Refactoring for Software Architecture Smells

Ganesh Samarthyam
CodeOps Technologies LLP
C V Raman Nagar,
Bangalore, India
ganesh@codeops.tech

Girish Suryanarayana
Siemens Corporate Research
& Technologies,
No. 84, Electronics City,
Bangalore, India
girish.suryanarayana@
siemens.com

Tushar Sharma
Dept of Management Science
and Technology,
Athens University of
Economics and Business,
Athens, Greece
tushar@aueb.gr

ABSTRACT

Code smells and refactoring have received considerable interest from the academia as well as from the industry in the past two decades. The interest has given birth to various tools, processes, techniques, and practices to identify smells and refactor them. Despite the high interest, architecture smells and corresponding refactorings haven't received as much focus and adoption from the software engineering community. In this paper, we motivate the need of architecture refactoring, discuss the current related research, and present a few potential research directions for architecture refactoring.

CCS Concepts

•Software and its engineering → Software architectures; Software maintenance tools; Designing software;

Keywords

Architecture smells; Architecture refactoring; Software architecture

1. INTRODUCTION

Cities grow, cities evolve, cities have parts that simply die while other parts flourish; each city has to be renewed in order to meet the needs of its populace [...] Software-intensive systems are like that.

Grady Booch uses the evolution of a city as a metaphor to emphasize the need for refactoring software systems (in the forward of the book by Girish et al. [27]). Some of the indicators of an ailing city include congested roads and traffic jams, pollution of water bodies and air, and piles of garbage. Such a city requires "renewing" at multiple levels and scale. When a city fails to address the growing needs of its populace, it can lead to a "crisis" situation and ultimately to its abandonment.

The evolution of software systems is akin to evolution of a city. With evolution of a software system, complexity of the system increases unless efforts are made to maintain and reduce it [16]. Refactoring is a well-known technique defined as "behavior preserving program transformations" [8] to cope up with increasing complexity and keep the software maintainable. Failure to perform periodic refactoring leads to accumulation of technical debt [4] and ultimately to "technical bankruptcy". Hence, periodic refactoring is essential for long-lived software to improve and maintain structural quality of the software.

In general, smells and their corresponding refactorings are applied at various granularity levels. Smells could be categorized as implementation smells [8], design smells [27], and architecture smells [9]. The categorization of a smell depends on factors such as their scope and the impact on the rest of the system. Implementation smells have limited scope (typically confined to a class or file) and have a limited local impact. On the other hand, architecture smells span multiple components and have a system level impact. Since smells differ in their scope, impact, and effort required for refactoring, it is pragmatic to classify the smells into implementation, design, and architecture smells. Similar to smells, refactoring techniques applied to refactor these smells can also be classified as implementation refactoring, design refactoring, and architecture refactoring.

Due to a wide adoption of agile and lean methods of software development, refactoring has received considerable focus from the academia and the industry. Several excellent tools, processes, techniques and practices have been developed to adopt refactoring as an integral part of the development lifecycle [3, 6, 30]. However, architecture refactoring hasn't received as much focus from the software engineering community. In this paper, we summarize the state of the art on refactoring for architecture smells and provide a research outline for the road ahead.

2. MOTIVATION

Industrial software systems are typically complex and long-lived. Consider Windows operating system for example. It has grown to more than 50 million LOC over the last 25 years. Evolution at such a scale of time and size poses a threat to the structural quality of the software. Hence, periodic architecture refactoring is required to maintain the structural quality of such a complex and evolving software system. In addition, such refactorings are mandatory for the success of

IWoR'16, September 4, 2016, Singapore, Singapore
© 2016 ACM. 978-1-4503-4509-5/16/09...
http://dx.doi.org/10.1145/2975945.2975946

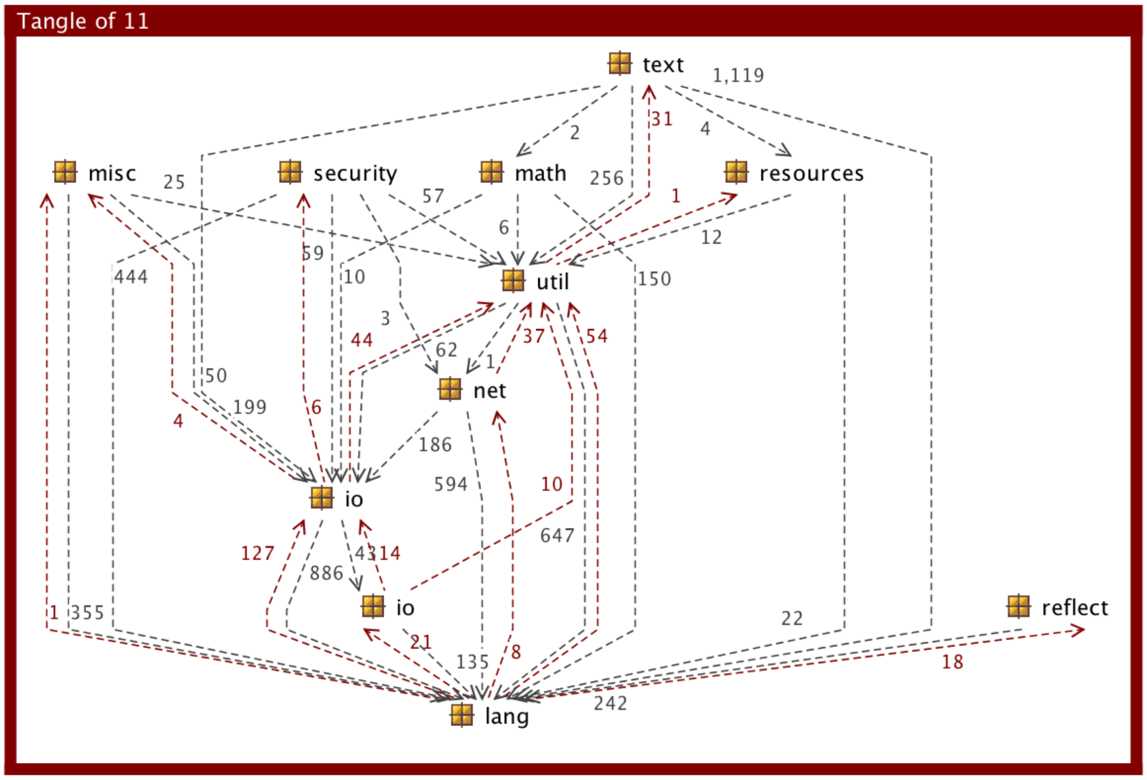

Figure 1: Tangle in Java core library depicting "dependency cycles between packages" smell

the software product as they facilitate an easier integration of new features. For instance, a major refactoring effort was carried out for Windows while evolving from Vista to Windows 7 version [12]. The primary goal of this refactoring was to remove undesirable dependencies among modules so that layering violations can be addressed and dependency structure can be improved.

In this case study, the authors refer to architecture refactoring without using the terms "architecture refactoring" or "architecture smell" anywhere in their paper; instead they use other related terms such as "system-wide refactoring". We argue that, this work (among numerous others) illustrate that "architecture refactoring" is an emerging domain that is yet to find its place in common vocabulary used by researchers and practitioners working in the area of software engineering in general and refactoring in particular.

Now, let us consider the Java core library (JDK). JDK has evolved considerably in size and complexity since the release of its first version in 1995. For instance, consider the "tangle" (i.e., a part of a dependency graph in which all the items are directly or indirectly dependent on all other items) in the Java 8 core library (refer Figure 1; the figure is generated by using Structure 101 tool). The packages involved in the cycle have to be used, reused, tested, and deployed together leading to various maintenance woes. It is referred as "dependency cycles between packages" architecture smell [17]. This smell negatively affects maintainability, reusability, testability, reliability, and deployability of the software. With changes being performed in Java 9 (project Jigsaw) [11], it is expected that the modularity concerns will be addressed, including the removal of considerable number of such cyclic dependencies between packages in the JDK.

From these case studies, it is evident that architecture smells (such as "large module" and "dependency cycles between packages" smells) are distinct from smells arising at other granularities such as implementation smells (for instance, "large method" [8]) and design smells (for instance, "cyclically-dependent modularization" smells [27]). As discussed earlier, the scope and impact of architecture refactorings is relatively higher when compared to refactoring at implementation and design granularities. Additionally, architecture refactoring require significant amount of effort and time compared to with implementation and design refactorings. Refactoring in these two case studies has taken many years with considerable planning and team co-ordination involved. At the same time, the benefits of architecture refactoring are also significantly high and serve relatively longer than other types of refactorings.

3. RELATED WORK

Garcia et al. [9] defined architecture smell as "a commonly (although not always intentionally) used architectural decision that negatively impacts system quality".

Researchers use various terms for architectural smells, including "architectural bad smells" [9], "architecture smells" [17], "anti-patterns" [2], "architecturally-relevant code smells" [1], "contra-indicated patterns" [15], "architectural defects" [23], "[accidental] architectural anti-patterns" [14]. Further, researchers have explored specific categories of architecture smells; for example, Rama et al. discusses "modularity smells"

[22], and Ouni et al. outlines "web service antipatterns" [21] in the context of Service Oriented Architectures (SOA).

On the other hand, architecture refactoring is defined as "a coordinated set of deliberate architectural activities that remove a particular architectural smell and improve at least one quality attribute without changing the system's scope and functionality" [32]. Architectural refactorings are also referred to as "high-impact refactorings" [7], "architecture transformation" [10], and "large refactorings" [17]. We have observed practitioners using other terms such as "architecture-oriented refactorings".

In our earlier work on design smells [24,27], we focused our study on the vast literature on object-oriented design smells. We cataloged and classified a number of recurring structural design smells based on how they violate key object oriented design principles.

There have been a few attempts to catalog architecture smells or refactorings [9, 26]. A few architecture refactoring case studies performed on industrial projects also have been reported. For example, Kumar et al. [13] presents an architecture refactoring case study performed on a mission critical application. There are exploratory studies on emerging topics in architecture refactoring, such as architecture refactoring in software product lines [5] and architecture technical debt [20].

4. POTENTIAL RESEARCH DIRECTIONS

Despite many attempts to explore architecture smells and corresponding refactorings in diverse dimensions, the domain is far from maturity. Based on the related work in this area (described in the previous section) and our experience as practicing architects, we envision the following areas for further research on this topic:

- **Catalog of architectural smells:** A comprehensive catalog of architecture smells and refactorings with an appropriate classification would better guide a software developer in understanding and addressing potential issues in the architecture of his/her software system. As discussed above, there has been a few attempts to catalog architecture smells and refactorings. However, a comprehensive catalog of architectural smells with detailed explanation, examples, context, their technical and economic impacts on the software product, and refactoring suggestions is still missing. We envision a taxonomy of architecture smells and corresponding refactorings, classified based on dimensions such as *Structure* and *Behavior*.

- **Tool support:** Although, there are a few tools such as Sonargraph [25] that may help us detect architecture smells. However, currently available tools are insufficient from various aspects:
  - The present set of tools does not detect a comprehensive number of architecture smells.
  - Architecture smells are contextual in nature. The present set of tools does not consider the contextual information for smell detection.
  - Common IDEs (Integrated Development Environments) such as Eclipse, Visual Studio, IntelliJ IDEA, and Netbeans allow developers to performing code refactoring automatically. However, IDEs lack support for detecting architecture smells and do not help perform architecture refactorings effectively.

  An ecosystem of tools for detecting architectural smells, providing refactoring recommendations, and helping perform the architecture refactorings is sorely lacking [29].

- **Economics of architecture refactoring (quantifying architecture technical debt):** Architecture debt is a significant component of technical debt for a software system. It is not trivial to quantify architecture debt given the challenges associated with it. Major challenges include detecting architecture smells automatically with associated severity, defining a quantification model which can be customizable and adjustable based on the project properties and mapping the detected smells to the model, and quantifying the interest aspect of the debt correctly. Researchers have proposed a few solutions, such as the one by Xiao et al. [31]; despite this, more focused research is required to develop a comprehensive yet practical quantification method for architecture debt. In general, more studies on evaluating the benefits of architecture refactoring (from software engineering economics perspective) is needed.

- **Empirical studies:** Though there are a few case studies reported on architecture smells or refactoring (such as Kumar et al. [13]), there is a huge potential for empirical studies on architectural smells or refactoring and their impact on various aspects of software development. Such an empirical study on large software systems may reveal many interesting insights about the characteristics of architecture smells and their impact on the software and the development efforts.

- **Refactoring and emerging architecture styles and patterns:** In the last few years, new architecture styles and patterns (such as Microservices [19] and Containerization [28]) have received considerable attention in the software industry. For example, refactoring from monolithic to microservices is a topic often discussed by practitioners [19]. However, architecture smells and refactoring in the context of these new architecture styles and patterns is yet to be explored by the software engineering researchers.

5. CONCLUSIONS

Practices such as refactoring are key to overcome or avoid the negative effects of software aging because they place change and evolution in the center of the software development process [18]. Knowing architecture smells for a software system and performing associated refactorings could avoid architecture erosion. However, architecture smells and refactoring are yet to receive extensive focus. In this paper, we have discussed related research and outlined a few potential areas for further research in the domain of architecture smells and refactoring. We hope that this work will spark more ideas and research in this domain and ultimately their wider adoption in industry.

6. REFERENCES

[1] I. M. Bertran. Detecting architecturally-relevant code smells in evolving software systems. In *33rd International Conference on Software Engineering (ICSE)*, pages 1090–1093, May 2011.

[2] W. H. Brown, R. C. Malveau, H. W. S. McCormick, and T. J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis.* John Wiley & Sons, Inc., 1st edition, 1998.

[3] Y. Cai and R. Kazman. Software architecture health monitor. In *Proceedings of the 1st International Workshop on Bringing Architectural Design Thinking into Developers' Daily Activities*, BRIDGE '16, pages 18–21, New York, NY, USA, 2016. ACM.

[4] W. Cunningham. The wycash portfolio management system. *SIGPLAN OOPS Mess.*, 4(2):29–30, 1992.

[5] H. S. de Andrade, E. Almeida, and I. Crnkovic. Architectural bad smells in software product lines: An exploratory study. In *Proceedings of the WICSA 2014 Companion Volume*, WICSA '14 Companion, pages 12:1–12:6, New York, NY, USA, 2014. ACM.

[6] S. Demeyer, S. Ducasse, and O. Nierstrasz. *Object Oriented Reengineering Patterns.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[7] J. Dietrich, C. McCartin, E. D. Tempero, and S. M. A. Shah. On the detection of high-impact refactoring opportunities in programs. *CoRR*, 2010.

[8] M. Fowler. *Refactoring: Improving the Design of Existing Programs.* Addison-Wesley Professional, 1 edition, 1999.

[9] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic. Toward a Catalogue of Architectural Bad Smells. In *Proceedings of the 5th International Conference on the Quality of Software Architectures: Architectures for Adaptive Software Systems*, pages 146–162. Springer-Verlag, 2009.

[10] L. Grunske. Identifying "good" architectural design alternatives with multi-objective optimization strategies. In *Proceedings of the 28th International Conference on Software Engineering*, pages 849–852. ACM, 2006.

[11] Project Jigsaw. http://openjdk.java.net/projects/jigsaw/, 2016. [Online; accessed 11-Jun-2016].

[12] M. Kim, T. Zimmermann, and N. Nagappan. An empirical study of refactoring challenges and benefits at microsoft. *IEEE Transactions of Software Engineering*, 40(7):633–649, July 2014.

[13] M. R. Kumar and R. H. Kumar. Architectural refactoring of a mission critical integration application: A case study. In *Proceedings of the 4th India Software Engineering Conference*, pages 77–83. ACM, 2011.

[14] A. Lauder and S. Kent. *Legacy System Anti-Patterns and a Pattern-Oriented Migration Response*, pages 239–250. Springer London, 2000.

[15] A. Lauder and S. Kent. Systems engineering for business process change. pages 225–240. Springer-Verlag New York, Inc., 2002.

[16] M. M. Lehman. Laws of software evolution revisited. In *Proceedings of the 5th European Workshop on Software Process Technology*, pages 108–124. Springer-Verlag, 1996.

[17] M. Lippert and S. Roock. *Refactoring in large software projects: performing complex restructurings successfully.* John Wiley & Sons, 2006.

[18] T. Mens, M. Wermelinger, S. Ducasse, S. Demeyer, R. Hirschfeld, and M. Jazayeri. Challenges in software evolution. In *Proceedings of the Eighth International Workshop on Principles of Software Evolution*, IWPSE '05, pages 13–22, Washington, DC, USA, 2005.

[19] S. Newman. *Building Microservices.* O'Reilly, 2015.

[20] R. L. Nord, I. Ozkaya, P. Kruchten, and M. Gonzalez-Rojas. In search of a metric for managing architectural technical debt. In *Joint Working IEEE/IFIP Conference on Software Architecture (WICSA) and European Conference on Software Architecture (ECSA)*, pages 91–100, Aug 2012.

[21] A. Ouni, M. Kessentini, K. Inoue, and M. O. Cinneide. Search-based web service antipatterns detection. *IEEE Transactions on Services Computing*, PP(99):1–1, 2015.

[22] G. M. Rama. A desiderata for refactoring-based software modularity improvement. In *Proceedings of the 3rd India Software Engineering Conference*, pages 93–102. ACM, 2010.

[23] R. Roshandel, B. Schmerl, N. Medvidovic, D. Garlan, and D. Zhang. Using multiple views to model and analyze software architecture: An experience report. Technical Report USC-CSE-2003-508, University of Southern California (CSE), 2003.

[24] G. Samarthyam, G. Suryanarayana, T. Sharma, and S. Gupta. Midas: A design quality assessment method for industrial software. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 911–920, 2013.

[25] Sonargraph. https://www.hello2morrow.com/products/sonargraph, 2016. [Online; accessed 11-Jun-2016].

[26] M. Stal. Software architecture refactoring. In *Tutorial in The International Conference on Object Oriented Programming, Systems, Languages and Applications*, 2007.

[27] G. Suryanarayana, G. Samarthyam, and T. Sharma. *Refactoring for Software Design Smells: Managing Technical Debt.* Morgan Kaufmann, 1 edition, 2014.

[28] M. H. Syed and E. B. Fernandez. The software container pattern. In *22nd Conference on Pattern Languages of Programs*, PLoP, 2015.

[29] R. Terra, M. T. Valente, K. Czarnecki, and R. S. Bigonha. Recommending refactorings to reverse software architecture erosion. In *16th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 335–340. IEEE, 2012.

[30] L. Tokuda and D. Batory. Evolving object-oriented designs with refactorings. *Automated Software Engg.*, 8(1):89–120, Jan. 2001.

[31] L. Xiao, Y. Cai, R. Kazman, R. Mo, and Q. Feng. Identifying and quantifying architectural debt. In *Proceedings of the 38th International Conference on Software Engineering*, pages 488–498. ACM, 2016.

[32] O. Zimmermann. Architectural refactoring: A task-centric view on software evolution. *IEEE Software*, 32(2):26–29, Mar 2015.