Membership and Participation in Object Oriented and Procedural Paradigms

Grant Nelson
Gianforte School of Computing
Montana State University
Bozeman, USA
grant.nelson@student.montana.edu

Clemente Izurieta
Gianforte School of Computing
Montana State University
Pacific Northwest National Laboratory
Idaho National Laboratory
Bozeman, USA
clemente.izurieta@montana.edu

Derek Reimanis
Gianforte School of Computing
Montana State University
Bozeman, USA
derekreimanis@montana.edu

***Abstract*—Analyzing technical debt (TD) periodically with software tools is an important activity to help mitigate maintenance issues and sustain high levels of quality in software. Software written in Object Oriented languages (OO) have more dedicated TD analysis tools than software written in Procedural languages (PL) since many of the analysis techniques describe metrics that implicitly use a method's membership to a class in their calculations. The research described in this paper provides the conceptual foundation for using a membership matrix (used in current TD analysis techniques) and a participation matrix in TD analysis. Our early-stage research work proposes the participation matrix as a superset of the membership matrix and defines the mathematical requirements that must be met by a participation matrix constructed by any procedure. The participation matrix is defined as a fuzzy estimate of memberships and can be used to leverage TD analysis designed for OO to analyze PL programs. The operational implications of this work can allow practitioners to significantly benefit from improved TD tools that are also available in PL software.**

***Index Terms*—software maintenance, technical debt, membership, participation, object oriented, procedural, design recovery**

I. Introduction

High-level programming languages can be classified by many different paradigms. Two common paradigms are Object Oriented languages (OO) and Procedural languages (PL). OO is usually focused on objects or classes which encapsulate data and methods. These are languages like C++, C#, and Java. PL is focused on procedures and functions which act on data that can be grouped into objects or records. These are languages like C, Pascal, and Go.

Regardless of the paradigm, Technical Debt (TD) will build up in a program. TD slows down development and makes future changes more complicated [1]. To help mitigate (paydown and reduce) TD, developers need to maintain the software and need to determine which parts of the program require attention. Many programmatic TD analysis tools have been created to assist developers with locating, prioritizing, and mitigating TD.

Unsurprisingly, practitioners prefer automated tools, i.e. that run on a computer without human interaction, so that developers can obtain feedback easily as part of the development cycle. Many companies will use tools such as CodeScene and SonarQube as part of automated testing such as CI/CD pipelines or GitHub Actions. This allows Quality Assurance, Release Management, and other practitioners to periodically receive code health scores, which may include results from TD analysis.

Nelson and Izurieta [2] found that analysis tools for PL have not kept up with OO analysis tools. Nelson and Izurieta discuss complications that make the analysis of PL not trivial, specifically the difficulty with determining patterns and heuristics of objects in a PL program. This is caused by the difficulty to determine which function is associated with an object. In this paper, we provide the foundation to tackle this challenge by defining a participation matrix.

Less detailed TD analysis tools for PL mean that companies and practitioners using PL currently do not have the same development feedback and support as they do with OO. The lack of reported TD issues does not mean a lack of TD issues that need to be mitigated. Providing analysis tools that analyze PL programs is just as important to practitioners as tools that analyze OO programs.

For simplicity, we refer to functions, procedures, and methods, as methods. We also refer to objects, structures, records, and classes, as classes. The majority of the distinctions are irrelevant to this research and any salient differences will be highlighted.

The goal of this paper is to provide the conceptual foundation for an approach to determine the methods that are associated with their corresponding classes such that existing TD tools can be adapted to analyze PL programs. The approach described herein defines a participation matrix and describes how classes participate in methods performing the actions associated with said method.

To explain how the participation matrix can be leveraged by TD tools, we first discuss using a membership matrix in sections III and IV. Then, in section V we discuss the new idea of using a participation matrix in place of a membership matrix. We then provide concluding remarks and a discussion, with an outline of additional research regarding the participation matrix in sections VI and VII.

II. Related Work

To maintain and refactor PL programs [3]–[7], abstract reusable components [5], and to migrate PL programs to OO [8]–[12], classes with their related methods that had implicitly been built using the PL paradigm needed to be identified. The process of identifying classes in a PL paradigm is called object identification and design recovery (DR). DR in these related works is performed via clustering methods using a mix of different metrics. The clusters define how to partition the program into classes with related data and methods that function together. In these related works, the identified classes are used to either assist in manual maintenance, refactoring, or perform a conversion, i.e. transformation or migration, with minimal human interaction.

DR is related to participation as both utilize estimations of methods' membership to classes abstracted from programs that do not necessarily and explicitly define membership. Many of the DR metrics discussed in these related papers are different ways to determine related methods based on shared data. In other words, how classes (as groups of data) participate in methods performing their functions. The difference between DR and participation comes from how the membership estimations are used. DR uses the estimates in clustering and partitioning to assign each method to the most related class. Participation does not necessarily assign a method to a single class but instead allows one or more class candidates to be related to a method.

In several of these related works, DR requires manual input of information or practitioner interactions to help guide the identification and recovery. For participation to be programmatically performed as part of automated testing, manual interactions may not be used. However, since the goal of participation is to be used in TD analysis, there is no need for manual interactions to perform the analysis, instead interactions are only needed when interpreting and acting on the analysis results. Any misidentification can either be ignored or have analysis flags (e.g. code directives) added, as is already done when existing analysis tools identify issues that the practitioner feels are false positives.

Belady and Evangelisti [3] defined a basic process for clustering components (i.e., methods and data), into maintainable sets of components (i.e., classes).

Hutchens and Basili [4] extended Belady and Evangelisti's process by adding a "data binding" metric based on how the data is shared between methods and by using hierarchical clustering.

Biggerstaff [5] discusses the complications of DR for design reuse and reverse engineering. Biggerstaff defines a system where a developer is required to work with the results generated by analysis in order to interpret and refine useful information. The analysis system examines the importance of informal information, such as the names used for classes, methods, and data as well as developers comments.

Liu and Wilde [6] propose a system similar to Biggerstaff, however, Liu and Wilde focus more on the programmatic determination aspect by expanding on Hutchens and Basili and other similar work. Unlike prior research which discuss their approaches in terms of modules and components, Liu and Wilde are specifically looking at objects while equating them to OO classes. Liu and Wilde define two metrics used for clustering. The first is similar to Hutchens and Basili's metric based on shared data between methods and the second metric defines a binary matrix describing type and method interactions while taking into account subtypes.

```
public class Person {
    private String title;

    public String Name() {
        return this.title;
    }
}
```

Fig. 1. Example Java Membership.

Livadas and Johnson [7] define a third metric based on receiver parameter types in a method. They define a receiver as the "type of a parameter that is modified in at least one execution path" of a method. Livadas and Johnson also describe a more detailed way of aggregating the candidates objects together while clustering.

González, Czarnecki, and Pidaparthi [8] propose design transformations to convert a PL program into an OO program while keeping the overall design similar to improve the maintainability of the transformed program.

Pidaparthi, Zedan, and Luker [9]–[11] extend González, Czarnecki, and Pidaparthi's work by adding a resource usage matrix metric which can be used for both DR and for design transformation.

Islam, Toma, Selim, Gias, and Khaled [12] define a different hierarchical clustering technique for partitioning a program and a design migration from PL to OO.

III. Membership

Membership in OO paradigms is straightforward and is already being used in analysis of OO programs. A method is a member of a class if the method is defined in the program as part of the class. For example, in Fig. 1 the method `Name` is a member of the class `Person`. In OO, if a method is a member of a class, it is usually written within the definition of a class and the compiler allows the class or an instance of the class to be used as a receiver of a method call. It follows that the method of the class can be referenced as `this` or `self`. Membership does not involve language nuances such as subtyping, inheritance, generics, templates, lambda functions, closures, abstract classes, inner or child classes, etc.

Given $\mathcal{C}$ as the set of all of the classes in a program and $\mathcal{F}$ as the set of all of the methods in a program, then $\mathcal{B}_c$ is the set of methods which are members of a class $c \in \mathcal{C}$, defined as

$$\mathcal{B}_c = \{m \mid m \in \mathcal{F} \text{ and } m \text{ is a member of } c\} \qquad (1)$$

The $\mathcal{B}_c$ for any class will not intersect any other class since a method may only have membership to a single class. The following will remain true for membership since every method will have one and only one class to which it is a member:

$$\mathcal{B}_a \cap \mathcal{B}_b = \emptyset \ \ \forall\ a \in \mathcal{C}, b \in \mathcal{C}, a \neq b \tag{2}$$

$$\bigcup_{c \in \mathcal{C}} \mathcal{B}_c = \mathcal{F} \tag{3}$$

A. Membership as a Matrix

All the memberships in a program can be represented by a matrix, M. The columns of M are all the methods in the whole program, $\mathcal{F}$, and the rows of M are all the classes, $\mathcal{C}$.

$$M_{|\mathcal{C}| \times |\mathcal{F}|} = \begin{bmatrix} M_{1,1} & M_{1,2} & \cdots & M_{1,|\mathcal{F}|} \\ M_{2,1} & M_{2,2} & \cdots & M_{2,|\mathcal{F}|} \\ \vdots & \vdots & \ddots & \vdots \\ M_{|\mathcal{C}|,1} & M_{|\mathcal{C}|,2} & \cdots & M_{|\mathcal{C}|,|\mathcal{F}|} \end{bmatrix} \tag{4}$$

where

$$M_{c,m} = \begin{cases} 1 & \text{if } m \in \mathcal{B}_c \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

Since a method can only have a membership with one class, the sum of each column should always be one, as stated in (6). This is another way of representing the requirement shown in (2).

$$\sum_{c \in \mathcal{C}} M_{c,m} = 1 \quad \forall\, m \in \mathcal{F} \tag{6}$$

IV. Membership in Technical Debt Analysis

Membership is used to perform pattern matching in code to detect TD items, such as code smell detection. One such TD item is a code smell referred to as God class (GC). A GC, as defined by Marinescu [13], is an overly complicated class that takes on too much responsibility and, therefore, becomes hard to maintain [14]. GCs have also been called God Objects, Large Class Bloaters, Monster Classes, or Junk Drawers. GCs are detected by assessing the values of three metrics, Weighted Method Count (WMC), Tight Class Cohesion (TCC), and Access to Foreign Data (ATFD). Membership influences all three metrics.

WMC is the sum of McCabe's cyclomatic complexity [15] for all the methods in a class as defined by Chidamber [16]. Using McCabe's cyclomatic complexity measurement of the method m, CC_m, Chidamber's WMC for a class c is defined as:

$$WMC_c = \sum_{m \in \mathcal{B}_c} CC_m \tag{7}$$

As can be seen by (7), the method membership in classes influences the result of WMC. To use M, define a vector, CV, with all the cyclomatic complexity measurements for all methods. Multiplying M by CV gives a vector containing all the WMC scores for each class, $\mathcal{A}^*$.

$$CV_{|\mathcal{F}| \times 1} = \begin{bmatrix} CC_1 \\ CC_2 \\ \vdots \\ CC_{|\mathcal{F}|} \end{bmatrix} \tag{8}$$

$$\mathcal{A^*}_{|\mathcal{C}| \times 1} = M\,(CV) = \begin{bmatrix} WMC_1 \\ WMC_2 \\ \vdots \\ WMC_{|\mathcal{C}|} \end{bmatrix} \tag{9}$$

It can be shown that a single row, $\mathcal{A^*}_c$, from the vector $\mathcal{A}^*$ for a specific class, c, has the same value as the WMC calculated using (7) for the same class, WMC_c. The set of all methods in a row of M for the class c with the value of one, is the $\mathcal{B}_c$ by the definition in (5). Therefore when M and CV are multiplied, only the CC for all the methods in $\mathcal{B}_c$ are multiplied by one and all others are multiplied by zero.

$$\mathcal{A^*}_c = \sum_{m \in \mathcal{F}} M_{c,m} CC_m = \sum_{m \in \mathcal{B}_c} CC_m = WMC_c \tag{10}$$

Instead of CC, other method metrics can be used similarly when calculating a class metric as some part of a TD analysis. For example, to find the total lines of code (LOC) for a class, create a vector with the LOC of each method. Then when that vector is multiplied by M, the resulting vector will be the LOC per class.

ATFD can be calculated using M as well. ATFD is the use of data either via public fields or accessor methods of another class by a method and can calculated by creating a similar metric vector as CV except with the number of foreign data accessed per method. When multiplied by M the result is the ATFD per class.

TCC, as defined by Bieman and Kang [17], indicates how many directly connected methods exist in a class when compared to the total number of possible connections, where directly connected methods are methods that access one or more common instance variables of a class. TCC relies on knowledge of the membership of methods so that directly connected methods can be counted. Furthermore, TCC uses membership to calculate the total number of possible connections.

Beyond the GC determination, TD analysis includes identifying design patterns and code smells. Some code smells, such as Feature Envy, leverage similar metrics as ATFD and TCC, thus also rely on membership. When it comes to design pattern detection there are many patterns which rely on the methods within a class as well as the inheritance of other classes and implementation of interfaces. As an example, when detecting a pattern such as a Chain of Responsibility, as described in Design Patterns [18], a detector algorithm may scan for a method that calls another method with the same signature as the calling method, in a class stored in an instance variable. To effectively determine if a class or collection of classes are

```
class Person {
    void Play(Dog d, Toy t) { ... }
    void Fetch(Dog d, Toy t) { ... }
    void Throw(Dog d, Toy t) { ... }
}
class Dog { }
class Toy { }
```

Fig. 2. Memberships with Person.

```
class Person {
    void Play(Dog d, Toy t) { ... }
}
class Dog {
    void Fetch(Person p, Toy t) { ... }
}
class Toy {
    void Throw(Person p, Dog d) { ... }
}
```

Fig. 3. Refactored Memberships.

used in a Chain of Responsibility pattern, the membership of methods to classes is needed. For nearly all patterns and code smells a similar requirement on membership is required. However, a few patterns, such as Singletons, do not always rely on membership.

A. Changing Membership

As was shown in (7), the cyclomatic complexities of methods in a class all contribute towards the WMC for that class. This section discusses how refactoring a program changes the membership matrix thereby changing the TD for the program.

To exemplify with a grounded example of a membership matrix, suppose starting with a program which has three classes, $\mathcal{C} = \{Person, Dog, Toy\}$ and three methods, $\mathcal{F} = \{Play, Fetch, Throw\}$. All three classes are used in all three methods, meaning that any developer writing this program may decide to define each method as a member of any class. Each method will be the member of a class and take the other classes as parameters.

One possible implementation of this program is to put all the methods in the $Person$ class as shown in Fig. 2. The membership for the program would be:

$$M_{3\times3} = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \tag{11}$$

If a developer refactored the program to Fig. 3 without any changes to the metrics of any method other than the membership, the the M changes to:

$$M_{3\times3} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{12}$$

It can be seen that metrics such as the WMC will change only because of the changes to M. The ATFD and TCC would also change, suggesting that the labelling of GC's may also change.

```
type Person struct {
    title string
}

func (p *Person) Name() string {
    if p == nil {
        return `Unknown`
    }
    return p.title
}
```

Fig. 4. Example Go Struct And Function.

Each time the program is refactored, it is changing from one possible membership to another. Each possible combination contributes to the cognitive complexity of each method to different classes. The overall WMC for the entire program remains the same since the cognitive complexity of each method remains the same in this scenario.

For the prior example program there are 27 possible ways to assign membership. However, programs typically do not have every method use every class. More realistically, a method would only use a small subset of all the classes in a program. A program has many constraints on a method which makes a possible membership with some subset of classes better than any other class as described by the related works.

V. Defining Participation

Prior sections discussed how membership is used in TD analysis, how to represent membership as a matrix, and the constraints on memberships. This section discusses how to extend these concepts to construct a participation matrix, P, and how a participation matrix can be used in place of the membership matrix in TD analysis, thus allowing programmatic TD analysis designed for OO to be used on PL.

The most important difference between OO and PL is that methods in PL do not have membership with classes. The classes, typically called structures or records, are collections of data which a method performs some function on. This lack of membership is the motivation behind the related DR research.

Go is an example of a modern PL language. Go has receivers to allow convenient invocation of methods and to allow implicit subtyping of an interface based on method names and signatures without the receiver. When invoked, a method with a receiver only needs to know the receiver type; meaning that the method may be invoked with a null instance of the receiver type. A null dereference error will only occur if a field on a null receiver or null parameter is used. This distinction is the reason that the Go developers recommend that the receiver is not named `this` or `self`, as shown in Fig. 4. When a Go class is not constrained to implement an interface, then the receiver may be interchanged among the parameters of the method while only affecting how the method is invoked.

Go and other PLs do not have the same scoping constraints since a private field in a class has no use if a method can not access it. With these differences and no true membership of a method to a class, the utilization of traditional membership

matrices to perform TD estimates is misaligned. Instead, we propose a fuzzy estimate to determine method membership; that is, a fuzzy value that provides indication as to the degree to which a method is a member of a class. We base these fuzzy estimates on how methods use difference classes, or alternatively, how classes participate with a method to accomplish the method's goal.

A. Participation as a Matrix

As with the definition of M, the columns of P represent all the methods in the whole program, $\mathcal{F}$, and the rows of P represent all the classes, $\mathcal{C}$.

$$P_{|\mathcal{C}|\times|\mathcal{F}|} = \begin{bmatrix} P_{1,1} & P_{1,2} & \cdots & P_{1,|\mathcal{F}|} \\ P_{2,1} & P_{2,2} & \cdots & P_{2,|\mathcal{F}|} \\ \vdots & \vdots & \ddots & \vdots \\ P_{|\mathcal{C}|,1} & P_{|\mathcal{C}|,2} & \cdots & P_{|\mathcal{C}|,|\mathcal{F}|} \end{bmatrix} \tag{13}$$

However, instead of only zero or one, as permitted for $M_{c,m}$, $P_{c,m}$ represents the fuzzy estimate which c participates in m performing its work. This fuzzy estimate is normalized to be between zero and one inclusively.

$$P_{c,m} \in [0,1] \subset \mathbb{R} \tag{14}$$

The same requirement as (6) also applies to P such that (15) remains true. The sum of any column must always be equal to one. This is so that the metric of any method is properly reported. For a metric such as cognitive complexity, this ensures that the overall WMC for the entire program is reported the same with P.

$$\sum_{c\in\mathcal{C}} P_{c,m} = 1 \quad \forall\, m \in \mathcal{F} \tag{15}$$

In a PL program some methods might only use primitives. For example, suppose we have a method to calculate an iteration of the Fibonacci sequence which takes only an integer and returns an integer. There are a few ways to handle these kinds of methods discussed in the DR research. Either the package, file, and usage of the method helps determine which classes participate in method, or pseudo-classes are created for the primitives and added to $\mathcal{C}$.

It can be seen that for all possible M and P for a program, that $M \subset P$, since a membership is simply the case where for any method all participation is contributed to only one class. This means that the P can be used for OO as well as PL. The method's membership to a class in an OO program can signify a high probability that the class participates in the method's purpose, since the developer wrote the method as a member of the class. However, sometimes developers have constraints, such as deadlines, therefore the membership is less than ideal.

B. Using Participation

The participation matrix can be used in TD analysis in the same way that the membership matrix is used. Given the same CV as defined in (8), we can modify the WMC (9) to use P.

```
type Person struct { }
type Dog struct { }
type Toy struct { }

func Play(p *Person, d *Dog, t *Toy) { ... }
func Fetch(p *Person, d *Dog, t *Toy) { ... }
func Throw(p *Person, d *Dog, t *Toy) { ... }
```

Fig. 5. Go Participation Example.

$$\mathcal{A}_{|\mathcal{C}|\times 1} = P\,(CV) = \begin{bmatrix} AWMC_1 \\ AWMC_2 \\ \vdots \\ AWMC_{|\mathcal{C}|} \end{bmatrix} \tag{16}$$

The values calculated for each class, $AWMC_c$, are the approximated WMC_c. The approximated WMC_c for a class is the sum of the percentage of cognitive complexity for each method that the class participates in. This approximation may be the same as the original WMC depending on the probabilities described by the participation matrix. The participation matrix can be used when determining TCC, ATFD, and estimating patterns while analyzing the TD of a program.

C. Participation Example

Using the OO program described in Fig. 2, suppose that the classes, $\mathcal{C} = \{Person, Dog, Toy\}$, participate with the methods, $\mathcal{F} = \{Play, Fetch, Throw\}$, in such a way to produce the following participation matrix:

$$P_{3\times 3} = \begin{bmatrix} 0.4 & 0.0 & 0.8 \\ 0.4 & 0.4 & 0.0 \\ 0.2 & 0.6 & 0.2 \end{bmatrix} \tag{17}$$

In the example program all the methods have a membership with $Person$, however P shows that the developer maybe should have written the memberships differently and the program may need to be refactored. The $Fetch$ method probably should not be a member of $Person$ since $Person$ does not participate in $Fetch$. The $Throw$ method probably should not take Dog as an argument for the same reason.

Using P in TD analysis, such as calculating WMC, will return the estimation for the metrics weighted by the percentage of participation. If P shows that all the methods heavily use Toy, then Toy might be determined to be a GC that needs to be broken up, regardless of the methods' membership to $Person$.

A similar participation matrix can be constructed from an equivalent PL program, shown in Fig. 5, if the methods still use the classes in the same way. A membership matrix can not be determined since there are no explicit memberships defined. We can only estimate how the methods relate to the classes. Therefore, the initial refactoring suggestions using the disparity between the participation matrix and the membership matrix can not be determined for a PL program. However, if the participation matrix is constructed via some DR procedure, then the existing TD analysis for OO could be run on a PL program to get an estimation of the same TD metrics.

VI. Conclusion

The goal of this research is to provide the conceptual foundation for an approach to determine a fuzzy estimate membership of methods in PL programs that can be used with existing TD analysis calculations. We can accomplish this goal by defining a participation matrix that follows (15), and we detail how it can be used in place of membership for TD analysis. Participation is not specific to any language paradigm, therefore it can be derived from a program written in either an OO or a PL paradigm. This assumes that there is a procedure to construct participation matrices, without using membership, which provide statistically similar results to using a membership matrix.

VII. Discussion and Future Work

Research still needs to be performed to determine different procedures for creating participation matrices using the source code of a program. The procedure will likely leverage several of the techniques and processes designed for DR as discussed in section II.

Future research will first focus on creating a participation matrix for an OO language so that the existing TD analysis methods can be compared against the same analysis using participation instead of membership, thus reducing uncertainty in TD measurements [19]. The procedure for creating a participation matrix can then be tested on several PL programs. The results of those studies should resemble empirical test results for other TD studies on OO languages.

Other research opportunities will arise once a procedure has been determined for creating participation matrices. One such research opportunity is determining a participation matrix for an OO program, and then comparing the class which participates the most in a method against the method's given membership to a class. This could provide an indication of which methods have been added to a less-than-ideal class. Another opportunity for research is to look for methods where the top two or more classes have similar values of participation as an indication that a method may need to be broken up into two or more methods.

Another set of opportunities will be to augment a participation matrix creation procedure with natural language processing or other machine learning that could leverage code comments and design documentation when available, and the identifiers found in the source code. Doing so could help provide estimations of memberships closer to the practitioners' intent. This additional analysis may be able to replicate the manual information entry parts found in the DR research.

References

[1] Clemente Izurieta, Ipek Ozkaya, Carolyn Seaman, and Will Snipes, "Technical debt: A research roadmap report on the eighth workshop on managing technical debt (mtd 2016)," ACM SIGSOFT Software Engineering Notes, 42:28–31, 03 2017. doi: 10.1145/3041765.3041774.

[2] Grant Nelson and Clemente Izurieta, "A gap in the analysis of technical debt in procedural languages: An experiential report on go," IEEE Software, 38(6):71–75, 2021. doi: 10.1109/MS.2021.3103710.

[3] L. A. Belady and C. J. Evangelisti, "System partitioning and its measure," Journal of Systems and Software, 2(1):23–29, 1981. ISSN 0164-1212. doi: 10.1016/0164-1212(81)90043-1.

[4] David H. Hutchens and Victor R. Basili, "System structure analysis: Clustering with data bindings," IEEE Transactions on Software Engineering, SE-11(8):749–757, 1985. doi: 10.1109/TSE.1985.232524.

[5] Ted J. Biggerstaff, "Design recovery for maintenance and reuse," Computer, 22(7):36–49, 1989. doi: 10.1109/2.30731.

[6] Sying-Syang Liu and Norman Wilde, "Identifying objects in a conventional procedural language: An example of data design recovery," In Proceedings. Conference on Software Maintenance 1990, pages 266–271, 1990. doi: 10.1109/ICSM.1990.131371.

[7] Panos E. Livadas and Theodore Johnson, "A new approach to finding objects in programs," J. Softw. Maintenance Res. Pract., 6:249–260, 1994.

[8] Néstor A. González, Chris Czarnecki, and Sagar Padaparthi, "Migrating software from procedural to object-oriented architecture," In SMC'98 Conference Proceedings. Cat. No. 98CH36218, volume 5, pages 4872–4877 vol.5, 1998. doi: 10.1109/IC-SMC.1998.727624.

[9] Sagar Pidaparthi, Hussein Zedan, and Paul A. Luker, "Resource usage matrix in object identification and design transformation of legacy procedural software," 1997.

[10] Sagar Pidaparthi, Hussein Zedan, and Paul Luker, "Conceptual foundations for the design transformation of procedural software to object-oriented architecture," 1998.

[11] Sagar Pidaparthi, Paul Luker, and Hussein Zedan, "Reengineering procedural software to object-oriented software using design transformations and resource usage matrix." In H. Zedan and A. Cau, editors, Object-Oriented Technology and Computing Systems Re-engineering, ch. 13, pages 182–197. Woodhead Publishing, 1999. doi: 10.1533/9781782420613.182.

[12] Mohayeminul R. Islam, Tajkia Rahman Toma, Mohammad Reza Selim, Alim Ul Gias, and Shah Mostafa Khaled, "Design migration from procedural to object oriented paradigm by clustering data call graph," International Journal of Information Engineering and Electronic Business, 8:1–13, 2016.

[13] Radu Marinescu, "Detection strategies: Metrics-based rules for detecting design flaws," In Proceedings of the 20th IEEE international Conference on Software Maintenance, pages 350–359, ICSM. IEEE Computer Society, Washington, DC, 2004.

[14] Nico Zazworka, Michele A. Shaw, Forrest Shull, and Carolyn Seaman, "Investigating the impact of design debt on software quality," In Proceedings of the 2nd Workshop on Managing Technical Debt, volume MTD'10, 978-1-4503-0586-0/11/05, New York, NY, USA, 2011. Association for Computing Machinery. doi: 10.1145/1985362.1985366.

[15] Thomas J. McCabe, "A complexity measure," IEEE Transactions on Software Engineering, SE-2(4):308–320, 1976. doi: 10.1109/TSE.1976.233837.

[16] Shyam R. Chidamber and Chris F. Kemerer, "A metrics suite for object oriented design," IEEE Transactions on Software Engineering, 20(6):476–493, 1994. doi: 10.1109/32.295895.

[17] James M. Bieman and Byung-Kyoo Kang, "Cohesion and reuse in an object-oriented system," In SSR '95, 1995.

[18] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software," Pearson/Addison-Wesley, 1995. ISBN 978-0-201-63361-0.

[19] Izurieta C., Griffith I., Reimanis D., and Luhr R., "On The Uncertainty of Technical Debt Measurements," IEEE ICISA 2013 International Conference on Information Science and Applications, Pattaya, Thailand, June 24-26, 2013.