Software, Is It Poetry or Prose?
Conceptual Content at the Higher Abstraction Levels

Iaakov Exman[1] and Alessio Plebe[2]
[1]*Software Engineering Dept., The Jerusalem College of Engineering – JCE - Azrieli, POB 3566, Jerusalem, Israel*
[2]*Department of Cognitive Science, University of Messina, Messina, Italy*

Keywords: Software Theory, Conceptual Content, Runnable, Understanding, Poetry, Prose.

Abstract: Software is it Poetry or Prose? It is part Poetry, part Prose. But it has much more in common with both forms of natural language, than usually admitted: software concepts, rather than defined by syntactic oriented computer programming languages, are characterized by the semantics of natural language. This paper exploits these similarities in a two-way sense. In one way the software perspective may be relevant to the analysis of natural language forms, such as poems. In the other way round, as its central message, this paper uses properties of both Poetry and Prose to facilitate a deeper understanding of highest-level software abstractions.

1 INTRODUCTION

This paper takes the position that before proposing a theory of software engineering one must understand the nature of software itself. Thus, this work focuses on theoretical implications of natural language aspects of highest-level software abstractions. It was triggered by a dialogue of the authors during a festive dinner at a conference in Rome last year and continued by electronic means. The dialogue inspired by the style of Galileo (Galilei, 1632) is partially transcribed below as an introduction to the issues at stake in this work.

1.1 Dialogue Concerning Two Chief Software Views

Alessio – I may agree that in some sense software is deeper than Chomskian theories, for me especially in that Chomsky made a sharp division between syntax and semantics, moving almost all the burden of language on the syntax side. It is not, I think, the way (natural) language works. Of course I'm not the only one on this position. Most exponents of the so-called cognitive linguistics enterprise challenged the syntactocentricsm of generative grammar. One of the first was George Lakoff, a former student of Chomsky, who, trying to find examples of linguistic expressions supporting the alleged autonomy and independence of syntax, found so many counterexamples instead, to become convinced of the contrary (Lakoff, 1986). He became one of the leading exponents of cognitive linguistics, together with Langacker (Langacker, 1987), Fauconnier (Fauconnier, 1997) and several others. But maybe you have in mind other reasons why the Chomsky account of language is limited with respect to software.

Iaakov – Since I like gedanken (thought) experiments (Brown, 2011) as much as the cremeschnitte we ate at the dinner, I ask you to imagine the following experiment. Assume that from the birth of a person until age of fifteen one is supposed to learn the mother tongue and use it strictly according to grammatical, syntactic rules. From the age of fifteen until the age of twenty one gradually uses words with the same meaning as before, but more and more liberated from grammatical, syntactic rules. From the age of twenty onwards one is totally free to speak poetry instead of prose. Since the meaning of a word does not follow from grammar, but may be assumed to be dependent on a context – defined by an ontology – in such a world Chomskian theories would be unimportant. So is software, less grammar, more and more concepts.

Alessio – Your second issue is the analogy with poetry, I must say I didn't caught it in Rome, maybe now I can understand a bit more. Repeating a poem to myself (mentally or aloud) corresponds somehow

Exman, I. and Plebe, A..
Software, Is It Poetry or Prose? - Conceptual Content at the Higher Abstraction Levels.
In *Proceedings of the 6th International Workshop on Software Knowledge (SKY 2015)*, pages 9-17
ISBN: 978-989-758-162-5
Copyright © 2015 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

to "executing" it. That's interesting. As the execution of software will affect hardware components, registers, memory content and so on, the "execution" of a poem will elicit responses, in emotional brain centers, recall long-term memories, activate semantic networks. In both cases the meaning of the code (poetic or software) is in the activation resulting from its execution. It sounds fine. Of course, one may raise several possible objections. For example: what is special for poetry in this analogy? Wouldn't be similar when reading a novel, or a newspaper article? Maybe I'm still far from catching completely the intents of your analogy.

Iaakov – Poetry is paradoxical. On the one hand, it is more constrained by structures. On the other hand, it is less grammatical, more audacious. This is like object oriented software, more structured, and freer in conceptual terms. But we are also aware of other software assets which are prose-like.

Alessio – Now, I come to my point. What I shared with you was a thought I had since long, but never articulated in detail: the possibility that the road taken by computation toward software in the 60's has been the result of the influence of Chomsky. It had the consequence of a paradigm shift from mathematics to linguistics. In the early years of computation, it was entirely within mathematics, with central concepts like Dedekind's recursive functions. Turing devised its foundational machine in 1937 as a contribution to Hilbert's Entscheidungsproblem (Turing, 1937). Even the introduction of "compilation" by Grace Murray Hopper in 1953 (Hopper, 1953) was totally unrelated with linguistics. It was only after the publication of Chomsky's "Syntactic Structures" in 1957 (Chomsky, 1957), and his huge success, that John Backus (Backus, 1959) and John McCarthy (McCarthy, 1960) launched the concept of "programming languages", hinging at large inside the Chomskian tools: generative grammar, tokenization, parsing, translation, and so on.

Iaakov – I get your point.

Alessio – I remember you objecting that history does not go with alternatives, it has no sense in imagining a different destiny if contingencies were different. That is correct, but my point is not historical, is more ontological. There is an underlying widespread assumption, that software is actually a sort of language, not a spoken language, but one that follows the same rules of natural languages, because it *is* a language in its essence. I'm not saying that this is wrong belief; I'm just saying that it is a belief rooted in the contingencies of history – see also (Nofre, 2014), for a sociological account of this historical contingency – which could be true or possibly wrong.

Iaakov – It is easier for me to agree with an ontological point than with a historical one. I would prefer to state that software proper – the runnable part, not the requirements and other assets – is in essence a complex semantic structure, rather than a language. It is runnable meaning.

Alessio – The two alternatives do not affect anything with regard to how efficient is to treat computation using linguistic tools. But it is clearly important when dealing with the ontological status of software.

Iaakov – I would add a cautious caveat. Efficiency has more to do with the underlying machine, than with the runnable meaning itself.

Alessio – Good, I will stop for now... Let me just add that this sort of conversation is quite new for me. But I'm interested in continuing, and it touches two sides of my interests: from one side, the philosophy of computing and on the other side linguistic meaning. I'll do it with pleasure.

1.2 Paper Organization

The remaining of the paper is organized as follows. Section 2 deals with software as Poetry, section 3 with refactoring Poetry, section 4 with software as Prose, section 5 with Conceptual Software. The paper ends with a discussion and conclusion.

2 SOFTWARE AS POETRY

Here we point out to features that poetry has in common with software – see also (Gabriel, 2008) for another paper analysing poetry, having in mind software. We display poems in diagrams as if we were describing a kind of software.

2.1 Poetry Has Structure

From the earliest to most modern samples, poems have structure. Fig. 1 displays a modern sonnet by Edna St. Vincent Millay (Millay, 1921). It has four stanzas, with respectively 4, 4, 3 and 3 verses, and classical rhymes, e.g. in the 1st stanza, *ended* rhymes with *extended,* and *all* rhymes with *fall*.

Sonnet - *E. St. Vincent Millay*

Only until this cigarette is ended,
A little moment at the end of all,
While on the floor the quiet ashes fall,
And in the firelight to a lance extended,

Bizarrely with the jazzing music blended,
The broken shadow dances on the wall,
I will permit my memory to recall
The vision of you, by all my dreams attended.

And then, adieu,—farewell!—the dream is done.
Yours is a face of which I can forget
The colour and the features, every one,

The words not ever, and the smiles not yet;
But in your day this moment is the sun
Upon a hill, after the sun has set.

Figure 1: SONNET by E. St. Vincent Millay – Classical structure of a modern poem with four stanzas, with 4, 4, 3 and 3 verses, displaying classical rhymes at the end of verses.

To make the comparison of poetry with software more concrete, we treat this poem as a piece of software, providing its UML "class diagram". Each stanza is assumed to be a different class. This is seen in Fig. 2. If the reader is not familiar with UML (Booch, 2005), (OMG, 2015), one can think it as an ontology graph containing concepts (classes).

Structure reflects meaning, thus class names were chosen as the most meaningful word in each stanza. Class attributes are significant nouns, and the class functions are the significant verbs. Inheritance links classes with related themes. Association links a class with the previous one, of which it is aware. The overall sonnet class diagram resembles a typical software design pattern – like Observer or Mediator (Gamma, 1995).

2.2 Poetry Has Metaphors

A metaphor is a figure of speech in which a term refers to an object or action that it does not literally denote, implying a resemblance. Metaphors are a common way to generate new meanings of words, indeed new words and idioms.

A neutral dictionary definition of cigarette is just a smoking device: a small roll of finely cut tobacco for smoking, enclosed in a wrapper of thin paper. But if one reads just the first stanza in St. Vincent Millay's sonnet, a cigarette resembles a peculiar device to measure time – by its gradual shortening due to the falling ashes. It also implies momentary memories of the end of an affectional relation – if one reads the third and fourth stanzas.

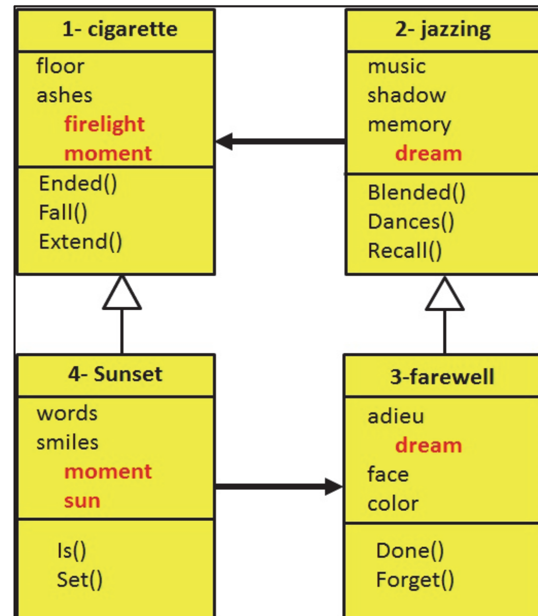

Figure 2: UML CLASS DIAGRAM of the previous sonnet – each class (in a yellow rectangle) corresponds to one sonnet stanza. Class names are numbered by stanzas order. The 1st one is "cigarette". The middle part of each class contains "nouns" (attributes). The lowest part contains "verbs" (functions). The vertical white triangle arrowheads denote inheritance, i.e. similar themes. Indeed the bold red words are common to pairs of classes. The horizontal black arrowheads denote associations, i.e. a class aware of the previous one.

Similarly with the so to speak sunset, of the faraway sun, the closest star to planet earth which is actually rotating around the sun. Here the sun represents a less peculiar device to measure a short time and perhaps human vanity.

2.3 Poetry Is Runnable

Running a poem is to read it in one's head or aloud, once and again, to understand its contents. Running means understanding.

The statechart in Fig. 3 shows the actual reading steps – as transitions between states – of the above poem by one of the authors of this paper in order to be satisfied by his understanding of the poem. The starting point is the upper-left arrow out of the black circle. The reader decides about the ending point that could be at the *sunset* state, but not necessarily.

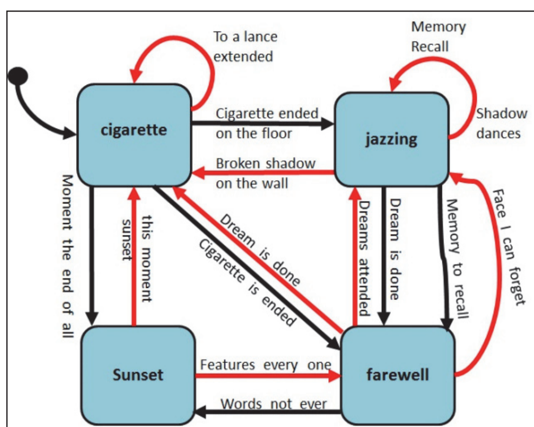

Figure 3: RUNNING (reading in one's head) THE SONNET by E. St. Vincent Millay – In each of the four states one reads the respective stanza (class). One may proceed to the next state, to a related state or return to a previous state. The reason for a specific transition is written close to the transition arrow: this reason may be either a difficulty to be solved or just an associative link. For instance, in the upper transition from *cigarette* to *jazzing* one has "cigarette ended on the floor". Here "on the floor" is associatively related to "on the wall" in the opposite direction transition.

3 REFACTORING POETRY AS IF IT WERE SOFTWARE

In this section we analyse a poem which illustrates a case of analogy to software in which one needs to refactor (Mens, 2004), (Fowler, 1999) classes.

3.1 Interrupted Stanzas

Poems may have shorter than expected stanzas, which convey meaning by their very interruptions. Fig. 4 displays a poem – named Edge – written by Sylvia Plath (Plath, 1963). This poem has modern characteristics, such as quite free structure and the absence of rhyme.

Plath's poem structurally has 10 very short stanzas of 2 verses each. But after some "poem running" in one's head, one perceives that many of the sentences of the poem are broken into consecutive verses. For instance let us look at the sentence:

Her bare
Feet seem to be saying:

It starts in the second verse of the third stanza (*Her bare*) and continues in the fourth stanza (*Feet seem to be saying:*). There is a whole blank line between the stanzas strongly suggesting a deliberate interruption – as part of the poem significance – despite the fact that the word *Feet* begins with a capital letter insinuating that it starts a new sentence.

Thus, the structure of the poem – in particular the many interruptions – conveys semantics. Further examination of the poem links these interruptions to its main meaning.

Searching the Web – e.g. (Wikipedia, 2015) – one finds that *Edge* was written a short time before Sylvia Plath's relatively young age suicide – a drastic intentional interruption of her life. Given this information, the title *Edge*, its broken sentences, and the overall poem gets a possible meaning.

3.2 Refactoring Poetry Classes

If one persists in the one-to-one correspondence between stanzas and UML classes, one would obtain ten classes for Plath's *Edge* poem class diagram.

In Fig. 5 one can see the corresponding class diagram of Plath's *Edge* poem. It contains just three classes – named by their most significant words – with attributes and functions given by considerations similar to those that lead to the diagram in Fig. 2.

Why three instead of ten classes?

Resuming the "poem running" in one's head, one finds that in spite of the broken sentences, one has a clear sense of continuity among groups of stanzas. Paradoxically, the interruptions are rather links between consecutive stanzas. One can divide the latter into three groups.

The first group has stanzas 1 to 4. The common subject is the woman and her body. It is perfected, an accomplishment, and a Greek tragedy brings it beyond the *Edge*, it is over.

The second group has stanzas 5 to 8. The garden may be associated with a kindergarten, with the Garden of Eden (the serpent and the woman), the flowers and odors.

The third group with just two stanzas 9 and 10, where a distant – detached – moon is staring at the tragedy, but "nothing is to be sad about", she (the moon? the woman?) is used to this sort of thing.

So, instead of many too short stanzas, we refactor the poem classes into just three consistent ones.

Refactoring classes, see e.g. (Mens, 2004), (Fowler, 1999) is a software technique which is based upon semantics (and also efficiency) considerations. Its aim is to facilitate comprehension of the software system, for diverse purposes, such as maintenance, reuse, and so on.

Edge – *Sylvia Plath*

The woman is perfected.
Her dead

Body wears the smile of accomplishment,
The illusion of a Greek necessity

Flows in the scrolls of her toga,
Her bare

Feet seem to be saying:
We have come so far, it is over.

Each dead child coiled, a white serpent,
One at each little

Pitcher of milk, now empty.
She has folded

Them back into her body as petals
Of a rose close when the garden

Stiffens and odors bleed
From the sweet, deep throats of the night flower.

The moon has nothing to be sad about,
Staring from her hood of bone.

She is used to this sort of thing.
Her blacks crackle and drag.

Figure 4: "EDGE" POEM by Sylvia Plath – It has ten very short stanzas of only two verses. One perceives broken sentences, which rather link stanzas into three groups.

A potentially interesting usage of the analogy between poetry and software is for educational purposes. It should be an instructive exercise for software engineering students, to practice concurrent factorization of poems and software systems and their respective class diagrams. This would emphasize and clearly illustrate the importance of semantics for software.

4 SOFTWARE AS PROSE

Software assets other than runnable programs, e.g. requirements, specification documents or user's guides, are essentially like prose. Let us see their common properties.

4.1 Prose Is Easily Readable

By prose we mean all texts – literary, journalistic or professional – excluding the following types: poetry, text containing formulas (e.g. mathematical or chemical), texts very specialized like philosophical ones. The immediate quality of prose is to be easily readable, as people are used to speak and write in prose.

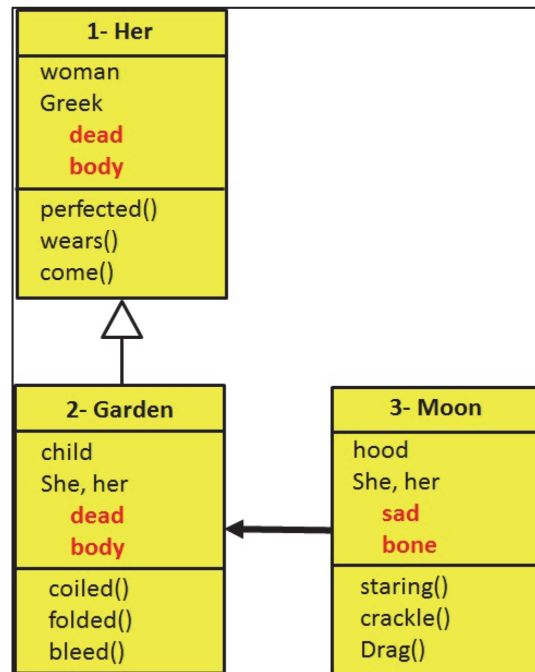

Figure 5: UML CLASS DIAGRAM of the Edge POEM – each class (a yellow rectangle) corresponds to a group of stanzas (see text). Class names are numbered by groups order. The 1st one is "Her". The middle class part has "nouns" (attributes). The lowest part has "verbs" (functions). The vertical white triangle arrowhead denotes inheritance (similar themes): the bold red words are common to pairs of classes. The horizontal black arrowhead denotes association (a class aware of the previous one).

4.2 Prose Also Has Metaphors

On the other hand, prose texts have varying degrees of sophistication. Thus all the complex characteristics of natural language – say ambiguity, synonymy, use of metaphors etc. – may appear in ordinary prose texts, eventually demanding deeper comprehension.

5 SOFTWARE AS CONCEPTUAL CONSTRUCTS

Software systems are hierarchical. From bottom up,

one roughly encounters assembly language the closest to a machine, next a high-level language as Java, then UML and on top a small set of ontologies relevant to the system domains. From the ontology concepts (classes) one can derive UML classes. Thus the highest-level of abstraction is conceptual content, closest to human beings.

5.1 Conceptual Content of Abstract Factory

In order to illustrate the idea of conceptual content of software, we use a software design pattern class diagram as a case study.

The class diagram of the *Abstract Factory* software design pattern (Gamma, 1995) is seen in Fig. 6. The purpose of this design pattern is to provide an interface to create families of related objects, without specifying their concrete classes.

A general observation, when looking at this class diagram, is that none of its concepts is part of the vocabulary (the reserved words) of a programming language. So they are not intrinsic software artefacts. Let us now specifically examine some of these concepts. We start with "*Abstract Factory*". Both words "abstract" and "factory" appear in the English dictionary as independent words with specific meanings, having no necessary relation to software. The word "abstract" has been incorporated with a specific meaning into software. It denotes a special kind of class.

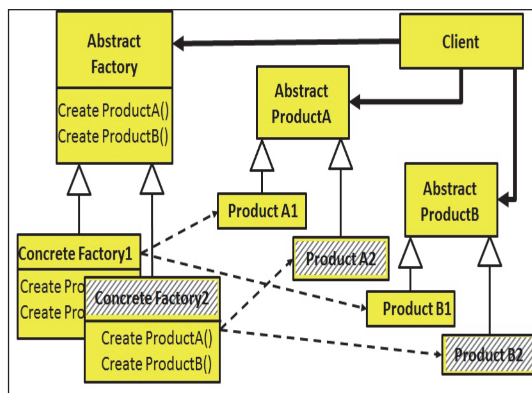

Figure 6: CLASS DIAGRAM OF THE ABSTRACT FACTORY DESIGN PATTERN – The abstract factory class may have any number (here 2) of concrete factory sub-classes. Each concrete factory has the same family of products (here Product A and Product B). For instance Concrete Factory 2 has two products Product A2 and Product B2. Concrete Factory 2 and its products have class names with hatched background.

5.2 Software Also Has Metaphors

The concept "*Abstract Factory*" is a metaphorical usage of natural language terms incorporated into software. If one looks carefully at the Abstract Factory class diagram, indeed ***all*** the concepts result from metaphorical usage.

"Create" is another such example, meaning construction of an object in an object oriented programming language. A "factory" is thus a class whose main purpose is to construct "objects".

Moreover, "*concrete factory*" is a typical example of ambiguity. It is not a factory that fabricates concrete or itself made of concrete (in the dictionary: *concrete* is a strong construction material made of sand, conglomerate gravel, in a cement matrix).

Even the explanation of the purpose of the abstract factory design pattern using the term "families" of related objects extends by analogy the meaning of a non-software word, viz. "family".

Summarizing our claim, the highest-level abstraction of software is as set of concepts that may be extracted from domain ontologies and included in UML class diagrams. Since these concepts are ordinary natural language words, all the complexities of natural language, such as ambiguity, synonymy, figures of speech as metaphors, are fundamental for the understanding, development and maintenance of software, and therefore should be part of a basic theory of software.

Are these complexities so fundamental for software? The immediate answer is that if software itself should automatically manipulate software, these complexities should be resolved.

5.3 A Metaphor Design Pattern

Given the fact that metaphors are so common in the highest abstraction levels of software, we propose a generic *Metaphor* design pattern. The purpose of this design pattern is as follows: easily change or addition of a new context for a given term with multiple meanings.

This proposed design pattern is modelled after the *Strategy* pattern (Gamma, 1995), with a sort of inversion of roles. In the Strategy pattern a context is fixed and strategies are variable. In the Metaphor pattern a given term is fixed and its contexts providing different meanings are variable.

The class diagram of this proposed Metaphor pattern is shown in Fig. 7. Its generic classes are:

- *Metaphor* – it contains the several meanings of a given term; it receives a term meaning as input

and sets its specific context;

- *Term* – it is an (abstract) interface declaring a SetContext() function and corresponding Actions();
- *Contexts* – these are concrete classes with different domains, each say given by an ontology and its specific actions.

An example of a term is "*bridge*". This term has numerous different but metaphorically related meanings given by the respective contexts: e.g. civil engineering, odontology, card games, and even design patterns.

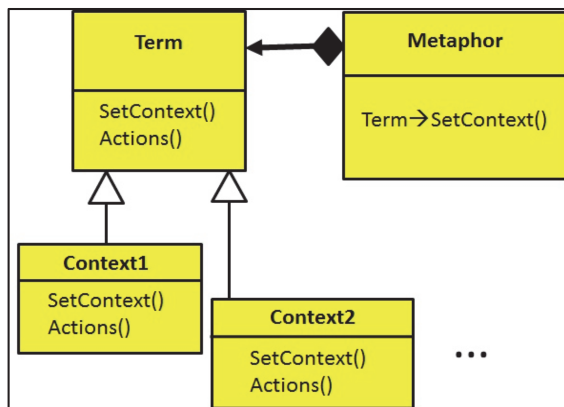

Figure 7: CLASS DIAGRAM OF A METAPHOR DESIGN PATTERN – In a metaphor a single term is fixed and its various meanings are set by variable contexts. In this diagram only two contexts are shown, but they imply that any number of contexts can be added.

6 DISCUSSION

This discussion deals with foundational issues, practical implications for software and theoretical implications.

For the purposes of this discussion, instead of referring in separate specifically to the similarity of software either to poetry or to prose, we jointly refer to both under the rubric of software *natural language* conceptual issues.

Before we embark in the discussion proper, it should be pointed out that this paper stressed metaphors. But different works also refer to other figures of speech. For instance, (Noble, 2004) takes care to differentiate metaphor from metonymy, especially in the context of software design patterns. In contrast to our approach, they refer to the pattern functionality and not to the terms naming the pattern classes.

6.1 Foundational Issues

Significant issues have been opened in the literature concerning software conceptual contents. Is software semantics intrinsic (inner) or extrinsic (outer)? In other words, is software semantics just given by the inner workings of the computing machine or is it deeply related to the human conceptual (outer) world?

White mentions various obstacles to solve this symbol grounding problem (White, 2006): e.g. the difficulty to assign a clear boundary between inside and outside of the computer system. Piccinini argues for "computation without representation" (Piccinini, 2006), i.e. instead of semantics, meaning of symbols and states is given by functional properties of computational systems. According to Smith's notion of participatory computation (Smith, 2002), any physical computing system is inherently situated in its environment in a manner in which its processes extend beyond the physical boundaries of the system, which stands in semantic relations to distal states of affairs.

Another foundational issue refers to *universality*. Do the conceptual contents of software affect only restricted classes of software systems? Absolutely not: the referred characteristics of natural languages – ambiguity, synonymy, figures of speech such as metaphoric usage for new word invention – not only are extensive at a given time, but also are expected to persist along time. They are inherent to the vitality and evolution of natural languages.

In this work we provided case studies to make plausible that the highest software abstraction levels do not refer at all to the machine and its semantics works much like in ordinary human language. This will be discussed at length in a subsequent paper.

6.2 Conceptual Software: Praxis

From the praxis criterion viewpoint, explicit consideration of conceptual contents of software enables various ways of software system development in an ontology-oriented fashion – see e.g. (Pan, 2013), (Exman, 2013). There also exist tools for improvement of software system modularity in terms of conceptual analysis (Lindig, 1997), (Ganter, 1999), (Exman, 2015).

The practical importance of semantic considerations of the natural language concepts found in the higher abstraction levels of software, as opposed to the dominantly syntactic concerns in lower abstraction levels – viz. code in programming languages – refer to different aspects:

a- *Natural Language for Non-programmers* – nowadays software applications in mobile devices – typically smartphones – are increasingly used by non-programmers, meaning that software is more and more exposed and should be understood by people that do not "speak" programming languages.
b- *Software Systems Complexity* – software systems are growing in size, complexity and criticality, with potentially life-threatening situations – e.g. autonomous vehicles, remote surgery, and largely automatic power stations. Design of such complex systems is presented in increasingly high abstraction levels to enable design comprehension.

6.3 Conceptual Software: Theory

From a theoretical viewpoint an important issue is *formality*. FCA (Formal Concept Analysis) (Ganter, 1999), (Ganter, 2005) is a well-developed formalism dealing with concepts. It involves lattice theory and related algebraic domains of mathematics. Besides its theoretical importance, it has been shown to have a variety of practical applications, including software development.

One raises the issue of boundaries of the formalism applicability: are there software systems for which this formalism is insufficient? We encourage exploration beyond these boundaries, eventually leading to new discoveries.

6.4 Future Work

A final theoretical criterion we should consider is *precision and measurability* with regards to formal concept analysis. This is currently a topic of our research, and we have enough reasons to assume that results of interest are attainable.

Possible directions of measurability are comparisons of two numerical values: 1- a *structure refactoring ratio*, say in sub-section 3.2 we obtained the class diagram by reducing the number of classes in a 3/10 ratio; 2- a *semantic meaning ratio* which would express sizes of sets of terms needed to convey the same meaning.

7 CONCLUSION

The main contribution of this work is raising issues concerning the importance of conceptual analysis for software theory – which follows from inherent characteristics of natural languages, rather than from programming languages.

ACKNOWLEDGEMENTS

The authors wish to acknowledge significant suggestions by two anonymous referees.

REFERENCES

Backus, J. W., 1959. The syntax and semantics of the proposed international algebraic language of the Zurich acm-gamm conference, in Proc. Int. Conf. on Info. Proc.,Paris.

Booch, G., Rumbaugh, J. and Jacobson, I., 2005. *The Unified Modeling Language User Guide*, 2nd ed., Addison-Wesley, Boston, MA, USA.

Brown, J. R. and Fehige, Y., 2011. Thought Experiments, *The Stanford Encyclopedia of Philosophy*, E. N. Zalta (ed.). Web site: http://plato.stanford.edu/archives/fall2011/entries/thought-experiment/

Chomsky, N., 1957. *Syntactic Structures*, Mouton.

Exman, I. and Yagel, R., 2013. ROM: an Approach to Self-consistency Verification of a Runnable Ontology Model, in CCIS Vol. 415, 271-283, Springer.

Exman, I. and Speicher, D., 2015. Linear Software Models: Equivalence of Modularity Matrix to its Modularity Lattice, in Proc. 10th ICSOFT Int. Joint Conference on Software Technologies, Colmar, France, pp. 109-116, (July 2015). DOI = 10.5220/0005557701090116

Fauconnier, G., 1997, *Mappings in Thought and Language*, Cambridge University Press, Cambridge (UK).

Fowler, M., Beck, K., Brant, J., Opdyke, W. and Roberts, D., 1999. *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, Boston, MA, USA.

Gabriel, R. P., 2008. Designed as Designer, in OOPSLA'08.

Galilei, Galileo, 1632. *Dialogue Concerning The Two Chief World Systems*, Italian, English translation by Stillman Drake, University of California Press, Berkeley, CA, USA, 1953.

Gamma, E., Helm, R., Johnson, R. and Vlissides, J., 1995. *Design Patterns*, Addison-Wesley, Boston, MA, USA.

Ganter, B. and Wille, R., Formal Concept Analysis: Mathematical Foundations, Springer, New York, USA, 1999.

Ganter, B., Stumme, G. and Wille, R., 2005. *Formal Concept Analysis - Foundations and Applications*. Springer-Verlag, Berlin, Germany.

Hopper, G. M., 1953. Compiling Routines, *Computers and Automation*, vol. 2, (4), pp. 1-5, (May).

Lakoff, G., 1986, A Principled Exception to the Coordinate Structure Constraint, in *Proceedings of the*

the Twenty-First Regional Meeting Chicago Linguistic Society, Chicago Linguistic Society.

Langacker, R. W., 1987, *Foundations of Cognitive Grammar*, Stanford University Press, Stanford (CA).

Lindig, C. and Snelting, G., 1997. Assessing Modular Structure of Legacy Code Based on Mathematical Concept Analysis, *in ICSE'97 Proc. 19th Int. Conf. on Software Engineering*, pp. 349-359, ACM. DOI: 10.1145/253228.253354.

McCarthy, J., 1960. Recursive functions of symbolic expressions and their computation by machine, Part I., Comm. ACM, Vol. 3 (4), pp. 184-195 (April).

Mens, T. and Tourwe, T., 2004. A survey of software refactoring, IEEE Trans. Software Eng., Vol. 32, pp. 126-139. DOI: 10.1109/TSE.2004.1265817.

Millay, E. St. V., 1921. *Second April*, "Unnamed Sonnets I-XII", pp. 97-110, Mitchell Kennerley, New York, NY, USA,

Modern American Poetry, Online Poems, Web site: www.english.illinois.edu/maps/poets/m_r/millay/online_poems.htm

Noble, J., Biddle, R. and Tempero, E., 2002. Metaphor and metonymy in object-oriented design patterns, in ACSC'02 Proc. 25th Australasian Conf. Comp. Sci., Vol. 4, pp. 187-195. DOI: 10.1145/563857.563823

Nofre, D. et al., 2014. When Technology Became Language, The Origins of the Linguistic Conception of Computer Programming, 1950-1960, Technology and Culture, vol 55, pp. 40-75.

OMG (Object Management Group), 2015. "Unified Modeling Language" (UML) Specification version 2.5, (June 2015). URL Accessed September 2015: http://www.omg.org/spec/UML/2.5.

Pan, J. Z. et al., 2013. (eds.), *Ontology-Driven Software Development*, Springer, Heidelberg, Germany.

Piccinini, G., 2006. Computation without Representation, Philosophical Studies. DOI= 10.1007/s11098-005-5385-4.

Plath, S., 1963. Edge Poem, Web site: http://www.poetryfoundation.org/poem/178970

Smith, B. C., 2002. The foundations of computing, in: Scheutz, M. (ed.), Com*putationalism: New Directions*, pp. 2358, MIT Press, Cambridge, MA, USA.

Turing, A., 1937. On Computable Numbers with an Application to the Entscheidungsproblem, Proc. London Math. Society, vol. 42, pp. 230-265.

White, G., 2011. Descartes among the Robots – Computer Science and Inner/Outer Distinction, *Minds & Machines*. DOI= 10.1007/s11023-011-9232-4.

Wikipedia, 2015. Sylvia Plath, Wikipedia. Web site: https://en.wikipedia.org/wiki/Sylvia_Plath