Lecture Notes in
Computer Science

Edited by G. Goos and J. Hartmanis

322

S. Gjessing K. Nygaard (Eds.)

ECOOP '88
European Conference on Object-Oriented Programming

Oslo, Norway, August 1988
Proceedings

Springer-Verlag

Lecture Notes in Computer Science

Edited by G. Goos and J. Hartmanis

322

S. Gjessing K. Nygaard (Eds.)

ECOOP '88
European Conference on Object-Oriented Programming

Oslo, Norway, August 15–17, 1988
Proceedings

Springer-Verlag
Berlin Heidelberg New York London Paris Tokyo

Editors

Stein Gjessing
Kristen Nygaard
Department of Informatics, University of Oslo
P.O. Box 1080 Blindern, N-0316 Oslo 3, Norway

CR Subject Classification (1988): D.2.2, D.3.3, H.2.1, F.3.3

ISBN 3-540-50053-7 Springer-Verlag Berlin Heidelberg New York
ISBN 0-387-50053-7 Springer-Verlag New York Berlin Heidelberg

Printing and binding: Druckhaus Beltz, Hemsbach/Bergstr.
2145/3140-543210

Lecture Notes in Computer Science

Vol. 219: Advances in Cryptology – EUROCRYPT '85. Proceedings, 1985. Edited by F. Pichler. IX, 281 pages. 1986.

Vol. 220: RIMS Symposia on Software Science and Engineering II. Proceedings, 1983 and 1984. Edited by E. Goto, K. Araki and T. Yuasa. XI, 323 pages. 1986.

Vol. 221: Logic Programming '85. Proceedings, 1985. Edited by E. Wada. IX, 311 pages. 1986.

Vol. 222: Advances in Petri Nets 1985. Edited by G. Rozenberg. VI, 498 pages. 1986.

Vol. 223: Structure in Complexity Theory. Proceedings, 1986. Edited by A.L. Selman. VI, 401 pages. 1986.

Vol. 224: Current Trends in Concurrency. Overviews and Tutorials. Edited by J.W. de Bakker, W.-P. de Roever and G. Rozenberg. XII, 716 pages. 1986.

Vol. 225: Third International Conference on Logic Programming. Proceedings, 1986. Edited by E. Shapiro. IX, 720 pages. 1986.

Vol. 226: Automata, Languages and Programming. Proceedings, 1986. Edited by L. Kott. IX, 474 pages. 1986.

Vol. 227: VLSI Algorithms and Architectures – AWOC 86. Proceedings, 1986. Edited by F. Makedon, K. Mehlhorn, T. Papatheodorou and P. Spirakis. VIII, 328 pages. 1986.

Vol. 228: Applied Algebra, Algorithmics and Error-Correcting Codes. AAECC-2. Proceedings, 1984. Edited by A. Poli. VI, 265 pages. 1986.

Vol. 229: Algebraic Algorithms and Error-Correcting Codes. AAECC-3. Proceedings, 1985. Edited by J. Calmet. VII, 416 pages. 1986.

Vol. 230: 8th International Conference on Automated Deduction. Proceedings, 1986. Edited by J.H. Siekmann. X, 708 pages. 1986.

Vol. 231: R. Hausser, NEWCAT: Parsing Natural Language Using Left-Associative Grammar. II, 540 pages. 1986.

Vol. 232: Fundamentals of Artificial Intelligence. Edited by W. Bibel and Ph. Jorrand. VII, 313 pages. 1986.

Vol. 233: Mathematical Foundations of Computer Science 1986. Proceedings, 1986. Edited by J. Gruska, B. Rovan and J. Wiedermann. IX, 650 pages. 1986.

Vol. 234: Concepts in User Interfaces: A Reference Model for Command and Response Languages. By Members of IFIP Working Group 2.7. Edited by D. Beech. X, 116 pages. 1986.

Vol. 235: Accurate Scientific Computations. Proceedings, 1985. Edited by W.L. Miranker and R.A. Toupin. XIII, 205 pages. 1986.

Vol. 236: TEX for Scientific Documentation. Proceedings, 1986. Edited by J. Désarménien. VI, 204 pages. 1986.

Vol. 237: CONPAR 86. Proceedings, 1986. Edited by W. Händler, D. Haupt, R. Jeltsch, W. Juling and O. Lange. X, 418 pages. 1986.

Vol. 238: L. Naish, Negation and Control in Prolog. IX, 119 pages. 1986.

Vol. 239: Mathematical Foundations of Programming Semantics. Proceedings, 1985. Edited by A. Melton. VI, 395 pages. 1986.

Vol. 240: Category Theory and Computer Programming. Proceedings, 1985. Edited by D. Pitt, S. Abramsky, A. Poigné and D. Rydeheard. VII, 519 pages. 1986.

Vol. 241: Foundations of Software Technology and Theoretical Computer Science. Proceedings, 1986. Edited by K.V. Nori. XII, 519 pages. 1986.

Vol. 242: Combinators and Functional Programming Languages. Proceedings, 1985. Edited by G. Cousineau, P.-L. Curien and B. Robinet. V, 208 pages. 1986.

Vol. 243: ICDT '86. Proceedings, 1986. Edited by G. Ausiello and P. Atzeni. VI, 444 pages. 1986.

Vol. 244: Advanced Programming Environments. Proceedings, 1986. Edited by R. Conradi, T.M. Didriksen and D.H. Wanvik. VII, 604 pages. 1986

Vol. 245: H.F. de Groote, Lectures on the Complexity of Bilinear Problems. V, 135 pages. 1987.

Vol. 246: Graph-Theoretic Concepts in Computer Science. Proceedings, 1986. Edited by G. Tinhofer and G. Schmidt. VII, 307 pages. 1987.

Vol. 247: STACS 87. Proceedings, 1987. Edited by F.J. Brandenburg, G. Vidal-Naquet and M. Wirsing. X, 484 pages. 1987.

Vol. 248: Networking in Open Systems. Proceedings, 1986. Edited by G. Müller and R.P. Blanc. VI, 441 pages. 1987.

Vol. 249: TAPSOFT '87. Volume 1. Proceedings, 1987. Edited by H. Ehrig, R. Kowalski, G. Levi and U. Montanari. XIV, 289 pages. 1987.

Vol. 250: TAPSOFT '87. Volume 2. Proceedings, 1987. Edited by H. Ehrig, R. Kowalski, G. Levi and U. Montanari. XIV, 336 pages. 1987.

Vol. 251: V. Akman, Unobstructed Shortest Paths in Polyhedral Environments. VII, 103 pages. 1987.

Vol. 252: VDM '87. VDM – A Formal Method at Work. Proceedings, 1987. Edited by D. Bjørner, C.B. Jones, M. Mac an Airchinnigh and E.J. Neuhold. IX, 422 pages. 1987.

Vol. 253: J.D. Becker, I. Eisele (Eds.), WOPPLOT 86. Parallel Processing: Logic, Organization, and Technology. Proceedings, 1986. V, 226 pages. 1987.

Vol. 254: Petri Nets: Central Models and Their Properties. Advances in Petri Nets 1986, Part I. Proceedings, 1986. Edited by W. Brauer, W. Reisig and G. Rozenberg. X, 480 pages. 1987.

Vol. 255: Petri Nets: Applications and Relationships to Other Models of Concurrency. Advances in Petri Nets 1986, Part II. Proceedings, 1986. Edited by W. Brauer, W. Reisig and G. Rozenberg. X, 516 pages. 1987.

Vol. 256: Rewriting Techniques and Applications. Proceedings, 1987. Edited by P. Lescanne. VI, 285 pages. 1987.

Vol. 257: Database Machine Performance: Modeling Methodologies and Evaluation Strategies. Edited by F. Cesarini and S. Salza. X, 250 pages. 1987.

Vol. 258: PARLE, Parallel Architectures and Languages Europe. Volume I. Proceedings, 1987. Edited by J.W. de Bakker, A.J. Nijman and P.C. Treleaven. XII, 480 pages. 1987.

Vol. 259: PARLE, Parallel Architectures and Languages Europe. Volume II. Proceedings, 1987. Edited by J.W. de Bakker, A.J. Nijman and P.C. Treleaven. XII, 464 pages. 1987.

Vol. 260: D.C. Luckham, F.W. von Henke, B. Krieg-Brückner, O. Owe, ANNA, A Language for Annotating Ada Programs. V, 143 pages. 1987.

Vol. 261: J. Ch. Freytag, Translating Relational Queries into Iterative Programs. XI, 131 pages. 1987.

Vol. 262: A. Burns, A.M. Lister, A.J. Wellings, A Review of Ada Tasking. VIII, 141 pages. 1987.

Vol. 263: A.M. Odlyzko (Ed.), Advances in Cryptology – CRYPTO '86. Proceedings. XI, 489 pages. 1987.

Vol. 264: E. Wada (Ed.), Logic Programming '86. Proceedings, 1986. VI, 179 pages. 1987.

Vol. 265: K.P. Jantke (Ed.), Analogical and Inductive Inference. Proceedings, 1986. VI, 227 pages. 1987.

Vol. 266: G. Rozenberg (Ed.), Advances in Petri Nets 1987. VI, 451 pages. 1987.

Vol. 267: Th. Ottmann (Ed.), Automata, Languages and Programming. Proceedings, 1987. X, 565 pages. 1987.

Vol. 268: P.M. Pardalos, J.B. Rosen, Constrained Global Optimization: Algorithms and Applications. VII, 143 pages. 1987.

Vol. 269: A. Albrecht, H. Jung, K. Mehlhorn (Eds.), Parallel Algorithms and Architectures. Proceedings, 1987. Approx. 205 pages. 1987.

Preface

"..... *object oriented* seems to be becoming in the 1980s what *structured programming* was in the 1970s."

Brian Randell and Pete Lee

This quotation is from the invitation to the annual Newcastle University Conference on Main Trends in Computing, September 1988. It seems to capture the situation quite well, only that the object orientation is being materialised in languages and language constructs, as well as in the style of programming and as a perspective upon the task considered.

The second European Conference on Object Oriented Programming (ECOOP'88) was held in Oslo, Norway, August 15-17, 1988, in the city where object oriented programming was born more than 20 years ago, when the Simula language appeared. The objectives of ECOOP'88 were to present the best international work in the field of object oriented programming to interested participants from industry and academia, and to be a forum for the exchange of ideas and the growth of professional relationships.

The richness of the field was evidenced before the conference: the conference had 22 slots for papers, and the Program Committee was faced with the very difficult task of selecting these papers from 103 submissions. Many good papers had to be rejected. We believe that the papers presented at the conference and in these proceedings contain a representative sample of the best work in object oriented programming today. When the names of the authors were made known to the Program Committee, it turned out that it had selected the 22 papers from 13 different countries, another indication of the widespread interest in object oriented programming.

Without the help of the Norwegian Computer Society in general, and Kersti Larsen in particular, ECOOP'88 would not have been possible. We would also like to thank the Program Committee members and all the other referees. Special thanks go to Birger Møller-Pedersen, who took on many additional duties during the Program Chairman's stay abroad.

Oslo, May 1988

Stein Gjessing

Kristen Nygaard

Conference Chairman

Stein Gjessing, University of Oslo, Norway

Program Chairman

Kristen Nygaard, University of Oslo, Norway

Program Committee Members

G. Attardi, DELPHI SpA, Italy
J. Bezivin, Lib UBO/ENSTRBr. France
A. Borning, University of Washington, Seattle, USA
P. Cointe, Rank Xerox, France
S. Cook, University of London, UK
C. Hewitt, MIT, USA
S. Krogdahl, University of Oslo, Norway
B. Magnusson, University of Lund, Sweden
J. Meseguer, SRI International, USA
B. Møller-Pedersen, Norwegian Computing Center, Norway
J. Vaucher, University of Montreal, Canada
A. Yonezawa, Tokyo Institute of Technology, Japan

Exhibition Chairman

Oddvar Hesjedal, Enator A/S, Norway

Organization

DND, The Norwegian Computer Society

Vol. 270: E. Börger (Ed.), Computation Theory and Logic. IX, 442 pages. 1987.

Vol. 271: D. Snyers, A. Thayse, From Logic Design to Logic Programming. IV, 125 pages. 1987.

Vol. 272: P. Treleaven, M. Vanneschi (Eds.), Future Parallel Computers. Proceedings, 1986. V, 492 pages. 1987.

Vol. 273: J.S. Royer, A Connotational Theory of Program Structure. V, 186 pages. 1987.

Vol. 274: G. Kahn (Ed.), Functional Programming Languages and Computer Architecture. Proceedings. VI, 470 pages. 1987.

Vol. 275: A.N. Habermann, U. Montanari (Eds.), System Development and Ada. Proceedings, 1986. V, 305 pages. 1987.

Vol. 276: J. Bézivin, J.-M. Hullot, P. Cointe, H. Lieberman (Eds.), ECOOP '87. European Conference on Object-Oriented Programming. Proceedings. VI, 273 pages. 1987.

Vol. 277: B. Benninghofen, S. Kemmerich, M.M. Richter, Systems of Reductions. X, 265 pages. 1987.

Vol. 278: L. Budach, R.G. Bukharajev, O.B. Lupanov (Eds.), Fundamentals of Computation Theory. Proceedings, 1987. XIV, 505 pages. 1987.

Vol. 279: J.H. Fasel, R.M. Keller (Eds.), Graph Reduction. Proceedings, 1986. XVI, 450 pages. 1987.

Vol. 280: M. Venturini Zilli (Ed.), Mathematical Models for the Semantics of Parallelism. Proceedings, 1986. V, 231 pages. 1987.

Vol. 281: A. Kelemenová, J. Kelemen (Eds.), Trends, Techniques, and Problems in Theoretical Computer Science. Proceedings, 1986. VI, 213 pages. 1987.

Vol. 282: P. Gorny, M.J. Tauber (Eds.), Visualization in Programming. Proceedings, 1986. VII, 210 pages. 1987.

Vol. 283: D.H. Pitt, A. Poigné, D.E. Rydeheard (Eds.), Category Theory and Computer Science. Proceedings, 1987. V, 300 pages. 1987.

Vol. 284: A. Kündig, R.E. Bührer, J. Dähler (Eds.), Embedded Systems. Proceedings, 1986. V, 207 pages. 1987.

Vol. 285: C. Delgado Kloos, Semantics of Digital Circuits. IX, 124 pages. 1987.

Vol. 286: B. Bouchon, R.R. Yager (Eds.), Uncertainty in Knowledge-Based Systems. Proceedings, 1986. VII, 405 pages. 1987.

Vol. 287: K.V. Nori (Ed.), Foundations of Software Technology and Theoretical Computer Science. Proceedings, 1987. IX, 540 pages. 1987.

Vol. 288: A. Blikle, MetaSoft Primer. XIII, 140 pages. 1987.

Vol. 289: H.K. Nichols, D. Simpson (Eds.), ESEC '87. 1st European Software Engineering Conference. Proceedings, 1987. XII, 404 pages. 1987.

Vol. 290: T.X. Bui, Co-oP A Group Decision Support System for Cooperative Multiple Criteria Group Decision Making. XIII, 250 pages. 1987.

Vol. 291: H. Ehrig, M. Nagl, G. Rozenberg, A. Rosenfeld (Eds.), Graph-Grammars and Their Application to Computer Science. VIII, 609 pages. 1987.

Vol. 292: The Munich Project CIP. Volume II: The Program Transformation System CIP-S. By the CIP System Group. VIII, 522 pages. 1987.

Vol. 293: C. Pomerance (Ed.), Advances in Cryptology – CRYPTO '87. Proceedings. X, 463 pages. 1988.

Vol. 294: R. Cori, M. Wirsing (Eds.), STACS 88. Proceedings, 1988. IX, 404 pages. 1988.

Vol. 295: R. Dierstein, D. Müller-Wichards, H.-M. Wacker (Eds.), Parallel Computing in Science and Engineering. Proceedings, 1987. V, 185 pages. 1988.

Vol. 296: R. Janßen (Ed.), Trends in Computer Algebra. Proceedings, 1987. V, 197 pages. 1988.

Vol. 297: E.N. Houstis, T.S. Papatheodorou, C.D. Polychronopoulos (Eds.), Supercomputing. Proceedings, 1987. X, 1093 pages. 1988.

Vol. 298: M. Main, A. Melton, M. Mislove, D. Schmidt (Eds.), Mathematical Foundations of Programming Language Semantics. Proceedings, 1987. VIII, 637 pages. 1988.

Vol. 299: M. Dauchet, M. Nivat (Eds.), CAAP '88. Proceedings, 1988. VI, 304 pages. 1988.

Vol. 300: H. Ganzinger (Ed.), ESOP '88. Proceedings, 1988. VI, 381 pages. 1988.

Vol. 301: J. Kittler (Ed.), Pattern Recognition. Proceedings, 1988. VII, 668 pages. 1988.

Vol. 302: D.M. Yellin, Attribute Grammar Inversion and Source-to-source Translation. VIII, 176 pages. 1988.

Vol. 303: J.W. Schmidt, S. Ceri, M. Missikoff (Eds.), Advances in Database Technology – EDBT '88. X, 620 pages. 1988.

Vol. 304: W.L. Price, D. Chaum (Eds.), Advances in Cryptology – EUROCRYPT '87. Proceedings, 1987. VII, 314 pages. 1988.

Vol. 305: J. Biskup, J. Demetrovics, J. Paredaens, B. Thalheim (Eds.), MFDBS 87. Proceedings, 1987. V, 247 pages. 1988.

Vol. 306: M. Boscarol, L. Carlucci Aiello, G. Levi (Eds.), Foundations of Logic and Functional Programming. Proceedings, 1986. V, 218 pages. 1988.

Vol. 307: Th. Beth, M. Clausen (Eds.), Applicable Algebra, Error-Correcting Codes, Combinatorics and Computer Algebra. Proceedings, 1986. VI, 215 pages. 1988.

Vol. 308: S. Kaplan, J.-P. Jouannaud (Eds.), Conditional Term Rewriting Systems. Proceedings, 1987. VI, 278 pages. 1988.

Vol. 309: J. Nehmer (Ed.), Experiences with Distributed Systems. Proceedings, 1987. VI, 292 pages. 1988.

Vol. 310: E. Lusk, R. Overbeek (Eds.), 9th International Conference on Automated Deduction. Proceedings, 1988. X, 775 pages. 1988.

Vol. 311: G. Cohen, P. Godlewski (Eds.), Coding Theory and Applications 1986. Proceedings, 1986. XIV, 196 pages. 1988.

Vol. 312: J. van Leeuwen (Ed.), Distributed Algorithms 1987. Proceedings, 1987. VII, 430 pages. 1988.

Vol. 313: B. Bouchon, L. Saitta, R.R. Yager (Eds.), Uncertainty and Intelligent Systems. IPMU '88. Proceedings, 1988. VIII, 408 pages. 1988.

Vol. 314: H. Göttler, H.J. Schneider (Eds.), Graph-Theoretic Concepts in Computer Science. Proceedings, 1987. VI, 254 pages. 1988.

Vol. 315: K. Furukawa, H. Tanaka, T. Fujisaki (Eds.), Logic Programming '87. Proceedings, 1987. VI, 327 pages. 1988.

Vol. 316: C. Choffrut (Ed.), Automata Networks. Proceedings, 1986. VII, 125 pages. 1988.

Vol. 317: T. Lepistö, A. Salomaa (Eds.), Automata, Languages and Programming. Proceedings, 1988. XI, 741 pages. 1988.

Vol. 318: R. Karlsson, A. Lingas (Eds.), SWAT 88. Proceedings, 1988. VI, 262 pages. 1988.

Vol. 319: J.H. Reif (Ed.), VLSI Algorithms and Architectures – AWOC 88. Proceedings, 1988. X, 476 pages. 1988.

Vol. 320: A. Blaser (Ed)., Natural Language at the Computer. Proceedings, 1988. III, 176 pages. 1988.

Vol. 322: S. Gjessing, K. Nygaard (Eds.), ECOOP '88. European Conference on Object-Oriented Programming. Proceedings, 1988. VI, 410 pages. 1988.

Table of Contents

What object-oriented programming may be - and what it does not have to be

Ole Lehrmann Madsen,
Dept. of Computer Science, Aarhus University,
Ny Munkegade, DK-8000 Aarhus C, Denmark
email: olm@daimi.dk

Birger Møller-Pedersen
Norwegian Computing Center,
P.O.Box 114 Blindern, N-0314 Oslo 3, Norway
ean: birger@vax.nr.uninett

Abstract

A conceptual framework for object-oriented programming is presented. The framework is independent of specific programming language constructs. It is illustrated how this framework is reflected in an object-oriented language and the language mechanisms are compared with the corresponding elements of other object-oriented languages. Main issues of object-oriented programming are considered on the basis of the framework presented here.

1. Introduction

Even though object-oriented programming has a long history, with roots going back to SIMULA [SIMULA67] and with substantial contributions like Smalltalk [Smalltalk] and Flavors [Flavors], the field is still characterized by experiments, and there is no generally accepted definition of object-oriented programming.

Many properties are, however, associated with object-oriented programming, like: "Everything is an object with methods and all activity is expressed by message passing and method invocation", "Inheritance is the main structuring mechanism", "Object-oriented programming is inefficient, because so many (small) objects are generated and have to be removed by a garbage collector" and "Object-oriented programming is only for prototype programming, as late name binding and run-time checking of parameters to methods give slow and unreliable ("message not understood") systems".

There are as many definitions of object-oriented programming as there are papers and books on the topic. This paper is no exception. It contributes with yet another definition. According to this definition there is more to object-oriented programming than message passing and inheritance, just as there is more to structured programming than avoiding gotos.

While other programming perspectives are based on some mathematical theory or model, object-oriented programming is often defined by specific programming language constructs. Object-oriented programming

Part of this work has been supported by NTNF, The Royal Norwegian Council for Scientific and Industrial Research, grant no. ED 0223.16641 (the Scala project) and by the Danish Natural Science Research Council, FTU Grant No. 5.17.5.1.25.

is lacking a profound theoretical understanding. The purpose of this paper is to go beyond language mechanisms and contribute with a conceptual framework for object-oriented programming. Other important contributions to such a framework may be found in [Stefik & Bobrow 84], [Booch 84], [Knudsen & Thomsen 85], [Shriver & Wegner 87], [ECOOP 87] and [OOPSLA 87,88].

2. A conceptual framework for object-oriented programming

Background

The following definition of object-oriented programming is a result of the BETA Project and has formed the basis for the design of the object-oriented programming language BETA, [BETA87a].

Many object-oriented languages have inherited from SIMULA, either directly or indirectly via Smalltalk. Most of these languages represent a line of development characterized by everything being objects, and all activities being expressed by message passing. The definition and language is build directly on the philosophy behind SIMULA, but represent another line of development.

SIMULA was developed in order to describe complex systems consisting of objects, which in addition to being characterized by local operations also had their own individual sequences of actions. In SIMULA, this lead to objects that may execute their actions as coroutines. This has disappeared in the Smalltalk line of development, while the line of development described here, has maintained this aspect of objects. While SIMULA simulates concurrency by coroutines, the model presented here incorporates real concurrency and a generalization of coroutines.

The following description of a conceptual framework for object-oriented programming is an introduction to the basic principles underlying the design of BETA. In section 3 the framework is illustrated by means of the BETA language. It is not attempted to give complete and detailed description of the basic principles nor to give a tutorial on BETA. Readers are referred to [DELTA 75], [Nygaard86] and [BETA 87a] for further reading. This paper addresses the fundamental issues behind BETA, but it has also a practical side. The Mjølner [Mjølner] and Scala projects have produced a industrial prototype implementation of a BETA system, including compiler and support tools on SUN and Macintosh.

Short definition of object-oriented programming

In order to contrast the definition of object-oriented programming, we will briefly characterize some of the well-known perspectives on programming.

> *Procedural programming.* A program execution is regarded as a (partially ordered) sequence of procedure calls manipulating data structures.

This is the most common perspective on programming, and is supported by languages like Algol, Pascal, C and Ada.

> *Functional programming.* A program is regarded as a mathematical function, describing a relation between input and output.

The most prominent property of this perspective is that there are no variables and no notion of state. Lisp is an example of a language with excellent support for functional programming.

> *Constraint-oriented (logic) programming.* A program is regarded as a set of equations describing relations between input and output.

This perspective is supported by e.g. Prolog.

The definitions above have the property that they can be understood by other than computer scientists. A definition of object-oriented programming should have the same property. We have arrived at the following definition:

Object-oriented programming. A program execution is regarded as a *physical model*, simulating the behavior of either a real or imaginary part of the world.

The notion of physical model shall be taken literally. Most people can imagine the construction of physical models by means of e.g. LEGO bricks. In the same way a program execution may be viewed as a physical model. Other perspectives on programming are made precise by some underlying model defining equations, relations, predicates, etc.. For object-oriented programming the notion of physical models have to elaborated.

Introduction to physical models

Physical models are based upon a conception of the reality in terms of *phenomena* and *concepts*, and as it will appear below, physical models will have elements that directly reflects phenomena and concepts.

Consider accounting systems and flight reservation systems as examples of parts of reality. The first step in making a physical model is to identify the relevant and interesting phenomena and concepts. In accounting systems there will be phenomena like invoices, while in flight reservation systems there will be phenomena like flights and reservations. In a model of an accounting system there will be elements that model specific invoices, and in a model of a flight reservation system there will be elements modelling specific flights and specific reservations.

The flight SK451 may belong to the general concept of flight. A specific reservation may belong to the general concept of reservation. These concepts will also be reflected in the physical models.

2·20
JIM DAVIS
I GOTTA GET OFF THIS DIET!
© 1981 United Feature Syndicate, Inc

Figure 1. In the object-oriented perspective *physical* models of part of the real world are made by choosing which phenomena are relevant and which properties these phenomena have.

In order to make models based on the conception of the reality in terms of phenomena and concepts, we have to identify which aspects of these are relevant and which aspects are necessary and sufficient. This depends upon which class of physical models we want to make. The physical models, we are interested in, are models of those part of reality we want to regard as information processes.

Aspects of phenomena

In information processes three aspects of phenomena have been identified: *substance*, *measurable properties* of substance and *transformations* on substance. These are general terms, and they may seem strange at a glance. In order to provide a feeling for what they capture, we have found a phenomenon (Garfield) from everyday life and illustrated the three aspects by figures 2- 4.

Substance is physical matter, characterized by a volume and a position in time and space. Examples of substances are specific persons, specific flights and specific computers. From the field of (programming) languages, variables, records and instances of various kinds are examples of substance.

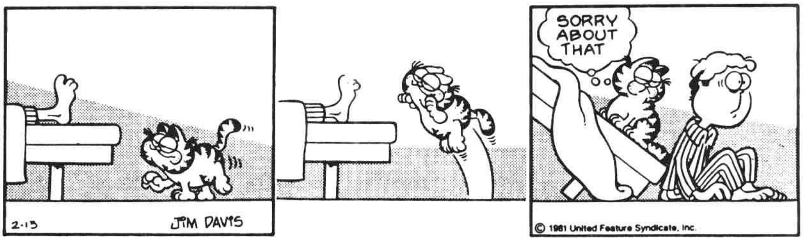


Figure 2. An aspect of phenomena is substance. Substance is characterized by a volume and a position in time and space.

Substance may have *measurable properties* . Measurements may be compared and they may be described by types and values. Examples of measurable properties are a person´s weight, and the actual flying-time of a flight. The value of a variable is also an example of the result of the measurement of a measurable property.

LET'S MEASURE YOUR TUMMY, GARFIELD
JIM DAVIS
9-19
WHEN YOU TAKE FOUR INCHES OFF YOUR WAIST, YOU MAY GO OFF YOUR DIET
© 1984 United Feature Syndicate, Inc.
AND THAT DOESN'T COUNT

Figure 3. Substance has measurable properties

A *transformation on substance* is a partial ordered sequence of events that changes its measurable properties. Examples are eating (that will change the weight of a person) and pushing a button (changing the state of a vending machine). Reserving a seat on a flight is a transformation on (the property reserved of) a seat from being free to being reserved. Assignment is an example of a transformation of a variable.

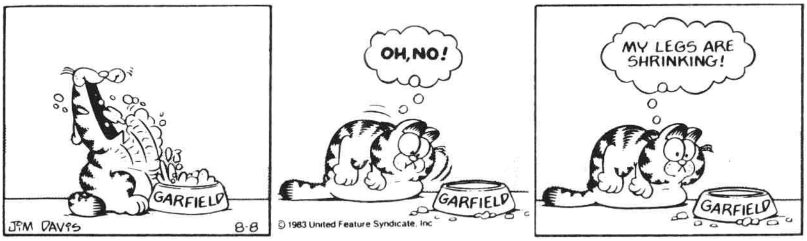


Figure 4. Actions may change the measurable properties of substance

Aspects of concepts

Substance, measurable properties and transformations have been identified as the relevant aspects of phenomena in information processes. In order to capture the essential properties of phenomena being modelled it is necessary to develop abstractions or concepts.

The classical notion of a concept has the following elements: *name*: denoting the concept, *intension*: the properties characterizing the phenomena covered by the concept, and *extension*: the phenomena covered by the concept.

Concepts are created by *abstraction*, focussing on similar properties of phenomena and discarding differences. Three well-known sub-functions of abstraction have been identified. The most basic of these is *classification*. Classification is used to define which phenomena are covered by the concept. The reverse sub-function of classification is called *exemplification*.

Concepts are often defined by means of other concepts. The concept of a flight may be formed by using concepts like seat, flight identification, etc. This sub-function is called *aggregation*. The reverse sub-function is called *decomposition*.

Concepts may be organized in a *classification hierarchy*. A concept can be regarded as a *generalization* of a set of concepts. A well-known example from zoology is the taxonomy of animals: mammal is a generalization of predator and rodent, and predator is a generalization of lion and tiger. In addition, concepts may be regarded as *specializations* of other concepts. For example, predator is a specialization of mammal.

Elements of physical models: Modelling by objects with attributes and actions

Objects with properties and actions, and patterns of objects

Up till now we have only identified in which way the reality is viewed when a physical model is constructed, and which aspects of phenomena and concepts are essential. The next step is to define what a physical model itself consists of.

A physical model consists of *objects,* each object characterized by *attributes* and a sequence of *actions*. Objects organize the substance aspect of phenomena, and transformations on substance are reflected by objects executing actions. Objects may have part-objects. An attribute may be a reference to a part object or to a separate object. Some attributes represent measurable properties of the object. The *state* of an object at a given moment is expressed by its substance, its measurable properties and the action going on then. The state of the whole model is the states of the objects in the model.

In a physical model the elements reflecting concepts are called *patterns*. A pattern defines the common properties of a category of objects. Patterns may be organized in a classification hierarchy. Patterns may be attributes of objects.

Notice that a pattern is not a set, but an abstraction over objects. An implication of this is that patterns do not contribute to the state of the physical model. Patterns may be abstractions of substance, measurable properties and action sequences.

Consider the construction of a flight reservation system as a physical model. It will a.o. have objects representing flights, agents and reservations. A flight object will have a part object for each of the seats, while e.g. actual flying time will be a measurable property. When agents reserve seats they will get a display of the flight. The seat objects will be displayed, so that a seat may be selected and reserved. An action will thus change the state of a seat object from being free to become reserved.

Reservations will be represented by objects with properties that identifies the customer, the date of reservation and the flight/seat identification. The customer may simply be represented by name and address, while the flight/seat identification will be a reference to the separate flight object/seat object. If the agency have a customer database, the customer identification could also be a reference to a customer object.

As there will be several specific flights, a pattern Flight defining the properties of flight objects will be constructed. Each flight will be represented by an object generated according to the pattern Flight.

A travel agency will normally handle reservations of several kinds. A train trip reservation will also identify the customer and the date of reservation, but the seat reservation will differ from a flight reservation, as it will consists of (wagon, seat). A natural classification hierarchy will identify Reservation as a general reservation, with customer identification and date, and Flight Reservation and Train Reservation as two specializations of this.

Actions in a physical model

Many real world systems are characterized by consisting of objects that perform their sequences of actions *concurrently*. The flight reservation system will consist of several concurrent objects, e.g. flights and agents. Each agent performs its task concurrently with other agents. Flights will register the reservation of seats and ensure that no seats are reserved by two agents at the same time. Note that this kind of concurrency is an inherent property of the reality being modelled; it is not concurrency used in order to speed up computations.

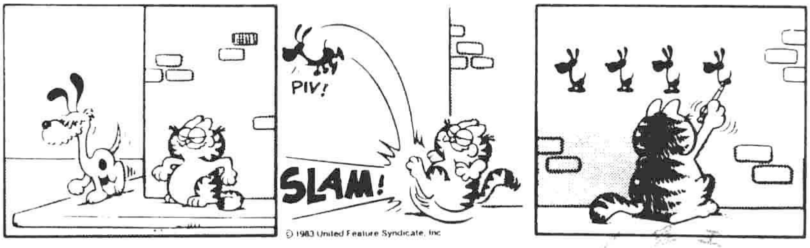
Complex tasks, as those of the agents, are often considered to consist of several more or less independent activities. This is so even though they constitute only one sequence of actions and do not include concurrency . As an example consider the activities "tour planning", "customer service" and "invoicing". Each of these activities will consist of a sequence of actions.

A single agent will not have concurrent activities, but *alternate* between the different activities. The shifts will not only be determined by the agents themselves, but will be triggered by e.g. communication with other objects. An agent will e.g. shift from tour planning to customer service (by the telephone ringing), and resume the tour planning when the customer service is performed.

The action sequence of an agent may often be decomposed into *partial action sequences* that correspond to certain routines carried out several times as part of an activity. As an example, the invoicing activity may contain partial action sequences, each for writing a single invoice.

Actions in a physical model are performed by objects. The action sequence of an object may be executed *concurrently* with other action sequences, *alternating* (that is at most one at a time) with other action sequences, or as *part* of the action sequence of another object.

The definition of physical model given here is valid in general and not only for programming. A physical model of a railroad station may consist of objects like model train wagons, model locomotives, tracks, points and control posts. Some of the objects will perform actions: the locomotives will have an engine and the control posts may perform actions that imply e.g. shunting. Patterns will be reflected by the fact that these objects are made so that they have the same form and the same set of attributes and actions. In the process of designing large buildings, physical models are often used.



© 1983 United Feature Syndicate, Inc

Figure 5. States are changed by objects performing actions that may involve other objects.

Object-oriented programming and language mechanisms supporting it

The notion of physical models may be applied to many fields of science and engineering. When applied to programming the implication is that the *program executions* are regarded as physical models, and we rephrase the definition of object-oriented programming:

Object-oriented programming. A program execution is regarded as a *physical model*, simulating the behavior of either a real or imaginary part of the world.

The ideal language supporting object-orientation should be able to prescribe models as defined above. Most elements of the framework presented above are represented in existing languages claimed to support object-orientation, but few cover them all. Most of them have a construct for describing objects. Constructs for describing patterns are in many languages represented in the form of classes, types, procedures/methods, and functions.

Classification is supported by most existing programming languages, by concepts like type, procedure, and class. Aggregation/ decomposition is also supported by most programming languages; a procedure may be defined by means of other procedures, and a type may be defined in terms of other types.

Language constructs for supporting generalization/specialization (often called sub-classing or inheritance) are often mentioned as the main characteristic of a programming language supporting object-orientation. It is true that inheritance was introduced in Simula and until recently inheritance was mainly associated with object-oriented programming. However, inheritance has started to appear in languages based on other perspectives as well.

Individual action sequences should be associated with objects, and concurrency should be supported. For many large applications support for persistent objects is needed.

Benefits of object-oriented programming

Physical models reflect reality in a natural way

One of the reasons that object-oriented programming has become so widely accepted and found to be convenient is that object orientation is close to the natural perception of the real world: viewed as consisting of object with properties and actions. Stein Krogdahl and Kai A. Olsen put it this way:

> "The basic philosophy underlying object-oriented programming is to make the programs as far as possible reflect that part of the reality, they are going to treat. It is then often easier to understand and get an overview of what is described in programs. The reason is that human beings from the outset are used to and trained in perception of what is going on in the real world. The closer it is possible to use this way of thinking in programming, the easier it is to write and understand programs."
>
> (translated citation from "Modulær- og objekt orientert programming", DataTid Nr.9 sept 1986).

Physical model more stable than the functionality of a system

The principle behind the Jackson System Development method (JSD, [JSD]) also reflects the object-oriented perspective described above. Instead of focussing on the functionality of a system, the first step in the development of the system according to JSD is to make a physical model of the real world with which the system is concerned. This model then forms the basis for the different functions that the system may have. Functions may later be changed, and new functions may be added without changing the underlying model.

3. A language based on this model and comparisons with other languages

The definition above is directly reflected in the programming language BETA. The following gives a description of part of the transition from framework to language mechanisms. Emphasis is put on conveying an understanding of major language mechanisms, and of differences from other languages.

Objects and patterns

The BETA language is intended to describe program executions regarded as physical models. From the previous it follows that by physical model is meant a system of interacting objects. A BETA program is consequently a description of such a system. An object in BETA is characterized by a set of attributes and a sequence of actions. Attributes portray properties of objects. The syntactic element for describing an object is called an *object descriptor* and has the following form:

```
(#
  Decl1; Decl2; ...; Decln
do
  Imp
#)
```

where $\mathtt{Decl}_1$; $\mathtt{Decl}_2$; ... $\mathtt{Decl}_n$ are declarations of the attributes and `Imp` describes the actions of the objects in terms of imperatives.