UDC 004.415.2(043.3)
DOI: 10.15587/1729-4061.2019.175665

DEVELOPMENT OF AN APPROACH TO USING A STYLE IN SOFTWARE ENGINEERING

N. Sydorov
Doctor of Technical Sciences, Professor, Head of Department*
E-mail: nyksydorov@gmail.com
N. Sydorova
PhD, Associate Professor*
E-mail: nika.sidorova@gmail.com
E. Sydorov
PhD, Associate Professor,
Senior Principal Software Engineer
P&S Integrated Media Enterprise
Avid Development GmbH
Paul-Heyse-Straße, 29,
München, Germany, 80336
E-mail: Eugen.sidorov@live.com
O. Cholyshkina
PhD, Dean**
E-mail: greenhelga5@gmail.com
I. Batsurovska
Doctor of Pedagogical Sciences,
Associate Professor
Department of Information Security**
E-mail: batsurovska_ilona@outlook.com
**Interregional Academy of Personnel Management
Frometivska str., 2, Kyiv, Ukraine, 03039
*Department of Computer and Information Technologies**

Розроблено підхід керованого онтологією застосування стилів в інженерії програмного забезпечення. Сутність підходу полягає у використанні онтології не тільки для представлення стилів, но також для контролю застосування стилів під час створення і супроводження програмного забезпечення. При цьому, для представлення стилю створюється відповідна онтологія та засоби підтримки розробника, а для контролю застосування стилю в робочих продуктах фаз життєвого циклу програмного забезпечення створюються засоби (ризонери) на основі онтологічної бази знань. За представленням у дескриптивній логіці база знань містить дві складові – термінологічну (TBox) та фактичну (ABox). Перша складова створюється заздалегідь, шляхом виконання доменного аналізу. Друга складова створюється під час аналізу представлення відповідного робочого продукту.

З метою типізації, в контексті підходу, що розроблено, створено шаблони стилю онтології ядра інженерії програмного забезпечення, шляхом аналізу поняття стилю в різних доменах. Сформульовані основні характеристики стилю як доменне незалежного поняття, які представлено в шаблонах. При цьому, для обрання кількості шаблонів, що необхідні для представлення стилю, застосовано паттерн Work Product Pattern Application з Unified Foundational Ontology. Паттерн описує дії, що можуть існувати відносно стилю робочого продукту (Work product).

Розглянуто приклад реалізації підходу, шляхом дослідження запропонованого методу, керованого онтологією застосування стилю програмування в інженерії програмного забезпечення та архітектури засобу, що його реалізує. З застосуванням Protege показано побудову онтології стиля програмування і асистування програмісту. Розроблено і реалізовано архітектуру засобу контролю застосування стилю в робочому продукті фази конструювання – тексті програми. Основу архітектури складає база знань про відповідний стиль. Термінологічна складова бази знань містить інформацію відносно мов і стилю програмування і створюється заздалегідь розробником онтології. Фактична складова створюється ризонером для кожного представлення робочого продукту – тексту програми.

Засоби, що створено в контексті запропонованого підходу, автоматизують процеси, які мають місце під час застосування стилів в робочих продуктах фаз життєвого циклу програмного забезпечення

Ключові слова: інженерія програмного забезпечення, шаблон стиля, онтологія, дескриптивна логіка, стиль програмування

Received date 04.07.2019
Accepted date 25.07.2019
Published date 23.08.2019

1. Introduction

To date, methods and tools have become widely common in creating and maintaining reusable software products. The application of these methods and tools requires the programmer (software developer) to read, analyse and understand a significant number of work product representations of various phases of the lifecycle. Reusability is now largely expected from the specifications of the requirements as well as derivative texts and documentation. Therefore, it is primarily essential for software to be clear. The developer activity will be more efficient, the software will be clearer, and the programme development and maintenance will be cheaper when styles (standards) are applied while creating software to make the work products of different phases of the lifecycle understandable.

The use of a style in software engineering has traditionally been associated with construction, but today, due to the aforementioned circumstances, styles should be applied to all other software lifecycle processes. This is required by the special nature of software creation and maintenance processes, namely collective development and reuse. Applying a style means improving the quality and efficiency of software creation and maintenance. Therefore, the use of a style is very important, but it involves additional spending while solving the problems of learning the style description and adhering to it when creating work products of the lifecycle phases. These tasks are virtually unsolved except for individual construction phase processes.

A style description is a representation of the knowledge of a style, and it depends on the form of that representation, in particular the convenience of learning the style, the effectiveness of the appropriate means of its observance, and the ability to apply the style in different phases of the software lifecycle. It is proposed to use ontologies as a form of representing style knowledge, regardless of the lifecycle phase, to solve the problems of studying the description of a style and adherence to it when creating software products.

2. Literature review and problem statement

The notion of style has historically evolved in two scientific disciplines, namely philology and art. Nevertheless, the notion of style is now widely used and researched. In general, in terms of a style as a domain-independent concept, style is defined as a means of expressing an ideology or idea in a human activity [1]. The literature review shows [2] that there is no definition of style in the fields of human activity that could be used in software engineering. However, drawing on the main studies in these fields, the authors formulate general provisions that are the basis for developing such a concept of style. According to the literature analysis [2], there are three characteristics of styles: properties, tools, and factors. The properties of a style include the following: unity, ideology, task (creativity), emergence-disintegration, and value (aesthetic); the style tools presuppose media and elements as well as categories. A style as a complex system of elements arises under the influence of factors such as historical, social, stylistic, and style forming. Engineering methods for designing advanced software and lifecycle models based on component development and reuse have now become widespread [3]. Besides, software development also entails agile methodologies (such as extreme programming) [4], obfuscation [5] and egoless programming [6].

In this connection, tasks are posed to be related not only to reading the texts of software written by different programmers, in different programming languages and at different times but also representations of other work products created in the same way. It is known that the nature of representations is influenced by decisions made about a work product, such as architecture, algorithm, and programming language. In addition, the nature of the presentations is affected by the ideological, cultural and gender characteristics of the developer and the time period in which the software is created [7, 8]. Two documents are considered by the developer when applying a style – a description of the style and presentation of the work product in which the style is applied. Therefore, there are two processes, namely studying the description of an appropriate style and controlling the use of the style in the representation of the work product. The style description consists of rules and restrictions adopted to represent a particular work product. The impact of both processes on the efficiency and cost of development requires their automation, which is missing today. Of particular importance is the form of providing knowledge about both the style and the representation of the work product. This form can be an ontology. In [9], the results of ontology applications in software engineering are presented. Software engineering work products and software products are knowledge-oriented and are the result of knowledge-oriented actions.

Therefore, knowledge is a major component of software engineering, and forms of representation, methods, and tools for processing and applying knowledge play a significant role in software development [9]. Ontologies have been shown to be an effective means of representing the diverse knowledge that is used in software creation and maintenance processes. Today, ontologies are the best means of presenting and processing software engineering knowledge [1]. However, the question of using an ontology to solve style application problems in all phases of the software lifecycle remains unsolved, which is a consequence of traditional style application only in the coding phase and partly in the design phase when knowledge of the programming style (coding standard) is provided in the so-called configuration file in *XML* or *HTML*. Therefore, there are several studies dedicated to the use of ontology for applying a style in the design and coding phases [2, 10, 11]. However, they are aimed at solving the problem of presenting the appropriate style. The use of an ontology to control the use of a style in relevant work products in these studies was not investigated. This is often the reason for abandoning the style in lifecycle processes and leading to loss of productivity. Thus, ontologies play an important role in the implementation of stylesheets and, unlike common stylesheets such as *XML*, ensure improved software development and maintenance.

The sources of knowledge in software engineering are three types of domains – application, implementation, and problem [12]. In the process of building a domain ontology, it is preferable to use categorization, which constitutes a hierarchy or a network of ontologies, namely from the top-level ontology to the ontology of use in the application domain. There are also three approaches to building a domain ontology [13, 14]. The above categorization and approach are applied to building the ontology of software engineering styles. Building a top-level ontology is based on the use of existing, so-called foundational or formal ontologies [15]. Such a use is seen as a promising approach to building an ontology because it greatly accelerates the process of its construction. Only the basic ontology can now be used for a style domain. However, after building an ontology network for software engineering styles that is partially implemented in the study, applying the approach will ensure its effective development.

Given that there are many domains that can apply the notion of style in software engineering, although the general notion of style does not depend on the domain, it is advisable to use pattern-based templates to build an ontology [15]. Therefore, using the Work Product Pattern Application (WPPA) pattern of [16], which describes the *Style* artefact actions existing with respect to the *Work Product* in the case under consideration, style kernel ontology templates were created. Thus, the templates were established for domain-independent style concepts and processes for creating and applying the style. By reusing the style templates, it will be possible to build a kernel ontology for many relevant domains. Knowledge templates were introduced in [15] by applying the notion of a software engineering design template to denote a classified, parameterized representation of knowledge. Templates were suggested to retain the best practices in knowledge modelling. Later, in [17], a formal representation of the ontology template was given as follows: $T(p_1, ..., p_n) :: Q_T$, here $T(p_1, ..., p_n)$ is called the head and it contains the designation of the template T and a list of formal template parameters; Q_T is called the body and is a knowledge base. When applying a template, an instance of the head is created as $T(f_1, ..., f_n)$, which contains a list of actual parameters $(f_1, ..., f_n)$ whose values replace the notation of formal parameters $(p_1, ..., p_n)$ in the template body for $1 \le i \le n$ when processing the instance denoted by T.

Thus, a style representation is a form of knowledge of the agreed rules for creating a work product in the relevant aspect. Meanwhile, the tasks of using modern ontology representations to support professionals both in the study of a style and in its application remain unsolved. In part, for the sole purpose of solving the task of representing a style by an ontology, the design and construction phases have been completed. The absence of studies investigating the use of an ontology to represent a style in different phases of the lifecycle, as well as studies on implementing ontology-driven style control tools, suggests that research proposed by the authors of the use of ontology to automate the execution of both processes is appropriate when it is related to the use of a style in any phase of the lifecycle.

3. The aim and objectives of the study

The aim of the study is to solve the problems of applying styles in the phases of the software lifecycle by using ontology as a modern form of knowledge representation.

To achieve this aim, the following objectives were set and done:

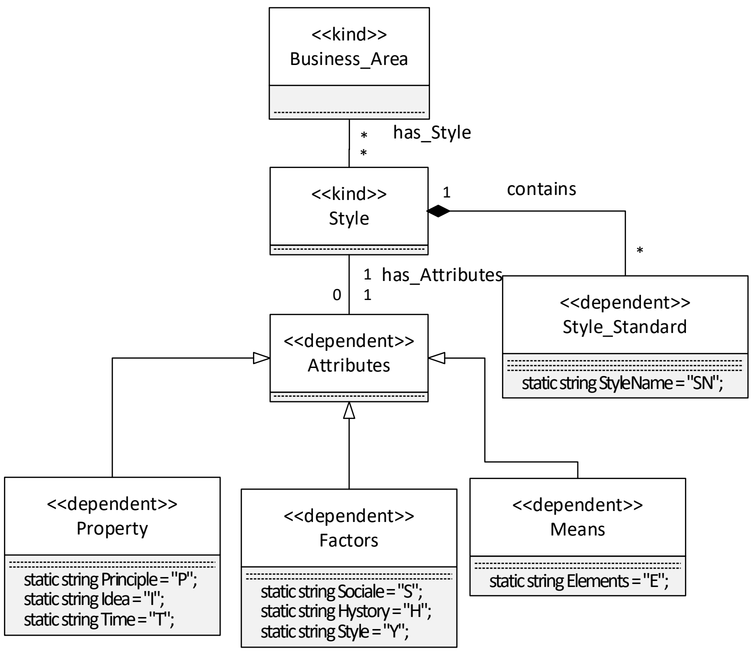
– to develop an approach to the application of a style in the work products of the phases of the software lifecycle through the use of an ontology;

– to create ontology templates for the presentation and application of a style in the work products of the software lifecycle phases in the context of the approach;

– to apply the style of programming in the work products of the construction phase to investigate the approach by implementing the ontology of style knowledge and the ontology-driven style application when using the created templates.

4. An ontology-driven approach to the application of a style in software engineering

Based on the WPPA pattern, the involvement of the *Style* artefact in software engineering can be of three types. First, the creation of a style; second, the application of the style; and third, the change of the style. We will consider the latter as a style creation. Thus, to build a kernel ontology in the study, three kernel ontology templates were created by applying the results of the style domain research and the WPPA pattern. One template is intended to describe the notion of style (Fig. 1), and the other two are aimed at describing the basic processes associated with the style in domains.



<<kind>> Business_Area
has_Style
*
*
<<kind>> Style
1
contains
*
1 has_Attributes
0
1
<<dependent>> Style_Standard
static string StyleName = "SN";
<<dependent>> Attributes
<<dependent>> Property
static string Principle = "P";
static string Idea = "I";
static string Time = "T";
<<dependent>> Factors
static string Sociale = "S";
static string Hystory = "H";
static string Style = "Y";
<<dependent>> Means
static string Elements = "E";

Fig. 1. An ontology of a style concept template

The templates are based on reusing the following stereotypes of the basic ontology (ELDM [18]): *Kind*, *Event*, *Category*, *Dependent*, and *Associative*. In addition, the concept of *Business Area* was used to refer to the domain and the term *Party* designated a social group or an individual. This explains that *Party* has the stereotype *Category* [18]. The *Style* concept is used to indicate a style in the template of the notion of style (Fig. 1). With the term *Style_Standard*, the style concept template describes the style; in software engineering, it may be standard. In the template there is an association of the composition type to one of the concepts, namely *Style_Standard* (Fig. 1). Individuals belonging to this concept are descriptions of the style or modifications thereof. These descriptions are presented either verbally, in relevant standard guides, or formally, for example in descriptive logic, when the appropriate processes of a style application are automated. For each template, a body was created, that is, an appropriate knowledge base, and formal parameters were defined, and the values of the corresponding actual parameters were determined to apply the templates in the construction of the programming style ontology. For example, the description of a style concept template would look like this (using *ALCQ* logic, or rather *ALCQIkey*, because the conceptual model is described in UML [19]). The template head is

$$TStyle(Business\_Area,\ Style,\ SN,\ P,\ I,\ T,\ S,\ H,\ Y,\ E) :: \\ :: TBox_{TStyle},$$

where *Business_Area, Style, SN, P, I, T, S, H, Y,* and *E* are the list of formal parameters of the *TStyle* template, and the template body $TBox_{TStyle}$ (knowledge base) is described by the following TBox:

{

$CN=\{Kind, Dependent, Business\_Area, Style, Attributes, Property, Factors, Means, Style\_Standard\}$, $RN=\{has\_Style, has\_Attributes, contains\}$, $Business\_Area \sqsubseteq Kind$,

$Style \sqsubseteq Kind$, $Attributes \sqsubseteq Dependent$, $Style\_Standard \sqsubseteq Dependent$, $Property \sqsubseteq Dependent$, $Factors \sqsubseteq Dependent$, $Means \sqsubseteq Dependent$, $\exists has\_Style.\top \sqsubseteq Business\_Area$, $\top \sqsubseteq \forall has\_Style.\ Style$, $Business\_Area \sqsubseteq \ge 0\ has\_Style.Style$, $Style$

$\sqsubseteq \geq 0\ has\_Style^{-}.Business\_Area,\ \exists has\_Attributes.\top \sqsubseteq Style,\ \top \sqsubseteq \forall\ has\_Attributes.\ Attributes,\ Style \sqsubseteq \leq 1\ has\_Attributes.\ Attributes,\ Attributes \sqsubseteq =1\ has\_Attributes^{-}.\ Style,\ \exists\ contains.\top \sqsubseteq Style,\ \top \sqsubseteq \forall\ contains.\ Style\_Standard,\ Style \sqsubseteq \geq 0\ contains.\ Style\_Standard,\ Style\_Standard \sqsubseteq =1\ contains^{-}.\ Style,\ Property \sqsubseteq Attributes,\ Factors \sqsubseteq Attributes,\ Means \sqsubseteq Attributes,\ Property \sqsubseteq \neg\ Factors,\ Factors \sqsubseteq \neg\ Means,\ Property \sqsubseteq \neg\ Means, Attributes \sqsubseteq Property \sqcup Factors \sqcup Means,\ Style\_Standard \sqsubseteq =1\ StyleName.string \sqcap \exists StyleName.\ \{SN\},\ Property \sqsubseteq =1 Principle.string \sqcap \exists Principle.\ \{P\},\ Property \sqsubseteq =1 Idea.string \sqcap \exists Idea.\ \{I\},\ Property \sqsubseteq =1 Time.string \sqcap \exists Time.\ \{T\},$

$Factors \sqsubseteq =1 Sociale.string \sqcap \exists Sociale.\ \{S\},\ Factors \sqsubseteq =1 Hystory.string \sqcap \exists Hystory.\ \{H\},\ Factors \sqsubseteq =1 Style.string \sqcap \exists Style.\ \{Y\},\ Means \sqsubseteq =1 Elements.string \sqcap \exists Elements.\ \{E\}$

}

The same descriptions are created for process templates. Using these descriptions and the reasoner, the template knowledge bases were tested for compatibility. When a style ontology is created for the kernel of the corresponding domain, then, by parameterizing the actual parameter values from the domain, the template body $TBox_{TStyle}$ is completed for the corresponding *Business Area.* In this case, the parameterization of the template is performed in accordance with the concept of style in the *Business Area* that is being currently considered. The types of values that formal parameters can take are not specified at this time, because the study has not considered the task of constructing template calculation software. However, this task can be accomplished by implementing a suitable macroprocessor for a language that will be used to describe knowledge (such as *OWL* or *RDF*).

To apply a style to any *Business Area*, it must be created. Therefore, an ontology template was developed for the style creation process (Fig. 2).

By creating a style, we primarily mean the process of determining the attributes of the style notion for a specific domain (for example, an individual of the *Business_Area* concept), and secondly, the creation of a description (standard) of a style (an individual of the *Style_Standard* concept, Fig. 1) for the individual of the *Style* concept (Fig. 3). Details of the style creation process are given in the corresponding individual belonging to the *Created_Style_Guide* concept (Fig. 2). Thus, the above-described *TStyle* template call was used to describe the ontology template for the style creation process. The created *Business_Area_Style* (Fig. 3) is used in the construction of an artefact (an individual of the *Artefact* concept) of the corresponding domain. Thus, the object acquires the property of having the corresponding *Style.* To describe the process of applying the style, a template of the corresponding ontology is created (Fig. 3).

The description of the ontology template for the style application process will look like this: $UsingStyle(Business\_Area, Style) :: TBox_{UsingStyle}$.

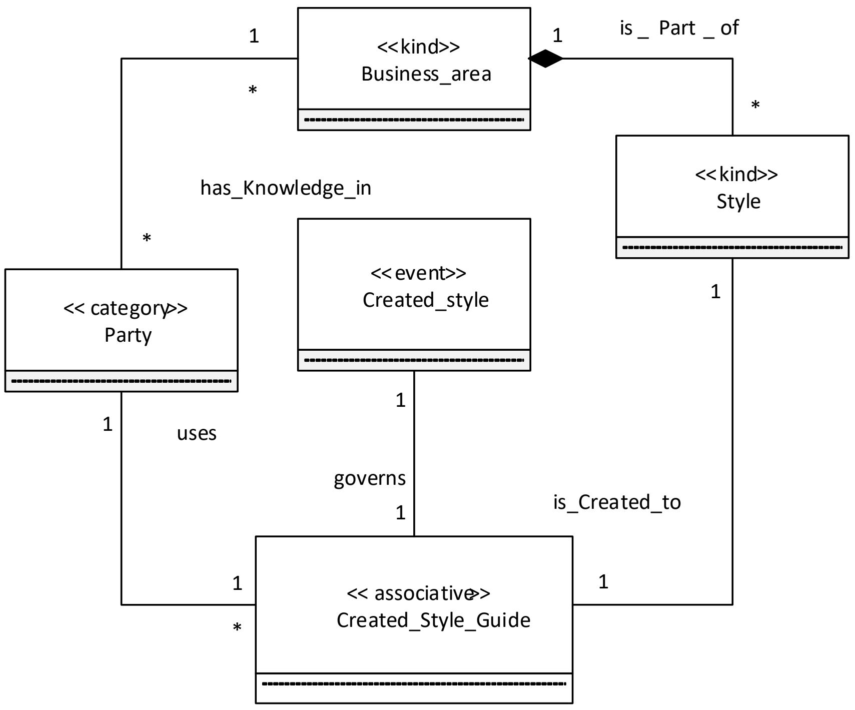


Fig. 2. An ontology template for the style creation process

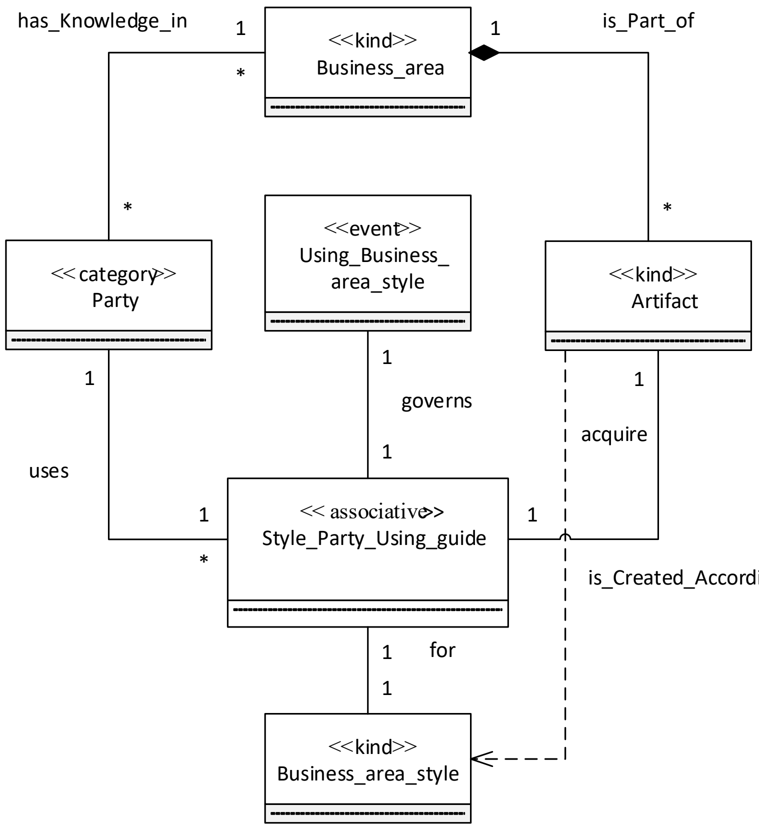


Fig. 3. An ontology template for the style application process

Style and process templates will be used to build the ontology of the kernel of any domain from the software engineering ontology network (Fig. 4). Therefore, the domain style in process templates is referred to as *Business Area Style*.

Using kernel templates, we create a kernel style ontology for the *Business Area*, with the *Software_Engineering* concept in terms of the use of a style in software engineering (Fig. 5).

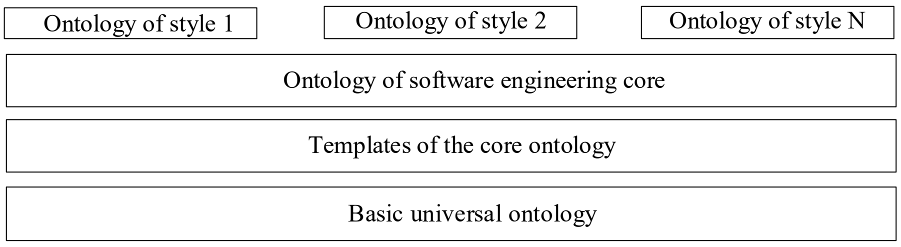


Fig. 4. An ontology network

In this case, for example, the values for the corresponding parameters *Property*, *Factors*, and *Means* of the *TStyle* style concept template (Fig. 1) for the design domain can be used from Table 1, which is based on [2].

In the ontology of the software engineering core, in its part regarding the style creation process (Fig. 6), the *Party* concept is considered as consisting of *Team* and *Person* concepts, since a style can be created by both a team and an individual performer of the software creation process.

It is assumed that a style is created for the relevant type of work product (the concept of *Work_Product_Style*, Fig. 6), that is, the result of any phase of the software lifecycle such as requirements, architecture, testing, and the programme text. In this case, the ontology of the style of a corresponding work product, for example, the style of a programme, is created by applying the style ontology and, in particular, the individual *Style_Standard* concept for the appropriate programming language and a description of the desired *TBox*. Of course, this creates an ontology of that part of the work product that represents the concept of *Work_Product_Style*, which is described by the individual concept of *Style_Standard* (Fig. 2).

Part of the ontology of the software engineering kernel regarding the style application process is presented in Fig. 7.

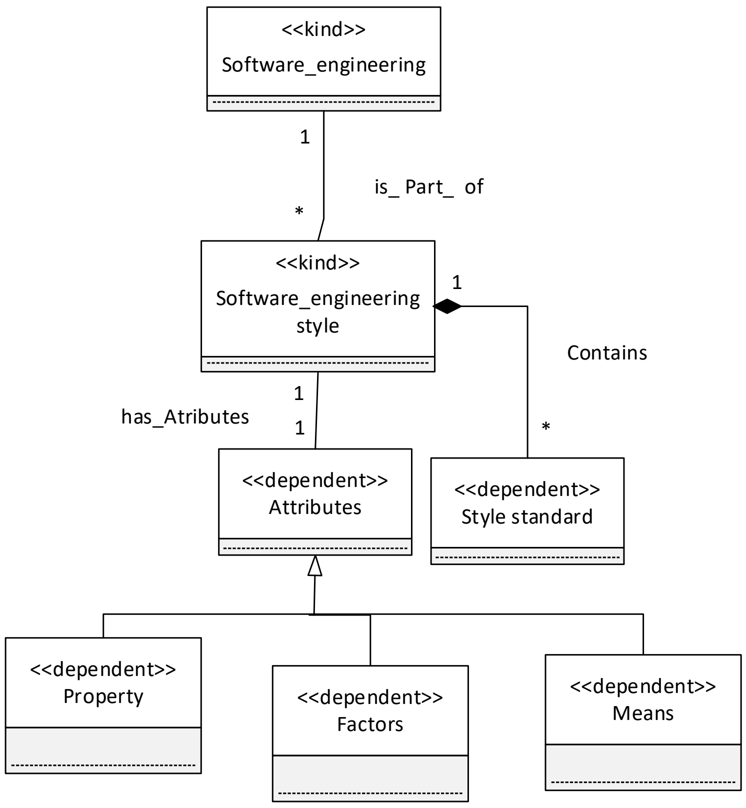


Fig. 5. Part of the kernel ontology regarding the concept of style for software engineering

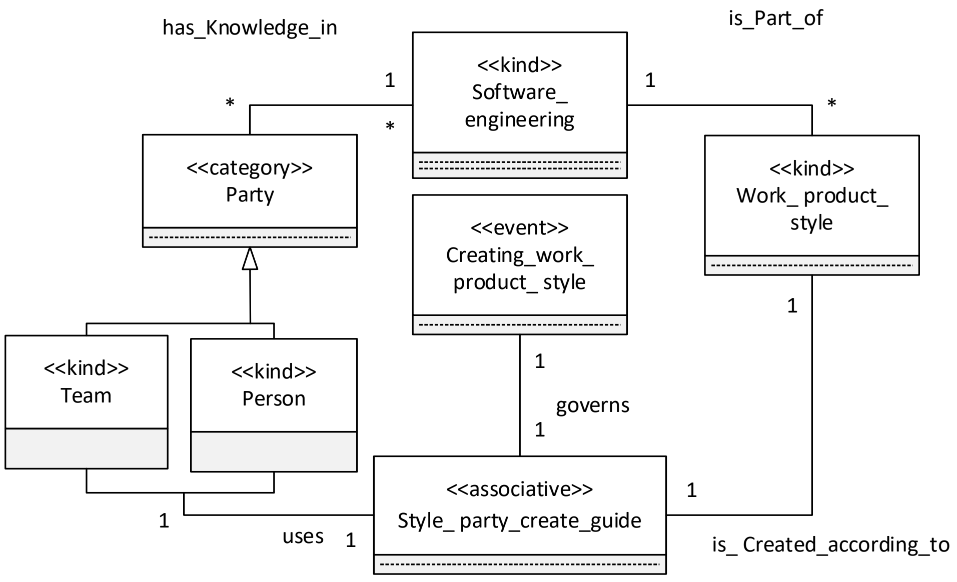


Fig. 6. Part of the core ontology as to creating a style

Table 1

Epochs and characteristics of styles

Epoch	Characteristics						
	Property			Factors			Means
	Term	Idea	The principle of importance	Historical	Social	Stylistic	Elements
before structural programming	1951–1975	Efficiency	Techniques of running a programme	A processor, a programme operator	A programmer, but all is from scratch	Low level programming	An operator GO TO
of structural programming	1975–1990	Intelligibility	Techniques of programming	A human programmer, a programme reader	Several programmers, but all is from scratch	Structural programming	A structural operator
after structural programming	1990–1996	Reusability	Techniques of using experience	A human programme developer	A team of programmers; there is experience	Modular and object-oriented programming	A module
of software engineering	1996	Creation of software in the given conditions	Techniques of proven software engineering	A software engineer	A team of programmers; there are requirements	Empirical programming	Documents

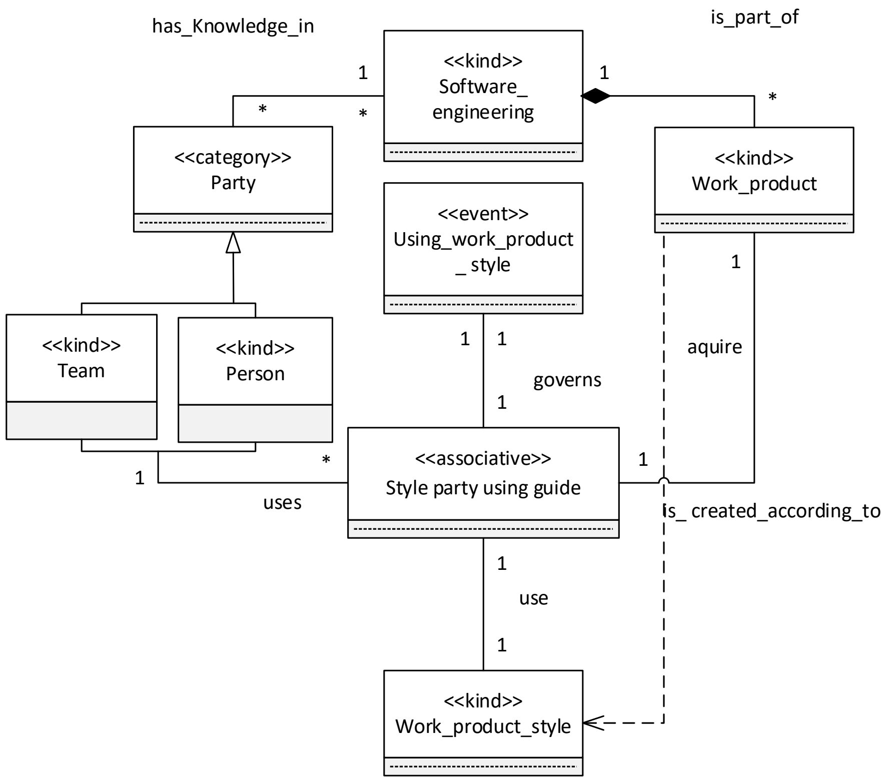


Fig. 7. Part of the core ontology as to the style application process

Part of the ontology of the software engineering core (Fig. 7) describes that the work product style is used when creating the work product (programme, architecture, or document). That is, the work product is created according to a style, by directly using the style description that was created (the *Work_Product_Style* concept). The work product (the *Work_Product* concept) acquires the properties of having a work product style (the *Work_Product_Style* concept).

5. Implementation of the approach by using a programming style

Table 1 shows that the concept of a programming style and its application in software engineering began to take shape as early as the 1970s. Today, there are known tools of static code analysis in *IDE*, such as *NetBeans*, *Eclipse*, *IntelliJ IDEA*, *Xcode*, and *Microsoft Visual Studio*, which check some rules of the use of a programming style. In such cases, knowledge of a programming style (coding standard) is provided by a description in a so-called configuration file in *XML* or *HTML*, but there are tools that present knowledge in a different form, such as *DLL*. Of course, such a presentation is inconvenient in terms of setting the tool for the appropriate coding standard. The study, based on the approach under consideration, proposed a method of ontology-driven programming styles [20].

Using the entered templates, let us build an ontology of creating a style in a domain (Fig. 8) and an ontology of using a programming style (Fig. 9).

A key function in these processes is performed by the description of the programming language style, which is usually represented by a coding standard (the concept of *Standard* in the description of the ontology). The coding standard is represented by a set of rules that govern the style of the programming language.

Thus, the programmer, when coding a programme, uses a programming style ontology, both to study the style and to check the style compliance in the programme. Therefore, two tools are needed – one to create an appropriate ontology and to support the programmer during coding [21] and the other to control the application of the programming style in the programme's derivative text (Fig. 10) [22].

Applying *Protégé*, a style analyst adjusts the ontology to the appropriate programming style and creates the *TBox*. After the set-up, the programmer learns the programming style with the help of *Protégé* while using the ontology. The second tool in terms of performing functions is similar to the reasoner. In terms of descriptive logic, the reasoner verifies the ontology compatibility (Fig. 11). In the tool created, this feature is added to the *Style Errors* feature.

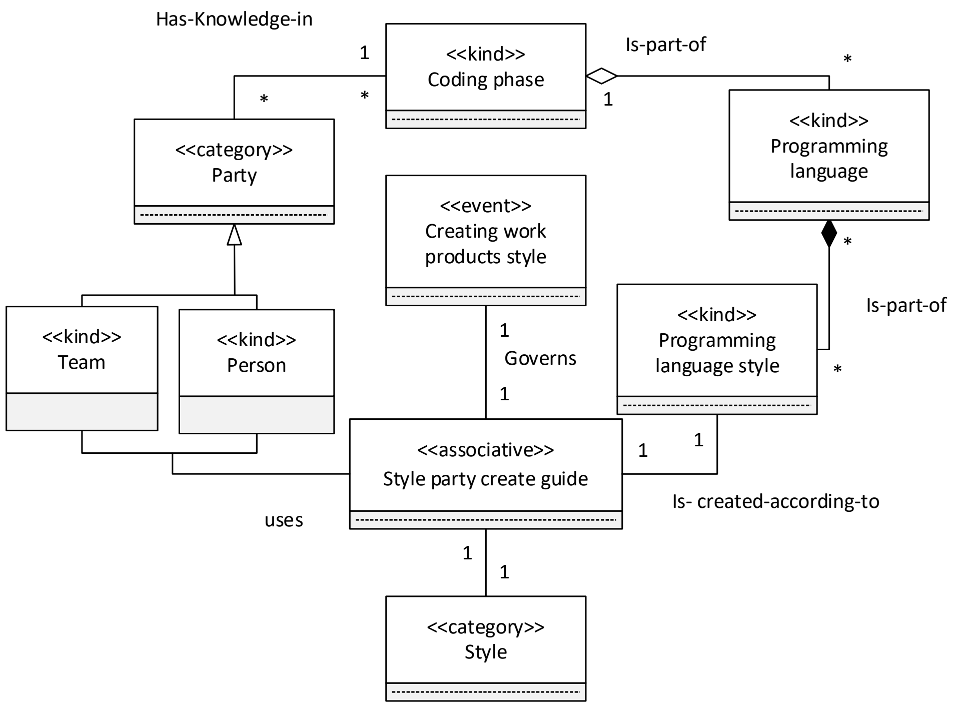


Fig. 8. An ontology of creating a programming style

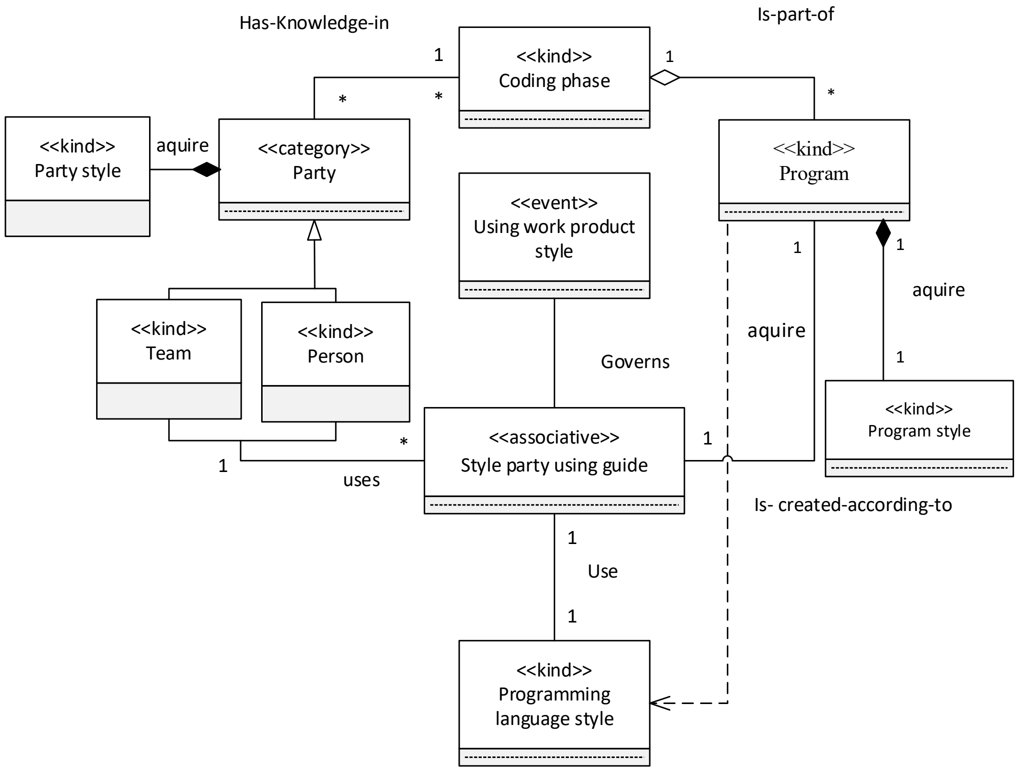


Fig. 9. An ontology of using a programming style

Protégé is used to create the *TBox*, a part of an ontology that contains terms that describe a programming style. Assertions about a target code (*ABox*) written by a programmer are created by the appropriate part of the tool called a reasoner, as it provides the appropriate service using a knowledge base (*TBox* and *ABox*). The service includes, first, verification of the ontology compatibility (a direct function of the reasoner), and second, the search for stylistic errors in the programme's target text. It is because of the necessity to implement the second function that the use of the 'regular' reasoner becomes impossible. Therefore, the study has produced a reasoner that performs this function – a *Style Ontology Reasoner* (*SOReasoner*). In addition, this reasoner provides an ontology template for standard rules for flexible construction of ontologies of different styles and programming languages.

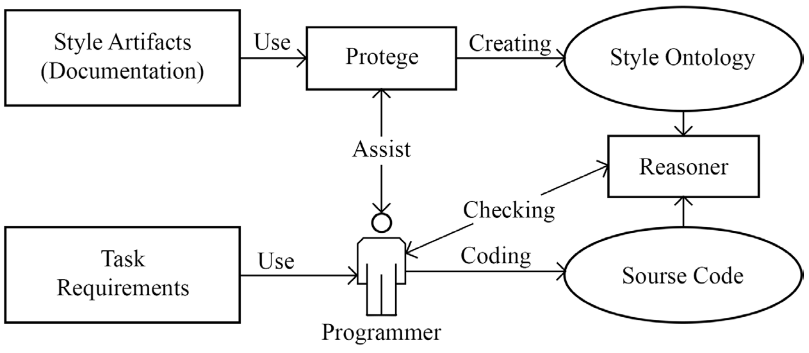


Fig. 10. A flowchart of using the tools

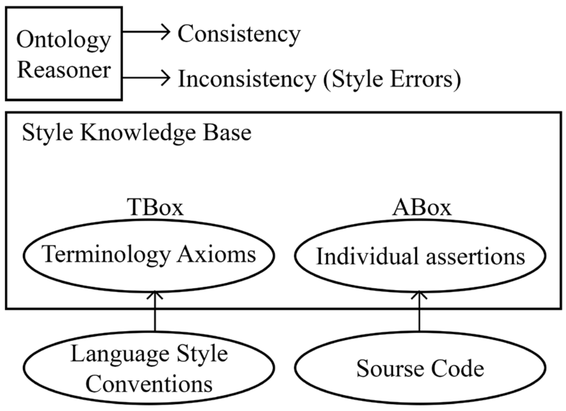


Fig. 11. A base of the programming style knowledge

In order to develop the appropriate software, it is necessary to create requirements for the *SOReasoner*, which must do the following:

– create a template of the ontology style rules and ensure its flexible structure;

– scan to process the ontology to create a set of rules of a programming style;

– check the target code (for example, in the Java language) and verify it for compliance with the rules of the programming style;

– provide a clear and detailed feedback to the programmer.

The following *SOReasoner* structure was chosen to satisfy the aforementioned functional requirements (Fig. 12).

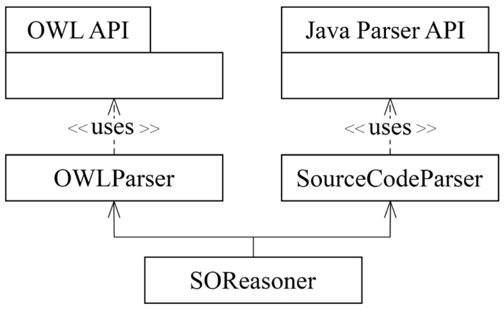


Fig. 12. A chart for the *SOReasoner*

Fig. 13 shows a sequence diagram of the representing the full functionality of the *SOReasoner*.

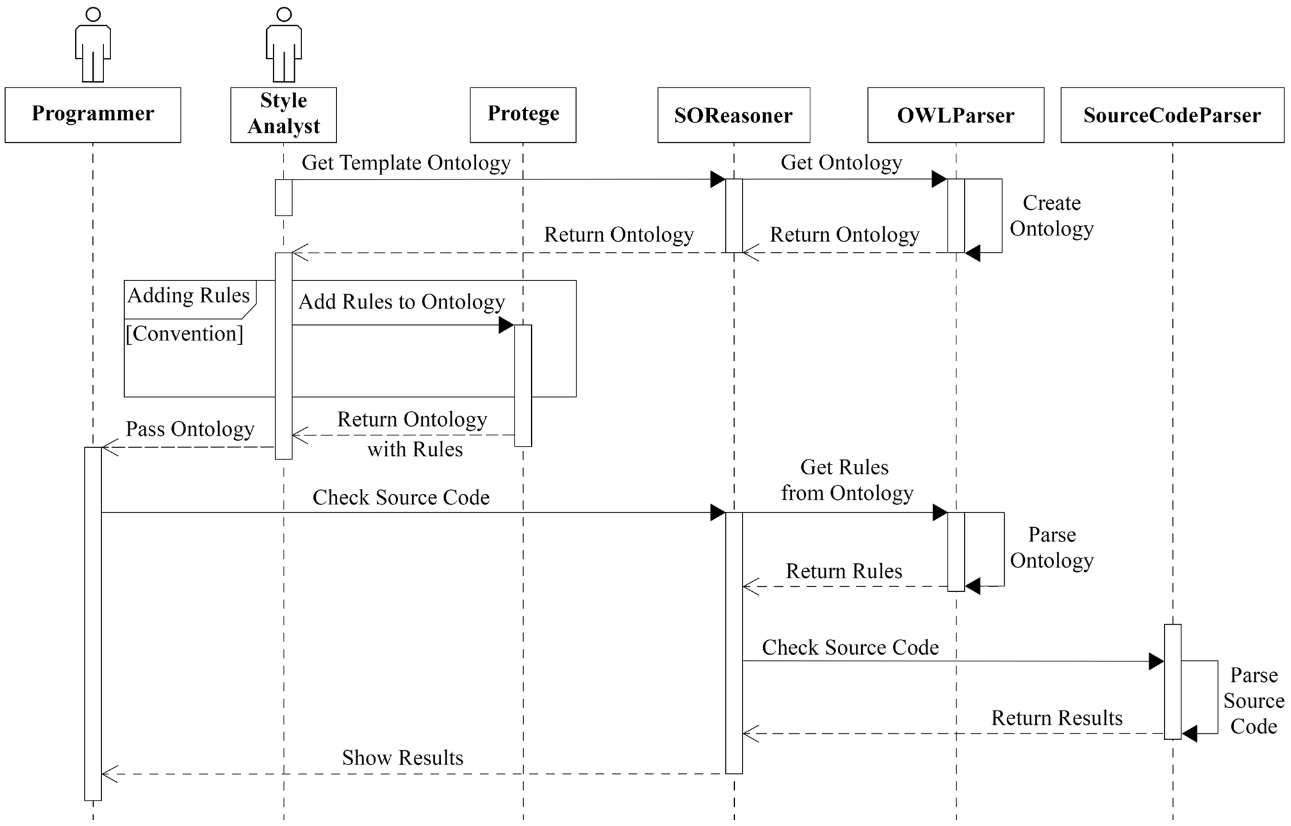


Fig. 13. A sequence diagram of the *SOReasoner*

6. The check-up of implementing the approach while using a programming style

Two types of tests were performed to verify the results of the suggested method on the basis of the described approach. First, the workability of the method and means was investigated, and second, the effectiveness of the tools developed was tested against the manual use of coding standards.

The results of the first type of research are discussed below. To test the performance of the method and the developed *SOReasoner*, an example of its application is considered for the use and control of the naming rules of *Java Convention*. As it has been noted, an ontology rule template was created and all rules describing the standards of naming were added. Then samples of the target code were verified and feedback was obtained through both the *SOReasoner* interface and the target code comments. In the *Protégé* interface, the corresponding structures can be visualized as *OntoGraf* (Fig. 14).

Once created, the ontology and the derived Java text are transmitted to the *SOReasoner* (Fig. 15).

As a result of text processing, the *SOReasoner* produces an error log (Fig. 16).

The log shows the percentage of correct identifiers and erroneous spots in the text, dividing them into groups by the rule types. Error messages are also posted in the target text in the form of comments.

Thus, the study performed to test using the programming style shows the correctness of the proposed approach and method based on it as well as the efficiency of the appropriate tools implemented. The use of ontology instead of the broadly used so-called *XML* or *HTML* configuration files ensures greater efficiency of the created tools.

In order to test the effectiveness of the tools by the second type of research, an experiment was conducted. The *Phillips Healthcare – C#* coding standard was selected for this experiment, and a model with appropriate metrics was created to investigate the results using the *Goal Question Metric* (*GQM*) method [23]. The following was done for the experiment:

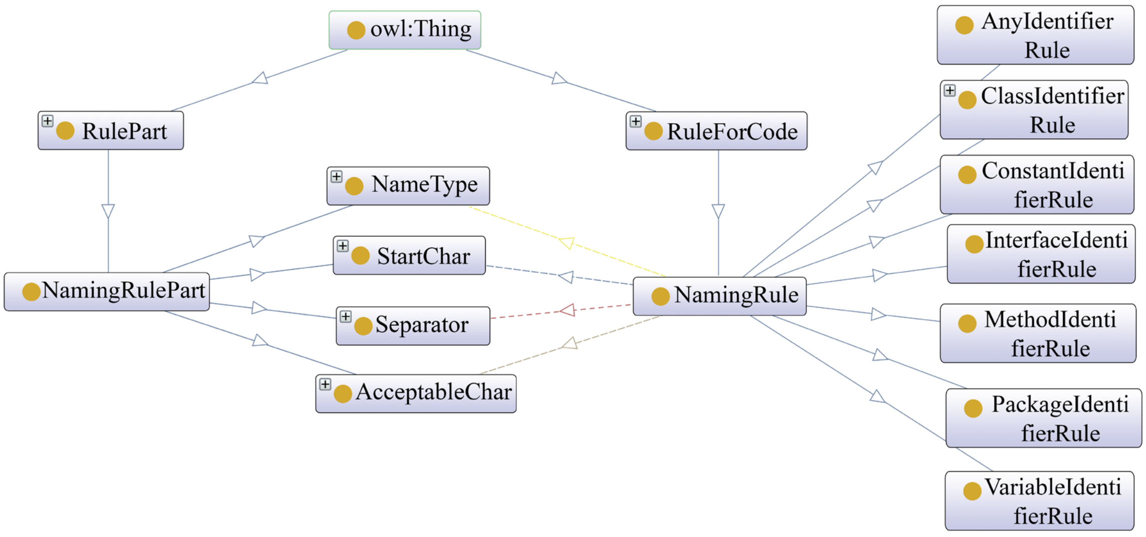


Fig. 14. An *OntoGraf* presentation of the rules of naming in *Protégé*

```
  public class SimpleClass{
    public int count;
  //Problems with idetifier '_value' [Start char,  Acceptable Chars]
  private int value;
    //Problems with idetifier 'Noun' [Name type, Start Char]
        public double Noun() {
  double d - (doble)(count+_value)
                return d;
                   }
}
```

Fig. 15. An example of a snippet of a derivative text before verification

```
Parse Log: 87.5% (7/8) of identifiers is correct):
 * Log from IdentifierVisitor *
 - [Variable] _value is BAD -> [Start Char, Acceptable Chars]
 - [Method] Noun is BAD -> [Name Type, Start Char]
```

Fig. 16. A snippet of an error log from the *SOReasoner*

1. Two teams of programmers (5th year students) were selected.
2. A task was set to maintain the style of programming (in terms of names), which was the same for everyone.
3. One group of the programmers was provided with a paper description of the *Philips Healthcare Coding Standards* for reference.
4. The second group of the programmers was provided with the *Protégé* tool and the ontology developed by the approach.
5. Both groups performed the task according to the created *GQM* model.

The results of the experiment show that the efforts spent on applying the standard (on studying the standard, coding, error identification, and defect correction) were generally reduced when using the ontology and the developed tool. However, the slight decrease in the efforts spent on the standard coding in verbal representation, as opposed to the efforts spent on standard coding when using the tool *probably* indicates that some time was spent on using the tool's interface and a small number of rules required to solve the task. Of course, the efficiency will be higher when specialists accumulate demand for the use of the tool, and the number of standard rules to be used will increase.

7. Discussion of the results of the ontology-driven approach to the use of styles in software engineering

The developed approach provides the style application to work products of the software lifecycle phases by using a single ontology representation tool. This demonstrates the possibility of creating a unified perspective on both the knowledge base design tools that describe the work product styles of the different phases and the specialist support tools for applying styles.

The proposed use of style templates and their application processes in the context of an ontology network shortens the time to create it. In particular, this was confirmed during the implementation of the ontology for the application of a style in the design phase to test the approach using an example of a programming style.

Automating the style control process by creating tools based on descriptive logic, unlike the known tools based on *XML* and *HTML*, simplifies the mechanism of customization controls, which in turn makes it possible to automate the processes of applying style in the work products of different phases of the software lifecycle.

To continue the implementation of the proposed approach, research is also performed on applying the ontology of style use in architectural design and documentation of software as well as in reverse engineering of software [24]. In general, the goal is to create a unified approach for all domains related to software creation and maintenance.

It is clear that the use of the approach in phases other than design and construction, on the one hand, requires preliminary research on the concept of style and the processes of applying it in these phases. For example, this concerns requirements specification, testing, or domain analysis. On the other hand, the problem of using non-verbal representation, such as *UML* graphics, of work products has not been investigated. This, however, is not an obstacle to using the approach.

8. Conclusion

1. An ontology-driven approach to the application of styles in software engineering has been suggested in the study. The essence of the approach is to use an ontology to represent styles when creating and maintaining work products of the phases of the software lifecycle. In the context of the approach, it is proposed to use the ontology network and its construction by applying style ontology templates. The use of ontology paves the way for the automation of style application processes in work products, which promotes the use of styles in software lifecycle processes, the creation of comprehensible work products, and it consequently increases the efficiency of professionals.
2. The *Work Product Pattern Application* pattern was applied to determine the number and nature of the style ontology templates for their use in software engineering ontology networks. The established templates of the network core style ontology were the concepts of style and processes of creating and applying a style. The use of the templates provides the construction of a core ontology for the network domains of the software engineering ontology.
3. The approach was tested on the example of a programming style for work products of the construction phase through the implementation of the ontology of style knowledge and the tools of the ontology-driven application of the style.

References

1. Sidorov, N. (2006). Software stylistic. Problems of programming, 2-3, 245–254.
2. Sidorova, N. (2015). Programming style ontologies and automated reasoning – systematic mapping study. Software Engineering, 3, 38–44.
3. Boehm, B. (2007). Software Engineering. John Wiley & Sons, 832.
4. Hazzan, O., Dubinsky, Y. (2009). Agile Software Engineering. Springer. doi: https://doi.org/10.1007/978-1-84800-198-5
5. Behera, C. K., Bhaskari, D. L. (2015). Different Obfuscation Techniques for Code Protection. Procedia Computer Science, 70, 757–763. doi: https://doi.org/10.1016/j.procs.2015.10.114
6. Weinberg, G. (1971). The Psychology of Computer Programming. Van Nostrand Reinhold, 276.
7. Raijlich, V., Wilde, N., Buckellew, M., Page, H. (2001). Software cultures and evolution. Computer, 34 (9), 24–28. doi: https://doi.org/10.1109/2.947084
8. Holovatyi, M. (2014). Multiculturalism as a means of nations and countries interethnic unity achieving. Economic Annals-XXI, 11-12, 15–18.
9. Calero, C., Ruiz, F., Piattini, M. (Eds.) (2006). Ontologies for Software Engineering and Software Technology. Berlin, 343. doi: https://doi.org/10.1007/3-540-34518-3

10. Pahl, C., Giesecke, S., Hasselbring, W. (2009). Ontology-based modelling of architectural styles. Information and Software Technology, 51 (12), 1739–1749. doi: https://doi.org/10.1016/j.infsof.2009.06.001
11. Abuhassan, I., AlMashaykhi, A. (2012). Domain Ontology for Programming Languages. Journal of Computations & Modelling, 2 (4), 75–91.
12. Sydorov, N. A., Sydorova, N. N., Mendzebryovsky, I. B. (2018). Software engineering ontologies categorization. Problems in Programming, 1, 55–64. doi: https://doi.org/10.15407/pp2018.01.055
13. Suárez-Figueroa, M. C., Gómez-Pérez, A., Motta, E., Gangemi, A. (Eds.) (2012). Ontology Engineering in a Networked World. Berlin, 446. doi: https://doi.org/10.1007/978-3-642-24794-1
14. Ghosh, M. E., Naja, H., Abdulrab, H., Khalil, M. (2016). Towards a Middle-out Approach for Building Legal Domain Reference Ontology. International Journal of Knowledge Engineering, 2 (3), 109–114. doi: https://doi.org/10.18178/ijke.2016.2.3.063
15. Clark, P., Thompson, J., Porter, B. (2000). Knowledge patterns. KR, 591–600.
16. Guizzardi, G., Wagner, G., Almeida, J. P. A., Guizzardi, R. S. S. (2015). Towards ontological foundations for conceptual modeling: The unified foundational ontology (UFO) story. Applied Ontology, 10 (3-4), 259–271. doi: https://doi.org/10.3233/ao-150157
17. Skjæveland, M., Forssell, H., Klüwer, J., Lupp, D. (2017) Pattern-Based Ontology Design and Instantiation with Reasonable Ontology Templates. Workshop on Ontology Design and Patterns (WODP2017), 15.
18. Department of Defense (2011). Data modelling guide (DMG) for an enterprise logical data model (ELDM). Version 2.3, USA, 184.
19. Calvanese, D. (2003). Description logic for conceptual data modelling in UML. ESSLLI, 23.
20. Sidorova, N. (2015). Ontology-Drived Method Using Programming Styles. Software Engineering, 2 (22), 19–28.
21. Sidorova, N. (2015). Ontology-Driven Programming Style Assistant. Software Engineering, 2 (24), 10–19.
22. Sidorov, N., Sidorova, N., Pirog, A. (2017). Ontology-driven tool for utilizing programming styles. Proceedings of the National Aviation University, 71 (2), 84–92. doi: https://doi.org/10.18372/2306-1472.71.11751
23. Basili, V., Caldiera, G. (1994). Goal Question Metric Paradigm. Maryland, 60.
24. Sidorov, N., Chomenko, V., Sidorov, E. (2008). Reengineering of the Legacy Software: the air simulator case study. Proceedings of the third world Congress "Aviation in the XXI–ST century, Safety in a aviation and space technology", 2, 33.88.–33.96.