Advances in Science, Technology and Engineering Systems Journal Vol. 2, No. 1, 248-254 (2017)

www.astesj.com

Special Issue on Computer Systems, Information Technology, Electrical and Electronics Engineering

ASTESJ

ISSN: 2415-6698

Using Naming Patterns for Identifying Architectural Technical Debt

Paul Mendoza del Carpio*

Research Professor, Department of Software Engineering, Universidad La Salle, Peru

ARTICLE INFO

Article history:
Received: 10 December, 2016
Accepted: 18 January, 2017
Online: 28 January, 2017

Keywords:
Architectural technical debt
Naming pattern
Code analysis

ABSTRACT

Hasty software development can produce immediate implementations with source code unnecessarily complex and hardly readable. These small kinds of software decay generate a technical debt that could be big enough to seriously affect future maintenance activities. This work presents an analysis technique for identifying architectural technical debt related to non-uniformity of naming patterns; the technique is based on term frequency over package hierarchies. The proposal has been evaluated on projects of two popular organizations, Apache and Eclipse. The results have shown that most of the projects have frequent occurrences of the proposed naming patterns, and using a graph model and aggregated data could enable the elaboration of simple queries for debt identification. The technique has features that favor its applicability on emergent architectures and agile software development.

1. Introduction

This paper is an extension of work originally presented in the 8th Euro American Conference on Telematics and Information Systems (EATIS) [38]. Taking an easy solution on short-term in an activity of any phase of software development (i.e., requirements, design, implementation), can generate an accumulated technical debt, which, in a given period of time, can become big enough to affect future deliveries, making hard getting a successful outcome [6,24,37]. The debt comprises any aspect known as inappropiate which has not been addressed in due time (e.g., complex source code that needs refactoring) [24]. This debt is a topic whose interest has been increased over the years [36]. Frequently the technical debt, when is inserted, is less visible for decision makers in the software development [5]. The development of techniques for identifying and monitoring incidences of technical debt, is important for making explicit the debt and it could be resolved in due time [3,11,22,24,35,37].

The technical debt can be inserted by not complying the architectural design, or by not using conventions or standards of programming [35]. Including this as a decision factor inside the software development, requires information about the incidences of technical debt in the software system, where these are located, and their magnitude; such information can be gotten through source code analysis [5].

The objective of this work is to present:

1. An analysis technique for identifying architectural technical debt by non-uniformity of patterns.
2. A set of naming patterns across the package hierarchy of the software system.

2. Architectural Technical Debt (ATD)

ATD is a kind of technical debt which comprises sub-optimal solutions regarding internal or external quality attributes defined in the intended architecture, mainly compromising the attributes of maintainability and evolvability [2,11].

Changes related to design qualities but not related directly to external behavior of the system, are frequently postponed or neglected to reduce delivery time of the software system [3], increasing the incidences of ATD.

ATD is a debt very related to source code [24], however, in practice, is hard to be identified because this does not provide observable behavior to final users [11,36], and can change with time due to information gotten from implementation details [2]. Therefore, the ATD cannot be completely identified at an initial stage [2].

*Corresponding Author: p.mendozadc@ulasalle.edu.pe

In [2], a set of ATD is introduced. Among them, ATD by non-uniformity of patterns is related to name conventions applied in part of the system which are not followed in another parts [2]. This instance of ATD is addressed in this work.

Furthermore, several agile approaches consider the architecture as an emergent feature where there is no early design; but the source code is refactored and the architectural elements are refined [21]. The refactoring is a regular practice used in agile approaches, and is often applied on source code [1]; this contributes to the emergence of a successful architecture, improving the internal structure of the application, making the architectural elements more comprehensible, and avoiding the architecture decay, specially in them defined slightly [15,21]. Performing an incomplete refactoring is a cause of ATD that can insert part of ATD and generates new debt [2]. The refactoring can be performed manually, or semi or fully automatic. The fully automatic approach carry out the identification and transformation of code elements, nevertheless a human commits modifications [1,16]. This work enables a fully automatic refactoring, taking into account the identification by the proposed analysis, and applying a transformation through a renaming of classes. The last is a kind of global refactoring (i.e., affects classes in more than one package) [10] with API level (Application Programming Interface) [30], which is often used automatically in programming environments [8,30] with aims of organization and conceptualization [25], standing out over other refactoring forms by supporting the software traceability [1].

3. Naming Patterns

As a software evolves, its code becomes a source of information that is up to date and contains relevant information about the application domain [14]. Complex code is a major source of technical debt [22]; the correct use of naming conventions defined by the architecture accelerates and makes easy the activities of software comprehension [34]. Nevertheless, these conventions could not be followed throughout the software system. Such phenomenon can be amplified in agile teams [2]; where the teams are empowered in terms of design, different development teams working in parallel accumulates differences in design and architecture, and naming policies are not always defined explicitly and formally, arising divergences and requiring effort [2].

The relevance of class names lies in determining the code legibility, portability, maintainability, and accessibility to new team members, and relating the source code to the problem domain [19]. Also, industry experts highlight the importance of identifier names in software [12,28,31]. Therefore, such importance can reach architectural analysis levels, where identifying component terms is a task less complicated when identifiers are comprised by complete words or meaning acronyms [9,14]. The following subsections present a set of naming patterns inspired on the organization of source code through packages; the patterns are defined taking into account the frequent use of terms in class names inside the subjacent package hierarchy. Examples are taken from several real projects of the organizations Apache and Eclipse.

3.1. Pattern: Package

In this pattern the term is often used by classes included in a same package. As an example, figure 1 shows packages of Apache MyFaces. f is defined as a value of minimal frequency; T is the set of terms used in class names; P is the set of packages; C(p) is the set of classes of p ∈ P; and C(p,t) is the set of classes of p which have names with the term t ∈ T. The terms t of this pattern are such that (| C(p,t) | / | C(p) |) ≥ f, and | C(p) | > 2.

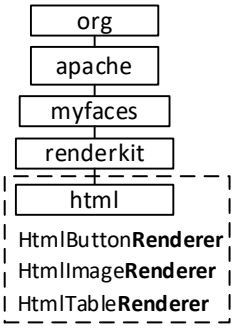

Figure 1: Example of Pattern Package

3.2. Pattern: Package Name

The term is often used by classes included in packages with same name. Figure 3 shows packages of Eclipse EGit. M is defined as the set of names of packages; F(m) is the set of packages with name m ∈ M; and F(m,t) is the set of packages with name m which contain classes having the term t in their names. The terms t of this pattern are such that (| F(m,t) | / | F(m) |) ≥ f, and | F(m) | > 2.

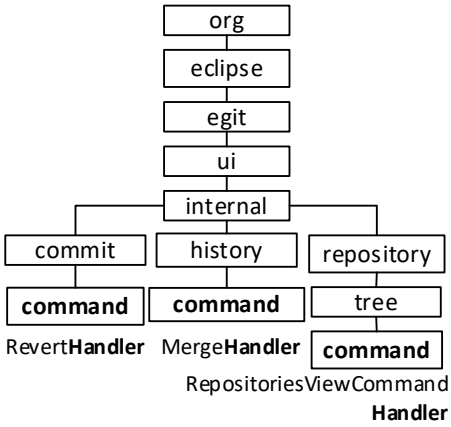

Figure 2: Example of Pattern Package Name

3.3. Pattern: Package Name and Level

The term is often used by classes included in packages with same name at same level of the package hierarchy. As an example, figure 3 shows packages of Apache Hadoop. N is defined as the set of package levels; G(n,m) is the set of packages with name m which are located at level n ∈ N; and G(n,m,t) is the set of packages with name m, at level n, which contain classes having the term t in their names. The terms t of this pattern are such that (| G(n,m,t) | / | G(n,m) |) ≥ f, and | G(n,m) | > 2.

3.4. Pattern: Package immediately superior

The term is often used by classes included in packages that are located in the same superior package. Figure 5 shows packages of Eclipse BPMN2. H(p) is defined as the set of packages located in package p ∈ P; and H(p,t) is the set of packages located in p which contain classes using the term t in their names. The terms t of this pattern are such that (| H(p,t) | / | H(p) |) ≥ f, and | H(p) | > 2.

4. Analysis Procedure

The analysis procedure performs the following steps: reading of packages and classes; creation of a graph of packages and classes; creation of a graph of terms with aggregated nodes; and querying of frequent terms and their frequency in the graph. The following subsections provide major detail about the relevant features.

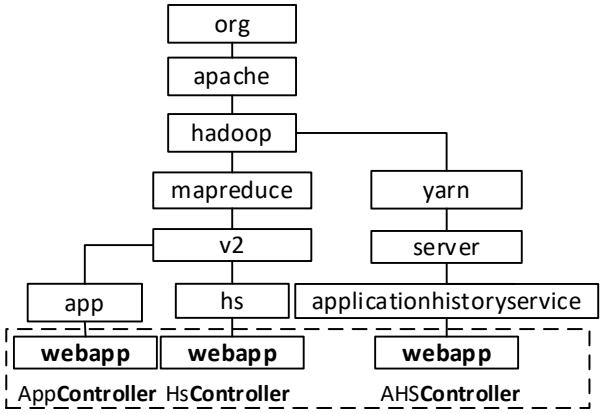

Figure 3: Example of Pattern Package Name and Level

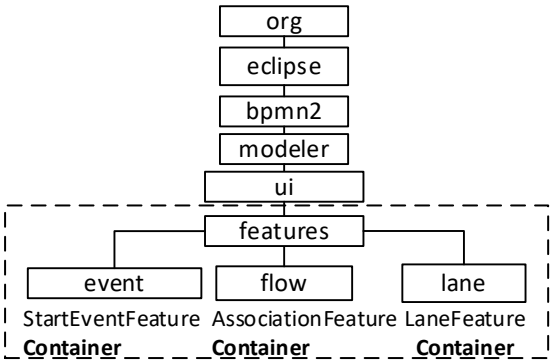

Figure 4: Pattern Package Immediately Superior

4.1. Graph based storage

The gotten terms are stored in a graph based database, such model was chosen due to its visualization capabilities, its ease of adding labels to nodes and creating nodes with aggregated data.

CQL (i.e.; Code Query Language) has been developed to perform exhaustive analysis on source code [27]. However, querying the source code directly without aggregating data, could affects response time. In this work, the graph query language is used as CQL with aims of taking the most of the database query mechanisms, which are developed to manage considerable amounts of data, and visualizing the results graphically. Moreover, having a graph enables software architects to query and visualize the data for purposes beyond this work.

4.2. Analysis of Term Frequency

The procedure of identifying frequent terms uses an analysis based on term frequency with collection range [32], frequency related to the number of times that a term occurs in a collection (e.g.; names of classes organized in packages). The frequency is computed by taking the percentage of term occurrences in same package (pattern Package) or in several packages. For each occurrence, the term position inside the name is considered (e.g., for ClientProtocol, the term Protocol is located in the second position).

The creation of the graph of terms is performed querying the names of classes and storing the occurrence of terms. The new nodes are created aggregating the number of occurrences for each naming pattern: by package, by package name, by package name and level, and by package immediately superior (these nodes will be denominated “aggregated nodes”); such data aggregation enables the simplification of graph queries. Then, aggregated nodes are labeled as frequent terms when they reach a minimal frequency.

5. Results

This work can be considered a valid proposal for ATD, because it corresponds to ATD by non-uniformity of patterns [2], and it takes into account a debt that affects maintainability and evolvability of software, without been included in not accepted topics as technical debt [37]. The approach of this proposal gives relevance to class names, and these determines the maintainability and legibility of software, between others [4,7,18,28,29,33,34]. Looking at the standard ISO/IEC FDIS 25010, maintainability includes the following quality attributes: modularity, reusability, analyzability, modifiability and testability. Considering that identifying components is less complicated task when the identifiers are comprised by significant terms [9,14], the presented analysis can support the analyzability and modifiability, getting significant terms by their frequent use (been representatives). Furthermore, if the naming patterns are not found in a software implementation, it could evidence poor choices of design and implementation with regard to used terms, affecting the test case artifacts [17]; in this sense, the analysis can also support the testability.

Table 1 shows some data about the projects considered henceforth: LOC (lines of code), QF (quantity of files), QP (quantity of packages), and QT (quantity of terms).

For evaluating the proposed analysis technique, an application was implemented to getting the terms used in class name following the CamelCase coding style (predominant style due to its ease of writing and adoption [7,13]), storing terms in a Neo4j database (standard graph database in the industry [26]). The application was executed on twenty projects of the organizations Apache and Eclipse (see table 1). All the source code was gotten from the repositories of Apache and Eclipse in GitHub (https://github.com/). Some project names were simplified to be shown; their names in GitHub are: eclipselink.runtime, hudson.core, scout.rt, servicemix-components.

This evaluation employs a minimal frequency of 0.8 to find frequent terms. Tables 2, 3, 4, and 5 show the following data for patterns 1, 2, 3, 4 (i.e., their order in section Naming Patterns) respectively: N (quantity of frequent terms), Min (minimal frequency found in frequent terms), Max (maximal frequency found), Avg (average frecuency), Stdv (standard deviation of frequency), TN (quantity of terms with a frequency lesser than 1).

Pattern Package is the most used pattern in the set; and Pattern Package Name and Level is the most restrictive and less used. The quantity of projects which does not have occurrences for any pattern is very low. In general, the frequent terms complies some pattern in more than ninety percent of their occurrences (i.e.; average value of 0.9), having cases with one hundred percent.

Table 1: Evaluated projects

	Proyecto	LOC	CA	CP	CT
Apache	JMeter	156900	842	102	526
	Hadoop	860382	4246	404	1403
	MyFaces	161973	816	76	419
	Camel	543222	4070	518	1172
	OpenJPA	324139	1385	54	619
	Wicket	288225	1814	277	804
	ActiveMQ	315613	2122	151	656
	OpenEJB	432266	2758	204	1050
	Geronimo	133804	1087	120	621
	ServiceMix	91837	621	132	314
Eclipse	Birt	1883941	7743	746	1384
	Egit	137718	775	78	402
	BPMN2	190873	1109	96	408
	Scout	415805	3021	691	812
	Xtext	396344	2699	360	1011
	OSEE	593489	6141	815	1496
	EclipseLink	890456	3643	324	994
	Hudson	146540	904	83	687
	EMF	478261	1228	179	475
	Jetty	259521	1315	151	639

Table 2: Frequency of terms for Pattern Package

Proyecto	N	Min	Max	Avg	Stdv	TN
JMeter	15	0.8	1	0.930	0.086	9
Hadoop	72	0.8	1	0.972	0.057	25
MyFaces	17	0.8	1	0.956	0.068	8
Camel	188	0.8	1	0.970	0.062	51
OpenJPA	9	0.8	1	0.914	0.096	5
Wicket	54	0.8	1	0.941	0.082	25
ActiveMQ	34	0.8	1	0.959	0.065	13
OpenEJB	28	0.8	1	0.942	0.077	12
Geronimo	16	0.8	1	0.921	0.078	9
ServiceMix	31	0.8	1	0.960	0.077	10
Birt	90	0.8	1	0.946	0.074	51
EGit	11	0.8	1	0.933	0.080	7
BPMN2	21	0.8	1	0.909	0.079	18
Scout	71	0.8	1	0.945	0.083	26
Xtext	36	0.8	1	0.951	0.074	16
OSEE	125	0.8	1	0.937	0.079	70
EclipseLink	71	0.8	1	0.958	0.066	32
Hudson	18	0.8	1	0.981	0.056	2
EMF	35	0.8	1	0.946	0.075	23
Jetty	34	0.8	1	0.954	0.071	13

TN values show the quantity of ATD incidences by non-uniformity of patterns. The percentage of TN in N shows the percentage of frequent terms, which were not applied uniformly. The maximal accepted value for this percentage can be defined by the development team, in accordance with the degree of use of naming conventions and how well defined is the architecture.

With aims to show the simplicity of queries, figure 6 shows the following query in Cypher language, which gets frequent terms with their respective packages for the pattern Package.

```
MATCH(t:FrequentTerm:PPackage),
(p:Package {fullName:t.packageFullName}) RETURN t,p
```

Code conventions can often be expressed as common practices which follows certain consensus before than as imposed rules [19]. The proposed analysis enables identifying a consensus of terms in following the naming patterns. Taking into account that refactoring can insert poor choices of design and implementation, evidencing such emergent consensus in the source code is useful before performing refactoring [2,19].

Table 6 shows some frequent terms which can be highlighted by their matching with concepts used in popular designs and architectures; showing that is possible to getting emergent and significant concepts from names of source code artifacts. The following query gets the TN terms for all naming patterns.

```
MATCH (t:FrequentTerm) WHERE t.percentage < 1
RETURN DISTINCT t.term
```

Similar works to this proposal were searched in the following digital libraries: ACM, IEEE Xplore and ScienceDirect; the search queries are shown.

For ACM:

```
recordAbstract:(+"technical debt" name names naming identifier identifiers)
```

For IEEE Xplore:

```
("Abstract":technical debt) AND ("Abstract":name OR "Abstract":names OR "Abstract":naming OR "Abstract":identifier OR "Abstract":identifiers)
```

For ScienceDirect:

```
ABS("technical debt") AND (ABS(name) OR ABS(names) OR ABS(naming) OR ABS(identifier) OR ABS(identifiers))
```

The quantities of gotten results for ACM, IEEE Xplore and ScienceDirect are 64, 0, and 1, respectively. The result gotten in ScienceDirect is a book chapter about refactoring advices. Many of the results from ACM are studies about the scope, causes, impact, and features of the technical debt; a few results are slightly related to this work, they address static analysis of source code at a low level, inspecting the source code content (i.e., operations and code sentences). Consequently, it can be affirmed that there is not similar proposals to this work, which is focused in naming of source code artifacts.

Table 3: Frequency of terms for Pattern Package Name

Proyecto	N	Min	Max	Avg	Stdv	TN
JMeter	2	0.938	1	0.969	0.044	1
Hadoop	34	0.800	1	0.965	0.073	7
MyFaces	2	1.000	1	1.000	0.000	0
Camel	21	0.800	1	0.949	0.085	8
OpenJPA	6	1.000	1	1.000	0.000	0
Wicket	9	1.000	1	1.000	0.000	0
ActiveMQ	1	1.000	1	1.000	0.000	0
OpenEJB	1	0.800	1	0.900	0.141	1
Geronimo	0	0.000	0	0.000	0.000	0
ServiceMix	10	0.800	1	0.911	0.090	5
Birt	34	0.800	1	0.951	0.081	11
EGit	5	0.800	1	0.960	0.089	1
BPMN2	3	1.000	1	1.000	0.000	0
Scout	59	0.800	1	0.935	0.089	26
Xtext	10	0.833	1	0.933	0.086	4
OSEE	41	0.800	1	0.963	0.072	10
EclipseLink	17	0.800	1	0.980	0.060	2
Hudson	1	1.000	1	1.000	0.000	0
EMF	17	0.857	1	0.960	0.056	7
Jetty	4	0.889	1	0.944	0.064	2

Table 4: Frequency of terms for Pattern Package Name and Level

Proyecto	N	Min	Max	Avg	Stdv	TN
JMeter	2	0.857	1	0.952	0.082	1
Hadoop	17	0.800	1	0.945	0.082	6
MyFaces	0	0.000	0	0.000	0.000	0
Camel	18	0.833	1	0.991	0.038	1
OpenJPA	0	0.000	0	0.000	0.000	0
Wicket	2	1.000	1	1.000	0.000	0
ActiveMQ	0	0.000	0	0.000	0.000	0
OpenEJB	0	0.000	0	0.000	0.000	0
Geronimo	0	0.000	0	0.000	0.000	0
ServiceMix	9	0.875	1	0.963	0.060	2
Birt	18	0.833	1	0.995	0.029	1
EGit	5	1.000	1	1.000	0.000	0
BPMN2	2	1.000	1	1.000	0.000	0
Scout	63	0.800	1	0.977	0.063	8
Xtext	2	0.800	0.8	0.800	0.000	2
OSEE	20	0.800	1	0.974	0.069	3
EclipseLink	1	1.000	1	1.000	0.000	0
Hudson	1	1.000	1	1.000	0.000	0
EMF	17	0.833	1	0.967	0.063	6
Jetty	4	1.000	1	1.000	0.000	0

Table 5: Frequency of terms for Pattern Immediate Superior

Proyecto	N	Min	Max	Avg	Stdv	TN
JMeter	3	1.000	1	1.000	0.000	0
Hadoop	6	0.800	1	0.967	0.082	1
MyFaces	0	0.000	0	0.000	0.000	0
Camel	26	0.800	1	0.954	0.075	10
OpenJPA	0	0.000	0	0.000	0.000	0
Wicket	2	0.896	1	0.965	0.060	1
ActiveMQ	60	0.800	1	0.915	0.037	57
OpenEJB	2	0.875	0.889	0.882	0.008	2
Geronimo	3	0.857	1	0.952	0.082	1
ServiceMix	6	0.906	1	0.974	0.042	2
Birt	41	0.800	1	0.974	0.059	8
EGit	1	1.000	1	1.000	0.000	0
BPMN2	5	0.800	1	0.922	0.078	4
Scout	20	0.800	1	0.950	0.075	9
Xtext	7	0.800	1	0.919	0.089	4
OSEE	29	0.800	1	0.952	0.084	9
EclipseLink	33	0.800	1	0.977	0.062	7
Hudson	0	0.000	0	0.000	0.000	0
EMF	16	0.800	1	0.947	0.085	3
Jetty	5	0.833	1	0.900	0.091	3

Table 6: Frequent terms

Proyecto	Términos
JMeter	Controller, Converter, Editor, Gui, JDBC, Meter.
Hadoop	Chain, Client, Container, Event, Scheduler.
MyFaces	Handler, Html, Impl, Implicit, Renderer, Tag.
Camel	Bean, Cache, Command, Filter, Task, Yammer.
OpenJPA	Concurrent, Distributed, Identifier, Managed.
Wicket	Bean, Checker, Handler, Resolver, Socket.
ActiveMQ	Adapter, Bridge, Broker, Command, Factory.
OpenEJB	Binding, Command, Entity, Factory, Thread.
Geronimo	Command, Deployment, Manager, Validation.
ServiceMix	Component, Factory, Filter, Interceptor, Ws
Birt	Action, Adapter, Filter, Handler, Validator.
EGit	Blame, Command, Git, Handler, Index, Node
BPMN2	Adapter, Editor, Event, Flow, Task, Validator.
Scout	Activity, Browser, Inspector, Job, Page, Service,
Xtext	Facet, Fragment, Module, Page, Resource, Ui
OSEE	Action, Command, Client, Service, Word.
EclipseLink	Accesor, Converter, Query, Resource, Table.
Hudson	Team, X
EMF	Action, Adapter, Command, Factory, Model.
Jetty	Bean, M, Response, Socket, Web

6. Conclusions

The naming patterns presented frequent ccurrences in several projects of the organizations Apache and Eclipse, showing that most of the frequent terms complies each pattern by ninety percent of their occurrences.

The proposed analysis identifies architectural technical debt by non-uniformity of naming patterns; which are applied frequently, but not followed in all the system. The used approach, based on naming patterns of source code artifacts, differs from other approaches which uses the source code content (e.g.; operations, sentences) for identifying technical debt.

The use of a graph based database was relevant, to enable using the database query capabilities as CQL, avoiding the limitations that could present a conventional CQL tool [27]; performing data aggregation in new nodes and making easy the elaboration of queries, which could be more complex or hard to be defined with a conventional CQL.

The proposal is applicable under an agile approach, which promotes focusing on product features and taking care about uncertainty in respect of ATD [2]. The analysis performed on source code does not require an architecture specification as input, and could be automatic through the continuous execution of queries during the software development, enabling the tracking of ATD. Additionally, the frequent terms, which were discovered, can be useful for identifying new emergent concepts in the software architecture.

References

[1] A. Mahmoud, N. Niu, "Supporting requirements to code traceability through refactoring", Requir. Eng., 19(3), 309-329, 2014.

[2] A. Martini, J. Bosch, M. Chaudron. "Investigating Architectural Technical Debt accumulation and refactoring over time", Inf. Softw. Technol. 67, 237-253, 2015.

[3] A. Ampatzoglou, A. Ampatzoglou, A. Chatzigeorgiou, P. Avgeriou, "The financial aspect of managing technical debt", Inf. Softw. Technol. 64, 52-73, 2015.

[4] B. Liblit, A. Begel, E. Sweetser, "Cognitive perspectives on the role of naming in computer programs" in Proceedings of the 18th Annual Psychology of Programming Workshop, 2006.

[5] C. Seaman, Y. Guo, C. Izurieta, Y. Cai, N. Zazworka, F. Shull, A. Vetrò, "Using technical debt data in decision making: potential decision approaches" in Proceedings of the Third International Workshop on Managing Technical Debt (MTD '12), IEEE Press, Piscataway, NJ, USA, 45-48, 2012.

[6] C. Sterling, Managing Software Debt: Building for Inevitable Change (1st ed.), Addison-Wesley Professional, 2010.

[7] D. Binkley, M. Davis, D. Lawrie, J. I. Maletic, C. Morrell, B. Sharif. "The impact of identifier style on effort and comprehension", Empirical Softw. Engg. 18, 2 (April 2013), 219-276, 2013

[8] E. Murphy-Hill, C. Parnin, and A. P. Black, "How We Refactor, and How We Know It", IEEE Trans. Softw. Eng., 38(1), 5-18, 2012.

[9] F. Deissenboeck, M. Pizka, "Concise and Consistent Naming" in Proceedings of the 13th International Workshop on Program Comprehension (IWPC '05), IEEE Computer Society, Washington, DC, USA, 97-106, 2005.

[10] G. Soares, R. Gheyi, E. Murphy-Hill, B. Johnson, "Comparing approaches to analyze refactoring activity on software repositories", J. Syst. Softw. 86(4), 1006-1022, 2013.

[11] I. Mistrik, R. Bahsoon, R. Kazman, Y. Zhang, Economics-Driven Software Architecture (1st ed.), Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2014.

[12] K. Beck, Implementation patterns, Addison Wesley, 2008.

[13] L. Guerrouj, "Normalizing source code vocabulary to support program comprehension and software quality", in Proceedings of the 2013 International Conference on Software Engineering (ICSE '13), IEEE Press, Piscataway, NJ, USA, 1385-1388, 2013.

[14] L. Guerrouj, M. Penta, Y. Guéhéneuc, G. Antoniol, "An experimental investigation on the effects of context on source code identifiers splitting and expansion", Empirical Softw. Engg., 19(6), 1706-1753, 2014.

[15] L. Chen, M. Ali Babar, "Towards an Evidence-Based Understanding of Emergence of Architecture through Continuous Refactoring in Agile Software Development" in Proceedings of the 2014 IEEE/IFIP Conference on Software Architecture (WICSA '14), IEEE Computer Society, Washington, DC, USA, 195-204, 2014

[16] M. Katić, K. Fertalj, "Towards an appropriate software refactoring tool support" in Proceedings of the 9th WSEAS international conference on Applied computer science (ACS'09), 2009.

[17] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, D. Poshyvanyk, "When and why your code starts to smell bad", in Proceedings of the 37th International Conference on Software Engineering (ICSE '15), Vol. 1, IEEE Press, Piscataway, NJ, USA, 403-414, 2015.

[18] M. Allamanis, E. T. Barr, C. Bird, C. Sutton, "Suggesting accurate method and class names" in Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015), ACM, New York, NY, USA, 38-49, 2015.

[19] M. Allamanis, E. T. Barr, C. Bird, C. Sutton, "Learning natural coding conventions", in Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014), ACM, New York, NY, USA, 281-293, 2014.

[20] M. Vakilian, N. Chen, S. Negara, B. A. Rajkumar, B. P. Bailey, R. E. Johnson, "Use, disuse, and misuse of automated refactorings" in Proceedings of the 34th International Conference on Software Engineering (ICSE '12), IEEE Press, Piscataway, NJ, USA, 233-243, 2012.

[21] M. A. Babar, A. W. Brown, and I. Mistrik, Agile Software Architecture: Aligning Agile Processes and Software Architectures (1st ed.), Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2013.

[22] N. A. Ernst, S. Bellomo, I. Ozkaya, R. L. Nord, I. Gorton, "Measure it? Manage it? Ignore it? software practitioners and technical debt" in Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015), ACM, New York, NY, USA, 50-60, 2015.

[23] N. Zazworka, M. A. Shaw, F. Shull, C. Seaman, "Investigating the impact of design debt on software quality" in Proceedings of the 2nd Workshop on Managing Technical Debt (MTD '11), ACM, New York, NY, USA, 17-23, 2011.

[24] N. Alves, T. Mendes, M. de Mendonça, R. Spínola, F. Shull, C. Seaman, "Identification and management of technical debt: A systematic mapping study", Information and Software Technology, 70, 100-121, 2016.

[25] N. Tsantalis, V. Guana, E. Stroulia, A. Hindle, "A multidimensional empirical study on refactoring activity" in Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research (CASCON '13), IBM Corp., Riverton, NJ, USA, 132-146, 2013.

[26] P. Macko, D. Margo, M. Seltzer, "Performance introspection of graph databases" in Proceedings of the 6th International Systems and Storage Conference (SYSTOR '13), ACM, New York, NY, USA, 2013.

[27] R. Urma, A. Mycroft, "Programming language evolution via source code query languages" in Proceedings of the ACM 4th annual workshop on Evaluation and usability of programming languages and tools (PLATEAU '12), ACM, New York, NY, USA, 35-38, 2012.

[28] R. C. Martin, Clean Code: A Handbook of Agile Software Craftsmanship (1 ed.), Prentice Hall PTR, Upper Saddle River, NJ, USA, 2008.

[29] S. Butler, M. Wermelinger, Y. Yu, H. Sharp, "Exploring the Influence of Identifier Names on Code Quality: An Empirical Study, in Proceedings of the 2010 14th European Conference on Software Maintenance and Reengineering (CSMR '10), IEEE Computer Society, Washington, DC, USA, 156-165, 2010.

[30] S. Negara, N. Chen, M. Vakilian, R. E. Johnson, D. Dig, "A comparative study of manual and automated refactorings" in Proceedings of the 27th European conference on Object-Oriented Programming (ECOOP'13), Springer-Verlag, Berlin, Heidelberg, 552-576, 2013.

[31] S. McConnell, Code Complete, Second Edition, Microsoft Press, Redmond, WA, USA, 2004.

[32] T. Roelleke, Information Retrieval Models: Foundations and Relationships (1st ed.), Morgan & Claypool Publishers, 2013.

[33] V. Arnaoudova, M. Di Penta, G. Antoniol, "Linguistic antipatterns: What they are and how developers perceive them", Empirical Software Engineering, 1-55, 2015.

[34] W. Maalej, R. Tiarks, T. Roehm, R. Koschke. 2014. On the Comprehension of Program Comprehension, ACM Trans. Softw. Eng. Methodol., 23(4), 2014.

[35] Z. Codabux, B. J. Williams, N. Niu, in Proceedings of the International Conference on Software Engineering Research and Practice (SERP'14), 2014.

[36] Z. Li, P. Liang, P. Avgeriou, N. Guelfi, A. Ampatzoglou, "An empirical investigation of modularity metrics for indicating architectural technical debt" in Proceedings of the 10th international ACM Sigsoft conference on Quality of software architectures (QoSA '14), ACM, New York, NY, USA, 119-128, 2014.

[37] Z. Li, P. Avgeriou, P. Liang, "A systematic mapping study on technical debt and its management", J. Syst. Softw., 101, 193-220, 2015.

[38] P. Mendoza del Carpio, "Identification of architectural technical debt: An analysis based on naming patterns", in Proceedings of the 2016 8th Euro American Conference on Telematics and Information Systems (EATIS), IEEE Computer Society, Washington, DC, USA, 10, 2016.