A History of Object-Oriented Programming Languages and their Impact on Program Design and Software Development

A program is simply a sequence of commands instructing a computer what to do. The degrees of freedom available in devising this list, make programs potentially the most complex and intricate entities ever envisioned by humans. For this reason, it is imperative that they are subdivided, otherwise they would soon become unmanageable and incomprehensible. It is essentially the different ways in which this can be accomplished which has engendered the development of programming design methods and subsequently languages which facilitate their implementation.

Early languages could be categorised as procedural and applicative. The former predominantly embody a series of instructions to assign values to variables, while the latter resemble mathematical function definitions and ideally have no statements, only expressions without side effects (Parker 1988). During the 1960s, a new discipline, object-orientation emerged. Although the first language of this type originated towards the end of the 1960s, it is only in the last decade that its employment has become widespread because of the benefits it confers over existing methods. As the requirement for systems becomes ever more prodigious and elaborate, it is perceived as a means by which greater reliability and easier maintenance can be achieved.

In order to understand the advantages stipulated by the inventors of object-oriented languages, it is important to appreciate the conceptual distinctions between these and traditional imperative programming languages. In the latter, a software schema may be data or process driven, with functions and variables attributed different levels of importance. Rather than considering procedures and data as being separate, object-orientation unifies them into an ensemble called an object. An object's state is a function of its data or instance variables, which can only be accessed by sending a message asking the object, an operation which entails calling its functions known as methods or member functions (Lafore 1991). This occasions the notion of an object possessing behaviour, with messages sent to it causing a modification of that behaviour. The enclosure and protection of data is called encapsulation and results in the state of an object being hidden from procedures external to it. The importance of this in planning a program cannot be underestimated. It encourages a delegation of behaviour to objects, whilst hindering the cross-coupling of the internals of an object with other areas of the system (Computer Applied Technology 1993). This should diminish the probability of a rogue program being capable of altering data, since it should be concealed from all but the pertaining object. Additionally, when modelling a real world item, it is more natural to

imagine it in terms of an object than a function, which assists the design process.

In essence, object-orientation is a form of organisation and can be performed to some degree with any programming language. However, in order to exploit the full potential of this methodology, several languages have been conceived specifically for this purpose.

The first of these was Simula. Although it never became widely used, it was the archetype of the concepts of object-oriented programming and has been highly influential on its successors. Its designers were Ole-Johan Dahl and Kristen Nygaard, who worked at the Norwegian Computing Centre in Oslo. What they had originally envisaged in 1962 was a simple simulation procedure package along with a pre-processor for ALGOL 60, the foremost programming language in Europe at that time. Their endeavours culminated in a prototype Simula I compiler running on a UNIVAC 1107 mainframe in 1964. To create an all-purpose, standalone language, the designers wanted to generalise and consolidate the concepts they had formulated and when Tony Hoare imparted his idea of a record class construct in ALGOL bulletin no. 21, 1965, they realised they needed some kind of object with record class properties. When prefixing was introduced in 1966, they had what they were after: an object consisting of a prefix layer and a main layer, the former containing references to its predecessor and successor and the latter accommodating its attributes. Combined with the exploration of string handling and input/output for the language by Bjoern Myhrhaug and the concept of classes, Simula 67, later renamed Simula was born. Work began on the production of compilers for an assortment of mainframes, including IBM, UNIVAC, CDC and DEC machines (Holmevik 1995).

Nygaard's language development did not cease, his efforts being devoted to the systems description language Delta completed in 1975 and to the refinement of the related notions of classes, records, types and procedures into a higher-level abstraction mechanism, the pattern, incorporated in the language, Beta (Knudsen 1997).

One of the languages that embraced the concepts of class and message of Simula, was Smalltalk. A project commenced at Xerox PARC in the early 70s with the aim of creating the quintessential dynamic object-oriented language. This is one which allows new classes, objects and behaviour to be appended on the fly by representing the class hierarchy, objects and methods of a program as meta-data at run-time. Early versions appeared biannually between 1972 and 1978, but unfortunately, the language was not standardised hence there are a number of commercial dialects in existence today (Latta 1995). This is not a problem with Eiffel

which was produced with similar aims by Bertrand Meyer in 1985 to increase productivity and software quality (Arnaud 1998).

Object-orientation could only take off if programmers could be weaned on to it by evolving a new language from an existing one, so that old source code would not have to be rewritten and so the new techniques could be progressively mastered. Arguably the most successful of these is C++. C had been invented by Dennis Ritchie of Bell Laboratories in 1972 and it had become the prevalent language of its time partly due to its translation into efficient machine code, its portability and versatility, but principally because it underpins the UNIX operating system. C++, which was credited to Bjarne Stroustrup of AT&T Bell Laboratories Computing Science Research Centre in 1983, has few incompatibilities with C, permitting gradual familiarisation with the procedural improvements and features to support data abstraction and the object model (Holmes 1992). It was not the only undertaking to extend C, another example being Objective-C by Brad Cox in 1984, but it has become far more popular amongst programmers (Schoenmakers 1998). To retain compatibility with C, there are compromises which make it a retrograde step from Smalltalk. For example, by implementing static object-orientation, any change in a class necessitates total recompilation and its lack of garbage collection can generate severe memory management problems.

Other languages have been expanded to encompass object-orientation. Microsoft have made their own version of BASIC called Visual BASIC and Borland, with their successful Turbo Pascal environment have been expeditious in introducing the methodology to their product and have also released their own visual tool, Delphi (Kinnersley 1998, Andersson 1997). While these languages tie programmers to one operating system, others have been standardised to institute greater cross-platform availability. As successors to Modula 2, in the late 80s, DEC and Olivetti designed Modula 3 and Wirth and Gutknecht, Oberon, now renamed Component Pascal (Harbison 1992, Oberon Microsystems 1997). Meanwhile, a recommendation for an object-oriented version of Ada, eventually to be Ada 95, was composed by the Ada Board, a Federal advisory committee to the Ada Joint Project Office set up by the US Department of Defence (Guerby 1996). The architects of Smalltalk strived to overcome its shortcomings and their research centred on supplementing Lisp with the Common Lisp Object System, through Flavors (Franz Incorporated 1998). Not even the venerable COBOL is being ignored. Commercial object-oriented variants already exist, such as Micro Focus Object COBOL, although standardisation efforts by ANSI (J4) and ISO (WG4) groups have not yet finished (Klein 1998).

The revolution in methodology has not been avoided by the scripting languages[1], with Perl and REXX being upgraded, the latter undergoing a name change to Object REXX (Christiansen 1996, Cowlishaw 1997). Python is an example of a new scripting language, the origin of its syntax and semantics being Modula 3 (Rossum 1998).

A list of object-oriented languages in existence or under development is given in Appendix A, but this examination of their history would not be complete without reference to Java. Delivered by Sun Microsystems in 1995, its original objective was to let devices, peripherals and appliances possess a common programming interface. However, the tremendous upsurge in browsing the Internet enabled its machine independence to be utilised in facilitating far more functionality to be offered by web sites through Java applets. A Java compiler outputs a pseudo-code which must be executed on any computer having an appropriate interpreter known as a Java virtual machine. With a similar syntax and many improvements to C++, like garbage collection, its deployment has been extremely rapid (Harold 1998). A recent survey reveals that over 40% of US businesses asked are already using it and over 30% intend to soon (Andrews 1998).

All these languages possess certain distinguishing characteristics: classes, inheritance, polymorphism and ease of modification.

A class is a definition of the type of an object. In procedural languages, the type may be an elementary data type, such as an integer or a collection of basic types within some sort of record or structure, for example a ‘struct’ in C. The fundamental difference between these and classes are that the latter define functions as well as data types. This has the advantage that the behaviour is not hardwired and classes can be developed as abstractions from particular problems, allowing expression of not just the data to be manipulated, but the methods that will operate on that data.

Enlarging the concept of class is inheritance. When declaring a new class, it can be derived from one or more base or parent classes. When an object of the child class is instantiated, its behaviour depends upon the members and instance variables of the child and parent classes. It can respond in the same way to the same stimuli as objects of the parent class and regulating its behaviour involves altering only the base class (Computer Applied Technology 1993). By inheriting the capabilities of the parent class, functions can be reused and the connection between elements of an object-oriented program is made manifest.

[1] Scripting language: interpreted high-level language generally used for system administration and text manipulation tasks within operating systems. A DOS batch file is an example of a script.

Stemming from inheritance is polymorphism. Several functions can be declared with the same name, but they must have some other identifying trait. This could be the function type or the number of or data types of input parameters. This requires dynamic binding, in which the correct method to execute must be determined at run-time. Operators like '+' and '=' may also be polymorphic, in which case they are said to be overloaded, but this is contingent upon the language (Lafore, 1991). The member functions of a base class may be overridden in a derived class, by duplicating the parent's declaration in the child. Thus, objects of the two classes may both react to the same messages, but can do so in distinct manners.

One of the goals of object-oriented languages is flexible programs. A minor shift in requirement of a program should not demand an entire rewrite and if other programs share any of its features, adaptation of the relevant sections could save time and money. The data dependencies inherent in procedural programs present a formidable obstacle to this goal, but one which has been largely surmounted by the localisation of change made possible by encapsulation, provided discipline is imposed on the design of classes. Future software production can be expedited by the construction of libraries of classes, the division of programs into objects making this task relatively straightforward.

The development of languages has not abated and there are those who are working on progressing beyond object-orientation. Research teams are investigating blending object-oriented and functional concepts to create new programming languages. Another idea, as illustrated by the language Beta, is to abstract further to a basic construct from which classes, types, procedures and records are composed (Knudsen 1997).

A new object model under research at the University of Twente in the Netherlands is the composition filters object model which amongst other benefits, enables dynamic inheritance and delegation, reflection on the interaction between, and the abstraction of communications among objects (Koopmans 1996).

It is to be hoped that these advancements will complement the improvements in development, maintenance and reusability of software inaugurated by object-orientation and the languages which conform to the objectives of this methodology.

Appendix A

List of some of the Object-Oriented Programming Languages in Existence Today

1. A#
2. ABCL
3. Abel
4. Actor
5. Acttalk
6. Ada
7. ADES
8. ADVSYS
9. Agora
10. Alcool-90
11. ALLOY
12. ALTRAN
13. AppleScript
14. ASDL
15. A'UM
16. BeBOP
17. Beta
18. BLAZE 2
19. Blue
20. Bob
21. BOPL
22. C*
23. C++
24. C+@
25. Cantor
26. Cecil
27. CESP
28. CHARM++
29. CIEL
30. Classic-Ada
31. Click-n
32. CLIPS
33. CLIX
34. Clu
35. Cluster-86
36. Common Lisp Object System (CLOS)
37. CommonLoops
38. Common Objects
39. Concurrent Aggregates (CA)
40. Concurrent Smalltalk

41. ConstraintLisp
42. cooC
43. COOL
44. CSSA
45. Denali
46. DinnerBell
47. DOWL
48. DRAGOON
49. DROOL
50. Dylan
51. Echidna
52. Eden
53. Eiffel
54. Ellie
55. ELLIS
56. ELSIE
57. Emerald
58. EMPL
59. ETHER
60. EXTRA
61. FLEX
62. FMPL
63. FOOP
64. Formes
65. Fresco
66. G
67. GEL
68. HERAKLIT
69. HOOK
70. Hybrid
71. IDOL
72. InnovAda
73. ISLisp
74. Jade
75. Java
76. Kaleidoscope
77. Kevo
78. LAMINA
79. Late-bound Encapsulated Name Spaces (LENS)
80. LAURE
81. Leda
82. LIFE
83. LITHE
84. LOGLAN
85. LOOKS
86. LOOPN
87. LOOPS
88. Lore

89. LUKKO
90. MACE
91. MCS
92. MELD
93. MeldC
94. Mentat
95. MEROON
96. Mode
97. MODSIM II
98. Modula-3
99. MooZ
100. Neon
101. Newton
102. Oaklisp
103. Oberon-2
104. Objective-C
105. Object-CHILL
106. Object Lisp
107. ObjectLOGO
108. Object Oberon
109. Object-Oriented COBOL
110. Object-Oriented Fortran
111. Object-Oriented Turing
112. Object Pascal
113. Object REXX
114. ObjectTCL
115. ObjVlisp
116. Oblique
117. Oblog
118. Omega
119. Ondine
120. Ontic
121. Orca
122. Orient84/K
123. OSCAR
124. O'small
125. Oz
126. Parasol
127. Parlog++
128. Pascal Plus
129. PECOS
130. PHOCUS
131. Pict
132. Polka
133. POLYGOTH
134. POOL2
135. POP++
136. PopTalk

137. Probe
138. Prolog++
139. PROOF/L
140. Python
141. Real-Time Mentat
142. ROME
143. Rossette
144. RTC++
145. Sather
146. SCOOP
147. SCOOPS
148. ScriptX
149. SDL-92
150. Self
151. Sina
152. Siri
153. SmallWorld
154. SNOOPS
155. Solve
156. Spool
157. TAO
158. TELOS
159. Theta
160. Traits
161. Trellis
162. Tuple Space Smalltalk
163. Turbo Pascal V.6
164. UC++
165. United Functions and Objects (UFO)
166. USSA
167. VDM++
168. Venari
169. XDL
170. XLISP
171. XScheme
172. yacc
173. Z++

References

Andersson, M. (1997, May 28) *Delphi Frequently Asked Questions.* <http://www.sbrain.syh.fi/delphi/delphi_faq_1.html>

Andrews, D. (1998, March) "Survey Reveals Java Adoption Plans." *Byte Magazine*. 23(3). pp.26,30.

Arnaud, F. (1998, March 12) *Eiffel Frequently Asked Questions.* <http://sunsite.doc.ic.ac.uk/usenet/usenet-by-hierarchy/comp/lang/eiffel/comp.lang.eiffel_Frequently_Asked_Questions_(FAQ)>

Austern, M. *What is Java?* (1997, November 19) <http://reality.sgi.com/employees/austern/java.html>

Christiansen, T. (1996) *What is Perl5?* <http://language.perl.com/info/perl5-brief.html>

Computer Applied Technology (1993) *What is object-oriented design?* Computer Applied Technology CBT.

Cowlishaw, M. (1997, July 30) *Where do we stand today?* <http://www2.hursley.ibm.com/orexx/section2.htm>

Davis, R. (1997, June 20) *The Object-Oriented Page.* <http://www.well.com/user/ritchie/ritchie.html>

Encyclopaedia Britannica, Inc. (1997) *Computer Science: Programming Languages.* Britannica CD. Version 97.

Encyclopaedia Britannica, Inc. (1997) *Computers: Programming Languages.* Britannica CD. Version 97.

Franz Incorporated. (1998, March 4) *Enabling Applications that Adapt to Changing Needs.* <http://www.franz.com/tech/wp.html>

Guerby, L. (1996, July 16) *Evolution of Ada95.* <http://www.adahome.com/LRM/9X/Rationale/rat95html/rat95-p1-1.html>

Harbison, S.P. (1992) *Modula-3*. Prentice Hall.

Harold, E.R. (1998, March) *Java Frequently Asked Questions.* <http://sunsite.doc.ic.ac.uk/usenet/usenet-by-hierarchy/comp/lang/java/programmer/comp.lang.java_FAQ 28>

Hathaway, R.J. (1998, February 3) *Object-Orientation FAQ.* <http://www.cyberdyne-object-sys.com/oofaq2/>

Holmes, B.J. (1992) *Convert to C and C++*. DP Publications.

Holmevik, J.R. (1995, February 23) *History of Simula.* <http://www.cs.chalmers.se/ComputingScience/Education/Courses/z3sim/SIMULA-HISTORY.txt>

Interactive Software Engineering Inc. (1998) *About Eiffel.* <http://www.eiffel.com/doc/eiffel.html>

Kempe, Magnus (1997, February 5) *ADA Frequently Asked Questions.* <http://www.adahome.com/FAQ/comp-lang-ada.html#title>

Kinnersley, B. (1998, April 11) *Language List.* <http://cuiwww.unige.ch/cgi-bin/langlist?object-orient>

Klein, W.M. (1998, April 10) *COBOL Frequently Asked Questions.* <http://sunsite.doc.ic.ac.uk/usenet/usenet-by-hierarchy/comp/lang/cobol/COBOL_FAQ>

Knudsen, J. L. (1997, December 8) *Beta Frequently Asked Questions.* <http://sunsite.doc.ic.ac.uk/usenet/usenet-by-group/comp.lang.beta/FAQ:_BETA_Programming_Language_(version_1.11_-_08_Dec_97)>

Koopmans, P. S. (1996, July 17) *The Composition Filters Object Model.* <http://www.trese.cs.utwente.nl/sina/cfom/index.html>

Lafore, Robert (1991) *Object-oriented programming in C++*. Waite Group Press.

Latta, C. (1995, August 28) *Smalltalk Frequently Asked Questions.* <http://xcf.berkeley.edu/pub/misc/smalltalk/FAQ/index.html>

Lutz, M.(1996) *Programming Python.* O'Reilly & Associates.

Meyer, B.(1992) *Eiffel: The Language.* Prentice Hall.

Nierstrasz, O. (1998, February 3) *Object-Oriented Information Sources.* <http://www.iam.unibe.ch/cgi-bin/ooinfo?language>

Oberon Microsystems (1997, July 17) *A Brief History of Pascal.* <http://www.oberon.ch/docu/history.html>

Parker, S.P. (1988) *Computer Science Source Book.* McGraw Hill.

Rossum, G. (1998, April 10) *The Whole Python FAQ.* <http://grail.cnri.reston.va.us/cgi-bin/faqw.py?req=all>

Schneider, M (1998, April 5) Object-Oriented Languages: Overview. <http://www.parallax.co.uk/cetus/top_languages.html>

Schoenmakers, P.J. (1998, March 18) *Objective-C Frequently Asked Questions.* <http://sunsite.doc.ic.ac.uk/usenet/usenet-by-hierarchy/comp/lang/objective-c/comp.lang.objective-c_FAQ,_part_1_3:_Answers>

Sutherland, J. (1995, May) "C++, OO Cobol, and Smalltalk: Good, Better, Best." *Object Magazine.* 5(2). pp.32-35.

Topper, A. (1994, February) "Object-Oriented COBOL Standard." *Object Magazine.* 3(6). pp.39-41.