The Evolution of Imperative Programming Paradigms as a Search for New Ways to Reduce Code Duplication

T Avacheva¹ and A Prutzkow²

¹Department of Mathematics, Physics, and Medical Computer Sciences, Ryazan State Medical University named after I. P. Pavlov, Ryazan, Russian Federation
²Department of Computational and Applied Mathematics, Ryazan State Radio Engineering University named after V. F. Utkin, Ryazan, Russian Federation

E-mail: mail@prutzkow.com

Received xxxxxx
Accepted for publication xxxxxx
Published xxxxxx

Abstract

The cause for imperative programming paradigm shift is the impossibility of developing software systems of a new level of complexity. We consider the evolution of programming paradigms: structured, procedural, and object-oriented. We demonstrate new ways of code duplication reducing have appeared with the shift of paradigm. We conclude the factor of code duplication reducing determines the direction of programming paradigm evolution. We discover the constraints, which were introduced in the paradigms, simplify the development of software systems. We conclude the new constraints allow the development of more complex software systems. The main reason for the code duplication is the low qualification of programmers. Therefore, in the process of learning programming, one should pay attention to code duplication and ways to reduce it.

Keywords: programming paradigms, structured programming, procedural programming, object-oriented programming

1. Introduction

The main objective of the imperative programming paradigm evolution is to decrease expenses of program development as well as maintenance. Programming paradigms shift one another when the old programming paradigm makes developing of a program system of new complexity level impossible. However details are more interesting. Why the programming paradigms are evolving from structured programming, via procedural programming to object-oriented programming? This question inevitably arises when teaching the history of the programming paradigms.

There are ideas of it, such as:

1. “The evolution of modern programming languages is closely coupled with the development (and formalization) of the concept of data type” [1].
2. “This evolution of languages also has been heavily influenced by progress in the theory of computing, which has led to a formal understanding of the semantics of statements, modules, abstract data types, and processes” [2].
3. “Object-oriented programming developed as the dominant programming methodology during the mid-1980s, isn’t a revolution in informatics but rather a long evolutionary process aiming to improve the SP [structured programming] efficiency for complex system development” [3].

But there is another idea.

The purpose of writing a program is an algorithmic solution of a problem. However, there may be several solutions. The
best solution is chosen in accordance with some criterion related to the time complexity of the problem solution and the complexity of the program structure.

A natural criterion for choosing the structure of a program is the length of its text (number of lines): the shorter the program, the more readable and easier to debug. One of the ways to shorten the text of a program is to use the go to statement. However, small but unstructured programs required a lot of time for maintenance. Therefore, at present, the following is used as a criterion: «program structure should be such as to anticipate its adaptation and modification» [4].

To improve compliance with this criterion, additional program structures were introduced into programming languages, including those associated with the appearance of new programming paradigms. Thus, the driving force behind the evolution of paradigms and programming languages was the need to increase the degree of compliance with the criterion for choosing a program structure. The main way to increase the degree of compliance with the criterion was to reduce code duplication.

2. Code Duplication and Related Problems of Code Reuse and of Redundant Code

Code duplication is the presence in the program text of a fragment that has a functionally equivalent fragment in another part of the program.

Code duplication of program text fragments can be divided into two types:
1. Identical: identical line in text program.
2. Reducible: a fragment can be altered to another one by such as a change of code line sequence, renaming of variable, type or routine names, replacing a control construct by functionally equivalent one, and so forth.

Code duplication complicates a program modification, since it requires making the same changes to its duplicate. The proportion of duplicate lines of text in a program is, according to various estimates, from 5% to 23%, but sometimes it reaches 59% [5–6]. Code duplication is a bad programming practice [7–8]. Along with the code duplication, there are concepts of reusable code and redundant lines of program text.

Code reuse is a practice of reapplying of already developed modules in a new developing program. Also there are reuses of personnel, of specifications, of designs, of «patterns», of specified components, of abstracted modules [9].

Redundant code is a superfluous code, which can be eliminated without program functionality and correctness loss. One kind of redundant code is dead code [7–8].

3. Constraints Imposed by the Programming Paradigms

Each programming paradigm introduces constraints in programming. The constraints put the programmer in some framework that does not allow him to use the capabilities of programming languages, which contradict the paradigm, but reduce the complexity of program development.

4. The Purpose of the Study

The purpose of the study is to analyze the evolution of the paradigms of structured, procedural, and object-oriented programming and to discover ways to reduce code duplication.

Also we will discover the constraints that were introduced in these programming paradigms to simplify the development of software systems.

5. Structured Programming

5.1 The Fundamentals of Structured Programming

The founder of structured programming is Edsger Dijkstra [10, 11]. Structured programming is based on the following fundamentals [4]:

1. The use in the program of control structures of sequence, selection, and repetition, having one entry and one exit. It allows the program to compare the stage of the computational process with a fragment of the program that describes this stage. This mapping is used when debugging a program.

2. Step-wise program composition (top-down development). First, the task is divided into several stages. In the next step, each stage is divided into sub-stages again. This process continues until the stages are described by programming language statements. For a top-down design, only one decision is made at each step (for example, choosing a data structure). The remaining decisions are postponed until development reaches the desired step. Step-wise program composition allowed programs to be made more modular through staged development.

3. Proof of program correctness. «Program testing can be used to show the presence of bugs, but never to show their absence» [12]. To prove the absence of errors in the program, it is necessary to prove the theorem that the program processes any input data into the expected output. The program should be structured, that is, it should include only the structures of sequence, selection and repetition. The proof uses a mathematical apparatus, including mathematical induction.

The first two fundamentals were practically unchanged adopted by following programming paradigms. The third fundamental was never developed into practically applicable [11], therefore it had only theoretical significance.

5.2 Constraints of Structured Programming

Constraints of structured programming are:
1. Use only sequence, selection, and repetition control constructs in programs.
2. Develop a program using step-wise composition.
5.3 Routines in Structured Programming

One of the ways to reduce code duplication is to carry such duplicates into routine. Routines appeared in the 1950s and were used by programmers as a means of reducing the program length, and only in some cases for structured organization [4]. Data is transferred to routines both through parameters and global variables [13]. The use of routines was not a necessary condition for implementing structured programming [11]. The reasons for this attitude of programmers to the routines were as follows:

1. Programs did not reach such a length that would require mandatory division of them into routines.
2. The call of routines slowed down the execution of the programs, since the loading of the routine into memory, its execution, and then the release of this memory, required additional time slow by modern standard processor.

Also the division of the program into modules was optional. One of the criteria for dividing a program into modules is the length of the module [11]. For example, if a module must contain a fragment of program text no more than 50 lines, then the program text with a length of 300 lines will be divided into 6 modules. Such a division is possible through the use of global variables.

The use of structured programming decreases costs in the development of software systems [14, 15]. However, structured programming did not provide a way to reduce code duplication (except cycles), despite the existence of routines in programming languages. Routines were called in a procedural programming paradigm.

6. Procedural Programming

6.1 The Fundamentals of Procedural Programming

The length of program texts grew and it became clear that it was impossible to make a program understandable and easily modifiable by structuring alone. Structured programming appears to fall apart when applications exceed 100,000 lines or so of code [16]. Therefore, to reduce code duplication, a concept that existed in structured programming was the use of routine.

Procedural programming appeared when the division of a program into routines in order to avoid code duplication ceased to be a recommendation and became a requirement.

In procedural programming, the concept of levels of abstraction was used [17], which also appeared during the structured programming times. Each level performs a group of related functions [18]. Level functions can only access functions immediately lower level.

We define this paradigm in the follow way.

Procedural programming is a structured programming extended by additional principles:

1. Functions of each level of abstraction are described by routines.
2. Each routine performs one function at its own abstraction level.
3. The routine is single-entry, single-exit control construct, receives the input data through parameters and returns the result through parameters or through the return statement.

Routines in a programming language perform three functions:

1. Provide modularity.
2. Divide the program into abstraction levels.
3. Reduce code duplication.

6.2 Constraints of procedural programming

The constraints are 3 additional principles of procedural programming.

6.3 Procedural and imperative programming

Imperative programming is a paradigm whereby a program is a sequence of commands that change its state. Structured programming and procedural programming are kinds of imperative programming. However, in some books (for example, [19, 20]), procedural programming is a synonym for imperative programming, which differs from the paradigm of procedural programming used in this paper and, for example, in [3, 21].

Procedural programming had the only way to reduce code duplication. It is the division of the program's text into routines, each of which solves a specific function. New ways have appeared in the paradigm of object-oriented programming.

7. Object-Oriented Programming

7.1 The Fundamentals of Object-Oriented Programming

Object-oriented programming appeared at the same time as structured. For example, E. Dijkstra's ideas about structured programming and the description of the Simula object-oriented programming language were in various chapters of one book [4]. However, object-oriented programming was not claimed, as the possibilities of structured programming and procedural programming were not exhausted. Object-oriented programming slowly developed and it became the dominant programming paradigm in the 1990s. Indeed new ways to reduce code duplication allowed object-oriented programming to crowd out other paradigms.

Object-oriented programming is based on three basic principles: inheritance, encapsulation, and polymorphism. Each of these principles reduces code duplication.
7.2 Inheritance

«Inheritance is also fundamental in gathering commonalities and removing duplications» [9].
«In principle, inheritance can be implemented by code duplication. For every object class of objects defined by inheritance, there is a corresponding definition that does not use inheritance; it is obtained by expansion of the definition so that inherited code is duplicated. The importance of inheritance is that it saves the effort of duplicating (or reading duplicated) code and that, when one class is implemented by inheriting from another, changes to one affect the other. This has a significant impact on code maintenance and modification» [22].

7.3 Polymorphism

Polymorphism (with dynamic binding) allows reducing code duplication in two aspects:
1. When calling a polymorphic method, it is not necessary to check its belonging to a class (Listing 1). The dynamic binding mechanism does this automatically.

```java
if (object instanceof ClassA) {
   ((ClassA) object).someMethod();
}
```

2. Polymorphism is used to reduce code duplication in some design patterns, for example, to get rid of duplication of case values in the switch statements [7, 8].

7.4 Encapsulation

Encapsulation was born in the structured programming times as information hiding [23]. So we consider encapsulation, which specific to object-oriented programming: restrict an access to instance variables in a class only by special methods.

Encapsulation reduces code duplication due to the fact that the rules for accessing the field are implemented only in special methods, and not with each reference to the field.

7.5 Constraints of Object-Oriented Programming

Constraints of object-oriented programming:
1. Describe the solution of the problem by classes and their instances.
2. Classes and their interrelations must comply with the design principles: the Single-Responsibility principle, the Open-Closed principle, the Liskov Substitution principle, the Interface-Segregation principle, the Dependency-Inversion Principle [24].
3. Adequately use inheritance, encapsulation, and polymorphism to simplify program maintenance.
4. When designing a software system and a class system, use design patterns [25].

Note the refactoring of the program's class system also reduces code duplication [7].

8. Summary

We considered ways to reduce code duplication and constraints of the programming paradigms (see the Table). Each paradigm came from the previous one, so it inherited all existing ways to reduce duplication and constraints.

9. Conclusions

We have made the following conclusions.
I. We have analyzed the imperative programming paradigms: structured, procedural, and object-oriented for compliance with the criterion for choosing the structure of the program. For each paradigm, ways to reduce code duplication were discovered. We have concluded the evolution of programming paradigms was influenced by the search for new ways to solve the problem of code duplication.
II. Despite the appearance in programming languages of new ways to reduce code duplication, the problem remains actual. The reason for the appearance of duplication is not the shortcomings of a programming language or software that eliminates it, but a low qualification of programmers. Therefore, the main way to reduce code duplication is to focus attention on this problem during programming teaching, its negative consequences and ways to reduce it, which exist in modern programming languages.
III. If the possibilities of reducing duplication are exhausted, then it will be necessary to choose another way to increase the degree of compliance with the criterion and the evolution of programming languages will follow a new way.
IV. We have listed the constraints introduced by programming paradigms. The introduction of constraints allowed the development of more complex software systems. We can't develop more complex system by more simple programming tools.
V. The evolution of the programming paradigms considered in the paper influenced other paradigms (for example, functional [21]).

Avoiding code duplication is an obvious way to reduce the expenses of developing a program: write a program faster. However, with the paradigm of object-oriented programming, new ways to reduce expenses have arisen: encapsulation of program details, design patterns, etc.

«Every time you see duplication in the code, it represents a missed opportunity for abstraction. That duplication could probably become a subroutine or perhaps another class outright. By folding the duplication into such an abstraction, you increase the vocabulary of the language of your design. Other programmers can use the abstract facilities you create.
Coding becomes faster and less error prone because you have raised the abstraction level» [8].

Table. Ways to reduce code duplication and constraints of structured, procedural, and object-oriented programming.

| Programming paradigm                  | Ways to reduce code duplication | Constraints of the paradigm                                      |
|---------------------------------------|---------------------------------|-----------------------------------------------------------------|
| Structured programming                | Routines, but they were not used| 1. Use only sequence, selection and repetition control constructs in programs. 2. Develop a program using step-wise composition. |
| Procedural programming                | Routines                        | 1. Functions of each level of abstraction are described by routines. 2. Each routine performs one function at its own abstraction level. 3. The routine is single-entry, single-exit control construct, receives the input data through parameters and returns the result through parameters or through the return statement. |
| Object-oriented programming           | Inheritance, encapsulation, and polymorphism | 1. Describe the solution of the problem by classes and their instances. 2. Classes and their interrelations must comply with the design principles: the Single-Responsibility principle, the Open-Closed principle, the Liskov Substitution principle, the Interface-Segregation principle, the Dependency-Inversion Principle. 3. Adequately use inheritance, encapsulation and polymorphism to simplify program maintenance. 4. When designing a software system and a class system, use design patterns. |

References

[1] Finkel R A 1996 Advanced Programming Language Design (Menlo Park: Addison-Wesley)
[2] Booch G, Maksimchuk R A, Engle M W, Young B J, Conallen J and Houston K A 2007 Object-Oriented Analysis and Design with Applications (Redwood: Addison-Wesley)
[3] Istakova G 2006 Proc. of Int. Conf. on Computer Systems and Technologies – CompSysTech
[4] Dahl O-J, Dijkstra E W and Hoare C A R 1972 Structured programming (London: Academic Press)
[5] Fontana F A, Zanoni M and Zanoni F 2015 Agile Processes in Software Engineering and Extreme Programming eds C Lassenius et al (Berlin: Springer) pp 5–14
[6] Koschke R 2008 Software Evolution eds T Mens and S Demeyer (Berlin: Springer) pp 15–36
[7] Fowler M, Beck K, Brant J, Opdyke W and Roberts D 1999 Refactoring Improving the Design of Existing Code (Menlo Park: Addison-Wesley)
[8] Martin R 2009 Clean Code. A Handbook of Agile Software Craftsmanship (New York: Prentice Hall)
[9] Meyer B 1997 Object-Oriented Software Construction (Englewood Cliffs: Prentice Hall)
[10] Mills H D 1971 Debugging Techniques in Large Systems ed R Rustin (Englewood Cliffs: Prentice Hall) pp 41–55
[11] Yourdon E 1975 Techniques of Program Structure and Design (Upper Saddle River: Prentice Hall)
[12] Dijkstra E W 1969 Software Engineering Techniques eds J N Buxton and B Randell (Brussels: NATO Scientific Affairs Division) pp 84–8
[13] Mills H D 1988 Software productivity (New York: Dorset House)
[14] Baker F T and Mills H D 1973 Datamation 19 58–61
[15] Hugo I ST J 1977 Proc. of AFIPS’77 pp 741–52
[16] Stein J 1988 Dr. Dobb’s Journal of Software Tools for the Professional Programmer 137 18
[17] Dijkstra E W 1968 Communications of the ACM 11 pp 341–6
[18] Liskov B H 1972 Proc. of Fall Joint Computer Conference pp 191–9
[19] Brooksheer J G 2012 Computer Science. An Overview (Reading: Addison-Wesley)
[20] O’Regan G 2012 A Brief History of Computing (London: Springer)
[21] Hyvönen E and Seppänen J 1987 2 Lisp-Maailma. Ohjelmointimenetelmät ja -jarjestelmät (Helsinki: Kirjayhtymä Oy)
[22] Mitchell J C 2002 Concepts in Programming Languages (New York: Cambridge University Press)
[23] Parnas D L 1972 Communications of the ACM 15 1053–8
[24] Martin R 2003 Agile Software Development: Principles, Patterns, and Practices (Upper Saddle River: Prentice Hall)
[25] Gamma E, Helm R, Johnson R and Vlissides J 1995 Design Patterns Elements of Reusable Object-Oriented Software (Reading: Addison-Wesley)