

Coursework 'Information Engineering' 1996/97  
**Object Oriented Software Development**

Michael Piefel

5 December 1996  
Department of Computing  
University of Bradford

**I The new way of thinking**

There has been much excitement about the new object-oriented paradigm. But it is rather unintelligible why nobody thought of it much earlier, for the basic principles are some 40,000 years old:

In apprehending the real world, men [people] constantly employ three methods of organization, which pervade all of their thinking:

- (1) the differentiation of experience into particular objects and their attributes — e. g., when they distinguish between a tree and its size or spatial relations to other objects,
- (2) the distinction between whole objects and their component parts — e. g., when they contrast a tree with its component branches, and

- (3) the formation of and the distinction between different classes of objects — e. g., when they form the class of all trees and the class of all stones and distinguish between them.

This is from the Encyclopdia Britannica<sup>1</sup> from 1986, a book which can hardly be accused of having a bias towards object-orientation and at a time when the term object did not have any special meaning in the minds of most computer scientists.

This shows what proponents of object-oriented programming (design, analysis, ...) are trying to say: That the 'new' approach is the most natural of all. In the past there had been the procedure-driven and the data-driven approach; so what remains? Simply, to do away with the artificial separation and put data and their immediate procedures together.

## 1.1 An object

There is still much discussion about how exactly to define many of the items in the object-oriented terminology (but see [PPP94]). When browsing the literature one might find dozens of different definitions; most books provides several (quoted from other books, so one might read the same definitions again and again). The consensus is approximately:

An object is an abstraction of something which exists in the real world or in our minds which belongs to the system we want to model and which we want to store information about.

An object could be a customer, a temperature sensor or a stack. This is of course nothing more than a simple record of data. As said above, data and procedures are meant to go together now. Therefore the above definition gets amended:

An object is an abstraction of something in the real world or our imagination comprising its attributes and the activities which can be performed by or on it.

Object

<sup>1</sup>[Bri86], citation from [CY91b].

This is by all means the key notion of object-oriented thinking. No longer will data and procedures be separated. The conception of the world is meant to be modelled as closely as possible. Objects in the real world shall be represented by abstractions in our models as a unit comprising its attributes (as [Bri86] says) and its activities. When representing a cat, we store its properties, such as the colour of its eyes and the length of the tail, and the actions, like purring and drinking milk. These should not float somewhere in the space of the program or the design, but they are associated with the cat object. After all, why should there be a purring-procedure somewhere—the cat itself knows best how to purr.

## 1.2 Classes

Classification is abstraction and people do it everywhere to master the inherent complexity of the world and the sheer number of objects in it. If two cats were two totally different entities, we would have to learn that ‘Cat 1’ likes to drink milk and purrs, and ‘Cat 2’ likes to drink milk and purrs.

Classification allows to see that ‘Cat 1’ and ‘Cat 2’ both belong to the class of all cats, therefore we know that they both like milk and purr. They share certain attributes such as the number of legs, but they are different in certain others such as their inclination to actually chase mice instead of lying lazily on the veranda.

A class represents an abstraction of a class of objects in the real world; it is the set of all objects sharing certain common properties.

Class

This of course reminds us at once of types in programming languages, where two integers have the same format but may hold different numbers. And in fact a class is a type in most programming languages.<sup>2</sup>

Once a class has been abstracted from the real world, the attributes and methods belonging to it are identified. Method is a common name for a procedure which belongs to an object, another name popular with the C++-community is member function. An attribute is some data we want

Method  
Attribute

<sup>2</sup>Others, like Smalltalk, introduce the concept of metaclasses. A class does not actually have to be a type. But this distinction is normally a rather academic one and does not apply to the majority of languages anyway.

## 2 Key concepts

to store about an object of the class, it is also called data member or, as in Pascal, field.

In mathematics the concept of classes is that given a certain equivalence relation one can divide the set of all values into disjoint classes which union will yield again the set of all values. Unfortunately it is in the real world often not feasible (or even not possible) to apply a real equivalence relation. We apply different criteria and may end up with objects belonging to more than one class, such as an electro-motor being something moving as well as something powered by electricity. It is not always clear how to identify classes and how to assign objects to the correct class.

### 1.3 Instances

An instance of a class is simply an element of the set that the class represents. Many authors (such as [Boo94, CY91b]) equate an object to an instance. Instance

This may often be sufficient, but just as a class does not need to be a type it seems that sometimes it could be useful to distinguish between object and instance. The author's view is that an object presents the thing of the real world and an instance is only the current representation of it within our application and abstraction.

The obvious case where an object and its corresponding instance cannot be considered equal is when e. g. an instance is passed to a procedure as a value parameter. There will suddenly exist two instances, but they may still denote the same object. Thus an instance is an implementation issue and may apply only to the programming stage.

## 2 Key concepts

Different authors provide a plethora of criteria which should be the key features of the object-oriented approach. For the OMG<sup>3</sup> for instance the identity of an object is of utmost importance, Object Identity

<sup>3</sup>Object Management Group

i. e. an object can always be identified by means of a key independent of its current attributes.<sup>4</sup> There are, however, some concepts which appear virtually everywhere and which are, as the author feels, so important that they mark the difference between object-oriented and non-object-oriented systems, and these will be enumerated in the following sections.

Other concepts are rather at a different level and do not really affect object-orientation only. This is for instance the whole-part or has-a relationship between two objects which can always be expressed since an object as an abstraction should be nothing other than a data item (with its operations, of course).

### 2.1 Encapsulation

In the early days programmers followed only procedural abstraction: Use the best algorithm for the problem at hand. Their programs remained however unstructured. Later procedures were gathered into modules, but often at random. The third step was data abstraction (the fourth is of course object-oriented abstraction, see [Str91]). In a refined form this is called encapsulation.

Encapsulation

The data is grouped together with the procedures working on them. Ideally, there is only an interface through which the data can be accessed comprising several procedures, but no direct access to the data. This is called data hiding. The procedures should not reveal the implementation used to manipulate the data. The whole concept is called information hiding.

Interface  
Data  
Hiding

When the data actually presents a coherent concept instead of just a conglomerate of unrelated items, the result is an abstract data type or ADT. This concept is not new to the object-oriented approach.

ADT

Many languages actively support information hiding. In Ada the package delivers exactly this notion, Modula has its modules and Pascal its units<sup>5</sup>. This is actually equivalent to an object, which led [Cox86] to the statement that Ada is an object-oriented programming language (or OOPL). This is not true for Ada 83, which is described in that book, but it is probably equally untrue even

<sup>4</sup>This is used for identifying objects which are passed using the OMG ORB or in object-oriented databases, where two objects are still distinct even when they share all attributes (see [Cat94]).

<sup>5</sup>It should be obvious here that the author is not referring to the obsolete standard Pascal which is almost useless in all its limitations.

for Ada 95 which supports all the key concepts of object-orientation but makes them awkward to use.

### 2.2 Inheritance

A plant is an organism. A tree is a plant. An oak is a tree. Hierarchies are everywhere in the world. It is possible to build hierarchies of objects.

Inheritance means that one class stands lower in a hierarchy than another, having all the attributes and methods of the class<sup>6</sup> standing higher, thereby inheriting them.

Inheritance

The class that stands higher is called the superclass or the base class, whilst the class inheriting is called subclass or derived class. The two classes stand in a is-a relationship to each other, i. e. we say the subclass is a (kind of the) superclass.

Superclass  
Subclass

This usually means that the superclass is more general (a tree), whilst the subclass is a specification of its superclass (an oak-tree). That is why this kind of relationship is sometimes called generalization-specialization (or simply gen-spec) relationship.

Inheritance can be used to map real-world hierarchies closely to the model. Often the superclasses are not one of the entities that we are looking at in the application. For instance consider a stack and a queue. These are objects of our imagination, and we would like to have classes for them. As it turns out, they have much in common: One can put an element in it and retrieve one. So it might be feasible to define a class container which represents the commonalities between the two classes and make stack and queue subclasses of the container. Since normally the container is too abstract to be used, it is called an abstract superclass and there will never be instances of it.<sup>7</sup>

Abstract

Some languages allow multiple inheritance, i. e. a class can have more than one superclass, others do not support this. It is not clear whether or not multiple inheritance is necessary or merely convenient.

<sup>6</sup>There are some, or rather very few, languages which allow to have less than all.

<sup>7</sup>This notion is supported in C++ through the use of a pure virtual function. A virtual function is made pure by initialising it with =0, and a class containing such a function cannot be instantiated. In Java there even exists the keyword **abstract**.

## 2 Key concepts

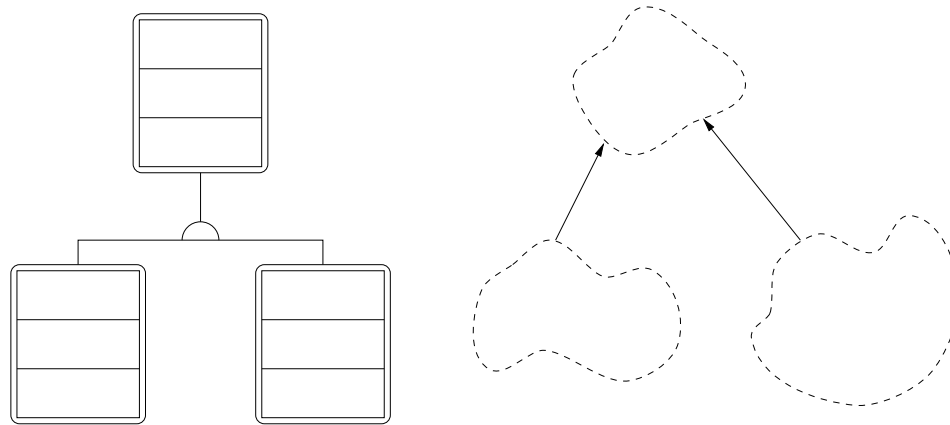


Figure 1: Inheritance notations for Coad-Yourdon and Booch methods

Figure 1 shows two out of a plethora of different notations for inheritance. Note that the arrow points towards the superclass, which is somewhat counterintuitive at first sight, but makes sense since usually a derived class knows which its base class(es) is (are), but a base class does not have knowledge about its derived classes.

### 2.3 Polymorphism

'Polymorph' is, of course, derived from the Greek  $\text{'}\pi\omicron\lambda\nu\text{'}$ , which means as a suffix 'very', 'much' or 'many', and  $\text{'}\mu\omicron\rho\phi\eta\text{'}$ , which is 'form'. When something is polymorph it can take different shapes.

Some authors do not see this a key feature, but it is very powerful and without it object-orientation cannot be fully achieved.

Polymorphism is supported when we can use one class where another is expected. This is sensible when the class which is actually provided has at least the one feature which is used in the context that the expected class did have. However, in order to avoid confusion (and actually, to improve performance of an OOPL) polymorphism is often used in conjunction with inheritance:

Polymorphism is achieved when it is possible to use a subclass of a class instead of that class itself.

Polymorphism

Consider the standard example: We have a hierarchy of geometrical objects such as squares, triangles, circles and so on. These are all subclasses of an abstract class called shape. This makes sense since they have all much in common, in particular they all can draw themselves on the screen. Given a collection of shapes the task of drawing them all can be accomplished the following way (in a C++ kind of syntax, the meaning should be obvious):

```
for (i=0; i<ShapeCollection.length; i++)
{
  if (ShapeCollection[i].type == square) drawSquare(ShapeCollection[i]);
  else if (ShapeCollection[i].type == circle) drawCircle(ShapeCollection[i]);
  else if ...
}
```

This is awkward and error prone. It has to be repeated everywhere in the program where such commonality of the object is used. When a new class is introduced, then the source code has to be changed everywhere. It is better to write:

```
for (i=0; i<ShapeCollection.length; i++)
  ShapeCollection[i].draw;
```

After all, the objects know how to draw themselves. What is done is that we use all objects which are derived from shape just as if they actually were a shape. Of course shape would have to have a 'draw' method so the compiler can check for consistency. However, which function is actually called will be decided at run time, this is known as late or dynamic binding<sup>8</sup>.

Late  
Binding

The concept of polymorphism is what makes inheritance powerful, for it makes use of the commonality expressed by the inheritance relationship.

<sup>8</sup>Since this imposes a run time overhead, however small, some languages require that the programmer specifies when to use late binding. In C++ and in Pascal we have the notion of a virtual function. This makes sense since not all methods really need this functionality, and this way the overhead does not have to apply to them. In Smalltalk, on the other hand, all methods are virtual.

### 3 Application of the paradigm

Some of the object-oriented concepts have been known for decades without being referred to as object-oriented. It is noteworthy that object-oriented methods did not just appear over night, but instead object-oriented ideas evolved slowly until they reached their current popularity.

Many products nowadays are labelled object-oriented, not all of them are, of course. In the following section the author will provide an overview of where the object-oriented paradigm is likely to be easily applicable.

#### 3.1 Programming

Here is where it all began. In the late 1960s the first OOP was developed: Simula 67. It took, however, 20 years until the ideas were put into widespread use. Smalltalk, as one of the two most important OOPs, became object-oriented in its 76 version<sup>9</sup>

An OOP supports all the key concepts enumerated in the previous section. If a language does not support them, it is not object-oriented.

The concepts allow a refinement of all the structured programming techniques known up to then. Additionally they offer possibilities like polymorphism which were formerly impossible or only very clumsy. Many languages allow the overloading of procedures and operators,<sup>10</sup> thereby allowing the definition of a class with the complete set of operations so that they can be used just like the built-in types.

Programming can significantly profit from object-oriented support. However, if the analysis and design stages produced a merely procedural abstraction, then the use of an OOP is limited; in order not to waste one's opportunities it is preferable to have an object-oriented design.

<sup>9</sup>The previous versions were only object-based, they did not support inheritance. See [[Weg87](#)].

<sup>10</sup>Overloading is to distinguish between two procedures not only by their name, but by their whole signature. That is, it is possible to give procedures which do the same thing for different types the same name.

#### 3.2 Analysis and design

Even later than object-oriented programming object-oriented analysis and design were developed. Now there exist different methods such as described in [CY91a, CY91b, Boo94] for both stages.

It is obvious that when the design does not produce an object-oriented structure then the full power of an OOPL will be wasted. It is important to identify classes at the design stage and not take them just as an implementation issue. What can be achieved is a design which is much closer to the system we have to model. It is therefore more comprehensible. That in turn means that errors are more easily spotted, and when they are removed, then they will affect only a small part of the design, since the objects exist in their own right and are only weakly coupled.

When one's aim is to produce an object-oriented design it is advantageous to view the system and the task as object-oriented from the beginning on, and analysing the system using object-oriented techniques. It should, however, be noted, that the order in which the application of the object-oriented paradigm is enumerated in this article does not only roughly correspond to the order in which activities will be commenced during development of software (although it is obviously the reverse order), but mainly reflects the level as to how well-understood this particular activity is today. Whilst object-oriented programming is accepted and widely used, analysis is often carried out in more conventional ways. It is to be hoped that this will change in the near future (however, see section 4.2).

#### 3.3 Specification

This is a quite new application. Formal methods are widely accepted as a means to produce a clear specification, in fact the only means to produce it unambiguously. They are gaining popularity, for they yield more reliable software, which is increasingly important in our computerised world.

Many of these methods, however, have a grave flaw: They are hard to understand, particularly when the projects to specify are becoming bigger, which they inevitably are. There are different approaches to address this problem. One is Methods Integration, the attempt to combine formal methods with the structured ones; this has the advantage that they will be more readily accepted by customers.

The other is to extend the current formal specification languages with object-oriented features (see [MSP94, LH94]). Here object-orientation is used to master the complexity, to break the problem into smaller units and to tackle them one after another. It seems likely that object-oriented specification languages will replace conventional ones in the near future, for they fit into the overall object-oriented lifecycle.

Of course these two approaches are not contrasting, but complementary, so we can expect to see combinations of non-formal methods (although they are unlikely to be Yourdon-DeMarco or SSADM)<sup>11</sup> and formal object-oriented methods.

#### 3.4 User interfaces

One of the things that has most often been claimed to be object-oriented is the user interface. For many products this is nothing more than a buzzword meaning only that the vendor wishes to sell more of his product. Most are not even aware of what object-orientation means. But object-oriented user interfaces can, in fact, make sense.

Consider for instance a spreadsheet. In a conventional model one can mark a cell and then select from the main menu (and possibly several sub-menus) the action which one wants to be done with it. An object-oriented user interface would allow us to mark the cell and then to open some kind of context-menu (often done, for instance, by clicking with the secondary instead of the primary mouse button)<sup>12</sup> which offers exactly the actions which are applicable for that kind of selection. A cell containing a formula might have a formula editor associated with it, a cell containing text for a headline might be centered, for a block of cells both actions might not be sensible and so only copying is allowed. This is exactly what is meant by the object-oriented model: Not only the (visible) data of a certain kind of entity, but also the procedures which are associated with it are in the same place.

<sup>11</sup>There are however, attempts to adapt older methods to new paradigms, such as object-oriented SSADM (see [RB94]).

<sup>12</sup>This has, in fact, become a semi-standard on the PC.

## 4 Advantages and problems of the object-oriented approach

### 4.1 Advantages

Many have seen object-orientation as a panacea for the software crisis<sup>13</sup>. It is however, clear that this is not the case, since, as [Bro87] points out, software is inherently complex and it is highly unlikely that increases in productivity can ever be achieved in orders of magnitude. Notwithstanding this limitation object-orientation offers progress over old techniques.

The object-oriented approach is nearer to the systems it has to model. The structure of the problem domain can be directly represented. This has the advantage of increased understanding and is therefore less error-prone.

Since the notion of object-orientation is the same at all stages of the engineering process, it is possible to use the same terminology and the same notation throughout the whole engineering process. This in turn again increases the comprehensibility of the approach.

The ultimate advantage is supposed to be reuse. Due to the strong encapsulation the object-oriented approach proposes, software reuse is facilitated.

#### 4.1.1 Reuse

Reuse means to use something again which has been used before. The simplest form is to ‘copy & paste’ code fragments from other programs. Object-orientation allows us to reuse whole classes, and, if they do not fit our needs exactly, to extend them and adapt their behaviour through inheritance. Ideally it should be possible to reuse also the results of the analysis and design stages. The next time a team has to analyse a small firm, classes such as a booking have already been identified and only have to be changed to reflect the new requirements.

Reuse is said to increase productivity dramatically. Little research has actually been carried out to affirm this seemingly obvious fact. The research that has been done (see [BBM96]) shows that productivity is in fact increased. Unfortunately the expectations of some enthusiasts have not been achieved, and it seems that Brooks has yet to be disproved.

<sup>13</sup>[Boo94] suggests, however, that a problem prevailing for so long can hardly be called a crisis, but rather the norm.

## 4.2 Problems

The main problem with object-oriented technology seems to be the transition from older methods. There might be areas where other methods are in fact more applicable, but more often the problem is simply inertia.

Even if object-oriented technology is understood, and one is willing to introduce it, software engineers still have to learn anew, and likewise managers. They face considerable problems since the methodology used is often not suitable for a strictly managed process. Often analysis, design and programming can better be carried out in parallel, and this of course means that progress is difficult to assess. The way to object-orientation may not be gone in inappropriate haste (see [FTF96]).

Another problem is that it still seems to be difficult to define some of the terminology, and different authors may have different opinions. See, however, [PPP94].

## Annotated Bibliography

[BBM96] Victor R. Basili, Lionel C. Briand, and Walcelio L. Melo. How reuse influences productivity in object-oriented systems. *Communications of the ACM*, 39(10), October 1996.

For many years one could here the opinion that reuse, no matter whether software or design results, would increase productivity. Whilst that might seem obvious, there had not yet been any proof of the fact. This article is based upon a study carried out at the University of Maryland, Baltimore, U.S.A.

Eight small groups of students were assigned the same task. Some made extensive use of existing libraries, some did not use them at all. Although the data basis is too small to draw reliable and quantitative conclusions, the study shows a clear relationship between reuse and productivity which is nearly

## Annotated Bibliography

linear. However, the increase in productivity is far below an order of magnitude, a number which many proponents of reuse like to produce.

- [Ber93] Edward V. Berard. *Essays on object-oriented software engineering*. Prentice Hall, Englewood Cliffs, New Jersey, 1993.

This book is not hard-fact science, but essays from someone who had to work and teach in different companies, partially trying to persuade them to use object-orientation. These essays include many of his experiences and are very good reading.

- [Boo94] Grady Booch. *Object-Oriented Design with Applications*. Benjamin/Cummings, Redwood City, California, 2nd edition, 1994.

This excellent book gives a very comprehensive account of how to apply object-orientation in analysis and design and does not stop short of programming (although of course it cannot give great detail here, for that is too language-specific). It explains the object-oriented paradigm and introduces a methodology dealing with its full complexity including an extensive notation that can if necessary be used to express even the more esoteric requirements.

It also discusses some general issues and, as the name suggests, contains five applications of the method which are anything but trivial. This is one main advantage of the book, other books often leave much to be asked.

- [Bri86] *Encyclopdia Britannica*. Encyclopdia Britannica, Inc., Chicago, Illinois, 1986.

- [Bro87] Fred Brooks. No silver bullet. *IEEE Computer*, 20(4), April 1987.

No article on software engineering is complete without a reference to this article in which Brooks explains that due to the inherent complexity of software it is highly unlikely that we will find a cure for the software crisis.

## Annotated Bibliography

[Cat94] R. G. G. Catell. *Object Data Management: Object-Oriented and Extended Relational Database Systems*. Addison-Wesley, Reading, Massachusetts, 1994.

[Cox86] Brad J. Cox. *Object-Oriented Programming: An Evolutionary Approach*. Addison-Wesley, Reading, Massachusetts, 1986.

Describes the object-oriented approach as a cure-for-all. It is heavily biased towards Objective-C, which turns C into an object-oriented language by adding objects and message passing. The approach owes much to Smalltalk, since all (object-)types are resolved at run-time. Basic building block is the software-IC, which is a class including its binary representation.

The book contains a chapter about Smalltalk-80, Ada and C++ (the two latter of course in their then current version) and describes their object-oriented properties. The most striking feature of the book is that Ada 83 is described as a fully object-oriented language due the possibility of strong encapsulation and operator overloading [*sic*]. (Included here to show how much confusion there was (still is?) in the early days. See [Weg87].)

[Cre92] Stuart M. Creeley. *The object oriented approach. Its concepts, a development methodology and software reuse*. M. Sc. Dissertation, University of Bradford, Department of Computing, Richmond Road, Bradford, BD7 1DP, UK, 1992.

Contains an overview of what object-orientation is plus a good overview about reuse. It describes why object-orientation facilitates reuse (size and understandability, generality, encapsulation), but fails to mention some of the inherent problems associated. Further it describes a methodology for object-oriented analysis and design which is mainly based on the Coad-Yourdon approach [CY91a, CY91b], shown by means of an example (the Bradford Hilton).

[CY91a] Peter Coad and Edward Yourdon. *Object-Oriented Analysis*. Yourdon Press Computing Series. Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

- [CY91b] Peter Coad and Edward Yourdon. *Object-Oriented Design*. Yourdon Press Computing Series. Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

This is a surprisingly thin advertisement for the authors' tools and magazines, set in a big type.

It is not clear why this booklet is not in one volume with [CY91a], nor why it is widely regarded as so important apart from its early publication date. It contains very interesting reading and generally applicable reading on software engineering in chapters one and eight, presented from an object-oriented point of view. It offers, however, no introduction to the object-oriented approach.

Its main virtue lies in a series of interesting citations from various dictionaries including the reference to the Encyclopdia Britannica at the beginning of this article.

- [FTF96] Mohamed E. Fayad, Wei-Tek Tsai, and Milton L. Fulghum. Transition to object-oriented software development. *Communications of the ACM*, 39(2), February 1996.

Many software developers are still reluctant to use object-oriented approaches. They understand them now well, but they are not aware of the problems which managing object-oriented projects can bring. This article is based upon the experiences of the authors in guiding different companies to the new technique.

- [LH94] Kevin Lano and Howard Haughton, editors. *Object-oriented specification case studies*. Prentice-Hall, Englewood Cliffs, New Jersey, 1994.

Object-orientation is now widely regarded as a way to master complexity; formal methods are the only feasible way to avoid the ambiguities of human language. Combining these two brings their advantages together. The book offers an introduction into this young field (it assumes, however, a understanding of both approaches). It includes a description of several object-oriented formal

languages (Object-Z, VDM++, OOZE, MooZ, Fresco, Z++ and Small VDM) and a comparison between them as well as accompanying case studies applying them.

- [MSP94] Angelo Morzenti and Pierluigi San Pietro. Logical specification of time-critical systems. *ACM Transactions on Software Engineering and Methodology*, 3(1), January 1994.

In this article the language TRIO<sup>+</sup> is described. It is a formal specification language for real-time systems. It is based upon TRIO. This language is applicable to all problems, however for large projects it lacks structuring techniques and a graphical notation. The authors decided to extend TRIO with object-oriented features.

- [Nau96] Patrick Naughton. *The Java Handbook*. McGraw-Hill, Berkeley, California, 1996.

This book focuses on the Java programming languages and explains object-oriented issues only in a shallow way. It is, however, the only one which enumerates reasons why C++ might not be an ideal object-oriented languages (whilst Java is, of course). The reasoning is that C++ does not enforce object-oriented programming, since it is possible to define global data and functions.

- [PPP94] Francesco Parisi-Presicce and Alfonso Pierantonio. An algebraic theory of class specification. *ACM Transactions on Software Engineering and Methodology*, 3(1), January 1994.

Though object-orientation is now very popular and everybody uses it and seems to know exactly what they are doing, there is still much confusion about even the important key-concepts such as inheritance and what exactly a class is. This article tries to define these terms using a rigorous mathematical approach utilising algebraic notations. Even if different schools of thought might not be completely satisfied with exactly *how* some concepts are defined, the authors are showing a way in which it will be possible to find an agreement.

## Annotated Bibliography

- [RB94] Keith Robinson and Graham Berrisford. *Object-oriented SSADM*. Prentice-Hall, Englewood Cliffs, New Jersey, 1994.
- [Str91] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, Massachusetts, 2nd edition, 1991.

This book is a detailed description of the C++ programming language by its creator. It includes a reference manual and some general chapters and building libraries and object-oriented design. It concentrates on how to produce clean software and how not to fall back to old habits.

The general chapters present of course the world from the point of view of a C++ programmer, which is not bad, since often the tool we use shapes the way we think. It is therefore an invaluable book for everyone who is going to program in C++ and even other ‘hybrid’ languages, which tend to have books less than suitable from an object-oriented point of view.

- [Weg87] Peter Wegner. Dimensions of object-based language design. *ACM SIGPLAN Notices*, 22(12):168–182, December 1987. Also in proceedings of OOPSLA87.

This article describes several aspects of how to rate languages with respect to object-orientation. It introduces the terms ‘object-based’, ‘class-based’ and ‘object-oriented’; the first and third of which are still in wide use to rate object-oriented (or less object-oriented) languages.

- [You94] Edward Yourdon. *Object-Oriented Systems Design: An Integrated Approach*. Yourdon Press Computing Series. Prentice Hall, Englewood Cliffs, New Jersey, 1994.

This book has some things in common with [CY91b], some passages seem to be copied verbatim. It is therefore better not to have read the latter. However, this book has a different orientation and is much more comprehensive.

It provides an overview of the object-oriented way of thinking, covering all but specific programming topics. It includes a fairly intelligible introduction

### *Annotated Bibliography*

to the topic as well as chapters on analysis, design, management of object-oriented projects and CASE tools. It presents different viewpoints (which is why its subtitle is 'integrated approach') through the use of a plethora of references (though the book does this with footnotes and lacks a proper bibliography, which reduces its value) and summarizes them, giving a very good impression of what the term object-oriented means.

## **Index**

ADT, 5

Attribute, 3

Class, 3

Data Hiding, 5

Encapsulation, 5

Inheritance, 5

Instance, 3

Interface, 5

Method, 3

Object, 2

Object Identity, 4

Subclass, 5

Superclass, 5