Negligible Details - Towards Abstracting Source Code to Distill the Essence of Concepts

Christian Schindler*, Mirco Schindler† and Andreas Rausch‡
Institute for Software and Systems Engineering
Clausthal University of Technology
Clausthal, Germany
* Email: christian.schindler@tu-clausthal.de
† Email: mirco.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. To guarantee the problem-solving capabilities of patterns, a correct implementation of the design pattern is essential. As a context-specific adoption of the design pattern to the software system needs to be performed by the developers, we argue that their comprehension plays a crucial role in the creation and maintenance of 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. The primary intent of this paper is to separate essential syntactic information from varying aspects, given a set of implementation samples. We introduce an approach that abstracts given object-oriented implementations by semantically resolving and splitting an Abstract Syntax Tree into small paths. In analyzing paths from given samples we build a shared concept. In this paper, we build the shared concept from 230 example implementations containing the singleton design pattern and 230 counterexamples to classify new unseen java classes. The contribution this paper provides is composed of three parts. (i) A novel approach to abstract object-oriented code, (ii) an interpretable way to identify common parts extracted from multiple abstractions, and (iii) a way to classify unseen samples to implement the same concept.**

***Index Terms*—Software Abstraction, Object Oriented Language, Design Pattern, Source Code Comprehension, Software Architecture**

I. Introduction

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 [1], [2].

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 [3], [4].

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 [5]. 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 [6], [7]. 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 [8].

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?

Section 2 gives foundations on programming languages and the construction of the Abstract Syntax Trees (ASTs). Section 3 introduces the source code abstraction approach alongside two different levels of abstraction. Section 4 is the evaluation of the stated RQs with a discussion of the results

 ISBN: 978-1-61208-951-5

and limitations. Section 5 presents an overview of related work. Finally, the conclusion and an outline of future work are given in Section 6.

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 [9].

These concepts, defined by the programming language, are called **atomic concepts** 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 [10]. Thus, the words are well-formed concerning a fixed and finite set of formal production rules [11]. Moreover, the lexical grammar of a programming language is usually context-free [12].

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: $N \rightarrow (\Sigma \cup N)^*$
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 III-D) with a dedicated root node and containing no cycles. Finally, each recognized token is converted to a node in the tree. Than, a semantic analysis is performed since not all rules, especially context-dependent ones, can be checked during derivation. This step also resolves the types, 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 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

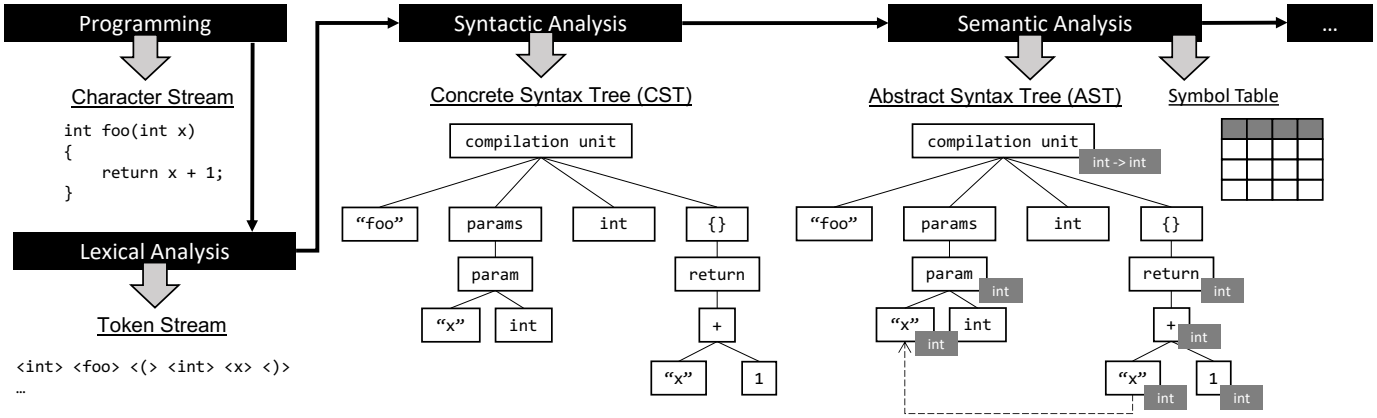


Fig. 1: First steps of a compilation process [12]

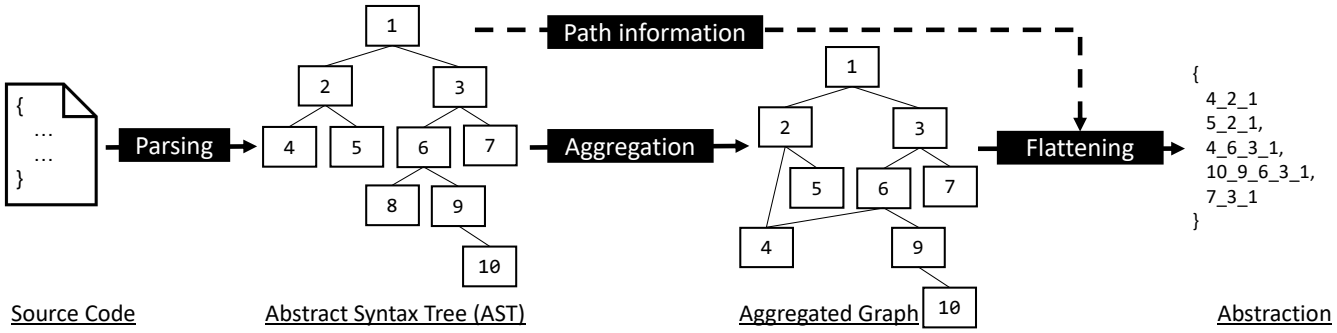


Fig. 2: Overall approach of the source code abstraction

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 analyse 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}

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

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

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.

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

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

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

```
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*

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:

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

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 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.

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*).

 ISBN: 978-1-61208-951-5

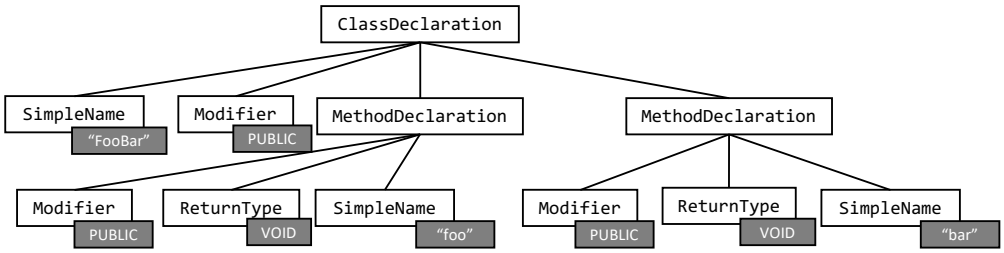


Fig. 4: AST of program 1

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

In Figure 7, all the paths are in the context of to 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.

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:

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

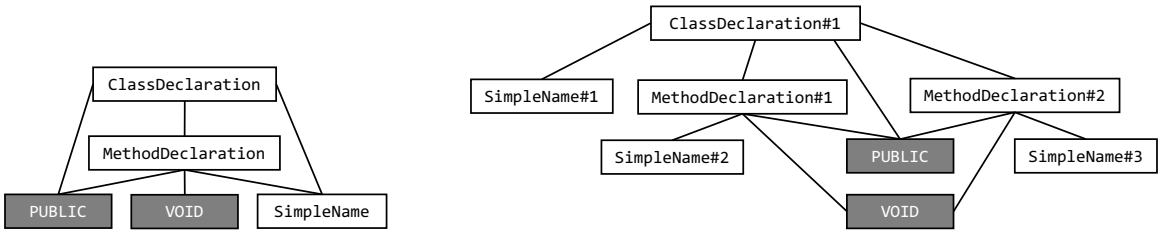
Finding such a *bijection* (candidate) of a subgraph consists of two steps, (i) fixing a suitable subgraph and the (ii) one-to-one correspondence. The verification of such a candidate can be done with the Formula 9. The graphs q and m are converted to adjacency matrices (see Formula 10) and the *bijection* 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:

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



(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

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:

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

with $P_{in} \cap P_{ex} = \emptyset$

$$\forall p_{in} \in P_{in} \wedge \forall a_{in} \in A_{in} \mid p_{in} \in a_{in}$$
$$\forall p_{ex} \in P_{ex} \exists a_{ex} \in A_{ex} \mid p_{ex} \in a_{ex}$$
$$\forall p_{ex} \in P_{ex} \nexists a_{in} \in A_{in} \mid p_{ex} \in a_{in}$$

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_{out} we hypothesize that the shared concept is containing business-domain-independent overlap.

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

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

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.

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 that were 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 [13]. 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 approach of abstraction 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 all abstractions of a given set of abstraction. The sets show that the range on 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 of a sample, indicating that on average there are things in the samples than they can share (as this is what at most can be overlap).

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

The data shown in the table support the assumption that the abstractions embody similarity 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 [14]) 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 a 1996 part 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. 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 [15] 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*.

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 associated indices that conform to the singleton pattern (described in Section 5.2.). Here we then aligned the indices of the samples (using subgraph isomorphism).

As a given sample can be classify 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

 ISBN: 978-1-61208-951-5

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

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

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 for 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 conclusions that at least one sample contains almost the bare minimum needed to implement a singleton in java.

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	$.\overline{6}$	.5	.994	.571
Low	12	13	0	0	1.0	1.0	1.0	1.0

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 runs 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 indicating 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.*). As of two main reasons, (i) the computation needed to find a subgraph isomorphism is NP-complete [16], 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

 ISBN: 978-1-61208-951-5

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.

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.

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 [17](being NP-Hard [16]). We do not have an implementation of this in our prototype.

V. Related Work

A similar approach to the one we propose is code2vec [18], [19], 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, valid for this scope and the defined task. But as shown in [20], the relation between source code elements for higher concepts (like classes) is essential to perform structural or behavioral related tasks. As shown in [21] 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. [20] 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 to detect 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 [22], 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 [23], [24]. In Schindler et al. [24] demonstrated that these methods are well suited for project-specific domain models but not for identifying general patterns. Natural language identifiers can be an indication but not a robust criterion. An example on how the AST is able to be enriched by additional features, e.g., by using ML, is described in [25] and [26].

In addition, tools and frameworks should also be mentioned, which could also be applied, though in part with restrictions. For example, jQAssistent [27] 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 [28], Structure101 [29], and Dependometer [30] 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, requires 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. [31] 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. Wang et al. [32], 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

 ISBN: 978-1-61208-951-5

domain-specific adaptation, we are not interested in finding direct copies.

VI. Conclusion and Future Work

We have shown how to extract the essence of a shared concept, driven by available implementations, so that the formulation is interpretable by humans. Moreover, what we have not found in the literature, is the formulation of what should explicitly not be part of the implementation. Future work planned includes addressing the stated limitations and collecting a high quality and high quantity data set of different design patterns, including also different variants of a pattern.

The abstraction presented in this paper produces a set of paths from a semantically aggregated syntax graph. We plan on utilizing the shown approach as a preprocessing step in the direction of ML techniques. For example, to train classifier or cluster samples to identify variants or the inner parts of a pattern, e.g., roles.

In Herold et al. [33] and Knieke et al. [34], a holistic approach is described to mitigate architecture degradation using ML. For such approaches, it is essential to have relevant training data available and to understand which expected patterns are not present in the implementation. This would also be a use case supported by the method presented here.

References

[1] 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.

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

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

[4] 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.

[5] 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.

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

[7] 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.

[8] 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.

[9] 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.

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

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

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

[13] 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.

[14] 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.

[15] P-mart pattern-like micro-architecture repository. [retrieved: 03, 2022]. [Online]. Available: https://www.ptidej.net/tools/designpatterns/index_html

[16] 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.

[17] 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.

[18] 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.

[19] ——, "code2vec: Learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–29, 2019.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] jqassistant — your software . your structures . your rules. [retrieved: 03, 2022]. [Online]. Available: https://jqassistant.org

[28] Unit test your java architecture - archunit. [retrieved: 03, 2022]. [Online]. Available: https://www.archunit.org

[29] Structure101 software architecture development environment (ade). [retrieved: 03, 2022]. [Online]. Available: https://structure101.com

[30] Dependometer. [retrieved: 03, 2022]. [Online]. Available: https://github.com/dheraclio/dependometer

[31] 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.

[32] 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.

[33] 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.

[34] 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.

 ISBN: 978-1-61208-951-5