

Qualitative analysis of the relationship between design smells and software engineering challenges

Asif Imran

aimran@csusm.edu

California State University San Marcos
San Marcos, California, USA

Tevfik Kosar

tkosar@buffalo.edu

University at Buffalo
Buffalo, New York, USA

Abstract

Software design debt aims to elucidate the rectification attempts of the present design flaws and studies the influence of those to the cost and time of the software. Design smells are a key cause of incurring design debt. Although the impact of design smells on design debt have been predominantly considered in current literature, how design smells are caused due to not following software engineering best practices require more exploration. This research provides a tool which is used for design smell detection in Java software by analyzing large volume of source codes. More specifically, 409,539 Lines of Code (LoC) and 17,760 class files of open source Java software are analyzed here. Obtained results show desirable precision values ranging from 81.01% to 93.43%. Based on the output of the tool, a study is conducted to relate the cause of the detected design smells to two software engineering challenges namely "*irregular team meetings*" and "*scope creep*". As a result, the gained information will provide insight to the software engineers to take necessary steps of design remediation actions.

CCS Concepts: • Software and its engineering → Risk management.

Keywords: design smell detection, software maintenance, design debt, design challenges

ACM Reference Format:

Asif Imran and Tevfik Kosar. 2022. Qualitative analysis of the relationship between design smells and software engineering challenges. In *2022 The 3rd European Symposium on Software Engineering (ESSE 2022)*, October 27–29, 2022, Rome, Italy. ACM, New York, NY, USA, 8 pages.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ESSE 2022, October 27–29, 2022, Rome, Italy

© 2022 Association for Computing Machinery.

ACM ISBN 978-1-4503-9730-8/22/10...\$15.00

<https://doi.org/10.1145/3571697.3571704>

1 Introduction

Software design engineering is an important activity which requires careful application of design guidelines. Design issues contribute to 64% of software defects as a study by Jones et.al. [11] highlighted. Hence, quality and maintainability of software are significantly affected by design problems. One of the important design issues is a software suffering from design smells. Design smells violate the architectural principles and negatively affect the design of the software [19]. A software which has large volume of design smells contributes to design debt which in-turn increases technical debt. In recent years, a greater emphasis to reduce design smells has been given by software companies, engineers and researchers. Despite the increased importance, both developing and established software suffer from design smells, which is undesirable and hampers the long term sustainability. Research efforts are needed to identify the issues in software engineering practices deployed by companies which results in the design smells.

List of 25 design smells were provided by Suryanarayana et al. [19] which focused on all the fundamental design aspects. A design debt prioritization using portfolio matrix was proposed by Plosch et al. [16]. A model for detecting cyclic dependency and hub-like dependency smells using link prediction techniques was discussed by Diaz-Pace [6]. A catalogue to list all architectural smell detection tools together with their operating platforms was provided by Azadi et al. [3]. However, the accuracy of the tool requires further improvement. Additionally, teamwork issues in real life software development that contributes to design smells were not analyzed.

Based on the above motivation, this paper contributes primarily to determining a tool for design smell detection followed by establishing a relationship between the occurrence of a design smell to software engineering challenges in real life software development. First, the paper proposes a tool for design smell detection based on pseudo-model generation using *Abstract Syntax Trees (AST)*. It is used to analyze 409,539 Lines of Code (LoC) and 17,760 class files of open source Java software. Finally, precision and recall are calculated to identify the performance of the tool by 16 industry experts. Next, a survey is conducted with the same experts to identify a relationship between presence of the

detected design smells to software engineering challenges like *"irregular team meetings"* and *"scope creep"*.

This paper focuses on nine design smells highlighted in Table 1 which were selected from existing literature [19]. We also obtained source codes of release versions for 11 software which formed the testbed for our experiments. The names of the software have not been presented due to confidentiality issues and they are addressed as $S_1 \dots S_{11}$ respectively. Meta-data about the software are highlighted in Table 2. A total of 4,020 design smells are detected in our study. Quantitative values of the frequency of occurrence for all the smells is provided in the results analysis section. The results are desirable as precision values ranging from 81.01% to 93.43% are obtained for the analyzed software. Next, we conduct a qualitative study to identify the cause of the smells based on two software engineering challenges namely *"irregular team meetings"* and *"scope creep"*. These two challenges are selected mainly due to their influence on quality software development [2]. We conduct a survey among the 16 software experts with an aim to determine this relationship. The findings show significant impact of the two issues on the occurrence of design smells.

Based on the above information, the major contributions of this paper can be stated as follows:-

- Provide a tool for design smell detection for Java software and analyze its performance.
- Establish relationship between key software engineering challenges and the occurrence of design smells.

Rest of the paper proceeds as follows. Section 2 illustrates the research questions, identifies the selected software systems for this study and describes the implementation of the tool for design smell detection and recording. Section 3 provides the obtained results and analyses those. Section 4 describes the conducted survey to establish relationship between software engineering challenges and occurrence of design smells. Section 5 highlights the threats to validity and credibility, Section 6 discusses the related work, and Section 7 concludes the paper and discusses future research directions.

2 Empirical study setup

This qualitative study focuses on detecting design smells and identifying their causes based on feedback from industry experts. The following subsections identify the research questions and data collection strategy.

2.1 Research Questions

The following research questions are addressed in this paper.

RQ1: How to detect design smells in Java source codes?

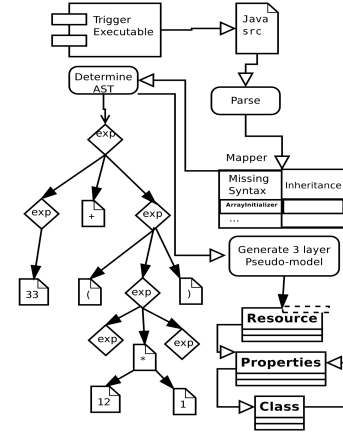


Figure 1. Representation for generating AST and pseudo-model for design smell analysis

This paper provides a tool to detect design related smells. The answer to this research questions will help software engineers detect design smells in the code, hence saving critical design debt and time.

RQ2: To what extent does real life software engineering challenges impact the presence of design smells?

The software team faces critical challenges like *"irregular team meetings"* and *"scope creep"* which impacts the design of the software. Such issues may also result in the occurrence of design smells in the final software product. We aim to study this relationship here.

2.2 Selection of Software Systems

Table 2 summarizes the list of eleven selected systems. For each of the systems, we highlight the life-time, *KLoc*, number of commits. Next, we tested those software to detect the design smells using the tool described in the following subsection.

2.3 Implementing Smell Detection Tool

The tool described here is motivated by the *Designite* tool [18]. However, *Designite* is designed for detection of smells for C# code whereas our tool is modified for detecting design smells in Java code. Figure 1 identifies the process of AST and pseudo-model generation for obtaining the design smells. It is stated that 33.27% of the software in the world are coded using Java. Presence of design smells in Java code significantly challenges the software engineer during incorporation of new updates. Also design smells possesses significant issues during final deployment of the software. Therefore, we are motivated to provide a tool to Java developers which can be used to detect design smells in their software, thus give them information related to the design smells present in their code.

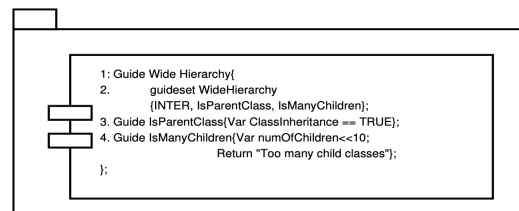
Unutilized Abstraction: This design smell occurs when an abstraction is declared, however its implementation is missing in the code. For example, a class <i>ABC</i> may be declared by the software developer, however if it is not used anywhere in the code, then it is a <i>Unutilized Abstraction</i> design smell.
Insufficient Modularization: When an abstraction is large and complicated, providing a scope of further modularization, it is said to suffer from this design smell. This smell occurs when the class is large in size, multiple class definitions are present in a file or the complexity of the class is high.
Broken Hierarchy: This occurs when the IS-A linkage between the supertype and subtype is absent. This interferes with the substitution capability among the two classes.
Deficient Encapsulation: When an abstraction given more accessibility to the users than it is required, it threatens the security of the software. The presence of this smell is called <i>Deficient Encapsulation</i> .
Cyclic Dependent Modularization: This smell occurs when the software violates the acyclic modularization technique. It occurs when one abstraction is cyclically dependent on another. If this design smell persists, then a change in one abstraction can have a ripple effect on other abstractions in the software. If the cyclic relationship is complicated in a large software, it is a challenge to detect that smell.
Unnecessary Abstraction: When a software has multiple layers of abstraction and many of the intermediate layers are not required, the software is said to suffer from <i>Unnecessary Abstractions</i> .
Wide Hierarchy: This smell is prevalent when an inheritance tree has a wide breadth and the intermediate types are missing. Our experiments show that it may appear in smaller volume in the software.
Missing Hierarchy: It occurs when a class has limited functionality within an operation. It may also happen when an operation is covered to a class. It can have a significant influence on the object oriented design, hence it is considered a design smell.
Multifaceted Abstraction: When an abstraction is responsible for multiple functionalities, managing it may become troublesome, resulting in this design smell. Also, many functionalities might be affected if an error occurs in the abstraction.

Table 1. Description of the selected design smells

Firstly, Java source codes are taken as input which is then used to form an *Abstract Syntax Tree (AST)*. The tokenized source text is read, to convert it to a tree. The first phase includes lexical analysis followed by syntax analysis. To form the *AST*, *Janett* [14] has been used which converts constructs and calls to Java libraries to C# format. Hence the *AST* can be formed using *NRefactory* [8] after parsing the Java code with *Janett*. Special Java constructs like *AbstractClasses*, *ArrayInitializer*, etc which are not present in C# are converted using a mapper. *IKVM* library is used to translate syntax and constructs, using inheritance whenever is necessary. Once the *AST* is ready, it is read via a script which translates it to a simple pseudo model.

After following the stated steps we can have a simple version of pseudo-model from the *AST* consisting of 4 layers. The lowest layer consists of the data elements and objects in Java code. The second layer is a description of the elements found in the lowest layer. The third layer consists of the data types and namespaces of the objects. Using the information captured in the pseudo-model, design smells in the Java code are detected with pre-specified rule cards [9]. Example of a rule card for detecting wide hierarchy design smell is provided in Figure 2. Finally, those smells are written in a provenance file which can be used for analysis.

System	Branch History	KLoC	Commits
<i>S</i> ₁	02/14-12/21	29.051	1871
<i>S</i> ₂	09/14-03/21	64.448	88
<i>S</i> ₃	05/14-02/18	23.002	1644
<i>S</i> ₄	08/09-03/19	24.267	385
<i>S</i> ₅	11/20-05/22	17.201	412
<i>S</i> ₆	07/10-05/19	19.391	352
<i>S</i> ₇	03/12-04/19	141.038	5531
<i>S</i> ₈	09/06-03/19	27.271	868
<i>S</i> ₉	08/05-03/19	24.698	15052
<i>S</i> ₁₀	07/02-03/19	49.534	5446
<i>S</i> ₁₁	10/19-11/21	17.331	788

Table 2. List of selected systems for the study**Figure 2.** Rule card for detecting wide hierarchy design smell

To use the system, users will require a Linux platform with terminal access. The *CLI* will enable running the tool to obtain frequency of specific design smells for software. Firstly, the executable version of the tool needs to be triggered. This is followed by passing the path of the Java source files of the target software. Next, the parameter and threshold values are passed. Afterwards, the design smells can be detected. Once detected, the tool writes the output of detection to a *provenance.log* file which can be later further analyzed considering it as a structured data-set of smells to obtain frequency of design smell prevalent in the software.

3 Precision and Recall of the Detection Mechanism

In this subsection we analyze the precision and recall for the design smell detection tool. Previous results are capable of detecting four types of design smells for Java [13] whereas this approach detects nine types of smells with desirable accuracy.

The suspicious classes were identified during the detection phase and those were manually analyzed by a team of 16 software professionals to validate the findings. We analyzed the number of detected design smells in Table 3. Details of the software professionals are highlighted in Table 5. It should be mentioned that the team of software professionals are from the two companies who designed and developed S_5 and S_{11} . The classes where design smells were detected by our tool in S_5 and S_{11} were presented to the respective software teams who further analyzed it to determine true positives for each smell. We extend the effort to the area of information retrieval and detect precision and recall [7]. More specifically, precision identifies the smells out of the total which could be successfully detected. Recall assesses the total number of detected and undetected smells.

Table 4 present the precision and recall values of 2 from the list of 11 software analyzed during this research. Overall, precision value for 9 smells were obtained as 81.01% for S_5 . For S_{11} , average precision of 93.43% was recorded. The number of suspicious classes which were manually analyzed by six software engineers are 16 and 33 for S_5 and S_{11} respectively, which are 17.4% and 19.0% of the total number of classes for the two aforementioned software. The low percentage of classes which were suspected allowed manual analysis of those within a reasonable time frame compared to having to analyze a total of 266 classes otherwise for those two software.

The detection tool has 100% recall value since no smell were missed within the scope of the nine types of smells considered in this research. Next the 16 software professionals were assigned a survey to determine the relationship between software development challenges like "irregular team meetings" and "scope creep" to the occurrence of design smells.

4 Survey design for analyzing causes of design smells

We conducted a survey among two software organizations out of the eleven considered in this study. The goal of this survey was to find out if the identified design smells were caused by some issues in the software development approach. Earlier we identify the two software as S_5 and S_{11} respectively. We try to determine if the cause of the design smells were due to two software engineering malpractices which are *irregular team communication* and *scope creep*. We conduct a survey with the 16 software engineers who designed and developed S_5 and S_{11} . The same set of 16 respondents were asked to establish the precision and recall of the design smell detection tool.

4.1 Description of selected projects:

Figure 4 identifies the teams of respondents who participated in the projects namely S_5 and S_{11} . Similar to the software names, we preserved confidentiality of the respondents by identifying them as $Rs_1..Rs_{16}$. Project S_5 was developed for in-house use. The software was designed and developed by a team of 10 members who are included in the 16 respondents for this survey. The team was divided into subteams among which the core-team was responsible for various activities such as design, and the other subteams were responsible for implementation, and validation of the project. The core-team initially committed to interact with the development teams on a bi-weekly basis. The communication was supposed to be online due to Covid-19 and that some members were located in different geographic locations. The team did not follow any defined software design model to the best of our knowledge.

Project S_{11} was primarily developed for the public and it was made available to be downloaded over the web. The team were able to conduct both in-person and online meetings post-Covid because they were located in the same geographic location. The team aimed to followed incremental software development process, however, members did not conduct regular team communication.

4.2 Survey mechanism:

We identified the respondents and selected S_5 and S_{11} from our list of 11 software mainly because we were able to contact them via Github and based on our prior association with the two companies who designed and developed these projects. To assure diversity, we focused on these two projects since they had developers from various backgrounds as identified in Table 5. The respondents primarily came from small-sized companies since we wanted to focus on the software engineering challenges faced by these companies. The interviews were semi-structured and were conducted after the team could give their views on the accuracy and precision of our design smell detection tool. we presented a questionnaire

Design smells	S_3	S_6	S_4	S_5	S_1	S_2	S_7	S_9	S_{10}	S_8	S_{11}
Unutilized Abstraction	164	69	96	56	99	341	884	104	114	56	89
Insufficient Modularization	7	9	28	3	22	38	189	62	48	41	43
Broken Hierarchy	49	31	24	16	21	41	47	6	4	0	0
Deficient Encapsulation	39	33	49	19	24	62	18	21	29	16	27
Cyclic-Dependent Modularization	23	0	17	2	5	102	252	34	30	27	33
Unnecessary Abstraction	4	3	5	8	0	26	76	15	11	14	12
Multifaceted Abstraction	3	1	3	1	2	1	18	20	13	11	17
Wide Hierarchy	3	3	7	2	6	4	3	1	0	0	0
Missing Hierarchy	0	0	1	0	1	3	0	2	0	0	0

Table 3. List of considered design smells

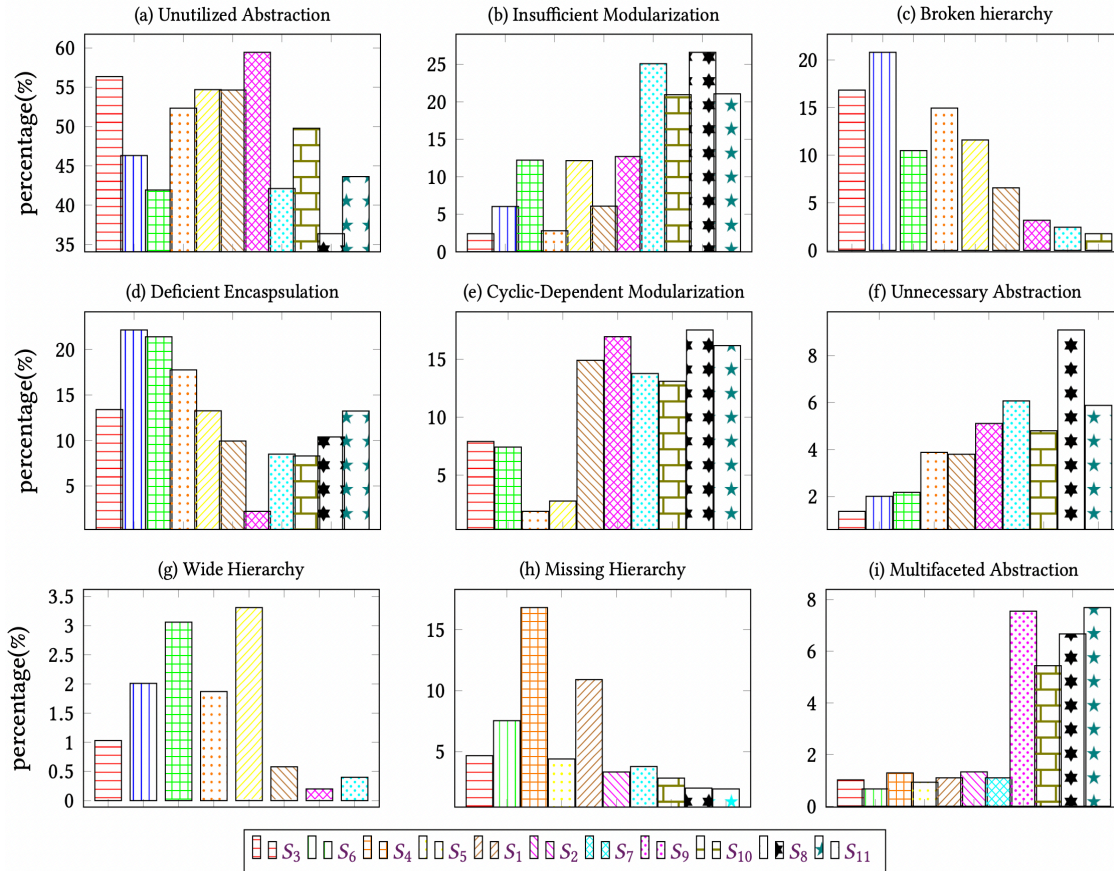


Figure 3. Percentage (%) of occurrence of specific design smells to the total number of smells

which constituted of 12 questions and required 24.5 minutes to complete on average. We asked whether the design smells which they confirmed as successfully detected by our tool were caused by two software engineering malpractices namely "irregular team communication" or "scope creep". Our goal was to determine if solving these software engineering practices at an early stage of software development significantly impacts the occurrence of design smells.

We conducted this interview separately from the precision calculation activity and identified the mapping between occurrence of the design smell to that of the issues in software engineering practice namely *irregular team communication* and *scope creep*. The questionnaire precisely asked the respondents to relate the cause of the detected design smells to one of the two specific flaws of software design considered here. We recorded all the responses and mapped those to the two software development malpractices. We discuss the survey results in the following.

Software \ Design Smell	Unutilized Abstraction	Insufficient Modularization	Broken Hierarchy	Deficient Encapsulation	Cyclic Modularization	Unnecessary Abstraction	Multifaceted Abstraction	Wide Hierarchy	Missing Hierarchy
S_5	60 56 93.3%	4 3 75%	17 16 94.1%	19 19 100%	2 2 100%	8 8 100%	2 1 50%	3 2 66.7%	2 1 50%
S_{11}	103 89 86.4%	47 43 91.5%	0 0 100%	33 27 81.8%	38 33 86.8%	12 12 100%	18 17 94.4%	0 0 100%	0 0 100%

Table 4. Results of the design smell determination mechanism for nine types of smells. (In each row, the first line identifies the quantity of number of suspected smells detected by the tool, the second line shows the number of actual design smells (true positives), the third line shows the precision as percentage.)

Experience	4 years and above -> 2 2-4 years -> 4 1-2 years -> 9
Responsibility	Project manager -> 2 Software developer -> 8 Quality assurance -> 4 Network administrator -> 8
Demographic	North America -> 12 Europe -> 2 Latin America -> 1 Asia -> 1
Java experience	Spring -> 10 Hibernate -> 5 GWT -> 4 Grails -> 3 *some respondents have multiple expertise

Table 5. Demographic and expertise related information about the respondents

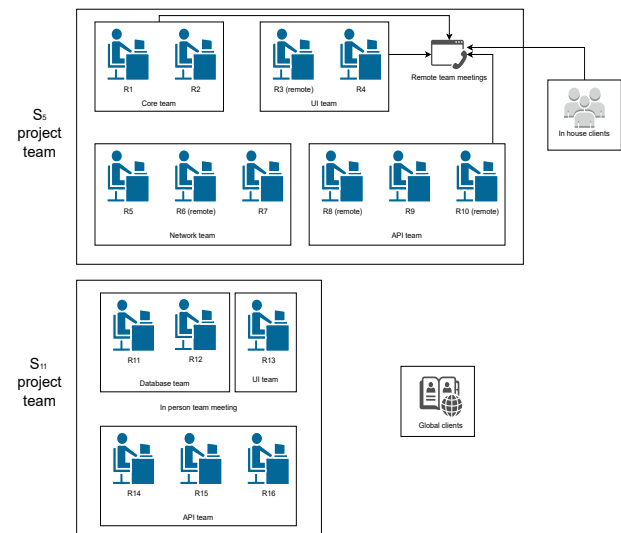


Figure 4. Project team structure for the surveyed software

4.3 Survey results

4.3.1 Irregular team communication. : The respondents related 67.60% of design smells detected by our tool to the deficiency in regular team meetings. Respondents from both companies stated that team communications were ineffectively managed, and important team discussions were not documented. One responded from S_5 stated that "although we were supposed to have pre-scheduled team meetings, the key stakeholders cancelled team meetings regularly citing conflicts with another high priority meeting, I believe this lack of regularly monitoring progress caused so many design smells in the end". Another response from the same company was such that "we had to miss some team meetings since we were handling multiple projects at one time and hence our time was divided".

Regarding project S_{11} a response to one survey question noted "at the start, we were asked to implement the software by reading white-papers and case studies of similar projects from internet, however, no meeting was held by the manager highlighting technical aspects, that contributed to the poor design". It was seen that both the organizations lacked effective and regular meetings for these two projects. Both the organizations were understaffed at the core team level and regular team meetings could not happen mainly due to the fact that the core team was managing other projects at the same time. Also, it was expected that key requirements of the design would be kept in mind by the developers hence no design documents were generated.

4.3.2 Scope creep. : Another critical problem which caused significant number of design smells was *scope creep*. It was a

phenomenon that caused substantial changes to the project scope and design during development without proper change management procedure. This caused unexpected delays and expensive change incorporation. Scope creep was highlighted as a major challenge for both the teams and they attributed that sudden introduction of new requirements made the design unmanageable and introduced design smells. Following responses obtained from the survey regarding project S_5 which highlighted this issue. One respondent from the S_5 project mentioned *"Our team was initially formed to implement a data transfer system, so it was composed of members who had expertise in networking and programming in Java. However, as days passed and we were finishing the third prototype, the management of our company wanted the software to be able to optimize data transfer via machine learning. To be honest, none of our team members had knowledge on advanced ML techniques so the design has flaws as detected by the tool."*

Another notable comment we received during survey from a member of S_{11} 's project team was that *"change to scope was inevitable, we received a new requirement which drastically changed one method and added many functionalities, it became a god method, we added many for-loops in the method for the various activities it conducted, so the tool found many cyclic dependent modularization design smells in that method. We did not expect the scope of that method to change so much, the new features introduced suddenly also took a lot of time to implement"*. Based on the above response, we highlighted that both the projects suffered from *scope creep* which introduced longer time to market and the final version had design smells. As a result, within the scope of the research we related the cause of the design smells to the two software engineering malpractices.

The outcome of the survey indicates that improved and regular team meetings are required during the design phase to prevent design smells from occurring in the final version of the software. Regular team meetings contribute to a cohesive team and help to clarify design-related questions. Also, key minutes of the team meetings need to be recorded. On the other hand, it has been found that scope creep plays an important role in introducing design smells. Sudden changes in project scope will introduce new requirements, and it is seen within the scope of the two projects analyzed here that engineers find it a challenge to comprehend the design changes in a short time. As a result, project managers need to be aware of not allowing scope creep in their projects and document a change management plan in this regard.

5 Threats to validity and credibility

We identify the common threats to validity as experienced by similar qualitative research. Generalization of the survey results beyond the scope of the sampled respondents needs to be conducted with care. We interviewed only industry experts and not the clients of the software, also we interviewed

the team related to the software we tested for design smells, the opinion may vary among teams working on different software projects in the same companies. We limited our scope to small software companies only, however, the findings may not be reflected across different sized companies at different geographic locations.

6 Related Work

A review of exiting tools to detect architectural smells is provided by Azadi et.al. [3] which groups those based on the detected smells. The authors evaluated 9 tools which are currently in place and ignored those which became obsolete. Smell definitions on which the tools detect the smells are provided and a high level detection mechanism is described by analyzing the current literature. Although the tools which detect architectural smells are provided, accuracy of the tool to aide developers regarding which smells to focus on during different stages of software life cycle was not addressed.

Model depicting the collected efforts required to detect architectural design smells is proposed based on study of related literature [4]. The model identifies negative affects, challenges, refactoring areas and relationship between each activity. The design of the model was motivated from the literature study conducted earlier in the paper. Although the model considers important aspects of design smells, it is a high level design and performance analysis of it is not provided as it was not tested on any software. Also the ability of the proposed architecture to detect smells of software was not discussed.

The affect of developer's seniority, frequency of commits and interval of commits on reducing design debts in a software was evaluated by Alfayez et al. [1]. The authors determined that seniority and frequency of commits are negatively correlated to reducing design debt, whereas interval of commits is positively correlated. The authors used multiple statistical analysis tests to validate the affects of developer behavior on the design smells.

Nahar et al. [15] studied the collaboration challenges between ML team and other teams in software projects when the software requires incorporation of an ML component. In this regard, a survey was conducted among software practitioners and the results emphasized on the importance of effective communication and team meeting for successful ML incorporation. However, the impact of *scope creep* on such collaboration have not been studied to the best of our knowledge.

Although many software analysis tools provide information on various metrics, there is a limitation in the number of open source tools which can detect design smells [5]. *AI Reviewer* identifies code and design smells for C++ projects [3]. *Hotspot Detector* is capable of detecting only 4 design smells in Java [12]. *Designite* is a commercial software which detects 25 design smells for C# projects only [18]. *Lattix* is another

commercial tool which uses dependency matrix to detect modularity violations [20]. Other commercial tools such as *Structure101*, *Sonargraph* and *Cast* detect cyclic dependency smells [17]. To the best of our knowledge, although there are tools to detect code smells, those which detect design smells for Java are limited to detection of 4 or less smells.

Imran et al. [10] identified the impact of human factors on software sustainability. The authors conducted a survey to show the challenges which software engineers face when incorporating sustainability into their software. They identified human factors which highly impact software sustainability such as *leadership*, *communication*, *peer pressure*, and *acknowledgement of efforts*. However, the impact of *irregular team meetings* was not considered. Additionally, how the human factors impact the design of the software was not studied.

7 Conclusions and Future Work

This paper detected design smells using an *Abstract Syntax Tree (AST)* based tool. Next, it identified specific design qualities and set up relationships between violation of those and occurrence of design smells. After testing the tool on the large volume of LoC and class files of the Java projects analyzed here, the following conclusions were drawn.

- The tool was required for correct and time-friendly detection of design smells.
- Software engineers could focus on the challenges they faced in terms of team communication and change management of scope to prevent design smells from occurring.

In the future, it may also be helpful to perform a longitudinal study that detects how the occurrence of design smells in a program suite is affected by other software design challenges not identified here. A minor but still interesting point is that the pseudo-model tool can be extended to include a new layer showing how design smells change as the software matures.

Acknowledgments

This project is in part sponsored by the National Science Foundation (NSF) under award numbers OAC-1724898, OAC-1842054 and CCF-2007829.

References

- [1] Reem Alfayez, Pooyan Behnamghader, Kamonphop Srisopha, and Barry Boehm. 2018. An Exploratory Study on the Influence of Developers in Technical Debt. In *Proceedings of the 2018 International Conference on Technical Debt* (Gothenburg, Sweden) (*TechDebt '18*). ACM, New York, NY, USA, 1–10. <https://doi.org/10.1145/3194164.3194165>
- [2] Mauricio Aniche, Joseph Yoder, and Fabio Kon. 2019. Current challenges in practical object-oriented software design. In *2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. ACM, 113–116.
- [3] Umberto Azadi, Francesca Arcelli Fontana, and Davide Taibi. 2019. Architectural Smells Detected by Tools: a Catalogue Proposal. In *International Conference on Technical Debt (TechDebt 2019)*.
- [4] Terese Besker, Antonio Martini, and Jan Bosch. 2016. A systematic literature review and a unified model of ATD. In *2016 42th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, 189–197.
- [5] Andrea Biaggi, Francesca Arcelli Fontana, and Riccardo Roveda. 2018. An Architectural Smells Detection Tool for C and C++ Projects. In *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, 417–420.
- [6] Jorge Andrés Díaz-Pace, Antonela Tommasel, and Daniela Godoy. 2018. Towards Anticipation of Architectural Smells using Link Prediction Techniques. In *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 62–71.
- [7] William Bruce Frakes and Ricardo Baeza-Yates. 1992. *Information retrieval: Data structures & algorithms*. Vol. 331. prentice Hall Englewood Cliffs, NJ.
- [8] Daniel Grunwald. 2012. *NRefactory*.
- [9] Asif Imran. 2019. Design Smell Detection and Analysis for Open Source Java Software. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 644–648. <https://doi.org/10.1109/ICSME.2019.00104>
- [10] Asif Imran and Tefvik Kosar. 2021. *The Impact of Human Factors on Software Sustainability*. Springer International Publishing, Cham, 287–300. https://doi.org/10.1007/978-3-030-69970-3_12
- [11] Capers Jones. 2012. *Software Quality in 2012: A Survey of the state of the art*. Retrieved May 3, 2019 from <http://stal.blogspot.com/2008/02/removing-unnecessary-abstractions.html>
- [12] Ran Mo, Yuanfang Cai, Rick Kazman, and Lu Xiao. 2015. Hotspot patterns: The formal definition and automatic detection of architecture smells. In *2015 12th Working IEEE/IFIP Conference on Software Architecture*. IEEE, 51–60.
- [13] Naoel Moha, Yann-Gael Gueheneuc, Laurence Duchien, and Anne-Francoise Le Meur. 2009. Decor: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering* 36, 1 (2009), 20–36.
- [14] Mehdi Mohammadi. 2014. *Janett*. Retrieved April 13, 2019 from <https://github.com/mehdimoh/janett>
- [15] Nadia Nahar, Shurui Zhou, Grace Lewis, and Christian Kästner. 2022. Collaboration Challenges in Building ML-Enabled Systems: Communication, Documentation, Engineering, and Process. In *Proceedings of the 44th International Conference on Software Engineering (Pittsburgh, Pennsylvania) (ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 413–425. <https://doi.org/10.1145/3510003.3510209>
- [16] Reinhold Plösch, Johannes Bräuer, Matthias Saft, and Christian Körner. 2018. Design debt prioritization: a design best practice-based approach. In *2018 IEEE/ACM International Conference on Technical Debt (TechDebt)*. IEEE, 95–104.
- [17] R Roveda. 2018. Identifying and evaluating software architecture erosion (Doctoral dissertation). (2018). https://boa.unimib.it/retrieve/handle/10281/199005/287440/phd_unimib_723299.pdf
- [18] T. Sharma, P. Mishra, and R. Tiwari. 2016. Designite - A Software Design Quality Assessment Tool. In *2016 IEEE/ACM 1st International Workshop on Bringing Architectural Design Thinking Into Developers' Daily Activities (BRIDGE)*. 1–4. <https://doi.org/10.1109/Bridge.2016.009>
- [19] Girish Suryanarayana, Ganesh Samarthayam, and Tushar Sharma. 2014. *Refactoring for software design smells: managing technical debt*. Morgan Kaufmann.
- [20] Sunny Wong, Yuanfang Cai, Miryung Kim, and Michael Dalton. 2011. Detecting software modularity violations. In *2011 33rd International Conference on Software Engineering (ICSE)*. IEEE, 411–420.