14 PRINCIPLES OF LANGUAGE DESIGN

14.1 GENERAL REMARKS

The Perfect Programming Language

It is natural to ask if there is a perfect programming language. If there is such a language, then we should strive to identify its characteristics so that we do not waste more time on imperfect languages. In this section we argue that there is not, and cannot be, such a thing as a perfect language.

What would a perfect language be? Presumably it would be a language ideally suited to all situations—for all users, for all applications, and for all computers. This seems highly unlikely; almost every artifact or tool exists in a variety of forms designed for different situations. Consider cameras. We have cameras for novices, cameras for professionals, black-and-white cameras, color cameras, high-speed cameras, manual cameras, automatic cameras, inexpensive cameras, expensive cameras, and so on. Even an artifact as simple as a shoe exists in many different designs for the various situations in which it might be used.

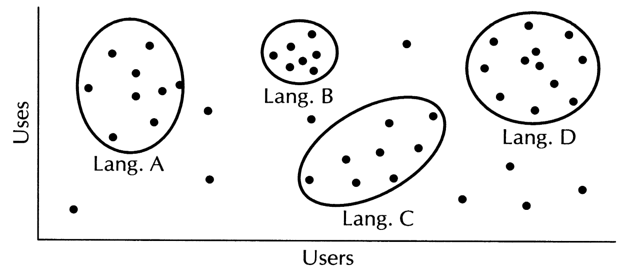
What leads to this diversity in all things? We can make the following general observations about the situations in which programming languages occur:

- There are many different classes of *uses* of programming languages.
- There are many different classes of *users* of programming languages.
- There are many different classes of *computers* on which programming languages can be implemented.

Each of these classes has different characteristics that dictate different designs.

Consider this analogy with aircraft: A training plane must be more tolerant than other planes of pilot error, even if this results in lower performance. A cargo plane must be able to carry a very large load, even if this requires it to fly slower. On the other hand, a fighter will have a lower cargo capacity but greater speed and maneuverability. Transcontinental planes must have a high ceiling to enable them to cross mountain ranges, and so forth.

Each class of uses, users, and computers leads to design decisions that are often inappropriate for the other classes. The diversity of situations in which programming languages are used is so great that any language that tried to accommodate all of these situations would be a poor compromise. A much better solution is to identify broad classes of similar uses, users, and computers and to create languages whose design decisions are optimized for these classes. This is depicted in the following diagram:



We show just two of the axes—uses and users. The points represent combinations of uses and users that frequently occur. We have drawn circles around clusters of points representing languages oriented to particular combinations of the classes. This is a better engineering approach than either extreme: designing one language to cover all points or designing a different language for each point. Of course, it is a difficult engineering problem to determine the optimal number of languages. This is particularly true since points are appearing and disappearing as computer technology evolves. Indeed, languages themselves can create new classes of users, uses, and computers.

The Perfect Language Framework

The arguments we have given against the possibility of a perfect language do not necessarily preclude a perfect *language framework.* The idea of a language framework is based on the observation that although languages may differ in their details, they are often similar in their general structures. For example, a scientific programming language might have real, integer, and complex data types, extensive array handling, and many mathematical operations. A language for string processing might have character and string data types, a record data type, and pattern-matching operations. Thus, each language has sets of application-oriented data types, functions, and operators that optimize it for a particular class of uses. The choice of these application-oriented parts also affects the class of users and computers for which the language is appropriate.

Notice that the application-oriented features are embedded in a matrix of application-independent facilities, such as definite and indefinite iterators, selectors (**case**- and **if**-statements), function and procedure declarations, and blocks. These are the same in many languages regardless of the application-oriented facilities provided. They are also relatively independent of the particular characteristics of the different classes of uses, users, and computers. Such a collection of application-independent facilities is called a *language framework.*

Can there be a perfect language framework? Our previous argument does not rule this out since the framework is independent of most of the situation-specific details. Therefore, there is some hope that a small number of simple, broadly applicable facilities can be combined into a language framework that can be used as the basis for a number of more specialized languages. Some computer scientists believe that function-oriented languages similar to LISP may provide this framework. Others think that object- or logic-oriented programming is a better choice. This question remains an important research area.

■ ***Exercise 14-1*:*** Write a report defending function-, object-, or logic-oriented programming as a basis for a universal language framework. As an alternative, write a report defending the thesis that there can be no universal language framework.

14.2 PRINCIPLES

"Knowing how to apply maxims cannot be reduced to, or derived from, the acceptance of those or any other maxims."
—Gilbert Ryle

In this section we collect the language design principles that have been illustrated throughout this book. Before presenting them, however, it is well to consider how they should be interpreted. For example: Are they ironclad laws, never to be violated? Do they form a mutually consistent set of language design *axioms*? Do they constitute an algorithm, or at least a set of formal constraints, for language design? The answer to all these questions is *no.*

First, observe that these principles are not independent; some of them are corollaries of the others. For example, the Zero-One-Infinity Principle is a corollary of the Regularity Principle since one way to make a language more regular is by limiting the numbers in its design to zero, one, and infinity. Similarly, the Orthogonality Principle is a corollary of the Simplicity Principle since an orthogonal design is usually a simpler design. We have included these derivative principles because they focus on important special cases of the other principles.

These principles are sometimes contradictory; if one is satisfied, it may mean that another cannot be satisfied. For example, strong typing satisfies the Security Principle but at the cost of adding to the complexity of the language (thus violating the Simplicity Principle) and adding to the overall cost of the implementation (thus violating the Localized Cost Principle). Combinations of facilities that are not very useful may result from obeying the Orthogonality Principle, but it would be a violation of the Simplicity Principle to exclude them.

How are we supposed to use a collection of nonindependent, sometimes contradictory, design principles? This is the difficult part of language design. It is also the difficult part of any engineering design process. For example, one principle of airplane design might be to minimize weight; another might be to maximize safety. Yet we must sometimes increase weight to increase safety. The trade-offs among the various goals of design, as embodied in these principles, require great sensitivity to the intended use of the artifact.

In many engineering disciplines, at least some of these trade-offs can be made *quantitatively*. In other words, we can compute how much a certain safety feature will weigh and perhaps make up for the extra weight by leaving something else out (say, a few passengers). For the most part, there are (as yet) no useful quantitative measures of the properties of languages. A number of computer scientists are working in this area, so we can hope that eventually at least some parts of the language design process will be quantifiable. On the other hand, we have seen (Section 4.3) that in other engineering disciplines, aesthetic principles may be a more effective guide to good engineering design than extensive mathematical analysis. Either way, in the meantime we must make our trade-offs on the basis of qualitative judgments according to principles such as these:

1. **Abstraction:** Avoid requiring something to be stated more than once; factor out the recurring pattern.
2. **Automation:** Automate mechanical, tedious, or error-prone activities.
3. **Defense in Depth:** Have a series of defenses so that if an error is not caught by one, it will probably be caught by another.
4. **Elegance:** Confine your attention to designs that *look* good because they *are* good.
5. **Impossible Error:** Making errors impossible to commit is preferable to detecting them after their commission.
6. **Information Hiding:** The language should permit modules designed so that (1) the user has all of the information needed to use the module correctly, and nothing more; and (2) the implementor has all of the information needed to implement the module correctly, and nothing more.
7. **Labeling:** Avoid arbitrary sequences more than a few items long. Do not require the user to know the absolute position of an item in a list. Instead, associate a meaningful label with each item and allow the items to occur in any order.
8. **Localized Cost:** Users should pay only for what they use; avoid distributed costs.
9. **Manifest Interface:** All interfaces should be apparent (manifest) in the syntax.
10. **Orthogonality:** Independent functions should be controlled by independent mechanisms.
11. **Portability:** Avoid features or facilities that are dependent on a particular computer or a small class of computers.
12. **Preservation of Information:** The language should allow the representation of information that the user might know and that the compiler might need.
13. **Regularity:** Regular rules, without exceptions, are easier to learn, use, describe, and implement.
14. **Responsible Design:** Do not ask users what they want; find out what they need.
15. **Security:** No program that violates the definition of the language, or its own intended structure, should escape detection.
16. **Simplicity:** A language should be as simple as possible. There should be a minimum number of concepts, with simple rules for their combination.
17. **Structure:** The static structure of the program should correspond in a simple way to the dynamic structure of the corresponding computations.
18. **Syntactic Consistency:** Similar things should look similar, different things different.
19. **Zero-One-Infinity:** The only reasonable numbers are zero, one, and infinity.

■ ***Exercise 14-2*:*** State and illustrate at least one language design principle other than those listed above. Describe a situation under which it would be desirable to violate your principle.

EXERCISES*

1. We have shown that certain of the principles are corollaries of certain of the other principles and that certain principles contradict other principles. Explore in detail the interrelationships of the principles, explaining which are corollaries and which are contradictories.

2. Write a report on "Beyond Programming Languages" by Terry Winograd (*Commun. ACM 22*, 7, July 1979).

3. Write a report on "The Future of Programming" by Anthony Wasserman and Steven Gutz (*Commun. ACM 25*, 3, March 1982).

4*. The main ideas of functional programming had been set out by Peter Landin by 1966; the main ideas of object-oriented programming appear in Simula 67, designed in 1967, and Smalltalk was implemented by 1972; the first logic-programming system was also operational by 1972. Thus the foundations of the three major-fifth-generation paradigms were laid by 1972. Defend or attack the following claim: "There have been no fundamentally new programming language ideas since 1972."

5**. Design a programming language.