Software System Theory of the Forbidden Within Discrete Design

Iaakov Exman(✉)

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

Abstract. Many "theoretical" frameworks have been proposed for software systems design with a plethora of techniques, scopes and degrees of sophistication. However, a clear delineation of the *forbidden* in software design terms is almost universally absent in all these frameworks. This absence is surprising, as other engineering disciplines obviously display forbidden regions. This paper claims that an acceptable software design theory should clearly demarcate the forbidden in contrast to the possible. Algebra is argued to be the mathematical field appropriate to determine boundaries of forbidden regions. To this end, a spectral approach is demonstrated, in which matrix eigenvectors play a central role. Such boundaries of forbidden regions are illustrated by a case study.

Keywords: Software theory · Forbidden regions · Forbidden domains · Boundary · Algebra · Eigenvectors · Models · Discrete software design · Hierarchical software systems

1 Introduction

Forbidden regions surrounding possible solutions to engineering problems are ubiquitous in mature engineering disciplines and their underlying basic sciences. A central claim of this work is that Software Engineering deserves such a theory implying a definition and delineation of forbidden regions or domains. This is true in particular for software embedded in larger systems, which without forbidden domain restrictions may cause critical failures and endanger human life.

An aeronautical engineering example is a case that really happened above the Atlantic Ocean, of an airplane flying at high altitude that inadvertently entered a weather storm area. The pilots should have tried to either totally avoid the well-known storm area, or once inside that area to escape the storm from below, gaining speed through the airplane descent. Instead the inexperienced pilots, tried to climb above the storm causing an increasing loss of speed. The final outcome was the free fall of the airplane in the middle of the ocean. This is just one example that there are clearly *forbidden maneuvers* for any given aircraft – dictated by aerodynamics theory – that may result in total loss of control with unfortunate consequences.

Another example, this one in civil engineering, is the famous Tower of Pisa. The tower was obviously planned to be vertical, with a nice view of the surrounding area. Along time, it gradually became inclined. Without reinforcements, it would continue to

E. Cabello et al. (Eds.): ICSOFT 2016, CCIS 743, pp. 264–283, 2017.
DOI: 10.1007/978-3-319-62569-0_13

increase its angle relative to the vertical axis, and finally fall. The *statics* theory, an old branch of physics, states the laws of the relevant *forbidden region*. Roughly one says that, the tower cannot be more inclined than some angle, in which the projection of the tower center of mass passes a threshold distance of the building ground basis.

There are uncountable cases of forbidden regions in science and technology. We shall, later on, motivate forbidden regions in software, by two examples of physics.

1.1 Models of the Possible are *not* a Theory

We wish to emphasize, early in this paper, that models of the possible are not a theory; therefore they do not produce forbidden regions. The widely used UML (Unified Modeling Language) diagrams [25] are an example of software design models and not a theory. They can be indefinitely modified at will by software engineers while developing a software system. They impose no restrictions, and do not highlight design problems, since they do not imply any design quality criteria.

The same occurs with code in a programming language, say Java or Python. These languages are below a suitable level of abstraction to generate software design criteria. Thus, the currently used compilers help in eliminating language syntax bugs, but otherwise allow indefinite program variations.

1.2 Forbidden Domains are Essential for a Software Theory

The main thrust of this paper – which is an update and extension of the paper by Exman [11] – is the claim that forbidden regions or domains are essential for a Software Theory. This is based on the following assumptions:

- *Hierarchical software system composition problem* – a theory of software composition should solve the design problem of a hierarchical software system, through increasingly simpler subsystems, down to indivisible components;
- *Existence of forbidden region boundaries* – these boundaries should restrict composition variability, for pragmatic reasons, which limit the search effort in design space, and for more fundamental reasons, such as keeping conceptual integrity, facilitating system development, comprehension and maintenance.
- *Formal algebraic criteria* – representing software systems by matrices, enables the full power of linear algebra formalism, obtaining boundaries determined by quality of design criteria, viz. suitable eigenvectors of those matrices.

1.3 Related Work

This concise literature review focuses into two topics, forbidden regions and linear algebra. It omits non-software references, such as “forbidden transitions” in pure physical systems.

a- Forbidden Regions

The notions of forbidden regions or forbidden domains have appeared in several contexts in the scientific literature, with differing meanings. The common idea of all the contexts is the existence of a problem sub-space where a solution cannot be found. We provide here just a limited sample of papers specifically referring to algorithms in embedded and/or pure software systems.

Aneja and Parlar [2] describe transportation-related algorithms for optimal single facility location problems with forbidden regions. These regions are those were location is not permitted, but one can travel through them, such as a lake. Wu et al. [27] estimate answer sizes for XML queries by excluding forbidden regions and assuming some distribution over the remainder of a two-dimensional diagram.

A whole area of embedded systems referring to forbidden regions is that dealing with robots. Abbot et al. [1] discuss ways of preventing robot manipulators to enter forbidden regions of a workspace. Payandeh and Stanisic [17] state that in order to train a novice operator of a robotic manipulator, one may define "forbidden regions virtual fixtures" (FRVF); when an operator moves the manipulator in these regions, a graphical clue can be generated, a force feedback can be generated or an embedded command in the FRVF can maintain the robot at a safe configuration.

Devadas and Aydin [4] discuss real-time dynamic power management in which they explicitly enforce device sleep intervals, the so-called forbidden regions. The goal is to enhance energy savings. This is done by time-demand analysis, which determines duration and frequency of forbidden regions to preserve the temporal correctness of all the tasks. They show that the problem of generating feasible schedules for preemptive periodic real-time tasks in which all device sleep intervals are longer than the device break-even times, is NP-Hard in the strong sense.

b- Linear Algebra

Matrices of several types have been used to analyze software design, in which an essential feature is a spectral approach using matrix eigenvectors, delimiting "forbidden regions". For comparisons about the applicability of the referred matrices to software design, we refer the reader to e.g. Exman [8] and Exman and Sakhnini [9].

Besides the Modularity Matrix one finds the Laplacian matrix [26], see e.g. Exman and Sakhnini [9], the Design Structure Matrix (DSM), see e.g. Sullivan et al. [24], and the affinity matrix, see e.g. the work by Li and Guo [15].

1.4 Organization of the Paper

The remainder of the paper is organized as follows. Section 2 deals with forbidden domains in physical systems, to motivate the later sections which refer to software systems. Section 3 introduces the software algebraic theory that we use. Section 4 describes a generic design algorithm dealing with forbidden regions. In Sect. 5, we discuss a case study, focusing on a design pattern, to illustrate the theory and the idea of forbidden regions. A discussion in Sect. 6 ends the paper.

2 Sources of Forbidden Domains: Physical Metaphors

In this section we deal with sources of forbidden domains within two physical realms. These serve as metaphors motivating the software theory to be introduced in the next section. One metaphor refers to transverse standing waves generated with a Slinky toy. The other refers to wave-functions of the so-called "particle in a box".

2.1 The 1st Physical Metaphor: Standing Waves in a Slinky

Slinky is a toy made of a pre-compressed helical spring – cf. e.g. Slinky [20, 21]. It has been used for concrete and intuitive demonstrations of properties of physical waves. Here we focus on transverse waves.

Let us imagine the following experiment. A slinky is stretched horizontally on the floor – or on a table – by two persons, grasping its end-points. If both persons move their hands laterally, in parallel to the floor (see the arrows in Fig. 1), but perpendicularly to the slinky axis, each person generates *transverse waves* travelling towards the other person. In Fig. 1 a stretched slinky depicts an **S** due to the schematic grasping hands lateral motion.

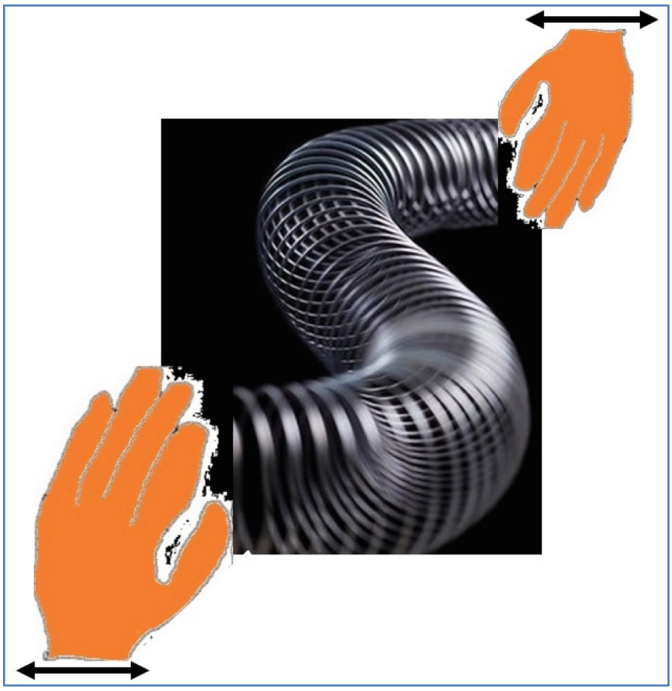

Fig. 1. Slinky transverse wave – two schematic hands move laterally a slinky, the helical spring. The motion, in the direction shown by the arrows, is perpendicular to the slinky axis. The hands' oscillation back and forth generates a transverse wave with an **S** form.

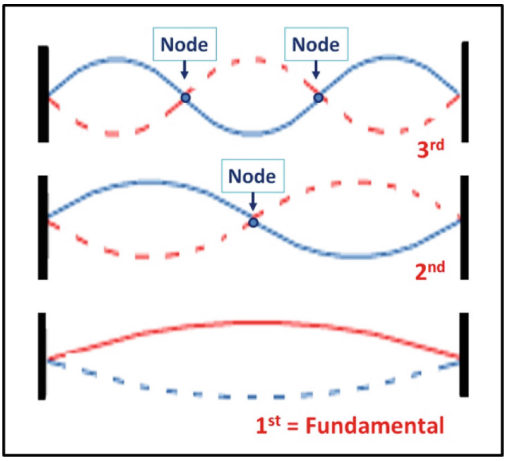

Fig. 2. Transverse standing waves in a slinky bounded by its end-points – the dashed lines show the amplitude of wave oscillation in each slinky point. Three permitted wave modes are shown: 1- The 1st fundamental (lowest) wave, the mode number is 1 and it has no nodes; 2- The 2nd wave mode number is 2, because it has two half-waves with a node in between; its continuous (blue) line is a full sinusoidal wave since it has zero vibration at the left-most point, goes up before the node, goes down after the node and returns to zero at the right-most point; 3- The 3rd wave mode number is 3 and is divided by two nodes. Fractional mode numbers are forbidden (see text). Figure adapted from Ref. [11]. (Color figure online)

Once the hands' motions of both persons are synchronized, *standing waves* are obtained. Standing waves divide the slinky in an integer number of equal parts, delimited by the *nodes* (see Fig. 2). These nodes are fixed in space and do not oscillate, despite the fact that overall, the slinky is oscillating as a whole. Oscillation modes are characterized by *mode numbers*, i.e. the number of sinusoidal half-waves of the vibration, explained in Fig. 2. Thus, we state the limitations imposed on slinky motions and the oscillation modes forbidden by the physical nature of the motions:

- ***Boundaries*** *on slinky behavior* – besides the material and geometry of the slinky itself, the nature of the *boundaries*, be they fixed walls or hands in motion, is the most significant limitation of behavior;
- ***Forbidden*** *slinky oscillation modes* – *Standing waves* can be obtained only for integer mode numbers; fractional mode numbers are forbidden by the destructive interference of waves travelling in opposite directions.

The slinky toy is very intuitive and its demonstration is easily reproduced. Dynamic views of oscillating standing waves can be seen in a graphical simulation – as shown by the standing wave in [22] – and in a video – see e.g. standing waves on a slinky in [23].

2.2 The 2nd Physical Metaphor: Wave-Functions of a Particle in a Box

Our 2nd metaphor, the particle in a box, is a gradual intermediate transition from the slinky physical metaphor (in the previous Subsect. 2.1), to the software theory to be described in the next Sect. 3. The two physical metaphors have in common the same "wave" solutions, as seen below. This 2nd metaphor and the software theory in Sect. 3 have in common that solutions are obtained by means of eigenvectors, either from an eigenvalue equation here, or from the eigenvectors of a matrix in the software theory.

This 2nd metaphor demands deeper physics knowledge to fully understand its details. But this should not discourage a reader which is not familiar with this specialized knowledge. The reader may skip the details; they are not essential to understand the overall meaning of this example. For a gradual elementary introduction to the subject, the reader may look at [18].

The "particle in a box" is a simple quantum mechanics' problem – see e.g. Messiah [16]. The particle has mass $\boldsymbol{m}$. The so-to-speak box is one-dimensional!, has finite length ℓ, and zero potential. The particle is confined and cannot escape the two bounding walls with infinite potential.

The problem to be solved is an eigenvalue problem. In such a problem, when a matrix or operator H multiplies an eigenvector vk one obtains back the same eigenvector multiplied by a constant, the respective eigenvalue λk. This has the form:

$$H \cdot vk = \lambda k \cdot vk \tag{1}$$

Specifically this is the Schrödinger equation in which H is the Hamiltonian operator, and the k^{th} eigenvector vk fits the eigenvalue λk, standing for an energy value. As the potential inside the box is null, the particle Hamiltonian reduces just to a Laplacian. Some solutions of this problem, the *wave functions*, are seen in Fig. 3.

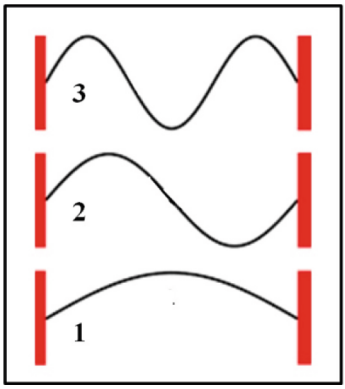

Fig. 3. Wave functions of the particle in a one-dimensional box – these waves are indexed by integers, seen in the above diagram. Just three of the possible solutions are shown in this figure. Intermediate energy values are forbidden. Note that their form is identical to the slinky waves of Fig. 2. Figure adapted from Ref. [11].

The meaning of the *wave functions* of the particle in a box problem is certainly different from the slinky transverse standing waves, but their form (seen in Fig. 3) is identical to the slinky waves (in Fig. 2) for similar reasons. Also here the wave functions vanish in the confining walls.

The conclusions from this 2nd metaphor are analogous to the slinky ones:

- ***Boundaries*** *of behavior of a Particle in a box* – besides the mass of the particle itself the *boundaries*' nature, i.e. the infinite potential in the fixed walls and the box length ℓ, is the most important behavior limitation;
- ***Forbidden*** *energy values* – Wave functions, the solutions to the eigenvalue problem, have discrete energy values, indexed by integers; other energy values are forbidden.

2.3 Common Features of Both Physical Metaphors

The two physical metaphors described in this section have very different underlying physical systems. A slinky is a real macroscopic toy made of metal or plastic materials, obeying classical mechanics, which can be hand held, stretched and oscillated. A particle in a box is a microscopic system obeying quantum mechanics, and rather serves as a thought experiment useful to demonstrate the simplest quantum system enclosed between walls.

Nonetheless, there are some striking similarities between these two kinds of systems. These common features are abstracted from the conclusions we extracted from each of them:

- ***Boundaries*** *on behavior* – the nature of the *boundaries* of these systems, either fixed walls or hands in motion, have a significant influence on the system behavior. For instance, a particle confined by finite (instead of an infinite) potential has a very different behavior.
- ***Forbidden*** *non-solutions* – The actual solution *waves* obtained are discrete and can only be indexed by integer numbers. For instance, fractional number indices are forbidden by the system constraints.

We shall compare these characteristics with those of software systems design in the next sections. It turns out that software systems also have clear similarities to these physical systems.

3 A General Software Theory of the Forbidden

A general Software Theory of the Forbidden is presented in this section in two parts: a set of basic axioms and mechanisms to delimit forbidden domains from algebraic structures which describe design sub-spaces for the desired software system.

3.1 Basic Axioms

The starting point of our software theory is a couple of basic axioms formulated in the two next text-boxes and explained in the paragraphs following the axioms.

Axiom 1 – Bounded Design Space

The design space of any particular software system is composed of a discrete and finite number of components.

The *Design Space* of a software system, in terms of numbers of components, is larger than the actual final design of the desired software system. The former contains all the potential components for that system. On the other hand, the final output design of a certain software system is obtained by the search results within the Design Space limited by the boundaries of the forbidden domains. *Components* are used here in the generic sense of Sect. 1.2. They are either subsystems in a given hierarchical level or the smallest indivisible parts of the system.

Axiom 2 – Hierarchical Design Sub-spaces

The design space of any particular software system is hierarchical, with each design sub-space corresponding to an abstraction level of the hierarchy. Adjacent sub-spaces are related by collapsing/expanding operations.

Any particular software system is assumed to be composed as a hierarchy of abstraction levels. One goes up in the hierarchy by collapsing sub-systems into a higher system level. One goes down the hierarchy by expanding a higher system level into sub-systems in the lower level, recursively until one reaches the lowest indivisible components. Likewise, the Design Space of a software system is hierarchical, with sub-spaces corresponding to the abstraction levels.

These axioms are needed for two purposes:

a. ***Design Space Bounded size*** – to assure that the search process for the final Software System output design is efficient;
b. ***Software System Comprehensibility*** – comprehensibility is a far reaching demand from the Software System design. It concerns system development, maintenance, improvement and fundamental principles, as conceptual integrity – see Brooks [3], Jackson [14], and Exman [12].

The two axioms are needed for design efficiency, since just discreteness and finiteness of design space, in the 1st axiom, are not enough to guarantee a small enough space for efficiency. The 2nd axiom is necessary, as we still envision the design process

as not fully automatic. Automated computation is alternated with human intervention, justifying the importance of comprehensibility.

3.2 Algebraic Structures and Forbidden Software Compositions

Design Sub-Spaces and the final Software System design at a certain hierarchical abstraction level are both represented by an algebraic structure. Typically such algebraic structure is a matrix – say the Modularity Matrix, see e.g. Exman [7] – or a Laplacian Matrix – see e.g. Exman and Sakhnini [9]. The algebraic structure may also be a graph obtained from a matrix, for instance the algebraic structure of a Modularity Lattice – see e.g. Exman and Speicher [10] – or a bipartite graph which originates and is intimately linked to the Laplacian Matrix. In this paper we focus on matrices.

The physical metaphors of Sect. 2 are clearly suggestive of our software theory of the forbidden. Its most important characteristics are as follows:

a. ***Boundaries around a software system and its modules*** – the Software System boundaries idea is ubiquitous in object oriented software, and known as *encapsulation*. An outer boundary separates the software system from its environment. The inner boundaries separate system modules from each other.
b. ***Forbidden compositions are delimited by matrix Eigenvectors*** – the above referred boundaries imply forbidden regions. Eigenvectors fitting certain eigenvalues of the chosen matrices delimit forbidden compositions. One still needs conjunction with a formal definition of cohesion – see e.g. Exman [8] and Exman and Sakhnini [9]. The final design discrete components are determined by suitable elements of the relevant eigenvectors.
c. ***Outliers in forbidden regions eliminated by redesign*** – outlier matrix elements in forbidden regions point out to undesirable couplings between modules. These should be eliminated by software system redesign, usually done by human intervention of software engineers.

These characteristics are put together in a software system design algorithm, capable to deal with forbidden regions. This algorithm is presented in the next section, in pseudo-code format.

4 Generic Design Algorithm with Forbidden Regions

In this section we present our algorithm with boundaries capable of excluding forbidden regions. This is a generic algorithm displayed in pseudo-code. In order to design an actual software system, one must first choose a specific matrix type, say Modularity Matrix or Laplacian matrix. Then, suitable specific procedures should be applied to select eigenvalues and get modules from their respective eigenvectors.

The generic algorithm consists of four phases:

a. ***Initialize*** a matrix and a cohesion threshold;
b. ***Search Loop*** calculating eigenvalues and corresponding eigenvectors, to obtain modules;

c. ***Check modules' cohesion*** whether they comply with the threshold;
d. ***Redesign*** if indicated by outliers.

The generic design algorithm is shown in the next text-box.

Generic Design Algorithm – with Forbidden Regions

Init:
Design Sub-Space = obtain suitable matrix;
Set lower cohesion threshold;

Search Loop – obtain modules:
While (there are low cohesion modules)
Do {
Obtain matrix eigenvalues/eigenvectors;
Select suitable eigenvalues;
Pick corresponding eigenvectors;
Get modules from eigenvector elements;
Calculate modules' cohesion;
Forbidden boundary – cohesion check:
If (module cohesion < threshold)
{split module;
Repeat while loop}
Else
End While}

Forbidden region redesign:
If (outlier left)
{Redesign matrix as needed;}

Cohesion is calculated by the inverse of the sparsity of a module (which is itself a sub-matrix). The sparsity is the ratio of zero-valued matrix elements to the total number of matrix elements in the matrix or sub-matrix. A typical sparsity threshold is 50%. Modules should have high-cohesion (low sparsity). The environment, i.e. matrix elements outside modules, should display low cohesion (high sparsity).

5 Case Study: Boundaries of the Forbidden

In this section we describe the well-known *Command* design pattern, given in the GoF (so-called "Gang of Four") book by Gamma et al. [13] as a case study. The main goal here is to illustrate the boundaries of the forbidden. The design pattern is first presented in terms of the UML class diagram. Next, we demonstrate the *Generic Design Algorithm* of Sect. 4, in a series of steps, starting with the chosen Modularity Matrix – see Exman [7]. We could as well choose a Laplacian Matrix – see Exman and Sakhnini [9]. The specific steps, following the algorithm are:

a. *Obtain a matrix* – we choose the Modularity Matrix to represent the design pattern; such a matrix is symmetrized and weighted by means of an affinity;
b. *Get eigenvalues/eigenvectors* – use the suitable approach for the chosen matrix; for a Modularity Matrix the eigenvectors are listed in decreasing order of their respective eigenvalues; then one takes the highest eigenvectors that completely span the matrix size;
c. *Obtain the module sizes* – from the respective eigenvector positive elements;
d. *Illustrate the case of an outlier* – by intentionally adding an arbitrary matrix element coupling two modules; this shows how one deals with elements in the forbidden region;
e. *Collapse sub-systems* – to illustrate the hierarchy of the Software System levels.

5.1 The Command Design Pattern – Its Class Diagram

The goal of the Command design pattern is to enable abstraction of commands, say in a text editor application. It decouples an object that invokes an action, by clicking a *Save* menu-item, from another object that actually performs the file saving action. Moreover, the Command pattern enables generic features such as Undo and Redo, independently of whether the specific action is a saving or printing a file.

Figure 4 shows a class diagram of the Command design pattern. It is similar to the Command class diagram of this pattern in the GoF book (see Gamma et al. [13], p. 233). The Command pattern in this class diagram has the following classes:

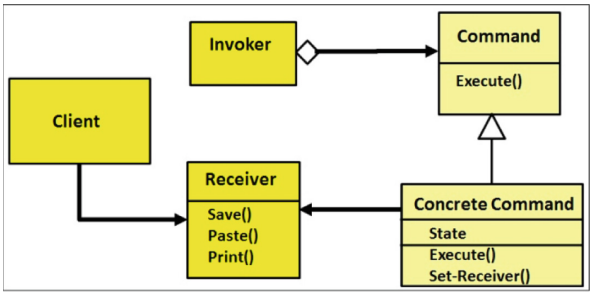

Fig. 4. Command Design pattern UML Class Diagram – the invoker, a menu-item or button, once clicked triggers commands execution. The Concrete-Command inherits the abstract Command class (both with a beige color) and actually executes an action on the Receiver (a document). Figure adapted from Ref. [11]. (Color figure online)

- An invoker, say a menu-item or button, to be clicked in order to activate execution of a command;
- An abstract Command which characterizes the pattern;
- A Concrete Command class inheriting the abstract Command to actually execute a specific command;
- A Receiver, which represents an abstraction of a document file.
- A client, which in fact does not belong specifically to this design pattern.

Design patterns, declared as reusable software architectural units, could be expected to have well-defined standard forms. But any such standard does not exist.

The Command section of the GoF book (Gamma et al. [13]) displays no less than four different class diagrams of this pattern, besides the pattern generic diagram similar to Fig. 4. The situation is even worse when considering the Internet literature on design patterns and their implementations in a variety of programming languages.

As was stated in Sect. 1.1, UML allows indefinite variability for any software system. It is a flexible design model, *not* a theory with forbidden regions. Therefore, the next logical step is to translate the Command class diagram into a Modularity Matrix, enabling a theory which limits forbidden regions.

5.2 Boundaries by the Modularity Matrix

A Modularity Matrix (see Exman [5–7]) was chosen for the Command pattern case study linking structors (generalizing classes) to provided functionals (generalizing methods). The standard Modularity Matrix, by the Linear Software Models is square and block-diagonal. The Command pattern Modularity Matrix displayed in Fig. 5 is indeed square and block-diagonal.

Structor →		Command	Concrete Command	Client	Invoker	History	Receiver
Functional ↓		S1	S2	S3	S4	S5	S6
execute	F1	1	1				
Set-receiver	F2	0	1				
Create-objects	F3			1	0	0	
Bind-cmd	F4			1	1	0	
Undo	F5			0	1	1	
Receiver-action	F6						1

Fig. 5. Command Design pattern Modularity Matrix – it is square and block diagonal with 6 Structors (columns) and Functionals (rows). Diagonal blocks (blue background) are modules: Top-Left = essential Command pattern roles, with structors S1, S2 and functionals F1, F2 designations (again marked by beige color); Middle = generic classes; Bottom-Right = Receiver. Zero-valued elements outside the modules are omitted for simplicity. Figure adapted from Ref. [11]. (Color figure online)

A Modularity Matrix, containing only the system structors and functionals, establishes a boundary between the software system and its environment. The diagonal blocks also set well-defined boundaries among modules.

5.3 Eigenvectors Delimit the Forbidden Regions

A spectral approach applied to the Modularity Matrix has been developed [8] to find the software system module sizes and eventual outliers, based upon the matrix eigenvectors and eigenvalues. The approach is formally described by an eigenvalue equation, entirely analogous to Eq. (1) in Subsect. 2.2:

$$M \cdot vk = \lambda k \cdot vkZ \tag{2}$$

M is a symmetrized and weighted Modularity Matrix; vk stands for the k^{th} eigenvector of M; the eigenvector vk fits to its eigenvalue λk. Symmetrizing and weighting details by an affinity expression are not essential to understand the arguments and conclusions of this paper. More details can be found in the paper by Exman [8].

The Command pattern Modularity Matrix eigenvectors and eigenvalues are shown in Fig. 6. The eigenvalues are sorted in decreasing order. One can easily verify that the positive eigenvector elements in the first three eigenvectors span the whole matrix. These eigenvector elements correspond to the module sizes shown in Fig. 5. The elements of these eigenvectors have only zero-valued or positive values, in contrast to the remaining eigenvectors.

	Modules					
Eigenvectors	**1**	**2**	**3**	**4**	**5**	**6**
	0	0.707	0	0	0.707	0
	0	0.707	0	0	-0.707	0
	0.5	0	0	-0.707	0	-0.5
	0.707	0	0	0	0	0.707
	0.5	0	0	0.707	0	-0.5
	0	0	1	0	0	0
Eigenvalues →	**1.52**	**1.367**	**1**	**1**	**0.633**	**0.48**

Fig. 6. Command pattern eigenvectors/eigenvalues – the three first eigenvectors fitting the first three eigenvalues – to the left of the red vertical separator – span the Modularity matrix *modules*. Positive eigenvector elements (blue background) obtain the matrix module sizes in Fig. 5. Here the module sizes fit the eigenvalues' order. Figure adapted from Ref. [11]. (Color figure online)

Had we chosen a Laplacian Matrix (see Exman and Sakhnini [9]) instead of the Modularity Matrix to solve our case study, the generic eigenvalue Eq. (2) would still be valid. On the other hand, the specific eigenvalues and eigenvectors would be different, as well as their meaning and the way to obtain the module sizes. These Matrix specifics are not essential for the understanding the results of this paper. The generic approach,

viz. the fact that eigenvectors delimit the boundaries of the forbidden regions, is the important message.

5.4 Redesign to Eliminate Forbidden Outliers

The treatment of existing outliers can be illustrated by intentionally adding a 1-valued matrix element to the block diagonal matrix of the Command pattern in Fig. 5 as follows. The outcome matrix in Fig. 7 has an added element, in row F2 and column S3, which indeed is an outlier. The latter element couples the *upper-left* module (overlapped by row F2) with the *middle* module (overlapped by column S3), while itself being outside the borders of both these modules.

The outlier in Fig. 7 is revealed by the *Forbidden boundary – cohesion check* within our Generic Design Algorithm (in Sect. 4) as follows:

1. ***The eigenvector module size*** – it fits a large module of size 5 * 5 which is the result of coupling of the *upper-left* module of size 2 * 2 with the *middle* module of size 3 * 3;
2. ***The cohesion of the large module*** – is too low, as it has a total of 16 zero-valued elements, 5 inside the coupled modules and 11 in the forbidden regions of the environment of these modules, viz. in rows F3 to F5 below the upper-left module and in columns S3 to S5 above the middle module. Its sparsity is then calculated as 16/25 = 0.64, which is higher than the threshold of 50%. Thus this larger coupled module must be split.

Structor →		Command	Concrete Command	Client	Invoker	History	Receiver
Functional ↓		S1	S2	S3	S4	S5	S6
execute	F1	1	1				
Set-receiver	F2	0	1	1			
Create-objects	F3			1	0	0	
Bind-cmd	F4			1	1	0	
Undo	F5			0	1	1	
Receiver-action	F6						1

Fig. 7. Command pattern Modularity Matrix with outlier – this is the matrix in Fig. 5, with an added outlier element in row F2 and column S3 (with dark blue hatched background). Zero-valued matrix elements outside the modules are omitted for clarity. Figure adapted from Ref. [11]. (Color figure online)

Our Generic Design Algorithm determines that outliers – 1-valued matrix elements in forbidden matrix regions – i.e. outside the diagonal modules, should be eliminated and the matrix redesigned.

5.5 Hierarchical Sub-spaces of the Command Design Pattern

We finally illustrate the meaning of the Hierarchical Design Sub-Spaces of Axiom 2 in Subsect. 3.1. The modules of the Modularity Matrix of the Command design pattern, in Fig. 5, have its structors and functionals explicitly shown. Each of these three modules may be collapsed into the next higher level of the hierarchy for this system, to obtain the Modularity Matrix in Fig. 8. This is a 3 * 3 matrix. Expanding this higher level matrix into the next lower level, obtains back the matrix in Fig. 5.

Structor →		Command roles	Generic Classes	Receiver
Functional ↓		S1	S2	S3
Execute command	F1	1		
Generic function	F2		1	
Receiver Action	F3			1

Fig. 8. Collapsed high-level Modularity Matrix of the Command Design pattern – modules of Fig. 5 were collapsed to single matrix elements: Top-Left = essential Command pattern roles; Middle = generic classes; Bottom-Right = Receiver of the action. Zero-valued matrix elements are omitted for clarity. Figure adapted from Ref. [11].

Performing one further collapsing operation into the highest level of the Command pattern hierarchy, one obtains the Modularity matrix in Fig. 9, which is a 1 * 1 matrix.

Structor →		Command Design pattern
Functional ↓		S1
Execute Command	F1	1

Fig. 9. Collapsed highest-level Modularity Matrix of the Command Design pattern – modules of Fig. 8 were collapsed into a single matrix element.

The whole hierarchy of the Command design pattern, viz. the upper-level system, the next level sub-systems and the lower-level sub-sub-systems, is shown in Fig. 10, to

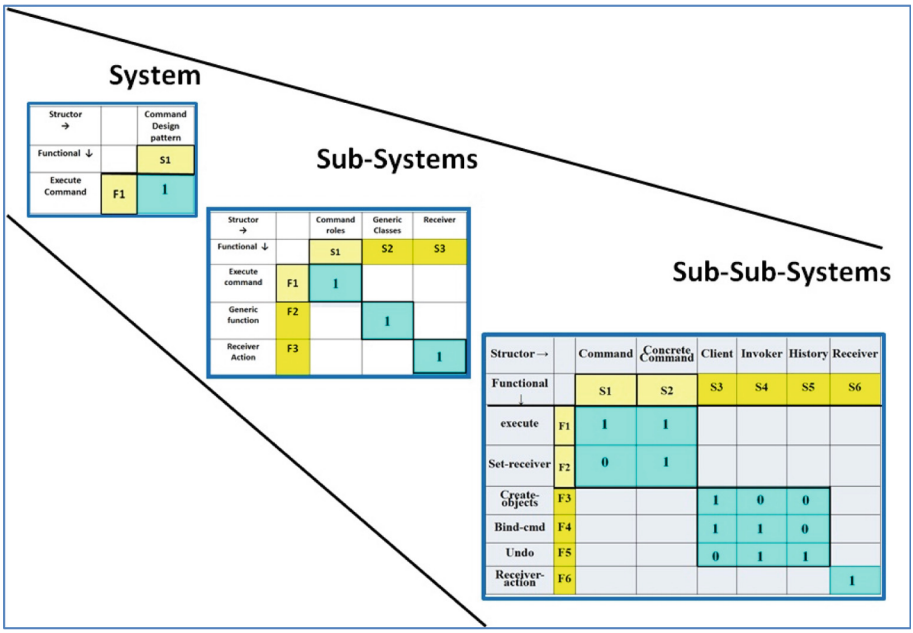

Fig. 10. Hierarchical System of Command Design pattern – the three Modularity Matrices in the Command design pattern hierarchy. Top-Left: fully collapsed system; Middle: collapsed sub-systems; Bottom-right: fully expanded sub-sub-systems to the resolution of Fig. 5.

illustrate the idea of a hierarchical software system. Note that this is the final designed system. The Hierarchical Design Sub-Spaces have the same pyramidal structure, in which each abstraction level corresponds to a sub-space, which in turn is represented by one Modularity Matrix.

6 Discussion

This paper has shown, motivated by physical systems' metaphors, that generic formal quality criteria for software system design are provided by Linear Algebra, within the theory of Linear Software Systems. Here we discuss the nature of such criteria and why they are essential.

6.1 The Theoretical Importance of Forbidden Regions

The idea of focusing on Forbidden Regions is somewhat surprising, since apparently one would most probably prefer an emphasis on positive rather than negative design criteria. But the purpose of real theories is to *simultaneously* provide positive and negative criteria. When one says that some decisions are desirable, one is concomitantly saying that other decisions are undesirable.

Positive criteria declare that some design space regions lead to desirable properties of a software system, such as modularity.

What is the theoretical importance of Forbidden Regions?

Their importance is to call the attention of software engineers to design problems, say undesirable coupling between software modules, which must be solved. Thus, Forbidden Regions are clear signals that the design process is still not fully accomplished.

6.2 Formalization of the Design of Artificial Systems

In principle one could refute the validity of the physical metaphors, since there is no special reason to assume that physical systems and software systems behave analogously. One must provide further arguments with this respect.

A first argument is the existence of a common body of knowledge covering both natural and artificial systems, which justifies similar treatment of both kinds of systems. An example, referring to the science of aerodynamics, is that designed artificial systems, be it an airplane or the software embedded in its computers, behave to a large extent like natural systems. Citing Herbert Simon from his book *The Sciences of the Artificial* [19] (in p. 7): "Given an airplane, or given a bird, we can analyze them by the methods of natural science without any particular attention to purpose or adaptation…".

A second argument is the hierarchical structure which is common to social systems, natural systems and software systems. Referring to software systems, we find it so important for this paper, that we explicitly stated hierarchy in Axiom 2, and in the explanations surrounding this axiom in Subsect. 3.1. Again citing Herbert Simon's book [19] (in p. 184): "…my central theme is that complexity frequently takes the form of hierarchy and that hierarchic systems have some common properties independent of their specific content." This is further and thoroughly discussed by Simon in Chap. 8 "The Architecture of Complexity: Hierarchic Systems" of the same book.

The issue of the validity of physical metaphors for software systems is much deeper, but space limitations of this paper prevent us to embark in the broader discussion that this issue deserves.

6.3 Why Eigenvectors?

Eigenvectors are important for dimensionality reduction. Specifically, in the context of software engineering, as shown in Sect. 5.3 of this paper, they enable modularity in a software system represented by a Modularity Matrix, or alternatively by a Laplacian matrix, in each of the abstraction levels of the hierarchical software structure.

Eigenvectors reduce and simplify the set of vectors describing a software system. Accordingly, the corresponding modules have the effect of reducing a large software system to a smaller set of sub-systems that are easier to comprehend.

Formally, software system modularity implies lack of dependence among different modules. In terms of Modularity matrices – e.g. the matrix in Fig. 5 – modules are

mutually independent since each module is composed by a set of structors and functionals which is disjoint to the sets of structors and functionals of all other modules.

Modules exactly reflect the eigenvectors' mutual orthogonality. Eigenvectors – e.g. the first three in Fig. 6 in Subsect. 5.3 – have zero-valued pairwise scalar products. The same is true for any structor from a given module: it is orthogonal to structors belonging to any other modules of the same Modularity Matrix. Mutatis mutandis, any functional from a given module is mutually orthogonal to any functional belonging to other modules of the same Modularity matrix.

6.4 Search Efficiency Issues

The 1st axiom on the Software System Design Space in Subsect. 3.1 of this paper, tells that the Design Space is discrete and finite. It does not guarantee that the Design Space is small. Search in the Design Space could still take a long time.

The 2nd axiom in Subsect. 3.1 – demanding a hierarchical Design Space for a software system – is the basis of an intuitive argument for the claim that, while the overall Design Space for the whole system may not be small, the Design Sub-Space for each subsystem in any level in the Design Space hierarchy is expected to be of bounded size.

For instance, looking at each abstraction level of our case study – the Command Design pattern shown in Fig. 10 – one sees that the maximal size of each module is bounded by a 3 * 3 matrix. In general, one expects for a multi-level hierarchy of a large system, that in each abstraction level the subsystem matrix size is bounded by a small integer. In other words, design space search in each module is efficient for all hierarchy levels.

6.5 Main Contribution

This paper claims that real software system theories need to be of practical use for software design. Such theories should provide formal design quality criteria, supporting system modularity. Concomitantly these theories should point out to *forbidden system compositions*, signaling undesired modules' coupling in need of software system redesign.

References

1. Abbot, J.J., Marayong, P., Okamura, A.M.: Haptic virtual fixtures for robot-assisted manipulation. In: Thrun, S., Brooks, R., Durrant-Whyte, H. (eds.) Robotics Research. Springer Tracts in Advanced Robotics, vol. 28, pp. 49–64. Springer, Berlin (2007). doi:10.1007/978-3-540-48113-3_5
2. Aneja, Y.P., Parlar, M.: Algorithms for weber facility location in the presence of forbidden regions and/or barriers to travel. Transp. Sci. **28**(1), 70–76 (1994). doi:10.1287/trsc.28.1.70

3. Brooks, F.P.: The Mythical Man-Month - Essays in Software Engineering – Anniversary. Addison-Wesley, Boston (1995)
4. Devadas, V., Aydin, H.: Real-time dynamic power management through device forbidden regions. In: Proceeding IEEE Real-Time and Embedded Technology and Applications Symposium, pp. 34–44 (2008). doi:10.1109/RTAS.2008.21
5. Exman, I.: Linear software models, extended abstract. In: Jacobson, I., Goedicke, M., Johnson, P. (eds.) Proceeding. GTSE 2012, SEMAT Workshop on a General Theory of Software Engineering, pp. 23–24. KTH Royal Institute of Technology, Stockholm (2012)
6. Exman, I.: Linear Software Models, GTSE 2012, SEMAT Workshop on a General Theory of Software Engineering. KTH Royal Institute of Technology, Stockholm (2012). Video presentation of Ref. [5]: http://www.youtube.com/watch?v=EJfzArH8-ls
7. Exman, I.: Linear software models: standard modularity highlights residual coupling. Int. J. Softw. Eng. Knowl. Eng. **24**(2), 183–210 (2014). doi:10.1142/S0218194014500089
8. Exman, I.: Linear software models: decoupled modules from modularity matrix eigenvectors. Int. J. Softw. Eng. Knowl. Eng. **25**(8), 1395–1426 (2015). doi:10.1142/S0218194015500308
9. Exman, I., Sakhnini, R.: Linear software models: modularity analysis by the Laplacian matrix. In: Proceeding 11th International Joint Conference on Software Technologies, ICSOFT-PT, vol. 2, pp. 100–108, Lisbon, Portugal (2016). doi:10.5220/0005985601000108
10. Exman, I., Speicher, D.: Linear software models: equivalence of modularity matrix to its modularity lattice. In: Proceeding 10th ICSOFT International Joint Conference on Software Technologies, Colmar, France, pp. 109–116 (2015). doi:10.5220/0005557701090116
11. Exman, I.: Software theory of the forbidden in a discrete design space. In: Proceeding 11th International Joint Conference on Software Technologies, ICSOFT-PT, vol. 2, pp. 131–137. Lisbon, Portugal, SciTePress (2016a). doi:10.5220/0006004601310137
12. Exman, I.: The modularity matrix as a source of software conceptual integrity. In: Proceeding SKY 2016 - 7th International Workshop on Software Knowledge, Porto, Portugal, pp. 27–35. SciTePress (2016b)
13. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Addison-Wesley, Boston (1995)
14. 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
15. Li, X.-Y., Guo, L.: Constructing affinity matrix in spectral clustering based on neighbor propagation. Neurocomputing **97**, 125–130 (2012). doi:10.1016/j.neucom.2012.06.023
16. Messiah, A.: Quantum Mechanics, vol. I. North-Holland Publishing Co., Amsterdam (1961). Chap. III, Reprinted by Dover Publications (2014)
17. Payandeh, S., Stanisic, Z.: On application of virtual fixtures as an aid for telemanipulation and training. In: Proceeding HAPTICS 2002 10th Symposium on Haptic Interfaces for Virtual Environment and Teleoperator Systems, pp. 18–23 (2002). doi:10.1109/HAPTIC.2002.998936
18. Particle in a Box. https://en.wikipedia.org/wiki/Particle_in_a_box
19. Simon, H.A.: The Sciences of the Artificial, 3rd edn. MIT Press, Cambridge (1996)
20. Slinky (2016a). https://en.wikipedia.org/wiki/Slinky
21. Slinky, Wave Phase changes at fixed end (2016b). http://hyperphysics.phy-astr.gsu.edu/hbase/sound/slinkv.html#c1
22. Standing wave (2016a). https://upload.wikimedia.org/wikipedia/commons/7/7d/ Standing_wave_2.gif
23. Standing wave, Standing waves on a Slinky (2016b). http://hyperphysics.phy-astr.gsu.edu/hbase/sound/slnksw.html#c1

24. Sullivan, K.J., Griswold, W.G., Cai, Y., Hallen, B.: The structure and value of modularity in software design. In: Proceeding ESEC/FSE 8th European Software Engineering Conference and 9th SIGSOFT International Symposium Foundations Software Engineering, pp. 99–108. ACM (2001). doi:10.1145/503209.503224
25. UML, Specification, OMG (Object Management Group) (2015). http://www.omg.org/spec/UML/
26. Weisstein, E.W.: Laplacian Matrix, From Mathworld–A Wolfram Web Resource (2016). http://mathworld.wolfram.com/LaplacianMatrix.html
27. Wu, Y., Patel, J.M., Jagadish, H.V.: Estimating answer sizes for XML queries. In: Jensen, C.S. et al. (ed.) EDBT 2002. LNCS, vol. 2287, pp. 590–608. Springer, Heidelberg (2002). doi:10.1007/3-540-45876-X_37