Conceptual Software Design: Modularity Matrix as Source of Conceptual Integrity

Iaakov Exman(✉)

The Jerusalem College of Engineering – Azrieli, Jerusalem, Israel
iaakov@jce.ac.il

Abstract. Conceptual Software Design is of utmost importance for software development due to its focus on the *Conceptual Integrity* of software systems. However, in order to turn it into actual standard practice in software design, a precise mathematical representation of Conceptual Design is necessary. This paper claims that Linear Software Models – by means of their basic algebraic structures, the Modularity Matrix or its corresponding Laplacian Matrix – guarantee Conceptual Integrity of the software system they represent. This is argued by first offering a concise Plausibility Path with a few formal steps towards *Conceptual Integrity* in terms of the Modularity Matrix. These steps clarify the role of the Modularity Matrix, both as a facilitator and as a formal source of the software modules' Conceptual Integrity. Then, the paper characterizes Conceptual Integrity as an intensive property of the software system. Finally, application in practice is demonstrated by providing explicit formulas to compute Conceptual Integrity principles, viz. propriety and orthogonality.

Keywords: Conceptual Software Design · Conceptual integrity
Linear Software Models · Modularity Matrix · Laplacian matrix
Modularity Lattice · Abstract Domain Conceptualization · Propriety
Orthogonality

1 Introduction

The original idea of *conceptual integrity* for software systems development was introduced by Brooks in his book "The Mythical Man-Month" [3]. There he argued for the utmost importance of *conceptual integrity* for software system design. We essentially agree with Brooks' qualitative statement, and claim that its formalization in mathematical terms should completely transform its practical applicability, actually enabling its usage for software systems.

This paper, an updated extension of [15] based in recent work, explicitly reformulates *conceptual integrity* in terms of Linear Software Models – our mathematical theory of software composition. In other words, the basic algebraic structures of this theory, viz. the Modularity Matrix by Exman [11] or its corresponding Laplacian Matrix by Exman and Sakhnini [14], guarantee the Conceptual Integrity of the software system they represent, by means of an iterative procedure. It is shown that the standard form of the Modularity Matrix is both the facilitator and a formal source of the software

A. Fred et al. (Eds.): IC3K 2016, CCIS 914, pp. 233–248, 2019.
https://doi.org/10.1007/978-3-319-99701-8_11

modules' conceptual integrity. In case one deviates from the standard form, the matrix highlights the system spots in need of redesign, within the iterative procedure.

This Introduction concisely overviews the ideas of software *conceptual integrity* as presented by Brooks, and reviews the main Modularity Matrix properties.

1.1 Overview of Software Conceptual Integrity

In a more recent book by Brooks "The Design of Design: Essays of a computer scientist" [4], *conceptual integrity* was verbally described by three principles in terms of system functions. These principles are as follows:

Propriety – a software system contains only essential functions;
Orthogonality – functions are mutually independent;
Generality – many usage ways for each function.

The obstacle to practical application of these principles is that there have been no known translations to precise mathematical formulas and effective algorithms. This work provides the desired precise formalization of conceptual integrity, gaining both a deeper comprehension of Brooks' ideas and a clear basis for concrete usage of the referred principles.

1.2 Modularity Matrix Concepts

The Modularity Matrix [9–11] enables to represent any level of a hierarchical software system, through sub-systems, to sub-sub-systems and so on, down to indivisible basic components. *Structors* – the matrix columns – stand for architectural structure units, generalizing classes of object-oriented languages. *Functionals* – the matrix rows – stand for architectural behavioral units, generalizing class functions, which may be invoked, but not necessarily.

Columns and/or rows reordering, together with algebraic manipulations [13], i.e. solving for the matrix eigenvectors, lead in the optimal situation to a square and block diagonal matrix. In this standard Modularity Matrix format, the blocks along the diagonal represent the *modules* of the current matrix level.

If there are outlier non-zero matrix elements beyond the boundaries of the diagonal modules, these outliers cause modules' coupling, which should be resolved by redesigning the system. This can be done, within an iterative procedure, by splitting modules or adding/removing structors and/or functionals. Figure 1 illustrates such an abstract Modularity Matrix, with one outlier matrix element, coupling two block-diagonal modules.

It has been shown by Exman and Sakhnini [14, 17], that a corresponding Laplacian Matrix can be generated from the Modularity Matrix. A similar procedure, also involving matrix eigenvectors, produces the same modules for the same system from both matrices. Thus, the Linear Software Models, indeed are a unified software composition theory.

Structor → Functional ↓	S1	S2	S3	S4	S5	S6
F1	1	0	0			
F2	1	1	0		1	
F3	0	1	1			
F4				1	0	
F5				1	1	
F6						1

Fig. 1. An abstract Modularity Matrix with an added outlier – A standard matrix means that it is strictly square and block-diagonal. This matrix is indeed square as it displays 6 structors (columns) and 6 functionals (rows). It is also almost block-diagonal, as it displays 3 modules seen as three blocks along the diagonal, (green background). A strictly block-diagonal matrix would have outside the modules (blank areas) only zero-valued matrix elements (values here omitted for increased clarity). But this matrix shows one outlier (in hatched dark blue background), a 1-valued matrix element in {F2, S5} coupling the top-left and the middle modules. This outlier hints at a need of software system redesign. (Color figure online)

1.3 Related Literature

Here we concisely review a sample of the related literature referring to Conceptual Integrity and algebraic structures, such as matrices and lattices which have been used for software system design.

Conceptual Integrity

Conceptual Integrity ideas for software design were first proposed in Frederick Brooks' books [3] and [4], as mentioned above in the beginning of this paper.

Jackson, starting from a research proposal [21], and co-authors elaborated Brooks' ideas by detailed explanations of case studies, e.g. on Git [7] and more recently De Rosso and Jackson [8]. Jackson has emphasized the importance of concepts for software systems, illustrating them by informal dependence graphs, from which simplified and more coherent subsets of concepts can be extracted [22].

Simplicity and regularity seem to be important characteristics of *Conceptual Integrity*. An example is a Technical Report by Kazman and Carriere [24], dealing with reconstruction of a software system architecture, using *conceptual integrity* as a guide. The architecture should in principle be built from small numbers of regularly connected components, with consistent functionality allocation to these components. Another

example by Kazman [23] describes a SAAMtool, with visualization capability. *Conceptual Integrity* is estimated by the number of primitive patterns that a system uses.

Still another example is given by Clements et al. in their book [6], referring to conceptual integrity as a unifying design theme. The system should do similar things in similar ways, with small numbers of data and control mechanisms in the system. Issues with some similarity to our approach in this paper are: a - they mean the system at all hierarchical levels; b - a more precise definition of conceptual integrity would be given by counting mechanisms.

Occasional references concerning Conceptual Integrity have appeared in the literature. For instance, Beynon et al. [2] explicitly refer to Conceptual Integrity, but do not go beyond some vague statements about what it means. Orthogonality, one of the conceptual integrity principles, also have appeared in the software design literature. Krone and Snelting [25] refer to it in a paper using conceptual lattices extracted from source code.

Most recently, Exman and Katz [16] starting from an axiomatic approach, began to make explicit calculations with quantities expressing the Conceptual Integrity principles.

Algebraic Structures for Software System Design

Other algebraic structures, besides the Modularity Matrix, have been used for software systems design. The DSM (Design Structure Matrix) included in the Design Rules approach by Baldwin and Clark [1] has been applied mostly outside software engineering. It should be remarked that the DSM has been mostly analyzed by a superimposed economic options theory, external to the DSM itself, in contrast to our pure algebraic theory. For a set of references to this approach see e.g. [11].

Exman and Sakhnini [14, 17], have shown that a Laplacian matrix can be obtained from any Modularity Matrix, by means of an intermediate bipartite graph. Although a clearly different matrix, the Laplacian matrix obtains the same modules as its corresponding Modularity Matrix, by a similar spectral method – using eigenvectors and eigenvalues.

Another algebraic structure applicable to software system design is the Conceptual Lattice, developed within FCA (Formal Concept Analysis) mainly by Wille, Ganter and collaborators, see e.g. [19, 20]. It has been applied by a few authors to software analysis, see e.g. Krone and Snelting, [25]. More recently, Exman and Speicher [12] have shown the equivalence of the Modularity Lattice to the Modularity Matrix, displaying in alternative ways the same modules for any software system.

1.4 Paper Organization

The remaining of the paper is organized as follows. In Sect. 2 a Plausibility Path to conceptual integrity is offered. In Sect. 3 Conceptual Integrity is characterized as an intensive quantity of software. In Sect. 4 Conceptual Integrity is directly calculated from the Modularity Matrix. In Sect. 5 a discussion concludes the paper.

2 A Plausibility Path to Software Conceptual Integrity

We propose a Plausibility Path from an Abstract Domain Conceptualization to Software Conceptual Integrity. We assume that *Conceptual Integrity* pre-exists, in the abstract domain, before being formalized. We provide a general perspective of the plausibility path, leading through the Modularity Matrix to Software Conceptual Integrity. We then focus on each of its steps.

The main idea behind the Plausibility Path is to make plausible transitions between an acceptable starting point – the notion of abstract mathematical domain conceptualization – and the final goal of Software Conceptual Integrity. We call it Plausibility Path since we make acceptable statements in a heuristic fashion, but do not provide rigorous formal proofs.

We shall make formal definitions and corresponding calculation formulas in Sect. 4.

2.1 Plausibility Path Perspective

There are three essential formal steps from Abstract Conceptual Integrity to Software Conceptual Integrity, passing through the Modularity Matrix as shown in Fig. 2.

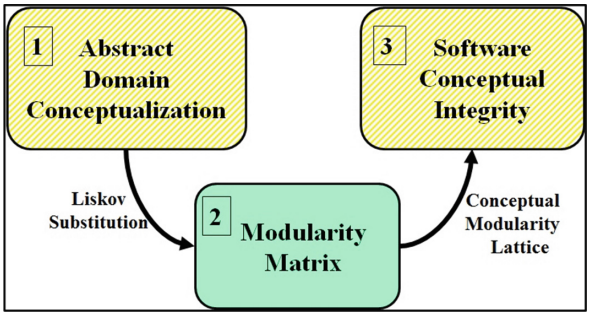

Fig. 2. From an Abstract Domain to Software Conceptual Integrity – The three steps are: the initial "Abstract Domain Conceptualization", the goal "Software Conceptual Integrity", and the intermediate Modularity Matrix. In between there are two transitions: "Liskov Substitution" and "Conceptual Modularity Lattice" to be later explained in the paper text.

The meaning of the three formal steps is as follows:

1. *Abstract Domain Conceptualization* – along the history, concepts in e.g. Mathematics were grouped in fields within hierarchies obeying conceptual integrity;
2. *Modularity Matrix* – the basic algebraic structure of Linear Software Models, plays the role of both a facilitator and formal source for Conceptual Integrity;
3. *Software Conceptual Integrity* – the desired goal of the formalization steps, should assure software system orthogonality and propriety.

The meaning of the two transitions between the above steps is as follows:

1 → 2 – *Liskov Substitution* – translates abstract mathematical concepts into software entities;
2 → 3 – *Conceptual Modularity Lattice* – is an algebraic structure that has been shown to be equivalent to the Modularity Matrix, while obtaining concepts of the software modules.

One can summarize the roles of the above steps, as shown in Fig. 3.

Step	Formal Tool	Goal	Role	Main Theorems
1	Domain Ontologies	**Abstract domain conceptualization**	Classify fields, hierarchies	Common concepts and functions
2	**Modularity Matrix**	Software system design	Source and facilitator	Modularization by spectral methods
3	Orthogonal Algebraic Structure	**Software conceptual integrity**	Assure propriety and orthogonality	Conceptual Integrity complies with Modularization

Fig. 3. Plausibility Path: Tools, Goals and Roles – This summarizes properties of its formal steps in terms of their tools, goals and roles. See detailed discussion in subsequent subsections.

2.2 Software Structure and Behavior

Preliminary definitions clarify each of the above steps. The ultimate goal of the Plausibility Path is conceptual integrity in software systems. We refer to structure and behavior, thinking in terms of software, even when dealing with an abstract domain.

Definition A – *Software Structure*
Software Structure is a relation among software architectural units ("structors", a generalization of classes) involving sub-classing and composition operators.

We use the same operators for software systems and for abstract ontologies. This follows common practice, emphasizing the analogies between abstract concepts and their respective software classes.

Definition B – *Software Behavior*
Software Behavior is the performance of a function computation. The outcome of the function computation is a state change of the software system. We call "functionals" (a generalization of functions) the software architectural units of behavior.

Structors provide Functionals but the latter are not necessarily invoked. One often, by linguistic license, refers to the functionals themselves – without the performance of a computation – as software behavior.

2.3 Abstract Conceptual Integrity

Abstract concepts are hierarchically classified by properties' similarity. The hierarchy determines which concepts are particular cases of other ones. We illustrate the idea with some examples.

A square is a subclass of a rectangle, which is a subclass of a parallelogram. The parallelogram, the most general instance in this small hierarchy (in Fig. 4), is a polygon with four sides, in which the opposite sides are parallel. A rectangle is a subclass of a parallelogram with four right angles. A square is a subclass of a rectangle with all four sides equal.

Each lower hierarchy class has all the properties of the upper classes. A square has 4 sides (as in any quadrilateral), which are parallel (as in the parallelogram), and 4 right angles (as the rectangle).

Hierarchy is also true regarding behavior, i.e. the outcome of the functionals' calculations for each concept (or class). As an example, the perimeter of any class in this hierarchy is obtained by summing the length of the four sides (which in principle may be all different, partially different or all equal).

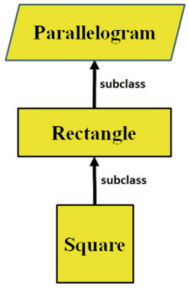

Fig. 4. The Quadrilateral Hierarchy – Each arrow (meaning subclass/subtype of) points from the particular concept (or class) to the more general concept. The parallelogram is the most general class of this hierarchy and Square is the most specific class. We visualize each class with the geometry of the class concept, instead of conventional UML rectangles for all classes.

A different hierarchy could contain a circle as a subclass of an ellipse. A yet different hierarchy would refer to 3-dimensional concepts such as a sphere as a subtype of an ellipsoid.

Each of the three referred hierarchies (quadrilaterals, ellipses, 3-D ellipsoids) display *conceptual integrity*, both intuitively and by some specific well-defined characteristic. For example, all quadrilaterals in Fig. 4 have linear segments as sides of a polygon (literally meaning "multiple angles"), while the ellipses have no linear segments and no angles in between at all the points in their perimeters.

These hierarchies, such as that the quadrilaterals in Fig. 4, are in fact small fragments of an ontology of geometric figures, see e.g. Rovetto [28], which may encompass the three referred hierarchies.

We summarize conceptual integrity in an abstract domain such as mathematics by means of the following Statement:

Statement 1 – *Conceptual Integrity in Abstract Domain Hierarchy of Concepts*
In a class hierarchy determined by sub-classing, in an abstract domain, all the concepts of the hierarchy have at least one common concept, and one common function defined in the most general member of the hierarchy. The common concept and the common function represent the conceptual integrity.

2.4 The Need for Liskov Substitution

We need Liskov Substitution to make the transition from an abstract domain, such as the quadrilaterals in Fig. 4, to actual software entities – say the same quadrilaterals which now have behavior, through their functionals, as represented by the Modularity Matrix. Thus Liskov Substitution attempts to translate, as faithfully as possible, concepts found in Abstract Mathematics to the software domain. This is possible, first of all, since the structure of both ontology fragments (hierarchies) in abstract mathematics and software hierarchies are based upon the same sub-typing operator.

The basic idea of Liskov Substitution which is relevant to *conceptual integrity* is to link "structure" to "behavior", effectively transforming concepts in an abstract (e.g. "mathematics") domain into generic software.

A formulation of Liskov Substitution [27] essentially links sub-classing to the behavior of software containing the relevant classes, when substituted by their subclasses. In such case, the software behavior should not change.

This is particularly interesting, as a "*structural*" class diagram (type T and its subtype S) in Fig. 5 is being linked to a "*behavioral*" condition, which is precisely what transforms an abstract domain to software concepts.

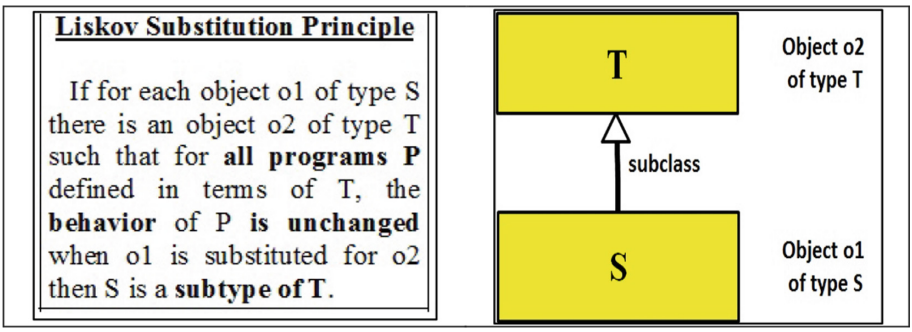

Fig. 5. Liskov Substitution Principle and Class Diagram – The principle is shown in the left-hand-side of the figure. The class diagram illustrating Liskov's principle is in the right-hand-side of the figure. T is a class (or type). S is a subclass (or subtype) of T. Substitution of Object o2 of type T, by an object o1 of type S should not change the software behavior. This diagram is analogous to an abstract hierarchy in Fig. 4. Figure adapted from the paper by Exman [15].

2.5 The Modularity Matrix Roles

The Modularity Matrix of a software system is built of structors preserving the notion of sub-classing. Thus, the Modularity Matrix implicitly conveys the hierarchical ideas formulated in the previous sub-sections.

The contribution of the Modularity Matrix for Conceptual Integrity is to be purposely built to maximize modularity, increasing software simplicity and maximal orthogonality among modules.

Statement 2 – *Conceptual Integrity in the Modularity Matrix*
If the Modularity Matrix is standard (square and block-diagonal), then specific structors provide related functionals within modules, and the modules conceptual integrity is preserved for the restricted set of software systems represented by the Matrix.

2.6 Concepts in the Modularity Lattice

The contribution of the Conceptual Modularity Lattice in the transition from the Modularity Matrix to the software Conceptual Integrity is to link the optimization in terms of Structors and Functionals from the Modularity Matrix to concepts.

This is possible, since the Conceptual Modularity Lattice has been shown by Exman and Speicher [12] to convey information equivalent to the Modularity Matrix, in terms of software system modularity. Moreover, by its very definition from Formal Conceptual Analysis [19] the Conceptual Modularity Lattice is an algebraic structure restricted to the *concepts* relevant to its software system. This is summarized in the following statement.

Statement 3 – *Conceptual Integrity in the Modularity Lattice*
Since the Modularity Lattice in terms of software design is equivalent to its corresponding Modularity Matrix, the concepts fitting to the Matrix modules preserve conceptual integrity and this can be explicitly tested for the restricted set of software systems represented by the Modularity Lattice.

3 Conceptual Integrity as an Intensive Property of Software

In this section we go beyond the principles formulated upon the Modularity Matrix. We present and discuss the idea that Conceptual Integrity is an Intensive property of Software. We explain the meaning of intensive property, give an analogy to physical systems, and deal with software systems.

3.1 Conceptual Integrity Is an Intensive Quantity

Conceptual integrity, besides being a property of a whole hierarchical software system, seems to be a recursive property of each of its subsystems down to basic blocks. If any subsystem does not have conceptual integrity, it is plausible that the whole system cannot display it either.

We now give an example to explain what are intensive versus extensive quantities. Suppose that our system is a vehicle – either a car or a truck. A family car typically has 4 wheels. A truck usually has a bigger number of wheels. The weight of a vehicle is an *extensive* quantity: the weight of a vehicle is the sum of the weights of its parts. For instance, additional wheels increase the weight of the vehicle.

In contrast, the speed of a vehicle is an *intensive* quantity: the speed of the vehicle is not the sum of the speeds of its parts. All the car parts move at the same speed. Specifically, the tangential speed of any of its wheels is the same as the speed of the vehicle, irrespective of the number of wheels.

Conceptual Integrity is an intensive quantity. It is not the sum of the conceptual integrities of the components of a software system.

3.2 Increasing Conceptual Integrity by Exchange of Module Components

Here we use a different physical metaphor as a further illustration for the idea of Conceptual Integrity being intensive.

Assume a system having four sub-systems as in Fig. 6:

1. glass container;
2. water contained by the glass;
3. sphere mostly filled with air partially floating in the water;
4. small solid metal cube inside the sphere.

Now, one heats the glass container by an external heat bath. Heat energy flows among the different sub-systems, from those with higher temperatures to those with lower temperatures, until the whole system reaches a uniform temperature.

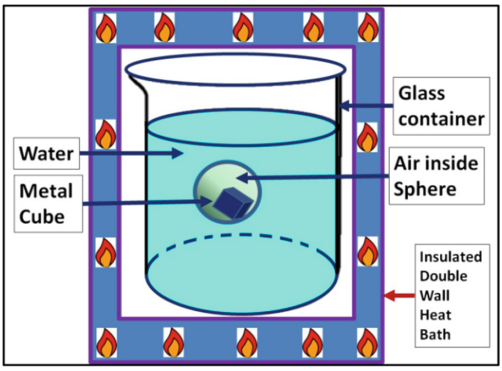

Fig. 6. Physical System Metaphor – The system has 4 sub-systems: a - glass container; b - water inside the glass; c - floating sphere filled with air; d - metal cube inside the sphere. A heat bath heats the glass container until the temperature is uniform, causing heat energy flow among the sub-systems. Figure reproduced from the paper by Exman [15].

In a software system, each sub-system usually has different computation characteristics – one dealing with data, another one with business logic, and so on. Moving concepts (classes) from one sub-system to another may increase conceptual integrity in both sub-systems. One could say that Conceptual Integrity in the whole system is optimized by flow of concepts (classes) among sub-systems. However, such flow of concepts and the intensive hypotheses of conceptual integrity, do not guarantee a single value of conceptual integrity anytime throughout a whole software system.

The previous physical metaphor suggests that *conceptual integrity* is not an extensive property, like heat energy, but an intensive property, like temperature.

4 Direct Computation of Conceptual Integrity from the Modularity Matrix

In this section we first assign a formal definition to two of the quantities behind Conceptual Integrity, viz. Propriety and Orthogonality. The proposed definitions are based on the criteria used to generate an optimized Modularity Matrix by an iterative procedure. Then we provide formulas to directly compute Conceptual Integrity quantities from the generated Modularity Matrix.

4.1 Propriety Formally Defined

Propriety has been verbalized in Subsect. 1.1 as "a software system contains only essential functions". Intuitively, this means that one is minimizing the number of functions. Formally, it is stated as in the following definition.

> **Definition 1: Module Propriety**
> Propriety of a module in the Modularity Matrix, in a certain hierarchical level of a given software system, means that all its structors are mutually linearly independent, and concomitantly all its functionals are mutually linearly independent.

Explaining the previous intuition, the demand of ***linear independence*** among vectors (structors among themselves or functionals among themselves), implies that there are no two identical vectors. Moreover, if a sub-set of vectors are linearly dependent, some of the vectors are superfluous and can be eliminated. The decision of which vectors to eliminate is left to the software engineer, with a good knowledge of the concepts of the software system under development.

Thus, propriety reflects the fact that the Modularity Matrix optimizes – in fact minimizes – the number of structors and their provided functionals, for all its modules. Therefore, the above definition for a module extends to the whole matrix, and the standard Modularity Matrix complies with *Propriety*.

4.2 Orthogonality Formally Defined

Orthogonality was intuitively verbalized in Subsect. 1.1 as "functions are mutually independent". Based upon the Modularity Matrix, this definition has two associated meanings:

- *Linear independence* – among structors and among functionals;
- *Strict orthogonality* – among modules, which is a stronger requirement than linear independence, and is easily visually recognized in the diagonal blocks of the Modularity Matrix, for instance in Fig. 1, when one ignores the outlier.

But, linear independence was already guaranteed by the Propriety definition 1. Therefore, the exact meaning of orthogonality is strict ***orthogonality*** among a specified sub-set of modules formally stated in the following definition.

> **Definition 2: Orthogonality among Modules**
> Orthogonality of a module with respect to a specified sub-set of other modules in the Modularity Matrix, in a certain hierarchical level of a given software system, means that all its structors are orthogonal to all the structors of the other modules in the specified sub-set, and concomitantly all its functionals are orthogonal to all the functionals of the other modules in the same specified sub-set.

The usage of the same term for the principle and for the linear algebra operation is not coincidence. It probably was suggested to conceptual integrity authors by the algebraic notion.

One could now ask about the third principle – Generality. Please see the Discussion Subsect. 5.1 item "d" for considerations on "Generality".

4.3 Direct Computation of Propriety from the Modularity Matrix

Since *Propriety* has been defined in terms of linear independent vectors within a module, it is calculated, according to linear algebra, by the rank $\boldsymbol{r}$ of the sub-matrix of the given module. Specifically, if $\boldsymbol{s}$ is the number of structors (columns) of the module sub-matrix, propriety is calculated by Eq. (1).

$$\boldsymbol{Propriety} = \boldsymbol{1} - ((\boldsymbol{s} - \boldsymbol{r})/\boldsymbol{s}) \tag{1}$$

Note that since module sub-matrices are square, one could use as well the number of functionals $\boldsymbol{f}$ (rows) instead of the number of structors. The module propriety quantity in this equation has a value between zero and the maximum propriety value of 1 obtained when **r** equals **s**. The need for this kind of normalization is to facilitate calculations of propriety for the whole system, given the values for all its modules, while preserving Conceptual Integrity as an intensive property.

4.4 Direct Computation of Orthogonality from the Modularity Matrix

Orthogonality has been defined for all vectors within a module, with respect to vectors in other modules in specified sub-set of modules in the Modularity Matrix. According to linear algebra, orthogonality of a pair of vectors $\boldsymbol{v_{M1}}$ and $\boldsymbol{v_{M2}}$, respectively belonging to modules $\boldsymbol{M_1}$ and $\boldsymbol{M_2}$, is calculated by the scalar product of the pair of vectors in Eq. (2).

$$\boldsymbol{Orthogonality} = \boldsymbol{1} - (\boldsymbol{v_{M1}} \bullet \boldsymbol{v_{M2}}) \tag{2}$$

Note that each of the vectors in this equation is normalized (se e.g. Weisstein [29]), i.e. all their elements are divided by the length of the respective vector. Thus, the calculated orthogonality for a pair of vectors has a value between zero and the maximal value of 1 obtained for zero scalar product. Again the need for normalization is to facilitate calculations of orthogonality for the whole system, given the values for all pairs of structors and all pairs of functionals for all modules, preserving Conceptual Integrity as an intensive property.

5 Discussion

Conceptual Integrity has been considered of fundamental importance for software system design, but has been only vaguely defined.

This paper's basic claim is that the Modularity Matrix is a facilitator and a formal source of Conceptual Integrity information. We have provided two lines of argumentation:

a. *Plausibility Path from Abstract Domains through the Modularity Matrix to Conceptual Integrity* – We started from the accepted conceptual integrity of abstract domains, made a transition to the Modularity Matrix fitting a set of software systems. Using the equivalence to the Modularity Conceptual Lattice, we returned to "conceptual" aspects, to finally reach Conceptual Integrity.
b. *Formal definitions and direct calculation* – The Modularity Matrix optimization procedure was the direct source of the defined quantities viz. propriety and orthogonality, in a formal way.

Two of the principles – propriety and orthogonality – have a neat definition derived from the standard Modularity Matrix properties.

Promising progress has been achieved in this work, but additional investigation, in particular calculation for extensive numbers of case studies is needed to further clarify issues detailed in the next sub-section.

5.1 Open and Controversial Issues

a. Conceptual Independence of Abstract Hierarchies

We have referred in Sect. 2.3 to two independent hierarchies, one of *polygons* and another one of *ellipses*, say a circle. However, they are not strictly independent. One may think of a circle as a regular polygon in the limit of an infinite number of sides,

enabling a transition between two of the above hierarchies. One can easily estimate the value of **π** in the perimeter of a circle **2 * π * Radius** by taking the limit of the perimeter of a polygon inscribed in the circle, when the number of polygon sides goes to infinity.

b. Stability Along Time of Conceptual Hierarchies
The situation is more complex than the naïve static view of Fig. 4 would suggest. Concepts evolve – see e.g. Lakatos [26] – in his book on "Proofs and Refutations" discussing the empirical contribution to the concept evolution of regular polyhedrons (from Euler's initial five). Concepts also can be said to expand along time – see e.g. Buzaglo [5] – according to the terminology of his book "The Logic of Concept Expansion".

c. The Single Brilliant Architect of Major Systems?
Brooks has argued in favor of the idea that only a single brilliant architect, can impart conceptual integrity to a major building, say an architect of a cathedral, or similarly to a major engineering enterprise such as a very large software system. Gabriel [18] challenges Brooks' position.

In our opinion, Brooks' position is difficult to be rationally proven for real systems. But its main drawback is the dependence on the existence and the opportunistic presence of a single brilliant mind. One obviously prefers a systematic construction of formal tools, based upon clear conceptual integrity ideas.

d. The Generality Principle of Conceptual Integrity
Generality, has been described as the quality that "a single function should be usable in many ways" in the same system. This intuitive formulation seems vague enough, being an obstacle to a formal interpretation. We shall return to this issue elsewhere.

5.2 Future Work

Open issues for future work include:

- extensive calculations on actual software systems;
- explanation of difficulties encountered with heavily used software systems such as Git [7].

5.3 Main Contribution

The main contribution of this work is that Linear Software Models – by means of the formal algebraic tools of Modularity Matrix or the Laplacian Matrix – guarantee Conceptual Integrity of the software system they represent.

References

1. Baldwin, C.Y., Clark, K.B.: Design Rules, Volume I. The Power of Modularity. MIT Press, Cambridge (2000)
2. Beynon, W.M., Boyatt, R.C., Chan, Z.E., Intuition in software development revisited. In: Proceedings of the 20th Annual Psychology of Programming Interest Group Conference, UK. Lancaster University (2008)

3. Brooks, F.P.: The Mythical Man-Month – Essays in Software Engineering, Anniversary edn. Addison-Wesley, Boston (1995)
4. Brooks, F.P.: The Design of Design: Essays from a Computer Scientist. Addison-Wesley, Boston (2010)
5. Buzaglo, M.: The Logic of Concept Expansion. Cambridge University Press, Cambridge (2002)
6. Clements, P., Kazman, R., Klein, M.: Evaluating Software Architecture: Methods and Case Studies. Addison-Wesley, Boston (2001)
7. De Rosso, S.P., Jackson, D.: What's wrong with git? A conceptual design analysis. In: Proceedings of Onward! Conference, pp. 37–51. ACM (2013). http://dx.doi.org/10.1145/2509578.2509584
8. De Rosso, S.P., Jackson, D.: Purposes, concepts, misfits, and a redesign of git. In: Proceedings of OOPSLA 2016, Conference, pp. 292–310. ACM (2016). http://dx.doi.org/10.1145/2983990.2984018
9. Exman, I.: Linear software models. In: Proceedings of GTSE 1st SEMAT Workshop on a General Theory of Software Engineering. KTH Royal Institute of Technology, Stockholm, Sweden, (2012). http://semat.org/wp-content/uploads/2012/10/GTSE_2012_Proceedings.pdf
10. Exman, I.: Linear software models, video presentation of paper [9] (2012). http://www.youtube.com/watch?v=EJfzArH8-ls
11. Exman, I.: Linear software models: standard modularity highlights residual coupling. Int. J. Softw. Eng. Knowl. Eng. **24**, 183–210 (2014). https://doi.org/10.1142/S0218194014500089
12. Exman, I., Speicher, D.: Linear software models: equivalence of modularity matrix to its modularity lattice. In: Proceedings of 10th ICSOFT International Conference on Software Technology, ScitePress, Portugal, pp. 109–116 (2015). https://doi.org/10.5220/0005557701090116
13. Exman, I.: Linear software models: decoupled modules from modularity matrix eigenvectors. Int. J. Softw. Eng. Knowl. Eng. **25**(8), 1395–1426 (2015). https://doi.org/10.1142/S0218194015500308
14. Exman, I., Sakhnini, R.: Linear software models: modularity analysis by the Laplacian matrix. In: Proceedings of ICSOFT 2016 11th International Joint Conference on Software Technologies, vol. 2, pp. 100–108 (2016). https://doi.org/10.5220/0005985601000108
15. Exman, I.: The modularity matrix as a source of software conceptual integrity. In: Proc. SKY'2016 7th International Workshop on Software Knowledge, ScitePress, Portugal, pp. 27–35 (2016). https://doi.org/10.5220/0006098300270035
16. Exman, I., Katz, P.: Conceptual software design: algebraic axioms for conceptual integrity. In: Proc. SEKE 2017, 29th International Conference on Software Engineering and Knowledge Engineering, pp. 155–160 (2017). http://dx.doi.org/10.18293/SEKE2017-148
17. Exman, I., Sakhnini, R.: Linear software models: bipartite isomorphism between Laplacian eigenvectors and modularity matrix eigenvectors. Int. J. Softw. Eng. Knowl. Eng. **28**(7), 897–935 (2018). https://doi.org/10.1142/S0218194018400107
18. Gabriel, R.P.: Designed as designer. In: Essay Track, ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages and Applications, Montreal, Canada (2007). http://dreamsongs.com/DesignedAsDesigner.html
19. Ganter, B., Wille, R.: Formal Concept Analysis: Mathematical Foundations. Springer, Berlin (1998). https://doi.org/10.1007/978-3-642-59830-2
20. Ganter, B., Stumme, G., Wille, R.: Formal Concept Analysis - Foundations and Applications. Springer, Berlin (2005). https://doi.org/10.1007/978-3-540-31881-1
21. Jackson, D.: Conceptual design of software: a research agenda. CSAIL Technical report, MIT-CSAIL-TR-2013–020 (2013). http://dspace.mit.edu/bitstream/handle/1721.1/79826/MIT-CSAIL-TR-2013-020.pdf?sequence=2

22. Jackson, D.: Towards a theory of conceptual design for software. In: Proceedings Onward! 2015 ACM International Symposium on New Ideas, New Paradigms and Reflections on Programming & Software, pp. 282–296 (2015). https://doi.org/10.1145/2814228.2814248
23. Kazman, R.: Tool support for architecture analysis and design. In: ISAW 96 Proceedings of 2nd International Software Architecture Workshop, pp. 94–97. ACM, New York (1996). https://doi.org/10.1145/243327.243618
24. Kazman, R., Carriere, S.J.: Playing detective: reconstructing software architecture from available evidence. Technical report CMU/SEI-97-TR-010, Software Engineering Institute, Carnegie Mellon University, Pittsburgh (1997)
25. Krone, M., Snelting, G.: On the inference of configuration structures from source code. In: Proceedings of ICSE-16 16th International Conference on Software Engineering (1994). https://doi.org/10.1109/icse.1994.296765
26. Lakatos, I.: Proofs and Refutations: The Logic of Mathematical Discovery. Cambridge University Press, Cambridge (1976)
27. Liskov, B.: Keynote address - data abstraction and hierarchy. ACM SIGPLAN Not. **23**(5), 17–34 (1988). https://doi.org/10.1145/62139.62141
28. Rovetto, R.: The shape of shapes: an ontological exploration. In: Proceedings of SHAPES 1.0 1st Interdisciplinary Workshop on Shapes, Karlsruhe (2011)
29. Weisstein, E.W.: "Normalized vector" from MathWorld–a wolfram web resource (2018). http://mathworld.wolfram.com/NormalizedVector.html