Journal of Theoretical and Applied Information Technology
31st March 2023. Vol.101. No 6
© 2023 Little Lion Scientific

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

EXPLORING A NOVEL PERSPECTIVE ON DESIGN PATTERN RECOVERY VIA VISUAL SIGNATURES AND CONTINUOUS-TIME SIGNALS

TARIK HOUICHIME[1] , YOUNES EL AMRANI[2]
[1,2]Mohammed V University In Rabat, ENSIAS, Laboratory of Software Project Management, Morocco
E-mail: [1]houichimetarik@yahoo.com, [2]y.elamrani@um5r.ac.ma

ABSTRACT

Design Pattern Recovery is the process of detecting and retrieving pre-existing design patterns inherent in a software application, which entails a comprehensive investigation of the software and its source code, as well as its dependencies. However, this process can be both time-consuming and resource-intensive. Moreover, the automation of this process poses a significant challenge, demanding a profound understanding of the system's design goals and dependencies that are context-based. Furthermore, the absence of standardization, and the potential for ambiguity arising from the multitude of implementations of a specific design pattern further complicates the automation process. In this work, we investigate a new perspective on the problem of Design Pattern Recovery by framing it in terms of visual signatures and continuous-time signals. The resulting visual signatures and signals capture the key features of general Object-Oriented codes and well-defined design pattern micro-architectures in a language-agnostic manner, serving as an intermediary transformation prior to the recovery phase and facilitating the identification of predefined design pattern signatures in the target code. Consequently, a twofold opportunity for the retrieval of potential design information from code is provided. This is manifested in the form of a feature-rich visual signature, which encapsulates the structural, communicational, and behavioral facets of the analyzed source code. The utilization of such visual signatures may serve as a facilitator for the straightforward application of state-of-the-art pattern recognition techniques in automated design pattern identification. Additionally, the features are also expressed as a scale-invariant continuous-time signal, thereby enabling the effective deployment of signal classification techniques for design pattern mining.

Keywords: *Design Pattern Recovery, Pattern Recognition, Visual Signatures, Continuous Time-Signals, Signal Processing.*

1. INTRODUCTION

The domain of software engineering encompasses a wide range of techniques that have been developed to optimize, control, and test various stages of software development, well studied design patterns [1] are an effective tool to build high-quality object-oriented software and producing coherent source codes, These patterns enable developers to create an abstract representation of real-world problems using objects, In addition, developers may use a combination of these patterns to achieve their desired outcome, undergoing multiple development iterations to build all required features, however, as the software development iterations become more complex and rigorous, the final product gains increased robustness, but its source code and architecture may become difficult to comprehend, therefore, without adequate documentation and source code explanations, a software piece built with complex techniques and architectures may become obsolete and unmaintained.

One problem to address in this context is whether it is possible to reverse-engineer a software's source code to recover the design pattern employed in its development, in other words, can we discern the design pattern that the developers had in mind during the development process? Recovering the design pattern underlying a source code is crucial for understanding its architecture and addressing potential issues with it. While manual approaches based on the product's documentation may be effective for this purpose, such an endeavor may be time-consuming and labor-intensive when the source code becomes longer and has multiple dependencies. Moreover, conventional approaches to automating the process may not be sufficient on their own to effectively address the challenges posed by this problem. One of these challenges is the complexity of design patterns themselves, which often involve multiple classes and relationships that can be difficult to discern and comprehend. Moreover, a significant challenge lies in the absence of standardization and uniformity in the definition and representation of the existing design patterns. This leads to disparities in the implementation of similar patterns across different codebases, resulting in difficulties with precisely identifying and extracting them from the code. Additionally, the lack of ground truth data may impede the assessment of the accuracy and dependability of the extracted patterns. Furthermore, the application of machine learning and artificial intelligence techniques to automate the process of design pattern recovery can introduce novel sources of inaccuracies and biases, which can further complicate the process and impact the accuracy of detecting the patterns, particularly if the components of the problem are not properly conditioned beforehand.

To overcome some of these challenges, one of the crucial impediment in the realm of automating the design pattern recovery process, is the need to appropriately frame the problem such that it can be effectively tackled by machine learning algorithms ensuring robustness and scalability. This would allow design pattern recovery methods to capitalize on the recent advancements in the field of machine learning, particularly in the areas of pattern recognition and computer vision. This would enable computers to assist in the software design recovery phase, which has traditionally been the exclusive domain of human developers due to their ability to handle abstract concepts and comprehend innovative ideas. The central hypothesis of our work suggests that, by developing the appropriate intermediate representation, the problem of design pattern recovery from code can be framed as a pattern recognition and signal processing problems. This transformation facilitates the utilization of these novel techniques to enrich and enhance the efficacy of existing methods for addressing this task.

To examine our hypothesis, we present a novel approach for generating visual signatures of well-defined design patterns [1], the generated signatures identify and characterize the key elements of the code's architecture, such as abstract classes, inheritance relationships, static and class methods, as well as the interactions among these elements. Our method transforms the problem of design pattern recovery into pattern recognition and signal processing problems by generating an intermediate representation that is easily amenable to further analysis and interpretation by state-of-the-art machine learning models and signal processing techniques (Figure 1) to extract valuable information about the predefined design patterns micro architectures, which is extremely useful to detect those design patterns in complex source codes.

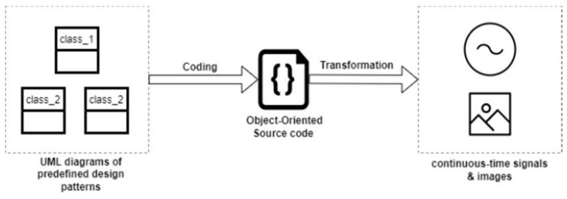
class_1
class_2
class_2
UML diagrams of predefined design patterns
Coding
Object-Oriented Source code
Transformation
continuous-time signals & images

Figure 1: Transforming The Design Recovery Problem Into A Pattern Recognition Problem By Generating Images Of Visual Signatures, And Into A Signal Processing Problem By Generating Continuous-Time Signals.

Additionally, Our method generates visual signatures that can be used to train supervised machine learning algorithms to detect similar design patterns in any source code built using the object-oriented programming paradigm. These generated signatures can also be utilized mathematically to easily analyze the source code using signal processing tools, enabling the detection of underlying design patterns. In section 2, previous studies are summarized and contextualized in order to provide a basis for our proposed approach. The unique aspects of our work are also highlighted in comparison to previous approaches. Section 3 introduces the general constructs, the mathematical tools and the transformation process used to generate the continuous-time signals and visual signatures. Section 4 introduces the experimental settings to generate a number of signals and visual signatures of well-defined design patterns micro architectures and discuss their properties and how they encode the unique features of their corresponding micro architectures. Section 5 concludes our experimental study and presents future work.

2. . RELATED WORK

The aim of this section is to furnish a contextual framework for our proposed approach to examine the task of design pattern recovery from a different perspective. Previous studies have investigated a range of techniques for this task, including machine learning, graph exploration, logical inference, metamodeling methods, and metrics-based methods. Many of these approaches involve the transformation of elements of the problem into an intermediate representation as a necessary step in the identification of patterns. This transformation is essential in providing suitable input for the identification process. This section classifies these transformation techniques into three categories based on their purpose and then discusses the distinctiveness of our approach in comparison to prior studies, as well as its contribution to the expanding knowledge base within the field.

2.1 Intermediate transformations for Machine Learning-Based recovery methods

Recently, [2] utilized a word-space model of Java files as an intermediate transformation by implementing the Word2Vec algorithm based on the syntactic and lexical representation (SSLR) of the Java source code. This process involves the creation of a call graph and the extraction of 15 source code features, after which a supervised machine learning classifier is utilized to identify patterns within the Java files. In [3] candidates identified by the Columbus matching algorithm [4] are labeled as true or false instances and their predictors are calculated. These values are then provided to a learning system, consisting of a decision tree-based method and a neural network, to generate a model that incorporates this acquired knowledge. This model can then be applied for pattern mining in unknown systems. A alternative approach for generating an intermediate representation for detecting design patterns within a source code is presented in [5], in their work, a parser is utilized to analyze the source code, extracts information, and preserves it in a metamodel. This metamodel represents each element in the source, including attributes, methods, classes, interfaces, and packages, and the hierarchical relationships between these elements, the extracted information is then used to generate a knowledge base in the form of Prolog facts, which are used in the pattern inference process. Another practical approach to encode both the structural and behavioral aspects of design patterns in the intermediate representation is introduced by [6], they use a combination of traditional predicate logic and Allen's interval-based temporal logic as their theoretical foundation. This combination is used to process formal specifications of each pattern, which have been converted into Prolog representations. Consequently, the use of logical inference allows for the recovery of both complete and partial patterns.

2.2 Intermediate transformations for Graph-Based recovery methods

Graphs exploration techniques have also been employed to address the problem at hand. These techniques involve the conversion of the problem into a graph-based analysis and manipulation task in order to discern patterns, [7] aims to detect commonly occurring sub-patterns within design pattern instances by transforming the source code and predefined design patterns into graphs, with classes as nodes and relationships as edges. Sub-pattern instances are identified through subgraph discovery and merged based on shared classes to determine if they match a predefined design pattern, moreover, the behavioral characteristics of method invocations are also compared to predefined method signature templates to identify final pattern instances. Adhering to the same transformation approach, the problem is also represented using graphs and matrices in [8], and a graph matching algorithm is employed to infer patterns directly from the graphs, It is stated that the properties of the chosen graph algorithm enable the method to recognize modified design patterns and leverage the presence of patterns in inheritance hierarchies to reduce the size of the analyzed graphs. Analogously, Polymorphism is also utilized in [9] to minimize the number of pattern definitions and the design recovery task is also conceptualized as a graph-based problem, however, the source code is transformed through the following process: it is first parsed into an abstract syntax graph (ASG) and analyzed using graph rewrite rules that annotate sub-graphs of the ASG. The parsing of source code is also performed in order to generate a rudimentary UML class diagram. Annotation objects are then introduced to the ASG to store information regarding the identified patterns and serve as the starting points for pattern searches. Finally, an inference engine based on Generic Fuzzy Reasoning Nets (GFRN) is utilized to recover design patterns and cliches in the legacy code. [10] employs a slightly different method for identifying design patterns in a program by comparing it to a "design motif," which can be thought of as a model or template of a design pattern. Nevertheless, the process involves converting the program and design motif into strings, represented as digraphs (graphs with directed edges), and then using a bit-vector algorithm to compare the strings and identify occurrences of the design motif in the code of the program.

2.3 Other intermediate Transformations techniques

Furthermore, Metamodeling, and other abstract modeling techniques have also been used to facilitate the task of pattern recovery, [11] demonstrates the ability to reason at a meta level about the structure of object-oriented source code in a language-independent manner, to accomplish this, they introduce a meta-level interface to extract detailed information about the structure of the source code. It is stated that the validity of this approach is demonstrated through the definition of a set of logic queries to detect object-oriented best practice patterns and design patterns in two distinct programming languages: Smalltalk and Java. [12] define a metamodel called Pattern and Abstract-level Description Language (PADL) to express the elements of the problem, which can be extended through inheritance. Then they use explanation-based constraint programming and constraint relaxation to identify microarchitectures that are similar to modeled design motifs, and it ensures traceability between the implementation and design by using the same language to describe different layers of the model and explicitly recording the set of lower-level elements that led to the existence of more abstract elements. Moreover, [13] employ the Sparx Systems Enterprise Architect (EA) modeling tool, EA is capable of reverse engineering source code from various programming languages, and creates an intermediate representation of the code in various form ats such as a database model, XML model, and class model. The authors utilize SQL queries to extract information about software artifacts from the intermediate representation. However, they find that the intermediate representation is not complete and must apply regular expressions to selected parts of the source code in order to extract additional information. It is stated that the authors successfully recover design patterns from mobile games with high accuracy, but they acknowledge that regular expressions have limitations in extracting nested information and plan to develop source code parsers in the future to address this issue. Additionally, other approaches have been utilized to model the problem, [14] propose an approach for identifying design patterns in object-oriented systems by analyzing an intermediate representation of the system in XMI format. The approach consists of three phases: structural analysis, which focuses on the system's structural characteristics such as classes and their relationships; behavioral analysis, which verifies the results of the structural analysis and may check back into the source code to eliminate false positives; and semantic analysis, which utilizes naming conventions and role information to distinguish between design pattern instances that have the same structural and behavioral characteristics. The approach has been automated in a tool called DP-Miner, which allows users to generate XMI input from existing tools. In [15] candidate design patterns are identified based on class diagram information obtained during the preliminary analysis using a visual language parsing technique, then, these candidates are validated through a fine-grained source code analysis. The approach has been implemented to recover Adapter, Bridge, Proxy, Façade, Composite, and Decorator patterns and has been tested on six public-domain programs and libraries using a design pattern recovery environment tool. [16] presents a Multiple Levels Detection Approach (MLDA) for recovering design pattern instances from Java source code. MLDA utilizes a Structural Search Model (SSM) to incrementally build the structure of design patterns based on a generated class-level representation of the system being investigated. Additionally, MLDA employs a rule-based approach to match the method signatures of candidate design instances to those in the subject system. [17] utilizes regular expressions, SQL queries, and annotations to extract relationships from legacy applications and source code models. The approach involves manually annotating the source code, reverse engineering the source code using an EA tool to obtain a model, defining pattern features through the definition of each pattern, translating the features into rules and searching for the desired pattern using SQL queries and regular expressions. Alternatively, [18] propose the use of the software metrics and a machine learning algorithm to experimentally fingerprint classes playing roles in design motifs in order to resolve the challenge of difficulty of identifying micro-architectures similar to these design motifs due to the large number of possible combinations of classes.

Our approach to generating an intermediate representation of the problem is distinguished by its ability to incorporate a new set of techniques for addressing the problem of design pattern recovery. Additionally, it offers a foundation upon which a decision support systems can be built. This is achieved through the generation of a dual dependent intermediate representation in the form of continuous-time signals and pattern images of the source code, as well as well-defined design pattern micro architectures. These facilitate the systematic classification of the structural elements of the problem using mathematical tools such as Fourier transformation, wavelet analysis and state-of-the-art machine learning classifiers. Meanwhile, the dynamic aspects of object-oriented architecture can be analyzed and predicted using time series prediction and regression techniques. The inclusion of time series analysis has the potential to inform the development of decision support systems that assist developers in refining their architectures based on previous examples of successful design decisions and best practices.

3. GENERAL CONSTRUCTS & PROBLEM TRANSFORMATION

3.1 Design patterns description

3.1.1 UML representation

In general, design patterns are depicted using UML diagrams, which present a template of the design that can be adapted to address a specific problem. These diagrams depict all relevant classes, including their types, attributes, and methods, as well as the relationships between these classes (an example is depicted in Figure 2). These relationships define how the elements of the design are related with one another when implemented to solve a given problem.

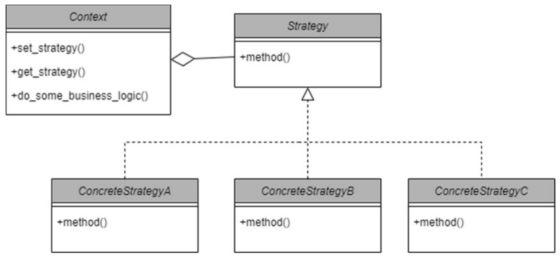

Figure 2: UML Representation Of The Strategy Design Pattern Micro Architecture.

3.1.2 Call graph representation

The call graph is a visual representation of the execution of the source code implementing the design pattern, which illustrates the chronological order in which the objects of the design interact with one another through method calls. It also provides information about the specific methods being called and the duration of each call. The call graph is generated through the execution of the source code and serves as a useful tool for understanding and analyzing the behavior of the design pattern.

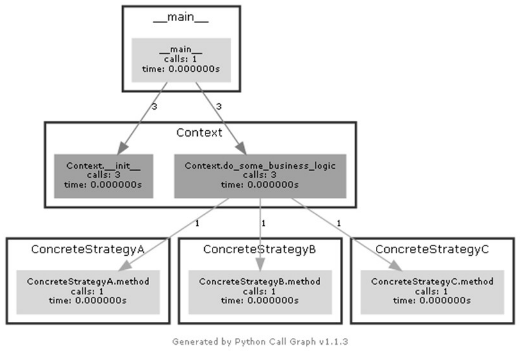

Figure 3: The Call Graph Generated From The Python Implementation Of The Strategy Design Pattern Micro Architecture Described In Figure 2.

The execution of a Python source code implementing the Strategy design pattern that is depicted in Figure 2 results in a series of calls, which are logged in the figure 3. The primary routine, “main”, initiates the call graph by invoking various methods of the Context class. From there, the Context class communicates with ConcreteStrategyA, ConcreteStrategyB, and ConcreteStrategyC in order to complete the processing. It is worth noting that the call graph also records the number of times each class method is called during the execution.

3.2 Combining UML diagrams, call graph logs and code keywords into a time-continuous signal spike

Our goal is to thoroughly integrate UML diagrams and call graph representations in order to fully encode the behavioral and structural aspects of a well-defined design pattern implementation into a continuous-time signal spikes, the generated signal is used later to generate a pattern image carrying the same amount of information previously provided by both of the diagrams. However, we opt to extract certain information including whether a method belongs to the static category or serves as a constructor for a specific object or a super constructor, through analysis of the code utilizing the specialized terms offered by the programming language to classify class methods, in Figure 4 we have summarized our approach of combining the data from the different sources. Moreover, we introduce a time dimension component to our signal which encodes the behavioral aspect of the code design by exploiting the call graph chronological nature, this makes the spikes in our generated signal occur in same order as the calls depicted in the correspondent call graph.

The frequency of each spike in the generated signal is used to encode the identities of the caller and the callee objects in a one-to-one reversible manner, this creates a permanent bond between every pair of objects in the call graph log and their correspondent signal spike, facilitating the ability of any future decision support system to target objects identities in its optimization processes. Finally, the amplitude of each spike of the signal encodes the structural information of the code design, to do that, we mine the source code keywords and decorators to determine the type of the class method used by the caller object to communicate with the callee object, then we determine the amplitude of the spike accordingly. It is noteworthy to mention that if the caller-callee pair of objects define an inheritance or abstraction relationship and the constructor method of the callee object is being used by the caller object implicitly or explicitly, this results in an additional spike in the signal. This information is obtained by checking the UML diagram.

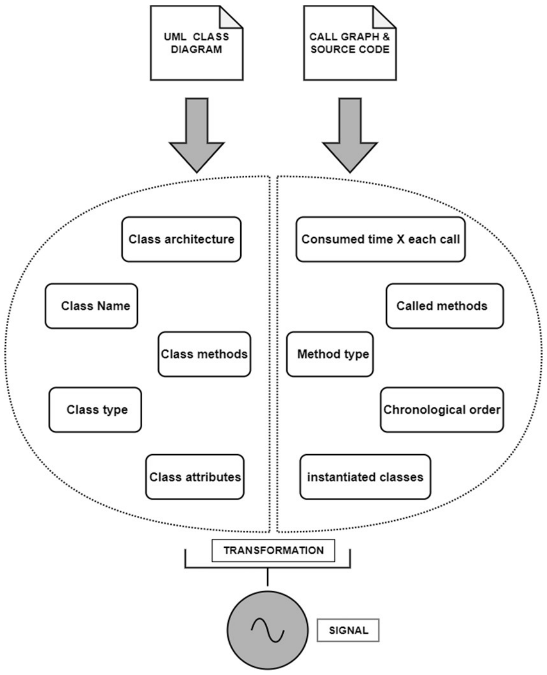

Figure 4: Gathering All Needed Information From The Three Sources : Source Code, Call Graph And UML Class Diagram, Then A Transformation Process Is Applied To Obtain A Continuous-Time Signal.

3.3 Problem transformation

As the final output of our transformation process is a time-continuous signal, we first need to model the signal components with respect to the previous information that we have obtained from combining the call graph logs and UML diagram representations in addition to the data from the code's keyword-based extraction process as described in section 3.2, to do that, we attribute a frequency f_i to each class c_i of the UML diagram, such as $f_i \in \mathbb{N}$ is unique for every element in the class space ξ of the UML diagram.

A_j is the amplitude of the signal spike with respect to its correspondent call $j \in \Omega$ where Ω represents the whole call graph log. A_j is defined to be a member of a predefined set χ, where each element of χ is a pair of the form (μ_k, ν_k), such as $\nu_k \in \mathbb{R}^*$ which represents the numerical value of amplitude. The variable μ_k represents the classification of the information conveyed by the call j, for the purposes of simplicity, we refer to the value of μ_k as the "Objective of the call" throughout the remainder of this paper. The possible values of μ_k that were investigated within the context of this research are listed in Table 1.

Table 1: List Of The μ_k Values Or "Objectives Of The Call".

μ_k	**"Objectives of the call"**
μ_1	Interface implementation
μ_3	Inheritance
μ_4	Object constructor call
μ_5	Getter or Setter method call
μ_6	General processing method call
μ_7	Cloning method call
μ_8	Static method call

In the context of the call graph diagram, a call j is typically characterized by three components: the caller, the callee, and the calling method. The caller is the object that initiates the call, while the callee is the object being called. The calling method is the class method utilized by the caller to facilitate communication with the callee. This calling method is contextualized and categorized by our transformation process to be represented by the variable μ_k as the "Objective of the call"**.** All the calls depicted in the call graph diagram are encoded using this structure of three components and logged in Ω following the same chronological order as shown in the diagram. Figure 5 illustrates the logged structure of the call and highlights the location of the μ_k variable within the structure. Consequently, upon determining the value μ_k we can utilize χ to access the amplitude value of the corresponding signal spike.

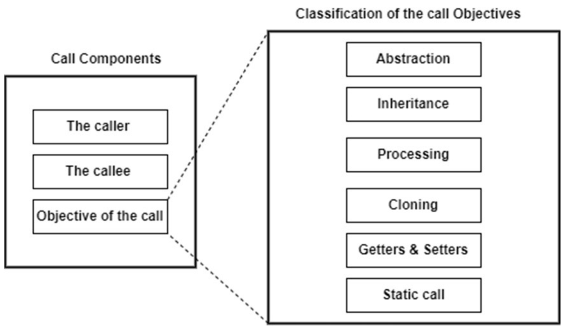

Figure 5: Each Call Is A Signal Spike With A Specific Frequency Obtained By Combining The Frequencies Of 'Caller Object', 'Callee Object', And A Specific Amplitude Defined By The Objectives Of The Call Set.

Finally, the frequency f_j of one call $j \in \Omega$ is obtained by combining both of the frequencies of the caller and the callee objects objects in a one-to-one reversible manner. In section 3.3.2 we present the detailed calculation of f_j. Meanwhile we summarized the problem transformation elements in Figure 6.

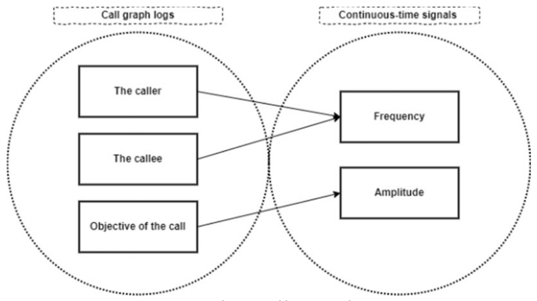

Figure 6: Mapping The Call Graph Log Components With A Continuous-Time Signal Components

3.3.1 Mathematical modelling & calculation of the signal spikes components

To generate time-continuous spikes for our signal we adopt a generic form of the sinusoidal wave function (1) defined on $\mathbb{R}^*$, then we parameterize it to fit our previous constraints with regards to the frequency and amplitude definitions.

$$S_j(t) = A_j \sin\left(\alpha_j\left(t - \beta_j\right)\right) \quad (1)$$
$$\text{Such as } j \in \Omega,\ t \in \mathbb{R}^*$$

S_j defines the signal spike of one call $j \in \Omega$, A_j is the amplitude of the spike. α_j, β_j are numerical parameters that are determined by the frequencies of the caller and callee objects, as well as the initial conditions of the equation. A call j is modelled by a spike with a period T_j of the signal over a continuous interval of time $[t_h, t_{h+1}]$, $h \in \mathbb{N}$, such as :

$$|t_{h+1} - t_h| = f_j = \frac{1}{T_j} \quad (2)$$

We have selected the constraint (2), because it ensures that the spike reaches its maximum value when $t = t_m$ such as :

$$\max_{t\in[t_h,t_{h+1}]} S_j(t) = S_j(t_m) = A_j \quad (3)$$

In order to fit the general form of sinusoidal curve, t_m must be exactly in the middle of the time interval $[t_h, t_{h+1}]$:

$$t_m = t_h + \frac{f_j}{2} \quad (4)$$

Consequently, α_j, β_j are obtained by solving three equations using the coordinates $(t_h,0)$, (t_m,A_j), and $(t_{h+1},0)$ to fit the sinusoidal curve form in the time interval $[t_h, t_{h+1}]$:

$$\begin{cases} S_j(t_h) = 0 \\ S_j(t_m) = A_j \\ S_j(t_{h+1}) = 0 \end{cases} \Leftrightarrow \begin{cases} \alpha_j = \frac{\pi}{f_j} \\ \beta_j = -\frac{f_j}{2} + t_m \end{cases} \quad (5)$$

By incorporating α_j, β_j expressions, the equation (1) becomes:

$$S_j(t) = A_j \sin\left(\frac{\pi}{f_j}\left(t + \frac{f_j}{2} - t_m\right)\right) \quad (6)$$
$$\text{Such as } j \in \Omega \text{ and } t \in [t_h, t_{h+1}]$$

Finally, the correspondent spike of one call j which has a frequency f_j and amplitude A_j is depicted using the equation (6) in Figure 7.

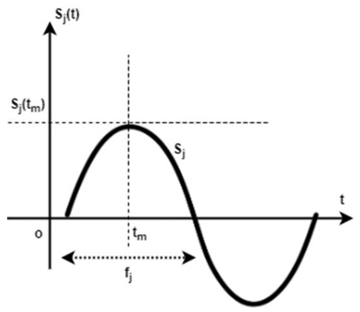

Figure 7: Graph Representation Of A Call Spike From Equation 8.

3.3.2 Calculation of f_j

Previously, it was stated that the frequency f_j must encodes information about both the caller and callee objects, if we let f_i, f_{i+1} denote the frequencies of the caller and callee objects, respectively, then f_j is a function of $f_i$and f_{i+1}. Additionally, we desire that the mapping between the spike and the objects involved in the call to be reversible one-to-one manner, so that we can determine which objects are responsible for a particular spike if necessary, for that reason we adopt the cantor pairing function [19] to calculate the frequency of the call with respect to f_i and f_{i+1} :

$$K(f_i, f_{i+1}) = \frac{1}{2}(f_i + f_{i+1})(f_i + f_{i+1} + 1) + f_{i+1} \quad (7)$$

Since $K(f_i, f_{i+1})$ results relatively in large natural numbers which will make our wave graphic representation inconsistent, we apply the TANH function on $K(f_i, f_{i+1})$ in order to limit our result inside the interval [0,1] and keep the one-to-one reversibility nature of our transformation:

$$f_j = \tanh(K(f_i, f_{i+1})) \quad (8)$$

3.3.3 Constitution of the signal

To encode the full call graph log Ω into multiple spikes of a continuous-time signal, we iterate over the calls in the log to generate the corresponding spikes using the equation (6). The union of these generated spikes forms a unified continuous signal ψ, which can be generalized using the following form:

$$\psi(t) = \cup_{j \in \Omega} \cup_{h \in \mathbb{N}} S_j(t) \quad (9)$$

4. EXPERIMENTS & RESULTS DISCUSSION

In this section, we begin by generating and examining the signals and visual signatures of three commonly utilized design patterns. We analyze the visual differences between the generated signals and visual signatures of the microarchitectures of these design patterns, and then explore how these intermediate representations encode the features of their corresponding design pattern models. Finally, we examine the construction of the visual signatures, which are presented as images, based on the generated signals and how they encode the same features as those contained in the generated signals.

4.1 Experimental setup & materials

In this study, we utilized a consumer laptop equipped with an Intel Core i7 3610QM CPU and 16 GB RAM to execute all computational tasks. Our method to generate the intermediate representations and the analyzed design pattern architectures were both implemented using the Python programming language. The scope of our investigation into object-oriented programming techniques was confined to the elements listed in the second column of Table 1 as "Objectives of the call".

4.2 Tests & results

❖ **Initializing the amplitude values**

To implement the signal spikes using equations 6 and 9, we propose the values of ν_k elements for the set . As described in section 3.3, ν_k corresponds to the numerical values of the amplitudes of the signal spikes . The (μ_k, ν_k) values, which can be found in the third column of Table 2, are constant throughout all tests.

Table 2: Assignment Of Initial Values For The χ Set.

μ_k	"Objectives of the call"	ν_k
μ_1	Interface implementation	-2
μ_3	Inheritance	-1
μ_4	Object constructor call	7
μ_5	Getter or Setter method call	10
μ_6	General processing method call	8
μ_7	Cloning method call	5
μ_8	Static method call	4

❖ **The Observer design pattern**

The Observer design pattern is a behavioral design pattern that defines a one-to-many dependency between objects. It allows a subject object to notify a set of observer objects when its state changes, and the observer objects automatically update themselves accordingly.

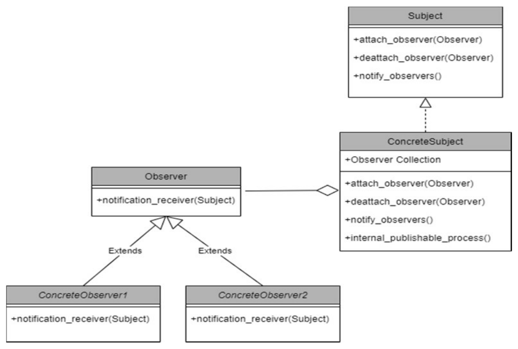
Subject
+attach_observer(Observer)
+deattach_observer(Observer)
+notify_observers()
ConcreteSubject
+Observer Collection
+attach_observer(Observer)
+deattach_observer(Observer)
+notify_observers()
+internal_publishable_process()
Observer
+notification_receiver(Subject)
Extends
Extends
ConcreteObserver1
+notification_receiver(Subject)
ConcreteObserver2
+notification_receiver(Subject)

Figure 8: A Simplified UML Class Diagram Of The Observer Design Pattern.

We implemented the simplified architecture of the Observer design pattern depicted in Figure 8 and generated its corresponding call graph diagram, which is illustrated in Figure 9.

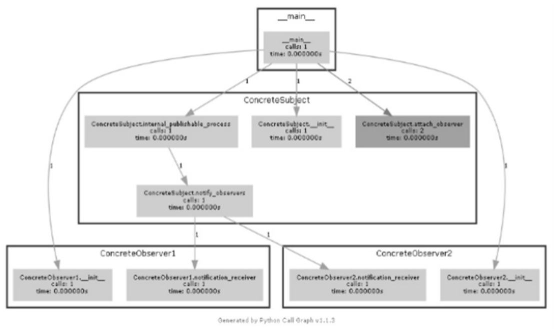
__main__
__main__ calls: 1 time: 0.000000s
ConcreteSubject
ConcreteSubject.internal_publishable_process calls: 1 time: 0.000000s
ConcreteSubject.__init__ calls: 1 time: 0.000000s
ConcreteSubject.attach_observer calls: 2 time: 0.000000s
ConcreteSubject.notify_observers calls: 1 time: 0.000000s
ConcreteObserver1
ConcreteObserver1.__init__ calls: 1 time: 0.000000s
ConcreteObserver1.notification_receiver calls: 1 time: 0.000000s
ConcreteObserver2
ConcreteObserver2.notification_receiver calls: 1 time: 0.000000s
ConcreteObserver2.__init__ calls: 1 time: 0.000000s
Generated by Python Call Graph v1.1.3

Figure 9: The Call Graph Of The Observer Design Pattern Micro Architecture's Implementation.

Additionally, we have implemented a routine that assigns a frequency, which is modelled to the variable f_i described in section 3.3.2, to each class/object of the UML class diagram and the call graph. In addition, the routine attributes a unique frequency to any object that encapsulates static methods and to all the interfaces and super classes of the architecture, it is worthy to note that all the interfaces and the super classes in the same architecture share the same frequency. The frequencies are listed in Table 3. Finally, we generated the correspond signal ψ of the implemented design pattern architecture by combining the data provided by the UML class diagram and the call graph diagram.

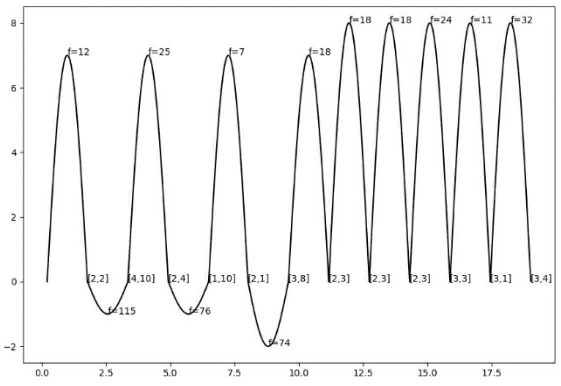

Figure 10: The Generated Signal Of The Observer Design Pattern Micro Architecture.

As observed in Figure 10, the signal preserves the structural and behavioral characteristics of the corresponding architecture, the frequency f_j (as defined in equation 6) of each spike is indicated next to the peak of its corresponding curve, and at the end of each spike we have indicated the frequencies f_i and f_{i+1} of the caller and callee objects, respectively, which were used to calculate the final frequency of the spike, for the purpose of conducting analytics, Table 3 can be utilized to identify the objects present in the signal using their frequencies. The signal oscillates at various amplitudes to encode the structural relationships among the objects of the architecture, For instance, the second spike of the signal reaches a negative peak corresponding to the value of -1, which according to Table 2 represents an inheritance relationship. The signal then reaches a positive peak of value 7, which indicates the use of a constructor. This implies that the object with frequency 4 (ConcreteObserver1 in our architecture) utilizes the properties of its superclass before it is instantiated by the object with frequency 2 (ConcreteSubject in our architecture). These two spikes represent the inheritance and aggregation depicted in the UML class diagram. The sixth spike with frequency 74 indicates that the object with frequency 3 (ConcreteSubject in our architecture) implements its interface before it is instantiated by Client (frequency 2), which represents the "main" routine in our call graph (Figure 9). The last five spikes, which oscillate at an amplitude of 8 corresponding to general processing methods represent the different objects calling each other's general methods for communication or data processing purposes. It is worth noting that the signal is always initiated by a first spike oscillating at an amplitude of 7, which represents the client object calling itself, corresponding to the invocation of the "main" method in our generated call graph.

Table 3: Runtime Assignment Of Frequencies To Each Object/Class Of The Call Graph And UML Diagram.

Object/Class	**Frequency**
Client (___main__)	2
ConcreteObserver1	4
ConcreteObserver2	1
Interface class	6
Super class	10
Static methods' object	8
ConcreteSubject	3

Afterward, we arranged the previously generated signal to create a 2D image that converts the signal into visual patterns (Figure 11). Each spike is represented in the same order as in the signal by a sequence of vertical tiles that change color according to the intensity of the amplitude and the frequency of the spike.

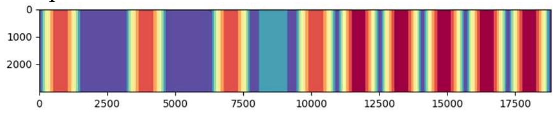
Figure 11: 2D Image of Patterns that represents the Observer design pattern micro architecture visual signature.

It is worth noting that negative spikes are represented in the image using their absolute value, which is generally depicted through shades of purple. The highest amplitude values tend to be represented in red.

❖ The Decorator design pattern

The decorator design pattern is a structural design pattern that allows the attachment of additional behavior or responsibilities to an object dynamically. It is an alternative to subclassing, in which new functionality can be added to an existing object by creating a new subclass.

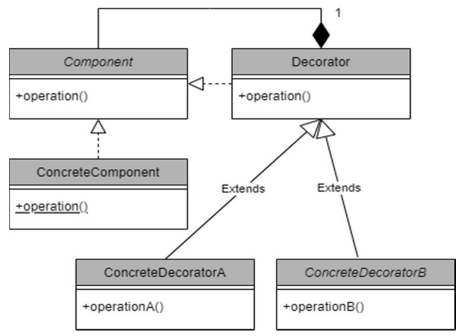

Figure 12: A Simplified UML Class Diagram Of The Decorator Design Pattern Micro Architecture.

As with the previous design pattern, we generated its call graph and assigned frequencies to the objects of the architecture, as shown in Table 4 and Figure 13.

Table 4: Runtime Assignment Of Frequencies To Each Object/Class Of The Call Graph And UML Diagram.

Class	**Frequency**
Client (___main__)	4
Code_caller	1
ConcreteDecoratorA	3
ConcreteDecoratorB	2
Interface class	9
Super class	11
Static methods' object	7
ConcreteComponent	5

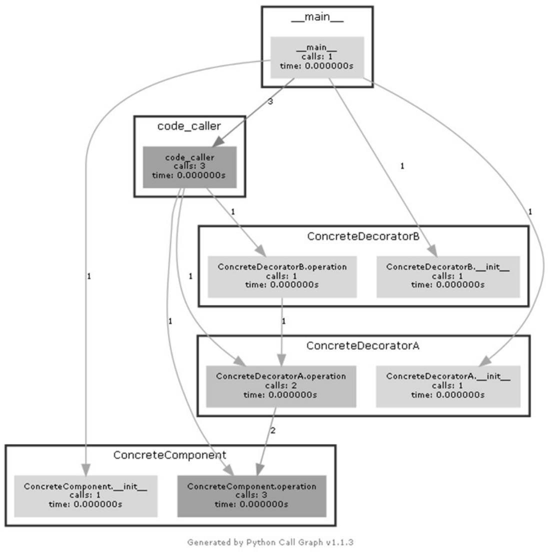

Figure 13: The Call Graph Of The Decorator Design Pattern Micro Architecture's Implementation.

Afterward, we generate the corresponding signal and image of patterns as depicted in Figure 14 & 15.

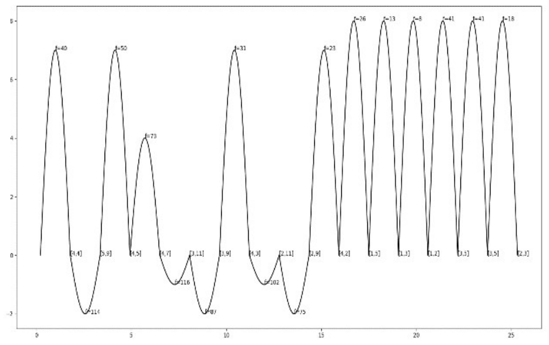

Figure 14: The Generated Signal Of The Decorator Design Pattern.

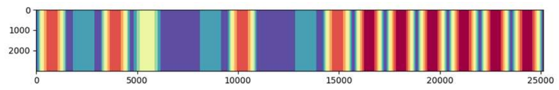

Figure 15: 2D Image Of Patterns That Represents The Decorator Design Pattern Micro Architecture Visual Signature.

The generated signal of the decorator differs significantly from the previous design pattern signal as it encodes new behaviors. For example, the fourth spike, which oscillates at an amplitude of 4 corresponding to the use of static methods, indicates that the client has utilized a static method. This behavior is also indicated by a special color in fourth group of tiles in the pattern image. Additionally, the fifth and sixth spikes represent the structural relationships as depicted in the UML class diagram (Figure 12) among the ConcreteDecoratorA, Decorator, and Component objects: ConcreteDecoratorA extends the Decorator class and the Decorator class also implements the Component interface. Other spikes are distributed among general processing calls and constructor calls.

❖ The Prototype design pattern

The Prototype design pattern is a creational design pattern that allows objects to be created by copying a prototype instance. It is an alternative to the more traditional approach of using a class hierarchy and a factory method to create new objects. In the Prototype pattern, a prototype object is created and registered with a prototype manager. When a client wants to create a new object, it asks the prototype manager for a copy of the prototype.

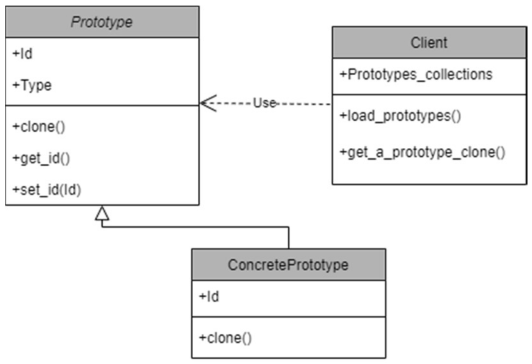

Figure 16: A Simplified UML Class Diagram Of The Prototype Design Pattern.

We followed the same steps as before to generate the corresponding frequency values and call graph depicted in Table 5 and Figure 17, respectively, and then generated the signal and visual signature of the Prototype design pattern, as illustrated in Figure 19.

Table 5: Runtime Assignment Of Frequencies To Each Object/Class Of The Call Graph And UML Diagram.

Class	**Frequency**
Client (___main__)	4
Copyreg	3
Get_a_prototype_clone	5
Interface class	10
Super class	12
Static methods Class	8
Copy	2
ConcretePrototype	1
Load_prototypes	6

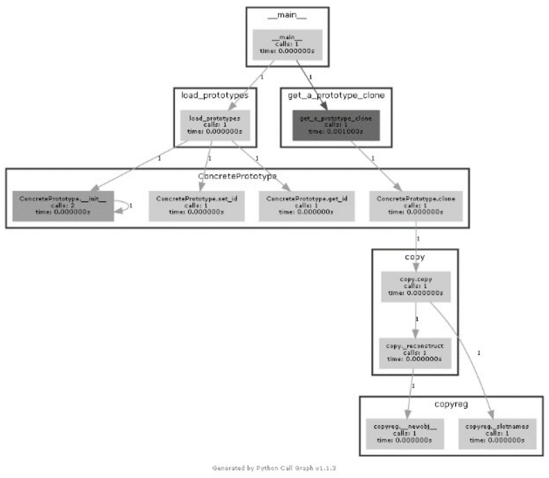

Figure 17: The Call Graph Of The Of The Prototype Design Pattern Micro Architecture's Implementation.

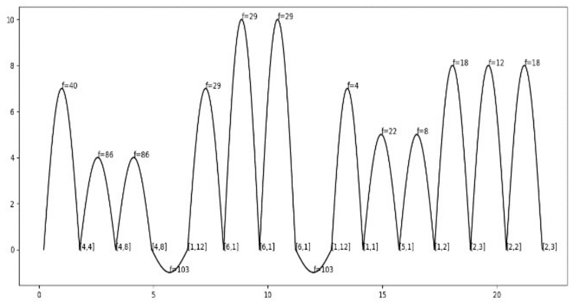

Figure 18: The generated signal of the Prototype design Pattern.

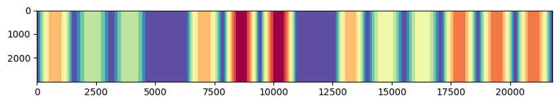

Figure 19: 2D Image Of Patterns That Represents The Prototype Design Pattern Micro Architecture Visual Signature.

The signal characterizes the Prototype design pattern's unique and principal feature, the cloning techniques, through the 10th and 11th spikes oscillating at amplitudes of 5. On the other hand, the invocation of getters and setters methods is represented by the sixth and seventh spikes oscillating at amplitudes of 10.

4.3 Discussion

The transformation approach applied to the well-defined design patterns micro architectures presented in this work maintained all structural properties of these architectures by assigning each property a specific amplitude. Additionally, This approach incorporates objects' and classes' identities in the construction of the final signal, by such, it retained the communicational aspects of each specific design pattern. The behavioral aspects are also maintained by incorporating the chronological aspect of the corresponding call graph of each micro architecture within the signal. Furthermore, the generated representation encompasses all mentioned aspects resulting in a feature-rich representation which allows identification and classification based on the specific features of the design pattern. The method of transformation possesses an inherent invariancy with regards to the scale of the architectural design. This is a consequence of the fact that the replication of objects, which exhibit identical behavior, leads to the generation of spike patterns that can be readily extracted from the signal through filtering techniques. All of the prior results align uniformly with the predictions made by the hypothesis stated in section 1, thereby providing substantial evidence in favor of its validity. Furthermore, The intermediate formats derived as results of this work are fundamentally dissimilar to those utilized in the previously reviewed approaches in section 2. This distinction arises from their capability to facilitate the utilization of advanced machine learning and signal processing methodologies to tackle the challenge, thereby making techniques originally intended for other purposes adaptable not only to the task of retrieving design patterns, but also to help providing a knowledge base to support the development of future autonomous software design support systems.

5. CONCLUSION & FUTURE WORK

In the present study, a dual representation is proposed to facilitate the task of design recovery through the utilization of both signal processing techniques and state-of-the-art pattern recognition techniques. The purpose of this representation is to provide a dual opportunity to recover design information from code. However, it should be noted that the scope of this study is restricted to a limited number of design properties that were previously outlined in Table 2. In forthcoming research, efforts will be directed towards differentiation of the superclasses and interfaces within an architecture by endowing each entity with a unique identity via a systematic procedure. Moreover, the proposed approach has the potential to be generalized to various open-source legacy software, incorporating sophisticated pattern recognition techniques to evaluate the effect of intermediate representation characteristics on the classification accuracy of the design pattern recovery process. Additionally, future studies will investigate the influence of the variations in the implementation of similar design patterns, as well as the scale of the architectures, on the precision of patterns identification.

REFERENCES:

[1] E. Gamma, R. Johnson, R. Helm, R. E. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Deutschland GmbH, 1995.

[2] N. Nazar, A. Aleti, and Y. Zheng, "Feature-based software design pattern detection," *J. Syst. Softw.*, vol. 185, p. 111179, Mar. 2022, doi: 10.1016/j.jss.2021.111179.

[3] R. Ferenc, A. Beszedes, L. Fulop, and J. Lele, "Design pattern mining enhanced by machine learning," in *21st IEEE International Conference on Software Maintenance (ICSM'05)*, Sep. 2005, pp. 295–304. doi: 10.1109/ICSM.2005.40.

[4] R. Ferenc, A. Beszedes, M. Tarkiainen, and T. Gyimothy, "Columbus - reverse engineering tool and schema for C++," in *International Conference on Software Maintenance, 2002. Proceedings.*, Montreal, Que., Canada, 2002, pp. 172–181. doi: 10.1109/ICSM.2002.1167764.

[5] R. Couto, A. Nestor Ribeiro, and J. Creissac Campos, "MapIt: A Model Based Pattern Recovery Tool," in *Model-Based Methodologies for Pervasive and Embedded Software*, Berlin, Heidelberg, 2013, pp. 19–37. doi: 10.1007/978-3-642-38209-3_2.

[6] H. Huang, S. Zhang, J. Cao, and Y. Duan, "A practical pattern recovery approach based on both structural and behavioral analysis," *J. Syst. Softw.*, vol. 75, no. 1, pp. 69–87, Feb. 2005, doi: 10.1016/j.jss.2003.11.018.

[7] D. Yu, Y. Zhang, and Z. Chen, "A comprehensive approach to the recovery of design pattern instances based on sub-patterns and method signatures," *J. Syst. Softw.*, vol. 103, pp. 1–16, May 2015, doi: 10.1016/j.jss.2015.01.019.

[8] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. T. Halkidis, "Design Pattern Detection Using Similarity Scoring," *IEEE Trans. Softw. Eng.*, vol. 32, no. 11, pp. 896–909, Nov. 2006, doi: 10.1109/TSE.2006.112.

[9] J. Niere, "Fuzzy logic based interactive recovery of software design," in *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*, May 2002, pp. 727–728. doi: 10.1145/581469.581473.

[10] O. Kaczor, Y.-G. Gueheneuc, and S. Hamel, "Efficient identification of design patterns with bit-vector algorithm," in *Conference on Software Maintenance and Reengineering (CSMR'06)*, Mar. 2006, p. 10 pp. – 184. doi: 10.1109/CSMR.2006.25.

[11] J. Fabry and T. Mens, "Language-independent detection of object-oriented design patterns," *Comput. Lang. Syst. Struct.*, vol. 30, no. 1, pp. 21–33, Apr. 2004, doi: 10.1016/j.cl.2003.09.002.

[12] Y.-G. Guéhéneuc and G. Antoniol, "DeMIMA: A Multilayered Approach for Design Pattern Identification," *IEEE Trans. Softw. Eng.*, vol. 34, no. 5, pp. 667–684, Sep. 2008, doi: 10.1109/TSE.2008.48.

[13] M. Khan and G. Rasool, "Recovery of Mobile Game Design Patterns," in *2020 21st*

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

International Arab Conference on Information Technology (ACIT), Nov. 2020, pp. 1–7. doi: 10.1109/ACIT50332.2020.9299966.

[14] J. Dong, D. S. Lad, and Y. Zhao, "DP-Miner: Design Pattern Discovery Using Matrix," in *14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'07)*, Mar. 2007, pp. 371–380. doi: 10.1109/ECBS.2007.33.

[15] A. D. Lucia, V. Deufemia, C. Gravino, and M. Risi, "Design pattern recovery through visual language parsing and source code analysis," *J. Syst. Softw.*, vol. 82, no. 7, pp. 1177–1193, Jul. 2009, doi: 10.1016/j.jss.2009.02.012.

[16] M. G. Al-Obeidallah, M. Petridis, and S. Kapetanakis, "A Structural Rule-Based Approach for Design Patterns Recovery," in *Software Engineering Research, Management and Applications*, R. Lee, Ed. Cham: Springer International Publishing, 2018, pp. 107–124. doi: 10.1007/978-3-319-61388-8_7.

[17] G. Rasool, I. Philippow, and P. Mäder, "Design pattern recovery based on annotations," *Adv. Eng. Softw.*, vol. 41, no. 4, pp. 519–526, Apr. 2010, doi: 10.1016/j.advengsoft.2009.10.014.

[18] G. Y-G, H. Sahraoui, and F. Zaidi, "Fingerprinting design patterns," in *11th Working Conference on Reverse Engineering*, Nov. 2004, pp. 172–181. doi: 10.1109/WCRE.2004.21.

[19] M. P. Szudzik, "The Rosenberg-Strong Pairing Function." arXiv, Jan. 28, 2019. Accessed: Dec. 17, 2022. [Online]. Available: http://arxiv.org/abs/1706.04129