


# A Proposal of Naturalistic Software Development Method


**Lizbeth Alejandra Hernández-González**

(Universidad Veracruzana, Xalapa, Veracruz, México)

 <https://orcid.org/0000-0002-8524-3209>, [lizhernandez@uv.mx](mailto:lizhernandez@uv.mx))


**Ulises Juárez-Martínez**

(Tecnológico Nacional de México Campus Orizaba, Veracruz, México,

 <https://orcid.org/0000-0002-5911-3136>, [ujuarez@ito-depi.edu.mx](mailto:ujuarez@ito-depi.edu.mx))


**Jezreel Mejía**

(Centro de Investigación en Matemáticas, Zacatecas, México)

 <https://orcid.org/0000-0003-0292-9318>, [jmejia@cimat.mx](mailto:jmejia@cimat.mx))

**Alberto Aguilar-Laserre**

(Tecnológico Nacional de México Campus Orizaba, Veracruz, México,

 <https://orcid.org/0000-0001-9813-9657>, [aaguilar@ito-depi.edu.mx](mailto:aaguilar@ito-depi.edu.mx))

**Abstract:** Naturalistic programming purports to include natural language elements in programming languages to increase software expressiveness. Even though natural language is inherently ambiguous, it is richer and thus more expressive than any artificial language. Currently, the Naturalistic Programming Paradigm (NPP) is supported by its conceptual model and three general-purpose naturalistic programming languages that can generate executable binary code. Nevertheless, to date, no research efforts have been concentrated on applying the NPP within a software development process. To address this gap, in this article, we propose a naturalistic software development method to test the advantages of the NPP. The method focuses on the analysis and design stages of the software development process and seeks to contribute to closing the gap between the problem and the solution domains. We also present an example of an implementation using Cal-4700, a naturalistic programming language, showing the differences in expressiveness of programming with a traditional programming language, like Python.

**Keywords:** Naturalistic Programming Method, Naturalistic Programming, Programming Language Expressiveness, Programming Paradigms, Software Engineering Methods, Software Engineering Processes

**Categories:** D.2.1, D.2.3, D.2.6, D.2.10

**DOI:** 10.3897/jucs.105637

## 1 Introduction

Software engineering relies on a wide range of quality software development methods, process models, techniques, and standards, among others [Jacobson et al. 12] [ISO/IEC 16]. However, when programmers fail to understand the abstractions used across the different software development stages, information is lost. Very often, user requirements must be translated into programming formalisms, instead of being written more transparently [Daun et al. 23]. Such a lack of expressiveness when programming

makes the gap between the problem domain and the solution domain open, increases rework costs, and causes users to be partially satisfied with their software product [Lopes et al. 03].

Lack of expressiveness (“The quality of expressing somebody’s thoughts and feelings,” according to the Oxford English Dictionary) has been long-standing problem in software development. It refers to programming languages being unable to clearly convey their ideas and get closer to the expert domain [Pulido-Prieto, Juárez-Martínez 17]. Conversely, expressive programming languages are able to transparently translate what users want into software requirements. Programming languages should stand as close as possible to the expert domain, since sometimes programmers fail to write software requirements as needed by their customers. In the process of translating software requirements into code, valuable information is often lost. That is, due to their current characteristics, programming languages unintentionally force programmers to write ideas that lie far from the best coding solution [Pulido-Prieto, Juárez-Martínez 17].

The Naturalistic Programming Paradigm (NPP) incorporates elements of natural language into general-purpose programming languages to increase expressiveness, thus decreasing the gap between the problem domain and the solution domain. Naturalistic programming aims at having general-purpose languages that implement natural language elements, such as reflection (i.e. ability to refer to previously or subsequently mentioned elements), and generate directly executable code without relying on programming abstractions [Pulido-Prieto, Juárez-Martínez 19]. Naturalistic programming languages would make the programming task more natural and outcome-focused, thus freeing programmers from great mental loads and the need to learn technical programming aspects.

As reported in the literature, the NPP has proven to be a valuable solution to a series of problems. It is also supported by its own conceptual model [Pulido-Prieto, Juárez-Martínez 19] and three general-purpose programming languages: Pegasus [Knöll, Mezini 06], Cal-4700 [Rzeppa, Rzeppa 19a] and SN [Pulido-Prieto, Juárez-Martínez 20]. However, the NPP has not yet been incorporated into software development processes to test its benefits, as reported in the literature. In this article, we propose a naturalistic software development method to test the advantages of the NPP and to analyze the impact of this paradigm in the learning and practice of software engineering. Using the proposed method will facilitate developers maintaining the traceability of the project and the flexibility to adapt it to agile frameworks, using a subset of effectively tested approaches.

The remainder of this paper is structured as follows: [Section 2] discusses the state of the art on the NPP by reviewing its most representative programming languages and their progress within software development. Next, in [Section 3] we introduce the foundations of our naturalistic method. Then, in [Section 4], we present said method, which is specifically proposed for the analysis (i.e. requirement analysis) and design stages of the software development process, and we also discuss an example of its implementation using a naturalistic programming language and a traditional programming language (Python), comparing the expressiveness of both. Finally, in [Section 5], we present our conclusions and future work.

## 2 State of the Art

Researchers in [Paterson, Hewitt 70], [Chandra, Manna 75], [Hoare 89], and [Felleisen 91] are pioneers in the study of programming language expressiveness. Similarly, other research proposals have approached this issue from different perspectives. Most notably, some researches emphasize on the advantages of using natural language as a programming tool. The majority of these trends are concentrated on works as: Domain-Specific Languages (DSLs) [Mernik et al. 05], Language-Oriented Programming (LOP) [Felleisen et al. 18], Temporal Logic Programming (TLP) [Baudinet 89].

Another naturalistic proposal includes Natural Language Processing (NLP). As a branch of Artificial Intelligence (AI), NLP outstands in the study of human language. Its beginning dates back to the decade of 1960, and since then, it has made remarkable progress [Rajput 20]. NLP relies upon multiple computational techniques for the automated analysis and representation of human language. However, there is great complexity and difficulty in representing human language, starting from the challenge of reaching a consensus in terms of what one understands in a given language [Fu 19]. Finally, NLP allows a computer to process natural human language and translate it into computer-understandable language [Rajput 20].

Common NLP applications include text summaries [Awasthi et al. 21], automated translations [Lample et al. 18], text analysis [Guetterman et al. 18], sentiment analysis [Ni et al. 21], hate speech detection [Parihar et al. 21], clinical decision-making support [Vasilakes et al. 21]), and mental health problem detection [Rajput 20].

Regarding natural language computer programming, most of the proposals are specific to a particular area. For instance, robotics considers End-User Programming (EUP), as reported in [Schlegel et al. 19]. Finally, the literature reports a great number of purpose-specific NLP applications in software development (e.g. [Cabot 10], [Liu, Wu 18], [Mai et al. 18], [Cabot 19] and [Wang et al. 19]).

About general-purpose programming paradigms, Object-Oriented Programming (OOP) is perhaps the most known. OOP uses abstract classes and objects to represent the real world in terms of its characteristics and interrelations. Key concepts in OOP include abstraction, encapsulation, modularity, hierarchy, typification, concurrence, and persistence [Booch 94]. On the other hand, Aspect-Oriented Programming (AOP), an orthogonal paradigm to OOP, allows programmers to represent non-functional software requirements, such as security and concurrency. According to [Kiczales et al. 97], AOP is more suitable for real-domain problems, yet some problems do not fit into either OOP or procedural programming. Most general-purpose programming paradigms (OOP, functional programming, procedural programming) perform a functional decomposition. AOP, however, works differently. This paradigm considers system de-composition units that are non-functional. Every time two software properties are programmed differently but coordinately, they are said to intersect each other [Nusayr 22].

Unfortunately, none of the aforementioned programming paradigms has been enough to solve all programming problems. Even though OOP approaches programming almost like humans think using objects with characteristics, methods, and object interrelations, what programmers write is not always entirely consistent with what the domain expert requires. In other words, information is distorted throughout the coding process [Ramdoo, Huzooree 15].

In [Lopes et al. 03], authors refer to naturalistic programming, or the NPP, by the first time. This paradigm seeks to go beyond AOP by allowing programming languages to incorporate natural language elements, thus making them more expressive, reflexive, and able to generate readily executable code. The interest in natural language elements lies in the fact that natural language is inherently rich enough to express ideas clearly. However, the expressive power of programming languages is often compromised since programmers must write with a limited, high-level grammar sub-set. The more expressive the programming language, the clearer and more understandable the ideas, which in turn ensures software requirements. Additionally, the NPP could make software products more easily maintainable and reduce software rework [Pulido-Prieto, Juárez-Martínez 20].

According to authors in [Pulido-Prieto, Juárez-Martínez 19], current programming paradigms have limited expressive power, highlighting the need to explore natural language descriptions. In this sense, natural language may be ambiguous, yet it is implicitly highly expressive. Moreover, the human cognitive process helps solve ambiguity in natural language during any communicative event. Unfortunately, computers lack this solving capacity. Naturalistic programming languages do not seek to use AI techniques; they do not infer, label, or learn; instead, naturalistic languages are built using a limited grammar subset of any natural language (e.g. English) as in the case in any other programming language.

[Lieberman, Liu 06] stated that the point of programming straightly in natural language without formal programming languages is an old dream in computer science. However, given the complexity of natural language, this dream has been difficult to achieve, even though the literature reports excellent progress. [Yin 10] complements previous thought, saying that although programming in free natural language is still difficult, it is practical, under specific restricted lexical and syntactic rules. Both thoughts are consistent with the fundamentals of the NPP.

The NPP was first supported and simultaneously directly promoted in [Knöll, Mezini 06], when the researchers developed the first general-purpose naturalistic language, named Pegasus. The researchers referred to the concept of naturalistic programming as “writing computer programs with the help of natural language”. Pegasus emerged from the belief that programmers should be able to write programs in the same way as they write a story. Pegasus can generate readily executable code. It is an interface where programmers can write an idea “in natural language.” In the beginning, Pegasus could write in both German and English.

The following example is a naturalistic program code for “Closing the open files” [Knöll, Mezini 06] requirement:

*“If there are open files, close them and inform the user”*

Also, the following example depicts the representation of the same requirement in Java:

```
boolean filesOpen = false;
for (File f:application.GetFiles())
  if (f.isOpen()) { filesOpen = true; f.close(); }
if (filesOpen) user.inform();
```

As can be observed, the naturalistic representation of the code can be considered as an approximation to self-documentation. The code is similar to the sentence defining the requirement, as if it were written in pseudo code. In other words, the gap between domains is almost non-existent. Anyone with programming knowledge can understand

what is depicted in the previous program; however, as requirements become more complex, programs are more difficult to understand and be maintained.

It is not enough to represent problems with objects or aspects, as programming needs go beyond that. Software development should be able to rely on programming languages that are free from expressiveness constraints and can make software products that are consistent with what users need [Kazman 17].

## 2.1 Naturalistic Programming

As a key characteristic, reflection allows programming languages to self-analyze to identify and refer to elements previously mentioned; e.g. “take the key and use it to open the safety box.” Researchers in [Lopes et al. 03] argue that primitive abstractions in programming languages should be drawn from the study of natural languages rather than from computational sciences, mathematics, or ad-hoc metaphors, such as objects.

The Cambridge English Dictionary defines the term naturalistic as something “similar to what exists in nature.” Hence, a naturalistic programming language should be one that is similar to natural language. According to [Pulido-Prieto, Juárez-Martínez 17], naturalistic language must be restricted to a formal representation that computers can process, but it must also have the highest possible level of abstraction, so that any stakeholder can understand it. Similarly, naturalistic programming languages should take into account the following elements in their functioning (definitions taken from The Oxford and The Cambridge English Dictionaries):

- Endophora. Indirect reference defined in the same text. It is divided into two classifications.
  - Anaphora. Indirect reference whose referent has been previously defined in the same text, e.g. “Take the key; use it in the lock.”
  - Cataphora. Indirect reference whose referent is subsequently defined in the same text, e.g. “After taking it, use the key in the lock.”
- Phrase. Word or series of words with syntactical order and function.

In [Pulido-Prieto, Juárez-Martínez 19], a conceptual model for designing general-purpose naturalistic languages is proposed. The authors claim that naturalistic languages should be able to make references not only to objects, functions, or aspects, but also to other code instructions.

The minimum elements considered in the model as characteristics of naturalistic programming languages are the following:

1. Noun, which can be either singular or plural.
2. Adjective, as complement to the noun.
3. Verb, as an action undertaken by the noun.
4. Circumstance, as a determinant that responds to events.
5. Phrase defines instructions with a complexity beyond that of the subject and predicate (e.g. imperative phrases).
6. Anaphoric capability.
7. Cataphoric capability.
8. Types defined explicitly and statically, they must be used in the construction of phrases.

According to the comparative study made in [Hernández-González et al. 21], only three general-purpose naturalistic languages reported in the literature—Pegasus, SN, and Cal-4700—possess the key elements described in the conceptual model in [Pulido-

Prieto, Juárez-Martínez 19], including reflection and the ability to generate directly executable code. The following section discusses these three languages and provides examples of their implementation to highlight their expressive power.

### 2.1.1. Pegasus

Pegasus emerged from the concern of its authors, as programmers are usually forced to transform their programming ideas to a specific programming language when writing programs. Pegasus can easily help write code in the native language of programmers without the need to think in technical terms or abstractions. This in turn makes code easily understandable to those speaking the same language. Moreover, Pegasus can translate between natural languages (i.e. English and German) and from high-level programming languages [Knöll 19]. The purpose of all these characteristics is that programmers express their ideas into code in the same way as they think. The major benefits of Pegasus can be summarized as follows:

- Reduced mental load. Programmers worry less about technical aspects or abstractions and rather focus on solving the problem.
- Self-documented code. There is no need to add explanations in the programs being developed.
- Programs (e.g. search algorithms) do not need to be written anew in other programming languages.
- Flattened learning curve of programming.
- Software can be reused more easily.

The goal of Pegasus is to write programs in a way that is similar to story-telling. The language relies on a limited subset of English grammar to reduce variability. For instance, when needing to print a string, many words may look appropriate, such as display, write, or print, which is why the researchers suggest making questions in case of doubts. The following code introduces the example of a program extract written with Pegasus. The program controls a humanoid robot [Knöll 19]. As can be observed, in a few lines programmers can write the “story” as desired:

*Lara, greet the person in the room. Go to the chair in the middle of the room and sit down. Tell us what you have done today. Lara, please get my glasses from the other room. Thank you!*

As its main disadvantage, Pegasus needs an external database to operate, which makes the program a little complex. This implies that the program needs further configuration, synchronization, and updating. In this sense, portability in Pegasus is compromised if some of its components are not updated.

### 2.1.2. SN (Sicut Naturali)

SN was built as a proof of concept of their conceptual model for naturalistic programming [Pulido-Prieto, Juárez-Martínez 19]. This language is structured in terms of its components: Nouns, these can be either singular or plural; Adjectives, these are noun characteristics and Attributes, these are elements referring to a noun.

SN also relies on the following: Verbs, Circumstances, Derived attributes, Noun phrases, Sentences, Iterators and Naturalistic decisions.

[Fig. 1] introduce an example of code extract from a daily planner program. The program operates using a MySQL relational database. In this case, the noun Person

must be created along with its corresponding entity, which is used to make the noun persistent.

### 2.1.3. Cal-4700

Cal-4700 is a plain-English programming language developed by father and son [Rzeppa, Rzeppa 19b], whose goal is to allow programmers to code at a natural language level in a relatively sloppy manner by using key parts of a sentence. As depicted in [Fig. 2] [Rzeppa, Rzeppa 19a], programs written with Cal-4700 must include a “To run” function, and programmers must define the types and skills as they write the code. This programming language relies on a noodle file for program compiling. The file contains a pre-defined series of types or things readily available for use, as well as type conversion functions, among other functions.

Program in [Fig. 2] shows a code extract from an anagram generator software program. The program relies on a lexicon of the 64,000 most commonly used English vocabulary words. The lexicon comes in the form of a text file; i.e. one word per line. The combination of a given set of letters is a valid word [Rzeppa, Rzeppa 19a].

Most of the listings of programs introduced in this section were tested with their corresponding compilers. However, in the case of Pegasus, we did not find any compiler available for testing. Also, notice that SN and Cal-4700 only offer user Command-Line interfaces (CLI), yet this is a solid start for promoting the NPP in software development and contributing to the paradigm's evolution.

According to the analysis shown in [Hernández-González et al. 21], Cal-4700 is the most robust and stable language, which is why it will be used to program the example of implementation presented in this article.

All software development stages are equally important. Every software project must be properly analyzed, designed, and tested. Any requirement documentation or specification errors may lead to an incorrect interpretation of what the program needs to do. Even though naturalistic phrases are practically transparent and easy to be processed, not all naturalistic software products possess these characteristics. Hence, there is still a need to rely on naturalistic methods that dictate step by step how to address each stage of the naturalistic software development process.

Similarly, such methods must also guide software engineers throughout the identification, specification, and implementation of non-functional software requirements, in the development of large-scale software, and in collaborative work. With such methods, naturalistic software would need a robust architecture in terms of the smallest unit of the programming language. The following section reflects on the elements needed to design a naturalistic software development process, while in [Section 4] we propose said method.

## 3 Considerations in a Software Development Method

In software engineering, quantifiable, disciplined, and systematic processes ensure to a great extent product quality. Any software development process implies 1) translating user needs into software requirements, 2) translating software requirements into software design, 3) implementing code into the design, 4) testing the code, and

sometimes, verifying and validating whether the software meets its operational purpose.

```

noun Person:
  attribute idperson as a Integer Number.
  attribute name as a String.
  attribute phone_number as a String.
main Test:
  myname is "Gerry". an Entity Person with the String as name.
  a Persistent String with "5596823471" as value.
  the phone_number of the Person is it.
  the driver of the Person is "com.mysql.cj.jdbc.Driver".
  the jar of the Person is "mysql-connector-java-8.0.23.jar".
  the user of the Person is "sn".
  the password of the Person is "sn".
  the URL of the Person is "jdbc:mysql://localhost:3306/agenda".
  some Strings.
  add "name" to these.
  add "phone_number" to these.
  insert the Strings into the Person.
  select the Strings from the Person.
  System prints these and newline.

```

*Figure 1: Daily Planner Code with Access to DB*

```

To run:
  Start up.
  Create the dictionary.
  Save the dictionary.
  Loop.
    Write "Enter a scrambled word: " on the console without advancing.
    Read a string from the console. If the escape key is down, break.
    Unscramble the string.
    Repeat.
    Write "Done..." on the console.
    Destroy the dictionary.
  Shut down.

To unscramble a string:
  Trim the string.
  Lowercase the string.
  Sort the string into a sorted string.
  Loop.
    Get an entry from the dictionary's entries.
    If the entry is nil, break.
    If the entry's sorted string is not the sorted string, repeat.
    Write the entry's string on the console.
  Repeat.
  Skip a line on the console.

```

*Figure 2: Code Extract from an Anagram Generator Program*

All these steps may happen either subsequently or simultaneously within an iterative process [ISO/IEC/IEEE 17]. According to the Software Engineering Body of Knowledge [IEEE 14], the intangible and moldable nature of software allows for a wide range of software development lifecycle models to exist; that is, there are multiple possible ways for a software to perform the same set of tasks but using different relationships [Budgen 20]. Software lifecycle models are independent from programming paradigms. Conversely, software development methods are often linked to a particular programming paradigm and provide a systemic approach to software specifications, design, development, and verification [IEEE 14].



Each stage in the software development cycle process (i.e. analysis, design, development, and testing) is approached differently in an effort to address the needs of each programming language and contribute to the learning of software engineering [ACM 20]. According to [IEEE 90], software quality can be defined as the extent to which a system, component, or process meets the specified requirements and user needs. This definition is a reference to quality to many international standards, such as [ISO/IEC/IEEE 17]. From this perspective, the stage of software requirement analysis is of vital importance to a successful software project. Moreover, when transitioning from one development stage to the other, it is important to ensure that information is not lost or distorted, otherwise the software product is most likely to be inconsistent with what the user needs. Since quality is a concern from the beginning, software engineers keep in mind that taking care of the software development process would ensure that their product is of quality. Nowadays, it is no longer enough to master state-of-the-art software development tools and technology. In order to achieve the desired results, engineers must learn skills such as negotiation, persuasion, collaborative work, and multidisciplinary work [Kazman 17]. Such efforts imply that end-users remain at the heart of the software development process, as software engineers seek to get closer to the problem domain.

Most software development proposals are linked to the OOP, which is currently the predominant programming paradigm. To overcome the challenge of developing a naturalistic software development method, it is important to analyze the development cycle stages with respect to the elements that they have in common from a software development perspective, regardless of the programming paradigm being used.

In this work, we propose a naturalistic software development method for the first stages, analysis and design, of the software development cycle. These two stages play a key role within the entire process [ISO/IEC/IEEE 17]:

- Requirement Analysis. The process of studying user needs to arrive at a definition of a system, hardware, or software requirements.
- Design. The process of defining the software architecture, components, modules, interfaces, and data for a software system to satisfy specified requirements.

### 3.1 Considerations for Software Analysis

Software requirements express the needs and imposed limitations of a given software product and contribute to solving a real-world problem. Currently, software analysis focuses more on developing software requirements than managing them. In this sense, the requirement development process comprises four stages: elicitation, analysis, specification, and validation. The tasks performed in each discipline are further discussed below [Wiegers, Beatty 13]:

1. Requirement elicitation. Leads to the discovery of software requirements and comprises tasks such as interviews, document analysis, and prototype analysis, among others.
2. Requirement analysis. Involves reaching a more precise understanding of each requirement and representing sets of requirements in multiple ways.
3. Requirement specification. Implies representing and documenting knowledge collected in the previous steps in a persistent and organized manner.
4. Requirement validation. It confirms that software developers have the correct

set of information to create a software solution that meets the goals of the business.

Considering that the second stage depends directly on the paradigm of choice, the proposed method focuses on this first step, Requirement analysis. Essential analysis tasks are the following:

1. Analyze information gathered from elicitation to understand the user goals, needs, quality expectations, business policies, and solutions, among others.
2. Break down high-level requirements into an appropriate level of detail.
3. Derive functional requirements from additional information.
4. Understand the importance of quality attributes.
5. Negotiate requirement implementation priorities.
6. Identify unnecessary requirements.
7. Model requirements either into text or graphically, e.g. User Histories Use Cases (UCs).
8. Task diagrams.

In the beginning, when representing software requirements analysis, engineers usually preserve some natural language expressions, which are closer to the user domain than the solution domain. In other words, such expressions lie far from technological expressions. Hence, current representations, such as UCs [Bosch 00], [Rzeppa 21] can be considered in a naturalistic approach, as shown in the proposed method. Moreover, the UC technique is widely employed in software development [Zaman et al. 20] and is aligned with the naturalistic programming philosophy, as it shows how natural language descriptions can be used to understand better and describe software requirements. Likewise, Prototyping technique is very useful in the transition from elicitation to requirements analysis [Lima et al. 22] [Ruiz, Hasselman 20] [Budgen 20] [ISO/IEC/IEEE 18], and therefore also considered

### 3.2 Considerations for Software Design

As defined by the 12207 Std [ISO/IEC/IEEE 08], software design comprises two activities that fit between software requirement analysis and software construction:

- Software architecture design (or high-level design). Develops software top-level structure and organization and identifies software components.
- Software detailed design. Specifies each component thoroughly to facilitate its construction.

According to [Budgen 20], the software design process begins with the specification of the software requirements. Next, the needs are analyzed, and a black box model is built (high-level design). A software solution is then proposed (detailed design) and validated with a system prototype. Finally, the solution is implemented. Additionally, the software design process must consider the following basic elements:

**Design principles.** Key notions serving as the grounds for many software development approaches and concepts [IEEE 14]:

- Coupling and cohesion.
- Decomposition and modularization.
- Encapsulation and information hiding.
- Separation of interface and implementation.
- Sufficiency and completeness.

- Separation of concerns.

In addition to software design principles, some other key issues must be addressed when designing software. Some of these issues are quality features that all software products must address, such as performance, security, reliability, and usability, among others. Other issues deal with some aspect of the behavior of the software that is not in the application domain, but which addresses some of the supporting [Bosch 00]. Such issues, which often crosscut the system's functionality, have been referred to as aspects that "tend not to be units of the software's final decomposition, but rather to be properties that affect the performance or semantics of the components in systemic ways" [IEEE 14]. Some of these crosscutting issues are listed as follows:

- Concurrency.
- Control and handling of events.
- Data persistence.
- Component distribution.
- Error and exception handling and fault tolerance.
- Data security.

**Quality attributes.** Multiple attributes contribute positively to quality in software design. There is an interesting distinction among quality attributes discernible at runtime (e.g. performance, security, availability, functionality, usability), those that cannot be discerned at runtime (e.g. modifiability, portability, reusability, testability), and those related to intrinsic qualities of the software architecture (e.g. conceptual integrity, correctness, integrity) [IEEE 14].

All the aforementioned quality principles, aspects, and attributes must be taken into account when designing software, while simultaneously considering the possible constraints (in terms of organization, resources, experience, and software reusability) to the process and defining how much it is known of the problem domain. Additionally, some software development methods consider risk factors and develop a contingency plan.

**Abstraction** is essential to the software design process. It allows designers to focus on the most important characteristics of a problem. In the context of software design, a representation is used to provide a particular abstraction of the system characteristics. In addition to abstraction, other key points about the design process are [Budgen 20]:

- Fitness for purpose.
- Design models as a key product of designing.
- Design decisions should be recorded.
- Design solutions need to be shared.

**Design representations.** [Budgen 20] claims that design representations can be either for the black box (architecture design) or the white box (detailed design). Black box design representations vaguely indicate what is wanted in terms of design, whereas white box representations indicate how those design requirements are wanted. According to [Rzeppa 21], author of Cal-4700, plain English programming is similar to procedural programming in so that both languages were built to understand phrases as naturally and normally as people dictate instructions to others, including computers, through procedural instructions using types and variables. This assumption implies that procedural software representations, used for decades, can be employed in naturalistic software design. Examples of procedural software representations include Jackson

structure charts, data flow charts, state-transition diagrams, and entity relation diagram [Budgen 20]. Data flow chart and entity relation diagram are included in the proposal as part of the transition of Requirement Analysis and Design.

For the time being, state-transition is not considered in the naturalistic paradigm. Nevertheless, naturalistic software modeling proposals must be consistent with naturalistic thinking. Due to some technical limitations inherent to the currently available naturalistic programming languages, some modeling aspects, such as the software architecture and quality attributes lie outside the scope of this research and will be discussed as remarks for future work in the corresponding section.

## 4 A Naturalistic Software Development Method

According to Pressman [Pressman 10], naturalistic software modeling principles promote software models that have a clear purpose and are simple, modifiable, and not perfect. Software engineers should not limit themselves to use only what they know or master; however, they should be careful when deciding to change something in their modeling processes.

[Fig. 3] depicts the workflow of our naturalistic software development method, which focuses on the Requirements Analysis tasks, beginning with the conception of ideas representing human thinking. This notion of ideas was first introduced in [Knöll, Mezini 06] to translate natural language into source code. The idea notation approach is a strength for requirement traceability and validation, yet we will further discuss this matter in the following sections. Also, our naturalistic software development method relies on artifacts to promote naturalistic thinking. The following paragraphs thoroughly explain the steps comprised within our method. Next, for further clarification, we describe an example as proof of concept of how the method can be implemented.

### 4.1 Requirements Analysis

According to the schema in [Fig. 3], software requirement analysis comprises two stages: identifying elements and analysis elaboration.

#### Identification stage

This stage includes the following steps, which can be followed either sequentially or non-sequentially, depending on the information available:

- **I.1. & I.2. Identify atomic ideas, composed ideas, and their connotations.** The term idea in naturalistic programming was conceptualized in Pegasus [Knöll, Mezini 06] to allow computers to interpret ideas in the translation process. Human perceptions are sets of ideas, and thinking implies composing such ideas. Ideas can be atomic (e.g. a noun), complex (associations between atomic and complex ideas), or composite. Composite ideas represent thoughts by combining multiple complex ideas for a brief moment. [Fig. 4] depicts how the idea notation is designed. This idea notation shows a composite idea. One of the connotations of this composite idea is the atomic idea, whereas the other connotation is another composite idea by itself, which is in turn composed of two ideas: a statement connotation and a command connotation. Such connotations are added by the language reader when analyzing the sentence. The authors of Pegasus refer to

sentences with the “???” marker as idea patterns. Hence, the idea notation for the sentence “If the first element of the second row of the matrix is smaller than 3, then write “I can understand you!” would be as follows:

(( (( (be, predicate), ((element, first, (row, second, (matrix))),  
 subject), smallness, comparative), ((three, than), object)  
 ), statement), (( (writing, predicate), (you, subject), (string, object)  
 ), command) ), condition)

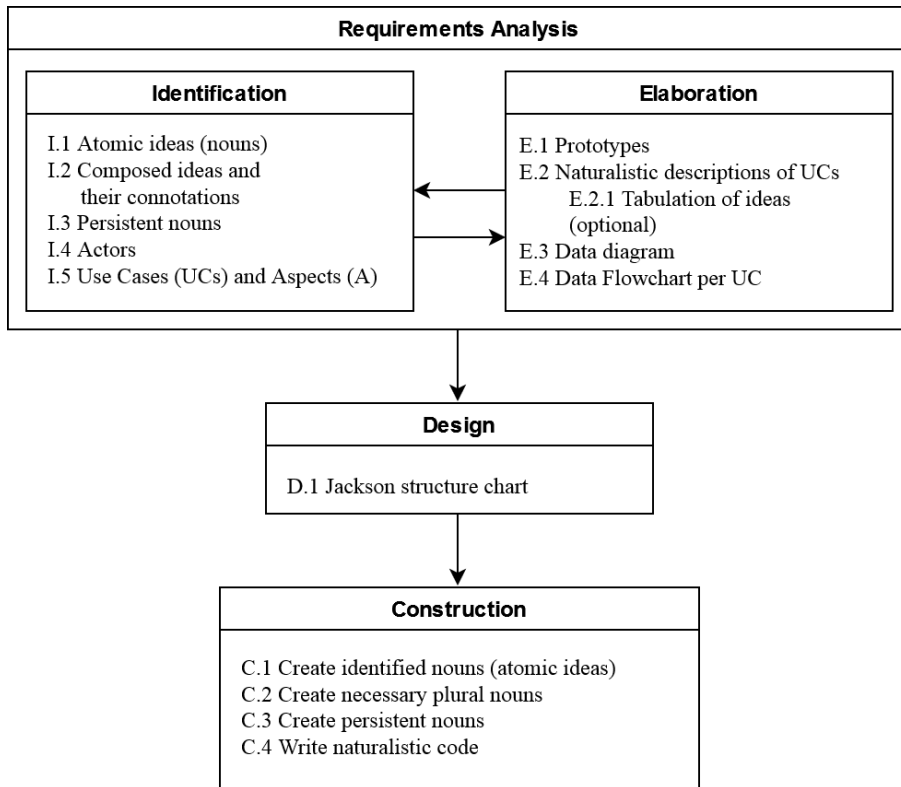


Figure 3: A Naturalistic Software Development Method

In natural language, composite ideas are a part of the description of each UC (flows of normal and alternative paths and exceptions). Even though [Knöll, Mezini 06] and [Mefteh et al. 18] use the same notation specifically for the process of translating requirements into source code, it is an important naturalistic conception to be considered in the naturalistic software design process. Hence, we propose further usage it to validate the traceability of the ideas.

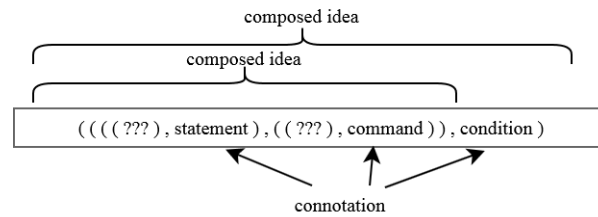


Figure 4: Idea Notation

In naturalistic languages such as SN and Cal-4700, thing is the smallest unit; hence, identified atomic ideas would be automatically be categorized as nouns or things by either of these programs.

- **I.3. & I.4. Identify which of these nouns will be system actors and which will be persistent nouns** (i.e. nouns that will remain after the program is ends). It is important to define which nouns are merely system actors that have no other role, such as that of data or thing within the software program.
- **I.5. Identify UCs and determine which of them will be aspects** (concerns crosscutting the system's functionality). UCs are described following an actor-action-object structure that makes them easier to be understood and facilitates the transition from one software development cycle stage to the other. Aspects are identified with the <<trigger>> stereotype, avoiding confusion among UCs relationships.

#### Elaboration stage

- **E.1. & E.2.** We propose **the naturalistic writing of normal and alternate flows and exceptions for each UC, along with creating the necessary prototypes** to validate the requirements. Naturalistic writing may include anaphoras, one of the main components of the NPP.
- **E.2.1.** Similarly, we propose a **tabular representation of the ideas (optional)** to facilitate idea traceability and validation. During the idea tabulation process, designers identify both the direct object and agent of each idea to define which nouns are actors and which nouns are persistent [see Tab. 2], [Section 4.3]. A direct object refers to the noun or thing that is the recipient of the action of a (transitive) verb, whereas an agent refers to the doer of the action.
- **E.3.** We also developed a **data diagram** to rely on a data persistence scheme. Given the current technological limitations, it is not yet feasible to define specific states for its implementation and thus naturalistic modeling. At the moment, a data diagram may be in the form of either an entity relation diagram or a relational diagram of the corresponding database, in the case of SN. As for Cal-4700, which only allows for data persistence via plain text files, it is possible to include a diagram reflecting the structure and/or interrelation of these files.
- **E.4.** Finally, we suggest building a **Data Flow Diagram (DFD) per UC** to graphically represent the necessary actions, system actors, and data involved. We recommend taking as reference the DFD proposed by [Yourdon 93], which includes easy notation and considers the aforementioned elements. The descriptions of each bubble in a DFD are written in abbreviated natural language to ensure their comprehension and traceability.

## 4.2 Design and Construction

For the **Design phase**, we suggest:

- **D.1. Building a Jackson structure diagram** [Jackson 02] to describe the sequential structure of the software solution with respect to the three classical structuring forms: sequence, selection, and iteration. According to [Budgen 20], this characteristic makes the Jackson diagram unique, since most notations rather focus on the representations of one point of view. Similarly, the Jackson structure diagram relies on natural language phrases aligned to the NPP.

### Construction phase

Finally, for the construction phase, at the moment we merely propose the following guidelines:

- **C.1.** Create the nouns previously identified as atomic ideas.
- **C.2.** Create the necessary plural nouns.
- **C.3.** Create persistent nouns, if needed, according to the capabilities of each programming language.
- **C.4.** Write the remaining naturalistic code.

## 4.3 Implementation of the Naturalistic Software Development Method

In this section, we implement our naturalistic software development method to model a course management system (CMS). In [Baniassad, Clarke 04], [Clarke, Baniassad 05], an aspect-oriented modeling approach was introduced (note: AOP is aligned with OOP). In addition to relying on an Aspect-Oriented model for software analysis and design, we discuss the implementation of one UC in Cal 47-000 and Python, one of the most programming languages used in the software industry<sup>[1]</sup>, to allow readers to appreciate and compare the expressive power of both programming languages. Python is one of the most The requirements of the CMS are the following:

- R1. Students can enroll in individual courses.
- R2. Students can drop individual courses.
- R3. Each student enrolling in an individual course must be registered.
- R4. Each dropped course must be registered.
- R5. Teachers can discharge students from an individual course.
- R6. Each student discharge must be registered.
- R7. When students are discharged from a course, they must be labeled as special.
- R8. Teachers can grade student coursework.
- R9. Each student grade must be registered.

[Fig. 5] depicts the UC diagram for the proposed CMS. We identified two types of actors—students and teachers—and three UCs: Enroll Student, Delete Student Enrollment, and Grade Student. Also, requirements R3, R4, R6, R7, and R9 were found to be cross-sectional with respect to the other requirements, since they all imply registering the resulting activity in a follow-up file. These requirements are encapsulated in an aspect identified as UC Register Follow-Up, and it is included in the rest of the UCs with stereotype <<trigger>>. Even though, in [Jacobson, Ng 04] posit

[1]<https://spectrum.ieee.org/top-programming-languages-2022>

that aspects should be represented using the UC extension, this proposal contradicts Advanced Use Case Modeling (UC 2.0), proposed by same authors [Jacobson et al. 16]. In UC 2.0, UCs are divided into portions, and each portion is handled as an extension. Hence, to avoid confusions, we included the <<trigger>> stereotype.

As the atomic idea, Student is a noun with Student ID Number and Student Name attributes. These three elements are used to build the system, and their representation is introduced in [Fig. 6a]. UC Enroll Student was implemented in Cal-4700. Its corresponding prototype is introduced in [Fig. 6b], whereas [Tab. 1] contains its description. In this case, for clarity and economic reasons, we will work with the normal flow of the UC, even though the programming language allows to write all types of routines, e.g. data validation or conditionals.

The table specifies the basic and alternate flows and the minimum elements required for said UC. Our UC description takes as reference the description template proposed in [Wieggers, Beatty 13]. Notice that the normal flow relies on anaphora—a distinctive element of naturalistic programming and thus natural language—in steps 2, 3, and 5.

[Tab. 2] introduces the representation of the composite ideas; that is, the user requirements included in the UC paths. For each idea, we identified the following:

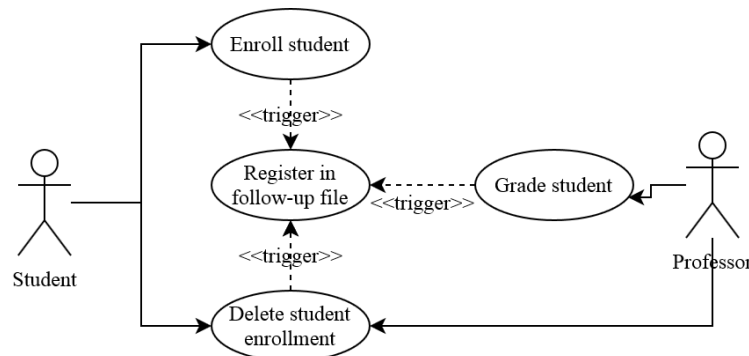


Figure 5: Course Enrollment UC

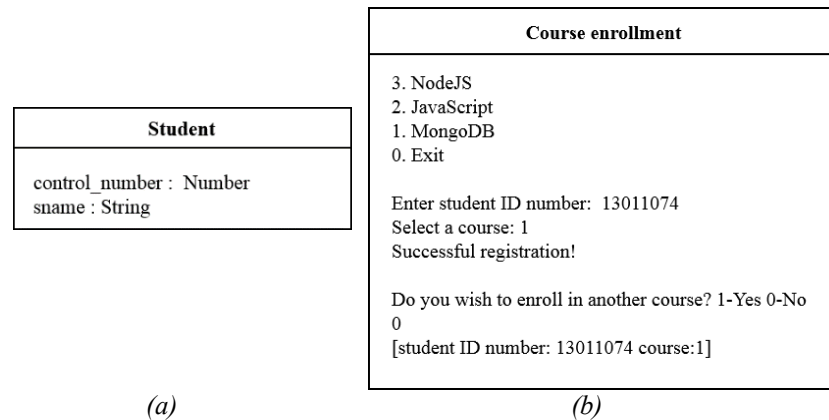


Figure 6: Atomic Idea Student and Enroll Student UC Prototype

1. Whether the connotation is a statement, a command, or a condition.



2. The agent.
3. The anaphoric relationship (if applicable).
4. The direct object.
5. The destination (optional).

<b>Description:</b>	The student selects the desired course(s) from the provided list.
<b>Normal Flow:</b>	<ol style="list-style-type: none"> <li>1) System deploys course list.</li> <li>2) System requests student ID number.</li> <li>3) Student enters it.</li> <li>4) System requests student course ID number.</li> <li>5) Student enters it.</li> <li>6) System asks: Do you wish to enroll in another course?                     <ol style="list-style-type: none"> <li>a) Yes: Return to step 4</li> <li>b) No: End UC</li> </ol> </li> <li>7) System prints out record of enrolled courses</li> </ol>
<b>Launches (trigger relation):</b>	Record follow-up

Table 1: Enroll Student UC Description

Consider, for instance, step 5: “Student enters it,” which refers to a student entering a course number. The idea is a statement whose action implies entering a number; Student is the agent, and the anaphoric reference in the word it is course number, previously mentioned in step 4. Steps 6a and 6b are taken in case the student wants (step 6a) or does not want (6b) to enroll in another individual course. Both steps are identified as commands—execute and exit, respectively, whose agent is the system. In step 6a, the direct object is step 4, thus implying that the student has decided to enroll into another course, and the system must request the corresponding course ID number.

No.	Statement (action)	Command (phrase)	Condition	Agent	Anaphoric Reference	Direct object *plural nouns, only	Destination (optional)
1	Deploy			System		Course	
2	Request			System		student ID number	
3	Capture			Student	student ID number		
4	Request			System		course ID number	
5	Capture			Student	course ID number		
6			Ask	System		Course	
6a		Yes: Execute		System		Step 4	
6b		No: Exit		System			
7	Print			System	Enrolled courses		

Table 2: Tabulation of ideas

Notice that in our proposal, any plural noun being a direct object must be marked with the “\*” symbol for the program to process it as a collection of things. We believe that representing ideas using this structure could allow software designers and developers to trace each program requirement better, since it becomes easier to create test cases for requirement verification and validation.

We used a relational database diagram to model data persistence in the program [Fig. 7]. The diagram includes a follow-up table. The DFD is depicted in [Fig. 8a] and defines the following UC elements: actor, actions, and repositories. Moreover, it considers aspect Register Follow-up (see dotted-line bubble) to ensure its traceability.

[Fig. 8b] introduces the Jackson structure diagram in which the Follow-up aspect appears once more with a <<trigger>> relationship. The noun Student is created at this step since it is a design decision. Finally, the source code in Cal-4700 for the Enroll Student UC is introduced in [Fig 9]. The language only allows data persistence via text files. The reader can also appreciate the use of anaphoric references.

According to the intentions of naturalistic programming, naturalistic code should be able to document itself; that is, anyone who reads it should be able to understand it without greater explanation. The code shows how the language registers Student, whose fields are *student\_id\_number* and *course\_ID\_number* and uses possessives (e.g. “student’s control\_number”).

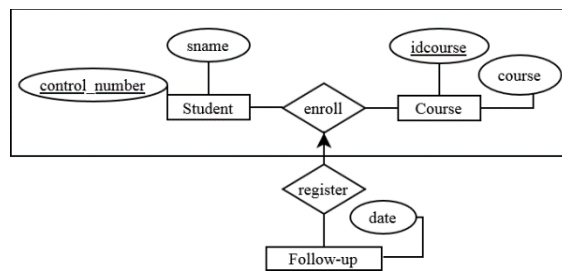


Figure 7: Entity Relation Diagram

Data persistence is achieved using an auxiliary buffer that stores registrations during program execution. Before it is finished, the buffer is stored in the *students.txt* text file (procedure *To save enrollments from a buffer*).

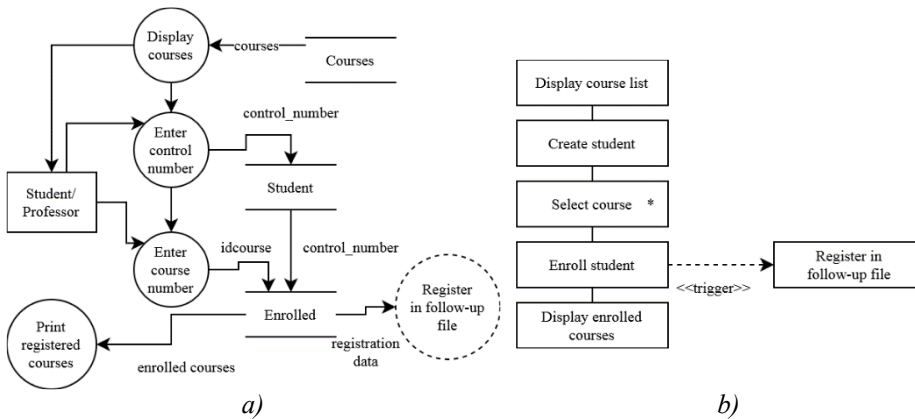


Figure 8: DFD and Structure Diagram for Enroll Student UC

[Fig. 10] presents the corresponding program in Python; as you can see, there are predetermined functions to facilitate programming with text files; therefore, if you have previous knowledge of the language, it can be a shorter program.

```

To enroll:
Loop.
  write "STUDENT ENROLLMENT" to the console.
  write "3. NodeJS" to the console.
  write "2. JavaScript" to the console.
  write "1. MongoDB" to the console.
  skip one line to the console.
  write "Enter control number:" to the console.
  read a string from the console.
  put the string into a student's control_number.
  write "Select a course:" to the console.
  read another string from the console.
  put the other string into the student's course_number.
  enroll the student into a buffer.
  write "¿Do you wish to subscribe to another course? Yes[1] No[0]: " to the console without advancing.
  read a string called condition from the console.
  if the condition is "0", break.
  skip one line to the console.
Repeat.
save enrollments from the buffer.

To enroll a student into a buffer:
Append the student's control_number then " - " then the student's course_number
then the crlf string to the buffer.
write "Student's number " then the student's control_number then " " then
" enroll into course " then the student's course_number to the console.
put "E" into an activity's action. put the system's date/time into the activity's date. [log]
put the student's control_number into the activity's control. put the student's course_number
into the activity's course.
register the activity.

To save enrollments from a buffer:
put "D:\cal-4700 initiative\CMS\cm system\files\students.txt" into a path.
if the path is not in the file system, create the path in the file system.
read the path into a buffer called original.
write the original then the buffer to the path.
If the i/o error is not blank, write the i/o error on the console.

```

Figure 9: Program Code for Enroll Student in Cal-4700

```

import os
def enroll_student():
    print("STUDENT ENROLLMENT")
    print("3.NodeJS")
    print("2.JavaScript")
    print("1.MongoDB")
    while True:
        control_number = input("\nENTER CONTROL NUMBER:")
        if control_number.strip():
            break
        else:
            print("Control number cannot be empty. Please try again.")
    print("SELECT A COURSE")
    while True:
        course_option = input()
        if course_option in ["1", "2", "3"]:
            break
        else:
            print("Invalid choice. Please try again.")
    course_name = course_option
    print(f"Student's number {control_number} enroll into course {course_name}")
    with open("students.txt", "a+") as file:
        if os.stat("students.txt").st_size == 0:
            file.write("Control_number - Course_number\n")
        file.write(f"{control_number} - {course_option}\n")
    return True

```

Figure 10: Program Code for Enroll Student in Python

## Results and analysis

The naturalistic programming technology needs to be improved to compare with the advanced characteristics of the common programming languages. We do not have available metrics data to compare agility or efficiency between programs made with naturalistic languages and those made with non-naturalistic languages. Even though we count on non-naturalistic languages data, the capabilities are different, so we can not construct the same programs to compare. In this example, we calculated the minutes that each programmer needed to build every program, and the reality was that the time was the same: approximately 420 minutes (7 hours).

One of the main goals of the NPP is to write more expressive computer programs, which leads to a more transparent development process. The idea is to move from one software development phase to another without losing information, avoiding ambiguity, and meeting customer requirements. As we can observe in the program made con Cal-4700 [Fig 9], instructions are imperative phrases in natural language, and they look as if the programmer were programming in pseudocode, which means just conveying their thoughts rather than thinking in complex abstractions. Any programmer could understand what the program does and quickly try to write their version following simple rules. Code is self-documented, more readable, and reusable, even for novice programmers.

An experienced programmer could understand what the program in [Fig. 10] does, but he should take more time to document himself if he needs to modify the program or make more complex corrections. In the case of Cal-4700, it is advantageous to have its library brain (*the noodle*) so that the language continues to evolve without recompiling it to have functions that facilitate the work according to the context being worked on.

## 5 Conclusions and Future Work

Once solid programming languages support a programming paradigm, what proceeds is to embark into the planning of software design and analysis activities. Hence, in this research we proposed a naturalistic software development method specific to the analysis and design stages of the software development process.

Naturalistic programming considers natural language elements in an effort to reduce the gap between the problem domain and the solution domain. As its main advantages, naturalistic programming makes software programs more expressive and minimizes the risk of information loss across the software development stages, thus reducing software rework. Overall, naturalistic programming leads to a more transparent and more expressive programming process that ensures user requirements are met, which is a key characteristic of software quality. Even though the syntax of a naturalistic program differs significantly from that of a procedural program, it is still similar to the way humans think and express themselves in natural language; that is, using instructions with a wide range of types and variables involved.

In this research, we proposed a method for naturalistic software development that focuses merely on the analysis and design stages of software development. At the requirement analysis stage, we relied on the UC technique with instructions written in natural language to simplify their understanding. UCs are commonly reported in the

software development literature and vastly used in the software industry. On the other hand, using natural language elements is consistent with the principles of natural programming and facilitates requirement traceability across the multiple software development stages.

Our method is based on the concept of an idea. Ideas reflect user thoughts, which are then transferred as user requirements. Our idea representation technique is another traceability tool of the method that provides a transparent path from the moment a requirement is conceived to its construction, validation, and verification in software analysis and design. The possibility of using Use Cases in its version 2.0, provide flexibility in so that it can adapt to agile frameworks. Additionally, our method employs data flow diagrams and the Jackson structure diagram, which are visual tools that enable programmers to write more detailed requirements when needed, thus increasing the expressive power of programming languages.

As a naturalistic method, our proposal demonstrates that natural language not only benefits from the notion of idea in human thinking, but it can also model a subset of effectively tested approaches, which when implemented in an orderly fashion, lead to an initial software proposal. This is an important advantage to the use and embracement of the NPP.

The full CMS modelling and code using Cal-4700, can be found at <https://github.com/lahernand/np-method>, yet further testing may be needed to confirm the method effectiveness under more complex implementation scenarios. Additionally, further analysis could help determine the consequences of naturalistic programming in software engineering. The current advantages of the NPP suppose that several software development activities could be rendered more agile and less prone to errors, yet further research is needed to confirm this claim. Undoubtedly, regardless of their main goal, software development trends may benefit from the NPP. Consequently, research efforts should concentrate on considering all design elements, for instance, how software could be modularized in such a way that its architecture could be modelled, should be discovered. In addition, one could think on priority tasks in the software development process, perhaps at the requirement analysis stage, to be automated. Meanwhile, authors are working on the method documentation according to international standards. Each challenge must be overcome one step at a time.

### **Acknowledgements**

This work was supported by National Technological Institute of Mexico and University of Veracruz.

### **References**

- [ACM 20] ACM.: “Computing Curricula Report”; (2020), URL <https://www.acm.org/binaries/-content/assets/education/curricula-recommendations/cc20.pdf>.
- [Awasthi et al. 21] Awasthi, I., Gupta, K., Bhogal, P. S., Anand, S. S., Soni, P. K.: “Natural Language Processing (NLP) based Text Summarization - A Survey”; In 2021 6th International Conference on Inventive Computation Technologies (ICICT), (2021), pages 1310–1317. doi: 10.1109/iciict50816.2021.9358703.

- [Baniassad, Clarke 04] Baniassad, E., Clarke, S.: “Theme: An Approach for Aspect-Oriented Analysis and Design”; In Proceedings. 26th International Conference on Software Engineering, (2004), pages 158–167. doi: 10.1109/ICSE.2004.1317438. ISSN: 0270-5257.
- [Baudinet 89] Baudinet, M.: “Temporal Logic Programming is Complete and Expressive”; In Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL ’89, (1989), pages 267–280, New York, NY, USA. Association for Computing Machinery. ISBN 9780897912945. doi: 10.1145/75277.75301.
- [Booch 94] Booch, G.: “Object-Oriented Analysis and Design with Applications”; Addison-Wesley, second edition. ISBN 0-8053-5340-2. (1994)
- [Bosch 00] Bosch, J.: “Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach”; ACM Press, (2000)
- [Budgen 20] Budgen, D.: “Software Design: Creating Solutions for Ill-Structured Problems”; Chapman and Hall/CRC, New York, 3 edition, ISBN 9781315300078, (2020), doi: 10.1201/b21883.
- [Cabot 10] Cabot, J.: “From UML and OCL to Natural Language”; (2010), URL <https://modeling-languages.com/umlocl-natural-language-using-sbvr-pivot/>
- [Cabot 19] Cabot, J.: “Slack as a UML Modeling Tool with Natural Language”; (2019), URL <https://modeling-languages.com/slack-uml-modeling/>
- [Chandra, Manna 75] Chandra, A., Manna, Z.: “On the Power of Programming Features”; Computer Languages, (1975), 1 (3): 219 – 232. ISSN 0096-0551. doi: [https://doi.org/10.1016/0096-0551\(75\)90032-6](https://doi.org/10.1016/0096-0551(75)90032-6).
- [Clarke, Baniassad 05] Clarke, S., Baniassad, E.: “Aspect-Oriented Analysis and Design, The Theme Approach”; Object Technology. Addison-Wesley, first edition ISBN 0-321-24674-8. (2005)
- [Daun et al. 23] Daun, M., Grubb, A.M., Stenkova, V. et al. “A systematic literature review of requirements engineering education”; Requirements Eng 28, (2023), pages 145–175. <https://doi.org/10.1007/s00766-022-00381-9>
- [Felleisen 91] Felleisen, M.: “On the Expressive Power of Programming Languages”; Science of Computer Programming, (1991), 17 (1): 35–75. ISSN 0167-6423. doi: 10.1016/0167-6423(91)90036-W.
- [Felleisen et al. 18] Felleisen, M., Findler, R. B., Flatt M., Krishnamurthi, S., Barzilay, E., McCarthy, J., Tobin-Hochstadt, S.: “A Programmable Programming Language”; Communications of the ACM, (2018), 61 (3): 62–71. ISSN 0001-0782. doi: 10.1145/3127323.
- [Fu 19] Fu, Z.: “An Introduction of Deep Learning Based Word Representation Applied to Natural Language Processing”; In 19 International Conference on Machine Learning, Big Data and Business Intelligence (MLBDBI), (2019), 92–104. doi: 10.1109/MLBDBI48998.19.00025.
- [Guetterman et al. 18] Guetterman, T. C., Chang, T., DeJonckheere, M., Basu, T., Scruggs, E., Vydiswaran, V. G. V.: “Augmenting Qualitative Text Analysis with Natural Language Processing: Methodological Study”; Journal of Medical Internet Research, (2018), 20 (6): 231. doi: 10.2196/jmir.9702.
- [Hernández-González et al. 21] Hernández-González, L. A., Juárez-Martínez, U., Alducin-Francisco, L. M.: “Evolution of Naturalistic Programming: A Need”; In J. Mejía, M. Muñoz, I. Rocha, and Y. Quiñonez, editors, New Perspectives in Software Engineering, Advances in Intelligent Systems and Computing, (2021), pages 185–198, Cham. Springer International Publishing. ISBN 9783030633295. doi: 10.1007/978-3-030-63329-5\_13.

- [Hoare 89] Hoare, C.: “The Varieties of Programming Languages”; In International Joint Conference on Theory and Practice of Software Development, Lecture Notes in Computer Science, (1989), pages 1–18. Springer. Berlin.
- [IEEE 14] C. S. IEEE “Software Engineering Body of Knowledge”; IEEE, v3 edition. ISBN 978-0-7695-5166-1. (2014).
- [IEEE 90] IEEE Std 610.12-1990 “Standard Glossary of Software Engineering Terminology”; (1990), doi: 10.1109/IEEESTD.1990.101064.
- [ISO/IEC 16] ISO/IEC TR 29110-1:2016 – “Systems and Software Engineering-Lifecycle profiles for Very Small Entities (VSEs)-Part 1: Overview”; (2016)
- [ISO/IEC/IEEE 08] ISO/IEC/IEEE Std 12207- “Standard for Systems and Software Engineering - Software Life cycle Processes”; (2008)
- [ISO/IEC/IEEE 17] ISO/IEC/IEEE Std 24765- “International Standard - Systems and Software Engineering–Vocabulary”; (2017)
- [ISO/IEC/IEEE 18] ISO/IEC/IEEE Std 29148 “Systems and Software Engineering - Life Cycle Processes - Requirements Engineering”; (2018).
- [Jackson 02] Jackson, M.: “Jackson Development Methods”; American Cancer Society. ISBN 9780471028956. (2002), doi: <https://doi.org/10.1002/0471028959.sof173>.
- [Jacobson et al. 12] Jacobson, I., Ng, P. W., McMahon, P. E., Spence, I., Lidman, S.: “The Essence of Software Engineering: the SEMAT Kernel”; Communications of the ACM, (2012), 55 (12): 42. ISSN 00010782. doi: 10.1145/2380656.2380670.
- [Jacobson et al. 16] Jacobson, I., Spence, I., Kerr, B.: “Use-case 2.0”; Communications of the ACM, (2016), 59 (5): 61–69. ISSN 0001-0782. doi: 10.1145/2890778.
- [Jacobson, Ng 04] Jacobson, I., Ng, P. W.: “Aspect-Oriented Software Development with Use Cases”; Addison-Wesley Object Technology Series. Addison-Wesley Professional. ISBN 0321268881. (2004)
- [Kazman 17] Kazman, R.: “Software Engineering”; Computer, (2017), 50 (7): 10–11. ISSN 0018-9162. doi: 10.1109/MC.17.184.
- [Kiczales et al. 97] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-m., Irwin, J.: “Aspect-Oriented Programming”; In Proceedings of the European Conference on Object-Oriented Programming (ECOOP), (1997), pages 220–242. Springer-Verlag.
- [Knöll 19] Knöll, R.: “Pegasus Natural Programming”; TU Darmstadt. (2019), <http://www.pegasus-project.org/en/Welcome.html>.
- [Knöll et al. 11] Knöll, R., Gasiunas, R., Mezini, M.: “Naturalistic Types”; In Proceedings of the 10th SIGPLAN Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! (2011), pages 33–48, New York, NY, USA. ACM. ISBN 9781450309417. doi: 10.1145/2048237.2048243. event-place: Portland, Oregon, USA.
- [Knöll, Mezini 06] Knöll, R., Mezini, M.: “Pegasus: First Steps Toward a Naturalistic Programming Language”; In Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications, OOPSLA '06, (2006), pages 542–559, New York, NY, USA. ACM. ISBN 9781595934918. doi: 10.1145/1176617.1176628. event-place: Portland, Oregon, USA.
- [Lample et al. 18] Lample, G., Ott, M., Conneau, A., Denoyer, L., Ranzato, M.: “Phrase-Based & Neural Unsupervised Machine Translation”; (2018).

- [Lieberman, Liu 06] Lieberman, H., Liu, H.: “Feasibility Studies for Programming in Natural Language”; *HumanComputer Interaction Series*. (2006), pages 45–473, Dordrecht: Springer Netherlands
- [Lima et al. 22] Lima, R., Filippetto, A. S., Heckler, W., Barbosa, J. L., Leithardt, V. R.: “Towards Ubiquitous Requirements Engineering Through Recommendations Based on Context Histories”; *PeerJ Computer Science*, (2022), 8: e794. doi: 10.7717/peerj-cs.794.
- [Liu, Wu 18] Liu, X., Wu, D.: “From Natural Language to Programming Language”; chapter 4, (2018), pages 110–130. IGI Global. ISBN 9781522559696. doi: 10.4018/978-1-5225-5969-6.ch004.
- [Lopes et al. 03] Lopes, C.V., Dourish, P., Lorenz, D. H., Lieberherr, K.: “Beyond AOP: Toward Naturalistic Programming”; *SIGPLAN Notices*, (2003), 38 (12): 34–43. ISSN 0362-1340. doi: 10.1145/966051.966058.
- [Mai et al. 18] Mai, P. X., Pastore, F., Goknil, A., Briand, L. C.: “A Natural Language Programming Approach for Requirements-Based Security Testing”; In *Proc. IEEE 29th Int. Symp. Software Reliability Engineering (ISSRE)*, (2018), pages 58–69. doi: 10.1109/ISSRE.2018.00017.
- [Mefteh et al. 18] Mefteh, M., Bouassida, N., Ben-Abdallah, H.: “Towards Naturalistic Programming: Mapping Language-Independent Requirements to Constrained Language Specifications”; *Science of Computer Programming*, (2018), 166: 89–119. ISSN 0167-6423. doi: 10.1016/j.scico.2018.05.006.
- [Mernik et al. 05] Mernik, M., Heering, J., Sloane, A. M.: “When and How to Develop Domain-Specific Languages”; *ACM Computing Surveys*, (2005), 37 (4): 316–344. ISSN 0360-0300. doi: 10.1145/1118890.1118892.
- [Ni et al. 21] Ni, Y., Zhang T., Xu, L., Han, P.: “Research on the evolution path of sentiment analysis technology based on bibliometrics”; (2021), pages 150–157. IEEE. ISBN 978-1-6654-2491-2. doi: 10.1109/CAIBDA53561.2021.00039.
- [Nusayr 22] Nusayr, A.: “Extending the Aspect Oriented Programming Joinpoint Model for Memory and Type Safety”; *International Journal of Computer and Information Engineering*, (2022) 16 (9): 390–393. ISSN 1307-6892.
- [Parihar et al. 21] Parihar, A. S., Thapa, S., Mishra, S.: “Hate Speech Detection Using Natural Language Processing: Applications and Challenges”; In *2021 5th International Conference on Trends in Electronics and Informatics (ICOEI)*, (2021), pages 1302–1308. doi: 10.1109/ICOEI51242.2021.9452882.
- [Paterson, Hewitt 70] Paterson, M. S., Hewitt, C. E.: “Comparative Schematology”; In *Comparative Schematology*, (1970), pages 119–127. Association for Computing Machinery, New York, NY, USA. ISBN 9781450348126.
- [Pressman 10] Pressman, R. S.: “Software Engineering, A Practitioner’s Approach”; McGraw-Hill, seventh edition. ISBN 9780073375977. (2010)
- [Pulido-Prieto, Juárez-Martínez 17] Pulido-Prieto, O., Juárez-Martínez, U.: “A Survey of Naturalistic Programming Technologies”; *ACM Comput. Surv.*, (2017), 50 (5): 70:1–70:35. ISSN 0360-0300. doi: 10.1145/3109481.
- [Pulido-Prieto, Juárez-Martínez 19] Pulido-Prieto, O., Juárez-Martínez, U.: “A Model for Naturalistic Programming with Implementation”; *Applied Sciences*, (2019), 9 (18): 3936. doi: 10.3390/app9183936.



- [Pulido-Prieto, Juárez-Martínez 20] Pulido-Prieto, O., Juárez-Martínez, U.: “Naturalistic Programming: Model and Implementation”; *IEEE Latin America Transactions*, (2020), 18 (07): 1230–1237. ISSN 1548-0992. doi: 10.1109/TLA.20.9099764.
- [Rajput 20] Rajput, A.: “Chapter 3 - Natural Language Processing, Sentiment Analysis, and Clinical Analytics”; *Next Gen Tech Driven Personalized Med&Smart Healthcare*, (2020), pages 79–97. Academic Press. ISBN 9780128190432. doi: 10.1016/B978-0-12-819043-2.00003-4.
- [Ramdo, Huzooree 15] Ramdo, V., Huzooree, G.: “Strategies to Reduce Rework in Software Development on an Organisation in Mauritius”; *International Journal of Software Engineering and its Applications*, (2015), 6. doi: 10.5121/ijsea.2015.6502.
- [Ruiz, Hasselman 20] Ruiz, M., Hasselman, B.: “Can We Design Software as We Talk?”; In *Enterprise, Business-Process and Information Systems Modeling*, (2020), pages 327–334. doi: 10.1007/978-3-030-49418-6\_22.
- [Rzeppa 21] Rzeppa, G.: “The Osmosian Order of Plain English Programmers”; *Personal communication (Personal communication)*, (2021, November).
- [Rzeppa, Rzeppa 19a] Rzeppa, G., Rzeppa, D.: “The Osmosian Order of Plain English Programmers Blog”; (2019), URL <https://osmosianplainenglishprogramming.blog>
- [Rzeppa, Rzeppa 19b] Rzeppa, G., Rzeppa, D.: “The Osmosian Order of Plain English Programmers”; (2019), URL <http://www.osmosian.com/>
- [Schlegel et al. 19] Schlegel, V., Lang, B., Handschuh, S., Freitas, A.: “Vajra: step-by-step programming with natural language”; In *Proceedings of the 24th International Conference on Intelligent User Interfaces, IUI '19*, (2019), pages 30–39, Marina del Ray, California. Association for Computing Machinery. ISBN 9781450362726. doi: 10.1145/3301275.3302267.
- [Vasilakes et al. 21] Vasilakes, J., Zhou, S., Zhang, R.: “Chapter 6 Natural language processing”; In *Machine Learning in Cardiovascular Medicine*, (2021), pages 123–148. doi: 10.1016/b978-0-12-820273-9.00006-3.
- [Wang et al. 19] Wang, C., Pastore, F., Goknil, A., Briand, L. C.: “Automatic Generation of System Test Cases from Use Case Specifications: an NLP-based Approach”; (2019), arXiv: 1907.08490.
- [Wieggers, Beatty 13] Wieggers, K., Beatty, J.: “Software Requirements”; Microsoft Press, third edition, ISBN 978-0-7356-7966-5, (2013)
- [Yin 10] Yin, P.: “Natural Language Programming Based on Knowledge”; In: *2010 International Conference on Artificial Intelligence and Computational Intelligence*, (2010), vol. 2, pp. 69-73
- [Yourdon 93] Yourdon, E.: “Modern Structured Analysis”; Prentice-Hall, First edition, ISBN 0135986249, (1988)
- [Zaman et al. 20] Zaman, Q. u., Nadeem, A., Sindhu, M. A.: “Formalizing the use case model: A model-based approach”; *PLOS ONE*, (2020), 15 (4): e0231534, doi: 10.1371/journal.pone.0231534.