Quotes from*
Object-Oriented Software Construction

Bertrand Meyer

Prentice-Hall, 1988

Preface, p. xiv

We study the object-oriented approach as a set of principles, methods and tools which can be instrumental in building "production" software of higher quality than is the norm today.

Object-oriented design is, in its simplest form, based on a seemingly elementary idea. Computing systems perform certain actions on certain objects; to obtain flexible and reusable systems, it is better to base the structure of software on the objects than on the actions.

Once you have said this, you have not really provided a definition, but rather posed a set of problems: What precisely is an object? How do you find and describe the objects? How should programs manipulate objects? What are the possible relations between objects? How does one explore the commonalities that may exist between various kinds of objects? How do these ideas relate to classical software engineering concerns such as correctness, ease of use, efficiency?

Answers to these issues rely on an impressive array of techniques for efficiently producing reusable, extendible and reliable software: inheritance, both in its linear (single) and multiple forms; dynamic binding and polymorphism; a new view of types and type checking; genericity; information hiding; use of assertions; programming by contract; safe exception handling. Efficient implementation techniques have been developed to allow practical application of these ideas.

[T]he book relies on the object-oriented language **Eiffel**.

4.1 Process and Data (p. 41)

A software system is a set of mechanisms for performing certain actions on certain data.

*Selected by Tom Verhoeff, Eindhoven University of Technology, Department of Mathematics and Computing Science.

When laying out the architecture of a system, the software designer is confronted with a fundamental choice: should the structure be based on the actions or on the data? ...

The discussion will use interchangeably the words "function", "action" or "process" when considering the first approach, and similarly "data" or "objects" when considering the second. ...

[T]he rest of this discussion will advocate the second approach: basing system structure on data structure.

4.2 Functions, Data and Continuity (pp. 42–43)

To evaluate the quality of an architecture (and the method that produced it), we should not just consider how easily this architecture is initially obtained: it is just as important, and perhaps more, to ascertain how well the architecture will weather the process of change. Here objects have a decisive edge over functions. ...

As a system evolves, its tasks may change dramatically. Much more persistency may be found in the classes of data it manipulates, *at least if viewed from a sufficient level of abstaction.*

4.3 The Top-Down Functional Approach (pp. 43–44)

Classical design methods typically use the functions, not the objects, as the basis. The best-known method is top-down functional design. ...

The top-down method is based on the idea that software should be based on stepwise refinement of the system's abstract function. ...

But top-down design suffers from several flaws.

- The method fails to take into account the evolutionary nature of software systems.
- The very notion of a system being characterized by one function is questionable.
- Using the function as the basis often means that the data structure aspects is neglected.
- Working top-down does not promote reusability.

4.5 Object-Oriented Design (pp. 50–51)

We ... consider a first definition of object-oriented design.

> **Definition 1:** Object-oriented design is the method which leads to software architectures based on the objects every system or subsystem manipulates (rather than "the" one function it is meant to ensure).

The above definition provides a general guideline for designing software the object-oriented way. It also raises a number of issues:

- How to find the objects.
- How to describe the objects.
- How to describe the relations and commonalities between objects.
- How to use objects to structure programs.

4.6 Finding the Objects (p. 51)

Newcomers to the object-oriented method often ask how the objects are to be found. ... [A] proper formulation ... involves *classes* rather than objects. But we may take a first look at it by noting that the answer is, in some practical cases, surprisingly simple. ...

[A] well-organized software system may be viewed as an **operational model** of some aspect of the world. ... When software design is understood as operational modeling, object-oriented design is a natural approach: the world being modeled is made of objects ... and it is appropriate to organize the model around computer representations of these objects. ...

There is more to say about finding the classes than this discussion implies. But the above simple remarks are often remarkably fruitful: just use as your first software objects representations of the obvious external objects.

4.8 A Precise Definition (p. 59)

We ... provide a more technical definition of object-oriented design.

> **Definition 2:** Object-oriented design is the construction of software systems as structured collections of abstract data type implementations.

4.9 Seven Steps toward Object-Based Happiness (pp.60–62)

Level 1 (*Object-based modular structure*): Systems are modularized on the basis of their data structures.

Level 2 (*Data abstraction*): Objects should be described as implementations of abstract data types.

Level 3 (*Automatic memory management*): Unused objects should be deallocated by the underlying language system, without programmer intervention.

The next step is the one which, in our opinion, truly distinguishes object-based languages from the rest of the world. ... [T]rue object-oriented programming all but identifies the notion of module with the notion of **type**. ... This fusion of two apparently distinct notions is what gives object-based design its distinctive flavor, so disconcerting to programmers used to more classical approaches.

Level 4 (*Classes*): Every non-simple type is a module, and every high-level module is a type.

Level 5 (*Inheritance*): A class may be defined as an extension or restriction of another.

Level 6 (*Polymorphism and binding*): Program entities should be permitted to refer to objects of more than one class, and operations should be permitted to have different realizations in different classes.

Level 7 (*Multiple and repeated inheritance*): It should be possible to declare a class as heir to more than one class, and more than once to the same class.

4.10 Key Concepts ... (p. 63)

Object-oriented systems are built as collections of classes. Every class represents a particular abstract data type implementation, or a group of implementations. Classes should be as general and reusable as possible; the process of combining them into systems is often bottom-up.

5.7 Classes vs. Objects (pp. 94, 100)

[T]he distinction between class and object ... is actually an easy one: a class is a type; an object is an instance of the type. Furthermore, classes are a **static** concept: a class is a recognizable element of program text. In contrast, an object is a purely **dynamic** concept, which belongs not to the program text but to the memory of the computer, where objects occupy some space at run-time once they have been created...

In [Eiffel], these concepts belong to different worlds: the program text only contains classes; at run-time, only objects exist.

7.1 The Notion of Assertion (p. 112)

Although there is no magical recipe to ensure correctness, the general direction is clear: since correctness is the conformance of software implementations to their specifications, you should try to reduce the risk of descrepancies between the two. One way to do this is to include specification elements within the implementations themselves. More generally, you may associate

with an element of executable code – instruction, routine, class – an expression of the element's purpose. Such an expression (which states what the element must do, independently of how it does it) will be called an **assertion**.

7.3 Contracting for Software Reliability (p. 115)

Preconditions and postconditions can play a crucial role in helping programmers write correct programs – and know why they are correct.

The presence of a precondition and postcondition in a routine should be viewed as a **contract** that binds the routine and its caller.

One may refer more generally to the two parties in the contract as the class and the clients. ... [A]ny good contract entails obligations as well as benefits for both parties. ...

- The precondition binds clients: it defines the conditions under which a call to the routine is legitimate.
- The postcondition, in return, binds the class: it defines the conditions that must be ensured by the routine on return.

The benefits are, for the client, the guarantee that certain results will be obtained after the call; for the class, the guarantee that certain assumptions will be satisfied whenever the routine is called.

7.4 Class Invariants ... (pp. 123–124)

Preconditions and postconditions describe the properties of individual routines. There is also a need for expressing global properties of instances of a class, which must be preserved by all routines. ...

A class invariant is ... a list of assertions, expressing general consistency constraints that apply to every class instance as a whole...

7.6 Representation Invariants (p. 132)

The representation invariant is the one part of the class's assertions that has no counterpart in the abstract data type specification. It relates no to the abstract data type, but to its representation; formally, it defines the **domain** of the abstraction function, when this function, as is usually the case, is partial.

7.7 Side-Effects in Functions (pp. 132, 136)

[S]ide-effects are permitted in functions, but they should only affect the concrete state, not the abstract state.

The object-oriented approach is particularly favorable to clever implementations which, when computing a function, may change the concrete state behind the scenes without producing a functionally meaningful side-effect.

7.9 Using Assertions (p. 143)

What is the purpose of writing assertions? These are the four main applications:

- Help in writing correct software.
- Documentation aid.
- Debugging tool.
- Support for software fault tolerance.

The first two are uses of assertions as methodological tools... The other two have to do with the run-time effect of assertions...

7.10 Coping with Failure: Disciplined Exceptions (p. 147)

[E]xceptions are a potentially dangerous technique, which has been subjected to much misuse. Too often exceptions are simply used as a form of goto instruction, making it possible to jump out of a routine when some condition other than the standard case is encountered.

There remains a role for an exception mechanism, however, in two situations:

- When the programmer wants to take into account the possibility that errors remain in the software and to include mechanisms that will handle any resulting run-time failure, either by exiting cleanly or by attempting recovery.
- When a failure is due to some abnormal condition detected by the hardware or operating system, for example a failed attempt at input and output.

7.11 Discussion (pp. 160–161)

[A]ssertions play an even more important role in the object-oriented approach than described here. With the introduction of inheritance, we shall see that assertions are essential to preserve the semantic integrity of classes and routines in the presence of the powerful mechanisms of polymorphism and redefinition.