ISSN 2658-5014 (Print)
http://siit.ugatu.su

ISSN 2686-7044 (Online)
2026. Vol. 8, No. 2(26)

SYSTEMS ENGINEERING AND INFORMATION TECHNOLOGIES
Scientific paper

UDC 004.8 DOI: 10.54708/SIIT-2026-no2-p29 EDN: HGXVAJ

What is a good program? A space way

A. V. Prutzkow

Ryazan State Radio Engineering University
Lipetsk State Pedagogical University

To write a good program, you need to know what it is. Existing definitions of a good program are simply enumerations of its attributes. The purpose of the study is to formalize the concept of a good program, which is essential for program development. We introduce a program model that will help us define a good program. A program is a point in multidimensional space. The dimensions of space are the program attributes. The dimensions of space are interdependent. A change in the value of a coordinate in one dimension does not necessarily entail an opposite change in the value of a coordinate in another dimension, as in a trade-off triangle with schedule, cost, and quality at its corners. A good program is a set of points in a subspace of the program space. As an example, we consider a four-dimensional space with easily-modifiability, reliability, security, and performance as dimensions. For each dimension, we survey the measurement methods. The proposed program model reduces ambiguity from the conceptual level to the level of defining the space's dimensions.

Good program; space; model; measure; metrics; dimension; easily-modifiability; reliability; security; performance.

Introduction

Every good programmer wants to write good programs. It's difficult to do it in many aspects. One of the aspects is vagueness of the "good program" concept.

Retrospection

There are retrospective considerations when defining a good program.

D. Cassel [Cas83] summarizes that a good program (1) works according to the specs, (2) does not use excessive memory, (3) runs efficiently, (4) is easy to read, (5) is easy to understand, (6) is easily debugged and tested, and (7) can be maintained with minimum of effort.

K. Wehmeyer [Weh84] considers traits of a good program. A good program (1) works according to specification, (2) is done on time, (3) is simple, (4) is well structured, (5) is efficient, (6) is cost effective, (7) is changeable, and (8) is well documented.

I. Bratko [Bra12] collects criteria for judging how good a program is: correctness, user-friendliness, efficiency, readability, modifiability, robustness, and documentation.

A. Behforooz and F. Hudson [Beh96] reveal measurable attributes that a good program has. These are readability, understandability, and comprehensibility (RUC), logical structure, physical layout, robustness, CPU efficiency (speed of execution), memory efficiency, complexity, human factors, system interfaces, and reusable code.

G. Weinberg [Wei71] reports specifications, schedule, adaptability, and efficiency make a program good.

User-Oriented and Developer-Oriented, Internal and External Attributes

Almost each attribute of a program can be user-oriented or developer-oriented. User-oriented attribute is considered from the viewpoint of convenience and cost of use, and developer-oriented attributes – the complexity of creation and modification. For example, if developer-oriented productivity is the time to complete a task, then user-oriented productivity is the time for a program response to a user command. This multiple consideration necessitates different definitions and measurements of attributes, that complicates the "good program" concept as well. Reliability/availability, rapid delivery, and low cost are primarily user-oriented rather than developer-oriented attributes [Mus04].

N. Fenton and J. Biman [Fen15] divide attributes into internal and external. Internal attributes are measured purely in terms of the product, process, or resource itself. External attributes are measured only with respect to how the product, process, or resource relates to its environment.

Systematizing

There is a trade-off triangle with schedule, cost, and quality at its corners as a general management fundamental.

We agree with S. McConnell on "In software, however, having a "quality" corner on a trade-off triangle doesn't make much sense. A focus on some kinds of quality reduces cost and schedule, and on other kinds increases them. In the software arena, a better way to think of trade-offs is among schedule, cost, and product. The product corner includes quality and all other product-related attributes including features, complexity, usability, modifiability, maintainability, defect rate, and so on [Mcc96] (Fig. 1).

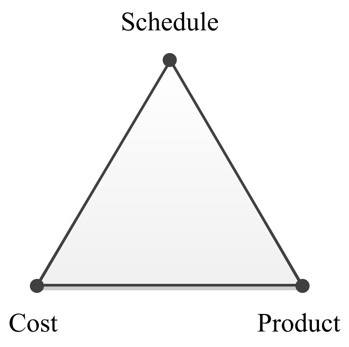

Fig. 1. Software trade-off triangle

According to the ISO/ IEC 25010:2023 standard, the product quality model categorizes product quality properties (attributes) into nine characteristics: functional suitability, performance efficiency, compatibility, interaction capability, reliability, security, maintainability, flexibility, and safety. From an edition to an edition, the standard is expanded by new attributes that make the term of quality closer to the McConnell's product term.

Hewlett-Packard (HP) uses the FURPS+ model for software quality to define the priorities in measurable terms [Gra92]. FURPS is an acronym for Functionality, Usability, Reliability, Performance, and Supportability. After several campaigns in HP, the model was extended and its name was widen with the "+".

Program and Spaces

Programs are associated with spaces in the following models.

Software behavior is described as the Input-Program-Output model [Fen15, Yam16] (Fig. 2). Input datasets are Input Space, and the output datasets are Output Space. The program is a mapping of a point in Input Space to the point in Output Space. Input datasets are divided into tested from the subspace T and untested from the subspace U_T. Some untested datasets cause a fault. Output datasets fall into expected and unexpected from the subspaces E and U_E respectively. Tested input datasets have mappings into the subspace of expected output datasets, and untested input datasets have mappings into the subspace of expected or unexpected output datasets.

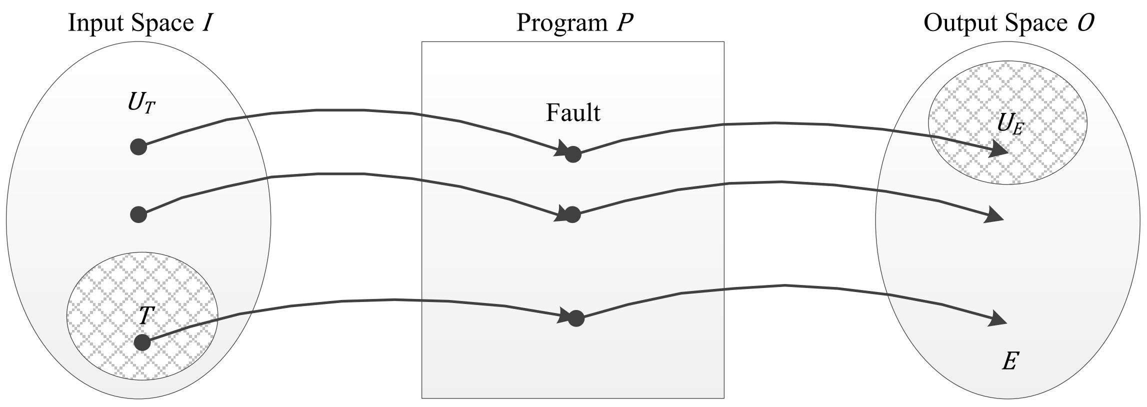

Fig. 2. An Input-Program-Output model for software behavior. T is a tested subspace, U_T is an input untested subspace, E is an output expected subspace, U_E is an output unexpected subspace

There is an open multi-dimensional model of software performance [Bug99]. Dimensions are

- the Economic one as the managers' viewpoint;
- the Social one as the users' viewpoint;
- the Technical one as the developers' viewpoint.

The model combines the dimensions within a single value and is named as the QEST model (Quality, and Economic, Social, Technical performance).

The compiler optimizes program code to reduce its execution time. Optimization involves analyzing the code and, if the analysis is positive, replacing code fragments with optimization sequences. Optimization sequences form optimization sequence space [Pur13]. A program class has a near-optimal optimization sequence in a small set. It's an alternative for a single universally optimal sequence.

One of the approaches to search for a program with specified attributes in space is genetic algorithms [Dej87, Fel98].

D. Garvin [Gar87] proposed critical dimensions of quality: performance, features, reliability, conformance, durability, serviceability, aesthetics, and perceived quality. These dimensions are also called categories.

The Purpose of the Study

The purpose of the study is to formally describe the concept of a good program, which is necessary for program development.

Program Model

The program model is as follows. A program is a point in n-dimensional space:

$$p = (x_1, x_2, \dots, x_{ND}),$$

where $x_1, x_2, \dots, x_{ND}$ are the values of the dimension coordinates.

Dimensions are program attributes (see "Introduction"), but are not only these ones.

We believe resulting (or correctness) is not a dimension, since a program must produce the expected result. If resulting is made a dimension rather than a requirement, then any sequence of characters (including zero-length ones) can be considered as a program that does not produce the expected result.

Dimensions of space are not independent. Changing the value of a coordinate in one dimension affects the values of coordinates in other dimensions. Interdependency of the dimensions are described by the following formulas:

$$(x_1 + \Delta x_1) = f_1(x_2 + \Delta x_2, x_3 + \Delta x_3, \dots, x_{ND} + \Delta x_{ND}),$$
$$(x_2 + \Delta x_2) = f_2(x_1 + \Delta x_1, x_3 + \Delta x_3, x_4 + \Delta x_4, \dots, x_{ND} + \Delta x_{ND}),$$
$$\dots$$
$$(x_{ND} + \Delta x_{ND}) = f_n(x_1 + \Delta x_1, x_2 + \Delta x_2, \dots, x_{ND-1} + \Delta x_{ND-1}),$$

where $f_i(x_1, x_2, \dots, x_{i-1}, x_{i+1}, \dots x_{ND})$ is a function that determines the value of the coordinate of the ith dimension when at least one value of the coordinate of another dimension changes, $-\infty < \Delta x_i < \infty$, $i = 1,2,\dots,ND$.

Unlike the software trade-off triangle a change in the value of a coordinate in one dimension does not necessarily entail an opposite change in the value of a coordinate in another dimension. As reliability increases, safety may improve as well or remain unchanged. C. A. R. Hoare stated, “The unavoidable price of reliability is simplicity.” The fault could be an inefficient class method [Mus04]. It’s an example of a relationship between reliability and performance.

Based on the model, a good program is points in the subspace of program space such that

$$x_1^{lo} < x_1 < x_1^{hi}, \quad x_2^{lo} < x_2 < x_2^{hi}, \quad \dots, \quad x_{ND}^{lo} < x_{ND} < x_{ND}^{hi}.$$

To determine whether a program is good, you must answer three questions:

1. How many attributes a program has or how many dimensions does program space have?
2. How are the coordinate values of each dimension calculated?
3. What coordinate values does a good program have? (assigning values x_1^{lo}, x_2^{lo}, ..., x_{ND}^{lo}, x_1^{hi}, x_2^{hi}, ..., x_{ND}^{hi}).

The dimensions of space and the way in which values are calculated depend on the task.

In the rest of the article, we will demonstrate an example of space and some ways to calculate the values of dimensions. ISO/IEC 25023:2016 standard defines quality characteristic metrics from the ISO/IEC 25010:2023 standard. But we'll omit some of them.

Example of Program Space

In [Пру25], we identified the following developer-oriented attributes or program space dimensions:

1. Easy-modifiability [Pru21] (or maintainability) – designing classes and their relationships that can be change with the least amount of effort.
2. Reliability – prevention unpredictable behavior of the program.
3. Safety – prevention appearances in a program vulnerabilities for its unauthorized use, eliminating impact malicious programs; security and safety are not the same; security is a system attribute that allows the system to perform its mission or critical functions despite the risks posed by threats, whereas functional safety is a freedom from unacceptable risk [Lis18].
4. Performance – providing that the program completes the task within a certain time.

Easily-Modifiability/Maintainability

D. Coleman et al. [Col94] set metrics for evaluating software system maintainability, rather than measuring:

$$M = 171 - 5.2\ln(H_v) - 0.23C_c - 16.2\ln(LOC_A) + 50\sin\sqrt{2.46\,C_P}.$$

where H_v stands for Halstead's volume metric [Hal77], a composite metric based on the number of (distinct) operators and operands in source code, C_c is the cyclomatic complexity metric [Mcc76], LOC_A stands for the average number of lines of code per module, and C_P is the percentage of comment lines per module.

The coefficients were obtained after calculating the metrics for 8 suites of programs [Oma94].

Instead of the previous metrics, I. Heitlager et al. [Hei07] proposed a practical maintainability model. ISO 9126 attributes of maintainability are mapped onto well-chosen source-code attributes:

volume, complexity per unit, duplication, unit size, unit testing (see table). ISO 9126 is a previous edition of ISO/IEC 25010:2023. Source-code attributes are ranked with a simple scale: ++ / + / o / - / --. The ISO 9126 attributes are ranked via mapped source-code attributes. In the table, the poor testability (-, see the last column) can be traced back to very high complexity (--) and high unit size (-), while unit testing is present, though not complete.

Table

Mapping ISO 9126 attributes of maintainability onto source code attributes

		Source-code attributes					
		volume	complexity per unit	duplication	unit size	unit testing	
		++	--	-	-	o	
ISO 9126 maintaina-bility	Analysability	x		x	x	x	o
	Changeability		x	x			-
	Stability					x	o
	Testability		x		x	x	-

A system has components. The components are layered by a stepwise refinement. A logical unit is formed by:

- components that do not form a cycle with other components;
- components that form a cycle.

Contribution of each node to maintainability c_i is [Zit22]:

$$c_i = \frac{n_L\left(1-\frac{n_I}{n_H}\right)}{n},$$

where n is the total number of components, n_L is the number of components in the logical node, n_I is the number of components influenced by the ith node, and n_H is the number of components in higher levels.

Maintainability level ML is [Zit22]:

$$ML = 100 \times \sum_{i=0}^{NN} c_i,$$

where NN is the total number of logical nodes.

The maintainability level is measured as a percentage. The higher the ML value, the better the maintainability.

Reliability

A software reliability model is probabilistic. Software reliability is the probability that a system or a capability of a system will continue to function without failure for a specified period in a specified environment [Mus04]. A failure is the departure of the external results of system operation from user needs [Mus04]. A failure is caused by a program fault.

Key parameters of the model are:

- $MTTF$: mean time to failure;
- $MTTR$: mean time to repair;
- $MTBF = MTTR + MTTF$: mean time between failures.

The most common failure model is the exponential distribution [Lai06]:

$$R(t) = e^{-\lambda t}, \qquad \lambda = \frac{1}{MTTF}.$$

There are three primary functions in reliability theory [Lai06]:

1. $f(t)$– probability distribution function of failures;
2. $F(t)$– probability of failure by time t;
3. $R(t) = 1 - F(t)$ – probability of no failure by time t.

Software reliability relates to availability A [Pre10]:

$$A = \frac{MTTF}{MTBF} \times 100\%.$$

The software reliability model is designed for forecasting [Gla79], for example, mean time to ith failure [Fen15].

Security

P. Mell et al. [Mel07] provide the Common Vulnerability Scoring System (CVSS). CVSS is an open framework for communicating the characteristics and impacts of IT vulnerabilities. CVSS consists of three groups:

1. The Base group, the intrinsic qualities of a vulnerability.
2. The Temporal group, the characteristics of a temporal vulnerability.
3. The Environmental group, the characteristics of a vulnerability that are unique to any user's environment.

The Temporal and Environmental groups are optional. Each group defines metrics, rules that convert a textual metric value into a numerical value, and an equation. The equation produces a numeric score ranging from 0 to 10.

J. A. Wang et al. [Wan09] use CVSS to calculate security metrics m_s based on the representative weakness of the software:

$$m_s = \sum_{i=1}^{NPW} p_i w_i,$$

where w_i is the ith severity of those weaknesses representative in the software, p_i is probability that represents the risk of the corresponding weakness.

The severity of weakness w_i is the average of CVSS base scores $v_1^{(i)}, v_2^{(i)}, \ldots, v_{NV}^{(i)}$:

$$w_i = \frac{\sum_{j=1}^{NV} v_j^{(i)}}{NV}.$$

S. Islam and P. Falcarin [Isl11] identify security requirements for (1) identification, authentication and authorization, (2) integrity, (3) privacy, (4) service availability, (5) non repudiation, (6) malicious activities detection, (7) security auditing, (8) backup and recovery. Requirements are a questionnaire. Score calculates as

$$s = \frac{\sum_{i=1}^{NQ} q_i}{NQ} \times 100\%,$$

where NQ is the total number of question, q_i is the question score. A maximum value of the score is 1.

Performance

Performance make up the following metrics [Eva24]:

- throughput represents the rate of work a system or subsystem can perform;
- latency is the time taken to process a single transaction and see a result;
- capacity the number of units of work that can be simultaneously ongoing in the system;
- utilization is a proportion of resource workload;
- efficiency is a proportion of the throughput to a system by the utilized resources;

- scalability is the change in throughput as resources are added;
- degradation is change of throughput the system under additional load.

The values of these metrics are measured during a test run of the program. There is four classes of software performance and scalability tests [Liu09]:

1. Performance regression testing for tracking performance metrics from release to release.
2. Performance optimization and tuning testing for supporting code optimization and program environment tuning. Program environment is a hardware platform and a software platform.
3. Performance benchmarking testing for comparison with the standard result [Lai06].
4. Scalability testing for checking out whether your software can meet customers' growing business needs.

In this classification (and in his book), H. Liu extracts scalability from performance.

Resource consumption data is the source for performance analysis [Ale00]. Benchmarks are hard to maintain, methodologies for measuring and analyzing performance are hard to develop because of innovations in the programming field [Bac25]. Examples of innovations are the rise of parallelism and latency, moving applications from mobile to the data center.

You could deepen into considerations when measuring performance in [Woo22].

Final Considerations when Measuring a Program

Goodhart's Law states, "When a measure becomes a target, it ceases to be a good measure" [Ros22]. Actually, measurement is the solution. [Woo22] Software measurement is an excellent abstraction mechanism for learning what works and what doesn't. [Dum99] "You can't control what you can't measure" [Dem82].

A metric as a measurement is not the only way to obtain the value of a system attribute. Other methods include event logs and tracing [Sri18].

The reasons for the infrequent use of metrics by modern developers are:

- developers and architects don't know a lot about metrics or how to use them [Zit22];
- there are no tools for obtaining metrics, or they obtain metrics with limitations, or are not widely known [Zit22]; HP developed and used HP B1487A Software Performance Analyzer for software measurement in the 1990s;
- most metrics require a certain level of expertise to be used effectively [Zit22] and to interpret them.

The history of program measurements and fundamental milestones are explored in [Zus99].

If you want to measure software or something else, but don't know where to start, then use the Goal-Question-Metric (GQM) technique [Bas84].

Conclusions

The proposed program model reduces ambiguity from the level of the concept to the level of defining the space dimensions (see the questions in the section "Program Model"). Existing metrics are imprecise and do not take into account all the factors influencing the coordinate value, even in a single dimension.

We have simplified the example to include only four developer-oriented program dimensions. Software projects have more dimensions.

СПИСОК ЛИТЕРАТУРЫ | REFERENCES

[Ale00] Alexander W.P. et al. (2000) A Unifying Approach to Performance Analysis in the Java Environment // IBM Systems Journal, 39(1). EDN: FNFMXJ.

[Bac25] Backburn S. et al. (2025) Rethinking Java Performance Analysis // ACM International Conference on Architectural Support for Programming Languages and Operating Systems. URL: https://doi.org/10.1145/3669940.3707217.

[Bas84] Basili V.R., Weiss D.M. (1984) A Methodology for Collecting Valid Software Engineering Data // IEEE Transactions on Software Engineering, 10(6).

[Beh96] Behforooz A., Hudson F. (1996) Software Engineering Fundamentals. Oxford University Press.

[Bra12] Bratko I. (2012) Prolog Programming for Artificial Intelligence. Addison-Wesley.

[Bug99] Buglione L., Abran A. (1999) Multidimensional Software Performance Measurement Models: A Tetrahedron-Based Design // Software Measurement.

[Cas83] Cassel D. (1983) The Structured Alternative Program Design, Style, and Debugging. Reston Pub. Co..

[Col94] Coleman D.M. et al. (1994) Using Metrics to Evaluate Software System Maintainability // Computer, 27(8).

[Dej87] De Jong K. (1987). On using genetic algorithms to search program spaces // Proceedings of the Second International Conference on Genetic Algorithms on Genetic algorithms and their application. USA, 210–216.

[Dem82] DeMarco T. (1982) Controlling Software Projects. Yourdon Press.

[Dum99] Dumke R., Abran A. (eds.) (1999) Software Measurement. Springer.

[Eva24] Evans B., Gough J. (2024) Optimizing Cloud Native Java. O'Reilly.

[Fel98] Feldt R. (1998) Generating Diverse Software Versions with Genetic Programming: An Experimental Study // IEEE Proceedings-Software, 145(6).

[Fen15] Fenton N., Bieman J. (2015) Software Metrics: A Rigorous and Practical Approach. "Chapman & Hall / CRC".

[Gar87] Garvin D. (1987) Competing on the Eight Dimensions of Quality // Harvard Business Review 65, no. 6.

[Gla79] Glass R. (1979) Software Reliability Guidebook. Prentice-Hall.

[Gra92] Grady R. (1992) Practical Software Metrics for Project Management and Process Improvement. Prentice-Hall.

[Hal77] Halstead M.H. (1977) Elements of Software Science. Elsevier.

[Hei07] Heitlager I. et al. (2007) A Practical Model for Measuring Maintainability // International Conference on Quality of Information and Communications Technology. 30-39.

[Isl11] Islam S., Falcarin P. (2011) Measuring Security Requirements for Software Security // IEEE International Conference on Cybernetic Intelligent Systems.

[Lai06] Laird L.M., Brennan M.C. (2006) Software Measurement and Estimation. Wiley-Interscience.

[Lis18] Lisova E. et al. (2018) Safety and Security Co-Analyses: A Systematic Literature Review // IEEE Systems Journal, 13(3).

[Liu09] Liu H. (2009) Software Performance and Scalability. John Wiley & Sons.

[Mcc76] McCabe T.J. (1976) A Complexity Measure // IEEE Transactions on Software Engineering, 2(4).

[Mcc96] McConnell S. (1996) Rapid Development. Microsoft Press.

[Mel07] Mell P. et al. (2007) CVSS: A Complete Guide to the Common Vulnerability Scoring System, version 2.0. URL: http://www.first.org/cvss/cvss-guide.html.

[Mus04] Musa J. (2004) Software Reliability Engineering: More Reliable Software Faster and Cheaper. AuthorHouse.

[Oma94] Oman P.W., Hagemeister J.R. (1994) Construction and Testing of Polynomials Predicting Software Maintainability // Journal of Systems and Software, 24(3).

[Pre10] Pressman R. (2010) Software Engineering. McGraw-Hill.

[Pru21] Prutzkow A. (2021) Criterion and Principles of Easily-Modified Program // Workshop on Materials and Engineering in Aeronautics, IOP Conference Series: Materials Science and Engineering, 1027, 012025. EDN: IQLYTO.

[Пру25] Пруцков А. В. Психология в программировании: аспекты междисциплинарного подхода // Преступление, наказание, исправление: сб. тез. выступл. и докл. 7-го Междунар. пенитенциарного форума. Рязань: Акад. ФСИН России, 2025. Т. 1. С. 318-320. EDN: VVIBGK. [[Prutzkow A. V. Psychology in Programming: Aspects of an Interdisciplinary Approach // Crime, Punishment, Correction: Collection of Abstracts of Speeches and Papers of Participants of the International Penitentiary Forum. Ryazan, 2025. Vol. 1. Pp. 318-320. (In Russian).]]

[Pur13] Purini S., Jain L. (2013) Finding Good Optimization Sequences Covering Program Space // ACM Transactions on Architecture and Code Optimization, 9(4).

[Ros22] Rosa J. (2022) Scaling an Organization: The Central Role of Software Architecture // Software Architecture Metrics.

[Sri18] Sridharan C. (2018) Distributed Systems Observability. O'Reilly.

[Wan09] Wang J.A. et al. (2009) Security Metrics for Software Systems // ACM Southeast Regional Conference.

[Weh84] Wehmeyer K. (1984) What Every Engineer Should Know About Microcomputer Program Design. Marcel Dekker.

[Wei71] Weinberg G. (1971) Psychology of Computer Programming. Van Nostrand Reinhold.

[Woo22] Woods E. (2022) The Role of Measurement in Software Architecture: Chapter 7 // Software Architecture Metrics. O'Reilly Media.

[Yam16] Yamada S., Yoshinobu T. (2016) OSS Reliability Measurement and Assessment. Springer.

[Zit22] Zitzewitz A. von. (2022) Using Software Metrics to Ensure Maintainability: Chapter 9 // Software Architecture Metrics. O'Reilly Media.

[Zus99] Zuse H. (1999) Thirty Years of Software Measurement // Software Measurement.

ОБ АВТОРЕ | ABOUT THE AUTHOR

ПРУЦКОВ Александр Викторович
Рязанский государственный радиотехнический университет имени В. Ф. Уткина, Липецкий государственный педагогический университет им. П. П. Семенова-Тян-Шанского, Россия.
mail@prutzkow.com ORCID: 0000-0002-4110-5269.
Проф. каф. вычислительной и прикладной математики,
проф. каф. информатики, информационных технологий
и защиты информации.

PRUTZKOW Alexander
Ryazan State Radio Engineering University,
Lipetsk State Pedagogical University, Russia.
mail@prutzkow.com ORCID: 0000-0002-4110-5269.
Prof., Dept. of Computational and Applied Mathematics.
Prof., Dept. of Computer Science, Information Technologies, and Information Security.

МЕТАДАННЫЕ | METADATA

Заглавие: Что такое хорошая программа? Пространственный подход.

Авторы: Пруцков А. В.

Аннотация: Чтобы написать хорошую программу, необходимо знать, что это такое. Существующие определения хорошей программы являются перечислениями ее параметров. Целью работы является формальное описание понятия хорошая программа, необходимое при ее разработке. Предложена модель программы, которая поможет дать определение хорошей программы. Программа – это точка в многомерном пространстве. Измерениями пространства являются ее параметры. Измерения пространства взаимозависимы. Увеличение значения координаты не всегда вызывает уменьшение значения координат остальных измерений, как в треугольнике компромиссов со сроком производства, стоимостью и качествоv в его углах. Хорошая программа – это совокупность точек в подпространстве пространства программ. В качестве примера мы рассмотрели четырехмерное пространство с легко-изменяемостью, надежностью, безопасностью и производительностью в качестве измерений. Для каждого измерения сделан обзор способов измерения. Предложенная модель программы опускает неясность с уровня понятия на уровень определения измерений пространства.

Ключевые слова: Хорошая программа; пространство; модель; метрика; пространственное измерение; легко-изменяемость; надежность; безопасность; производительность.

Язык: Русский.

Статья поступила в редакцию 15 января 2026 г.

Title: What is a good program? A space way.

Authors: Prutzkow A. V.

Abstract: To write a good program, you need to know what it is. Existing definitions of a good program are simply enumerations of its attributes. The purpose of the study is to formalize the concept of a good program, which is essential for program development. We introduce a program model that will help us define a good program. A program is a point in multidimensional space. The dimensions of space are the program attributes. The dimensions of space are interdependent. A change in the value of a coordinate in one dimension does not necessarily entail an opposite change in the value of a coordinate in another dimension, as in a trade-off triangle with schedule, cost, and quality at its corners. A good program is a set of points in a subspace of the program space. As an example, we consider a four-dimensional space with easily-modifiability, reliability, security, and performance as dimensions. For each dimension, we survey the measurement methods. The proposed program model reduces ambiguity from the conceptual level to the level of defining the space's dimensions.

Key words: Good program; space; model; measure; metrics; dimension; easily-modifiability; reliability; security; performance.

Language: English.

The article was received by the editors on 15 January 2026.