Conceptual Software: The Theory Behind Agile-Design-Rules

Iaakov Exman
Software Engineering Department
The Jerusalem College of Engineering – JCE - Azrieli
Jerusalem, Israel
iaakov@jce.ac.il

Abstract—**Software practices often evolved sets of efficient software design rules embodying together a kind of methodology. However, methodologies per se are no substitute for a rigorous software theory. Methodologies can live side by side with a software theory which explains and justifies the widely accepted wisdom of the field. This paper shows that Linear Software Models, an algebraic Software Theory together with its basis, the Conceptual Integrity principles, indeed explain the deeper contents of the so-called "four rules of simple design", which we concisely name as Agile-Design-Rules. These rules are a succinct expression of agile design methodologies that emerged from Extreme Programming (XP). Thus one obtains the best benefits from Software Theory and methodology co-existence: 1st, the explained rules reinforce the Software Theory plausibility; 2nd, the Software Theory selectively clarifies roles of the Agile-Design-Rules enabling quantitative calculations for their application in practice; 3rd, co-existence leads to the idea of *Design Tests* as illustrated by case studies.**

Keywords: Conceptual Software; algebraic Software Theory; Agile-Design-Rules; Software Design; Linear Software Models; Modularity Matrix; Modularity Lattice; Conceptual Integrity; Propriety; Orthogonality.

I. INTRODUCTION

A well-known set of rules for software system design is the so-called "four rules of simple design" which we concisely name as Agile-Design-Rules. These rules were first formulated by Kent Beck (see page 57 in [2]) at the end of the previous century, within the context of Extreme Programming (see e.g. [3]), commonly abbreviated as XP. This well-known kind of agile design methodology had a significant influence on approaches to practical software system development.

Some outstanding XP development characteristics [3] are:

- *Simple Design* – expressed, e.g. by the Agile-Design-Rules;
- *Tests* – software development is driven by tests written and run in parallel to the software itself.
- *Pair Programming* – production code is often written by two people at one screen/keyboard/mouse.

DOI: 10.18293/SEKE2018-182

We claim, that however successful in practice, any development methodology such as XP is no substitute for a rigorous Software Theory. On the other hand, a theory totally disconnected from practical directives in a field eminently application-oriented such as Software Engineering is of no use.

This paper has two-way goals: a- to argue that the algebraic Software Theory is a rigorous basis for applicable software design methodologies; b- to show that the Agile-Design-Rules essence is selectively explained and justified by the referred Software Theory. We introduce the Agile-Design-Rules, the basics of Conceptual Software design, and the algebraic Linear Software Models.

A. *Agile-Design-Rules for Software Development*

There are several formulations of the Agile-Design-Rules, differing by the wording of each rule and the rules' order. The rules' essence is common to all formulations. Here, we choose a formulation by Ron Jeffries [25], reordering rules 2 and 3:

1. ***Test Everything*** – All the tests for the SUD (Software Under Development) are passing;
2. ***Explicit Intent*** – Express the ideas the software's author wants to express;
3. ***Eliminate Duplication*** – Contain no duplicate code;
4. ***Minimize Entities*** – Minimize classes and methods.

The selective interpretation of these rules will be given later on in this paper, after the theory basics are explained.

B. *Conceptual Integrity*

Conceptual Integrity is a deep software design idea proposed by Frederick Brooks [6], [7], much earlier than the Agile-Design-Rules. Historically, the earlier ideas were not recognized as the basis for the practical rules.

Three principles suggested by Brooks [6] were verbally, not formally, explained by Jackson et al. [10], [23], [24]. We focus on the two most relevant to clarify Agile-Design-Rules:

1. ***Orthogonality*** – individual functions should be independent of one another;
2. ***Propriety*** – a product should have just the functions essential to its purpose and no more;

The Conceptual Integrity principles can be expressed in terms of modularity and design simplicity. Orthogonality is a basic modularity mechanism. Propriety is an optimization: the fewer the functions performing *exactly* the same tasks, the simpler the software product. One immediately perceives their relevance to the Agile-Design-Rules. Full interpretation will follow from the theory, presented next.

C. Linear Software Models: the Modularity Matrix and the Modularity Lattice

We concisely characterize two Linear Software Models' algebraic structures, representing a software system. These are the Modularity Matrix and the equivalent Modularity Lattice. Here we explain the primitive terms of these models relevant to this work; for further details, see [11], [12].

Software systems are assumed to be hierarchical. *Structors*, the Modularity Matrix columns, are vectorial expressions of software structure, generalizing classes for any hierarchical level. *Functionals*, the Modularity Matrix rows, are vectorial expressions of software behavior, generalizing functions which are provided by structors. The standard Modularity Matrix is block-diagonal. *Modules*, illustrated in Fig. 1, are sub-system blocks along the Modularity Matrix diagonal, made of structor and functional sub-sets, disjoint to other module sub-sets.

As shown by Exman and Katz [18], Modularity Matrix design optimization neatly corresponds to Conceptual Integrity principles. Propriety justifies Linear Independence of structors among themselves and functionals among themselves. Orthogonality implies the existence of modules.

The Modularity Lattice [14] is obtainable from the Modularity Matrix, by well-known algorithms embodied in software tools (e.g. Concept Explorer) building the lattice from a given *Formal Context*. This Context is a rectangular Boolean matrix showing relations between a set of attributes and a set of objects. The standard Modularity Matrix can be seen as a special case of Context: it is square, with relations respectively between structors and functionals. A fitting Modularity Lattice is obtained (Fig. 2 corresponds to Fig. 1), in which the Top node contains the set of all functionals and the Bottom node contains the set of all structors. Modularity Lattice modules (see Exman and Speicher [14]) are the connected components, obtained by erasing the Top and Bottom nodes, which represent the whole system, and not specific modules.

The goal of Linear Software Models is to reach the standard Modularity Matrix for a software system, where all blocks are orthogonal. An outlier matrix element coupling modules, not orthogonal anymore, demands a system redesign. Due to the Modularity Lattice to the Modularity Matrix equivalence, all conclusions extracted from the Matrix are valid for the Lattice.

Paper Organization

The remaining of the paper is organized as follows. Section II describes related work. Section III concerns the intent of Conceptual Software Design. The central section IV formulates the quantitative algebraic software theory of Agile-Design-Rules. Section V illustrates Design Tests with case studies. Section VI concludes the paper with an overall discussion.

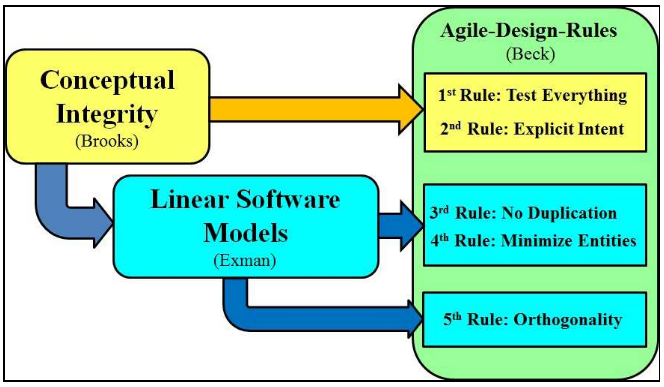
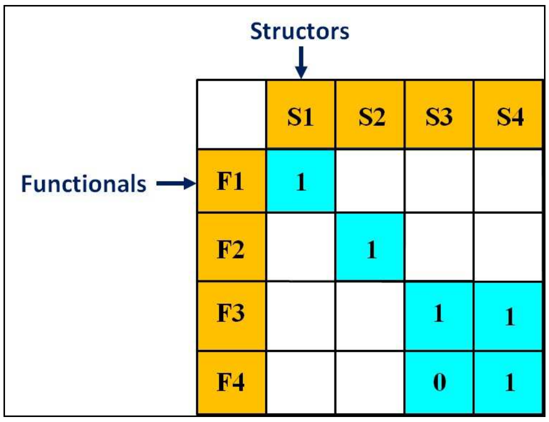


Figure 1. Software Theory explains Agile-Design-Rules – "*Conceptual Integrity*" directly explains the first two rules and is a conceptual basis for the algebraic "*Linear Software Models*". These Models directly explain the three other Agile-Design-Rules. This diagram motivates the paper organization.


Structors
Functionals

	S1	S2	S3	S4
F1	1			
F2		1		
F3			1	1
F4			0	1

Figure 2. An Abstract Standard Modularity Matrix – It has 4 Structors (matrix columns) and 4 Functionals (matrix rows). Three block-diagonal modules are seen (in blue): two strictly diagonal (S1, F1) and (S2, F2), and one 2*2 block (S3, S4, F3, F4). Matrix elements outside the modules (in white) have zero values (omitted for easier visualization). F3, an example of functional inheritance, is provided by both classes S3 and S4.

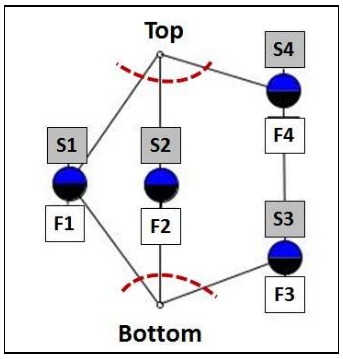


Figure 3. Abstract Modularity Lattice diagram – This Lattice exactly fits the Modularity Matrix in Fig. 1. It has three modules: two with just one-vertex (S1, F1) and (S2, F2), and one with two vertices (S3, S4, F3, F4). Structor labels are shown above and functional labels below the vertices. A vertex above another one, provides all the functionals below the higher one: e.g. S4 provides both F4 and F3, as also seen in the matrix in Fig. 1, while S3 provides only F3. Modules are the connected conponents remaining after cutting Top and Bottom nodes, as shown by the red dashed lines.

II. Related Work

A. Agile-Design-Rules for Sofware Design

Agile-Design-Rules were formulated by Kent Beck, in the first 1999 edition of his book *"Extreme Programming Explained: Embrace Change"* [2]. They were reformulated by Beck himself [3] and by several authors in different rules' order and specific wording. Martin Fowler collaborated with Kent Beck, writing together the "Planning Extreme Programming" book [4], and later on wrote a blog entitled Beck-Design-Rules [19] with his own version of the Rules.

Corey Haines published a whole book entitled "Understanding the 4 Rules of Simple Design" [21] using the Game of Life to illustrate the rules. Several other variations of the Agile-Design-Rules are due to Bekkers [5], Rainsberger [30] and Sironi [31], among others.

Hunt and Thomas [22] in their book "The Pragmatic Programmer" mention the simple design rules, stressing in Chapter 2 the relationship between Duplication (3rd rule) and Orthogonality: "The first warns not to duplicate knowledge throughout your systems, the second not to split any one piece of knowledge across multiple system components".

B. Applications of Conceptual Integrity

After Frederick Brooks' proposal of Conceptual Integrity as a fundamental idea for software development, researchers tried to apply Conceptual Integrity in practice. Despite the absence of formal quantitative criteria, these authors interpreted the Conceptual Integrity principles in ways similar to the Agile-Design-Rules, in particular rules 3 and 4.

Kazman and Carriere [27] extracted a meaningful software architecture using *conceptual integrity*. The guideline was a small number of components connected in regular ways, minimizing numbers of entities (rule 4). Kazman [26] described a SAAMtool, in which *Conceptual Integrity* is estimated by the number of primitive patterns of a system.

Clements et al. [9] interpreted *conceptual integrity* as "similar things should be done in similar ways", with parsimonious data and control, i.e. duplication avoidance and minimization of entities (rules 3, 4). They suggested counting entities as a way to quantify *Conceptual Integrity*.

C. Algebraic Structures for Software Systems

In this work we focus on the Modularity Matrix [13]. Other matrices have been used for modularity analysis. The Laplacian (von Luxburg [28]) has been used in various applications. Exman and Sakhnini [17] derived a Laplacian matrix with equivalent information to the Modularity Matrix, obtaining the same modular design for a given software system.

The 'Design Rules' by Baldwin and Clark [1], despite the name similarity to the Agile-Design-Rules, have a very different character. This approach is based upon a Design Structure Matrix (DSM), whose design quality is estimated by an external economic theory superimposed on the DSM. It has been mostly applied to non-software systems, and also to some software systems, e.g. Cai et al. [8]. A key difference from the Modularity Matrix is the lack of model linearity of the DSM.

Conceptual lattices, analyzed within Formal Concept Analysis (FCA) were introduced in Wille [32]. An overview of its mathematical foundations is given by Ganter and Wille [20]. The equivalence between Modularity Matrices and Conceptual Lattices has been shown by Exman and Speicher [14], which justifies dealing with structors as concepts.

III. Conceptual Software Design: Revealing Intention

We start here the systematic interpretation of the Agile-Design-Rules. By Fig.1 Conceptual Integrity directly explains the first two rules.

A. A Separability Principle for Software

The first Agile-Design-Rule deserves special consideration. To this end we need a Separability Principle for Software Engineering. We have formulated such a principle in [16]. It states the following:

"Software Proper vs. Human Concerns Separability Principle" – theories dealing with software proper are separable from theories dealing with human concerns of software engineering.

This Separability principle says that theories dealing with properties of the software system proper are independent of theories dealing with human stakeholder concerns, either developer processes or end-user interactions with developers.

B. Relevance to Agile Software Development

The Separability Principle is relevant to the possible meaning of the 1st Agile-Design-Rule, which tells us to continuously run all tests and make sure they still pass. This depends on the particular interpretation of this rule.

The first interpretation is trivial: a code with bugs is not runnable, and no next rule is applicable to code quality analysis. Successful tests are a pre-condition for the next rules.

Another interpretation directly touches pair-programming characteristics of XP. Pair-programming works by one person writing the code while the other person of the pair writes tests to be run on the written code, and then they switch the programmer/tester roles. From this viewpoint this rule concerns the human social aspects of development, and is not relevant to the software product proper.

In our view, the best interpretation touches the motivation for testing. The importance of tests it not just for finding bugs, but rather to enforce system redesign, in case design problems were identified. In this view, testing is an inherent part of the software product design and not an extraneous human concern. But was this the truly original motivation behind this rule?

C. 1st Agile-Design-Rule: Passing Design Tests

Whatever was the original motivation behind the first Agile-Design-Rule, we propose here a novel interpretation consistent with our emphasis on Design instead of implementation or development process.

The goal of the 1st Agile-Design-Rule is to pass systematic "*Design Tests*", viz. to reveal design problems conflicting with

Conceptual Integrity. This new focus on design means that the tests themselves should be carefully designed to be consistent with the SUD (System Under Design) Conceptual Integrity. Design test examples will be given in section V.

D. 2nd Agile-Design-Rule: Revealing Intention

The focus on design interpretation of the first rule is a suitable transition to the deep meaning of the 2nd Agile-Design-Rule. This rule in the formulation presented in the Introduction of this paper (in sub-section *A*) reads "Explicit Intent", viz. to explicitly express the ideas of the software author. In other words, the concepts embodied in the software design units should both reflect the main ideas of the software system and be clearly understood by other stakeholders reading the software. Summarizing, *Conceptual Integrity* is not only essential to high-quality design, it should be explicitly revealed in the software itself, and not just in its documentation.

IV. The Algebraic Software Theory is Quantitative!

To be applicable to the practice of software system design an actual Software Theory should be quantitative, as it is clear even in the naïve formulation of the rules: "*no duplication*" and "*minimize entities*". In this section we provide formulas for calculating the relevant quantities, to explain rules 3 and 4 and later on propose a 5th rule.

A. A Quantitative Theory of Agile-Design-Rules

The quantitative algebraic Software Theory, the Linear Software Models, which in turn is based upon Conceptual Integrity (see Fig. 1), obeys the following demands:

- ***Software represented by a mathematical structure*** – be it a matrix or a lattice; in this paper we chose the matrix representation;
- ***Quantities in formulas amenable to calculation*** – getting precise numbers for each obtained design;
- ***Standard Criteria for design quality*** – allowing comparison of proposed designs with standards;

Quantities involved in the Conceptual Integrity calculations are normalized. These quantities are independent of the vector/matrix sizes, by dividing results by relevant entity sizes.

B. 3rd Agile-Design-Rule: No Duplication

"No duplication" in terms of vectors, is the simplest case of linear independence: any set of identical structors are obviously linearly dependent and all but one should be eliminated. The same is true for identical functionals. Thus, the 3rd Agile-Design-Rule is a particular case of the 4th rule discussed next.

C. 4th Agile-Design-Rule: Minimize Entities i.e. Propriety

Following Exman and Katz [18], the naïve "Minimize Entities" rule corresponds to the generic linear independence *Propriety* principle of Conceptual Integrity. Linear independence within a module is evaluated by equation (1), in which **r** is the rank and **c** is the number of columns of the module sub-matrix. Since module sub-matrices are square, one could use as well the number of rows instead of the number of columns. The module propriety criterion in equation (1) has a value between zero and the maximum propriety value of 1 obtained when **r** equals **c**.

$$\textit{Propriety} = 1 - ((c - r)/c) \qquad (1)$$

D. Orthogonality

As already mentioned, Hunt and Thomas [22] linked in their book the "No duplication" rule with Orthogonality. The latter quantity is calculated as follows. Assume a pair of normalized vectors ***u*** and ***v*** i.e. all their elements are divided by the length of the respective vector. Their Orthogonality is calculated by equation (2), where $(u\bullet v)$ is the vectors' scalar product. Orthogonality has a value between zero and the maximal value 1 obtained for zero scalar product.

$$\textit{Orthogonality} = 1 - (u\bullet v) \qquad (2)$$

Software system calculations, using the above equations, should be done for the whole set of Modularity Matrix modules to obtain the combined system conceptual integrity.

V. Design Tests Illustrated by Case Studies

The Agile-Design-Rules are here illustrated by Case Studies. They are numbered and presented according to the rational interpretation given by the algebraic Software Theory, and adding a fifth Orthogonality rule.

A. 1st Agile-Design-Rule: Design Tests – ATM Conceptual Integrity Case Study

Design Tests are distinct from Unit Tests whose purpose is to find syntactic or logical errors. A design test, may check the Conceptual Integrity of a sub-system. For instance, an ATM (Automatic Teller Machine) is a reasonable machine to deposit or withdraw cash or deposit checks. But it is not currently an acceptable way to obtain a house mortgage.

Thus, a design test to verify an ATM design for Conceptual Integrity is a loop on a Financial Ontology, looking for and flagging for deletion all concepts appearing in the ATM design that are related or sub-types of the mortgage concept.

B. 2nd Agile-Design-Rule: Revealing Intention – Interdisciplinary Ambiguity Case Study

Revealing Intention is again a matter of Conceptual Design verification. Trivial cases are to demand naming of classes and functions by meaningful names such as "Bridge" or "Liquid", instead of meaningless names such as "X" or "Y" (see e.g. [29]), or even worse, misleading names.

Less trivial cases deal with ambiguity, for instance in an interdisciplinary software in which the same term has different meanings in two disciplines. An example is the usage of the "Bridge" software design pattern within an application for civil engineering dealing with tunnels and “bridges”. Another example is the usage of “Liquid” financial assets within an application about “Liquid” chemicals.

In order to verify ambiguity absence one may build an SUD (Software Under Development) Application Ontology, from the domain ontologies intersection, and check whether the same term appears in different Application Ontology branches dealing with the different disciplines.

C. 3rd Agile-Design-Rule: No duplication – Circle Functionals Case Study

As already stated above, “No duplication” is a particular simple case of Linear Dependence. Whenever there are two or more identical functionals (similarly for identical structors), one should eliminate all of them except one.

For instance, assume a geometrical application involving circles. The Modularity Matrix has a “circle” structor. Suppose it also has two functionals – calculate *area* by Π*Radius^2 and calculate *perimeter* by 2*Π*Radius. Then there are two identical rows in the matrix, in which there are 1-valued elements for these two rows in the same circle structor column. One should eliminate duplication, since both these functions depend only on the Radius variable; when one fixes either the Area or the Perimeter, the Radius is determined and also the value of the other function. These functionals are trivially dependent.

D. 4th Agile-Design-Rule: Minimize Number of Entities – General Propriety Case Study

The Propriety principle of Conceptual Integrity effectively minimizes the numbers of structors and respective functionals in a Modularity Matrix representation of a software system. Whenever there are linear dependences of row or column vectors within the matrix, one must eliminate some vectors to obtain total linear independence in the matrix. This is checked by equation (1), in which the matrix rank r should be equal to the number of structors (columns), or equivalently the number of rows (functionals). If Propriety is less than 1 by equation (1), some vectors must be eliminated by the software engineer, using semantic considerations.

For instance, in elementary trigonometry there are various cases of mutually dependent functions, in which one needs a lesser number of independent functions. To calculate the values of sine, co-sine and tangent fuctions of an angle in radians, one needs at most two of these functions.

E. 5th Agile-Design-Rule: Orthogonality – Redesign to Eliminate Coupling Case Study

The Software Theory leads us to add a fifth Agile-Design-Rule in our formulation to comply with the Orthogonality principle of Conceptual Integrity, which is obeyed by the standard Modularity Matrix. It means that all structors and functionals of a given module should be respectively orthogonal to all structors and functionals of all other modules in the software system represented by the Modularity Matrix. Orthogonality is calculated by repeated application of equation (2). If the overall matrix orthogonality is not 1, with some sparse modules, there is a case of coupling and the software system must be redesigned by the software engineer to eliminate coupling and assure orthogonality.

For instance, in a sub-system whose purpose is geodesy applications, a module performing proper geodetic calculations should be orthogonal to a module containing generic algebraic functions needed for e.g. matrix computations that may be needed within the geodetic calculations. Any redefinition of a generic algebraic function within a proper geodetic class, causes coupling of the geodetic and the algebraic modules, in need of redesign.

VI. Discussion

A. Agile-Design-Rules: Plausibility of the Conceptual and Algebraic Software Theory

Our analysis in this work of the four original Agile-Design-Rules in the formulation by Jeffries, as displayed in sub-section *A* of the Introduction to this paper, shows the following picture:

- For consistency of the 1st rule on running tests with the other rules, we proposed a novel interpretation in which tests should be essentially ***Design Tests***, instead of just debugging unit tests;
- The 2nd rule says that Conceptual Integrity besides being a general demand, it must be explicitly expressed in the names of the entities, such as classes and functions;
- The 3rd and 4th rules are completely explained by the Propriety principle which is part of the Conceptual Integrity approach; quantitatively it corresponds to the demand of Linear Independence among structors and among functionals in the Modularity Matrix;

Overall, the explanations for the Agile-Design-Rules reinforce the plausibility of the algebraic Linear Software Models, based upon Conceptual Integrity, as a Software Theory of software composition.

B. Rules Variability: Selectivity, Numbers and Order

Any theory proposed to explain and justify methodological rules of development, must be a self-consistent theory. A possible outcome is that justification must be selective, i.e. not all practical rules are derivable from the Software Theory and the theory may generate additional practical rules.

In the particular case of the Agile-Design-Rules, the 1st rule, on running tests, has a novel interpretation in order to comply with the Software Theory self-consistency. Furthermore, a new reasonable 5th rule of Orthogonality has been explicitly generated, as suggested by Hunt and Thomas [22].

The particular order of the rules seems less important, as long as they rigorously follow from the Software Theory. The rule order is perhaps of interest for rule classification, in which the 1st and 2nd rules strictly belong to a Conceptual viewpoint and the 3rd and 4th rules belong to an algebraic viewpoint.

C. Future Work

In order to solidify the explanation and justification for the Agile-Design-Rules one needs to analyze software system examples of a variety of sizes.

Another open issue is the applicability of these or similar rules to other development methodologies.

While linear independence is relevant to Modularity Lattices, their orthogonality deserves further investigation.

D. Main Contribution

There are three main contributions of this paper. 1st, it argues that Linear Software Models, the algebraic Software Theory based upon Conceptual Integrity, is a rigorous basis for software design methodologies. 2nd, it shows that the Agile-Design-Rules essence is selectively explained and justified by the Software Theory. 3rd, it proposed the idea of systematic Design Tests.

Acknowledgment

The author thanks Reuven Yagel for his useful suggestions which contributed to improve the paper.

References

[1] C.Y. Baldwin and K.B. Clark, *Design Rules*, Vol. I. The Power of Modularity, MIT Press, Cambridge, MA, USA, 2000.

[2] K. Beck, *Extreme Programming Explained: Embrace Change*, 1st edition, Addison-Wesley, Boston, MA, USA, 1999.

[3] K. Beck, "Embracing Change with Extreme Programming", IEEE Computer, Vol. 32, pp. 70-77, October 1999. DOI: 10.1109/2.796139

[4] K. Beck and M. Fowler, *Planning Extreme Programming*, Addison Wesley, Boston, MA, USA, 2000.

[5] N. Bekkers, "4 Rules of Simple Design", 2016. Web: https://www.theguild.nl/4-rules-of-simple-design/

[6] F.P. Brooks, *The Mythical Man-Month – Essays in Software Engineering* – Anniversary Edition, Addison-Wesley, Boston, MA, USA, 1995.

[7] F.P. Brooks, *The Design of Design: Essays from a Computer Scientist*, Addison-Wesley, Boston, MA, USA, 2010.

[8] Y. Cai and K.J. Sullivan, "Modularity Analysis of Logical Design Models", in *Proc. 21st IEEE/ACM Int. Conf. Automated Software Eng. ASE'06*, pp. 91-102, Tokyo, Japan, 2006.

[9] P. Clements, R. Kazman and M. Klein, *Evaluating Software Architecture: Methods and Case Studies*. Addison-Wesley, Boston, MA, USA, 2001.

[10] S.P. De Rosso and D. Jackson, "What's Wrong with Git? A Conceptual Design Analysis", in Proc. of Onward! Conference, pp. 37-51, ACM, 2013. DOI: http://dx.doi.org/10.1145/2509578.2509584.

[11] I. Exman, "Linear Software Models", video presentation of paper at GTSE 2012, KTH, Stockholm, Sweden, 2012b. Web site: http://www.youtube.com/watch?v=EJfzArH8-ls.

[12] I. Exman, "Linear Software Models: Standard Modularity Highlights Residual Coupling", Int. Journal of Software Engineering and Knowledge Engineering, Vol. 24, pp. 183-210, 2014. DOI: 10.1142/S0218194014500089.

[13] I. Exman, "Linear Software Models: Decoupled Modules from Modularity Matrix Eigenvectors", Int. Journal of Software Engineering and Knowledge Engineering, Vol. 25, pp. 1395-1426, 2015. DOI: http://dx.doi.org/10.1142/S0218194015500308

[14] I. Exman and D. Speicher, "Linear Software Models: Equivalence of the Modularity Matrix to its Modularity Lattice", in Proc. 10th ICSOFT'2015 Int. Conference on Software Technology, pp. 109-116, ScitePress, Portugal, 2015. DOI: 10.5220/0005557701090116

[15] I. Exman, "Linear Software Models: An Algebraic Theory of Software Composition", in Proc. 28th Int. Conf. on Software Engineering and Knowledge Engineering, Keynote Abstract, KSI Research, Redwood City, CA, USA, 2016.

[16] I. Exman, D.E. Perry, B. Barn and P. Ralph, "Separability Principles for a General Theory of Software Engineering: Report on the GTSE 2015 Workshop", ACM SIGSOFT Software Engineering Notes 41 (1): 25-27 (2016). DOI = 10.1145/2853073.2853093

[17] I. Exman and R. Sakhnini, "Linear Software Models: Modularity Analysis by the Laplacian Matrix", in Proc. 11th ICSOFT'2016 Int. Conference on Software Technology, Volume 2, pp. 100-108, ScitePress, Portugal, 2016. DOI: 10.5220/0005985601000108

[18] I. Exman and P. Katz, "Conceptual Software Design: Algebraic Axioms for Conceptual Integrity", in Proc. 29th Int. Conf. on Software Engineering and Knowledge Engineering, pp. 155-160 , KSI Research, Pittsburgh, PA, USA, 2017. DOI: https://doi.org/10.18293/SEKE2017-148

[19] M. Fowler, "Beck Design Rules", Blog, March 2015, URL: https://martinfowler.com/bliki/BeckDesignRules.html.

[20] B. Ganter and R. Wille, *Formal Concept Analysis: Mathematical Foundations*, Springer-Verlag, Berlin, Germany, 1998.

[21] C. Haines, "Understanding the Four Rules of Simple Design", Leanpub, 2014.

[22] A. Hunt and D. Thomas, *The Pragmatic Programmer: From Journeyman to Master*, Addison-Wesley, Boston, MA, USA, 1999.

[23] D. Jackson, "Conceptual Design of Software: A Research Agenda", CSAIL Technical Report, MIT-CSAIL-TR-2013-020, 2013. URL: http://dspace.mit.edu/bitstream/handle/1721.1/79826/MIT-CSAIL-TR-2013-020.pdf?sequence=2

[24] D. Jackson, "Towards a Theory of Conceptual Design for Software", in Proc. Onward! 2015 ACM Int. Symposium on New Ideas, New Paradigms and Reflections on Programming and Software, pp. 282-296, 2015. DOI: 10.1145/2814228.2814248.

[25] R. Jeffries, "Essential XP: Emergent Design", October 2001. URL: https://ronjeffries.com/xprog/classics/expemergentdesign/.

[26] R. Kazman, "Tool Support for Architecture Analysis and Design", in ISAW'96 Proc. 2nd Int. Software Architecture Workshop, pp. 94-97, ACM, New York, NY, USA, 1996. DOI: 10.1145/243327.243618

[27] R. Kazman and S.J. Carriere, "Playing Detective: Reconstructing Software Architecture from Available Evidence." Technical Report CMU/SEI-97-TR-010, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA, 1997.

[28] U. von Luxburg, "A Tutorial on Spectral Clustering", *Statistics and Computing*, 17 (4), pp. 395-416, 2007. DOI: 10.1007/s11222-007-9033-z

[29] K. Owen, "What's in a Name? Anti-Patterns to a Hard Problem", 2016. Web: https://www.sitepoint.com/whats-in-a-name-anti-patterns-to-a-hard-problem/

[30] J.B. Rainsberger, "The Four Elements of Simple Design", 2016. Web: http://blog.jbrains.ca/permalink/the-four-elements-of-simple-design

[31] G. Sironi, "The 4 rules of simple design", 2011. Web: https://dzone.com/articles/4-rules-simple-design

[32] R. Wille, "Restructuring lattice theory: an approach based on hierarchies of concepts" In: I. Rival (ed.): *Ordered Sets*, pp. 445–470, Reidel, Dordrecht-Boston, 1982.