Computer and Information Science; Vol. 15, No. 2; 2022
ISSN 1913-8989 E-ISSN 1913-8997
Published by Canadian Center of Science and Education

Transcendental Logic-Based Formalism for Semantic Representation of Software Project Requirements Architecture

Oleg V. Moroz[1], Oleksii O. Pysarchuk[2] & Tetiana I. Konrad[1]

[1] National Aviation University, Kyiv, Ukraine

[2] National Technical University of Ukraine «Igor Sikorsky Kyiv Polytechnic Institute», Kyiv, Ukraine

Correspondence: Oleg V. Moroz, Software Engineering Department, National Aviation University, 1, Liubomyra Huzara ave. Kyiv, Ukraine 03058

Received: January 3, 2022 Accepted: February 3, 2022 Online Published: February 14, 2022

doi:10.5539/cis.v15n2p15 URL: https://doi.org/10.5539/cis.v15n2p15

Abstract

This article is devoted to the analysis of the situation that has arisen in the practice of using artificial intelligence methods for software development. Nowadays there are many disparate approaches, models, and practices based on the use of narrow intelligence for decision-making at different stages of the life cycle of software products, and an almost complete lack of solutions brought to wide practical use. The article provides a comprehensive overview of the main reasons for the lack of the expected effect from the implementation of Agile and suggests a way to solve this problem based on the use of a self-organizing knowledge model. Based on the heuristic usage of transcendental logic in the terms of "ontological predicates", such a model makes it possible to create a formalism of the semantic representation of the requirements architecture of a software project, which could provide semantic interoperability and an executable semantic framework for automated ontology generation from unstructured informal software requirements text. The main benefit of this model is that it is flexible and ensures the accumulation of knowledge without the need to change the initial infrastructure as well as that the ontology inference engine is the part of the mechanism of collective interaction of active elements of knowledge and not some externally programmed system of rules that imitate the process of thinking.

Keywords: architecture of software project requirements, ontology learning, self-organizing knowledge, ontological predicates

1. Introduction

1.1 Common Situation and Trends in the Agile Software Development

In software engineering, Agile is the result of the development of a project-based approach to working in software engineering over the past several decades. However, despite its widespread use, it does not live up to the hopes that were pinned on it. Just like traditional methodologies, it does not meet all the needs of developers because “there is no definitive proof that the return on investment from agile projects is higher than that from traditional projects.” (Miller, 2013). Most software companies use Agile because they don't know anything else. They just must use what they have, although for many Agile has remained an insurmountable barrier: “While there is much anecdotal evidence that adopting agile practices and values improves the agility of software professionals, teams and organizations, the empirical evidence is mixed and hard to find” (Wikipedia, n.d.). The reason of this is that “practice software organizations are more concerned with sustaining Agile rather than adopting it” (Gregory et al., 2016). However, this does not mean that Agile has exhausted itself and should leave like its predecessors. On the contrary, Agile seems to have not yet fully revealed its capabilities.

Undoubtedly, the Agile facilitates planning, budgeting, forecasting, and project management due to improved communications, smoother collaborations, increased flexibility, and organizational self-management. For example, according to Standish Group 2018 Chaos Report (Standish Group, 2018) agile projects enjoy a 60% greater chance of success than non-agile projects. As for the two project factors, size, and approach, Chaos Report presents some genuinely interesting data. Large agile projects succeed at twice the rate of non-agile projects and fail half as often. Only in the small category does non-agile come close to agile. Therefore, Standish Group concludes that size trumps agile methodology. Besides that, complexity is one of the main reasons for all project failures. Large, very complex projects have both the highest challenges and failure rates. Thus, overly complex requirements, as well as scaling, are the main impact factor for project manageability. For small and simple projects, the choice of methodology does not matter; for large and complex projects, agility can only slightly improve the overall statistics but does not slay the “dragon” and does not guarantee final success. To effectively deal with the problem, you need to know all the vulnerabilities of the “dragon”. Agile projects often go beyond time and budget and fail. This is mainly due to two reasons. The first is since businesses are simply not ready to accept new ways of organizing work. The fact is that the key feature of the Agile methodology for software development, which consists in breaking the project into small iterations, during which part of the functionality is implemented, is, as is often the case, its most significant drawback. This means that for the successful application of this methodology it is necessary to constantly monitor the dependencies of the modules being developed on those that have already been implemented and a clear vision of the business goals of the project at all stages of its life cycle. The first allows developers not to make a lot of blunders, and the second – to concentrate attention and efforts on the most important positions at every moment of the Software Development Life Cycle. Otherwise, there will be inconsistency, repetition, and an increased risk of never releasing the final version of the product. That is why the greatest threat to Agile is its formal application, that is, the use of Agile model artifacts without considering the specifics of the project and, most importantly, without effective management of project knowledge.

The second reason for the failure of agile projects, as, indeed, and non-agile projects, is associated with the fact that the development life cycle of most software products is much longer than the time frame of the project that initiated it. If the house has been built, then it is built, and you will not add anything significant to its main functionality – you can only keep it up to date. Another thing is the knowledge that is embedded in the software product. Such knowledge is usually incomplete and there is always the opportunity to add something more to the existing functionality, and this opportunity only increases as organizational knowledge develops, transforming under certain conditions into a necessity. If a software product does not initially can adapt to changes in the organization's business goals in the future, such a product will sooner or later complete its life cycle.

Why is it so important for a software product to be adaptable to changes in business goals? Work in the modern economy is fundamentally ambiguous, unpredictable, and sometimes even chaotic. Typically, creating and maintaining software following specific business goals presents unique challenges and solutions, which leads to increased risks. This business reality is a key factor that makes software development so challenging and risky, which is why it is so important to be able to adapt to this business reality. Integrated Application Lifecycle Management (ALM) tools are called to solve this problem. Being the lifecycle management of a product, ALM is used in administrating a software application from its early phase until it is no longer used, and its main aim is to document and track the changes made to an application throughout its life circle (Aiello & Sachs, 2016; Rossberg, 2014). Not surprisingly, there are so few publications describing a scientifically based practical experience in developing and deploying enterprise-class ALM solutions. This is understandable since non-trivial and complex ALM processes involve the combination of various methods and tools and are used throughout the software development life cycle. The success of ALM implementation depends on many factors, and some barriers require the attention of both researchers and practitioners (Kääriäinen, 2011). ALM tools help software developers to achieve a higher maturity level and provide continuous workflow process improvement, but it will not solve the crucial problem – the complexity of the objectives forwarding, which is compounded by the complexity of the ALM solutions themselves. At the same time, the solution to this problem of complexity seems to be self-evident - it is the creation of a project knowledge base capable of integrating with the organization's knowledge management system. Only under this condition, the knowledge gained during the work on the project will be formalized and manageable, and the artifacts will be valid. The creation of such a knowledge base is the result of close cooperation between the customer, users, and the developer, and such cooperation is extremely important for the successful implementation of the project. Otherwise, there will be information chaos, to overcome which an agile approach to the software development was created. The short duration of iterations and the small amount of added functionality make it possible to correct inconsistencies and eliminate errors at a reasonable cost of time and effort. However, the constantly increasing software requirements complexity led, on the one hand, to the emergence of technologies such as Kubernetes (Lukša, 2017) for deploying and scaling applications and, on the other hand, must force researchers and practitioners more actively use artificial intelligence technologies to overcome challenges. A typical example of this trend is found in the field of Search-Based Software Engineering (e.g., Harman & Chicano, 2015; Ruchika et al, 2017; Ramírez et al., 2019), as well as works in the field of using probabilistic reasoning and machine learning in the software life cycle (Balikuddembe et al., 2009; Pandey et al., 2021; Jayagopal et al., 2021; Xu et al., 2016; Dell’ Anna et al., 2019). The most popular intelligent techniques for software development are as follows: reasoning under uncertainty (mainly, Bayesian network), search-based solutions, and machine learning (Perkusich et al., 2020). At the same time, the main purpose of applying intelligent methods is to support decision-making in such tasks as labor cost estimation, resource allocation, and requirement management: selection, analysis, and prioritization. However, most of these methods, as truly noted in (Perkusich et al., 2020), are still in their infancy for many applications, therefore much more theoretical and empirical research is needed yet.

1.2 Application of Artificial Intelligence to the Software Development

Many articles, reviews, books, conference proceedings are devoted to the use of Artificial Intelligence (AI) in software engineering in general. (e.g., Barenkamp et al., 2020; Batarseh et al., 2021; Shehab et al., 2020). The application of AI techniques covers all stages of software life-circle: software project planning, problem analysis, software design, software implementation, software testing and integration, and software maintenance. AI-enhanced software development tools are a good example of how AI can empower the routine developer’s work, but “it will take some time before they can learn to interpret the business value of each feature and advise on what to develop next” (Kurasinska et al., 2019). As for the most popular intelligent techniques in the context of Agile software development, the main purpose of using intelligent methods seems to be to support decision-making in the process of solving the Next Release Problem (Chaves-Gonz ález et al., 2015).

If to talk about the impact of AI on software development, the bulk of the interest in applying AI to software development lies in automated testing and bug detection tools, whereas the activities such as automata deployment pipelines, decrease defects, working with log analysis to optimize traffic routing cause minimal interest (Giudice, 2016). If saying in terms of decision-making support, AI is mostly used for highly structured tasks and much more extent for poorly structured and unstructured tasks requiring in-depth analysis of information, identifying relationships and patterns for accepting decisions. All this indicates AI technology is closed-minded about how to interpret business values.

In our opinion, the prospects for increasing the influence of AI on software development are associated, first, with the expansion of the use of AI at the stage of analyzing the initial data of the project, reducing the uncertainty in requirements and more efficient requirements management in the development process. Requirement development is essentially a mapping process between domain knowledge and the functional structure of the system developed. In the case of an iterative software life cycle model, such a process should be based on intelligent decision support models under fuzzy conditions, where the business goals of the project act as target functions since the requirements are not complete and can change during the implementation of design decisions.

1.3 Fundamental Programming Paradigm Shift

The inconstancy of requirements and the need for adaptive software implementation of the dynamics of the original subject area put forward certain requirements for the programming language, which must provide adequate tools for creating and manipulating the corresponding project artifacts. Unfortunately, the OOP paradigm, which today dominates programming like the Agile methodology in project management, is very limited in this regard, which introduces excessive complexity into the architecture of software systems, and noe always solutions based on an object-oriented approach turn out to be adequate business logic inherent in the requirements. Therefore, the urgent task is to create a different programming paradigm, focused not on objects interacting by passing messages to each other, but on facts reflecting the status and state of these objects, and manifesting and changing their properties through interaction with each other and with the environment.

There are two diametrically opposed points of view on this fundamental paradigm shift. The first was formulated by the Director of AI at Tesla Andrej Karpathy. In a blog post titled “Software 2.0” in November 2017 he wrote (Karpathy, 2017):“… Software 2.0 can be written in much more abstract, human unfriendly language, such as the weights of a neural network. No human is involved in writing this code because there are a lot of weights (typical networks might have millions), and coding directly in weights is kind of hard (I tried)”. And further: “The 2.0 stack can fail in unintuitive and embarrassing ways, or worse, they can “silently fail”…” Now you can see some excitement about the fascination with neural networks, but a frank remark by Andrej Karpathy looks like a fly in the ointment. Software development is a highly risky business, so software developers would not like to add to the objectively existing risks also the risks that come from working development tools.

The second point of view was expressed by the Co-founder, President at JetBrains Sergey Dmitriev in November 2004 in his article “Language Oriented Programming: The Next Programming Paradigm” (Dmitriev, & JetBrains s.r.o., 2004). The main limitation of the existing programming paradigms is, as S. Dmitriev accurately noted, that they make the programmer think like a computer, but it should be the other way around – a computer should think like a programmer. Accordingly, a program is not a set of instructions but "… any precisely defined model of a solution to some problem in some domain, expressed using domain concepts. … So, Language Oriented Programming (LOP) will not just be writing programs, but also creating languages in which to write our program". But easier said than done. The idea of LOP is based on using domain-specific languages (DSL) that are tailored to be highly productive in a specific problem domain. But how many domains the real program could involve? And how to ensure their DSL consistency and synchronization?

That is reasonably clear that we need a universal programming language, the logical structure of which, like the logical structure of a natural language, would reflect the structure of the real world. Within such a language, you can create any number of DSLs that are subsets of it, just like in the case of a natural language, which is the basis for creating languages for different subject areas. Unfortunately, today there are no such flexible and dynamic programming languages, but such a language can be created based on the principles of Wittgenstein's factual language (Stern, 1996) and the pictorial theory of sentences developed by him as the logical basis of language. In his early fundamental work "Tractatus Logico-Philosophicus" L. Wittgenstein offers fact as a combination of interconnected objects, not the object itself, to be the "elementary unit" of the world, just as he considers the sentence, not the name, to be the "elementary unit" of language. Proceeding from the fact that sentences represent facts, we conclude that the world that they define, and the language, which is a collection of sentences, have a similar structure or, in other words, have a common logical form. According to Wittgenstein, "only facts are capable of expressing meaning, the class of names cannot," and the meaning of the sentence is to depict a possible state of affairs in the real world. A sentence is a structural element in the model of reality, and its constituent elements are atomic facts (names). It should be borne in mind that this model refers to the formal abstract language of logic (metalanguage) and describes what the world really is but does not depend on our knowledge about it.

In his later works, Wittgenstein proposed the concept of a language game as the basis of natural language, the essence of which is that language and actions are considered as one whole. Various types of linguistic communication (language games) are possible, organized according to certain rules, where words can have different functions, and, therefore, relate differently to reality. The semantics of a word depends primarily on the context in which it is used in a language game. L. Wittgenstein wrote that the process of using words in a language is very similar to the process of children mastering their native language, emphasizing the importance of rules in the process of creating language games. It is no coincidence that teaching children are considered the most effective precisely in the process of games when they perform different roles.

In general, the scale of AI usage in software engineering correlates with the level of AI development and is mainly reduced to the selective use of narrow AI methods at certain stages of the life cycle of a software product. In some cases, AI can help fix certain problems and develop applications more efficiently. AI's limited use is due to its limited capabilities and, in some cases, unpredictable consequences and difficulties in interpreting the results. Non-software companies are also wary of their AI business strategies as suggested from the MIT Sloan Management Review, published in 2017 (Ransbotham et al., 2017): «The gap between ambition and execution is large at most companies. Three-quarters of executives believe AI will enable their companies to move into new businesses. Almost 85% believe AI will allow their companies to obtain or sustain a competitive advantage. But only about one in five companies has incorporated AI in some offerings or processes. Only one in 20 companies have extensively incorporated AI in offerings or processes. Less than 39% of all companies have an AI strategy in place. The largest companies – those with at least 100,000 employees – are the most likely to have an AI strategy, but only half have one». Wherein forms of AI in use today include predictive analytics (digital assistants), machine learning, natural language processing, voice recognition and response, virtual personal assistants (chatbots), diagnosis (recommendation engines) among others (Narrative Science, 2018).

1.4 Problem Statement

From the above, we can conclude that for effective management of project work in software engineering an urgent task is to implement a flexible structure of project knowledge that would provide monitoring and tracing of requirements and artifacts that are changed over time in the project and initiated by evolving organizational knowledge. To take a truly significant step towards increasing the efficiency of the development of complex software systems, it is necessary, first, to make efforts to uniform the representation of the project knowledge and, above all, project requirements as the basis of a conceptual scheme, and also to create a programming language that is not focused on building a hierarchical object model of the domain, but on reflecting the dynamic interaction of objects between themselves and the environment, through which the properties of these objects are manifested and changed.

Building a unified project knowledge base and keeping it up to date is the key to the success of the software project, but such a knowledge base should be able to integrate with the organizational knowledge management system, then any software system based on this knowledge can be constantly kept up to date. However, in this case, the concept of a project itself is leveled, since "to be truly competitive, an organization needs to be able to deliver a continuous stream of change. Managed properly, this negates the need for a project and the associated cost overheads" (Leybourn & Hastie, 2018). Moreover, for this, the methods of formalizing subject and project knowledge should be unified. Approaches to the formalization of knowledge can be different depending on the model of knowledge representation, but the most justified from the point of view of cognitive science is the ontology-based model. This article presents an ontological framework for the formalization and semantic modeling of informal project requirements, functioning based on the principle of self-organization of knowledge, that is, assuming that knowledge is inherent in the emergent ability to create new knowledge based on existing knowledge, and the corresponding model of knowledge representation (Moroz, 2020), which uses transcendental logic-based ontological relations to describe the relationship between structural elements (Moroz, 2021).

The article is further organized as follows. Section 2 provides a detailed overview of the key aspects of the requirements formalization process for a software development project, its problems, challenges, and failures, and most of all, automation issues. In addition, aspects of cognitive categorization and shortcomings of the main theories of knowledge categorization are considered. Section 3 discusses the role of a multi-level software project requirements architecture in the software development technology stack, the relationships of requirements models in this architecture, and presents an ontological framework of a typical requirements architecture based on transcendental logic-oriented ontological predicates as an alternative platform for ontology learning. In section 4, the main findings of the study are briefly summarized. In section 5, the article concludes with a general discussion about the prospect of creating a fully automated unsupervised semantic parsing tool.

2. Key Aspects of Software Project Requirements Formalization

2.1 Formalizing Project Requirements

Requirement specifications play a key role in software development since the analysis of requirements and their subsequent tracing during the development process has a significant impact on the quality of the design process and the final product. Nowadays computer tools for the conceptual phase of the design process are now emerging. The main challenge in creating design tools for this phase is to formulate functional and non-functional requirements in such a way that they can be processed on a computer. To this end, informal specifications presented in the textual form are refined to create formal specifications suitable for automatic processing. At the same time, the process of clarifying the requirements is iterative and usually requires the active participation of domain experts. Iteration with backtracks is perhaps the most reliable way to get the desired result, but it is very extensive being associated with significant resource costs due to the need for numerous repetitions and rework (e.g., Kulyamin et al., n.d.; Ghazel et al., 2015; Peres et al., 2012).

In addition to iteration with backtracks, goal-driven software development methodology is also used to clarify requirements that provide a visual modeling language to define an informal specification. In this case, the most used form of the representation of the structure and behavior of a software model are diagrams UML/SysML (UCL - SST/ICTM/INGI - Pôle en ingénierie informatique & van Lamsweerde, 2009). A classical implementation of the goal-driven paradigm brought to an acceptable level of maturity is the requirements behavior tree (Dromey, 2003). Behavior trees allow you to find a compromise between requirements and business goals, allowing changes to both when certain conditions and constraints are imposed, that is, inherently represent a variant of goal trees used in knowledge-based systems to model the inference engine. Accordingly, they have the same drawbacks, such as retraining or oversimplification of the situation, and besides, large trees are too difficult to exchange information with stakeholders.

In certain cases, methods of system dynamics methodology can be applied to clarify the requirements. (e.g., Li, 2021). They allow comparing alternative change management strategies in terms of project performance indicators. Such methods emphasize the formalization of the supported business process, which allows you to identify inconsistencies and gaps in this process, and therefore in the requirements.

These methodologies are very sensitive to the scale of the project and to the temporal evolution of the modeled system since they require a lot of effort to rework their models in the event of frequent changes in the business process, for example, in the PCDA cycle. In addition, presenting a conceptual diagram of a system as a set of specific diagrams, such as the UML, is developer-oriented and, despite their clarity, makes it difficult for the interaction of developers and stakeholders, which is contrary to one of the four fundamental principles of Agile.

As a rule of thumb, when it comes to formalizing requirements, this usually refers to functional requirements. Non-functional requirements are usually considered only informally due to their complexity, high level of abstraction, and lack of methodologies and support tools. At the same time, non-functional requirements define a significant part of the design decision-making process and limit the way the product functionality is implemented by imposing additional constraints in business process models (Shankar, 2020), so they are also known as the system's quality attributes. The approach to requirements management should be sure flexible, especially with respect to non-functional requirements, but they are much more difficult to systematically model, analyze and change, in contrast to functional requirements, the satisfaction of which can be easily tested (Moradi et al., 2018; Jackson et al., 2009). Thus, non-functional requirements should be included in the general scheme of requirement formalizing, however, the problem of formalizing such requirements and determining the degree of their satisfaction in the final product is still awaiting its solution (Rosa et al., 2004).

2.1.1 Requirement's Formalization Automation Issue

Although publications present many different approaches to automation of the requirement formalization, there is still not a fully rigorously substantiated and proven practical solution. The process of informal requirements formalization is usually taking to consist of three phases (Cimatti et al., 2009; Fatma Bozyiğit et al., 2021):

1) Informal analysis phase: informal specifications are structured and categorized using appropriate taxonomies.
2) Formalization phase: a conceptual model is automatically created from categorized fragments of requirements through a transformation model.
3) Formal validation phase: the resulting conceptual model is validated.

The transformation model allows you to define the elements of the conceptual model (entities, their attributes, and relationships, constraints that determine the conceptual element integrity), relying on the structured data obtained in the previous phase. Based on the analysis of publications on this topic, you can distinguish three main types of analysis and corresponding models, which are subjected to categorized fragments of requirements while transforming into a conceptual model: linguistic analysis, pattern-based analysis, ontology-based analysis.

A. Linguistic analysis.

The linguistic analysis focuses on comprehending the subject-logical content of the text, factual and conceptual information available in the text and includes morphology, syntax, semantic, and pragmatic analysis, among others. Morphology deals with the formations of words from the morphemes. The syntax works similarly to morphology but refers to sentence structure. Semantic focuses on studying the meanings of words. Pragmatics is like semantics but with words, phrases, and utterances being studied in context rather than independently. Natural language text processing may begin with morphological analyses, stemming from the terms involved. The aim of syntax analyses is to determine the characteristics of the words, recognition of their parts-of-speech, determining the words and phrases, and parsing of the sentences. A parser takes a sentence and turns it to the sentence diagrams. Once the sentence is parsed the one can find all the noun phrases, for example, that is sentence parsing is just the way to find concepts.

To determine the meaning of a sentence, semantic parsing is used. Semantic parsing deals with converting natural language utterances to a semantic representation. This representation is often referred to as logical forms, meaning representations, or programs that can be easily executed on a knowledge base. There are several semantic representation formalisms (Kamath & Das, 2018): first-order logic, graph-based formalism, and programming languages. The most used logic-based semantic representation is λ-calculus that represents the meaning of real-world objects using a set of ontologies and uses syntactic rules for composition. Basic formalism (or grammar) is used to obtain valid logical forms. Next, the parser searches for logical forms with suitable characteristics in accordance with some model that generates distributions over these valid logical forms. The model parameters are updated according to a specific learning algorithm using training examples.

Let's try to evaluate this scheme of machine-learning-based semantic analysis from the point of view of Wittgenstein's language-game theory, according to which language and actions should be considered as a whole and the truth of which hardly anyone doubts today. Language games, according to Wittgenstein, are infinitely counted, and their infinite variety creates an abundance of life practices. It is impossible to classify these conceptual speech practices and games since there are a huge number of their possible types and varieties. We can only try to identify the rules by which this or that language game "works". Wittgenstein wrote that people too often fail to notice the obvious. Obvious things are the hardest to understand since they are too close, and it is not customary to think about them reflectively. When we use words, each time we carry the meaning of the word into a new context, trying to discover new ways to use it. In other words, we are creating a new language game.

The same image or word can be perceived differently by different people. For example, in a disputable situation, one party may state: "This is my point of view." However, if the opponent has a different mental context for using the words that have become the subject of the dispute, then there is no way to resolve the conflict. The parties simply operate with different meanings, although they verbally pronounce the same words. Similarly, no matter how many times we highlight a certain phrase in the text, we will not be able to accurately assess its meaning if we do not know the rules of its original language game. Two disputing parties reach an understanding only if they accept the same rules of the game. This usually happens because of discussion and exchange of views. This means that to convey the exact meaning of the sentence of the language, it is necessary to bring the accompanying rules of the game to the perceiving side. For this, semantic schemes and ontologies are intended, but the question is how effective and expressive the formalisms used today to represent them are.

Semantic representations should be created in parallel with a chain of actions ready for execution, as it happens in the human brain while the perception of a phrase or sentence. Instead, learning semantic parsers are created to match natural language sentences with use case analysis, formal queries to the knowledge base, or programs in formal command languages, and there is little confidence in the validity of these matches. Methods such as deep learning add confidence to researchers because they remove all these multi-step transformations and metamodels in the process of semantic analysis, but first, you need to teach them how to evaluate the reliability of the responses of the semantic parser, learn from the knowledge already gained and work with models that cover at least several unrelated domains (Grefenstette et al., 2014).

Linguistic analysis computer’s tools are concerned with Natural Language Processing (NLP). The most common NLP approaches are symbolic, statistical, connectionist, and hybrid (Liddy, 2001). We will not try to dig into the details of these approaches but instead focus on the fundamental differences between them, because all of them are aimed at teaching the computer to human-like comprehension of natural language, translate and generate natural human text and language. They differ in the processes each approach follows, as well as in system aspects, robustness, flexibility, and suitable tasks. The symbolic approach to NLP is based on rules and lexicons developed by humans according to generally accepted patterns within a given language. Logic-based and rule-based systems and semantic networks are examples of symbolic systems. Despite the preferences of machine learning in NLP research, symbolic methods are still commonly used when the amount of training data is insufficient to successfully apply machine learning methods.

Statistical and connectionist approaches are based on the machine-learning paradigm. In the case of the statistical approach, a computer system can develop its own linguistic rules resulting from using various mathematical techniques and actual large text examples. Statistical methods in NLP research have been largely replaced by ANN-based machine learning, however, they continue to be relevant for contexts in which statistical interpretability and transparency are required. The connectionist models that combine the symbolic and statistical approaches operate statistically accepted rules of language, but allow adaptive transformation, inference, and manipulation of logic formulae.

The situation in the NLP body of research can be characterized as a pluralism of approaches and methods against the background of universal motivation by neural-network technology. In one case, one approach may be sufficient, in another case, to achieve the goal, it will be necessary to combine different approaches, which is the essence of the hybrid approach. Hybrid methods take advantage of the strengths of multiple approaches to solve NLP problems more effectively.

Known methods for linguistic analysis of informal natural-language requirements are developing within the NLP scope, so they allow a lot of disadvantages, but, in addition, they “miss a methodology for an adequate formal analysis of the requirements” (Cimatti et al., 2009). On the contrary, methods that are aimed at creating formal model-based specifications and use formal specification notations such as Z-language (Madhan et al., 2017) have sufficiently developed means for formal analysis of the functional requirements specifications. Based on set theory and first-order predicate logic, such notations are as expressive as they are poor for semantic analyses, therefore “they frequently ignore developers’ needs, target specific development models, or require translation of requirements into tests for verification; the results can give out-of-sync or downright incompatible artifacts” (Naumchev et al., 2017).

Linguistic analysis of informal requirements in a text form is prone to errors in part due to the quality of the texts (sentence parsing and moreover semantic parsing does not work properly when the quality of the text is low), but mainly due to ambiguity or incompleteness of the requirements. To reasonably eliminate uncertainty in requirements engineering metaheuristic search techniques (Li et al., 2017) are may be used, among which most widely applied to software engineering are follows: hill-climbing, simulated annealing, and genetic algorithms.

Search-based methods have been used for requirements selection and optimization with the goal of finding the best possible subset of requirements that satisfy certain constraints, considering interdependencies between requirements. Used in conjunction with machine learning, such methods could become the basis for enforcing the requirements engineering, however, the potential of machine learning concerning the formalization of informal requirements is clearly exaggerated, therefore, despite many years of history, "the current body of research is composed of new ideas, which have not yet been validated to its full extent by the scientific or industrial communities" (Iqbal et al., 2019).

Finally, linguistic analysis is also a 20th-century philosophical movement inspired by L. Wittgenstein and aimed at the formal theory of language. Wittgenstein's language-game theory has been used to improve modeling techniques within the Logico-Linguistic Modeling (LLM) framework (Gregory F.H. & Warwick Business School, 1993). Based on one of the maybe more widely used systems methods – Soft Systems Methodology (Burge & Burge Hughes Walsh Limited, n.d.), modal logic, and logic programming languages, LLM represents the most consistent, philosophically sound method of formal-logical systems design now. However, LLM models turn out to be so complex that you must abandon the operators of modal logic. The problem, in our opinion, lies not so much in Soft Systems Methodology, which is obviously also not without its shortcomings, but in the reductionist striving of researchers to decompose a complex problem into simpler components and describe their causal connections in the terms of formal logic. They forget that any problem can be complex depending on the context in question, which, if we talk about the interpretation of the surrounding world in an arbitrary form, is usually not limited and is often not unambiguously defined. It should also not be forgotten that a complex system is not just the sum of its constituent elements and often reveals properties that the constituent elements do not have. In summary, formalizing language games in an open-ended framework "requires a major paradigm shift in formal semantics. It doesn't reject logic, but it applies logic to a broader range of problems with a greater sensitivity to the way language is actually used by people at every stage of life" (Sowa, 2007).

B. Pattern-based analysis.

Specification patterns capture dependencies and connections between requirements, allowing specifying unambiguous requirements in natural language using predefined formalisms (Cimatti et al., 2009; Miao et al., 2015; Withall, 2010). They allow you to formalize requirements in less time and with less effort, and the resulting specifications contain fewer errors compared with other formal methods. However, specification templates reduce the entire spectrum of possible solutions to a small set of standardized archetypes, therefore they are well suited for formalizing standard requirements that contain functional descriptions of the system at a high level of abstraction but turn out to be useless in the case of complex non-standard conceptual schemes.

C. Ontology-based analysis.

Parsing software uses automatic mathematical and statistical analysis to extract lexicons that occur in a set of documents and the words that make up them. They assume that the distribution and frequency of occurrence of specific concepts in documents within a theme are the same. At first glance, these methods are useful for comparing documents and identifying similarities between them, which allows the methods to be used to categorize documents. Moreover, statistical models such as the Hidden Markov Model, which introduce probabilistic solutions based on attaching real weights to the characteristics of the input data (such models are currently used by many speech recognition systems), tend to be more robust when entering unfamiliar input data and input data containing errors. But if the goal is to understand the essence of the document, that is, information retrieval, as well as the most important elements or relationships in the document, then a method that reproduces or tries to reproduce human understanding will naturally be preferable to a method that based on word frequency statistics.

Semantic analysis software focuses on the meaning of words and how they are perceived by humans. This approach involves understanding the context in which words appear. The importance of understanding context is especially evident in natural language idioms. The key to understanding idioms is to never take them literally because the individual words do not fit together at all. They must be taken into context to understand their true meaning. However, understanding the context requires semantic models. Such models create a natural language environment that is used to conceptualize problem knowledge, integrate data from different resources, and link concepts to this data, computational models, and control structures to further expand the area of knowledge. Semantic modeling is provided by ontology engineering that is "the set of activities that concern the ontology development process, the ontology life cycle, and the methodologies, tools, and languages for building ontologies" (Corcho et al., 2007). Ontology engineering is widely used to model the semantics of complex systems in a wide variety of areas, including software development activities such as requirement engineering (including nonfunctional requirements), domain modeling, and process refinement (Pan et al., 2015; Ameller & Xavier, 2014).

The term "ontology" itself is an example of the contextual dependence of the meaning of a word. In the knowledge representation context, “ontology” is “a catalog of the types of things that are assumed to exist in a domain of interest D from the perspective of a person who uses a language L for the purpose of talking about D” (Sowa, 2012). In the artificial intelligence context, “ontology” is defined as “an explicit specification of a conceptualization” (Gruber, 1993). In software engineering, “ontologies are reusable and sharable artifacts that have to be developed in a machine-interpretable language” (Corcho et al., 2007). Finally, in the context of Wittgenstein's language-game theory, ontology could be defined as the prerequisites for semantic representation of linguistic communication rules.

Manual ontology development is a rather laborious and complex process aimed at integrating formalized knowledge and data obtained from different sources. The automatic or semi-automatic creation of ontologies is known as ontology learning (Maedche & Staab, 2001). Ontology learning approaches can be categorized according to data types (Lehmann & Völker, 2014): ontology learning from text, linked data mining, concept learning in description logics, and OWL, crowdsourcing. Today, the pressing problems of ontology learning are associated with progress in solving the problem of automated acquisition of ontology from unstructured text. Even though a lot of methods have been developed to solve this problem in such areas as machine learning, text mining, knowledge representation and reasoning, information retrieval, and natural language processing, ontology learning remains an intensively developing area of scientific research. Summarizing all the main disadvantages of these methods, then in ontology learning, you can achieve the desired result “by showing directions for answer validation, language-independent ontology generation and crowdsourcing usage for automatic ontology post-processing” (Asim et al., 2018).

There is an approach known as Ontology-Driven Software Development (ODSD) in software engineering. This approach, as well as Domain-Driven Design (DDD) approach (Evans, 2015), follows the concept of Model-Based Software Development, the main idea of which is how to better integrate high-level domain models into the software development life cycle (Valiente et al., 2011). The purpose of the ODSD is the same, but unlike DDD, it applies this idea in a much more radical way: domain models are used not only to generate code but also as executable artifacts at runtime (Knublauch, 2004; Lavbič & Bajec, 2011). The idea of executable (interpreted) semantic models seems especially promising in the context of extracting knowledge from textual data, since such models, together with the use of ontological design patterns (e.g., Almeida Falbo et al., 2013; Gangemi et al, 2007) can significantly reduce the influence of the human factor both on ontology learning and ODSD (Bhatia et al., 2018; Siddiqui & Alam, 2013).

Ontology Design Patterns (ODP), like architectural patterns, simplify the design process and ensure its quality. The ODP Master Directory is developed and maintained by the Association for Ontology Design & Patterns and contains several basic types of patterns that collectively constitute an ontological design framework (Association for Ontology Design & Patterns, n.d.): structural, correspondence, content, reasoning, presentation, and lexico-syntactic ODPs.

The semantic analysis of an arbitrary text should be sure based on the use of linguistic ontologies developed with respect to lexico-syntactic patterns and, above all, cross-linguistic discipline ontology, which is a network of concepts used to describe and analyze natural language (Schalley, 2019). However, it should be borne in mind the linguistic ontology’s role in interpreting the meaning of the text is secondary to the domain ontology’s role of specifying the requirements, but the two working together significantly improves information retrieval and the understanding of the information retrieved (Zyl & Corbett, 2001).

Although the problem of unsupervised ontology induction from text is still relevant (Poon & Domingos, 2010), some progress has been made in the field of semantic parsing based on the joint use of ontology learning and machine learning (e.g., Choi et al., 2015; Starc, & Mladenic, 2016; Cheng et al., 2018). Nevertheless, unsupervised machine-learning solutions are commonly very noisy, so they need predefined natural-language templates, moreover “supervised learning requires labeled data, which itself is costly and infeasible for large-scale, open-domain knowledge acquisition” (Poon & Domingos, 2010). Thus, up to date, there is not a fully automated unsupervised semantic parsing method for open domains.

If sum up, no matter what approach is used to formalize informal requirements, truly flexible adaptive software development is impossible without effective management of both subject knowledge and project knowledge, in other words, a universal semantic formalism is needed to formalize both. Traditionally, first-order logic and computer languages are used to formally represent ontologies, but their very limited capabilities to support conceptual schemes create insurmountable obstacles to creating an enterprise-sized requirements model, which provides informality especially during early requirements analysis.

2.1.2 Validation of Software Requirements

Another important aspect of the requirements formalization concerns that requirements are to be confirmed to guarantee their correctness. The requirements validation is a fundamental step in the development process, especially for safety-critical systems (Cimatti et al., 2009). Unlike formal code validation methods, informal requirements validation methods are poorly understood. Here we can note an approach based again on ontologies (Chen et al., 2020). Difficulties in this aspect are because the correct requirements must be adequate to the real situation, therefore the formalization and verification of requirements usually require the active participation of experts. In other words, the correctness of the formulation of requirements in a document, formal or informal, is determined by the adequacy of understanding the subject area and the needs of users (Kulyamin et al., n.d.). It would not be difficult to solve this problem if the requirements are described initially according to the representation formalism of domain knowledge. Ideally, requirements development, especially at the conceptualization stage, should use the same methodologies and tools which are used for domain knowledge management. For this, it is necessary, at least, to have a single knowledge representation model. Semantic models are the most accurate in representing knowledge but the existing approaches and methods for their creation, described above, are poorly compatible with the task of displaying the dynamic interaction of domain essences.

We use here the term “essence” to denote the interacting elements of the domain instead of the term “object”, assuming that the essence is the internal basis, the true nature of anything, without which it is unable to exist, whereas an object is a realized essence in its specific embodiment, that is, an essence that manifests itself in a context. Each essence can have many such implementations, depending on the context of use, for example, the car essence can be implemented as a trade item, a production item, a property item, etc. Understanding the etymology of these terms is essential for formalizing the knowledge contained in informal requirements specifications and for understanding the fundamental difference between object-oriented modeling and ontology-oriented modeling. In contrast to the object-oriented approach, the ontology-oriented one just allows considering the meaning context and different points of view on the mental concepts.

2.1.3 Software Requirements Models

The use of any intellectual methods for the analysis of texts is faced with the insurmountable problem of the existence of an almost unlimited set of contexts of their formal representation. The existence of verbal and situational contexts allows one to describe different and even contradictory knowledge. Again, elements of knowledge can be reused in different contexts, allowing different points of view to be synthesized in the overall context of the project. For example, in every project, we must clearly define at least three points of view and, accordingly, three types of requirements: requirements for the end-user, requirements for the customer, and requirements for the developer among others. However, much depends on what is meant by the context and how it is represented in conceptual and semantic schemes. The generally accepted definition of context is as follows (Dey, 2001): “Context is any information that can be used to characterize the situation of an entity. An entity is a person, place or object that is considered relevant to the interaction between a user and an application, including the user and the application themselves.” This is the definition of context in a narrow sense, it is often used for ontological modeling in context-awareness computing and “depends on the interpretation of the operations involved on an entity at a particular time and space rather than the inherent characteristics of the entity.” (Ejigu et al., 2007). If we are talking specifically about the inherent characteristics of the entity, the context should be understood in a broad sense, that is as something that arises always and everywhere where there are mental concepts that enable to change their identity since only in the context we can talk about any concepts or the meaning of something. Statements, judgments, actions are the use of concepts in context. Wittgenstein's language games are contextual because they are tied to reality, being its reflective image.

Among the various methods of context modeling, ontology is the most popular approach. In the structure of context ontologies, upper ontology and domain-specific ontologies are usually distinguished (e.g., Gu et al., 2020). The upper ontology is a high-level ontology that captures general context knowledge about the surrounded environment. An important function of the upper ontology is to support the semantic interoperability of many domain-specific ontologies. The terms of domain-specific ontologies are subordinate to the terms of the upper one. The domain-specific ontologies are a collection of low-level ontologies which define the details of general concepts and their properties in each subdomain. The low-level ontology contains interpretations of concepts from upper ontology and other low-level ontologies in the context of a particular domain, in other words, low-level ontologies can be related to each other and form a hierarchical structure in which each low-level ontology has her super ontology.

The main weakness of such a context ontology model is context reasoning, which should provide a semantic interpretation of contexts, including the choice of the context within the meaning of the expression if it is not explicitly indicated. Ontology-based systems have disadvantages just because the relation expression between domain entities can be complex a great. As a rule, an ontology inference engine is a logic system that reasons over the semantic model of an ontology. In the case of context ontologies in OWL, for example, an ontology inference engine should provide a set of rules for interpreting the semantic model and detecting inconsistency in the knowledge base. In a situation of uncertainty if insufficient knowledge, the truth value of a statement is not binary, so other models of context-aware ontology can be applied, for example, the Bayesian network (Mok & Min, 2015).

The absence of upper ontology will make the task of semantic analysis very difficult. Regarding the task of formalizing requirements in a software project, in this case, you will have to build a separate context model for each project and track conflict situations when knowledge elements from different contexts are shared in the text of requirements. It looks as if all stakeholders in the project have entered into an agreement of intent, which indicates the boundaries and degree of interest of each participant. To a certain extent, requirements specifications serve as such an agreement. The question is what to choose as this upper ontology. You can limit yourself to creating an ontology of the requirements specification, which, like ontologies for context-awareness systems, will consist of several classes of contexts each of which will correspond to one of the project artifacts. But you can go further and link this ontology directly to the basis domain ontology. In this case, any artifact in the project can be a context for arbitrarily many concepts from the domain ontology, and any changes in this or that concept will be automatically reflected on the associated artifacts. So it could be in theory, but in practice, in the process of developing and managing requirements, developers, like many years ago, are guided mainly by common sense and rely on their own experience. At the same time, the implementation of such an obvious scheme of their interaction can certainly fully provide such standard properties of requirements as adequacy, unambiguity, consistency, and completeness. The current situation in ontology engineering, unfortunately, does not allow for the direct dynamic connection of domain and project knowledge. This at least requires a universal upper ontology based on the process of cognizing the surrounding world in which people use both a priori knowledge and a posteriori knowledge.

In requirements engineering, both researchers and practitioners are struggling with the problem of the contextualization of knowledge. Something that can connect their efforts is the use of requirement models, proposed by Beatty and Chen (Beatty & Chen, 2015). They identify four major classes of requirement models: those dealing with objectives, people, systems, and data. In the context ontology for a software project, the classes of requirement models are low-level ontologies of subdomains of the software requirements domain, for which the ontology of requirements specifications will be as a super ontology.

Table 1. The major classes of requirements models (borrowed from (Beatty & Chen, 2015)).

Class	Description	Models	Bounding model
Objectives	Describe the business value of the system and help you prioritize features and requirements based on their value.	Business Objectives Objective Chain Key Performance Indicator Feature Tree Requirements Mapping Matrix	Business Objectives
People	Describe who is using the system, along with their business processes and goals	Org Chart Process Flow Use Case Roles and Permissions Matrix	Org Chart
Systems	Describe what systems exist, what the user interface looks like, how the systems interact, and how they behave.	Ecosystem Map System Flow User Interface Flow Display-Action-Response Decision Table Decision Tree System Interface Table	Ecosystem Map
Data	Describe the relationships between business data objects from an end-user perspective, the life cycle of the data, and how that data is used in reports to make decisions	Business Data Diagram Data Flow Diagram Data Dictionary State Table State Diagram Report Table	Business Data Diagram

Table 1 contains brief descriptions of these classes of models, a list of constituent models, as well as the main models (bounding model), on which all other models of this category depend and which are bounds the corresponding space: objective space, people space, system space, and data space.

Beatty and Chen provided a detailed description of software requirements models and their mutual influences. The reliability of their taxonomy is not in doubt, since it is the result of many years of practical work with projects, but perhaps the most significant result of the work is the conclusion that different requirements models are needed to identify, verify, and analyze requirements, combined into a multi-tier requirements architecture. Given an example of such an architecture that includes all kinds of models, Beatty and Chen note that the relationships between the requirements models are not fixed. The relations depend on the situation in the project, so there may be other relationships between the models in addition to those indicated in this example, or, conversely, there may be fewer of them if some models were not included in the requirements architecture. Also, besides the selected models a requirements architecture can include functional requirements, business rules, test cases, screens, or model elements such as process flow steps.

Thus, it's talking about a high-level view of the requirement architecture, which must be detailed for a more complete view. However, to do this it is necessary to have a clear unambiguous understanding of the nature of the interrelationships between models in such an architecture. Models can influence each other in different ways. For example, some models are based on others, if there are causal relationships between their elements or models can provide information for other models, if there is a certain sequence of actions in the process flow. In addition, there can be several levels of relationships (cascading relationships) in a project, where one model is linked to another model, which in turn is linked to one more model. All this leads to the need of creating a requirements architecture ontology that would describe the semantics of relations between at least the main requirements models, that is, those are often found in projects. Such an ontology can serve as a framework for building a requirements architecture in a specific project.

2.2 Categorizing Domain Knowledge

Consideration of the process of formalizing requirements in a software system development project cannot be complete without discussing the problem of formalizing organizational (domain) knowledge, which is based on the process of categorizing knowledge. The most complete picture of the situation in this area is provided most likely by the project of the W3 Semantic Web Consortium with its basic concepts based on the categorization of knowledge. Ontologies used in the Semantic Web are built using description logics, many of which can be viewed as fragments of predicate logic. Domain knowledge represented using description logic is subdivided into general knowledge about concepts and their interrelationships (intensional knowledge), and knowledge about individual entities, their properties, and relationships with other entities (extensional knowledge). Understanding the difference between intensional and extensional contexts is still the subject of philosophical discussions, since the essence of the semantic analysis of these contexts involves, first, identifying the features of their logical structure. This fact largely explains the difficulties that the implementation of the Semantic Web concept faces, up to its complete rejection. So, to the question "Is the Semantic Web dead?" Google's search engine replies: “Semantic technologies have long been heralded as the best way to add linked data to your site. But since the rise of AI, many are now asking, “Is the Semantic Web dead?” In short, yes." This conclusion is rather hasty, if only for the simple reason that the creation of artificial general intelligence is still far from completion, and there is no more or less substantiated alternative for a semantic approach to knowledge representation. The stumbling block for this approach, in general, seems to be the use of description logics for ontology reasoning.

Even though description logic is the most popular model for representing knowledge, there have been few studies of the disadvantages of description logics, understanding the cognitive difficulties presented by them (Warren et al., 2014). In these studies, attention is paid mainly to the issues of OWL syntax, the difficulties in using commonly used features due to the relationship between rule-based and model-based approaches in human reasoning. Indeed, a person in his reasoning uses both rules (logics) and mental models, but much more important is the question of how many a priori and post prior forms of thinking are in this reasoning, that is, what's the balance between formal logic and transcendental logic. The problem of such languages of knowledge representation as OWL is precisely in the complete absence of means of expressing transcendental relations. Such relationships do not provide an idea of the properties of entities but allow exposure of the properties and relationships of experience objects in general, limiting the search space within which the search for logical inference is conducted. They can’t be used directly in logical judgments but each of them may be represented as a set of a formal logic operators. Other experience-grounded categorization models as that in which “all relations are variations of categorical relations, such as Inheritance and Similarity" (Wang & Hofstadter, 2006) (that is essentially based on predicate logic), also does not appeal to a priori forms of thinking.

In cognitive theory, categorization is a fundamental cognitive process that involves assigning objects to a particular category. A category is usually understood as mental and linguistic groups of elements, distinguished by one or another attribute (attributes) and generality of functions. Since the category is not only the result of a mental reflection of reality but also expresses the specifics of its understanding, the categorization process, being part of the cognition process, is closely related to the mechanisms of coding knowledge. Thus, categorization reflects not only what knowledge we possess and how we apply it but the forms and methods of preserving this knowledge. As you know, cognitive categorization is the process of human cognitive activity, which includes: the formation of categories, the correlation of new knowledge about the structure of the world with already formed and a priori categories, the structuring of knowledge according to the chosen model of knowledge representation, as well as the classification of linguistic forms of the mental representations (concepts) arose from the categorization.

Thus, at the output of the categorization process, we must receive mental representations structured following the chosen model of knowledge representation. These mental representations are usually distributed over three levels: superordinate level (the highest level of abstraction, indicating generic concepts or essences), basic level (average level of abstraction, including a perceptually salient, easy to detect and memorize object-type concepts), and subordinate level (the lowest level of abstraction, which includes units that characterize in detail individual sides of any phenomenon or naming specific objects of reality). Basic level mental representations are most suitable for identifying domain-specific properties since they demonstrate great similarity within a category and large differences between categories, and category samples have a generalized identifiable form that is most convenient for perception. Therefore, in a true cognitive process-based ontology, one should distinguish not two levels, but three: upper-level ontology, which corresponds to the superordinate level and contains categorical knowledge; middle-level ontology, which corresponds to the basic level and contains mental concepts that form the semantic core of domain; and low-level ontology, which corresponds to the subordinate level and contains individual concepts and facts that are different factual knowledge in a context.

There are two main approaches to categorization: the classical approach based on the teachings of Aristotle and Plato, and the approach called the theory of family resemblance, based on the works of L. Wittgenstein. The essence of the classical approach is that all samples included in one category, or another must have the same set of essential features, have the same status in the process of categorization. This approach is usually used to create various kinds of taxonomies in science. This is the classical model of categorization. It does not consider the peculiarities of understanding the world by a person, that is, it does not reflect which objects and phenomena a person considers to be more typical, and which - to less typical.

The essence of the theory of family resemblance lies in the fact that identical signs may not be found in all specimens of the category, but only in some, and the set of signs may vary. For example, among the characteristics of members of one human family, some relatives have common character traits, others – external similarities, etc. The teachings of L. Wittgenstein gave impetus for the development of similarity-based theories of categorization, such as prototype theory and exemplar theory as well as appropriate formal models of categorization (Cohen & Lefebvre, 2017). Compared to classical theory, similarity-based theories have an advantage. In the prototype theory, a categorization decision is made by comparing a new sample with a category prototype, which is understood as the best, i.e., its most characteristic and illustrative example. Under the exemplar theory all known instances of a category are stored in memory as exemplars, so, if assessing membership in a category of an unfamiliar entity, samples from potentially relevant categories are retrieved from memory, and the entity's similarities with these samples are summed up to formulate a categorization decision.

The disadvantage of similarity-based theories is that they either ignore the properties of individual members of the category, focusing on the criteria for their membership in the category, or, on the contrary, blur the boundaries of the categories. As a result, the same instance can be attributed to several categories, or entities that have only a few common properties fall into the same category. Of course, cognitive analysis admits that natural categories tend to be fuzzy along their boundaries and inconsistent in the status of their constituent members, but the necessary and sufficient conditions on which the logic of predicates is built is rarely found in the categories of natural things, especially when it comes to such manifestations of human life as feelings, emotions, relations of modality, etc., and the question of whether a given mental entity belongs to one category or another is decided in the context. The question of the comparative merits and applicability of various categorization theories is also still controversial. The classical approach, especially enhanced by machine learning and fuzzy set theory (Fisher,

1987) applies to the categorization of scientific knowledge and uniquely identifiable objects (data). As applied to everyday life (open domain), it is better to rely on similarity-based theories due to their more grounded psychological foundation.

Above mentioned theories of categorization are focused on the inner content of categories, their essence. The internal knowledge structures of the various categorization models reflect the specific representations they use in the categorization process, which typically include patterns, prototypes, and rules. However, this is only one aspect of the problem. Theories of categorization do not consider the impact of knowledge transformation being produced both individually and collectively in the practice of organizations on the categorization process (Moroz, 2020) The mutual transformation of tacit and explicit knowledge in the process of formalization of both individual and collective knowledge is sure to play, under certain conditions, the role of constraints for the categorization process.

3. Semantic Representation of Software Requirements Architecture

3.1 Relationships between Models in Software Requirements Architecture

According to the well-known definition, a project ontology is a formalized description of knowledge about the design process of new or modernization of already known artifacts, including knowledge about the design object itself and artifacts close to it in properties, as well as a thesaurus or domain ontology. So, the project ontology should ensure the collection and processing of information about the design object as a system based on the study of the conceptual apparatus of the domain, analysis of deductive and inductive rules, and functioning models of the system. The model-driven development approach provides creating a domain model. Based on the domain model, a project requirements architecture can be created, including objectives, people, systems, data, and risk models. A software development model provides a systematic approach to software development, however, among the models of the software product life cycle known today, none of them has an absolute advantage, although in most cases the agile model is preferred. The choice of the correct software development model depends on many factors, among which the main ones include: the maturity of the development team, the requirements for the project such as goals and budget of the project, and a singularity of the target system.

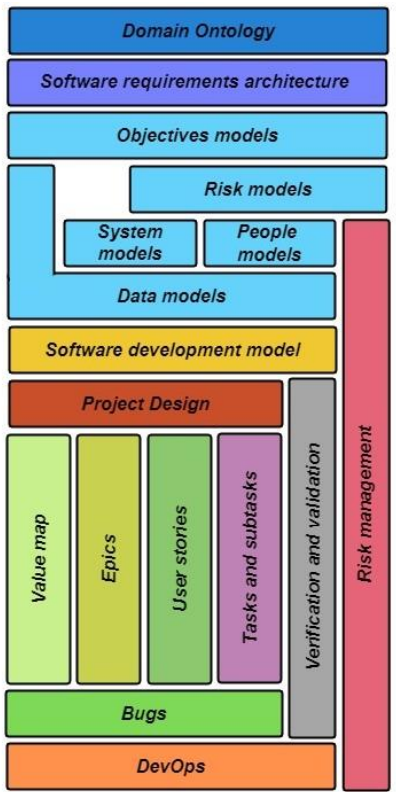

Figure 1. A software development technology stack centered on the software requirements architecture

In the software development technology stack, requirements are a key artifact, it influences all other project artifacts and is directly related to domain knowledge (figer 1). Nevertheless, the development of project requirements seems is not given due attention, at least we are not aware of other publications, except (Beatty & Chen, 2015), where a system analysis of the architecture of requirements would be carried out. It may look a little bit confusing that Beatty and Chen did not include risk models in their taxonomy of requirements classes. However, considering the numerous cascading relationships of risk models with the listed requirements models, especially with people models, this would significantly complicate their task, which is already quite complex though they handled it flawlessly. Without saying, risk models in a software project make a separate topic for research. Nevertheless, the above four main classes of requirements models should be surely added by one more – the risk models.

Software development process is based on the chain of artifacts: <objectives> – <features> – <processes> – <use cases> – <user interface> – <data>. Indeed, a software system is known to be created to achieve certain goals, providing functional support for a certain technological process, for which users of this system must perform certain actions within this process, using the interface of the software system and the necessary data for this. At the same time, creating such a system as well as the achievement of its goals may be associated with overcoming certain risks. This chain defines a mainstream of the requirements architecture.

Like the architecture of the system itself, the requirements architecture is obviously multi-level. Inside it, models form a hierarchy, which is rooted in objectives models (figer 2). Objectives models' level is followed by risk models' level. Closely interconnected people and systems models depend on risk models, although the human factor is dominant in terms of project risk management. And data models occupy the lowest level of the hierarchy. In figer 2, solid lines show the main dependencies, creating the core of the requirements architecture that covers most of the relationships between the requirements models. The dotted lines correspond to auxiliary dependencies, which additionally characterize the main links. Within the model categories, component models also form hierarchical structures rooted in the above-mentioned bounding models. Each bounding model bounds the corresponding space: objective space, people space, system space, data space, and risks space.

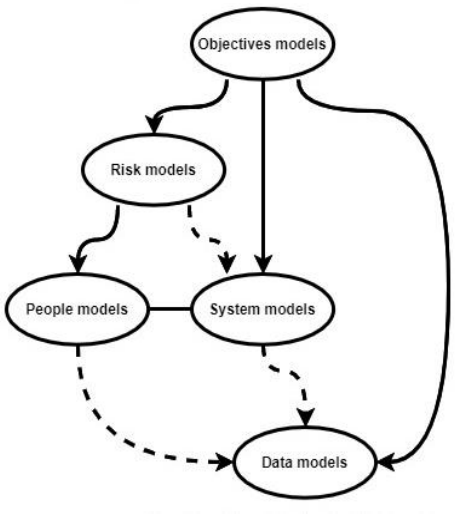

Fig. 2. Hierarchy of model types in requirements architecture

3.2 Software Requirements Ontological Framework

Ontology is a metaknowledge, linking element between non-formalized knowledge and its formal representation. The presence of such a link is inevitable if knowledge is in a passive form concerning knowledge processing programs. The situation is complicated by the need to use ontologies of several levels, that is, meta-metaknowledge, and the coordination of different languages of ontology representation and ontology pattern languages (Almeida Falbo et al., 2013), as well as due to the problem of combining several ontologies into one. This article suggests using instead of multilevel semantic metamodels a model of self-organizing knowledge tree (Moroz, 2020), which is based on the emergent ability of knowledge to self-organize in the permanent cognitive process of creating new knowledge. The knowledge tree represents the conceptual model of knowledge representation, arising from the hypothesis about collective interaction of a lot of intellectual atomic elements of knowledge (the knowledge quanta), which are combined into clusters like neurons in the brain. The knowledge quanta are elementary concepts and facts, from which more complex concepts are formed. Semantically closed concepts are combined into clusters, and, finally, the clusters are tied to specific nodes of the knowledge tree. Knowledge quanta, clusters, and knowledge tree nodes correspond to the mental representations of accordingly subordinate, basic, and superordinate levels of categorization. For binding the elements of knowledge at all levels, transcendental logic-based ontological relations are used. Using the universal principle of self-organization of knowledge allows solving the problem of ontology reasoning arising from the traditional use of formal logic. In other words, the knowledge tree is an executable transcendental logic-based ontology framework that can provide a solution to the problem of ontology learning automation. And in this paper, this knowledge representation model is used to develop an executable semantic framework for automated ontology generation from informal software requirements text. To describe the semantics relations between requirements models, "ontological predicates" are used, which are transcendental logic-based ontological relations followed from the I. Kant's categories of reason. Their peculiarity is that they do not show the properties of objects they relate to but indicate how we should, guided by them, reveal the properties and connections of experience objects in general. The philosophical rationale for all "ontological predicates" and a more detailed discussion of them with examples are described in the paper (Moroz, 2021).

Thereby, a precedent of semantic representation formalism for the requirements architecture, based on the knowledge representation model in the form of a knowledge tree, is created. The main benefit of this model is its flexibility, it ensures the accumulation of knowledge without the need to change the initial infrastructure to enable semantic interoperability, as well as that the ontology inference engine is part of the mechanism of collective interaction of active elements of knowledge, and not some externally programmed system of rules that imitate the process of thinking.

As an example of a requirements architecture for identifying semantic relationships between individual models both within and between categories, an instance of the requirements architecture borrowed from (Beatty & Chen, 2015). It contains typical relationships between requirements models that are most often encountered in practice. It is hard, if it is possible at all, to characterize these relationships using descriptive logic formalism due to their diversity and contextual dependence. One can only speak of the influence of some models on others, which did in (Beatty & Chen, 2015). The resulting ontology chart of a typical requirements architecture based on the "ontological predicates" is shown in figure 3. A brief description of those "ontological predicates" that were identified because of the analysis of relationships between models in the selected requirement architecture as well as corresponding color marking of the connections are given in table 2. It must be noted that only the most common relationships are presented in figure 3. Those relationships between models that are much less common can be attributed to another type of ontological relation setting the randomness of the established relationship, i.e., the relation of chaos. Such relations are not shown in figure 3 not to overload the drawing.

Finally, the requirements architecture ontology in the form of a knowledge tree may look like this. Each type of requirements model forms a cluster in such a knowledge tree. The individual requirements models form sub-clusters that are made up of context-sensitive instances of these models. The instances of requirements models combine a set of concepts represented by knowledge quanta or clusters. Both knowledge quanta and clusters are related through "ontological predicates". Binding requirements models to each other is implemented by placing a copy of the dependent model into the cluster of the model on which it depends. However, due to the ability of the knowledge quanta to independently search for the concepts necessary by the meaning of the problem being solved in the knowledge tree (knowledge quanta and clusters), this linking occurs directly, without using any external search mechanisms, except for the "native" tool of the knowledge quantum itself to search relevant information in the knowledge tree.

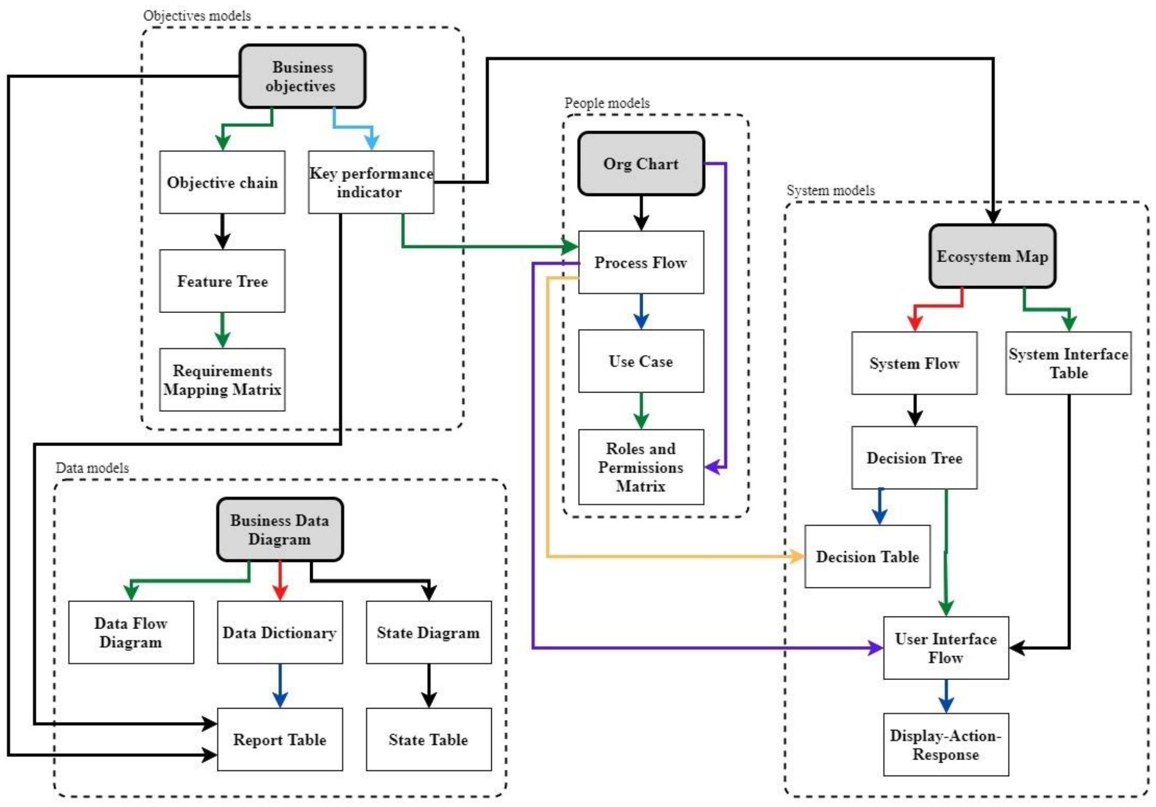

Figure 3. The ontological chart of the typical requirement architecture

Table 2. Transcendental logic-based ontological relations

Ontological relation	Description	Color marking
Relation of aggregation	Part-whole or part-of relation.	🟥
Relation of order	More–less, higher–lower, or previous–subsequent relation.	🟨
Relation of causality	Cause–effect relation.	⬛
Relation of tactility	Sets degree of semantic proximity between different concepts.	🟩
Relation of wisdom	Sets influence on mental abilities in a broad sense like logical–illogical, reasonable–stupid etc.	🟦
Relation of variability	Sets the ability to change the concept itself that is its state or properties.	🟦
Relation of effectiveness	Sets the ability to generate control instructions, the desire to be first, to make others dependent on yourself or others.	🟪

4. Conclusion

Agile methodology has already proven its usefulness and productivity for software development, but it has not fully unleashed its full potential yet, so there is some work to be done for researchers and practitioners alike. For an Agile approach to be successful, it must be backgrounded by the three “elephants”: continuous workflow process improvement (or Kaizen), team self-organizing, and knowledge self-organizing. Though it is relatively clear concerning the first two, and they are already in work, the latter still awaits its development. The constantly growing interest of researchers and practitioners in the use of artificial intelligence methods in the software life cycle indicates the need to develop an approach to knowledge management in a project that would not only provide decision support at different stages of the product life cycle but also create preconditions and context for the interaction of project artifacts (requirements, design, source code, test cases, and others) with the domain knowledge management system. To achieve this goal, it is necessary, first, to unify the methods of formalizing both project knowledge and domain knowledge. However, for this, an appropriate knowledge representation model is needed. Such a model can be created based on the principle of self-organization of knowledge, that is if knowledge has an inherent emergent ability to create new knowledge from existing knowledge and form a treelike structure in memory, a knowledge tree, to store the knowledge. Nodes of this knowledge tree consist of semantically closed concepts connected by transcendental logic-based ontological relations. In this paper, this knowledge representation model is used to develop an executable semantic framework for automated ontology generation from unstructured informal software requirements text. To describe the semantics relations between requirements models, "ontological predicates" are used, thereby creating a precedent of semantic representation formalism for the requirements architecture, based on the knowledge representation model in the form of a knowledge tree. The main benefit of this model is that it is flexible and ensures the accumulation of knowledge without the need to change the initial infrastructure to enable semantic interoperability, as well as that the ontology inference engine is part of the mechanism of collective interaction of active elements of knowledge, and not some externally programmed system of rules that imitate the process of thinking.

5. Discussion

In the field of using artificial intelligence for the development and maintenance of complex software systems, the current state of research is characterized by an abundance of different approaches and methods and an almost full absence of completed end-to-end enterprise solutions. This indirectly indicates weakness of fundamental concepts and models of knowledge representation and knowledge-based management and even more from models of thinking. Thus, the statement by John Searle about narrow AI that it “would be useful for testing hypotheses about minds but would not actually be minds” (Frankish & Ramsey, 2014) is relevant still now.

The knowledge representation model is key to the concept of artificial intelligence, however, knowledge is inherently an active substance, and, like thinking, it is a form of the existence of the mind as a being. The processes of knowledge accumulation and thinking are inextricably linked with each other; therefore, the artificial intelligence model should combine these two forms of being of the mind. Our initial hypothesis is that knowledge has an emergent ability to self-organize in the process of human cognition of the world. The knowledge tree model, used in this paper as a formalism for the semantic representation of the requirements architecture of a software project, is an instance of such a model. Exactly, this type of model is needed to create a fully automated unsupervised semantic parsing tool and, finally, a knowledge-driven system, which would be able to provide knowledge and control over the development process of a software system adapted to evolutionary changes in the domain knowledge associated with the system. However, this is surely no longer a task of the scope of narrow AI but of general AI.

References

Aiello, B., & Sachs, L. A. (2016). *Agile application lifecycle management: using devops to drive process improvment*. Boston, MA: Addison-Wesley.

Almeida, F. R. de, Barcellos, M. P., Nardi, J. C., & Guizzardi, G. (2013). Organizing Ontology Design Patterns as Ontology Pattern Languages. In P. Cimiano, Ó. Corcho, V. Presutti, L. Hollink & S. Rudolph (Eds.), *ESWC* (p./pp. 61-75): Springer. https://doi.org/10.1007/978-3-642-38288-8_5

Ameller, D., & Xavier, F. G. (2014). *Non-functional requirements as drivers of software architecture design*. Barcelona: Universitat Politècnica de Catalunya. Retrieved from http://hdl.handle.net/2117/95316

Asim, M. N., Wasim, M., Khan, M. U. G., Mahmood, W., & Abbasi, H. M. (2018). A survey of ontology learning techniques and applications. *Database: the Journal of Biological Databases and Curation, 2018*. https://doi.org/10.1093/database/bay101

Association for Ontology Design & Patterns (n.d.). *OPTypes.* Retrieved from http://ontologydesignpatterns.org/wiki/OPTypes

Balikuddembe, J. K., Osunmakinde, I. O., & Bagula, A. (2009). Software Project Profitability Analysis Using Temporal Probabilistic Reasoning; An Empirical Study with the CASSE Framework. In Kim H., Kim T., Kiumi A. (eds) *Advances in Security Technology. SecTech 2008. Communications in Computer and Information Science, vol 29*. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-10240-0_11

Barenkamp, M., Rebstadt, J., & Thomas, O. (2020). Applications of AI in classical software engineering. *AI Perspect 2*, 1. https://doi.org/10.1186/s42467-020-00005-4

Batarseh, F. A., Mohod, R., Kumar, A., & Bui, J. (2021). 10 - The application of artificial intelligence in software engineering: a review challenging conventional wisdom. Editor(s): Feras A. Batarseh, Ruixin Yang, *Data Democracy*, Academic Press, Pages 179-232. https://doi.org/10.1016/B978-0-12-818366-3.00010-1

Beatty, J., & Chen, A. (2015). *Visual models for software requirements*. Redmond, Wash: Microsoft Press.

Bhatia, M. P. S., Kumar, A., & Beniwal, R. (2018). Ontology Driven Software Development for Automated Documentation. *Webology, 15*. Retrieved from https://www.webology.org/data-cms/articles/20200515032021pma174.pdf

Burge, S., & Burge Hughes Walsh Limited (n.d.). *An Overview of the Soft Systems Methodology*. Retrieved from https://www.burgehugheswalsh.co.uk/Uploaded/1/Documents/Soft-Systems-Methodology.pdf

Chaves-González, J. M., Pérez-Toledano, M. A., & Navasa, A. (2015). Software requirement optimization using a multiobjective swarm intelligence evolutionary algorithm. *Knowledge-Based Systems, 83*, 105-115. https://doi.org/10.1016/j.knosys.2015.03.012

Chen, R., Chen, C. H., Liu, Y., & Ye, X. (2020). Ontology-based requirement verification for complex systems. *Advanced Engineering Informatics, 46*, 101148. https://doi.org/10.1016/j.aei.2020.101148

Cheng, J., Reddy, S., & Lapata, M. (2018). Building a Neural Semantic Parser from a Domain Ontology. *CoRR, abs/1812.10037*. Retrieved from http://arxiv.org/abs/1812.10037

Choi, E., Kwiatkowski, T., & Zettlemoyer, L. (2015). Scalable Semantic Parsing with Partial Ontologies. *ACL* (1) (p./pp. 1311-1320): The Association for Computer Linguistics. https://doi.org/10.3115/v1/P15-1127

Cimatti, A., Roveri, M., Susi, A., & Tonetta, S. (2009). From Informal Requirements to Property-Driven Formal Validation. *Lecture Notes in Computer Science, 5596*, 166-181. https://doi.org/10.1007/978-3-642-03240-0_15

Cohen, H., & Lefebvre, C. (eds.) (2017). *Handbook of categorization in cognitive science* (2nd ed.). Cambridge, MA: Elsevier.

Corcho, O., Fernández-López, M., & Gómez-Pérez, A. (2007). Ontological Engineering: What are Ontologies and How Can We Build Them? In Cardoso, J. (Ed.), *Semantic Web Services: Theory, Tools and Applications* (pp. 44-70). IGI Global. http://doi:10.4018/978-1-59904-045-5.ch003

Dell' Anna, D., Dalpiaz, F., & Dastani, M. (2019). Requirements-driven evolution of sociotechnical systems via probabilistic reasoning and hill climbing. *Automated Software Engineering: an International Journal, 26*(3), 513-557. https://doi.org/10.1007/s10515-019-00255-5

Dey, A. K. (2001). Understanding and Using Context. *Personal and Ubiquitous Computing, 5*, 4-7. https://doi.org/10.1007/s007790170019

Dmitriev, S., & JetBrains, S. R. O. (2004). *Language Oriented ProgrammingLanguage Oriented Programming: The Next Programming Paradigm*. Retrieved from https://resources.jetbrains.com/storage/products/mps/docs/Language_Oriented_Programming.pdf

Dromey, R. G. (2003). From requirements to design: formalizing the key steps. *First International Conference onSoftware Engineering and Formal Methods, Proceedings*, pp. 2-11. https://doi.org/10.1109/SEFM.2003.1236202

Ejigu, D., Scuturici, V. M. & Brunie, L. (2007). An Ontology-Based Approach to Context Modeling and Reasoning in Pervasive Computing. *PerCom Workshops* (p./pp. 14-19): IEEE Computer Society. https://doi.org/10.1109/PERCOMW.2007.22

Evans, E. (2015). *Domain-driven design reference: Definitions and patterns summaries*. Dog Ear Publishing, LLC. Retrieved from https://www.domainlanguage.com/wp-content/uploads/2016/05/DDD_Reference_2015-03.pdf

Fatma, B., Özlem, A., & Deniz, K. (2021). Linking software requirements and conceptual models: A systematic literature review. *Engineering Science and Technology, an International Journal, 24*(1), 71-82. https://doi.org/10.1016/j.jestch.2020.11.006

Feras, A. B., Rasika, M., Abhinav, K., & Justin, B. (2020). 10 - The application of artificial intelligence in software engineering: a review challenging conventional wisdom. Editor(s): Feras A. Batarseh, Ruixin Yang, *Data Democracy,* Academic Press, Pages 179-232. https://doi.org/10.1016/B978-0-12-818366-3.00010-1

Fisher, D. H. (1987). Knowledge acquisition via incremental conceptual clustering. *Mach Learn, 2*, 139-172. https://doi.org/10.1007/BF00114265

Frankish, K., & Ramsey, W. M. (eds.) (2014). *The Cambridge Handbook of Artificial Intelligence*. Cambridge, UK: Cambridge University Press. https://doi.org/10.1017/CBO9781139046855

Gangemi, A., Gomez-Perez, A., Presutti, V., & Suarez-Figueroa, M. (2007). Towards a Catalog of OWL-based Ontology Design Patterns. *12th Conference of the Spanish Association for Artificial Intelligence*, CAEPIA, Salamanca (Spain). Retrieved from https://oa.upm.es/5212/1/Towards_a_Catalog_of_OWL-based_Ontology_Design_Patterns.pdf

Ghazel, M., Yang, J., & El-Koursi, E. M. (2015). A pattern-based method for refining and formalizing informal specifications in critical control systems. *Journal of Innovation in Digital Ecosystems, 2*, 1-2, 32-44. https://doi.org/10.1016/j.jides.2015.11.001

Giudice, D. L. (2016). *How AI Will Change Software Development And Applications*. [Research report] Retrieved from https://www.forrester.com/report/How-AI-Will-Change-Software-Development-And-Applications/RES121339

Gomez, F. J., Aguilera, M. A., Olsen, S. H., & Vanfretti, L. (September 01, 2020). Software requirements for interoperable and standard-based power system modeling tools. *Simulation Modelling Practice and Theory, 103*. https://doi.org/10.1016/j.simpat.2020.102095

Grefenstette, E., Blunsom, P. L., de Freitas, N., & Hermann, K. M. (2014). *A Deep Architecture for Semantic Parsing*. https://doi.org/10.3115/v1/W14-2405

Gregory, F. H., & Warwick Business School. (1993). *A logical analysis of soft systems modelling: Implications for information system design and knowledge-based system design* (Unpublished PhD thesis). University of Warwick, Coventry, UK. Retrieved from http://wrap.warwick.ac.uk/2888/

Gregory, P., Barroca, L., Sharp, H., Deshpande, A., & Taylor, K. J. (2016). The challenges that challenge: engaging with agile practitioners' concerns. *Information and Software Technology, 77,* 92-104. https://doi.org/10.1016/j.infsof.2016.04.006

Gruber, T. R. (1993). A translation approach to portable ontology specifications. *Knowledge Acquisition, 5*(2), 199-220. https://doi.org/10.1006/knac.1993.1008

Gu, T., Wang, X., Pung, H. K., & Zhang, D. Q. (2020). An Ontology-based Context Model in Intelligent Environments. *CoRR, abs/2003.05055*. Retrieved from https://arxiv.org/ftp/arxiv/papers/2003/2003.05055.pdf

Harman, M., & Chicano, F. (2015). Search Based Software Engineering (SBSE). *Journal of Systems and Software, 103*, 266. https://doi.org/10.1016/j.jss.2015.01.051

Iqbal, T., Elahidoost, P., Lucio, L., & 25th Asia-Pacific Software Engineering Conference, APSEC 2018. (2019). A Bird's Eye View on Requirements Engineering and Machine Learning. *Proceedings - Asia-Pacific Software Engineering Conference, Apsec,* 11-20. https://doi.org/10.1109/APSEC.2018.00015

Jackson, E. K., Seifert, D., Dahlweid, M., Santen, T., Bjorner, N., & Schulte, W. (2009). *Specifying and Composing Non-functional Requirements in Model-Based Development*. Bergel, Alexandre; Software Composition; 72-89; Springer Berlin Heidelberg: Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-02655-3_7

Jayagopal, A., Kaushik, R., Krishnan, A., Nalla, R., & Ruttala, S. (February 01, 2021). Bug classification using machine and Deep Learning Algorithms. *Materials Today: Proceedings*. https://doi.org/10.1016/j.matpr.2021.01.188

Kääriäinen, J. (2011). *Towards an Application Lifecycle Management Framework: Dissertation*. VTT Technical Research Centre of Finland. Retrieved from http://www.vtt.fi/inf/pdf/publications/2011/P759.pdf

Kamath, A., & Das, R. (2018). *A Survey on Semantic Parsing*. Retrieved from http://arxiv.org/abs/1812.00978

Karpathy, A. (2017). *Software 2.0.* [Blog post] Retrieved from https://karpathy.medium.com/software-2-0-a64152b37c35

Knublauch, H. (2004). Ontology-Driven Software Development in the Context of the Semantic Web: An Example Scenario with Protege/OWL. In D. S. Frankel, E. F. Kendall & D. L. McGuinness (Eds.), *1st International Workshop on the Model-Driven Semantic Web (MDSW2004).* Retrieved from http://knublauch.com/publications/MDSW2004.pdf

Kulyamin, V. V. et al. (n.d.). *Formalizatsiya trebovaniy na praktike.* [Formalization of requirements in practice]. Retreived from http://panda.ispras.ru/~kuliamin/docs/Req-2006-ru.pdf

Kurasinska, L., Frak, M., & STX Next sp. z o.o (2019, September 13). *Will Artificial Intelligence Replace Software Developers?* [Blog post] Retrieved from https://www.stxnext.com/blog/will-artificial-intelligence-replace-developers/

Lavbič, D., & Bajec, M. (2011). Rapid Development of Executable Ontology for Financial Instruments and Trading Strategies. In: Mohamad Zain J., Wan Mohd W.M.., El-Qawasmeh E. (eds) Software Engineering and Computer Systems. ICSECS 2011. *Communications in Computer and Information Science, 179*. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-22170-5_21

Lehmann, J., & Völker, J. (2014). An Introduction to Ontology Learning. In J. Lehmann & J. Voelker (ed.), *Perspectives on Ontology Learning* (pp. ix-xvi). AKA / IOS Press. Retrieved from https://www.semanticscholar.org/paper/An-Introduction-to-Ontology-Learning-Lehmann-V%C3%B6lker/1f1936981ce32621d69c47a2b9eaa1fcca5abbb7

Leybourn, E., & Hastie, S. (2018). *#NOPROJECTS: a culture of continuous value*. [Place of publication not identified]: LULU COM.

Li, D., Deng, L., Zeng, X., & Cai, Z. (2021). Dynamic simulation modelling of software requirements changes management system. *Microprocessors and Microsystems, 83*. https://doi.org/10.1016/j.micpro.2021.104009

Li, L., Harman, M., Wu, F., & Zhang, Y. (2017). The Value of Exact Analysis in Requirements Selection. Ieee *Transactions on Software Engineering, 43*(6), 580-596. https://doi.org/10.1109/TSE.2016.2615100

Liddy, E. D. (2001). *Natural Language Processing*. In Encyclopedia of Library and Information Science, 2nd Ed. NY. Marcel Decker, Inc. Retrieved from https://surface.syr.edu/cnlp/11/

Lukša, M. (2017). *Kubernetes in Action*. Manning Publications. https://doi.org/10.3139/9783446456020.fm

Madhan, V., Kalaiselvi, V. K. G., & Donald, J. P. (2017). 2nd International Conference on Computing and Communications Technologies (ICCCT) (2017). *Tool development for formalizing the requirement for the safety critical software engineering process*. pp. 161-164. Retrieved from htpps://10.1109/ICCCT2.2017.7972273

Maedche, A., & Staab, S. (2001). Ontology learning for the Semantic Web. *IEEE Intelligent Systems, 16*(2), 72-79. https://doi.org/10.1109/5254.920602

Miao, W., Wang, X., & Liu, S. (2015). A Tool for Supporting Requirements Formalization Based on Specification Pattern Knowledge. *International Symposium on Theoretical Aspects of Software Engineering (TASE),* 127-130. https://doi.org/10.1109/TASE.2015.13

Miller, G. J. (2013). *Agile problems, challenges, & failures*. PMI® Global Congress 2013 – North America, New Orleans, LA. Newtown Square, PA: Project Management Institute. Retrieved from https://www.pmi.org/learning/library/agile-problems-challenges-failures-5869

Mohammad, S., Laith, A., Muath, I. J., Osama, A. A., & Mohammad, S. D. (2020) (AIAM2019) Artificial Intelligence in Software Engineering and inverse: Review. *International Journal of Computer Integrated Manufacturing, 33*(10-11), 1129-1144. https://doi.org/10.1080/0951192X.2020.1780320

Mok, J., & Min, H. (2015). Ontology-based context-aware model by applying Bayesian network. *FSKD* (p./pp. 2660-2664), : IEEE. https://doi.org/10.1109/FSKD.2015.7382377

Moradi, B., Daclin, N., Chapurlat, V., & 19th IFIP WG 5.5 Working Conference on Virtual Enterprises, PRO-VE 2018. (2018). Formalization and Evaluation of Non-functional Requirements: Application to Resilience. *Ifip Advances in Information and Communication Technology, 534*, 124-131. https://doi.org/10.1007/978-3-319-99127-6_11

Moroz, O. V. (2020). Model of Self-organizing Knowledge Representation and Organizational Knowledge Transformation. *American Journal of Artificial Intelligence, 4*(1), 1-19.

https://doi.org/10.11648/j.ajai.20200401.11

Moroz, O. V. (2021). Universal Transcendental Logic-Based Ontology. *Computer and Information Science, 14*(2). https://doi.org/10.5539/cis.v14n2p1

Narrative Science. (2018). *Outlook on Artificial Intelligence in the Enterprise*. [Research report] Retrieved from https://narrativescience.com/wp-content/uploads/2019/02/Research-Report_Outlook-on-AI-for-the-Enterprise.pdf

Naumchev, A., & Meyer, B. (2017). Seamless requirements. *Computer Languages, Systems & Structures*, *49*, 119-132. https://doi.org/10.1016/j.cl.2017.04.001

Pan, J. Z., Steffen S., Aßmann, U., Ebert, J., Zhao, Y., &. Springer-Verlag GmbH. (2015). *Ontology-Driven Software Development.* Berlin: Springer Berlin. https://doi.org/10.1007/978-3-642-31226-7

Pandey, S. K., Mishra, R. B., & Tripathi, A. K. (2021). Machine learning based methods for software fault prediction: A survey. *Expert Systems with Applications, 172*. https://doi.org/10.1016/j.eswa.2021.114595

Peres, F., Yang, J., & Ghazel, M. (2012). A Formal Framework for the Formalization of Informal Requirements. *The International Journal of Soft Computing and Software Engineering, 2*(8), 14-27. https://doi.org/10.7321/jscse.v2.n8.2

Perkusich, M., Chaves, S. L., Costa, A., Ramos, F., Saraiva, R., Freire, A., Dilorenzo, E., ... Perkusich, A. (2020). Intelligent software engineering in the context of agile software development: A systematic literature review. *Information and Software Technology, 119*. https://doi.org/10.1016/j.infsof.2019.106241

Poon, H., & Domingos, P. M. (2010). Unsupervised Ontology Induction from Text. In J. Hajic, S. Carberry & S. Clark (eds.), *ACL* (p./pp. 296-305): The Association for Computer Linguistics. Retrieved from https://www.aclweb.org/anthology/P10-1031.pdf

Ramírez, A., Romero, J. R., & Ventura, S. (2019). A survey of many-objective optimisation in search-based software engineering. *Journal of Systems and Software, Volume 149*, 382-395. https://doi.org/10.1016/j.jss.2018.12.015

Ransbotham, S., Kiron, D., Gerbert, P., Reeves, M., & Massachusetts Institute of Technology (2017). *Reshaping business with artificial intelligence. Closing the gap between ambition and action*. REPRINT #: 59181. Retrieved from https://sloanreview.mit.edu/projects/reshaping-business-with-artificial-intelligence/#chapter-10

Rosa, N. S., Freire Cunha, P. R., & Ribeiro Justo, G. R. (2004). An approach for reasoning and refining non-functional requirements. *J Braz Comp Soc, 10*, 62-84. https://doi.org/10.1007/BF03192354

Rossberg, J. (2014). *Beginning application lifecycle management*. Bercley, CA: APRESS. https://doi.org/10.1007/978-1-4302-5813-1

Ruchika, M., Megha, K., & Rajeev, R. R. (2017). On the application of search-based techniques for software engineering predictive modeling: A systematic review and future directions. *Swarm and Evolutionary Computation, 32*, 85-109. https://doi.org/10.1016/j.swevo.2016.10.002

Salih, A. M., Omar, M., & Yasin, A. (January 01, 2018). *Understanding Uncertainty of Software Requirements Engineering: A Systematic Literature Review Protocol*. In: Requirements Engineering for Internet of Things. Springer, Singapore, pp. 164-171. http://doi.org/10.1007/978-981-10-7796-8_13

Schalley, A. C. (2019). Ontologies and ontological methods in linguistics. Lang. *Linguistics Compass, 13.* https://doi.org/10.1111/lnc3.12356

Shankar, P., Morkos, B., & Yadav, D. et al. (2020). Towards the formalization of non-functional requirements in conceptual design. *Res Eng Design 31*, 449-469. https://doi.org/10.1007/s00163-020-00345-6

Shehab, M., Abualigah, L., Jarrah, M. I., Alomari, O. A., & Daoud, M. S. (January 01, 2020). (AIAM2019) Artificial Intelligence in Software Engineering and inverse: Review. *International Journal of Computer Integrated Manufacturing, 33*, 1129-1144. https://doi.org/10.1080/0951192X.2020.1780320

Siddiqui, F., & Alam, M. A. (2013). Ontology Based Feature Driven Development Life Cycle. *CoRR, abs/1307.4174*. Retrieved from https://arxiv.org/ftp/arxiv/papers/1307/1307.4174.pdf

Sowa, J. F. (2007). Language games, a foundation for semantics and ontology. *Current Research in the Semantics/pragmatics Interface, 18*, 17-37.

Sowa, J. F. (2012). *Knowledge representation: Logical, philosophical, and computational foundations*. Boston: Course Technology. https://doi.org/10.1163/9780080548524_003

Standish Group. (2018). *Project Resolution Benchmark*. [Research report] Retrieved from https://www.standishgroup.com/sample_research_files/DemoPRBR.pdf

Starc, J., & Mladenic, D. (2016). Joint learning of ontology and semantic parser from text. *CoRR, abs/1601.00901*. Retrieved from http://arxiv.org/abs/1601.00901

Stern, D. G. (1996) *Wittgenstein on mind and language*. Oxford: Oxford University Press. https://doi.org/10.1093/0195080009.001.0001

UCL - SST/ICTM/INGI - Pôle en ingénierie informatique, & van Lamsweerde, A. (2009). *Requirements Engineering: From System Goals to UML Models to Software Specifications*. Wiley. Retrieved from http://hdl.handle.net/2078.1/78351

Valiente, M.-C., Vicente-Chicote, C., & Rodriguez, D. (2011). An Ontology-Based and Model-Driven Approach for Designing IT Service Management Systems. *International Journal of Service Science, Management, Engineering, and Technology, 2*(2), 65-81. https://doi.org/10.4018/jssmet.2011040104

Wang, P., & Hofstadter, D. (2006). A logic of categorization. *Journal of Experimental & Theoretical Artificial Intelligence, 18*(2), 193-213. https://doi.org/10.1080/09528130600557549

Warren, P., Mulholland, P., Collins, T., & Motta, E. (2014). The Usability of Description Logics: Understanding the Cognitive Difficulties Presented by Description Logics. *Lecture Notes in Computer Science, 8465*, 550-564. https://doi.org/10.1007/978-3-319-07443-6_37

Wikipedia. (n.d.). *Agile software development*. Retrieved from https://en.wikipedia.org/wiki/Agile_software_development

Withall, S. (2010). *Software Requirement Patterns*. Sebastopol: Microsoft Press.

Xu, J., Chen, R., & Du, Z. (2016) Probabilistic reasoning in diagnosing causes of program failures. *Softw. Test. Verif. Reliab., 26*, 176-210. https://doi.org/10.1002/stvr.1583

Zyl, J., & Corbett, D. (2001). *A Framework for Comparing the use of a Linguistic Ontology in an Application*. Retrieved from http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.28.7212