A Formalism for Explaining Concepts through Examples based on a Source Code Abstraction

Mirco Schindler*, Christian Schindler† and Andreas Rausch‡
Institute for Software and Systems Engineering
Clausthal University of Technology
Clausthal, Germany
* Email: mirco.schindler@tu-clausthal.de
† Email: christian.schindler@tu-clausthal.de
‡ Email: andreas.rausch@tu-clausthal.de

***Abstract*—Design and architecture patterns are proven domain-independent solution approaches for common problems occurring in the development of software systems. Correct implementation of the design pattern is essential to guarantee the problem-solving capabilities of patterns. As the developers need to perform a context-specific adoption of the design pattern to the software system, we argue that their comprehension plays a crucial role in creating and maintaining such correct implementations over the system's lifespan. Even with migration and integration of legacy components into an adaptive System, where other paradigms are used, for example, must be compatible on a conceptual level. Given a set of implementation samples, this paper intends to separate essential syntactic information from varying aspects. We introduce an approach that abstracts given object-oriented implementations by semantically resolving and splitting an Abstract Syntax Tree into small paths. The contribution this paper provides is composed of two parts. First, we introduce an approach to extract negligible details of given concept examples to distill the essence of concepts, and the second part presents a formal foundation to describe and interact with concepts. Based on this foundation, we derive several underlying problem statements.**

***Index Terms*—Software Architecture; Architectural Concepts; Design Pattern; Concept Extraction; Source Code Comprehension.**

I. Introduction

This is an extended journal paper extending the work presented in [1]. Design patterns have been established for reusing proven solutions to a class of problems. Nevertheless, especially for a dynamic adaptive system, the correct implementation of adaptation mechanisms is essential for the quality of the overall system. Patterns are described informally or semi-formally as context-independent solution concepts. As a consequence, in order to apply a design pattern, it is necessary to embed it into the actual implementation context; to do so, a common understanding of the concept provided by the pattern had to be established [2] [3].

To relate implementation and architecture, the Unified Modeling Language (UML), for example, offers the mechanism of collaborations within the context of a composition structure diagram and the context-specific embedding in a given domain. Here, the description is separated from the actual application in modeling. Collaborations describe the composition of roles, which must be linked to specific parts of the application [4] [5].

Faulty implementations of patterns may produce functionally correct solutions but may lack the (mainly) non-functional properties provided by the pattern, such as specific modularity goals or specifications from the software architecture [6]. Inaccurate implementations can emerge not only in the initial implementation of the pattern but also from side effects introduced with changes, even elsewhere in the codebase [7] [8]. In particular, in a scenario where system parts and components are implemented and maintained heterogeneously and by different companies and development teams, as is unavoidable in an adaptive Software Ecosystem, for example [9].

If a legacy system or component is to be migrated and integrated, for example, to satisfy a specific adaptation mechanism, it is necessary to check the current implementation's compatibility. For this, it is helpful to find design patterns in existing code to comprehend the whole system better. Especially if it is written by other developers or not further documented. With a focus on code comprehension, it is necessary to extract more complex architectural patterns from simple code patterns iteratively. As a starting point, this paper contributes to recognizing design patterns by generating a data-driven interpretable representation of the design pattern from a set of implementation examples and counterexamples. No formal specification of the design pattern beforehand is needed. This paper addresses the following **Research Questions (RQs)**: RQ1: *Is it possible to abstract different concrete implementations of the same architectural design pattern so that the abstractions show a similarity?* RQ2: *Is it possible to formulate what the shared concept consists of across multiple samples?* RQ3: *Is it possible to classify unseen samples using the introduced formulation mechanism?*

The contribution of the extension is the provided formal context on top of the introduced approach. On the one hand, this is necessary to work out the underlying problems and, on the other hand, to provide a formal foundation for further work. First, we introduce the terms and understanding of a **Concept** (Definition 1) and **Context** (Definition 2) in general. Then, based on the extensional description of sets, we define

2023, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

an **Architecture Concept** (Definition 5) as a named Set of semantic equivalent examples. We introduce the **Abstract Syntax Graph (ASG)** (Definition 6) as an extension of the Abstract Syntax Tree (AST). Further on, we derive the term of an **Atomic Concept** (Definition 7) and a **Minimal Example** (Definition 10). Concerning the compositional characteristic of a concept, we propose a **Role** pattern (Definition 9). We close by introducing a **concept for instanziation** (Definition 11).

Furthermore, we have derived challenges from the proposed theory. We have outlined possible solutions to address them by introducing the so-called **Concept-Graph** (Definition 14) and deal with similarity instead of equality by defining the **Fuzzy-Hypergraph** (Definition 16) as an extension of the underlying graph representation.

Section II gives foundations on programming languages and the construction of the ASTs. Section III introduces the source code abstraction approach alongside two different levels of abstraction. Section IV is the evaluation of the stated RQs with a discussion of the results and limitations. Section V gives a formal approach for describing concepts. Section VI presents an overview of related work. Section VII opens challenges of extracting architectural concepts from given implementations. Finally, the conclusion and an outline of future work are given in Section VIII.

II. Foundation

This paper investigates the compositionality of abstract concepts. The inputs for the presented approach are syntactically correct but not executable source code artifacts. The focus is, therefore, on the static structure of a program. This structure is defined by the syntactic and semantic rules of a programming language. Each programming language consists of a set of programming concepts and specified paradigms, applying to modern programming languages that do not strictly follow one paradigm [10].

These concepts, defined by the programming language, are called **atomic concepts** (see Definition 7) in the following and manifest themselves in the source code by the language's **keywords**. Programming languages are formal languages because they consist of words over a given and finite alphabet [11]. Thus, the words are well-formed concerning a fixed and finite set of formal production rules [12]. Moreover, the lexical grammar of a programming language is usually context-free [13].

A grammar G consists of a four-tuple.

$$G(N, \Sigma, R, S) \tag{1}$$

with N : finite set of nonterminal symbols,
disjoint with the strings produced from G.
Σ : finite set of terminal symbols, disjoint from N.
R : finite set of production rules: $N \to (\Sigma \cup N)^*$
where * is the kleene star operator.
S : distinguished start symbol, $S \in N$.

We focus on object-oriented programming languages. Consequently, the type-system plays an important role and can be understood as an assurance to operations and documentation that can not be outdated. Types predefined by the programming language are so-called **atomic types**. Out of these atomic types, abstract types are constructed. The step of abstraction, which is also the foundation of the principle of information hiding, of abstract types is the structure defined by fields and an interface specified by the operations.

Since the languages considered here are formal, an automaton can be specified, which can process the character stream of the source code artifact. This is also the first step in compiling a program. Figure 1 shows the steps relevant to this paper of analyzing a program by a compiler. First, a scanner transforms the input stream into a language-specific token stream during lexical analysis. The tokens are also significant parts of a program, as they contain the atomic concepts of the programming language. This step reduces complexity, aggregates character, and identifies keywords. Then, a tree is generated from the token stream during syntactic analysis. A **tree** is a recursive data structure and a particular type of graph structure (a formal definition can be found in Section III-D) with a dedicated root node containing no cycles. Finally, each recognized token is converted to a node in the tree. Then, a semantic analysis is performed since not all rules, especially context-dependent ones, can be checked during derivation. This step also resolves the types, and names and annotates the tree's nodes to reflect this. Therefore, a symbol table is used to map each symbol with associated information like type and scope.

Through the instantiation of types, another kind of context-dependencies arises, which leads to the fact that the semantic meaning of a word derived by the grammar is no longer unique.

The challenge in extracting higher-level concepts up to architectural concepts is that these concepts are not included as concepts in the programming language. Instead, these can be understood as the composition of atomic concepts within a respective context. For program comprehension, it is essential to get a precise understanding of the concepts used in the implementation. Therefore with the increasing complexity and evolution of the program describing the essence of a concept in a comprehensible way to humans is a critical task.

It follows directly from the chosen class of language type that the set of generated concepts is countably infinite. Also, the set of reference implementations is infinite, with the difficulty that the same concept can be implemented in different ways. Thus, similarity could not be detected with a simple comparison of source code snippets.

III. Source Code representation

The main objective is a way to represent object-oriented source code samples on an abstract level compared to the raw source code files to enable interpretability on common parts and differences. Reducing information such as the naming of elements (e.g., methods, variables) or the order in which

2023, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

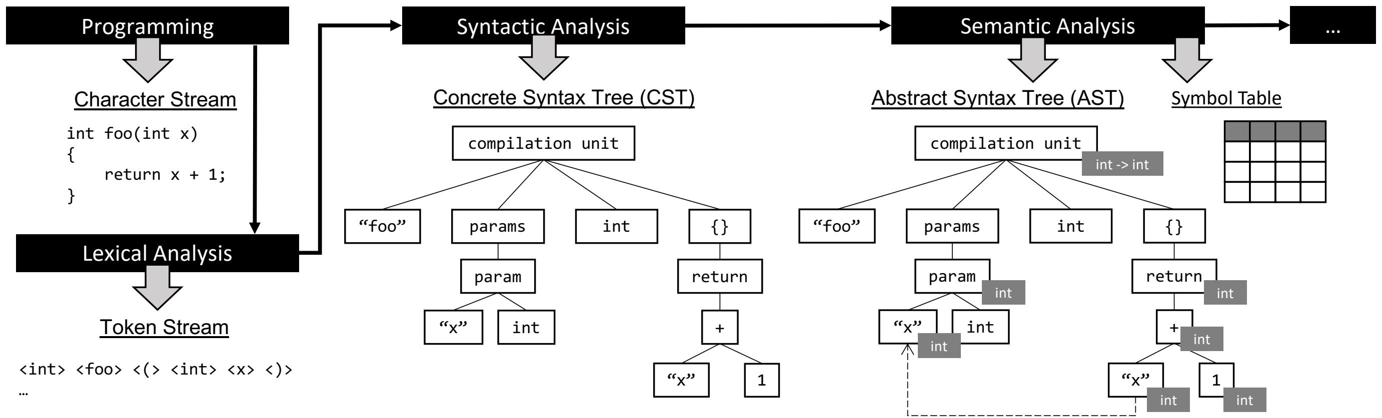


Fig. 1: First steps of a compilation process [13]

parts of the snippet (methods, variables) are declared or logic is handled (e.g., cases in a switch statement) help in this approach as it distracts from syntactical similarities.

We introduce two different levels of abstraction that both allow the expression of smaller parts reoccurring across different valid code snippets following the language's grammar rules. The **abstraction level** $High$ (Section III-B) is more abstract than level Low (Section III-C). The more concrete level of abstraction has superior expressiveness as it adds constraints across multiple reoccurring parts and allows for the distinction of elements (e.g., methods, variables).

We will elaborate on our general approach (Section III-A), being identical for both levels of abstraction first, then elaborating on $High$ (Section III-B), and adding in how we use the concept of uniquely identifying parts in Low. In Section III-C we explain how such constraints are added. In Section III-D we address how abstractions of different samples can be compared. Section III-E introduces the shared concept and how to construct it based on given code samples.

A. Source code abstraction approach

The approach, as illustrated in Figure 2, takes source code of arbitrary size as an input to generate an abstract representation in the form of a set of *Strings* that represent its syntax with additional information from the semantic analysis and aggregation. The Strings are sequences of tokens retrieved while processing the input that does not need to be exact sequences of the *Lexical Analysis*, as shown in Figure 1. A detailed walk-through example can be found in Sections III-B and III-C, Figure 1 contains only an illustrative one.

We analyze the code snippets *AST* to get a syntactic representation of the sample. The AST tokens get resolved during the aggregation phase constructing an *Aggregated Graph*. By combining the *ASTs* paths and the *Aggregated Graph*, we create the flattened *Abstraction*.

Subsequently, we formalize the required representations (AST, Graph, and the Abstraction) and concepts (path, aggregation function). Based on these definitions, we introduce the idea of a shared concept.

We define the **graph** $g \in$ GRAPH by the following signature:

$$g(V, E) := \{V = \{v_1, v_2, \ldots, v_n\}, E \subseteq V \times V\} \quad (2)$$

with V : finite indexed set of nodes.
E : finite indexed and ordered set of directed edges $\{v_i, v_j\}$

and a **tree** $t \in$ GRAPH being a special cycle-free graph with a root node v_{root} and a set of leaf nodes V_{leaf}

$$t(V, E, v_{root}, V_{\text{leaf}}) := \{g(V, E), v_{root}, V_{\text{leaf}}\} \quad (3)$$

$$\begin{aligned} \text{with } & V_{\text{leaf}} \subset V \wedge v_{\text{root}} \in V \\ & \forall v \in V \ \nexists v \mid \{v_{\text{root}}, v\} \in E \\ & \forall v_{\text{leaf}} \in V_{\text{leaf}} \ \nexists v \mid \{v, v_{leaf}\} \in E \end{aligned}$$

A path p in a tree t is a sequence of nodes V connected by edges E. The first node needs to be a leaf node and the final node needs to be the root node v_{root} of t.

$$p(V, E) := \{V, E\} \quad (4)$$

$$\begin{aligned} \text{with } & V := \{v_i \mid 1 \leq i \leq n\} \\ & v_1 \in t(V_{leaf}) \wedge v_n = t(v_{root}) \\ & E := \{\{v_{j-1}, v_j\} \mid 2 \leq j \leq n\} \end{aligned}$$

In the *Aggregation* step, the nodes of the AST get mapped to nodes of a resulting *Aggregated Graph*, by an aggregation function $f_{\text{aggregate}}(t) := V_t \rightarrow V_g$.

To construct the abstract representation a a concrete aggregation function combines the information of all paths P of the tree and the graph g itself. P is the set of paths containing each path from every leaf node of V_{leaf} to the root node v_{root}. It is defined by the following signature:

2023, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

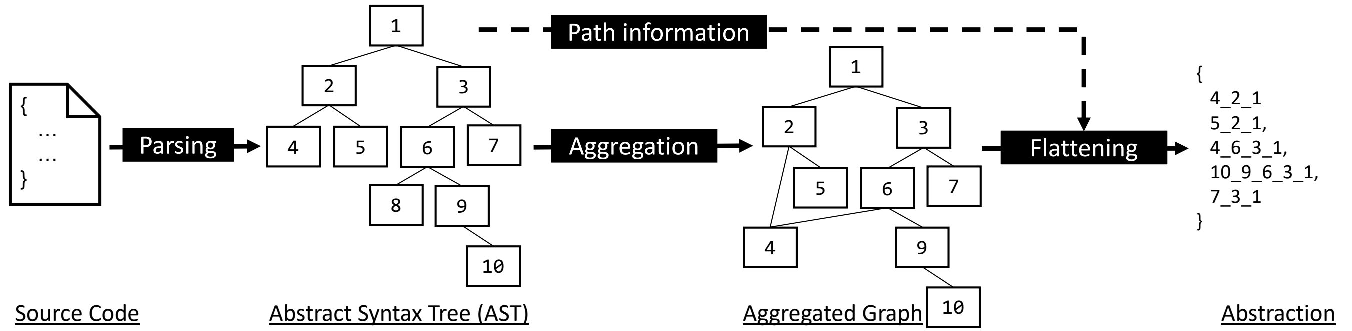


Fig. 2: Overall approach of the source code abstraction

```
1 public class FooBar {
2         public void foo() {…}
3         public void bar() {…}
4 }
```

Fig. 3: Java implementation of a class with two methods - *program 1*

$$P := \{p \mid p(v_1) \in t(V_{\text{leaf}}) \wedge p(v_n) = t(v_{\text{root}}) \wedge \forall v_{\text{leaf}} \in t(V_{\text{leaf}}) \exists! p \mid v_{\text{leaf}} \in p(V)\} \quad (5)$$

An **abstraction** is defined by the function f_{abstract} :

$$f_{\text{abstract}}(t, f_{\text{aggregate}}(t)) := (V_t, E_t) \times (V_g, E_g) \rightarrow P \quad (6)$$

To obtain the flattened abstraction, we combine the path information from the tree and the node information from the aggregated graph. The structure of the flattened Strings in the abstraction comes from the Paths P in the AST. The information of the relevant nodes results from applying the $f_{\text{aggregate}}$ function to the nodes of the paths $p \in P$. The final abstraction is a set of all distinct flattened Strings. In the example Figure 2, the aggregation merges the nodes *4* and *8* (from the AST). Those nodes represent the same semantic unit (e.g., the same literal) In this case p is *"8_6_3_1"*, after applying $f_{\text{aggregate}}$ the flattened String is *"4_6_3_1"*.

B. Abstraction level $High$

The nodes (tokens) in an AST have additional traits. We utilize the type of the node, which indicates what part of the language the node reflects (e.g., the declaration of a class or the call of a method). In addition, we use the information of more basic nodes (e.g., keywords, primitive operators) to represent individual nodes per manifestation (e.g., *TRUE* and *FALSE* for *Boolean* values) and one node per *Modifier* (e.g., *PRIVATE, PUBLIC, and STATIC*). On $High$, the aggregation step summarizes all nodes of the same type (e.g., all nodes that declare methods) into a single node.

Figure 3 shows a short code snippet that we will use for both abstraction levels to illustrate the approach and the resulting representations. The sample consists of a *public class FooBar* containing two methods (*foo* and *bar*). The content of the methods is left out, as it would be hard to display the resulting ASTs and graphs. As illustrated in Figure 2, we start with traversing the AST. The resulting tree is shown in Figure 4. In the tree, we can see the individual statements reflected by nodes and corresponding edges. Each node contains the information of the type of the node (e.g., *ClassDeclaration* for the root element) and, if available additional information such as the reflecting values associated with the nodes (e.g., *SimpleNames* reflecting the name of the class *FooBar* and the names of the methods *foo* and *bar*) or the proper modifier (in this case *PUBLIC* in all instances).

The higher-level **aggregation rules** of nodes are: (i) resolve keywords from the language. This includes *Primitive Operators, Primitive Types, Modifiers, TRUE, FALSE*, and (ii) reduce other nodes to the assigned types.

Figure 6a shows the resulting graph by applying the aggregation rules. Our abstraction aims to (i) consist of multiple small parts (ii) likely to be contained in multiple samples. From the tree (Figure 4), the graph (Figure 6a), and the aggregation rules , it is possible to construct the paths in Figure 5. Here underscore separates the nodes in a flattened path.

Carried information $High$: The paths extracted carry certain information enabling reasoning about the original program. For example, the second path states that there is a *PUBLIC ClassDeclaration* (line 1 of the code sample in Figure 3). The third path states a *PUBLIC MethodDeclaration* in a *ClassDeclaration*. From the information contained in the abstraction, we cannot tell which methods *foo* or *bar* this particular path represents.

On $High$, we cannot conclude across multiple paths. For example, it is impossible to state that the *MethodDeclaration* from paths 3 and 4 are part of the same *Method*. On the one hand, this shows that the abstraction level is capable of reflecting the general structures of the original code while being able to ignore the order of appearance in the original implementation. On the other hand, the abstraction lacks the distinction of different elements and the ability to connect multiple paths related to each other.

2023, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

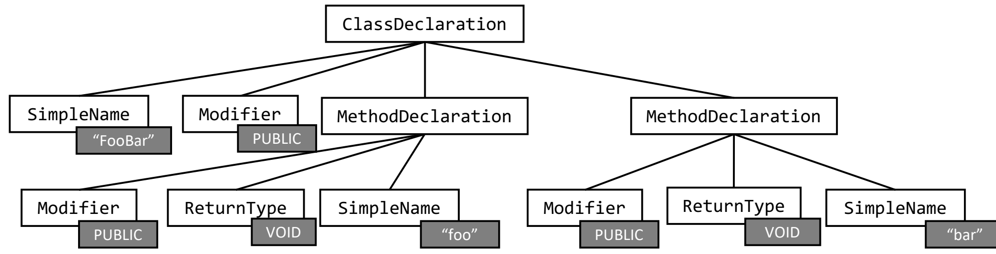


Fig. 4: AST of program 1

1 SimpleName_ClassDeclaration
2 PUBLIC_ClassDeclaration
3 PUBLIC_MethodDeclaration_ClassDeclaration
4 VOID_MethodDeclaration_ClassDeclaration
5 SimpleName_MethodDeclaration_ClassDeclaration

Fig. 5: Abstraction $High$ of program 1

C. Abstraction level Low

The stated drawbacks of $High$ get addressed at Low, containing more information from the original sample. The overall approach (Figure 2) still holds, with different steps in the aggregation phase. Semantic analysis of the AST is utilized to resolve elements. We introduce indices to those resolved elements, allowing the distinction of multiple nodes (of the same type and even across multiple types). The **aggregation rules** are as follows: (i) exactly as the first rule on $High$; (ii) identification of *Classes* and *Methods* by their signature; and (iii) resolution *(Simple)Names* with an index per unique name.

According to the stated rules, aggregation of the AST leads to the graph illustrated in Figure 6b. The indices allow the identification of elements. For example, we can still refer to the methods using index 1 and 2. The index is attached in the flat representation of the paths, separated by a hash symbol. The resulting paths of the code sample on Low are given in Figure 7. All the information of $High$ is still contained in this representation, as it is possible to remove all the indices and remove the duplicated paths resulting in Figure 5.

Carried information Low: The indices allow (i) to conclude across multiple paths, (ii) to distinguish multiple elements of the same type (e.g., the two *Methods*), and (iii) to express constraints that join different types seen in the aggregation process to superior entities (e.g., using one index for a specific *MethodDeclaration* and *MethodCallExpression*).

In Figure 7, all the paths are in the context of the same *ClassDeclaration(#1)*. We can draw conclusions about *MethodDeclaration(#1)* from paths 3 and 4 and state that it is *PUBLIC* and has the return type (*VOID*). The same holds for paths (6 and 7 respectively for the second *MethodDeclaration*). To distinguish elements across multiple paths the indices can be used similarly. We can tell that paths 5 and 6 are not belonging to the same *MethodDeclaration*.

D. Abstraction alignment

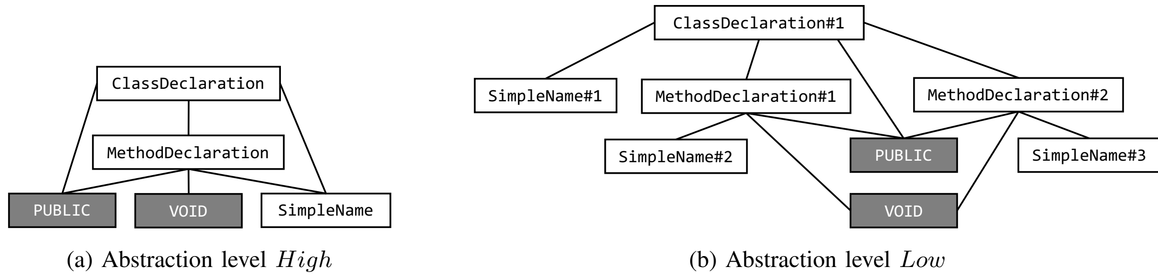
In the sections above, we introduced abstraction levels $High$ and Low for one single code snippet, both providing a set of paths representing the snippet. We showed how to reason across multiple paths of one abstraction. The next step in making use of the representation is to reason across multiple abstractions of different snippets x and y, by considering the sets of paths P_x and P_y, respectively, that they generate. We propose a *Jaccard Similarity* (Formula 7) based measurement, leading to a high similarity if a lot of paths are in both sets P_x and P_y, and little paths only in either set P_x or P_y.

$$jaccardSim(P_x, P_y) := \frac{\mid P_x \cap P_y \mid}{\mid P_x \cup P_y \mid} \tag{7}$$

On $High$ it is easy to be calculated without further steps needed, as no instance (e.g., multiple methods) are distinguished. On Low, the calculated similarity will depend on the indices assigned to the individual parts in the aggregation step, as the following example in Table I illustrates. The table is two parts, with the upper part containing different paths (left-hand side) and three abstractions (P_a, P_{b1}, and P_{b2}). An x in the respective cell means that the path is part of the abstraction. The lower part of the table contains the pairwise Jaccard similarity. The similarity calculated differs between *jaccardSim*(P_a, P_{b1}) and *jaccardSim*(P_a, P_{b2}) regardless of both P_{b1} and P_{b2} being equally valid representations of a *Class* having one *PRIVATE* and one *PUBLIC Method*.

In the presented approach (Figure 2) the indices get assigned in order of node processing. If a node (e.g., a *MethodDeclaration*) has been seen before, the assigned index is reused, otherwise, the next available index (per node type) gets assigned. This could lead to P_{b1} or P_{b2} for the same code sample, that are equally valid abstractions.

The idea to counteract this is by aligning the samples to improve the similarity measured without alternating the information contained in the abstractions. We achieve this by looking for (sub)graph isomorphism and corresponding permutations. In this example, a similarity-maximizing permutation of P_{b2} regarding P_a would be to swap the indices of the two *MethodDeclarations*. An important remark is that such a swap of indices needs to conform to the **permutation rules** (i) the swap of indices needs to be done for all occurrences to not invalidate a constraint and (ii) entities need to be respected, so the index of such related types need to be aligned uniformly.



(a) Abstraction level $High$

(b) Abstraction level Low

Fig. 6: Resulting graphs by aggregating nodes and edges of the example AST

TABLE I: SAMPLE ABSTRACTIONS AND CORRESPONDING PAIR-WISE JACCARD SIMILARITIES

paths on low abstraction level	P_a	P_{b1}	P_{b2}
PUBLIC_ClassDeclaration#1	x	x	x
PUBLIC_MethodDeclaration#1_ClassDeclaration#1	x	x	
VOID_MethodDeclaration#1_ClassDeclaration#1	x	x	x
PRIVATE_MethodDeclaration#1_ClassDeclaration#1			x
PUBLIC_MethodDeclaration#2_ClassDeclaration#1			x
VOID_MethodDeclaration#2_ClassDeclaration#1		x	x
PRIVATE_MethodDeclaration#2_ClassDeclaration#1		x	
jaccardSim with P_a	1	0.6	0.33
jaccardSim with P_{b1}	0.6	1	0.429
jaccardSim with P_{b2}	0.33	0.429	1

```
1 SimpleName#1_ClassDeclaration#1
2 PUBLIC_ClassDeclaration#1
3 PUBLIC_MethodDeclaration#1_ClassDeclaration#1
4 VOID_MethodDeclaration#1_ClassDeclaration#1
5 SimpleName#2_MethodDeclaration#1_ClassDeclaration#1
6 PUBLIC_MethodDeclaration#2_ClassDeclaration#1
7 VOID_MethodDeclaration#2_ClassDeclaration#1
8 SimpleName#3_MethodDeclaration#2_ClassDeclaration#1
```

Fig. 7: Abstraction Low of program 1

The **isomorphism** between two graphs is a *bijection* (one-to-one correspondence) between the nodes of the given graphs. As the graphs in our case are not guaranteed to be of the same size, we need to look into subgraph isomorphisms of the size of the smaller graph. A **subgraph** m of a graph g is denoted by:

$$m \subset g \iff V_m \subset V_s \land E_m \subset E_s \tag{8}$$

Finding such a *bijection* (candidate) of a subgraph consists of two steps, (i) fixing a suitable subgraph and the (ii) one-to-one correspondence. The verification of such a candidate can be done with Formula 9. The graphs q and m are converted to adjacency matrices (see Formula 10), and the *bijection* is formulated as a **permutation matrix** Q. Q is constructed with the nodes of one graph as rows, and nodes of the other graph as columns, the cells representing a correspondence are filled with 1, all others with 0. An adjacency matrix D_m contains a row and column for each node of the graph m, the respective cell is filled with 1 if there is an edge between those nodes, with 0 otherwise.

Let q be a graph isomorphic to m, for some *permutation matrix* Q:

$$q \cong m \iff \exists Q, D_m = Q \times D_q \times Q^T \tag{9}$$

Let D_m be the **adjacency matrix** of m, with:

$$D_m ij := \begin{cases} 1 & \text{if } \{i,j\} \in E_m \\ 0 & \text{otherwise} \end{cases} \tag{10}$$

After an isomorphism has been found, the indices can be aligned according to the permutation, allowing for the final check to see if the resulting paths match. This is needed as g (and D_m) do not contain the information of the original paths, so the graph will accept possible paths not contained in the abstraction.

E. Shared concept

We define a shared concept c_{shared} as the set of similarities and differences between a set of code snippets. The abstractions of code snippets, which contain the concepts c_{shared} are elements of the set A_{in} and code snippets, which are not an implementation of the concept c_{shared}, represent an element of the set A_{ex}.

Out of these two sets of abstractions of examples and counterexamples, the representation of the shared concept is derived as follows:

$$c(A_{in}, A_{ex}) := \{P_{in}, P_{ex}\} \tag{11}$$

$$\begin{aligned} \text{with } & P_{in} \cap P_{ex} = \emptyset \\ & \forall p_{in} \in P_{in} \land \forall a_{in} \in A_{in} \mid p_{in} \in a_{in} \\ & \forall p_{ex} \in P_{ex} \exists a_{ex} \in A_{ex} \mid p_{ex} \in a_{ex} \\ & \forall p_{ex} \in P_{ex} \nexists a_{in} \in A_{in} \mid p_{ex} \in a_{in} \end{aligned}$$

2023, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

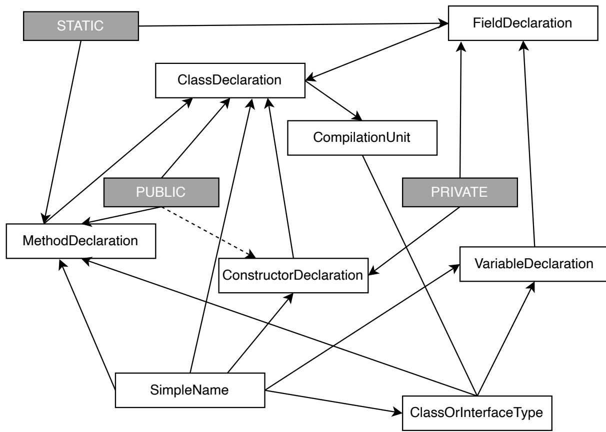


Fig. 8: Graph reconstructed from the P_{in} paths

Related to the above definition, a shared concept is described by two sets of paths P_{in} and P_{ex}. Each path $p_{in} \in P_{in}$ is included in every single abstraction of A_{in}. P_{ex} consists of paths p_{ex} retrieved by the set of abstractions A_{ex}. For a path to be included in P_{ex} it needs to be in at least one abstraction of A_{ex} and must not be in any abstraction of A_{in}. The idea of those exclusion paths is to handle paths seen in the programming language that have never been seen in a positive example that is expected to include the shared concept. By including samples from different repositories and business domains into the sets A_{in} and A_{ex} we hypothesize that the shared concept is containing business-domain-independent overlap.

After defining the shared concept in terms of P_{in} and P_{ex}, we can now reconstruct a graph based on the set of examples. We could obtain different graphs containing details of (i) all examples, (ii) only the examples A_{in}, or (iii) only examples A_{ex}, or even more fine-grained graphs. To construct the graph, we reverse the flattening step shown in Figure 2 in the overall approach. Note, that we are now working on the common paths of multiple examples and are constructing a graph that is not based on a single source code sample anymore. Figure 8 shows the newly constructed graph primarily based on P_{in} (Figure 9). Containing all the nodes and (solid) edges. In addition, the graph contains (dotted) edges coming from P_{ex} (Figure 10), which are between nodes that are present from P_{in} and not already present through P_{in}.

We hypothesize that the resulting graph contains the common entities and their syntactic relationships and other known syntactic relationships of the programming language that are not part of any example of the concept.

IV. EVALUATION

The evaluation starts with describing the data set, which was collected and annotated by the authors. The second part introduces the singleton design pattern, as this is the case study through the evaluation of the paper. The rest of the section addresses the stated RQs. We start by finding similarities on the abstraction levels (RQ1), calculating pair-wise Jaccard similarities on the abstraction levels, and analyze how the similarity compares on pairs that are both singletons, one of the samples being a singleton and non of the samples being a singleton. We formulate the shared concept as RQ2, by including all paths P_{in} we have seen in all samples (of the singleton), in addition, we formulated an exclusion set of paths P_{ex}, by specifically excluding paths that we have only seen in non-singleton samples.

Classifying new samples on the abstraction levels using the formulated shared concept (RQ3) is done as the last part of the evaluation.

TABLE II: ANALYSIS OF THE AMOUNTS OF PATHS IN THE ABSTRACTIONS

	min # of paths		max # of paths		avg. # of paths	
	low	high	low	high	low	high
singleton	17	17	2379	646	247.26	88.61
non singleton	6	4	2856	983	421.16	157.37
all samples	6	4	2856	983	334.21	122.99

A. Results

1) Preprocessing of the data set: The data set (*java-singleton*) collected and used to evaluate the abstraction approach consists of 230 java code samples labeled as part of this paper, containing the singleton design pattern and 230 additional samples that do not implement the singleton design pattern. The classes originate from different projects. The labels were applied by two authors, only containing samples confirmed by both authors. We chose the singleton pattern as a concept to evaluate as it combines a few criteria we consider beneficial as a showcase in this paper. The purpose of the pattern is widely understood and used in practice. The implementation is all in one place (the singleton class), leaving aside large search spaces [14]. Making it reasonable to identify samples in existing code but leaving room for the implementation to vary. It introduces manageable complexity to the task at hand while enabling us to collect a data set to evaluate the presented work, although the presented abstraction approach is not limited to the scope of a single class, file, or pattern. We abstracted all the samples on both levels of abstraction. Table II gives insights into the resulting abstractions. The table contains the minimum, maximum, and average amount of paths for all abstractions of a given set of abstractions. The sets show that the range of how many paths are in the samples varies a lot for each given set inspected. The average is also significantly higher than the minimum amount of paths. This indicates an overlap exists, and the samples have something to do with each other.

2) Results RQ1: As described in Section III-D we are going to measure similarity using the *Jaccard Similarity* (Formula 7). Table III summarizes details on the calculated similarities. Each row represents ten percent incremental thresholds, with the corresponding amount of sample pairs that are at least as similar as the threshold requires. The reported numbers are broken down into how many pairs are (i) both singletons, (ii) one of them is a singleton and, (iii) none of them is a singleton.

2023, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

TABLE III: NUMBER OF SIMILAR PAIRS ABOVE 10 PERCENT INCREMENTAL THRESHOLDS

	Low			$High$		
threshold	both singleton	one singleton	none singleton	both singleton	one singleton	none singleton
0.0	26335	52900	26335	26335	52900	26335
0.1	4843	389	31	22628	21859	7950
0.2	1444	4	4	10395	1630	600
0.3	372	0	1	3737	118	73
0.4	135	0	1	1589	9	28
0.5	73	0	0	669	0	16
0.6	43	0	0	289	0	8
0.7	32	0	0	153	0	1
0.8	28	0	0	95	0	1
0.9	27	0	0	63	0	0
1.0	25	0	0	30	0	0

This is done for both abstraction levels. The comparison of the samples with themselves is excluded from the table.

The data shown in the table support the assumption that the abstractions embody similarities related to the singleton design pattern. From the columns *both singleton* on both abstraction levels, we take that the stated RQ1 holds and that it is possible to abstract different concrete implementations of the same design pattern to show a similarity. As the similarity observed is significantly higher compared to the other columns in the table.

3) Results RQ2: We built a shared concept as introduced in our Definition 11. This part of the evaluation is limited to $High$ as no complete alignment of all samples has been calculated, leading to inaccurate results on Low. More on this is addressed in the limitations and future work section of the paper.

We follow common practice in Natural Language Processing (NLP) (compare stop word removal [15]) and trim the data so that we do not rely on too (un)common paths. We only keep paths in at least 5 percent and at most 95 percent of the samples of the dataset.

Table IV distinguishes the (non-)trimmed abstractions. It displays the number of paths belonging to specific subsets of the data set. For the non-trimmed row, many paths are exclusive to (non-)singletons $(4644 + 12813)$ compared to the 1996 paths shared. As the collected data set is small, contributing to infrequently observed paths, we focus on the trimmed column of the table. There are no paths left that are exclusive to the singleton samples. This allows us to ascertain, that there are no language constructs exclusively used to implement the singletons. In addition, eight paths are exclusive to non-singleton samples, which indicates that they are part of the programming language but not used to implement the singleton design pattern. No paths are seen across all non-singleton samples. The majority of paths are seen across both singletons and non-singletons. The shared concept retrieved from the data set *java-singleton* consists of twelve paths in P_{in} and eight paths in P_{ex}.

4) Results RQ3: To evaluate if it is possible to use the shared concept for classification of unseen code, we use a dataset [16] providing annotations of used design patterns. The dataset contains annotations for the following nine java projects: *QuickUML 2001, Lexi, JRefactory, Netbeans, JUnit, MapperXML, Nutch, PMD,* and *JHotDraw*.

```
1 STATIC_MethodDeclaration_ClassDeclaration_CompilationUnit
2 PUBLIC_MethodDeclaration_ClassDeclaration_CompilationUnit
3 SimpleName_VariableDeclarator_FieldDeclaration
    _ClassDeclaration_CompilationUnit
4 SimpleName_ClassOrInterfaceType_VariableDeclarator_FieldDeclaration
    _ClassDeclaration_CompilationUnit
5 SimpleName_MethodDeclaration_ClassDeclaration_CompilationUnit
6 SimpleName_ConstructorDeclaration_ClassDeclaration_CompilationUnit
7 PRIVATE_ConstructorDeclaration_ClassDeclaration_CompilationUnit
8 SimpleName_ClassOrInterfaceType_MethodDeclaration
    _ClassDeclaration_CompilationUnit
9 STATIC_FieldDeclaration_ClassDeclaration_CompilationUnit
10 SimpleName_ClassDeclaration_CompilationUnit
11 PRIVATE_FieldDeclaration_ClassDeclaration_CompilationUnit
12 PUBLIC_ClassDeclaration_CompilationUnit
```

Fig. 9: P_{in} paths on abstraction $High$ as shown in Table IV

```
1 SimpleName_NameExpr_MethodCallExpr_ObjectCreationExpr_ReturnStmt
    _BlockStmt_MethodDeclaration_ClassDeclaration_CompilationUnit
2 SimpleName_ClassOrInterfaceType_ClassOrInterfaceType
    _ClassDeclaration_ClassDeclaration_CompilationUnit
3 SimpleName_ClassOrInterfaceType_ObjectCreationExpr_ReturnStmt
    _BlockStmt_MethodDeclaration_ClassDeclaration
    _ClassDeclaration_CompilationUnit
4 SimpleName_MethodCallExpr_MethodCallExpr_MethodCallExpr
    _MethodCallExpr_ExpressionStmt_BlockStmt_MethodDeclaration
    _ClassDeclaration_CompilationUnit
5 SimpleName_NameExpr_MethodCallExpr_MethodCallExpr
    _MethodCallExpr_MethodCallExpr_ExpressionStmt_BlockStmt
    _MethodDeclaration_ClassDeclaration_CompilationUnit
6 PUBLIC_ConstructorDeclaration_ClassDeclaration_CompilationUnit
7 PUBLIC_ConstructorDeclaration_ClassDeclaration
    _ClassDeclaration_CompilationUnit
8 SimpleName_MethodCallExpr_ObjectCreationExpr_ReturnStmt
    _BlockStmt_MethodDeclaration_ClassDeclaration_CompilationUnit
```

Fig. 10: P_{ex} paths on abstraction $High$ (trimmed) as shown in Table IV

The authors of this paper validated the annotations. From the 13 annotations, we rejected seven, finding six additional singleton implementations that were not annotated as such before. Resulting in a total of 12 instances.

We conducted three experiments (Table. V)(i) *High incl.* only looking to include all the P_{in} paths, (ii) $High$ refers to in addition looking that none of the exclusion paths P_{ex} are present, and (iii) Low we used the inclusion paths P_{in} and

2023, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

TABLE IV: SUB SETS OF THE DATA SET AND THE AMOUNT OF THEIR EXCLUSIVE PATHS

	# paths only in		# paths in all		# paths seen
	singletons	non-singletons (P_{ex})	singletons (P_{in})	non-singletons	in both sets
trimmed	0	8	12	0	279
not trimmed	4644	12813	12	0	1996

associated indices that conform to the singleton pattern. Here we then aligned the indices of the samples (using subgraph isomorphism).

As a given sample can be classified containing a singleton (*Positive*) or not (Negative) and the ground truth label can tell if it is a singleton or not, we end up with the resulting combinations *True Positive (TP), True Negative (TN), False Positive (FP), and False Negative (FN)*. In our context, the classes mean: *TP*: prediction and ground truth agree on singleton; *TN*: prediction and ground truth agree on non-singleton; *FP*: prediction says singleton but it is not a singleton; and *FN*: predict says non-singleton but it is a singleton. To evaluate the performance of our classification of unseen samples we stick to the metrics of a confusion matrix used for the evaluation of Machine Learning (ML) models. Table V shows the results of the conducted experiments. Calculations of *Precision* also known as *Positive Predictive Value (PPV)*, *Recall* also known as *True Positive Rate (TPR), Accuracy (ACC)*, and *F1* are also calculated. A general remark is that the files were not changed or preprocessed. In the case of data set *java-singleton*, we isolated one class per code sample, contrarily those files used for the prediction are still untouched and possibly contain multiple classes.

B. Discussion

We have seen that abstractions produced by samples of various origins (different projects) carrying the same design pattern still carry a certain degree of similarity on the different levels of abstraction introduced in this paper. In terms of formulating the shared concepts, we were able to formulate a set of paths included in all samples and exclude a set of paths that we have only seen in other implementations that do not contain the same design pattern in the first place. The inclusion set P_{in} contains twelve paths, and the minimum number of paths seen in the set of singletons (see Table II) is only 17. This allows drawing the conclusion that at least one sample contains almost the bare minimum needed to implement a singleton in Java.

The exclusion set P_{ex} serves another important purpose, as it helps to explicitly describe what should not be part of the concept. In the case of the conducted evaluation, we reduced the exclusion set by trimming all paths that were in less than five percent of the samples, which allowed us to reduce the set from 12813 to only eight paths. We argue that this is useful because of the rather small sample size. We have not found another approach that similarly describes a concept by explicitly stating what is not part of the desired concept. Paths contained in P_{ex} were contrary to the definition of a singleton, as they contain paths for *Public Constructors*, and paths for creating new objects in the return statement of a method (which would bypass the singleton object, if it would be the *getInstance* method).

Also, the approach of the formulation of such a shared concept is flexible and adapts to the considered samples, and the more the samples share, the more is included. As the paths are interpretable, the abstraction levels introduced in this work also allow a formulation of such shared concepts from scratch, or to use only one example as a template to start with.

Both run on $High$ have a PPV around 0.5, while the TPR is higher, not making use of the exclusion paths P_{ex}. The ACC of both approaches is also nearly identical at 0.99. Caused by the data having a lot of *Negative* cases, in which both approaches are good at predicting. By comparing both runs, it is indicated that $High$ lowers the prediction of singleton (TP and FP) while introducing FN. The last part of the evaluation has been performed on Low. In this case, we introduced indices to the paths in P_{in}. We then aligned the indices of the samples, according to a valid permutation. The results have a PPV, TPR, and ACC of 1. This classification task was only performed on the 25 samples predicted as *TRUE* on the most permissive other approach (*High incl.*). For two main reasons, (i) the computation needed to find a subgraph isomorphism is NP-complete [17], and (ii) the previous check on $High$ for all P_{in} excludes all the other samples for not having all the needed paths. By knowing not all paths are present in the other samples (regardless of indices) it is not possible to find indices for those samples so that all paths are included afterward.

In terms of the classification performed, we have shown predictions with simple models, checking the exact inclusion and exclusion of specific paths on the $High$ and the same thing (after the computational intense subgraph isomorphism checking) on Low, with a perfect result as a reward. The prediction on $High$ is prone to overestimate the concept to be included, which is indicated by a precision around 0.5 for the not preprocessed unseen samples. Nevertheless, $High$ serves a valuable purpose in filtering the relevant samples to further look at Low.

TABLE V: RESULTS OF THE PREDICTION TASKS

	TP	TN	FP	FN	TPR	PPV	ACC	F1
$High\ incl.$	12	1914	13	0	1.0	.48	.993	.649
$High$	8	1919	8	4	$.\bar{6}$	.5	.994	.571
Low	12	13	0	0	1.0	1.0	1.0	1.0

C. Limitations

Although the approach introduced gives promising results in terms of the stated RQs, we have encountered some limitations on which we want to elaborate.

2023, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

The design pattern chosen is rather simple in terms of the variety the implementation offers. Looking at more complex structures (e.g., using general parts and specific refined parts could implement those as interfaces or (abstract) classes), in terms of the shown abstraction levels this would lead to not being reflected in P_{in} as of the current approach on building the inclusion set.

Assigning index-values to the shared concept Low was the only time (except the labeling) we relied on understanding the concept (of the singleton). To address that, the indexing can be seen as the maximum common subgraph problem [18] (being NP-Hard [17]). We do not have an implementation of this in our prototype.

V. A FORMAL APPROCH FOR DESCRIBING CONCEPTS

In this section, we discuss the term concept in general, and then a definition is derived for an architectural concept, which forms the foundation for this paper. The ”correct” naming of concepts is a very critical aspect. Therefore, first, the difficulty of naming ”things” from the perspective of natural language is discussed. Finally, the relation between the challenges of natural language, set theory, and graph theory is established.

A. Introduction of Concepts

The word concept is part of everyday language usage. It is applied in different contexts and domains. The term concept is generally defined as *”An idea or a principle that is connected with something abstract”* [19]. The term concept comes from the Latin word *concipere*: to put together, to formulate, to comprehend. From these definitions, essential characteristics that make up a concept can be derived.

The essential characteristic properties of a concept are:

1) made by people for people.
2) does not have to be realizable.
3) describes something abstract with the aim of comprehending a fact.
4) is context-independent and can therefore be applied in different contexts.
5) can be described in different ways.
6) can arise from concepts by hierarchical composition.
7) does not have to be explicit.

A concept has in common with a software architecture that people set it up as a communication instrument to create a shared understanding. Abstraction is an essential characteristic of a concept, as it is for software architectures. A realization, an example existing in reality, does not have to be present. A concept description is incomplete and limited to the essential elements of the concept. Whereby a concept, in contrast to the architecture of a software system, can be considered context-independent and has, therefore, more of the character of a reusable pattern. Although concepts can be applied in different contexts/architectures, they must be refined and adapted to the context.

Another common feature between architecture and a concept is the variety of description techniques. Everything is present, from natural language texts to complex, even executable description techniques. In the Latin origin of the word, it is already clear that a concept is something composed. Accordingly, the components of a concept play an essential role in the description and creation of shared knowledge. If a concept is compositional, it follows that there are different levels of abstraction, which have no global but only a relative reference. Implicit in the word definition, expressed by the synonyms plan, sketch, and draft is the intention to accomplish a task or solve a problem. Detached from any domain and referring to the previously listed characteristics, the term concept is defined as follows:

*Definition 1 (**Concept**): A concept is a context-independent, abstract description of a schematic solution to a class of problems generated by people for people.*

A concept can be described in two ways. On the one hand, by the description of the partial concepts and their composition or on the other hand by giving an example in the form of a sketch or an exemplary structure for concepts, which have an implementation in the real world. Both approaches have advantages and disadvantages and a direct correspondence to the extensional and the intensional definition of mathematical sets [20]. Describing by example is generally easier to understand but does not guarantee that the concepts' intention in all its manifestations has also emerged. However, a concept specification may result in a lack of reference to the real world and make it challenging to apply the concept. To understand the concept, an awareness of how it ”works” must be created. This can be done either through a clever selection of examples or an appropriate specification. Generally, these properties only come into play during composition and application in a concrete system (cf. [21] [22]).

In mathematics, the underlying principle is called conceptualism. According to this, mathematical terms are not fixed; they develop. The handling of the definitions in practice, the interaction with other mathematical concepts, and the exchange between people influence this development. This evolution of terms and mathematical concepts can lead to an increasingly uniform usage and, as in the case of set theory, to axiom systems. (cf. [23])

The definition and thus also its representatives can therefore change over time. How does a concept arise? – According to the identified characteristics, it arises by an extensional or intensional set definition, exactly when elements are composed in such a way that they contribute in at least one context to a problem solution. A set is intensionally described by the specification of a property that can be assigned to that set. An intensional definition in the context of free and arbitrary choice of property, antinomies, can be induced; an example is Russel's antinomy [24]. Such antinomies are met with an appropriate axiomatization so that the underlying universe is well-defined. An axiomatization for sets is the Zermelo-Fraenkel set theory [20]. In this, predicates are represented by sets, namely subsets of the powerset, constructed via the scheme of the axiom of elimination. If properties construct sets and concepts can be defined over sets, then properties must

2023, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

exist which represent concepts. Consequently, subsets of the power set exist with elements that fulfill a specific property and represent a concept.

It is easy to see that the essence of the concept must result from the semantic interpretation of its elements. Regarding the definition, a characteristic feature of a concept is schematic problem-solving. This is something that is not fundamentally true for properties. For a property, according to Platonic logic and as for the symbol $\in$, it is true that any object of a well-defined universe has a property or not. Additionally, the nature of a property does not change over time. From an epistemological point of view, this does not apply to concepts, as described earlier, for the principle of conceptualism. Because of this and the problem-oriented character, a concept can be understood as an embedding in an element to fulfill a particular property.

If elements are related to other elements, new properties can arise through emergence, attributed to the composed element. If a property becomes significant, then a property becomes a concept exactly if this property is additionally attributed to a problem-solving character. Both the significance and the solution of a problem are in the eye of the beholder and thus depend not only on the individual but also on the context.

According to the definition, a concept is a context-independent description. Thus, the application of a concept is its embedding in a concrete context. The word context comes from the Latin word contextus, close linkage, and relationship and can be generally defined as follows.

*Definition 2 (**Context**): A context represents a technical or situational setting that is meaningful for purposes and comprehension.*

B. The Difficulty of Naming Things

Another aspect that comes into effect when people interact with each other is verbalization, which becomes even more critical through context-sensitive embedding. This section explains why it is difficult to name things unambiguously so that communication partners have a shared, and equal understanding. It also discusses the differences and similarities between natural and formal languages, such as programming languages.

In linguistics, the term concept is used as the cognitive representation of an object or a cognitive category and is closely related to the meaning of a word [25]. This often makes it difficult to separate the concept from the context. It is not uncommon for words in a natural language to have different meanings. The meaning of a word arises from the context in which it is used. From this context, not only the meaning arises, but consequences are also connected, which are valid only in this context. In linguistics, the context is understood as the surrounding text of a linguistic unit, although it is not excluded that certain meanings remain open.

For most natural languages, words are formed by concatenation over an alphabet, given the construction rules of a concrete grammar. In a formal language, the language is entirely described by the grammar. However, natural languages are generally not formal languages. In this case, meaning is created only by combining letters to form words. Hence words are also referred to in linguistics as the atomic concepts of a language [26]–[28].

*Definition 3 (**Atomic Concept**): An atomic concept is a concept that is part of the language used to name realizations of the concept.*

It follows that atomic concepts can be clearly identified as part of language, as they can occur directly in the rules of grammar but are defined in dictionaries, especially in the case of natural languages. In linguistics and psychology, the meaning of complex expressions is attributed to the compositionality of concepts of language [28]. Thomas Ede Zimmermann describes the ordinary principle of compositionality as follows: *"The meaning of a complex expression functionally depends on the meanings of its immediate parts and the way in which they are combined."* from [29], p. 3 This definition goes back to the teachings of Aristotle and is known as the semantic compositionality principle (Frege principle). Named after the German mathematician Gottlob Frege (after [30] [31]).

The only fundamental assumption underlying this principle is that the atomic parts, which cannot be further decomposed, have a lexical meaning. Atomic concepts, like properties, have a stable meaning. For complex, i.e., composed concepts, the mere naming of real existing things or those of the imagination is sufficient to produce a lexical meaning.

In practice, this has its limitations, at least the same or similar naming, as Martin Fowler expresses in his article on the role pattern [32], which can lead to misunderstandings and misinterpretations. In the article *Dealing with Roles*, Fowler addresses the fact that just because a pattern is called a role pattern, this does not translate into the exact implementation and certainly does not intend the same semantics. However, if these implementations are conceptual, i.e., semantically identical, then the implementations can be substituted for each other. But he identifies about ten different patterns that are similar but not semantically identical. Furthermore, he lists the various advantages and disadvantages of these patterns, provided that it finally becomes apparent that a clear distinction of the pattern is necessary at the latest during the implementation.

What is the reason for the misinterpretation? – The cognitive scientist Lera Boroditsky investigated these phenomena and continued the theory about the relativity of language [33]. Boroditsky studied the influence of atomic concepts of a language on people's understanding and behavior in [34] [35]. She found in various experiments that different concepts of numbers, time, space, or of objects result in a generally different understanding of these concepts.

These and other experiments support the hypothesis that even the most basic concepts of human experiences, such as space, time, causality, and related objects, influence language and the nature of communication. The atomic concepts of language, its use, specific technical language, regional and even local differences, and the experience an individual has cause ways of thinking to be shaped by communication. This was investigated experimentally by selectively using metaphors that did not match the concepts of the language.

2023, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

After multiple repetitions, corresponding changes in thinking styles were observed in non-linguistic implicit association tasks [36].

One challenge of software engineering is the duality between understanding programming as creative creation on the one hand and the structured approach on the other. Suppose this is viewed from the perspective of linguistics and cognitive science. In that case, this explains the choice of variable names and identifiers and the phenomenon observed by Dahl, Dijkstra, and Hoare. They describe in [37] that with experienced programmers, a programming style similar to a style with artists in a painting is to be recognized, and the higher the understanding of the structured programming is, the more clearly higher-level structures are formed.

The aspects considered so far are mainly reflected in the identifiers and the more abstract structures, wherein a similarity between formal and natural languages exists. What is the practical difference between a formal and a natural language, except for the complete mapping in the corresponding grammar? – Marcus Kracht investigated the emergence of syntactic structures [31] [38], for which he used concepts of programming languages and set theory in particular. His work is based on a well-founded theory, which he summarized in the monograph, *The Mathematics of Language* [39]. He also concludes that the following three features are underrepresented in any semantics of a natural language: Indexing, Multiplicity, and Order.

To pick out just one aspect, we take a look at the concept of indexing. The idea behind it is the well-ordering of a set. This exists according to Cantor's well-ordering theorem for any set [40]. Although the ordering is not always obvious, the distinction between two objects is essential if they are, in fact, different. This becomes clear when we look at the AST. Each use of a variable is linked to the declaration of that variable via the symbol table. In a specific program context (visibility), a variable name always designates the same variable. Therefore, a new name and an index must be assigned when a new variable is declared. So, in a "text" written in a formal language, we know about the identity of each object.

As essential for formalism, the meaning must be independent of the naming. By indexing and distinguishing multiplicities, as is necessary for programming languages, the semantics result from the fact that different things can be identified. Kracht's motivation for his work stems from understanding the notion of compositionality. For natural language texts, almost everyone has an intuition about which constructs are compositional and which are not. The mother tongue can be mentioned here as an example. Children do not use complex composition operators to determine which words should be combined in which way. Instead, they develop a sense for this over time, which results in an intuition for new compositions. However, these notions are based on something other than a formal foundation. He, therefore, calls for meaning to be defined without mention of syntax. From his point of view, it is not part of semantics to specify how things are composed in syntax [38]. This follows from the fact that the functionality of a program does not change if variables are named differently.

Kracht therefore defines a concept as described in Definition 4 as set $R_{\approx}$ of semantic equivalent relations. By a relation we understand in the mathematical sense a set of ordered pairs. We write if it holds that the relation R_1 and R_2 are semantically identical: $R_1 \approx R_2$

Two relations are considered semantically identical if they can be transformed into each other by the operations given in Definition 3. The operation OP_L-1 is also called type extension because a new element of the universe is added to the relation. On the other hand, with the operation OP_L-4, the relation is extended by an argument already contained. Even if after the operation always, only the last one is appended, the operation OP_L-3 can be applied before in such a way permuted as long as the desired argument stands in the last place.

Definition 4, relations and thus orderings are described, but the definition of the concept itself is independent of it and, therefore, its semantics. According to Kracht, a concept is defined by a set of relations. This means that the order is only reflected in the concrete realization, which corresponds to the intuitive understanding. It is also easy to regard an active and passive sentence construction as semantically equivalent, although the order is fundamentally different. On the semantic level, concepts thus describe a linkage. Concepts can therefore be regarded as equivalent, even if they use different relations for representation.

Kracht, in his article *Using Each Other's Words* [41] states, similar to Boroditsky's findings, that for any two people, even if they use syntactically exact words, they do not assign the same meaning to them. In summary, naming things, especially complex composed constructs, is difficult because the composed meaning is derived from how certain parts are put together and the individual meanings of those same parts. The meaning of the atomic parts is learned individually and is thus preassigned an individual understanding. Based on this learned understanding, emergent meaning is composed. In other words, a shared understanding of a complex concept can only emerge if a shared understanding of its parts exists. If there is none or if there is a different one, misunderstanding is inevitable.

C. The Description of Concepts through Examples

If we take realizable concepts to be sets of implementations of that concept, they can be described in two ways, intensional and extensional. We refer to the exemplars of these sets as Examples. Under the term architecture concept, we subsume architecture patterns as well as design and implementation patterns. In this paper, we concentrate on the extensional description of concepts. However, even if concepts cannot be described thoroughly, it is nevertheless the better choice, related to the use case of the detection and extraction of so far unknown concepts.

1) The Concept as a Set of Examples: The definition of the architecture concept described as a **named set** (see

2023, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

Definition 4 (Concept (Linguistics), after ([31]p. 17 and [38], p. 65)): A concept is a set of relations $R_{\approx}$ with

$$R_{\approx} := \{R' \mid R \approx R'\} \tag{12}$$

Let Ω be the universe of a structure of first level predicate logic, then a relation R is a subset of Ω^n, $R \subseteq \Omega^n$. $R \approx R'$ then means that R' can be derived from R by any number of the following operations. Let $\vec{a} := (a_0, a_1, \ldots, a_n)$ and $\vec{a} \in R$, that is, an element from the relation R.

- $(OP_L$-1) Addition of an element from the basic set $R \mapsto R' \times \Omega = \{(\vec{a}, m) \mid \vec{a} \in P, \omega \in \Omega\}$
- $(OP_L$-2) Removing an element $R \times R' \mapsto R$
- $(OP_L$-3) Permuting the arguments $R \mapsto \pi(R)$
- $(OP_L$-4) Extension $R \mapsto \{(\vec{a}, a) \mid \vec{a} \in R\} = \{(a_0, a_1, \ldots, a_n, a_{n+1}) \mid (a_0, a_1, \ldots, a_n) \in R, a_{n+1} = a_n\}$

Definition 5 (Architecture Concept C (as named set)): Let CONCEPT be the universe of all known concepts. An architectural concept C is a named set of examples of the form

$$\begin{aligned} C := (id, R) \\ & id \text{ a finite string for which an injective mapping } f \text{ exists,} \\ & \quad \text{with } id \in \text{CONCEPT and } f : \text{CONCEPT} \to \mathbb{N} \\ & \quad \text{it applies } \forall C_i, C_j, f(C_i) = f(C_j) \Rightarrow C_i = C_j \\ & R \text{ a finite indexed set of semantically identical examples of the concept } C \\ & \quad R := \{r_i \mid \forall r_j \in R \text{ it follows } r_i \overset{C}{\approx} r_j\} \end{aligned} \tag{13}$$

As short notation it is defined $C_{id} := R = \{r_i \mid \forall r_j \in R \text{ gilt } r_i \overset{c_{id}}{\approx} r_j\}$. The notation $r_i \overset{C_{id}}{\approx} r_j$ means that the two examples r_i ans r_j are semantically identical with respect to the concept C_{id}.

Definition 5), is done via the duple (id, R) and not directly by a typical set-notation, to account for the special meaning of the id, i.e., the name of the concept. By naming the set and choosing an identifier from the words of a natural language, this set, and thus the concept, acquires an induced meaning for humans. However, this simplicity is countered by the complexity and diversity of the descriptive examples, which are context dependent. This means that each example represents the concept within its own context. A (shared) understanding exists when the context-free identifier and the set of contextual examples have generated a context-independent understanding in the observer.

The context, therefore, reflects the area of application in which the concept is embedded. It describes a refinement and concretization of the concept within this context. At this point, a further principle of software engineering comes into action – the principle of domain orientation. With the application of a concept, a mixture and a fusion with elements of the domain inevitably occur.

Similar to a design pattern, a concept can be described by the triplet of problem, context, and solution. As introduced in the section before (Section V-B) the name is not a unique identifier for the equality of two concepts. In general, two sets are equal if they contain the same elements. Though, under what criteria are two elements, i.e., examples? – This depends on the way the examples are represented. At this point, therefore, we can only refer to semantic equality, but not to structural equality. The semantic equality of two concepts is given as. If C_1 and C_2 are each concepts, and if $R_{C_1} \cap R_{C_2} = \emptyset$ holds, which means that the set of examples of the two concepts is disjoint, then these two concepts are semantically identical, iff.

$$\forall r_i \in R_{C_1} \land \forall r_j \in R_{C_2} \mid r_i \overset{C_1}{\approx} r_j \tag{14}$$

This leads directly to the following equivalences:

$$r_i \overset{C_1}{\approx} r_j \Leftrightarrow r_j \overset{C_1}{\approx} r_i \Leftrightarrow r_i \overset{C_2}{\approx} r_j \Leftrightarrow C_1 \approx C_2 \Leftrightarrow R_{C_1} \cup R_{C_2}$$

It must be remarked that this form of equality cannot be used to make a statement about the structural equality of two examples of a concept. In linguistics, two concepts that are equal in this sense would be called synonymous.

Suppose the examples are concrete implementations, i.e., idioms, so programming language-dependent descriptions. In that case, the context consists of the specific language and the parts of the business logic contained in the respective example. In addition, for more complex concepts, the locality principle does not apply. Instead, the pattern, more precisely, the roles of the pattern, are distributed among different system artifacts. It follows that it is legitimate for concrete source code excerpts to appear in examples of different concepts.

2) Example Representation and Atomic Concepts: In Section III, an example of a concept – the Singelton pattern – was examined. As a representation, a graph representation based on the AST was introduced. In this case, it was used to provide concrete instances of an example. Since this representation is a semantically identical model transformation, no abstraction occurs.

But is the structure of a graph suitable to represent higher-level concepts? – Basically, such a structure is suitable to represent any objects and their relations among each other. Moreover, the set of elements (nodes) and the set of relations (edges) can be extended freely, including aspects that cannot be extracted directly from the source code. The subject of this work is the semantic relations, not the syntactic ones. This means that a back transformation into source code is not required. Therefore, any extension of the graph is also allowed at this point. In the previous section, linguistic concepts were assumed to be understood as a set of semantically equivalent relations. In principle, however, only two-digit relations are represented by edges in a graph. However, David Hilbert and

2023, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

Wilhelm Ackermann found that any n-digit predicate can also be reduced to a less-digit one [42].

Generally, it is up to the modeling and thus the intended semantics of how the predicates are defined, thus also their rarity. This applies in this form only to the calculus of first-level predicate logic, but this is the object of investigation of Hilbert and Ackermann as well as of Kracht [31] [38]. A model based exclusively on triples, thus equivalent to a representation by a graph, is exemplified by the ontology [43].

We introduce the ASG to describe a more general formalism. In contrast to the concrete syntax tree, in which the nodes represent symbols of the grammar, the abstract syntax tree nodes represent concepts of the programming language and their hierarchical relationships. The structure is a simplification of the underlying grammar of the programming language. The edges represented in the AST are called syntactic edges E_{Syn} because they represent the syntax of the language. The edges constructed in the semantic analysis using the symbol table are called semantic edges E_{Sem}. By unifying these two sets of edges ($E_{\text{Syn}} \cup E_{\text{Sem}}$), the AST is extended from a tree to a directed graph, while it cannot be excluded that it is cycle-free.

The combination of syntactic and semantic edges is called Abstract Syntax Graph [44] and is specified in Definition 6. At this point, however, we must emphasize that this term is not a fixed term. The same applies to labeled AST, extended AST, or attributed AST, and it must also not be mixed up with the so-called term graph. Term graphs are often used in rewriting and automated refactoring and are often acyclic [45] [46].

The aggregated graph described in Section III-A and visualized in Figure 2 can be understood as a method-specific graph. The ASG here would be an intermediate representation between AST and aggregated graph. Compared to the ASG, the aggregated graph additionally has a node fusion operation performed. Like natural languages, atomic concepts exist for formal languages (cf. Definition 3). With respect to the representation of ASG, the notion of atomic concept is extended as follows:

Definition 7 (Atomic Concept): An atomic concept is a concept defined by the syntax, i.e., by terminal or non-terminal symbols of the programming language grammar, and represented by a node or edge type in ASG.

3) The Minimal Example: In this section, the reduction aspect of an example is discussed. From the definition for graphs, the following can be said in general about the features of a graph g.

1) It is directed,
2) can be a multi-graph (A multi-graph is a graph with more than one edge between two nodes),
3) has in general cycles (Cycle-free minimal examples exist, but they are trivial examples of atomic concepts) and
4) is connected.

In the previous sections, any graph was considered an example of a concept. It was only required that in the graph, at least once in the underlying program code, the named concept was applied. If several examples of a concept are compared, these may have a different number of nodes and edges. Therefore, it is not only possible to determine a minimal example among the existing ones, but a minimal example exists for each example. To illustrate this, the atomic concept *Class* is considered. If a method is added to a class, it remains an example of a class; even if this is repeated several times, the truth of the statement does not change. Even a class that has no methods can be considered an example of a class. This example illustrates two perspectives. From the context-sensitive view of the respective example, it can be examined which elements are not essential for the concept. From the context-independent view of the concept, the requirements for the example can be defined. If, for example, the concept *ClassWithMethod* is described, we expect at least one class with one method. The so-called minimum concept can be described as follows.

Definition 8 (Minimal Example of an Architectural Concept):

A minimal example of an architectural concept is an example, i.e., a piece of program code for which holds: if any character is removed from this piece of program code, then this is no longer an element of the set of examples of the respective concept.

What results for the ASG of a minimal example? – For an atomic concept, as in the example of the Class, this is defined by the programming language, namely by the minimal syntactically correct root tree where the root is of type class declaration. In this concrete case, it depends on the parser and grammar, different designations of the type are very likely. For non-atomic concepts, i.e., those that satisfy the compositionality principle, structures must exist that allow this concept to be decomposed. In the context of design patterns, which are widely accepted as architectural concepts, proven description templates have been discussed, as exemplified in [2] and [47].

One aspect of these description templates are the participants, also roles, of which the pattern is composed. Complex composed concepts are thus ascribed the existence of roles.

*Definition 9 (**Role** (according to [48])): A role is a observable behavioral aspect of an object in a concrete context.*

For a graph $g(V, E)$ representing an example of the concept C, it follows from conclusion 2 that there can be nodes that are not elements of a role, and thus not part of a behavioral aspect of the concept.

If we consider the collaborations of the UML as a description technique, then the parts, which are named collaboration roles, serve as a structural element [49]. In addition, other elements in UML are called roles in simple association, for example, as in the class diagram. In Figure 11, two classes and a binary association are shown. In such an association, the association ends are called roles. Here, the name of the association describes the semantics of the association, and the name of the role describes the meaning of the class related to this association [50]. We claim that a formalism should be independent of naming. This holds also for the naming of associations if we understand them as concepts. Moreover, we

2023, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

Definition 6 (Abstract Syntax Graph $g_{ASG}\left(t_{AST}(V_{Syn}, E_{Syn}), E_{Sem}, P, \tau\right)$): Let $t_{\text{AST}}(V_{\text{Syn}}, E_{\text{Syn}})$ be the AST (see Section III) of a program, then the abstract syntax graph is the extension corresponding to a tuple with the following signature:
$g_{ASG}(V, E) := \left(t_{\text{AST}}(V_{\text{Syn}}, E_{\text{yn}}), E_{\text{Sem}}, P, \tau\right)$

$$
\begin{array}{ll}
V_{\text{Syn}} & \text{a finite } \textbf{Set of nodes}. \\
E_{\text{Syn}} & \text{a set of } \textbf{syntactic edges}. \\
E_{\text{Sem}} & \text{a set of } \textbf{semantic edges}. \\
P & \text{a finite set of } \textbf{edge-terminals} \\
\tau & \text{a relation that maps each edge from } E_{\text{sem}} \text{ maps to an edge terminal.} \\
 & \tau := E_{em} \to P \\
 & \forall e \in E_{\text{Sem}} \mid !\exists p \in P \land \tau(e) = p
\end{array}
\tag{15}
$$

It holds that the node set of the ASG is equal to that of the AST:

$$V := V_{\text{Syn}} = \Lambda \,\cup\, T \cup\, \Sigma \tag{16}$$

Only the set of edges is extended, so that the following is true:

$$E := E_{\text{Syn}} \cup\, E_{\text{em}} = \gamma \,\cup\, \omega \cup\, \tau \tag{17}$$

speak in this case about associations, which are an object of subjective consideration since they no longer belong to the atomic concepts of the language; instead, they are composed concepts.

Fig. 11: Elements of a binary association according to [49], p. 135

Transferred to concepts according to Definition 5 results:

*Conclusion 1 (**Roles in Concepts**):*

A concept consists of a finite non-empty set of roles manifested in the examples or an atomic concept.

This conclusion is consistent with the role pattern, a widely used pattern of object-oriented analysis (OOA) [50] [51] [32]. A description of this pattern is given in the Figure 12. Three main features emerge from this description.

1) The item is role independent. In [47] this is also called the core class. It contains static, immutable functionality.
2) The item can take on different roles in any number of contexts.
3) The assignment can change dynamically and the item thereby aggregates the contexts to itself and thus dynamically receives its role-dependent properties and functionalities via the roles, which are described as association classes.

At this point, the variant Role-Relationship according to [32] was selected on purpose. According to the same paper, this variant is suggested if an item-object can take more than one role concerning another object, as is the case in this work if the system is chosen as context. However, this can also be true for an example of a concept. In the further course, this notion of role is further restricted to minimal examples.

The choice of modeling the role as an association class is also made on purpose. Other variants and especially perspectives in which the context is not explicitly modeled are given in [47] and [50]. The association class clarifies the membership of role-specific attributes and operations. It describes

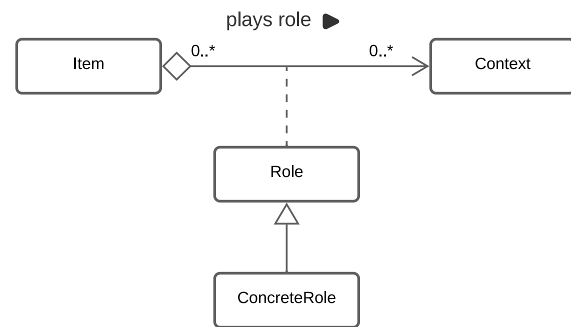


Fig. 12: The role pattern as UML class diagram and modeled as association class

properties that result from the relationship between the object and the context and cannot be meaningfully assigned to either class. The property of whether an object fills a particular role is no longer well-defined because the same class can occupy different roles in different contexts. The application of the role pattern destroys the well-definedness of the concept.

*Conclusion 2 (**Roles as Subgraphs**):*

With respect to the chosen representation, roles manifest themselves in subgraphs of a graph g of the conceptual example. Let h_1 and h_2 be roles, thus two real subgraphs of g, $h_1, h_2 \subset g$, then the following statements follow from the definition of role (Definition 9): (i)

1) $h_1, h_2 \subset g \nRightarrow h_1 \cap h_2 = \emptyset$
2) $g \setminus g' \neq \emptyset$ mit $g' := \bigcap\limits_{h_i \subset g}$

Roles do not define an equivalence relation on a concept.

In general, a complex concept consists of more than one behavioral aspect; at this particular step, only their existence is assured. The concrete roles are assumed to be unknown in the paper if not stated otherwise.

Therefore, for a minimal example, each node of the associated graph belongs to at least one role. All other nodes can therefore be removed from the graph. Furthermore, all nodes not essential for the concept can be removed; the same applies to the edges.

2023, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

This results in a minimal example of architectural concept defined in Definition 10.

*Conclusion 3 (**Connection of minimal Examples**):*

However, deletion operations may cause a minimal example to no longer be connected. For an example, since it is constructed from the ASG, it follows that it is connected. There may be dependencies which only arise dynamically at runtime, if such a link is the only one connecting two nodes, then this link is a bridge in the sense of graph theory and the graph decomposes into more than one component.

Another case represents the removal of the root element when the AST is considered as the spanning tree of the ASG. In many compilers, a parent node describing the combining unit is introduced as the root node, and in most cases, this is not essential to the concept and can therefore be removed.

In addition to removing, adding a node or an edge in an example is possible, also. The above example with the atomic concept of the Class could lead to the impression that these operations are possible without any problems. This is not true in general. For example, there are concepts where these operations can be repeated any number of times for particular nodes or edges, but there are also those for which this is not true. In the later part of the paper, the singleton pattern will be examined. If a public constructor is added to some variants of this pattern, it is no longer a valid example of this concept. This addition has destroyed the concept.

However, by introducing the minimal example, the instance term can be defined.

Definition 11 (Instance of a Concept):

Let $S \in \text{SYSTEM}$ be any system, g the associated ASG, then the subgraph $h \subseteq g$ is an instance of the concept $C(id, R)$ iff. h is a minimal example of C, so $h \in \hat{R}$ holds.

This also corresponds to the instance concept of object orientation. The correct formulation for a non-minimal example is that it contains an instance and does not represent it directly.

If an architectural concept C contains only minimal examples in its set of examples, then this is called a minimal architectural concept $\hat{C}$ as described in Definition 12.

Furthermore, it follows directly from Definitions 5 and 12 for architecture concepts:

$$\hat{C}_{id} \subset C_{id} \mid id = id \tag{19}$$

This means that there is a real subset relationship between the set of examples of an architectural concept and the set of examples of the associated minimal example. The size of the set of examples of a concept is countably infinite. A real subset of this set is formed by the minimal examples. Also for natural language concepts Kracht describes, relations which are minimal with respect to their length and thus in a certain sense even special. If $\hat{R}_i$ and $\hat{R}_j$ are two minimal relations of a linguistic concept (Definition 4), then $\hat{R}_j$ is a permutation of $\hat{R}_i$ and it holds [38]:

$$\pi(\hat{R}_j) = \hat{R}_i \tag{20}$$

Applied to architectural concepts, this would mean that two minimal examples $\hat{r}_i$ and $\hat{r}_j$ of a concept are isomorphic to each other, $\hat{r}_i \cong \hat{r}_j$. As a result, not only would there be a bijective mapping between these graphs, but the graph invariants, such as the number of edges or nodes, should not change. Nevertheless, this cannot be guaranteed in general. This is caused by operations such as fusion and contraction. As a result, the problem can be written as a partial graphisomophy problem Subgraph-Isomorphism (SGI) [52] [17], which is assigned to the complexity class of NP-complete problems. These aspects are discussed in more detail in the following chapter, where identification and extraction use cases are examined.

VI. Related Work

A similar approach to the one we propose is code2vec [53] [54], also working with an abstraction based on a set of paths. The main difference is the structure of the extracted path. All pairwise paths between the leaf nodes are examined and limited to a maximum number and length. They define the path-context by a triplet $< x_s, p, x_t >$, where x_s is the start leaf, x_t is the target leaf, and p the path between these nodes with the additional information whether a traversal takes place upwards towards the root element or downwards in the tree. The approach is presented here all paths from each leaf to the root are taken into account. Another limitation of code2vec is the abstraction context, which is one method. They argue that the order of source code statements is not relevant, or valid for this scope and the defined task. But as shown in [55], the relation between source code elements for higher concepts (like classes) is essential to perform structural or behavioral related tasks. As shown in [56] another limitation of code2vec is its sensitivity to naming. For tasks like those described in code2vec, where names of methods are predicted, names are of course essential, but for the extraction of abstract concepts, the uncertainty of the correct name is too high.

Yarahmadi et al. [55] have conducted an extensive and systematic literature review on how design patterns can be detected in code and therefore abstract the code to perform this task. The main findings of this study relevant to this paper are: Many of the approaches have been tested and evaluated only on small data sets or on limited code samples. The principle in almost all approaches that were reviewed is to reduce the search space by abstraction. Most approaches were limited in their ability to recognize different types of patterns. Another problem of many approaches is detecting different variants of a pattern. To make this possible, ML methods are often used. However, these methods require good data preprocessing because it is not possible to decide in a general way which parts should be selected for learning. A common approach to this problem is, as implemented in [57], a semi-automatic approach in which a human takes over feedback or labeling.

Another principle often used in addition to using the syntactic concepts of programming languages is to analyze the identifiers (e.g., classes, methods, or variable names) using natural language processing techniques [58] [59]. Schindler et al. [59] demonstrated that these methods are well suited for project-specific domain models but not for identifying general

2023, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

Definition 10 (Minimum Example $\hat{r}$ (formal)): Let C be a concept and r an example in which it is applied, then this is called minimal, written $\hat{r}$,iff. $\hat{r}$ is a minimal subgraph of r satisfying the following conditions. (i)

1) $g \setminus g' = \emptyset$ with $g' := \bigcap_{h_i \subset g}$ with $H := \{h_1, \ldots, h_n\}$ the set of all roles of C
2) $\forall v \in V_{\hat{r}} \mid V_{\hat{r}} \setminus \{v\} \Rightarrow \hat{r} \notin R_c$
3) $\forall e \in E_{\hat{r}} \mid E_{\hat{r}} \setminus \{e\} \Rightarrow \hat{r} \notin R_c$
4) $\forall v \in V_{\hat{r}},\ \forall m \in \mathbb{N} \mid 1 \leq m \leq\mid \mathfrak{P}_c \mid \wedge\ f_v(v) - (\vec{\mathfrak{p}}_{i,j})_m \Rightarrow \hat{r} \notin R_c$
5) $\forall e \in E_{\hat{r}},\ \forall m \in \mathbb{N} \mid 1 \leq m \leq\mid \mathfrak{P}_c \mid \wedge\ f_e(e) - (\vec{\mathfrak{p}}_{i,j})_m \Rightarrow \hat{r} \notin R_c$

Definition 12 (Architecture Concept $\hat{C}$ (minimal)):
Let CONCEPT be the universe of all known concepts. By Definition 5, an architecture concept C is a named set of examples. If all examples of this set are minimal, then this is denoted by the notation $\hat{C}$ and is defined as:

$$
\begin{aligned}
\hat{C} &:= (id, \hat{R}) \\
&\quad id \text{ a finite string for which an injective mapping } f \text{ exists,} \\
&\quad\quad \text{with } id \in \text{CONCEPT and } f : \text{CONCEPT} \rightarrow \mathbb{N} \\
&\quad\quad \text{it holds } \forall C_i, C_j, f(C_i) = f(C_j) \Rightarrow C_i = C_j \\
&\quad \hat{R} \text{ a finite indexed set of semantically identical minimal examples with respect to the concept } C. \\
&\quad\quad \hat{R} := \{\hat{r}_i \mid \forall \hat{r}_j \in \hat{R} \text{ gilt } \hat{r}_i \overset{C}{\approx} \hat{r}_j \wedge \hat{r}_i \text{ is minimal}\}
\end{aligned}
\tag{18}
$$

patterns. Natural language identifiers can be an indication but not a robust criterion. An example of how the AST is able to be enriched by additional features, e.g., by using ML, is described in [60] and [61].

In addition, tools and frameworks should also be mentioned, which could also be applied, though in part with restrictions. For example, jQAssistent [62] is a tool that transfers the AST into a Neo4j graph database, offers the possibility of manually enriching this graph with further information, and then using the query-language Cypher to define concepts and identify them in the graph. In contrast to the approach presented in this paper, a query needs to be formulated covering the concept for which the sample should be retrieved.

ArchUnit [63], Structure101 [64], and Dependometer [65] are based on the same principle of formulating rules that are checked automatically afterward. However, the creation and management of rules is costly with the increasing complexity of the concept, and require substantial expert knowledge. All of the mentioned approaches do not assist in expressing rules applying to a given set of samples.

The major problem in this kind of approach and any other approach based on a specific formal language is that it is difficult to define the concrete rules describing a pattern correctly. Rasool et al. [66] describe it as a lack of standard specification for design patterns.

The field of code clone detection is related to the approach presented in this paper since the input data is identical. In [67], four types of code clone detection are characterized, (i) syntactically identical code fragments, (ii) syntactically identical except names and literal values, (iii) syntactically similar fragments that differ in some statements but can be transformed to each other by simple operations and (iv) syntactically dissimilar code fragments but sharing the same functionality. In contrast to code clone detection, we do neither want to find syntactically identical fragments (i)-(iii) nor functionally identical ones (iv). Because of the domain-specific adaptation, we are not interested in finding direct copies.

VII. Challenges of Extracting Architectural Concepts

In Section V, a formal description of general concepts was derived based on set theory, graph theory, and linguistics. This description can only be understood as part of a higher-level methodical approach, described in, for example, in [61] [68] [69]. In this context, we are confronted with practical application challenges which arise in the exchange between programmer and architect. The cause of this tension is based on the different perspectives on the system, i.e., the architecture description on the one hand and the implementation on the other hand. The following four aspects are examples of this: (I.) *How can a **continuous mapping** between an architectural concept as part of an architectural description and its implementation be created?* (II.) *How to ensure **semantic correctness** between concepts in the architectural description and concepts in the implementation?* (III.) *How can the **completeness** of the concepts in the architecture documentation be ensured with respect to the concepts implemented in the implementation?* (IV.) *How can the maximum possible conceptual knowledge be extracted from **minimal data**?*

A. Continuous Mapping

Due to the selection of the considered challenges, marking arbitrary concepts in the ASGs is necessary. The highlighting must be possible so that different instances of the same concept and parts of the concept that do not manifest themselves according to the locality principle can be mapped, and the compositionality of concepts is considered.

A metaphor introduced for this purpose is the concept of **Color**. In graph theory, the concept of coloration is used for various questions [70]. Nevertheless, Color is also suitable as a metaphor independently of any mathematical structure since aspects such as nuances in the form of brightness and saturation, or even the creation of new colors by mixing primary colors, for example, can be easily imagined. Moreover, it can be applied to the composition of concepts. Color highlighting is also a very well-understood concept in different contexts.

2023, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

Definition 13 (Color $\mathfrak{p}$): Let COLOR *be the universe of all known colors. A color $\mathfrak{p} \in$ COLOR *is a label which a node or an edge of a graph may possess. A color is uniquely identified by its name, that is, there exists an injective mapping f :* COLOR $\to \mathbb{N}$*, where holds.*

$$\forall \mathfrak{p}_1, \mathfrak{p}_2 \in \text{COLOR} \mid f(\mathfrak{p}_1) = f(\mathfrak{p}_2) \Rightarrow \mathfrak{p}_1 = \mathfrak{p}_2 \quad (21)$$

If colors are to be used to mark both atomic and complex concepts, then it is easy to see that not only nodes and edges must be marked, but also entire subgraphs must be able to be assigned a color. In order to allow multiple applications of a concept as well, a mechanism of instantiation of colors concerning a concrete graph must be introduced. A corresponding indexing of colors in the following implements this.

Let g be a graph and $\mathfrak{P} \subseteq$ COLOR where $\mathfrak{P} := \{\mathfrak{p}_1, \mathfrak{p}_2, \mathfrak{p}_3\}$ is the finite set of colors to be used for marking in the graph and

$$\mathfrak{P}_g := \mathfrak{p}_{1,1}, \mathfrak{p}_{1,2}, \mathfrak{p}_{1,3}, \mathfrak{p}_{2,1}\mathfrak{p}_{2,2}, \mathfrak{p}_{3,1}\}$$

the concrete instances of the colors.

Conclusion 4 (System Boundary of Instances): The graph is a system boundary with regard to the instances of a color. Given two color instances $\mathfrak{p}_{i,n} \in \mathfrak{P}_g$ and $\mathfrak{p}_{j,m} \in \mathfrak{P}_h$, then holds: (i)

1) $\mathfrak{p}_i, \mathfrak{p}_j \in \text{COLOR} \mid \mathfrak{p}_i = \mathfrak{p}_j \Rightarrow i = j$
2) $\mathfrak{p}_{i,n} \in \mathfrak{P}_g,\ \mathfrak{p}_{j,m} \in \mathfrak{P}_h \mid i = j \wedge n = m \not\Rightarrow \mathfrak{p}_{i,n} = \mathfrak{p}_{j,m}$

As a result of the described gap between implementation and architecture, there is no direct injective mapping between these two artifacts. The implementation always represents a contextual application of architectural concepts. This makes setting up this mapping in particular difficult. Related to the coloring of the abstract syntax graph, continuous mapping exists precisely when all nodes and edges belonging to an instance of a concept are marked. Of course, this also assumes that if the example is according to Definition 5, all aspects of the concept have already been fully implemented in the implementation. Otherwise, it is possible to speak of a marking of the concept but not of an example.

The labeling principle is formulated in such a general way that even new concepts can be mapped intuitively in the sense of previously unknown. According to the definition of an architectural concept (Definition 5) and that of a color (Definition 13), a bijective mapping can be formulated between concepts and colors. In the form that the concept which is mapped to $n \in \mathbb{N}$ is represented by the color, which is also mapped to n. Defining a new concept is equivalent to creating a new color. However, what does it mean that a new color/concept is created? – Two cases must be distinguished: (i.) introduction of a new atomic concept and (ii.) introduction of a new composed concept.

The universe of colors is extended in both cases. One difference is that in case (i.) an extension of the underlying programming language takes place and is a cross-system concern. This is not true for case (ii.). The trigger for case (ii.) consists of the fact that a concept is to be selected, which is not yet named in the form. Accordingly, there is no color and no example yet for this concept. This labeling of a concept instance has the character of an annotation and is initially only valid for the system under consideration. Thus the ASG is extended by information that cannot be derived directly from the program code. For this reason, the so-called **Concept-Graph** is introduced for a conceptual separation. We define that an abstract syntax graph (Definition 6), which is colored accordingly, is called **Concept-Graph**. This graph may furthermore be extended by so-called **Concept-Nodes** $V_{Concept}$ / **Concept-Edges** $E_{Concept}$. A concept graph is thus defined as follows.

Definition 14 (Concept-Graph): Let $g_{ASG}(V, E)$ be an abstract syntax graph of a program with node set $V = V_{Syn}$ of AST and edge set $E := E_{Syn} \cup E_{Sem}$ of ASG). A colored directed Concept-Graph $g \in$ GRAPH is represented by the following signature: $g(g_{ASG}, \mathfrak{P}_{\text{ATOM}}, V_{Concept}, E_{Concept}, \mathfrak{P}_{Concept}, f_V, f_E)$

$$\begin{aligned}
&V \text{ a finite indexed set of nodes.}\\
&\quad V = V_{ASG} \cup V_{Concept}, V := \{v_1, v_2, \ldots, v_n\}\\
&E \text{ a finite indexed set of directed edges}\\
&\quad E \subseteq V \times V,\ E := \{e_1, e_2, \ldots, e_m\} \text{ mit } e_i = (v_j, v_k)\\
&\quad E = E_{ASG} \cup E_{Concept}\\
&\mathfrak{P} \text{ a finite indexed set of color instances}\\
&\quad \mathfrak{P} = \mathfrak{P}_{\text{ATOM}} \cup \mathfrak{P}_{Concept}\\
&\quad \mathfrak{P} := \{\mathfrak{p}_{1,1}, \mathfrak{p}_{1,2}, \ldots, \mathfrak{p}_{1,i}, \ldots, \mathfrak{p}_{n,1}, \ldots, \mathfrak{p}_{n,j}\}\\
&\quad \text{mit } \{\mathfrak{p}_1, \mathfrak{p}_2, \ldots, \mathfrak{p}_n\} \subseteq \text{COLOR}\\
&f_V \text{ Coloring function for nodes } f_V : V \to \mathbb{Z}_2^{|\mathfrak{P}|}\\
&f_E \text{ Coloring function for edges } f_E : V \to \mathbb{Z}_2^{|\mathfrak{P}|}
\end{aligned} \quad (22)$$

Even if the node types are represented by colors concerning the programming language's grammar, in general, we should no longer speak of typed nodes and edges, only of colored ones, because it is now possible to assign several colors to each node or edge. However, this is contrary to the definition of the type. On this basis, now arbitrary architecture concepts can be represented, and further operations on these can be defined.

B. Semantic Correctness

Suppose a named set of examples exists, validated in that the contained examples are semantically equivalent. In that case, it holds that the given examples are context-independent, as well as an interpretation of the name in the form of these examples is possible. Nevertheless, no complete description of the concept exists in the form of a specification. Furthermore, the examples are semantically equivalent but structurally different. This is caused by the embedding in an application context. In the Limitations (see Section IV-C), the maximum common subgraph problem has already been pointed out. One is confronted with the same NP-hard problem if we want to answer the question of whether a graph g should be included

2023, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

as an example in the set of examples of a concrete concept $C(id, R)$?

$$g \in R \text{ or } g \notin R$$

We can define the concept of a **Dector** as follows:

Definition 15 (Detector d_C): A detector $d_C \in$ DETECTOR classifies a graph g if the graph is an example of a given concept $C(id, R)$ or not.

$$d_C : g \rightarrow [\text{true}, \ \text{false}]$$

$$d_{c_i}(G) := \begin{cases} \text{true}, g \in R \\ \text{false}, g \notin R \end{cases}$$

The goal is to develop a methodology to identify known concepts (identified by identifiers and described by examples) in a given program using a heuristic to derive suggestions for new concepts from the extracted information. Since this is an NP-hard or partly an NP-complete problem, it is reasonable to modify the concept of semantic correctness so that we can refer to a semantic similarity instead. The consequence of this is that there must be an expert who takes over the validation related to the correctness. However, this can also be an advantage, as it makes it possible to create specific concept variants, including project or domain-specific implementations of a concept.

A corresponding adaptation can be achieved by fuzzifying the edges to hyperedges and the coloring functions of the concept graph. If these are fuzzified, then the binary vector becomes real in the interval 0 to 1. In this case, it is a so-called **Fuzzy-Hypergraph** as described in Definition 16 and would have to be considered accordingly in the example representation.

The idea behind the Fuzzy-Hypergraph is that nodes can be grouped with a given membership function σ, μ, and a given degree α. Each hyperedge here represents exactly one aspect or parts of a hierarchical aspect. This allows us to express for a concrete instance which influences specific nodes have in this context. Fuzzy-Hypergraphs are suitable for partitioning tasks and pattern recognition, among other things. Since this is an analytical approach, the quality of the results is strongly dependent on the modeling of the membership functions.

C. Complettness of Extraction

The documentation of a system's architecture is always an incomplete system description. This is, on the one hand, because only those aspects are listed which have relevance from the point of view of the architect and, on the other hand, due to the fact that programmers establish concepts during their development but do not communicate this. The completeness concept is dependent on the extraction of subjective feeling. We can say, therefore: The list of the implemented concepts is complete if the architect sees all the concepts known to him or concepts, which are already present in the knowledge base, are not proposed anymore and validated as positive. If other concepts are present in the implementation, and they will be, then they are not significant at the current time or dedicated as such.

The more interesting case, however, is the one where the programmer implements concepts that are not known to the architect. A case in extracting new concepts is when the previous knowledge consists only of atomic concepts. Here, no other coloring can be performed. Finding new concepts (colors) for nodes and edges can be traced to a clustering problem. Outlier detection, i.e., frequencies, outliers, or neighboring / overlapping graphs, must be searched for. Here, methods can be used as described in Section III-A. Given the formalization, it can be assumed that different methods must be combined for different aspects.

Having a procedure that is open to new concepts means that an integration mechanism must exist to incorporate them into a knowledge base. For example, higher-level concepts are formed from the programming language's atomic language concepts (colors) and their relationships to each other. In other words, relationships exist between the colors, and precisely these relationships must now be extracted. Thus also, the integration mechanism can be specified in the introduction of new colors as well as the way of assigning colors to nodes, edges, and graphs. This poses particular challenges to the methodology, not only for the one-time extraction but also for the evolution of the system under investigation over time.

This question highlights once again that the architecture of a system cannot be considered invariant, even if an architecture should rather be stable over the system's life cycle and represent a default for the implementation [72]. During implementation, situations may arise that require a change of concept. But such an architectural decision must be made consciously for all and ideally documented and justified.

D. Data Challenge

The more complex a concept grows, the fewer examples exist for this concept; as an example for the evaluation of the presented method (see Section III-A and IV), the Singelton pattern was used. With this pattern, it was possible to build up a corresponding dataset. However, this had to be validated manually because in the result set, despite the name Singelton pattern, there were examples that were not semantically equivalent to the chosen specification. The reason for this is explained in detail in Section V-B. Therefore, expert checking should always be considered.

Considering patterns that are more system-wide concepts, such as Pipes-And-Filters [73] or a Layered architecture, it is common for this concept to manifest itself only once in the system. This means that hundreds of systems with this corresponding architecture are needed for a similar number of examples as the extraction was performed for the Singelton pattern. On the other hand, the concept behind layered architecture, for example, can be described in a fraction of the size of the examples.

As a result, methods must be constructed that can work with a small amount of data, address the compositional aspect of a concept, and construct results that humans can easily

2023, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

Definition 16 (Fuzzy-Hypergraph $\tilde{\tilde{h}}\ (V, \mathcal{E}, \alpha, \sigma, \mu)$, see [71]): A Fuzzy-Hypergraph $\tilde{\tilde{h}}$ is represented by the following signature:

$$\begin{aligned}
&\tilde{\tilde{h}}\ (V, \mathcal{E}, \alpha, \sigma, \mu) \\
&\qquad V \text{ a finite indexed set of nodes, } V := \{v_1, v_2, \ldots, v_n\} \\
&\qquad \mathcal{E} \text{ a finite indexed set of Hyper-Edges. } \mathcal{E} \subset \mathcal{P}(V) \setminus \emptyset \\
&\qquad \mathcal{E} := \{\tilde{E}_1, \tilde{E}_2, \ldots, \tilde{E}_n\} \\
&\qquad \sigma \text{ a Fuzzy-Set } V_\alpha,\ \sigma : V \to [0, 1] \\
&\qquad V_\alpha := \{v_i \in V \mid \sigma(v_i) \geq \alpha \wedge \alpha \neq 0\} \\
&\qquad \mu \text{ a set of membership functions } \mu := \{\mu_1, \mu_2, \ldots, \mu_{|\mathcal{E}|}\} \\
&\qquad \tilde{E}_i = \{v_j \in V \mid (v_j, \mu_i(v_j)) \geq \alpha \wedge \alpha \neq 0\} \\
&\qquad \bigcup_{i=1}^{|\mathcal{E}|} supp(\tilde{E}_i) = V
\end{aligned} \tag{23}$$

Where $supp(A)$ is the support of the set and is defined as:

$$supp(A) := \{x \in A \mid \mu_A(x) \geq 0\} \tag{24}$$

interpret. For example, the approach described in this article. Moreover, a possible approach could be to investigate logic-based representations as described in Herold [74] and Deiters [21] [22]. In this way, an approach could be developed based on extensional and intensional concept descriptions.

VIII. Conclusion and Future Work

Starting from the approach, which was introduced in [1], an extension was made in this article. We derived a comprehensive theory and formalism, which makes it possible to establish holistic approaches as they are described in Herold et al. [75], Knieke et al. [68] and Schindler [61]. All of these approaches try to mitigate architecture degradation using ML. Focusing on extracting concepts from existing implementations, most common approaches like code2vec, for example, rely on large amounts of data, so they are unsuitable for this kind of problem. Because, on the one hand, these data have to be acquired and validated, and on the other hand, the results have to be interpretable by humans.

We have shown an approach to extract the essence of a shared concept driven by available implementations so that the formulation is interpretable by humans. Moreover, we formulate expressions that explicitly be not part of an implementation of the concept. In other words, if such a path were added to any concept example, it would destroy it.

Future work planned includes two directions, on the one hand addressing the way the semantics of a concept is described and, on the other hand, using the introduced representation and abstraction technique as a preprocessing step in the direction of ML techniques. For example, to train a classifier or cluster samples to identify variants or the inner parts of a pattern, e.g., roles. Including the addressed limitations and collecting a high quality and high quantity data set of different design patterns, including different variants of a pattern. We choose an extensional description for the semantics. Experiments have shown that if a set of examples of a concept is known and validated, describing them intensionally using predicate logic formulas could be possible. The question of describing and interpreting a composition operator for architectural concepts can be seen as essential and still open at the current research stage.

References

[1] C. Schindler, M. Schindler, and A. Rausch, "Negligible details - towards abstracting source code to distill the essence of concepts," in *ADAPTIVE 2022*, M. Kurz, Ed. Wilmington, DE, USA: IARIA, 2022, pp. 22–31. [Online]. Available: https://www.thinkmind.org/index.php?view=article&articleid=adaptive_2022_2_20_50009

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: Elements of reusable object-oriented software*, 2nd ed., ser. Addison-Wesley professional computing series. Boston: Addison-Wesley, 1997.

[3] J. Coplien, *Software Patterns*. SIGS Books & Multimedia, 1996.

[4] K. Bergner, A. Rausch, and M. Sihling, *Using UML for Modeling a Distributed Java Application*. TUM, 1997. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.42.6797

[5] G. Sunyé, A. Le Guennec, and J.-M. Jézéquel, "Design patterns application in uml," in *European Conference on Object-Oriented Programming*, 2000, pp. 44–62.

[6] S. Hussain, J. Keung, and A. A. Khan, "The effect of gang-of-four design patterns usage on design quality attributes," in *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 2017, pp. 263–273.

[7] C. Deiters and A. Rausch, "Assuring architectural properties during compositional architecture design," in *International Conference on Software Composition*. Springer, 2011, pp. 141–148.

[8] M. Paixao, J. Krinke, D. Han, C. Ragkhitwetsagul, and M. Harman, "Are developers aware of the architectural impact of their changes?" in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2017, pp. 95–105.

[9] M. Schindler and S. Lawrenz, "Community-driven design in software engineering," in *Proceedings of the 19th International Conference on Software Engineering Research & Practice, Las Vegas, NV, USA*, 2021.

[10] M. L. Scott, *Programming language pragmatics*, 4th ed. Amsterdam and Boston and Heidelberg and London and New York and Oxford and Paris and San Diego and San Francisco and Singapore and Sydney and Tokyo: Morgan Kaufmann/Elsevier, 2016.

[11] N. Chomsky and D. Lightfoot, *Syntactic structures*, 2nd ed., ser. A Mouton classic. Berlin: Mouton de Gruyter, 2002.

[12] N. Chomsky, "Three models for the description of language," *IEEE Transactions on Information Theory*, vol. 2, no. 3, pp. 113–124, 1956.

[13] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, techniques, & tools*, 2nd ed. Boston: Pearson Addison Wesley, 2007.

[14] J. Niere, J. P. Wadsack, and L. Wendehals, "Handling large search space in pattern-based reverse engineering," in *11th IEEE International Workshop on Program Comprehension, 2003.* IEEE, 2003, pp. 274–279.

[15] A. Rajaraman and J. D. Ullman, "Data mining," in *Mining of Massive Datasets*, A. Rajaraman and J. D. Ullman, Eds. Cambridge: Cambridge University Press, 2011, pp. 1–17.

2023, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

[16] P-mart pattern-like micro-architecture repository. [retrieved: 03, 2023]. [Online]. Available: https://www.ptidej.net/tools/designpatterns/index_html
[17] M. R. Garey and D. S. Johnson, *Computers and intractability: A guide to the theory of NP-completeness*, ser. A series of books in the mathematical sciences. New York u.a: Freeman, 1979.
[18] V. Kann, "On the approximability of the maximum common subgraph problem," in *Annual Symposium on Theoretical Aspects of Computer Science*. Springer, 1992, pp. 375–388.
[19] "Concept," 2022, [retrieved: 03, 2023]. [Online]. Available: https://www.oxfordlearnersdictionaries.com/definition/english/concept?q=concept
[20] E. Zermelo, "Über grenzzahlen und mengenbereiche," *Fundamenta Mathematicae*, vol. 16, no. 1, pp. 29–47, 1930. [Online]. Available: https://eudml.org/doc/212506
[21] C. Deiters and A. Rausch, "Assuring architectural properties during compositional architecture design," in *Software composition*, ser. Lecture Notes in Computer Science / Programming and Software Engineering, S. Apel, Ed. Berlin and Heidelberg: Springer, 2011, vol. 6708, pp. 141–148.
[22] C. Deiters, *Beschreibung und konsistente Komposition von Bausteinen für den Architekturentwurf von Softwaresystemen*, 1st ed., ser. SSE-Dissertation. München: Dr. Hut, 2015, vol. 11.
[23] H.-D. Ebbinghaus, *Einführung in die Mengenlehre*, 5th ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2021. [Online]. Available: http://nbn-resolving.org/urn:nbn:de:bsz:31-epflicht-1878742
[24] B. Russell, *The philosophy of logical atomism*, ser. Routledge Classics. Abingdon, Oxon: Routledge, 2009. [Online]. Available: http://site.ebrary.com/lib/alltitles/docDetail.action?docID=10330922
[25] D. Gutzmann, *Semantik: Eine Einführung*, ser. Lehrbuch J.B. Metzler. Berlin and Heidelberg: J.B. Metzler Verlag, 2020.
[26] W. Hodges, "Formalizing the relationship between meaning and syntax," in *The Oxford handbook of compositionality*, ser. Oxford handbooks in linguistics, M. Werning, W. Hinzen, and E. Machery, Eds. Oxford: Oxford Univ. Press, 2012.
[27] D. Hillert, Ed., *Die Natur der Sprache: Evolution, Paradigmen und Schaltkreise*. Wiesbaden: Springer, 2017.
[28] J. A. Hampton and Y. Winter, Eds., *Compositionality and Concepts in Linguistics and Psychology*, ser. Language, Cognition, and Mind. Cham: Springer International Publishing, 2017, vol. 3.
[29] T. E. Zimmermann, "Compositionality problems and how to solve them," in *The Oxford handbook of compositionality*, ser. Oxford handbooks in linguistics, M. Werning, W. Hinzen, and E. Machery, Eds. Oxford: Oxford Univ. Press, 2012.
[30] B. H. Partee, A. T. Meulen, and R. E. Wall, *Mathematical Methods in Linguistics*, 1st ed., ser. Studies in Linguistics and Philosophy Ser. Dordrecht: Springer Netherlands, 1990, vol. v.30.
[31] M. Kracht, "Compositionality: The very idea," *Research on Language and Computation*, vol. 5, no. 3, pp. 287–308, 2007.
[32] M. Fowler, "Dealing with roles."
[33] B. L. Whorf and P. Krausser, Eds., *Sprache, Denken, Wirklichkeit: Beiträge zur Metalinguistik und Sprachphilosophie*, 25th ed., ser. Rowohlts Enzyklopädie. Reinbek bei Hamburg: Rowohlt, 2008, vol. 55403.
[34] L. Boroditsky, "Linguistic relativity," in *Encyclopedia of cognitive science*, L. Nadel, Ed. Chichester, West Sussex Eng. and Hoboken, N.J: Wiley, 2005.
[35] ——, "How language shapes thought," *Scientific American*, vol. 304, no. 2, pp. 62–65, 2011.
[36] R. K. Hendricks and L. Boroditsky, "New space-time metaphors foster new nonlinguistic representations," *Topics in Cognitive Science*, vol. 9, no. 3, pp. 800–818, 2017. [Online]. Available: https://onlinelibrary.wiley.com/doi/full/10.1111/tops.12279
[37] O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, *Structured programming*, 11th ed., ser. APIC studies in data processing. London: Academic Press, 1972, vol. 8.
[38] M. Kracht, "The emergence of syntactic structure," *Linguistics and Philosophy*, vol. 30, no. 1, pp. 47–95, 2007.
[39] ——, *The Mathematics of Language*. UCLA, 2003. [Online]. Available: https://linguistics.ucla.edu/people/Kracht/courses/compling2-2007/formal.pdf
[40] Cantor, "Ueber unendliche, lineare punktmannichfaltigkeiten. 5. fortsetzung: Fortsetzung des artikels in bd. xxi, pag 51." *Mathematische Annalen*, vol. 21, pp. 545–591, 1883. [Online]. Available: https://eudml.org/doc/157080
[41] M. Kracht, "Using each other's words," in *The Road to Universal Logic*. Birkhäuser, Cham, 2015, pp. 341–349. [Online]. Available: https://rd.springer.com/chapter/10.1007/978-3-319-10193-4_15
[42] D. Hilbert and W. Ackermann, *Grundzüge der Theoretischen Logik*, 6th ed., ser. Grundlehren der Mathematischen Wissenschaften Ser. Berlin, Heidelberg: Springer Berlin / Heidelberg, 1972, vol. v.27.
[43] T. Berners-Lee and M. Fischetti, *Weaving the Web: The original design and ultimate destiny of the World Wide Web by its inventor*, 1st ed. San Francisco, Calif.: HarperCollins, 2000.
[44] R. Koschke, J.-F. Girard, and M. Wurthner, "An intermediate representation for integrating reverse engineering analyses," in *Proceedings / Fifth Working Conference on Reverse Engineering*. Los Alamitos, Calif.: IEEE Computer Society Press, 1998, pp. 241–250.
[45] H. P. Barendregt, M. C. J. D. van Eekelen, J. R. W. Glauert, J. R. Kennaway, M. J. Plasmeijer, and M. R. Sleep, "Term graph rewriting," in *PARLE*, ser. Lecture Notes in Computer Science, J. W. de Bakker, A. J. Nijman, P. C. Treleaven, and J. W. de Bakker, Eds. Berlin: Springer, 1987, pp. 141–158.
[46] D. Plump, "Term graph rewriting," in *Applications, languages and tools*, ser. Handbook of graph grammars and computing by graph transformation / /managing ed, H. Ehrig, G. Engels, H.-J. Kreowski, G. Rozenberg, H. Ehrig, and G. Rozenberg, Eds. Singapore: WORLD SCIENTIFIC, 1999, pp. 3–61.
[47] J. Goll, *Architektur- und Entwurfsmuster der Softwaretechnik: Mit lauffähigen Beispielen in Java*, 2nd ed. Wiesbaden: Springer Vieweg, 2014.
[48] D. Bäumer, D. Riehle, W. Siberski, and M. Wulf, "The role object pattern," in *Washington University Dept. of Computer Science*, 1998.
[49] C. Rupp and S. Queins, *UML2 glasklar: Praxiswissen für die UML-Modellierung*, 4th ed. München: Hanser, 2012. [Online]. Available: http://www.hanser-elibrary.com/action/showBook?doi=10.3139/9783446431973
[50] H. Balzert, *Lehrbuch der Objektmodellierung: Analyse und Entwurf ; mit CD-ROM*, ser. Lehrbücher der Informatik. Heidelberg and Berlin: Spektrum Akad. Verl., 1999.
[51] P. Coad, "Object-oriented patterns," *Communications of the ACM*, vol. 35, no. 9, pp. 152–159, 1992.
[52] S. A. Cook, "The complexity of theorem-proving procedures," in *Proceedings of the third annual ACM symposium on Theory of computing - STOC '71*, M. A. Harrison, R. B. Banerji, and J. D. Ullman, Eds. New York, New York, USA: ACM Press, 1971, pp. 151–158.
[53] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "A general path-based representation for predicting program properties," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 404–419.
[54] ——, "code2vec: Learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–29, 2019.
[55] H. Yarahmadi and S. M. H. Hasheminejad, "Design pattern detection approaches: a systematic review of the literature," *Artificial Intelligence Review*, vol. 53, no. 8, pp. 5789–5846, 2020.
[56] R. Compton, E. Frank, P. Patros, and A. Koay, "Embedding java classes with code2vec: Improvements from variable obfuscation," in *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020, pp. 243–253.
[57] G. Rasool, I. Philippow, and P. Mäder, "Design pattern recovery based on annotations," *Advances in Engineering Software*, vol. 41, no. 4, pp. 519–526, 2010.
[58] P. Warintarawej, M. Huchard, M. Lafourcade, A. Laurent, and P. Pompidor, "Software understanding: Automatic classification of software identifiers," *Intelligent Data Analysis*, vol. 19, no. 4, pp. 761–778, 2015.
[59] M. Schindler, A. Rausch, and O. Fox, "Clustering source code elements by semantic similarity using wikipedia," in *Proceedings of 4th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE)*, 2015, pp. 13–18.
[60] J. He, C.-C. Lee, V. Raychev, and M. Vechev, "Learning to find naming issues with big code and small supervision," in *2021 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21)*. ACM, 2021, pp. 1–16.
[61] M. Schindler and A. Rausch, "Architectural concepts and their evolution made explicit by examples," in *ADAPTIVE 2019, The Eleventh International Conference on Adaptive and Self-Adaptive Systems and Applications*, vol. 11, 2019, pp. 38–43.

2023, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

[62] jqassistant — your software . your structures . your rules. [retrieved: 03, 2023]. [Online]. Available: https://jqassistant.org
[63] Unit test your java architecture - archunit. [retrieved: 03, 2023]. [Online]. Available: https://www.archunit.org
[64] Structure101 software architecture development environment (ade). [retrieved: 03, 2023]. [Online]. Available: https://structure101.com
[65] Dependometer. [retrieved: 03, 2023]. [Online]. Available: https://github.com/dheraclio/dependometer
[66] G. Rasool and D. Streitfdert, "A survey on design pattern recovery techniques," *International Journal of Computer Science Issues (IJCSI)*, vol. 8, no. 6, p. 251, 2011.
[67] W. Wang, G. Li, B. Ma, X. Xia, and Z. Jin, "Detecting code clones with graph neural network and flow-augmented abstract syntax tree," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2020, pp. 261–271.
[68] C. Knieke, A. Rausch, and M. Schindler, "Tackling software architecture erosion: Joint architecture and implementation repairing by a knowledge-based approach," in *2021 IEEE/ACM International Workshop on Automated Program Repair (APR)*. IEEE, 6/1/2021 - 6/1/2021, pp. 19–20.
[69] C. Knieke, K. Marco, A. Rausch, M. Schindler, A. Strasser, and M. Vogel, "A holistic approach for managed evolution of automotive software product line architectures," in *ADAPTIVE 2017*, A. A. Enescu and A. Rausch, Eds. Wilmington, DE, USA: IARIA, 2017, pp. 43–52.
[70] B. Bollobás, *Modern Graph Theory*. New York, NY: Springer New York, 1998, vol. 184.
[71] H. Lee-Kwang and K.-M. Lee, "Fuzzy hypergraph and fuzzy partition," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 25, no. 1, pp. 196–201, 1995.
[72] L. Bass, P. Clements, and R. Kazman, *Software architecture in practice*, 3rd ed., ser. Safari Tech Books Online. Upper Saddle River, NJ: Addison-Wesley, 2013.
[73] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects*, 1st ed., ser. Wiley Software Patterns Series. s.l.: Wiley, 2013. [Online]. Available: http://gbv.eblib.com/patron/FullRecord.aspx?p=699910
[74] S. Herold, *Architectural compliance in component-based systems: Foundations, specification, and checking of architectural rules*, 1st ed., ser. SSE-Dissertation. München: Verl. Dr. Hut, 2011, vol. 5.
[75] S. Herold, C. Knieke, M. Schindler, and A. Rausch, "Towards improving software architecture degradation mitigation by machine learning," in *ADAPTIVE 2020, The Twelfth International Conference on Adaptive and Self-Adaptive Systems and Applications*, 2020, pp. 36–39.

2023, © Copyright by authors, Published under agreement with IARIA - www.iaria.org