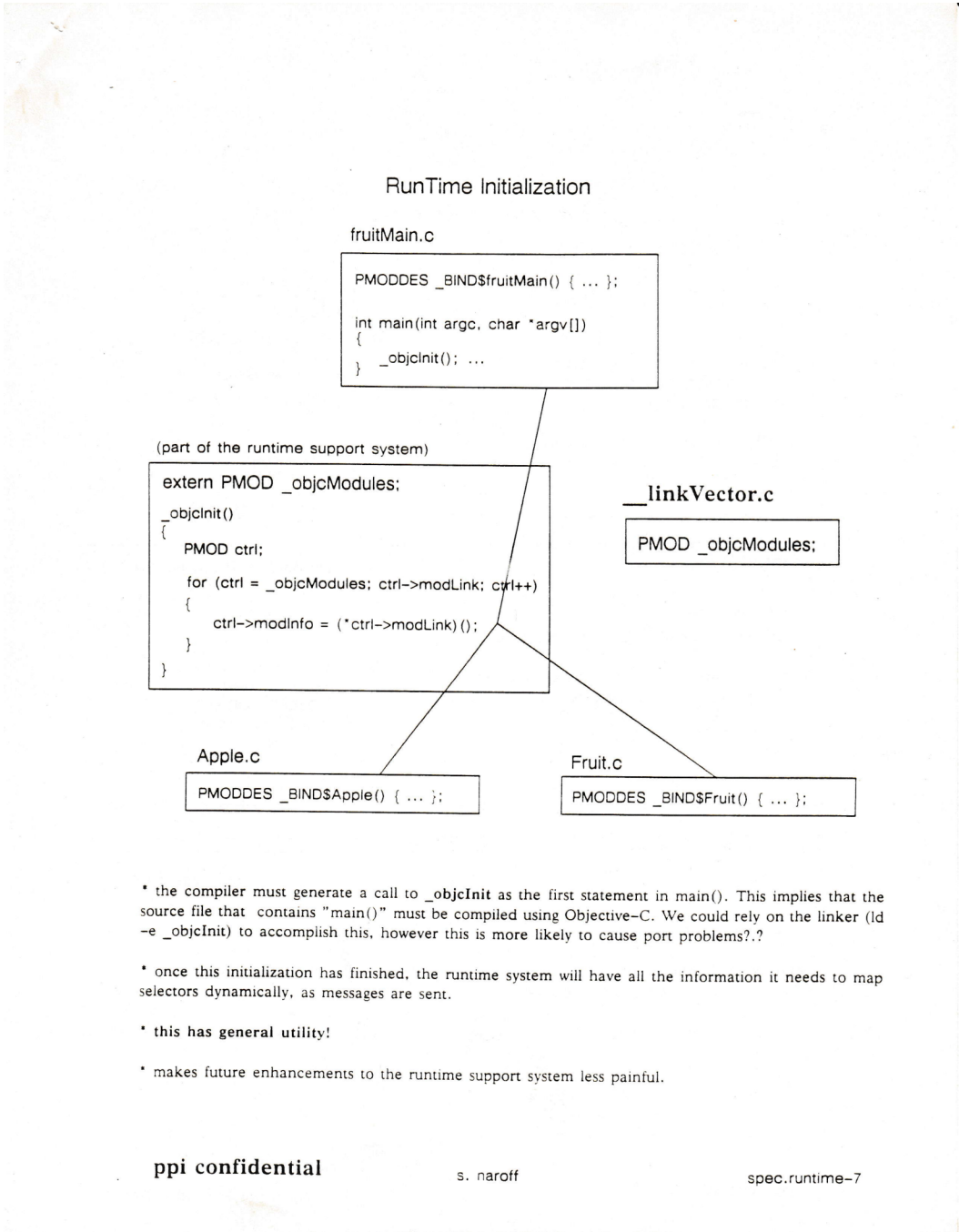
* replaces/automates "facilities" provided by the `@classes(ClassA, ClassB)` and `@messages` constructs.

* the control driver must be modified to incorporate the post link operation. This implies that applications written using Objective-C must be linked using the Objective-C control driver...(currently some "make" files might assume they can invoke the linker directly, this will no longer be acceptable).

ppi confidential

s. naroff

spec.runtime-6



Objective-C language constructs that are affected.

1. `@messages` – Tells the Objective-C compiler to generate a global table of unique selectors (codes).

```
static char __msgSelectors[] = { ... };
char *_minSelector= &__msgSelectors[0], *_maxSelector= &__msgSelectors[1369];
char *GroupA[] = { ... };
char *GroupB[] = { ... };
char *Primitive[] = { ... };
static char **__phylaTable[] =(
    GroupA,
    GroupB,
    Primitive,
    0
);
char ***_phylaTables = __phylaTable;
int _nPhyla = 3;
```

3. `@classes(A,B)`

Produces

```
extern struct _SHARED
    _A,
    _B,
    _Object;

// The factory objects belong with the object module for the class...
id    A = (id)&_A,
      B = (id)&_B,
      Object = (id)&_Object;

// the order of classes in this table "partially" determines the order in which classes are initialized.
static id *_clsTable[] =(
    &A,
    &B,
    &Object,
    0
);
id **_Classes = _clsTable; // modified by __insertClass(SHR *myClass)
int _nClasses = 3;
```

Research getting rid of this construct, without eliminating the concept of a class table. It would be nice to store A with the object module for the class.

2. `@selector` – This provides a compile-time way to get a handle on THE unique representation of a selector string. The selector code, if you will. This mechanism will now use the "`mapSelector()`" facility to obtain the unique representation, rather than rely on the global pools in the previous system.

ppi confidential

s. naroff

spec.runtime-9

Idiom

```
IMP cltnAtPut = [cltn methodFor:@selector(at:put:)];
(*cltnAtPut)(cltn, @selector(at:put:), (n), (obj));
```

Produces

```
IMP cltnAtPut = (*(id (*)(*))_msg)(cltn, Primitive[40]/*methodFor:*/, Primitive[61]);
(*cltnAtPut)(cltn, Primitive[61], (n), (obj));
```

6. @requires <classList> ;

Helps Objective-C manage lateral dependencies by writing this list to the "C_" file that represents the class being compiled. This is required mostly as a side-effect of present class and message group management. If the compiler stops managing these files automatically, perhaps this will become obsolete.

Usage

```
@requires Apple, Fruit;
```

Produces

```
extern id Apple, Fruit;
```

ppi confidential

s. naroff

spec.runtime-10

Objective-C runtime support macros/functions that are affected.

1. SEL, ISSELECTOR(sel), cvtToSel(STR aString)

Object.m:

```
+ (BOOL)instancesRespondTo:(STR) aSelector {
    if (!ISSELECTOR(aSelector))
        aSelector = (*_cvtToSel)(aSelector);
    if(aSelector == 0) // not a valid selector
        return NO; // quick answer
}
- (BOOL)respondsTo:(STR)aSelector { ... }
- perform:(STR)aSelector { ... }
- perform:(STR)aSelector with:anObject { ... }
- perform:(STR)aSelector with:obj1 with:obj2 { ... }
- doesNotRecognize:(STR)aMessage { ... }
- (IMP)methodFor:(SEL)aSelector { ... }
+ (IMP)instanceMethodFor:(SEL)aSelector { ... }
```

IdArray.m, Cltn.m, SortCltn.m:

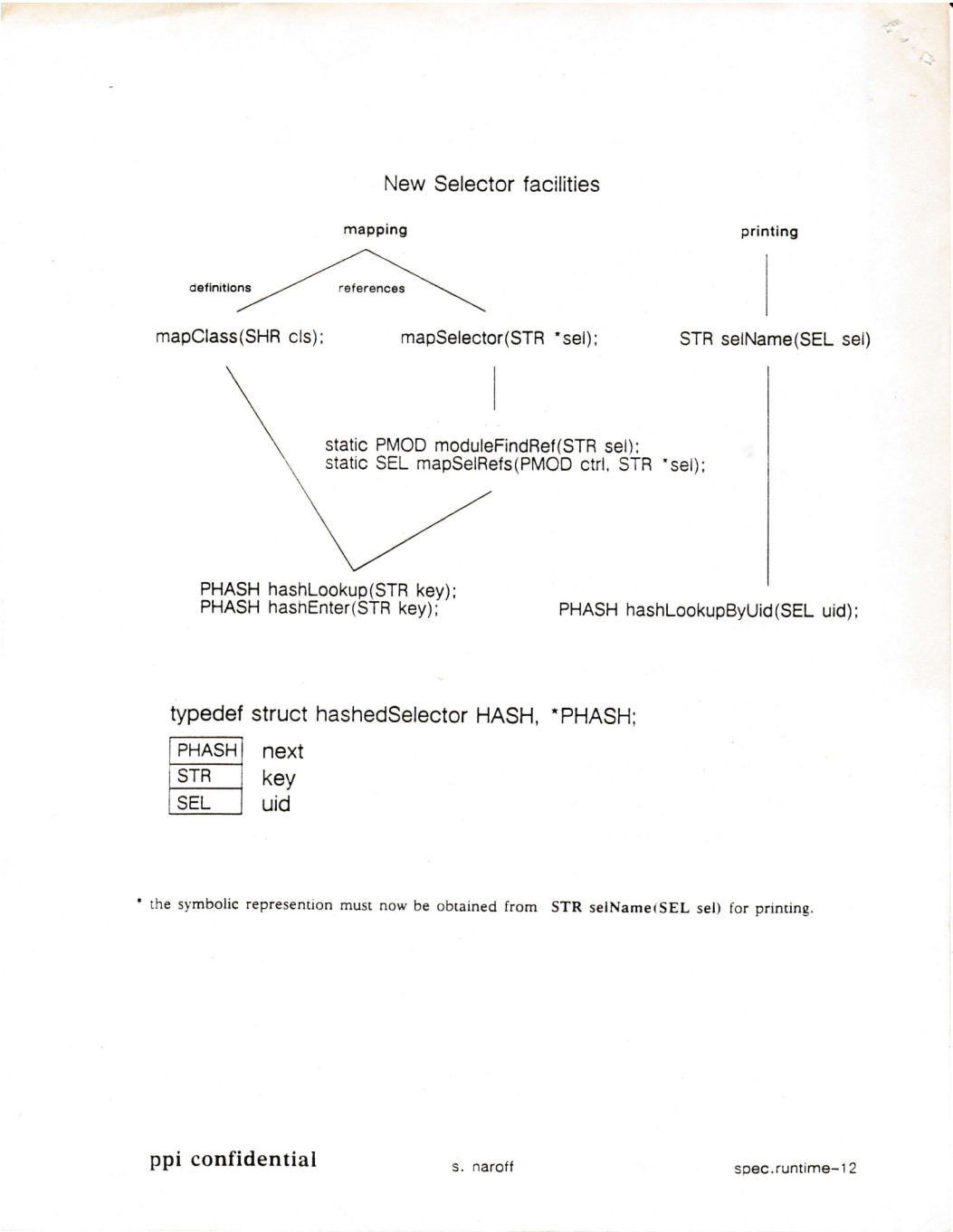
```
/*
 *      Objects that have selectors as instance variables that get
 *      filed out via the "AsciiFiler" present problems. For example:
 */
= SortCltn : BalNode (Collection, Primitive)
{
    SEL cmpSel; // selector to use for comparing
}
- elementsPerform:(SEL)aSelector { ... }
- elementsPerform:(SEL)aSelector with:obj1 { ... }
- elementsPerform:(SEL)aSelector with:obj1 with:obj2 { ... }
- elementsPerform:(SEL)aSelector with:obj1 with:obj2 with:obj3 { ... }

2. cvtToId(STR), __insertClass(SHR *)
3. _msg(id, SEL), _msgSuper(id, SEL), _msgImpFind(SHR, SEL, id *)
4. _prnFrame() { ... }, _osort(id *, int, SEL), _onExit(id, id, SEL)
5. "forloops.h" /* logic for looping through class/phyla tables */
```

ppi confidential

s. naroff

spec.runtime-11



REFERENCES

- Janet Abbate. 2012. Software crisis or identity crisis? Gender, labor, and programming methods. In *Recoding Gender: Women's Changing Participation in Computing*. MIT Press, Cambridge, MA, 73–111.
- Federico Biancuzzi and Shane Warden. 2009. Objective-C. In *Masterminds of Programming* (1st ed.). O'Reilly Media, Sebastopol, CA, 241–275. OCLC: 434042370.
- Barry W. Boehm. 1973. Software and its impact: A quantitative assessment. *Datamation* 19, 5 (May), 48–59.
- Frederick P. Brooks. 1975. *The Mythical Man-Month: Essays on Software Engineering* (1st ed.). Addison-Wesley, Reading, MA. Copy used: Jim Warren Book Collection, Lot X2595.2004, Box B14, Catalog 102676578, Computer History Museum, Mountain View, CA.
- Frederick P. Brooks. 1987. No silver bullet: Essence and accidents of software engineering. *Computer* 20, 4, 10–19. <https://doi.org/10.1109/MC.1987.1663532>
- Brad J. Cox. 1983a. The message/object programming model: A small change, at a deep conceptual level. In *Proceedings of Softfair: A conference on software development tools, techniques, and alternatives*. IEEE Computer Society Press, Arlington, VA (25–28 July), 51–60.
- Brad J. Cox. 1983b. The object oriented pre-compiler: Programming Smalltalk 80 methods in C language. *SIGPLAN Not.* 18, 1 (Jan.), 15–22. <http://doi.acm.org/10.1145/948093.948095> (retrieved 11 March 2013)
- Brad J. Cox. 1984. Message/object programming: An evolutionary change in programming technology. *IEEE Software* 1, 1 (Jan.), 50–61. <https://doi.org/10.1109/MS.1984.233398> (retrieved 4 March 2019)
- Brad J. Cox. 1986. *Object-Oriented Programming: An Evolutionary Approach* (1st ed.). Addison-Wesley, Reading, MA.
- Brad J. Cox. 1988. The Objective-C environment: past, present, and future. In *Digest of Papers. COMPCON Spring 88 Thirty-Third IEEE Computer Society International Conference*. IEEE, San Francisco, CA, USA (Feb.), 166–169. <https://doi.org/10.1109/CMPCON.1988.4852> (retrieved 4 March 2019)
- Brad J. Cox. 1989. Planning for software manufacturing. In *[1989] Proceedings of the Thirteenth Annual International Computer Software Applications Conference*. IEEE, Orlando, FL, USA (Sept.), 331–332. <https://doi.org/10.1109/CMPSAC.1989.65103> (retrieved 4 March 2019)
- Brad J. Cox. 1990a. Planning the software industrial revolution. *IEEE Software* 7, 6 (Nov.), 25–33. <https://doi.org/10.1109/52.60587> (retrieved 12 May 2009)
- Brad J. Cox. 1990b. There is a silver bullet: A software industrial revolution based on reusable and interchangeable parts will alter the software universe. *Byte* 15, 10 (Oct.), 209.
- Brad J. Cox. 2016. Oral history. 2 Aug. 2016. <https://www.computerhistory.org/collections/catalog/102717175> (retrieved 3 March 2020). CHM Oral History Collection, Lot X7863.2017, Catalog 102717175, Computer History Museum, Mountain View, CA.
- Brad J. Cox and Bill Hunt. 1986. Objects, icons, and software-ICs. *Byte* 11, 8 (Aug.), 161–176.
- Brad J. Cox and Andrew J. Novobilski. 1991. *Object-Oriented Programming: An Evolutionary Approach* (2nd ed.). Addison-Wesley, Reading, MA.
- Brad J. Cox and Kurt J. Schmucker. 1987. Producer: A tool for translating Smalltalk-80 to Objective-C. *SIGPLAN Not.* 22, 12, 423–429.
- Edsger W. Dijkstra. 1968. Letters to the editor: go to statement considered harmful. *Commun. ACM* 11, 3 (March), 147–148. <https://doi.org/10.1145/362929.362947> (retrieved 8 Aug. 2014). Also available in Edsger W. Dijkstra. 1979. Go to statement considered harmful. In *Classics in Software Engineering*, Edward Nash Yourdon (Ed.). Yourdon Press, Upper Saddle River, NJ, USA, 27–33. <http://dl.acm.org/citation.cfm?id=1241515.1241518> (retrieved 9 Aug. 2014).
- Nathan L. Ensmenger. 2010. *The "Computer Boys" Take Over: Computers, Programmers, and the Politics of Technical Expertise*. MIT Press, Cambridge, MA.
- Nathan L. Ensmenger and William Aspray. 2002. Software as labor process. In *History of Computing: Software Issues*, Ulf Hashagen, Reinhard Keil-Slawik, and Arthur L Norberg (Eds.). Springer, Berlin, 139–165.
- Blaine Garst. 2016. Oral history part 1. 25 July 2016. <https://www.computerhistory.org/collections/catalog/102717171> (retrieved 3 March 2020). CHM Oral History Collection, Lot X7853.2017, Catalog 102717171, Computer History Museum, Mountain View, CA.
- Adele Goldberg and David Robson. 1983. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, Reading, MA.
- James Gosling. 2019. Oral history part 2 of 2. 22 April 2019. <https://www.computerhistory.org/collections/catalog/102781105> (retrieved 3 March 2020). CHM Oral History Collection, Lot X8971.2019, Catalog 102781105, Computer History Museum, Mountain View, CA.
- Michael A. Hiltzik. 1999. *Dealers of Lightning: Xerox PARC and the Dawn of the Computer Age* (1st ed.). HarperBusiness, New York.
- Ted Kaehler and Dave Patterson. 1986. A small taste of Smalltalk. *Byte* 11, 8 (Aug.), 145–158.

- Alan C. Kay. 1993. The early history of Smalltalk. In *The second ACM SIGPLAN conference on History of programming languages (HOPL-II)*. Association for Computing Machinery, Cambridge, MA (March), 69–95. <https://doi.org/10.1145/154766.155364> (retrieved 10 May 2009). Also available as a chapter in Thomas J. Bergin, Jr. and Richard G. Gibson, Jr. (Eds.). 1996. *History of Programming Languages—II*. Association for Computing Machinery, New York, NY, USA, 511–598. <https://doi.org/10.1145/234286.1057836>.
- Glenn Krasner. 1984. *Smalltalk-80: Bits of History, Words of Advice*. Addison-Wesley, Reading, MA. OCLC: 266966827.
- Lamar Ledbetter and Brad J. Cox. 1985. Software-ICs: A plan for building reusable software components. *Byte* 10, 6 (June), 307–316.
- Tom Love. 1983. Experiences with Smalltalk-80 for application development. In *Proceedings of Softfair: A conference on software development tools, techniques, and alternatives*. IEEE Computer Society Press, Arlington, VA (25–28 July), 61–65.
- Tom Love. 1988. The economics of reuse (of software). In *Digest of Papers. COMPCON Spring 88 Thirty-Third IEEE Computer Society International Conference*. IEEE, San Francisco, CA, USA (Feb.), 238–241. <https://doi.org/10.1109/COMPCON.1988.4866> (retrieved 13 March 2019)
- Tom Love. 1993. *Object Lessons: Lessons Learned in Object-Oriented Development Projects*. SIGS Books, New York, NY. OCLC: 612808297.
- Tom Love. 2019. Skype interview. 17 April 2019. <https://www.computerhistory.org/collections/catalog/102781110> (retrieved 12 March 2020). CHM Oral History Collection, Lot X9026.2019, Catalog 102781110, Computer History Museum, Mountain View, CA.
- Donald MacKenzie. 2001. *Mechanizing Proof: Computing, Risk, and Trust*. MIT Press, Cambridge, MA.
- Michael S. Mahoney. 1990. The roots of software engineering. *CWI Quarterly* 3, 4, 325–334. <http://www.princeton.edu/~hos/Mahoney/articles/sweroots/sweroots.htm> (retrieved 3 March 2020). PDF version available at: <http://thecorememory.com/TROSE.pdf> (retrieved 3 March 2020). Extracts also available in Michael S. Mahoney. 2011. *Histories of Computing*. Harvard University Press, Cambridge, MA; London, England, 86–89.
- Michael S. Mahoney. 2002. Software: The self-programming machine. In *From 0 to 1: An Authoritative History of Modern Computing*, Atsushi Aker and Frederik Nebeker (Eds.). Oxford University Press, New York, 91–100. Also available in Michael S. Mahoney. 2011. *Histories of Computing*. Harvard University Press, Cambridge, MA; London, England, 77–85.
- Michael S. Mahoney. 2004. Finding a history for software engineering. *Annals of the History of Computing, IEEE* 26, 1, 8–19. <https://doi.org/10.1109/MAHC.2004.1278847> (retrieved 7 April 2009). Also available in Michael S. Mahoney. 2011. *Histories of Computing*. Harvard University Press, Cambridge, MA; London, England, 90–105.
- Steve Naroff. 2018. Oral history, part 1 of 2. 8 Oct. 2018. <https://www.computerhistory.org/collections/catalog/102717385> (retrieved 3 March 2020). CHM Oral History Collection, Lot X8800.2019, Catalog 102717385, Computer History Museum, Mountain View, CA.
- Steve Naroff and Alan Watt. 1987. Design issues for Objective-C v.?? DRAFT. 3 July 1987. <https://doi.org/10.5281/zenodo.3708485>
- Kurt J. Schmucker. 1986a. MacApp: An application framework. *Byte* 11, 8 (Aug.), 189–193.
- Kurt J. Schmucker. 1986b. Object-oriented languages for the Macintosh. *Byte* 11, 8 (Aug.), 177–185.
- Rebecca Slayton. 2013. *Arguments that Count: Physics, Computing, and Missile Defense, 1949-2012*. The MIT Press, Cambridge, MA.
- Bjarne Stroustrup. 1993. A history of C++: 1979–1991. In *The second ACM SIGPLAN conference on History of programming languages (HOPL-II)*. Association for Computing Machinery, Cambridge, MA, USA (March), 271–297. <https://doi.org/10.1145/154766.155375> (retrieved 30 April 2009). Also available as a chapter in Thomas J. Bergin, Jr. and Richard G. Gibson, Jr. (Eds.). 1996. *History of Programming Languages—II*. Association for Computing Machinery, New York, NY, USA, 699–769. <https://doi.org/10.1145/234286.1057836>.
- Larry Tesler. 1981. The Smalltalk Environment. *Byte* 6, 8 (Aug.), 90–147.
- Larry Tesler. 1986. Programming experiences. *Byte* 11, 8 (Aug.), 195–206.
- Larry Tesler. 2013. Oral history. 13 Feb. 2013. <https://www.computerhistory.org/collections/catalog/102746675> (retrieved 26 Feb. 2020). CHM Oral History Collection, Lot X6762.2013, Catalog 102746675, Computer History Museum, Mountain View, CA.
- Larry Tesler. 2016. Oral history, part 2 of 3. 16 Dec. 2016. <https://www.computerhistory.org/collections/catalog/102717269> (retrieved 3 March 2020). CHM Oral History Collection, Lot X8020.2017, Catalog 102717269, Computer History Museum, Mountain View, CA.
- James E. Tomayko. 2002. Software as engineering. In *History of Computing: Software Issues*, Ulf Hashagen, Reinhard Keil-Slawik, and Arthur L. Norberg (Eds.). Springer, Berlin, 139–165.
- John W. Verity. 1987. The OOPS revolution. *Datamation* 33 (May), 72–78.
- Copy used: Smalltalk press and clippings, 1983–2002, Adele Goldberg papers, Lot X5774.2010, Box 2, Folder 14, Catalog 102739382, Computer History Museum, Mountain View, CA.

John W. Verity and Evan I. Schwartz. 1991. Software made simple. *BusinessWeek* (30 Sept.), 92–100.

Copy used: Smalltalk press and clippings, 1983–2002, Adele Goldberg papers, Lot X5774.2010, Box 2, Folder 14, Catalog 102739382, Computer History Museum, Mountain View, CA.

Eva White and Rich Malloy. 1986. Object-oriented programming. *Byte* 11, 8 (Aug.), 137.

Xerox Learning Research Group. 1981. The Smalltalk-80 system. *Byte* 6, 8 (Aug.), 36–48.

NON-ARCHIVAL REFERENCES

Apple Inc. 2020a. Extensions. <https://docs.swift.org/swift-book/LanguageGuide/Extensions.html> (retrieved 26 Feb. 2020)

This website describes the Extensions feature in Swift, which allows the addition of new functionality to types for which the programmer does not have access to the source code, similar to categories in Objective-C. We refer to it to illustrate how important categories in Objective-C became to Apple’s language idioms and have continued in Swift, albeit under a new name.

Apple Inc. 2020b. Protocols. <https://docs.swift.org/swift-book/LanguageGuide/Protocols.html#ID521> (retrieved 26 Feb. 2020)

This website describes the Protocols feature of Swift, which defines an abstract data type interface that types can adopt, conform to, and implement. Protocols provide multiple inheritance of interface but not implementation, as conforming types provide their own implementation. Extensions on Protocols allow programmers to add methods and their implementations to a protocol which all conforming classes will automatically gain. We refer specifically to the section of the website describing Extensions on Protocols in Swift to document that this functionality originally proposed for Objective-C in 1995 but never shipped finally became available in the Swift successor language in 2014.

Clang Team. 2020. Objective-C Literals. <https://clang.llvm.org/docs/ObjectiveCLiterals.html> (retrieved 12 March 2020)

This website documenting Apple’s open source Clang compiler front end describes three new features for Objective-C: NSNumber literals for creating boxed numbers from scalar literal expressions, Collection literals providing short-hand for creating arrays and dictionaries, and Object Subscripting to provide a way to use subscripting with Objective-C objects, including collections. These features are available starting with Apple LLVM Compiler 4.0 or open source clang 3.1. These language additions simplify common Objective-C programming patterns, make programs more concise, and improve the safety of container creation.

Robert X. Cringely. 1996. Triumph of the Nerds: The Transcripts, Part III. June 1996. <http://www.pbs.org/nerds/part3.html> (retrieved 2 March 2020) Archived at Internet Archive: <https://web.archive.org/web/20200104110141/http://www.pbs.org/nerds/part3.html> (4 Jan. 2020 11:01:41)

Steve Jobs: “And they showed me really three things. But I was so blinded by the first one I didn’t even really see the other two. One of the things they showed me was object orient[ed] programming... but I didn’t even see that. The other one they showed me was a networked computer system... I didn’t even see that. I was so blinded by the first thing they showed me which was the graphical user interface.”

Mark Dalrymple. 2012. Objective-C Literals, Part 1. March 2012. <https://www.bignerdranch.com/blog/objective-c-literals-part-1/> (retrieved 12 March 2020)

This blog post describes three new features available in Apple’s open source Clang compiler front end: more concise NSNumber, NSArray, and NSDictionary creation syntax via literals, and more concise NSArray and NSDictionary access syntax via subscripting.

Thomas Haigh. 2010. Dijkstra’s crisis: The end of Algol and the beginning of software engineering: 1968–1972. http://www.tomandmaria.com/tom/Writing/DijkstrasCrisis_LeidenDRAFT.pdf (retrieved 3 March 2020)

This unpublished chapter written by Tom Haigh for the SOFT-EU Project Meeting of Sept 2010 argues that the relationship of the 1968 NATO Conference on Software Engineering and its invocation of the software crisis to later software engineering practice has been fundamentally misunderstood. That conference was primarily made up of computer scientists who had been part of the IFIP Working Group working on Algol-68, and the NATO conference was focused on the concerns of these computer scientists with promoting formal methods of programming. This had little to do with the kinds of organizational approaches actually taken by software engineers in the 1980s.

Tom Love. 2019. E-mail with Tom Love, Subject: Yesterday’s Questions. 1 June 2019.

1. How many customers did Stepstone have when I resigned from the company?

Hundreds of companies including: Apple, HP, IBM, Siemens, Philips, US West, Accuray (acquired by ABB), Port of Singapore, GE Medical, Brooklyn Union Gas, Prime Computer, Lawrence Livermore Labs, Artecon, Sun, Tektronix, NeXT, Ford Motor Company, and many more prominent companies and organizations including NSA in Japan, Germany, and the UK. Hiding in my store room is a notebook with business cards from these companies and many many others. Most of these companies that used Objective-C were described in my 1993 book, Object Lessons.

2. What were the business relationships with companies like HP, Apple and NeXT?

All purchased Objective C and later IC-pacs. HP surely had a corporate wide license as did NeXT. All signed a PPI and later Stepstone purchase contract. NeXT had the ‘sweetest deal’ \$5 for every device running Objective C. Just as I was leaving Stepstone, IBM, NeXT, and HP were negotiating a joint press release committing to the bundling of Objective-C on their company’s workstations. I naively [*sic*] thought this was a ‘sure thing’ and that Stepstone was launched and would continue to grow and prosper. I created Orgware, a one person consulting company doing work with VCs, start up companies, and large companies like Ericsson, Philips, JP Morgan, and Dun and Bradstreet.

3. Were there development contracts with these Objective-C companies?

Product purchases and associated contracts but generally not development contracts. I believe there was a development contract with NeXT after I left the company. Apple bought an early copy of Objective-C and later often said, “we never really used Objective-C, but we really appreciated the testing of our C compiler that PPI/Stepstone did for us”. HP used Objective-C extensively at many locations worldwide.

Microsoft. 2015. *Extension Methods — C# Programming Guide* (July). <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/extension-methods> (retrieved 2 Aug. 2019)

This website describes Extension methods in C#, which enable the addition of methods to a class without subclassing, recompiling, or otherwise modifying the original type. We refer to this website to document that C# has a feature similar to categories in Objective-C.

Patrick Naughton. 1997. Java on NewtonOS Rumor. 20 April 1997. <https://groups.google.com/forum/#!msg/comp.sys.newton.misc/Kb0JoVv2ljA/IyQl8DYhzJEJ> (retrieved 11 March 2020)

This newsgroup posting originally posted to comp.sys.newton.misc by Patrick Naughton explains how Objective-C was influential on the early Oak (later Java) project, both because Naughton himself liked it and had considered going to NeXT, and because the Oak team ended up hiring many former NeXT engineers. Naughton explains that “Java’s ‘interface’ is a direct rip-off of Obj-C’s ‘protocol’ which was largely designed by these ex-NeXT’ers...”

Also available at Patrick Naughton. [n.d.]. *Java Was Strongly Influenced by Objective-C*. <https://cs.gmu.edu/~sean/stuff/java-objc.html> (retrieved 2 March 2020) Archived at Internet Archive: <https://web.archive.org/web/20190617000753/https://cs.gmu.edu/~sean/stuff/java-objc.html> (26 Jan. 2003 20:40:50).

Stefan L. Ram. 2003. *Dr. Alan Kay on the meaning of “object-oriented programming”* (23 July). http://www.purl.org/stefan_ram/pub/doc_kay_oop_en (retrieved 2 March 2020) Archived at Internet Archive: https://web.archive.org/web/20200108225452/http://userpage.fu-berlin.de/~ram/pub/pub_jf47ht81Ht/doc_kay_oop_en (8 Jan. 2020 22:54:48)

This is an e-mail thread between Alan Kay and Dr. Stefan Ram dated July 23, and July 26, 2003, in which Kay clarifies his definition of the term “object-oriented,” which he coined. We refer to this to illustrate the difference between Kay’s meaning of the term “object-oriented,” which stressed message passing and late binding, and Bjarne Stroustrup’s definition, in which inheritance is the key feature.