Combining Object-Oriented Paradigm and Controlled Natural Language for Requirements Specification

Yegor Bugayenko
yegor256@gmail.com
Huawei
Moscow, Russia

Abstract

Natural language is the dominant form of writing software requirements. Its essential ambiguity causes inconsistency of requirements, which leads to scope creep. On the other hand, formal requirements specification notations such as Z, Petri Nets, SysML, and others are difficult to understand by non-technical project stakeholders. They often become a barrier between developers and requirements providers. The article presents a controlled natural language that looks like English but is a strongly typed object-oriented language compiled to UML/XMI. Thus, it is easily understood, at the same time, by non-technical people, programmers, and computers. Moreover, it is formally verifiable and testable. It was designed, developed, and tested in three commercial software projects in order to validate the assumption that object-oriented design can be applied to requirements engineering at the level of specifications writing. The article outlines key features of the language and summarizes the experience obtained during its practical application.

CCS Concepts: • **Software and its engineering → Requirements analysis**.

Keywords: Requirements, Natural Language Processing

ACM Reference Format:
Yegor Bugayenko. 2021. Combining Object-Oriented Paradigm and Controlled Natural Language for Requirements Specification. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Beyond Code: No Code (BCNC '21), October 17, 2021, Chicago, IL, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3486949.3486963

1 Introduction

Software Requirements Specification (SRS) should be correct, unambiguous, complete, consistent, ranked for importance and/or stability, verifiable, modifiable, and traceable [9]. Many times, it was demonstrated that an SRS document that lacks some of these properties has a high likelihood of becoming a root cause of scope and schedule problems in a project [5, 11, 17].

Natural Language (NL) is the main presentation mean in industrial requirements documents [10, 15], mostly due to its flexibility. Very often, contributors to requirements specifications are not experts in requirements engineering (RE). Impreciseness and flexibility of NL, being its advantage in informal human communications, turns into a serious drawback in RE. Problems that appear in NL requirement documents include ambiguity, vagueness, complexity, duplication, wordiness, and untestability [11].

To overcome problems associated with NL, some experts advocate the use of other notations for the specification of requirements, such as Z [18], SysML, UML [2], Petri Nets, and others. However, use of any of these non-textual notations often requires complex translation of the source requirements, which can introduce further errors. Such translation of requirements can serve to create an extra "language barrier" between developers and stakeholders. There is also a training overhead associated with the introduction of many notations [11].

A desired solution to the outlined problem would be a Controlled Natural Language (CNL), which would look like NL, but be precise, unambiguous, and consistent. A few CNLs were introduced and developed over the last decades. Closest to our work are Attempt to Controlled English (ACE) [8], Simplified Technical English (STE) [1], Easy Approach to Requirements Syntax (EARS) [11], Boeing's Computer Processable Language [6], and Schwitter's Processable ENGlish [14]. However, there are a few critical drawbacks that exist, to some degree, in all of these CNLs.

First, none can be mapped to objects hierarchy or a UML diagram. They are focused more on linguistic instead of object-oriented semantic. Some CNLs, like ACE, are based on first-order logic and can be mapped to Lisp or other functional languages. However, as of today, object-oriented systems dominate, especially in enterprise applications domain.

Second, they all require sequential flow of language constructs, while in modern Agile software projects wiki pages is one of the most popular storage for requirements documentation, instead of files or databases. Wiki pages, by definition, are not sorted and may contain pieces of requirements documentation that should be combined into a complete document in a random order.

Third, most do not allow uncertainty, while requirements development and requirements management processes need to be able to mark requirements as uncertain (also known as “TBD”). Requirements engineers need to have an ability to add a requirement into scope at one iteration and refine it later in a few iterations. This process, called “progressive elaboration” of requirements [3], must be supported by a CNL.

To overcome these drawbacks, a new language was introduced, implemented, and tested. Sec. 2 of the article presents the language and gives a simple example of its use, specifying requirements of a sample software system. Sec. 3 briefly outlines lexical rules of the language. Sec. 4 describes the syntax using extended Backus-Naur form and gives a few practical examples. Sec. 5 outlines a semantic layer of the compiler that is intended to detect vast majority of inconsistencies in a requirements document. Sec. 6 gives an example of to-UML mapping implemented by the compiler. Sec. 7 presents a numeric metric that measures ambiguity in a requirements specification and could be used as a guidance for system analysts. Sec. 8 presents empirical results obtained from three commercial Java web projects over the last few years. Sec. 9 presents conclusions and observations made by the author.

2 Controlled Natural Language

Requs[1] is an object-oriented CNL that resembles English and, because of that, can be read and understood by compilers, system analysts, end-users, and business people. Fig. 1 presents an example of a short requirements definition document that describes a calculator that asks its user for two floating point numbers and returns the result of their division.

The example has some uncertainty expressed with plain English text sentences in double quotes. This is an important feature that distinguishes the designed language from other CNLs. Every class and method may be specified either formally or by so-called “informal” text in double quotes. In NL requirements specifications, such uncertainties are usually marked with TBD (“to be determined”) placeholders [12]. Java equivalent of them are `TODO` comments. Informal texts in double quotes are also used to make references from functional requirements to customer needs, interface specifications, supplementary documents, etc.

[1] https://www.requs.org

```
SuD includes: user as User.
Fraction is a "math calculator".
Fraction needs:
numerator as Float, and
denominator as Float.
UC1 where SuD divides two numbers:
1. The user creates Fraction (a fraction);
2. The fraction "calculates" Float
  (a quotient);
3. The user "receives results" using
  the quotient.
UC1/2 when "division by zero":
1. The user "fails" using "can't
  divide by zero".
```

Figure 1. Software Requirements Specification of an example math calculator

`SuD` (System under Development) is a top level class that encapsulates all other objects and may not be “created” (similar to a singleton in OOP). The `Float` is a built-in class, as well as the `Text`, the `Integer`, and a few others. Classes have methods, which are defined as use cases with a single main flow and a few alternative flows [7].

In the example, the use case `UC1` is a method of the class `SuD` and its name is “divides two numbers.” Every flow step in a use case can either *a*) call other method, or *b*) create (instantiate) an object. In the example, the step 1 of the `UC1` is creating a new object `fraction` of the class `Fraction`, while the step 2 is calling a method “calculates” of the `fraction` in order to get back an object named `quotient` of the class `Float`.

Every clause ends with a period. Order of clauses is not important and this is yet another important feature of the language. In the example clause at the tenth line, defining an alternative flow may appear before the use case itself. The compiler will understand them correctly.

Fig. 2 contains a close equivalent program in Java.

Requs, in its current version, doesn’t have any lexical or syntax constructs for non-functional requirements (NFRs) specification. This is a subject for future research and implementation. The biggest expected challenge in this area is inventing (or borrowing) a method of quantification of NFRs.

3 Lexical Analysis

There are a few main lexical terms in Requs, which can be defined by regular expressions:

```
<class> ::= /[A-Z][A-Za-z]+/
<word> ::= /[a-z]+/
<use-case-ID> ::= /UC[0-9\\.]+/
<flow-ID> ::= /[0-9]+/
<informal> ::= /"([^"]|\\")+"/
```

```
class User {
  Fraction createsFraction() {}
  void receivesResult(Float quotient) {}
  void fails(String message) {}
}
class Fraction {
  private final float numerator;
  private final float denumerator;
  Fraction(float num, float denum) {
    this.numerator = num;
    this.denumerator = denum;
  }
  Float quotient() throws DivisionByZero {
  }
}
class SuD {
  private final User user;
  void dividesTwoNumbers() {
    Fraction fraction =
      this.user.createsFraction();
    try {
      this.user.receivesResult(
        fraction.quotient()
      );
    } catch (DivisionByZero ex) {
      this.user.fails(
        "denominator can't be zero"
      );
    }
  }
}
```

Figure 2. Java equivalent of Requs requirements document from the Fig. 1

In Fig. 1, the entities `SuD`, `Fraction`, `Float`, and `User` are classes. They start with capital letters and contain only letters (both lowercase and capital). The `user`, the `fraction`, and the `quotient` are the variables bound to instances of classes.

Besides, there are a few reserved words, which may have special meaning in certain places of the text, including `the`, `a`, `includes`, `requires`, etc. A full list of them is defined below in Sec. 4.

4 Syntax Analysis

A program consists of clauses (similar to statements in other languages). Every clause ends with a period, like English sentences. There are four types of clauses: class declaration, class construction, method declaration, and alternative flow declaration.

```
<SRS> ::= <clause>+
<clause> ::= <class-declaration>
  | <class-construction>
  | <method-declaration>
  | <alternative-flow-declaration>
```

4.1 Class Declaration

Class declaration is a clause that declares a class as a subclass of another class or as a standalone class. The central part of the clause is an `is a` term:

```
<class-declaration> ::= <class>
  <is-a> <parent-class>
<parent-class> ::= <informal> | <class>
<is-a> ::= `is' ( `a' | `an' )
```

The second line in Fig. 1 declares the class `Fraction` without any super-class but with an informal description `"math calculator"`.

4.2 Class Construction

Class construction is a clause that defines arguments of a class constructor, if necessary. By default, a constructor doesn't need any arguments:

```
<class-construction> ::= <class> <includes>
  `:' <slots>
<includes> ::= `includes' | `needs'
<slots> ::= <slot> ( `,' <slot> )*
<slot> ::= <variable> ( `as' <class> )?
<variable> ::= <word> `-s'?
```

Fig. 1 defines the arguments of the constructor only for the `Fraction` class. The class `User` don't have any arguments. The class `SuD` is a pre-defined class without arguments. The class `Float` is a pre-defined class with one argument.

Class without arguments in constructor normally means that it represents an entity outside of system scope, like the `User`.

Cardinality of associations between objects is configured by the `-s` prefix at the end of variable name. There are only two possible relationships types supported: one-to-one and one-to-many.

4.3 Method Declaration

Method declaration is a clause that defines a method for a class:

```
<method-declaration> ::= <use-case-ID> `where'
<class> <signature> `:' <flows>
<signature> ::= ( <word>+ | <informal> )
  <subject>? <using>?
<subject> ::= <class> <binding>?
  | `the' <variable>
<binding> ::= `(' `a' <variable> `)'
<using> ::= `using' <subject>
  ( `and' <subject> )*
<flows> ::= <flow> ( `;' <flow> )+
<flow> ::= <flow-ID> `.'
  (
    `The' <variable> <signature>
    |
    `Fail' `since' <informal>
  )
```

For traceability, all methods have unique names across the entire SRS starting with the UC (use case). In other words, methods are use cases owned by classes.

4.4 Alternative Flow Declaration

Alternative declaration is a clause that defines an alternative flow of a method:

```
<alternative-flow-declaration> ::=
  <use-case-ID> `/' <flow-ID>
  `when' <informal> `:' <flows>
```

Alternative flow explains exceptional situations in one of the flows of a method it refers to.

5 Semantic Analysis

In the language, as in any pure object-oriented language, everything is an object. Objects are grouped into classes. They have only methods and no other public properties and may be bound to variables. It is a strongly typed language.

The main goal of semantic analysis in the language is to make sure that: *a*) method calls match method declarations, *b*) object creating have necessary constructor arguments, *c*) variable bindings precede their usage, *d*) methods return objects of required classes, and *e*) failures are handled by alternative flows. If any of these checks fail, the entire document is rejected.

5.1 Method Calls Match Declarations

Every method call in every use case should either *a*) be informal (the 8th, 9th, and 11th lines in the example at Fig. 1), *b*) formally refer to the method declared in a class, or *c*) construct an instance of a class using a built-in method `creates` (the 7th line in the example).

By such a strict linking of use case flows and class methods, a consistency of the entire document is achieved. It is no longer possible to refer to some functionality in a use case, which doesn't exist. A rare mistake during the requirements development phase — but very popular during requirements management and maintenance — when one part of the document is updated aside from another part.

5.2 Object Creating Have Necessary Constructor Arguments

Every time an object is instantiated using the `creates` method, the compiler checks that all required parameters of required classes are supplied. Again, it is a very important validation that prevents inconsistency during requirements maintenance and refinement. In the example, class `Fraction` requires two parameters to be instantiated. Any attempt to create a fraction with just one parameter or two parameters of classes other than the `Float` would be rejected by the compiler.

5.3 Variable Bindings Precede Their Usage

A variable is declared and bound to its value by means of an article `a`. The seventh line of the example declares a variable `fraction`, which is later used with article `the` (the eighth line). If, by mistake, variable declaration is removed from the first step of the use case or the entire step is deleted, compilation will fail. Consistency of the document is ensured by this constraint.

5.4 Methods Return Objects of Required Classes

Similar to the check of parameter classes, every returned object is verified for compliance with the class expected. The eighth line of the example expects method "calculates" of the class `Fraction` to return an object of class `Float`. The method is still informal and is not defined in the requirements document, but when a requirements expert decides to make it formal and specify its details in a new use case, she has to make sure it returns an instance of the class `Float`. Otherwise, the compiler would complain with an error.

5.5 Failures Are Handled by Alternative Flows

Every method in the object-oriented way may throw an exception, using a build-in method `Fail since`. The method has one parameter of the class `Text`, which explains the reason. In the example in Fig. 1, method "calculates" of the class `Fraction` may throw an exception and it is handled by an alternative flow. In the language, all exceptions are checked, which means that they should be handled explicitly, by means of alternative flows of use cases.

6 UML and XMI

When semantic analysis is done the specification looks like an object of the class `SuD` that encapsulates other objects and methods. This object hierarchy is convertible to UML diagrams [2]. Specification in Fig. 1 would be converted to the following UML diagrams (to save space, the list is limited to the three most interesting diagrams):

- Class diagram, in Fig. 3
- Object diagram, in Fig. 4
- Sequence diagram of "divides two numbers" method, in Fig. 5

Every UML diagram is a formal interoperable document in XMI [13] that can be rendered in web, in LaTeX, in PDF, or translated to Java or any other object-oriented language. Fig. 6 shows an example XMI for the class diagram.

7 Ambiguity

With this approach, ambiguity of requirements becomes a formally measurable metric. It is calculated as a division of informally defined methods to the total number of methods:

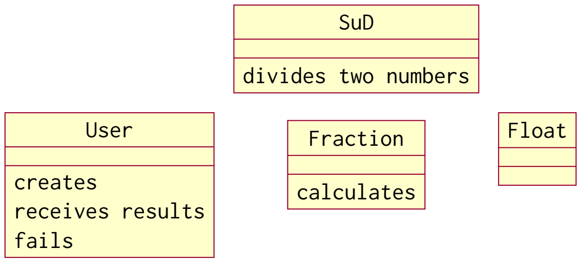

Figure 3. UML class diagram for the sample project

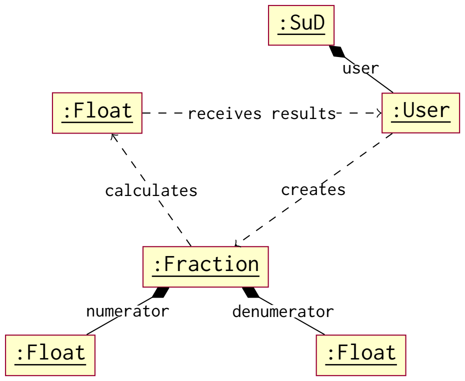

Figure 4. UML object diagram for the sample project

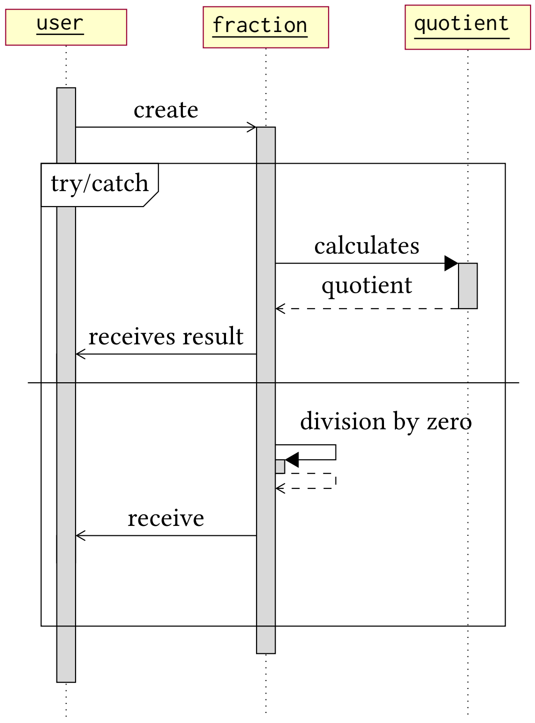

Figure 5. UML sequence diagram for "divides two numbers" method of class SuD of the sample project

```
<uml:Model xmi:version="2.1"
  xmlns:uml
    ="http://www.eclipse.org/uml2/2.0.0/UML"
  xmlns:xmi
    ="http://schema.omg.org/spec/XMI/2.1">
  <packagedElement name="SuD"
    xmi:type="uml:Class">
    <ownedOperation name="divides two numbers"
      xmi:type="uml:Operation">
      <ownedParameter direction="return"
        xmi:type="uml:Parameter"/>
    </ownedOperation>
  </packagedElement>
  <packagedElement name="User"
    xmi:type="uml:Class">
    <ownedOperation name="creates fraction"
      xmi:type="uml:Operation">
      <ownedParameter direction="return"
        xmi:type="uml:Parameter"/>
    </ownedOperation>
  </packagedElement>
  [...]
</uml:Model>
```

Figure 6. XMI representation of a UML diagram; a sample part

$$A = \frac{M_{\text{informal}}}{M_{\text{total}}}, \quad 0 \leq A \leq 1 \tag{1}$$

In Fig. 1, ambiguity equals to 3/4 since there are four methods in total and three of them are defined informally.

A method is formally defined when its signature consists of one specially reserved verb `creates` and an optional list of constructor arguments, as in the seventh line of Fig. 1. A formally defined method doesn't have any ambiguity since it is absolutely clear what is expected as its input and output. All it does is instantiate a new object and bind it to a variable. In Java that would mean calling a constructor.

The higher the ambiguity, the more system analysis is required for the requirements document to make it unambiguously understandable by everybody in a project. The ultimate goal of a system analyst is to break down requirements until a target ambiguity is reached. In the commercial projects explained in Sec. 8, the A metric was used for the planning of RE activities. Every iteration a group of RE specialists had a goal of decreasing the ambiguity until it reached a given value. It was not clear at the beginning how much work that would take, but soon the RE team obtained that knowledge and became capable of estimating complexity of work using A metric.

For example, in one of the projects, a decrease of ambiguity from 0.86 to 0.8 took five working hours of a system

Table 1. Empirically collected data from three commercial projects

	p_1	p_2	p_3
Total classes	39	12	19
Total methods	19	7	24
Total Java classes	80	75	295
Ambiguity achieved	0.80	0.95	0.68
Work spent on RE, staff-months	0.53	0.21	0.86
Total SRS contributors	4	7	2
Non-empty lines of Java code	9.5K	10K	32K
Project duration, months	7	26	11
Project budget, staff-months	5.5	7.5	9.5

analyst. During this work, eight tickets were produced for discussion, each of which took from one to three hours of work of requirements providers and other project stakeholders. Thus, a total estimate of 0.06 decrease of ambiguity costs approximately twenty hours of work. This is a very rough estimate and is applicable only to the project where it was measured, but in other projects it is possible to calculate similar metrics and manage RE activities according to them.

It was empirically observed that an SRS document becomes acceptable for implementation and doesn't confuse programmers when its ambiguity is less than 0.7. Of course, it is recommended to start implementation at earlier stages of a project, when ambiguity is rather higher, and generate requests for the RE team to improve SRS at certain places, where ambiguity has its peaks.

8 Empirical Results and Future Work

The language was used in three commercial projects. The numbers empirically collected are presented in Table 1.

The results collected from three software projects demonstrated that combining object-oriented paradigm with a CNL makes it possible to specify software requirements in a predictable and verifiable manner. The quality of requirements documentation was higher than in previously developed projects, according to the estimates of defects reported during project development and release phases.

It seems reasonable to analyze the effectiveness of the created CNL on a larger number of projects, including open source ones. In order to make such an analysis a methodology would have to be designed, to compare the effectiveness of different methods of requirements specification and identifying the advantages and disadvantages of each one.

9 Conclusions

A few important observations were made during commercial usage of the language. First, SRS document is much shorter than discussions around it (usually kept in bug tracking "tickets"). It takes days to discuss one small change in a class and just a few minutes to apply the change. Because of that, a traceability from SRS to discussion tickets is an important feature that has to be supported by a documentation management software. It is crucial to have an ability to trace back every requirement and remember the reasons behind it.

Second, complete and detailed error reporting is an important aspect of the compiler. Initial versions of it had a technical and simplistic errors reporting mechanism that didn't give enough information to system analysts and business owners. They were confused trying to edit a part of SRS in one of wiki pages and receiving a message like "incorrect syntax on line 45." Such situations required immediate attention from programmers. Soon it became obvious that the compiler, unlike programmers-oriented compilers of Java or C++, has to produce user friendly error messages (and warnings). In further versions of the language, a much richer reporting is going to be implemented.

Third, developing requirements specification with the language requires RE engineers to understand key principles of "object thinking" [4, 16], like "everything is an object" and "objects expose behavior, not state." Initial training was required in two projects.

References

[1] 2005. *ASD Simplified Technical English. International specification for the preparation of maintenance documentation in a controlled language.* Number Specification ASD-STE100. Simplified Technical English Maintenance Group (STEMG).

[2] 2005. *Unified Modeling Language: Superstructure.* Technical Report. Object Management Group.

[3] 2008. *Project Management Body of Knowledge, Guide* (4th ed.). Project Management Institute.

[4] Yegor Bugayenko. 2016. *Elegant Objects.* Vol. 1. Amazon.

[5] Betty H.C. Chen. 2009. Current and Future Research Directions in Requirements Engineering. In *Design Requirements Engineering: A Ten-Year Perspective In Design Requirements Engineering: A Ten-Year Perspective*, Vol. 14. 11–43.

[6] P. Clark, P. Harrison, T. Jenkins, J. Thompson, and R. H. Wo-jcik. 2005. Acquiring and Using World Knowledge Using a Restricted Subset of English. In *FLAIRS.*

[7] Alistair Cockburn. 2001. *Writing Effective Use Cases.* Addison-Wesley.

[8] Norbert E. Fuchs, Kaarel Kaljurand, and Gerold Schneider. 2006. Attempto Controlled English Meets the Challenges of Knowledge Representation, Reasoning, Interoperability and User Interfaces. In *FLAIRS.* Department of Informatics & Institute of Computational Linguistics.

[9] IEEE. 1998. *Recommended Practice for Software Requirements Specifications.* Technical Report IEEE Std 830-1998. The Institute of Electrical and Electronics Engineers.

[10] L. Kof. 2010. From Requirements Documents to System Models: A Tool for Interactive Semi-Automatic Translation. In *18th IEEE International Requirements Engineering Conference (RE).*

[11] Alistair Mavin, Philip Wilkinson, Adrian Harwood, and Mark Novak. 2009. EARS (Easy Approach to Requirements Syntax). In *17th IEEE International Requirements Engineering Conference (RE).*

[12] Tony Moynihan. 2000. 'Requirements-uncertainty': should it be a latent, aggregate or profile construct?. In *Australian Software Engineering Conference.* 181–188.

[13] OMG. 2011. *MOF 2.0/XMI Mapping, Version 2.4.1.* Technical Report. Object Management Group.

[14] Rolf Schwitter. [n. d.]. Processable ENGlish. http://web.science.mq.edu.au/~rolfs/peng/

[15] A. Sinha, A. Paradkar, H. Takeuchi, and T. Nakamura. 2010. Extending Automated Analysis of Natural Language Use Cases to Other Languages. In *18th IEEE International Requirements Engineering Conference (RE), 2010.*

[16] David West. 2004. *Object Thinking.* Microsoft Press.

[17] Karl Wiegers. 2003. *Software Requirements, Second Edition.* Microsoft Press.

[18] J. Woodcock and J. Davies. 1996. *Using Z-Specification, Refinement and Proof.* Prentice Hall.