Weakening the Sufficient Condition for the Constant Speed of the Software Development Process

Eugeniy Tyumentcev

Dostoevsky Omsk State University, Omsk, Russia
etyumentcev@gmail.com

Abstract. In the study [4], the amount of software development work was measured as a function of the number of machine instructions in 11 major software projects. It turned out that the dependence has the form of a power function with exponent 1.5.
Articles [5, 6] described a mathematical model of the process of software development as a process of editing program code. Based on this model, a sufficient condition was formulated, in which the development speed will not decrease with the growth of the project size.
In this paper, the weakened version (theorem 2) of this sufficient condition is proved. This theorem is proposed as a formal form of The Open-Closed Principle in the part of "Software entities (classes, modules, functions, etc.) should be ... closed for modification."

Keywords: Formalization · Software development process · Software · Productivity · Constant speed · Sufficient condition · The Open-Closed Principle

1 Introduction

In the study [4], the amount of software development work was measured as a function of the number of machine instructions in 11 major software projects. The results of Nanus and Farr's study plotted on Figure 1. The dashed line means the expected dependency. It was assumed that it would be linear. The solid line means the empirical dependency. It turned out that it has the form of a power function with exponent 1.5. This research was carried out more than fifty years ago and there is an opinion that such dependence is irrelevant for modern methodologies and software development tools.

Articles [5, 6] described a mathematical model of the software development process as a process of editing program code. Using this model, the example was constructed which shows the slowdown of the software development process

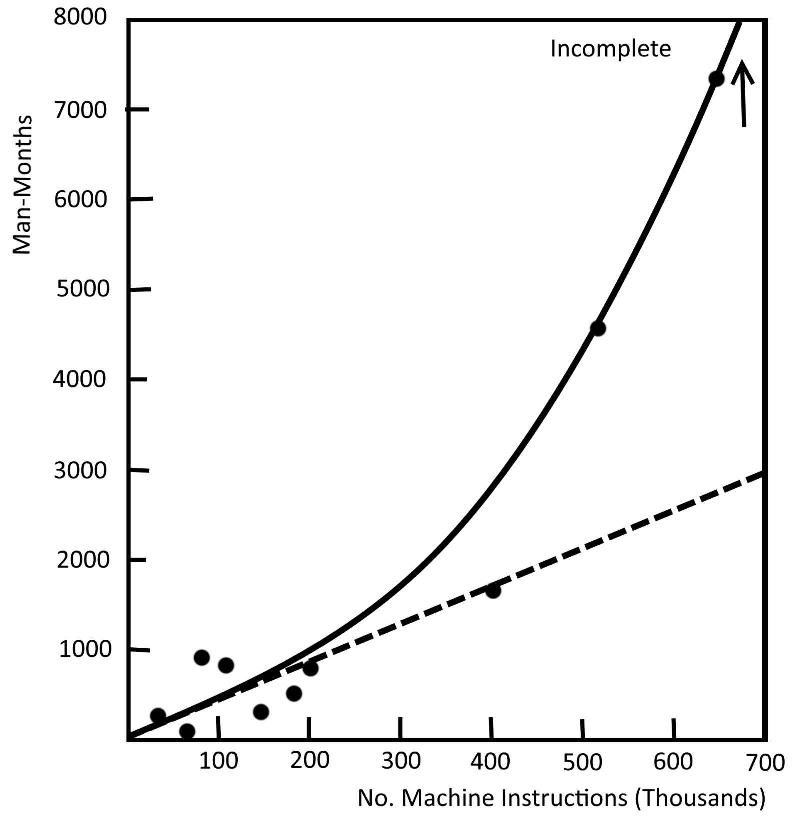

Fig. 1. The amount of work depending on the number of machine instructions

with the growth of the project size. This example can be easily reproduced in any programming language. Based on the model, in [6] the sufficient condition was formulated, in which the software development speed will not decrease with the growth of the project size.

In this paper, the weakened version of this sufficient condition is proposed (see Theorem 2). Theorem 2 explains formally one part of The Open-Closed Principle [3].

2 Formal Model of the Software Development Process

In section, there are key results of the articles [5, 6], since they have never been published in English, and acquaintance with them is necessary to understand the result of this article.

2.1 The Set of Operations

Let L be a formal language over the alphabet Σ. The set of all words over the alphabet Σ design as Σ^*, and the words themselves are symbols of the Greek alphabet α, β, γ, length of the word α design as $|\alpha|$. Remember, that Σ^* is a semigroup. In other words,

$$\forall \alpha, \beta \in \Sigma^* : \alpha\beta \in \Sigma^*.$$

Empty word in Σ^* is designed as ϵ.

Definition 1 (Delete Symbol). *Let's say that the word α' is obtained by removing the symbol $a \in \Sigma$ from the word $\alpha = \beta a \gamma$, where $\beta, \gamma \in \Sigma^*$, and one of words β or γ can be empty if and only if α' can be written in the form $\alpha' = \beta\gamma$.*

Definition 2 (Add symbol). *Let's say that the word α' is obtained by adding the symbol $a \in \Sigma$ to the word $\alpha = \beta\gamma$, where $\beta, \gamma \in \Sigma^*$, and one of words β or γ can be empty if and only if α' can be written in the form $\alpha' = \beta a \gamma$.*

Example 1. Consider the alphabet of C++. The word:

```
void f()
{
    int
}
```

is obtained by adding the character int to the word

```
void f()
{
}
```

And, conversely, the second word can be obtained from the first by removing the character int.

We want to introduce a formal definition of the software development process over the language L. In the framework of this process, the same words from Σ^* can appear. To distinguish between them we will not consider the words themselves, but the pairs (α, n), where $\alpha \in \Sigma^*$, $n \in \mathbb{N}$ and denote such pairs $\overline{\alpha}$, and call the word α the word $\overline{\alpha}$, if it is necessary.

Here and in the following text, we assume that the set of natural numbers $\mathbb{N}$ contains 0.

Let

$$S = \{\overline{\alpha}_1, \overline{\alpha}_2, \ldots, \overline{\alpha}_n\},$$

is a set of pairs (α_i, n_i), where $\alpha_i \in \Sigma^*$, $n_i \in \mathbb{N}$, $i \in \mathbb{N}$, $i \in (1, 2, \ldots, n)$, all the n_i are pairwise distinct.

Let's define four types of operations on the set of pairs S:

1. Let's say that the set of pairs S' is obtained from S by adding to S the pair (α, k), where $\alpha \in \Sigma^*$, $|\alpha| = 1$, $k \in \mathbb{N}$, if and only if
$$S' = S \cup \{(\alpha, k)\},\ (\alpha, k) \notin S,$$
and $\nexists(\beta, t) \in S : k = t$.
2. Let's say that the set of pairs S' is obtained from S by deleting the pair (α, k), where $\alpha \in \Sigma^*$, $k \in \mathbb{N}$, if and only if
$$S' = S \setminus \{(\alpha, k)\},\ (\alpha, k) \in S.$$
3. Let's say that the set of pairs S' is obtained from S by replacing the pair (α, k), where $\alpha \in \Sigma^*$, $k \in \mathbb{N}$, by (α', k), where the word α' was obtained from the word α by adding the character $a \in \Sigma$ in the sense of Definition 3, if and only if
$$S' = S \setminus \{(\alpha, k)\} \cup \{(\alpha', k)\},\ (\alpha, k) \in S.$$
4. Let's say that the set of pairs S' is obtained from S by replacing the pair (α, k), where $\alpha \in \Sigma^*$, $k \in \mathbb{N}$, by (α', k), where the word α' was obtained from the word α by deleting the character $a \in \Sigma$ in the sense of Definition 2, if and only if
$$S' = S \setminus \{(\alpha, k)\} \cup \{(\alpha', k)\},\ (\alpha, k) \in S.$$

Remark 1. $\forall \alpha \in \Sigma^*$ there exists a sequence of operations the result of which is the pair (α, n), for some $n \in \mathbb{N}$.

Proof. Let $\alpha \in \Sigma^*$. Suppose that $\alpha = a_1 a_2 \ldots a_n$, where $n > 0$. Then we take the following set of operations: Using the operation of type 1, we add the pair (a_1, n). Then, for each symbol $a_i, i > 1$, with the help of an operation of type 3, we replace the pair $(a_1 a_2 \ldots a_{i-1}, n)$ by a pair $(a_1 a_2 \ldots a_{i-1} a_i, n)$.

Remark 2 (Existence of a set of operations.). Let S is an arbitrary set of pairs. Then there is a sequence of operations, the result of which is a given set of pairs.

Proof. It is carried out by induction on the number of pairs in the set S and applying the previous remark to each pair.

2.2 Definition of the Software Development Process

Let $\hat{L}$ be a subset of words of the programming language L over some alphabet Σ. The subset $\hat{L}$ symbolizes the restrictions imposed by programmers on the language used in connection with the chosen programming methodology for example, procedural, object-oriented, etc. or the coding standards adopted in the development process, for example, the prohibition on the use of global variables.

Definition 3. *Let*
$$S = \{\overline{\alpha}_i | i \in (1, 2, \ldots, n)\},$$
where $n \in \mathbb{N}$, $\alpha_i \in \Sigma^*$, *is some set of pairs. S is called a program in the language L if and only if* $\forall \overline{\alpha}_i : \alpha_i \in \hat{L}$.

Definition 4 (The Software Development Process). *The sequence of sets of pairs:*

$$P_0 \xrightarrow{c_1} P_1 \xrightarrow{c_2} P_2 \xrightarrow{c_1} \dots$$

where $P_0 = \varnothing$, $P_i\,(i > 0)$ are sets of pairs, and P_i is obtained from P_{i-1} by using operation c_i of one of the above types, while for all operations of type 1 all the added pairs have the form (α, k), where $\alpha \in \Sigma^*$, k is a step number at which the operation was performed, is called the software development process Pr.

Definition 5. *Let's say that the program P is obtained using the software development process Pr, if $\exists i > 0 : P_i = P$.*

All further reasoning will be carried out on the assumption that there is some process of developing Pr:

$$P_0 \xrightarrow{c_1} P_1 \xrightarrow{c_2} P_2 \xrightarrow{c_1} \dots .$$

Remark 3. By definition of the software development process, either pair (α, k) does not exist, which was created in the software development process, either there is only one, for any $k \in \mathbb{N}$.

Let's define for convenience the sequence of sets of removed pairs as:

1. $D_0 = \varnothing$,
2. $D_i = D_{i-1} \cup \{\overline{\alpha}\}$, if $\overline{\alpha}$ was removed on the step i from the set P_{i-1} by using operation of type 2, $D_i = D_{i-1}$ otherwise.

Proposition 1. $\forall n \in \mathbb{N} : P_n \cap D_n = \varnothing$.

The proof is by induction by virtue of constructing the set of pairs P_n.

2.3 Speed of the Software Development Process

Next, let's define the sequence of functions

$$F_i : (\Sigma^* \otimes \mathbb{N}) \to \mathbb{N}, i \in 1, 2, ..., n :$$

on the set $\Sigma^* \otimes \mathbb{N}$ – the Cartesian product of the sets Σ^* and $\mathbb{N}$

1. $F_0(\Sigma^* \otimes \mathbb{N}) = 0$.
2. Suppose that all functions F_j, $j \le i$ were already defined. Let $\overline{\alpha} \in \Sigma^* \otimes \mathbb{N}$ be some pair. Then
   - $F_i(\overline{\alpha}) = 1$, if the pair $\overline{\alpha} = (\alpha, i)$ was adding to P_i by operation of type 1 at step i.
   - $F_i(\overline{\alpha}) = F_{i-1}(\overline{\alpha}')+1$, if the pair $\overline{\alpha} = (\alpha, k)$ was obtained from $\overline{\alpha}' = (\alpha', k)$ by operation of type 3 or 4 at step i.
   - $F_i(\overline{\alpha}) = F_{i-1}(\overline{\alpha}) + 1$, if the pair $\overline{\alpha} = (\alpha, k)$ was removed at step i from the set P_{i-1} by operation of type 2.
   - $F_i(\overline{\alpha}) = F_{i-1}(\overline{\alpha})$, otherwise.

Definition 6. *Let $\overline{\alpha} \in \Sigma^* \otimes \mathbb{N}$. Then the complexity of writing the pair $\overline{\alpha}$ on the step n in the software development process Pr is called $F_n(\overline{\alpha})$.*

Proposition 2.

$$\forall \overline{\alpha} \in \Sigma^* \otimes \mathbb{N}\, \forall k, m \in \mathbb{N} : k > m \Rightarrow F_k(\overline{\alpha}) \geq F_m(\overline{\alpha}).$$

It follows from the construction of a set of functions $F_i : \Sigma^* \otimes \mathbb{N} \to \mathbb{N}$, $i \in \mathbb{N}$.

Proposition 3. *Let $\overline{\alpha} \in \Sigma^* \otimes \mathbb{N}$. Then the sequence $\{F_i(\overline{\alpha}), i \in \mathbb{N}\}$ either converges to some $C \in \mathbb{N}$, either $\forall C \in \mathbb{N}\, \exists k > 0 : F_k(\overline{\alpha}) > C$.*

It follows from the definition of functions F_i, $i \in \mathbb{N}$, and proposition 2.

Definition 7. *Let $\overline{\alpha} \in \Sigma^* \otimes \mathbb{N}$. Then the asymptotic complexity of writing the pair $\overline{\alpha}$ in the software development process Pr is called $\lim\limits_{i \to \infty} F_i(\overline{\alpha})$.*

Definition 8. *Let $F : \mathbb{N} \to \mathbb{R}^+$ is some function from $\mathbb{N}$ to $\mathbb{R}^+$, where $\mathbb{R}^+ = \{x \in \mathbb{R}, x \geq 0\}$. Let's say that software development process Pr has speed not better, than F up to constant, if and only if*

$$\exists k \in \mathbb{N}\, \forall n > k\, |P_n| \leq F(n) \cdot n.$$

Here and below, $|P_n|$ denotes the cardinality of the set P_n.

Definition 9. *Let $F : \mathbb{N} \to \mathbb{R}^+$ is some function from $\mathbb{N}$ to $\mathbb{R}^+$, where $\mathbb{R}^+ = \{x \in \mathbb{R}, x \geq 0\}$. Let's say that software development process Pr has speed not worse, than F up to constant, if and only if*

$$\exists k \in \mathbb{N}\, \forall n > k\, |P_n| \geq F(n) \cdot n.$$

Definition 10. *Let $F : \mathbb{N} \to \mathbb{R}^+$ is some function from $\mathbb{N}$ to $\mathbb{R}^+$. Let's say that the software development process Pr has speed F up to constant if and only if*

$$\exists C_1 > 0, C_2 > 0\, \exists k \in \mathbb{N}\, \forall n > k\;\; C_1 \cdot F(n) \cdot n \leq |P_n| \leq C_2 \cdot F(n) \cdot n.$$

2.4 The Sufficient Condition for the Constant Speed of the Software Development Process

Let Pr be a software development process:

$$P_0 \xrightarrow{c_1} P_1 \xrightarrow{c_2} P_2 \xrightarrow{c_1} \ldots.$$

In terms of the previous section

Proposition 4.

$$\forall n \in \mathbb{N}\, |P_n| + |D_n| \leq n.$$

The proof is by induction on the number n.

Corollary 1. *By Proposition 4, the speed of any software development process can not be better than n.*

Proposition 5.

$$\forall n \in \mathbb{N} \sum_{\overline{\alpha} \in P_n \cup D_n} F_n(\overline{\alpha}) = n.$$

The proof is by induction on the number n.

Proposition 6. *Suppose that* $\exists C > 0, C \in \mathbb{N}\ \forall n \in \mathbb{N}\ \forall \overline{\alpha} \in P_n \cup D_n : F_n(\overline{\alpha}) < C$.

$$\frac{n}{C} \leq |P_n \cup D_n| \leq n.$$

Proof. The upper limit follows from propositions 1 and 4. The restriction from below follows directly from the condition and the proposition 5.

Theorem 1 (The Sufficient Condition for The Constant Speed of The Software Development Process). - *Under the assumptions of Proposition 6, suppose that* $\exists C_1 > 0\ \exists k_1 \in \mathbb{N}\ \forall n \geq k_1\ |D_n| \leq C_1 \cdot |P_n|$. *Then the software development process Pr has a constant speed.*

Proof. By Proposition 6

$$\frac{n}{c} \leq |P_n \cup D_n| \leq n.$$

By Proposition 1

$$|P_n \cup D_n| = |P_n| + |D_n|,$$

consequently, by the assumption of the theorem

$$|P_n| + |D_n| \leq |P_n| + C_1 \cdot |P_n| = (1 + C_1)|P_n|.$$

We get that

$$\frac{n}{c} \leq (1 + C_1)|P_n|,$$

that is

$$\frac{n}{C(1 + C_1)} \leq |P_n|.$$

On the other hand, $|P_n| \leq n$. We get

$$\frac{n}{C(1 + C_1)} \leq |P_n| \leq n.$$

By definition 10 we get that the speed F of the software development process Pr equals $F(n) = 1$.

3 The Weakened Version of the Sufficient Condition

The theorem 1 requires that the assimptotic complexity of writing each word of the program is bounded by some positive integer constant C. This requirement can be weakened.

Before we formulate a new sufficient condition, we introduce several notation. Let Pr be some software development process, $C > 0$, $C \in \mathbb{N}$, be some positive integer constant. Then $\forall n \in \mathbb{N}$

$$G_n^C = \{\overline{\alpha} \,|\, \overline{\alpha} \in P_n : F_n(\overline{\alpha}) \leq C\} -$$

the set of all pairs in P_n, that were obtained no more than for C operations.

$$B_n^C = \{\overline{\alpha} \,|\, \overline{\alpha} \in P_n : F_n(\overline{\alpha}) > C\} -$$

the set of all pairs in P_n, that were obtained more than for C operations.

Remark 4. By definition of sets G_n^C and B_n^C, $n \in \mathbb{N}$

$$\forall k \in \mathbb{N}\; G_k^C \cup B_k^C = P_k,$$

$$\forall k \in \mathbb{N}\; G_k^C \cap B_k^C = \emptyset.$$

Theorem 2 (The Sufficient Condition for The Constant Speed of The Software Development Process). *In the notation of this section, suppose that* $\exists C > 0, C \in \mathbb{N}\; \exists C_1 > 0\; \exists k \in \mathbb{N}\; \forall n > k$

$$C_1 \cdot (\sum_{\overline{\alpha} \in G_n^C} F_n(\overline{\alpha})) \geq \sum_{\overline{\alpha} \in B_n^C} F_n(\overline{\alpha}) + \sum_{\overline{\alpha} \in D_n} F_n(\overline{\alpha})$$

Then the software development process Pr has a constant speed.

Proof. By proposition 5

$$\forall n \in \mathbb{N} \sum_{\overline{\alpha} \in P_n \cup D_n} F_n(\overline{\alpha}) = n.$$

We rewrite this equation with the remark 4

$$\sum_{\overline{\alpha} \in G_n^C} F_n(\overline{\alpha}) + \sum_{\overline{\alpha} \in B_n^C} F_n(\overline{\alpha}) + \sum_{\overline{\alpha} \in D_n} F_n(\overline{\alpha}) = n.$$

By the assumption of the theorem $\forall n \in \mathbb{N} : n > k$

$$\sum_{\overline{\alpha} \in G_n^C} F_n(\overline{\alpha}) + C_1 \cdot \sum_{\overline{\alpha} \in G_n^C} F_n(\overline{\alpha}) \geq n.$$

Or,

$$(1 + C_1) \cdot (\sum_{\overline{\alpha} \in G_n^C} F_n(\overline{\alpha})) \geq n.$$

Since $C_1 > 0$, then $1 + C_1 > 1$. Hence,

$$\sum_{\overline{\alpha} \in G_n^C} F_n(\overline{\alpha}) \geq \frac{n}{1 + C_1}. \tag{1}$$

By definition of set G_n^C

$$\sum_{\overline{\alpha} \in G_n^C} F_n(\overline{\alpha}) \leq \sum_{\overline{\alpha} \in G_n^C} C = C \cdot |G_n^C|. \tag{2}$$

By (1) and (2) it follows

$$C \cdot |G_n^C| \geq \frac{n}{1 + C_1}.$$

Since, $C > 0$, then

$$|G_n^C| \geq \frac{n}{C \cdot (1 + C_1)}. \tag{3}$$

By remark 4 $|G_n^C| + |B_n^C| = |P_n|$ follows that

$$|G_n^C| \leq |P_n| \tag{4}$$

By (3), (4) and proposition 4 we get

$$\frac{n}{C \cdot (1 + C_1)} \leq |P_n| \leq n.$$

By definition 10 we get that the speed F of the software development process Pr equals $F(n) = 1$.

Remark 5. Theorem 1 is a special case of Theorem 2

Proof. Indeed, if in the conditions of Theorem 2 we take the process of the software development PR, for which the complexity of all words is less than some $C > 0, C \in \mathbb{N}$, then $B_n^C = \emptyset$ and by remark 4 $G_n^C = P_n$. We obtain the statement of theorem 1.

4 Practical Using

4.1 The Open-Closed Principle

The Open-Closed Principle (OCP) was first described by Bertrand Meyer in [1]. In 1996 Robert C. Martin [3] proposed its modern formulation:

"Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification."

OCP is one of five principles for which Robert C. Martin offered the acronym SOLID. It is believed that a program that has a "good" design must meet SOLID. However, till now in the community of programmers, there is no unanimous opinion concerning the obligatory observance of SOLID. For instance, Joel Spolsky, author of a popular blog and several books about programming, one of the developers of Visual Basic For Applications, in the 38th issue of Stack Overflow podcast said:

"Last week I was listening to a podcast on Hanselminutes, with Robert Martin talking about the SOLID principles . . . they all sounded to me like extremely bureaucratic programming that came from the mind of somebody that has not written a lot of code, frankly."

Nevertheless, SOLID and, in particular, OCP has a number of interesting consequences that require an answer to the question of the appropriateness of using SOLID.

For example, the following programming language constructions:

- multiple-choice operator *switch,*
- enumeration type *enum,*
- chain of nested operators *if-else-if*

as a rule, lead to violation of OCP.

Indeed, each of these constructions has a finite set of options. If the set of options is not exhaustive, then there is a possibility that in the future it will be necessary to modify such operator to add the missing option, which is a violation of OCP. In practice, one has to deal with situations where the entire set of options is unknown beforehand or changes in time.

4.2 The Open-Closed Principle and the Sufficient Condition

In our opinion, the reasons for the ambiguous assessment of SOLID are the following:

- The vagueness of the term "good" design. Everyone puts their meaning in the term "good" design. Therefore, it is difficult to establish a causal relationship between SOLID and own understanding of "good" design.
- It is believed that SOLID are the principles of object-oriented programming. So, for other methodologies, they are not applicable.
- Martin claims that "It should be clear that no significant program can be 100% closed. . . . Since closure cannot be complete, it must be strategic. That is, the designer must choose the kinds of changes against which to close his design. This takes a certain amount of prescience derived from experience. The experienced designer knows the users and the industry well enough to judge the probability of different kinds of changes."

The sufficient condition (theorem 2) allows one to answer some of these questions about OCP. Before proceeding to the answers, we note that OCP consists of two parts:

1. "Open For Extension". In this article, we will not touch Extension in any way. This is a topic for a separate article.
2. "Closed For Modification". Our further reasoning will only be about closure.

So, in OCP it is asserted that software entities should not be modified, but not all, only those that are "strategically" important for the program being created. In our model, the absence of modifications of any word of the program means, in accordance with the proposition 2, that the asymptotic complexity of writing this word converges to some constant. In OCP, nothing is said about the fact that the asymptotic complexity of writing any word should be limited to some general constant. In [5], the example of a development process was constructed in which the asymptotic complexity of writing any word is limited, but there is no general constant that limits the asymptotic complexity of writing any word at once. In this case, the speed of the software development process tends asymptotically to 0.

However, there is another practice for "good" code [2] p.p. 54-56 "classes, procedures, functions must be small". The concept of "small" implies the existence of a certain threshold for the size of the entity. If we combine this practice with OCP, we get that the software entities should be small and closed for modification. This is very similar to the fact that the assymptotic complexity of essences must be limited to one common constant.

In this case, the "strategic" closure in Theorem 2 is replaced by the requirement $\exists C > 0, C \in \mathbb{N}\ \exists C_1 > 0\ \exists k \in \mathbb{N}\ \forall n > k$

$$C_1 \cdot (\sum_{\overline{\alpha} \in G_n^C} F_n(\overline{\alpha})) \geq \sum_{\overline{\alpha} \in B_n^C} F_n(\overline{\alpha}) + \sum_{\overline{\alpha} \in D_n} F_n(\overline{\alpha})$$

Note also that in the case of Theorem 2, the complexity of all deleted words is taken into account. In OCP, nothing is said about this, so the following situation is possible: instead of editing, the entity is completely deleted, and instead of it, a new entity is created that takes into account the necessary changes. OCP is not broken, but the change is actually done.

Instead of a "good" design, Theorem 2 proposes a completely understandable result with obvious benefits: the development speed will not fall as the size of the project grows.

In addition, Theorem 2 is valid for any programming language and is not tied to any software development methodology.

5 Conclusion

In this article, we have obtained a weakened sufficient condition for the constant speed of the software development process compared to the sufficient condition in the article [6]. This sufficient condition is proposed as the formal form of The Open-Closed Principle in the part of "Software entities (classes, modules, functions, etc.) should be ... closed for modification."

References

1. Bertrand, M.: Object-Oriented Software Construction. Prentice Hall (1988)

2. Fowler, M.: Refactoring: Improving the Design of Existing Code. Simvol-Plus (2003)
3. Martin, R.: The Open-Closed Principle. Object Mentor (1996). http://www.objectmentor.com/resources/articles/ocp.pdf (revised date: 5.10.2015)
4. Nanus, B., Farr, L.: Some cost contributors to large-scale programs. In: AFIPS Proc. SJCC. Spring. vol. 25, pp. 239–248 (1964)
5. Tyumentcev, E.A.: About the formalization of the software development process. Mathematical Structures and Modeling 3(43), 96–107 (2017)
6. Tyumentcev, E.A.: Clarification of the article "About the formalization of the software development process". Mathematical Structures and Modeling 1(45), (accepted, in press) (2018)