Towards a Catalog of Enterprise Architecture Smells

Johannes Salentin[1], and Simon Hacks[2]

[1] RWTH Aachen University, Aachen, Germany;
[2] KTH Royal Institute of Technology, Division of Network and Systems Engineering, Stockholm, Sweden
johannes.salentin@rwth-aachen.de, shacks@kth.se

Abstract. Code Smells are well known in the domain of Technical Debt (TD). They hint at common bad habits that impair the quality of the software system. By detecting those smells it is possible to suggest a better solution or, at least, make the developers aware of possible drawbacks. However, in terms of Enterprise Architecture (EA), which is a more holistic view of an enterprise including TD, there does not exist such a concept of EA Smells.
Such EA Smells can be a component of EA Debt, working like a metric to rate the quality of data and estimate parts of the EA Debt in an EA Repository. The main goal of this work is to start the development of a catalog to facilitate future design and development of EAs. This catalog should be expanded and serve as food for thought to create a corresponding tool for the detection of smells.

Keywords: EA Smells, EA Quality, EA Debt, Prototype, Catalog.

1 Introduction

In the software development industry, Technical Debt is regarded as a critical issue in terms of the negative consequences such as increased software development cost, low product quality, decreased maintainability, and slowed progress to the long-term success of developing software [1, 2]. Technical Debt describes the delayed technical development activities for getting short-term payoffs such as a timely release of a specific software [3]. Seaman et al. [4] described Technical Debt as a situation in which software developers accept compromises in one dimension to meet an urgent demand in another dimension and eventually resulted in higher costs to restore the health of the system in future.

Furthermore, Technical Debt is explained as the effect of immature software artifacts, which requires extra effort on software maintenance in the future [5]. While the Technical Debt metaphor has further extended to include e.g. database design debt, which describes the immature database design decisions [6], the context of Technical Debt is still limited to the technological aspects.

Therefore, we have proposed to combine the concepts of EA (Enterprise Architecture), which provides a holistic view on an organization, and the concept of Technical Debt [7]. However, our definition still lacks a means to identify possible debts. To close this gap, we suggest transferring the concept of Code Smells to the

15th International Conference on Wirtschaftsinformatik,
March 08-11, 2020, Potsdam, Germany

domain EA Debts. Transferring a concept from the software architecture domain to the EA domain seems obvious, as they share some terminology, concerns, and notations [8]. Nonetheless, they focus on different domains (business vs. system) and, consequently, have fundamentally different stakeholders with different perspectives and viewpoints [8]. Accordingly, we will elaborate on aspects both domains have in common, especially structural aspects.

The main goal of EA Smells and their automated detection is to increase the quality of EAs and the awareness for common bad habits. Therefore, EA Smells should reflect quality flaws in the EA itself. However, if the EA model is used as an input to determine the EA Smell, the smell might only appear in the model and not in reality, based on the on the degree of how the model reflects the reality. Further, EA Smells can be considered in a prescriptive or descriptive manner. For prescriptive purposes different scenarios of evolution can be compared to each other [9]. In a descriptive manner, the actual status of the EA can be assessed and its development over time can be tracked. With a catalog of EA Smells, developers can rely on it during the design process or even afterwards while doing performance monitoring to assess the quality of data in EA repositories. Sophisticated tools for automated detection can then provide a convenient risk detection with built-in solution suggestions.

EA Smells can be a component of EA Debt, working like a metric to rate the quality of data and estimate parts of the EA Debt in an EA Repository. This represents a valuable usage of the EA Debt concept, even though EA Smells are just a subset of the overall EA Debt, which will influence the popularity of those ideas in a positive way, resulting in encouragement. For example, such an EA Smell could be a bloated service that has one large interface with many parameters or requires much data and performs mostly heterogeneous operations with low cohesion

In order to define EA Smells and develop a first program for automated detection, there are some steps, which must be considered. Our research questions will guide the process of finding suitable answers to the open problems and structure the course of action:

What corresponding EA Smells can be defined based on the existing Code Smells and anti-patterns?

Then EA Smells can be defined in a catalog based on their counterparts in Code Smells and a procedure is elaborated to detect some smells automatically in representative EA repositories.

The rest of this work is structured as follows: First, we sketch the applied research design. Next, we present the basic concepts that we will use to create a catalog of EA Smells, namely Code Smells and EA Debt. Afterwards, we introduce our EA Smells catalog by presenting the overall representation and giving some exemplary EA Smells we discovered. Those EA Smells serve as input for our prototype that allows identifying EA Smells in ArchiMate models. Before we conclude our work, we give an overview regarding related work.

2 Research Design

This work follows the research questions and the proposed methodologies of Peffers et al. [10] and Hevner et al. [11] of DSR (Design Science Research): DSR aims to produce an artifact that addresses a problem [11]. Further, the artifact should be relevant to the solution of an "heretofore unsolved and important business problem" [11]. Its "utility, quality, and efficacy" [11] must be rigorously evaluated. The research should represent a verifiable contribution and rigor must be applied in both the development of the artifact and its evaluation. The development of the artifact should be a process that draws from existing theories and knowledge to come up with a solution to a defined problem. Finally, the research must be effectively communicated to appropriate audiences [10, 11].

The problem is already identified and motivated Section 1. Based on this, we define our objective of creating a first set of EA Smells that can assess structural aspects of EA quality based on its model. Further, this set shall serve as basis for discussions with EA practitioners for the future development of EA Smells. To get a first set of EA Smells, we identify popular Code Smells and transform and adapted them to the domain of EA. For that purpose, a suitable representation for smells must be chosen (cf. Section 4.1). Eventually, some remarks to the validity and possible risks to this very validity are justified after the definition of EA Smells (cf. Section 4.2).

Section 5 deals with the challenges and requirements for the program that is to be drafted as well as some considerations on the actual implementation. The resulting program is tested on some examples to demonstrate the functionality of said program. Due to the early stage of EA Debt research, we do not conduct a comprehensive evaluation of our artifact.

Finally, the whole research and work is communicated through this work, also showing potential elements for future research in Section 7. The full resulting catalog of EA Smells[1] can be found in online repositories with a searchable web application.

3 Basic Concepts

3.1 Code Smells

Code Smells, introduced by Fowler and Beck as "certain structures in the code that suggest (sometimes they scream for) the possibility of refactoring" [12], are common bad habits that can impair the quality of a software system or the software architecture, when they are ignored. Like anti-patterns, they are seemingly good solutions that are commonly and repeatedly used but known not to provide any satisfactory results. Thus, corresponding symptoms, root causes and consequences along with solution suggestions should be specified [13].

They are a component of Technical Debt and by identifying them, one can determine how to refactor certain software artifacts. However, Fowler claims that no set of precise

[1] https://ba-ea-smells.pages.rwth-aachen.de/ea-smells/

metrics can be given to discover the need for refactoring. Code Smells are a compromise between precise source code metrics and unguided subjective evaluation [12, 14, 15]. Still they can indicate deeper problems, while they can also amplify other smells [16]. Often they are also called “bad smells” or “anti-patterns”. In many cases unnoticed, Code Smells cause problems, sometimes only recognized after several further implementations rest upon that faulty code. This harms the process of refactoring and makes this process very complex and extensive. Often categories of Business, Architecture and Application smells or patterns are distinguished [17–19].

In contrast to design pattern, which recommend a specific way of solving a problem with positive examples [20], anti-pattern and Code Smells identify suboptimal solutions and risks. However, an excessive use of design patterns can also cause bad smells, as well as the process of refactoring can involve the potential of introducing new ones when done wrong. Nevertheless, there should be some solution to the cause suggested along with the negative example, which maybe was not covered by some design pattern.

Along with those smells comes something like software evolvability or maintainability. The IEEE Standard Glossary of Software Engineering Terminology defined software maintainability as “the ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or to adapt to a changed environment” [21]. Mäntylä and Lassenius adapted this term to software evolvability, which refers to “the ease of further developing a software element” [14]. This has a close match with the term perfective software maintenance [21], which is defined as “software maintenance performed to improve the performance, maintainability, or other attributes of a computer program” [21].

3.2 EA Debt

EA Debt is defined as the counterpart to Technical Debt in the domain of EA. It was defined as:

> *“Enterprise Architecture Debt is a metric that depicts the deviation of the currently present state of an enterprise from a hypothetical ideal state.” [7]*

In that sense, also the organizational position and the principles governing its design and evolution must be considered. This includes the different layers, respectively domains, of Enterprise Architecture and its associated artifacts. Since the requirements differ for each enterprise, there does not exist a uniform approach according to EA models and quality issues. Therefore, the metaphor of EA Debt is a technique providing some crucial factors for the estimation of an EA's quality. Once again, increased awareness is a major benefit of the debt concept in combination with a common basis for communication and discussion.

In order to support many different cases flexibly, the metaphor relies on the artifact-based viewpoint of an EA, because especially in dynamically evolving environments robustness is very important. In this way, further artifacts can be added or removed for the purpose of adaption to the specific requirements [7].

4 Related Work

Following, we will illustrate related work and situate our research towards it. First, we refer to studies that occupy the analysis of the quality of EAs, or their models respectively, and its improvement. As, model quality is not only related to the domain of EA, we refer, second, to work from other information system disciplines, transferring the concept of smells.

A very popular framework for determining the success of information systems is The DeLone and McLean Model of Information Systems Success, most recently updated in 2003 [22]. This was adapted specifically to the domain of EA by Lange et al. depicting EA product quality, EA function setup quality, EA service delivery, and EA cultural aspects as components influencing the satisfaction and the intention of use [23]. However, these works elaborate on different aspects and their influence on the EA. Therefore, those approaches stay abstract and their relevance for practice is still limited. By using the concept of debts and expressing the quality flaws by a concrete number, we overcome this shortcoming and provide a more useful toolset for practice.

An Enterprise Architecture Model Quality Framework (EAQF) was developed by Timm et al. that structures relevant information and helps enterprise architects to reflect on their EA models [24]. This framework builds upon six principles to assess an enterprise model's quality by Becker et al., namely the principles of validity, relevance, economic efficiency, clarity, systematic model structure and comparability [25]. Furthermore, Pitschke provides a list of quality attributes and explains them in relation to business process models [26]. Compared to the model of DeLone And McLean, the beforehand introduced approaches are less abstract and might be applicable in praxis. However, they fail to make quality flaws quantitative measurable and rely mostly on interviews. Our EA Smells provide a further set of quantitative metrics that enable the (semi)automatic assessment of EA (model) quality.

In general, the overall EA quality does not necessarily relate to EA models as an artifact directly, but also to EA management processes and other services [24]. As a common sense, an EA model's quality has to be evaluated with respect to its purpose, environment and stakeholders' concerns [25]. This facet will be also included in our future research by suggesting EA Smells that cover such aspects. Smells that do not cover these aspects are not included so far, because the initial considered set of Code Smells does not include processual facets.

We will not situate our research on EA Debt and EA Smells on the green field, but also consider existing approaches for measuring EA related qualities. This includes e.g. research on anti-patterns like process anti-patterns [27]. They can be understood as a certain type of smell on the different layers of EA. Therefore, existing anti-patterns can be understood as siblings of our EA Smells. We aim not to replace them but to reuse and integrate them into the entire framework of EA Debts.

In terms of Code Smells, there are many studies investigating and transferring the concept to other domains. Fontana et al. analyzed the frequency of Code Smells in systems of different domains, which resulted in the domains incorporating a common set of Code Smells [28]. As an example, the application of the Code Smell metaphor to the different domain of Infrastructure as Code was proposed by Sharma et al.,

distinguishing implementation and design smells [29]. While implementation smells involve quality issues like naming conventions, style, formatting and indentation, design smells involve quality issues in the module design or structure of a project. This approach is also followed by Schwarz et al. [30]. More recent research created derivations with quality models for Puppet based on an empirical study [31] or by exploiting text mining techniques [32]. Moreover, the scope was extended to, for example, Docker by Cito et al. [33] or Android manifests by Jha et al. [34]. Besides new domains, also further and more specific approaches can be considered, like the differentiation between object-oriented and service-oriented environments. Extensive SLRs were conducted by Bogner et al [35] and Sabir et al. [36], highlighting idiosyncrasies of diverging approaches.

In general, those smells that rely on structures in an architecture and their corresponding translations in other domains miss a holistic view on the entire organization. We close this gap by proposing our initial set of EA Smells that take such aspects into account. However, those existing smells can already serve as a first idea for a particular smell int the domain of EA.

5 A Catalog for EA Smells

5.1 Representation of EA Smells

Enterprise Architecture Smells is a counterpart for Code Smells. They serve as negative examples and bad habits that impair the quality of an EA when ignored. They are seemingly good solutions or typical mistakes, known not to provide any satisfactory results, such that corresponding symptoms, causes, consequences and solutions should be mentioned. EA Smells belong to the concept of EA Debt. In the following, we will focus on EA Smells belonging to EA Models, although one could develop further smells in the future, also taking the management of EAs into account. This also means that Technical Debt along with its Code Smells belong to the overall EA Debt and EA Smells respectively. Of course, Code Smells occur in the implementation and not the model of an EA.

Similarly, the need for refactoring of an EA model and the corresponding real-world instance cannot be determined precisely by any metrics. However, EA Smells are a compromise between those metrics and subjective evaluation, which allows for indication of deeper problems. Usually the specification and documentation of patterns relies on a common format or scheme with well-known attributes like the Alexandrian Form [37]. This does not hold for antipatterns and Code Smells, which can profit from further attributes that are not mentioned in every representation. Based on some proposed forms of representations commonalities are searched, despite slight differences. It turns out, that the chosen data structure from Bogner et al. [35] encompasses all attributes of considered sources.

For the sake of documentation the *name*, a *description* and optional further *context* should be specified [16, 35, 38, 39]. Additionally, similar smells could be known under

multiple names, which justifies the *aliases* attribute and allows for a common understanding, even though something is named differently [16, 35, 39].

It is useful to point out the *consequences* of a smell together with a possible *cause* and opportunities for its *detection*, when those are not obvious. This way, a more applicable *solution* suggestion or other remarks can be made [16, 35, 38, 39]. With the help of an *example* rather complex problems can be illustrated and demonstrated for better understandability [16, 35, 39]. Wake [38], for example, even includes exercises to increase the comprehension of smells.

By means of *tags* additional categorization can be achieved, that does not need to follow a certain structure. A tag is added for each category or property that we want to emphasize [35, 39]. In order to relate smells to each other and expose their impact or even amplification, the attribute *relatedItems* exists. Thereby, for example, the relations "follows", "precedes" or simply "relates" can be expressed [35]. Sometimes this is not done explicitly, but it would facilitate to reveal the dependencies [16, 39]. Although the EA Smells will be introduced in the following section, we already want to provide a place for the *sources* (cf. [16, 35]) as evidence and for the purpose of transparency. This enables future development of the catalog and the repository, without losing track of occurrences of smells. Since the documentation varies for each source having slightly different attributes or not every attribute being specified, many attributes are only optional (cf. [16, 20, 35]). Still the *name*, *description,* and the *solution* should be specified for each smell.

5.2 Exemplary EA Smells

This section explains the process of translating existing Code Smells to EA Smells on some examples. To compile a catalog of EA Smells, we rely on already established Code Smells and transfer them to the domain of EA Debt. Therefore, we collected 56 Code Smells collected in several SLR (Systematic Literature Reviews) [14, 35, 36, 40]. In total we end up with 45 EA Smells in our final catalog. We created an EA Smell repository that contains further information, such as descriptions, details, relations and associated tags. This repository can be viewed as a web application with convenient browsing and searching[2].

A smell can be directly applicable to the new domain (e.g. *Weakened Modularity*) or it may require adaptions to the description to match the according terms and concepts (e.g. *Incomplete Node or Collaboration*). There are also smells that need a broad modification to fit to the new domain and yield their advantages (e.g. *Overgeneralization*), or those that combine the approaches of multiple existing smells (e.g. *Missing Abstraction*). Besides, a new EA Smell can be inspired by a smell, without having a direct ancestor in Code Smells (e.g. *Temporary Solution*).

Weakened Modularity. The smell *Weakened Modularity* is easy to translate to EA, because the original Code Smell simply states that each module must strive for high cohesion and low coupling. This idea is one of the most important design principles and, therefore, directly applicable to the new domain of EA.

[2] https://ba-ea-smells.pages.rwth-aachen.de/ea-smells/

Only the detection of this smell needs some alterations: Each cluster of an element, which contains the element itself and all successive sub-elements, is investigated separately and a modularity ratio is calculated by dividing the number intra-cluster references (cohesion) and the number of inter-cluster references (coupling). Although the basic approach of detecting the smell is not changed, it requires an additional check for a minimal internal relation count, because otherwise every component without internal relations (e.g. a simple service) would be reported as a smell. A further difficulty lies in finding appropriate thresholds for the modularity ratio, which can be dependent on the environment and specific use case. In general, this topic of thresholds of metrics should be inspected in future research.

Incomplete Node or Collaboration. The smell *Incomplete Node or Collaboration* is derived from *Incomplete Library Class*. Since in EA models there does not exist a library class, this idea must be transformed to nodes or collaborations that represent self-contained encompassing components that model, for example, application platforms or application collaborations. However, these components do not fulfill their responsibilities.

Those groups can be handled like libraries in software systems, as they can provide stand-alone functionality that is integrated by other elements or services. If those components come from a vendor, the functionality cannot be changed or extended by oneself to meet the present requirements, leading to more extensive changes in the current structure.

Overgeneralization. The smell *Overgeneralization* is derived from *Ambiguous Interface*, which describes an implementation-wise interface that is too general and does not publish all its provided functionality in a clear manner. For the purpose of fitting better to EA models, the smell is converted to the idea of overgeneralization: In order to assure that components provide the greatest extent of reusability, they must be flexibly applicable. This generalization can be overdone, which will result in overgeneralization. It will become more flexible than it needs to be.

Since not all functionality is provided and the operation is very general, all users of that component must do quite similar additional work to use the component for its purpose. As a solution, the always identical user modifications should be moved into the component, following the “Once and Only Once” principle. Otherwise, the single generic operation can be split into multiple more specialized operations to allow for more suitable applications.

Missing Abstraction. The smell *Missing Abstraction* is derived from two existing Code Smells. Firstly, *Functional Decomposition* describes the result of making every subroutine a class in an object-oriented environment, ignoring class hierarchy altogether. Secondly, *Primitive Obsession* describes the use of primitives instead of small objects for simple tasks or data clumps.

In general, the smells depict the absence of a valuable abstraction. Hence, the EA Smell Missing Abstraction is created. It is present when many rather small components exist that are not aggregated into encompassing super elements. The EA model should give a more abstract insight and, thus, provide components that are more general. Inside those abstractions, further elements can be introduced to enable extensive description of the architecture.

For example, instead of modelling contracts and the related customers together, only separate data objects exist that clutter the model. Alternatively, a complex business function is not directly modeled, but only by many sub-elements without them being aggregated into an overall Business Function.

Temporary Solution. The smell *Temporary Solution* is inspired by *Temporary Field*. Objects sometimes contain fields that do not seem to be needed at all the time. The rest of the time, the field is empty or contains irrelevant data, which is difficult to understand. Oftentimes, temporary fields are created for use in an algorithm that requires a large amount of inputs. Therefore, instead of creating a large number of parameters in the method, the programmer decides to create fields for this data in the class.

The idea of temporary information, that would be unnecessary in a better-organized structure, is then adopted to EA models with temporary solutions. Some components are introduced, in order to make a relatively fast adaption such that the established environment is working, although this temporary solution does not represent the desired or intended solution. This is the very common view of debt being taken. Oftentimes, those adaptions are made "on the fly", without a full model being designed. In order to keep the architecture working, adjustments must be made shortly. Those adjustments are no optimal solutions resulting in cluttered abstractions and messed up architecture.

As an example, the EA is currently relying on a system A. In the long run a system C should be used to realize the roadmap, but as the integration of that system is delayed a temporary solution B has to be integrated, because data from system A is still used. Such actions increase the dependency and hinder the intended development.

6 Demonstration

In the following, we will discuss different examples to explain and demonstrate the behavior of the developed program. This should help to understand, on the one hand, some weaknesses and, on the other hand, the potential of automated detection. The used models are accessible through the repository of the program[3]. The program itself is developed in Java and takes XML files as input that are in line with the ArchiMate Exchange Format [41]. The program consists of three main components: a package that handles the EA model, a package that provides different classes to detect different EA Smells, and a central control component that brings together the abstract EA Smells with the concrete EA model.

The *SmellExample.xml* is a made-up generic example that contains all currently detectable EA Smells, which means that at least one instance of each smell should be reported. It contains 47 ArchiMate elements with 88 relationships and was constructed mainly for testing the detection of those smells. Again, each one of them will be addressed in the following with an excerpt of the underlying model. Thereby, the smells are distributed to three main components of the example model, such that we will consider one component after the other.

[3] https://git.rwth-aachen.de/ba-ea-smells/program

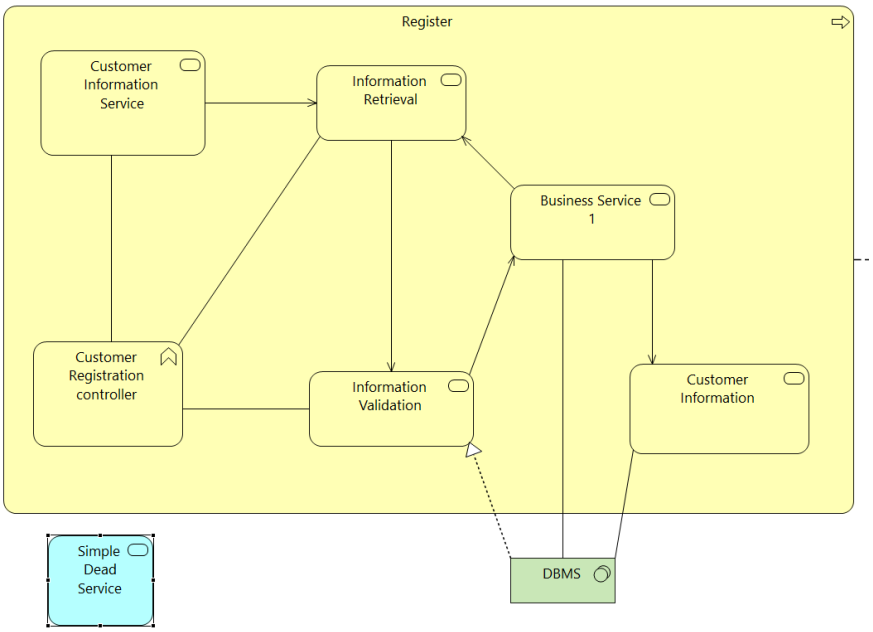

Figure 1. The "Register" component containing many EA Smells

First, no viewpoint is specified resulting in the *Ambiguous Viewpoint* smell at the view "Default View". Furthermore, an overall *Dense Structure* is detected with an average degree of around 1.87. For more detailed information the processes "Register" (average degree of 2.0) and "Complex Process with long Documentation" (average degree of around 2.29) are detected as *Dense Structures*, so that these components clutter the design. A visual impression for those *Dense Structures* can be found in Figure 1 and Figure 2 respectively.

Figure 1 contains a *Cyclic Dependency* of three business services "Information Retrieval", "Information Validation" and "Business Service 1". Additionally, a *Service Chain* as part of a *Message Chain* of length five exists, that also contains those elements. Besides, the business services "Customer Information Service" and "Customer Information" are named quite similar and could contain related functionality. Therefore, a *Duplication* is reported. Also based on the naming, the "Customer Registration controller" arouses interest, because it very likely only delegates tasks as a *Lazy Component*. Concerning the "DBMS" there are three direct relations to components of the business layer that have access to the DBMS, resulting in both *Shared Persistency* and *Strict Layers Violation*. Lastly, the "Simple Dead Service" is easily classified as a *Dead Component* as it is not referenced.

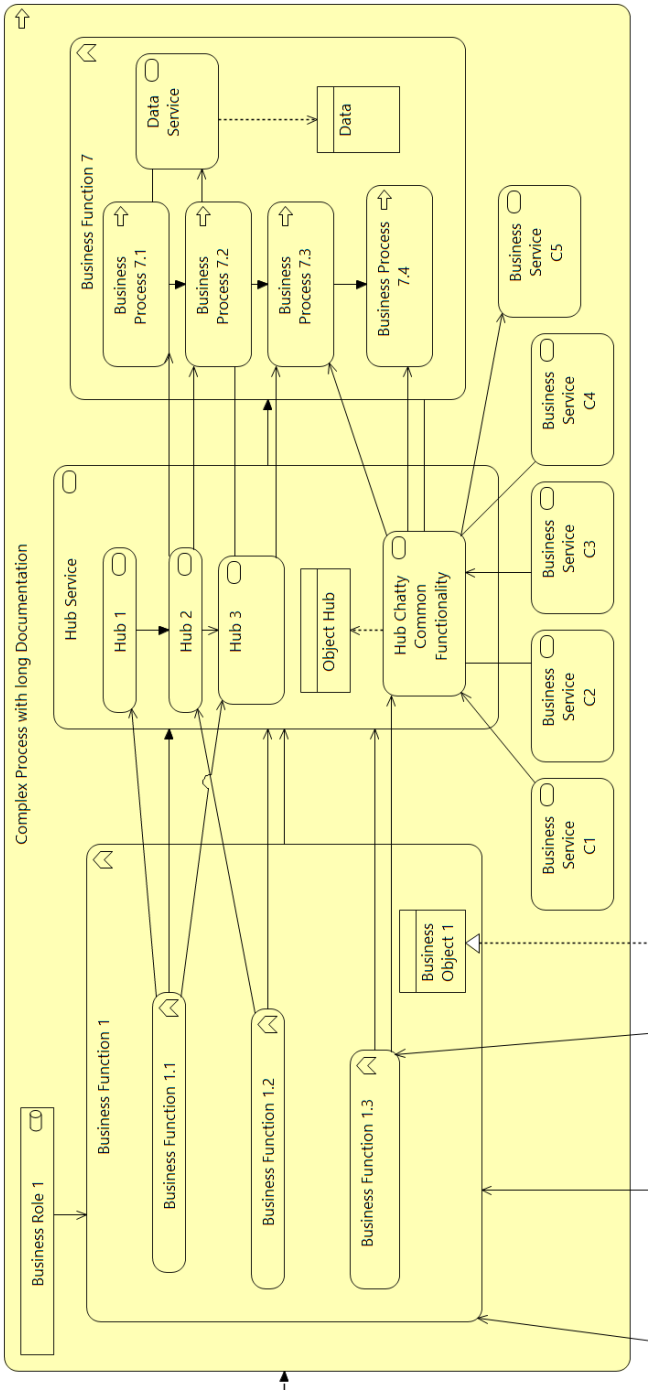

Figure 2. The “Complex Process with long Documentation” containing many EA Smells

Next, we consider the cluster of the “Complex Process with long Documentation” with a very long (423 characters) and unnecessary *Documentation* in Figure 2: When a service is related to a high number of other services, this may indicate a *Chatty Service*. Thus, the “Hub Chatty Common Functionality” is reported for further investigation. If, on the other hand, a service only references data objects, like a business object, without relating to other components, it is likely a *Data Service* only managing data access. Consequently, the “Data Service” is detected as a possible instance of that smell. Moreover, the “Hub Service” has many incoming and outgoing relations without containing many coherent services. As a result, this element is reported for *Hub-like Modularization* and *Weakened Modularity*.

Although the current program has some limitations regarding the detection of EA Smells, it still manages to give hints and warnings for various smells. This leads to increased awareness and makes the architects rethink and check their design. The provided feedback can encourage and motivate improvements in design of the model, so that the architects strive for better metric values and overall less detected smells. Those aspirations not necessarily represent the best approach to better model quality, but nevertheless can lead to developers making an extra effort, investing more time in well thought out solutions.

Of course, the manual human evaluation of a model is not replaced, but the program can easily suggest some aspects that may need further attention. Both processes combined can create a synergy that only yields advantages for the quality of an EA model. Running the program automatically needs no effort at all and does not negatively affect the development. Contrary, one additional process checking the quality may reveal overlooked shortcomings, that otherwise would be accepted inadvertently.

7 Conclusion

Based on well-known Code Smells in the domain of Technical Debt, a corresponding concept of EA Debt was defined lately that is now extended with EA Smells. It provides a promising approach to facilitate the design and improve the quality of EA Models, such that benefits of Code Smells can be transferred to the more holistic cause of EA. This includes increased awareness and in general easier processes of discovering deficits in model's designs.

First, a corresponding EA Smell Catalog was developed, where the smells are inspired and derived from existing Code Smells. Especially, the detection of those smells caused additional effort, because a program for automatic detection should be developed. Consequently, an EA Smell Catalogue is available for collaborative future work as a web. Third, the mentioned program was drafted and currently supports the detection of 14 EA Smells in EA Models compliant to The Open Group ArchiMate Model Exchange File Format Standard. In a last step, the emerged program was evaluated and explained in order to highlight the possibilities and advantages of an automated detection. Of course, there are also limitations to this process.

Although the definition of EA Smells for EA models is a step in the right direction, this concept can be expanded to smells for EA Management, addressing additional problems that may occur in an enterprise life cycle. Besides, suggested refactorings could be collected in a catalog as well, immediately providing a mapping to related EA Smells. Further, architecture principles are often perceived as the cornerstones of EA [42]. They provide design and representation rules and guidelines, which need to be translated into evaluation criteria [43]. EA Smells can serve as such evaluation criteria and, consequently, strengthen the implementation of architecture principles within the organization.

In terms of the program, a more sophisticated and universal detection framework can be designed, that can be integrated into Continuous Integration pipelines [44]. Furthermore, different approaches for detection can be tested, like genetic or evolutionary algorithms. However, we do not believe that all EA Smells can detected fully automatically. Therefore, tool-support for assessing EA Smells by means of interviews of experts will be necessary, too.

Lastly, we did not evaluate our EA Smells so far. In future, we plan first, to conduct case studies with different organizations, where we will check if our first set of EA Smells is applicable and if there are possibly other kinds of EA Smells that should be considered. Next, we will conduct interviews with different experts in the domain of EA to get further insights on the EA Smells. Last, we will conduct empirical studies to evaluate EA Smells in general and research their interrelations.

References

1. Tom, E., Aurum, A., Vidgen, R.: An exploration of technical debt. Journal of Systems and Software 86, 1498–1516 (2013)
2. Cunningham, W.: The WyCash portfolio management system. ACM SIGPLAN OOPS Messenger 4, 29–30 (1993)
3. Zazworka, N., Seaman, C., Shull, F.: Prioritizing Design Debt Investment Opportunities. Proceeding of the 2nd workshop on Managing technical debt - MTD ’11, 39–42 (2011)
4. Seaman, C., Guo, Y., Zazworka, N., Shull, F., Izurieta, C., Cai, Y., Vetrò Antonio: Using technical debt data in decision making: Potential decision approaches. In: 3rd International Workshop on Managing Technical Debt, MTD 2012 - Proceedings, pp. 45–48 (2012)
5. Guo, Y., Seaman, C.: A portfolio approach to technical debt management. In: Proceeding of the 2nd working on Managing technical debt - MTD ’11, p. 31 (2011)
6. Albarak, M., Bahsoon, R.: Prioritizing technical debt in database normalization using portfolio theory and data quality metrics. In: Proceedings of the 2018 International Conference on Technical Debt - TechDebt ’18, pp. 31–40 (2018)
7. Hacks, S., Höfert, H., Salentin, J., Yeong, Y.C., Lichter, H.: Towards the Definition of Enterprise Architecture Debts. In: Proceedings of the 2019 IEEE 23rd International Enterprise Distributed Object Computing Workshop, pp. 9–16 (2019)
8. Booch, G.: Enterprise Architecture and Technical Architecture. IEEE Software 27, 96 (2010)
9. Hacks, S., Lichter, H.: A Probabilistic Enterprise Architecture Model Evolution. In: 2018 IEEE 22nd International Enterprise Distributed Object Computing Conference (EDOC), pp. 51–57 (2018)

10. Peffers, K., Tuunanen, T., Rothenberger, M.A., Chatterjee, S.: A Design Science Research Methodology for Information Systems Research. Journal of Management Information Systems 24, 45–77 (2007)
11. Hevner, A.R., March, S.T., Park, J., Ram, S.: Design science in information systems research. MIS quarterly 28, 75–105 (2004)
12. Fowler, M.: Refactoring: improving the design of existing code. Addison-Wesley Professional (2018)
13. Brown, N., Cai, Y., Guo, Y., Kazman, R., Kim, M., Kruchten, P., Lim, E., MacCormack, A., Nord, R., Ozkaya, I., et al.: Managing Technical Debt in Software-reliant Systems. In: Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research, pp. 47–52. ACM, New York, NY, USA (2010)
14. Mäntylä, M.V., Lassenius, C.: Subjective Evaluation of Software Evolvability Using Code Smells: An Empirical Study. Empirical Software Engineering 11, 395–431 (2006)
15. IEEE: IEEE Standard for Software Maintenance. IEEE Std 1219-1998, 1–56 (1998)
16. Suryanarayana, G., Samarthyam, G., Sharma, T.: Refactoring for Software Design Smells: Managing Technical Debt. Morgan Kaufmann Publishers Inc, San Francisco, CA, USA (2014)
17. Král, J., Žemlicka, M.: The Most Important Service-Oriented Antipatterns. In: International Conference on Software Engineering Advances (ICSEA 2007), p. 29 (2007)
18. Bogner, J., Boceck, T., Popp, M., Tschechlov, D., Wagner, S. and Zimmermann, A.: Service-Based Antipatterns, https://xjreb.github.io/service-based-antipatterns/
19. Lippert, M., Roock, S.: Refactoring in large software projects: performing complex restructurings successfully. John Wiley & Sons (2006)
20. Shvets, A.: Refactoring: clean your code, https://refactoring.guru/refactoring
21. IEEE: IEEE Standard Glossary of Software Engineering Terminology. IEEE Std 610.12-1990 121990, 1–84 (1990)
22. DeLone, W.H., McLean, E.R.: The DeLone and McLean model of information systems success. A ten-year update. J. Manage. Inf. Syst. 19, 9–30 (2003)
23. Lange, M., Mendling, J., Recker, J.: A Comprehensive EA Benefit Realization Model-An Exploratory Study. In: 2012 45th Hawaii International Conference on System Sciences, pp. 4230–4239 (2012)
24. Timm, F., Hacks, S., Thiede, F., Hintzpeter, D.: Towards a Quality Framework for Enterprise Architecture Models. In: Lichter, H., Anwar, T., Sunetnanta, T. (eds.) Proceedings of the 5th International Workshop on Quantitative Approaches to Software Quality (QuASoQ 2017) co-located with APSEC 2017, pp. 10–17. CEUR-WS.org (2017)
25. Becker, J., Probandt, W., Vering, O.: Grundsätze ordnungsmäßiger Modellierung. Konzeption und Praxisbeispiel für ein effizientes Prozessmanagement. Springer Berlin Heidelberg, Berlin Heidelberg (2012)
26. Pitschke, J.: Gute Modelle-Wie die Qualität von Unternehmensmodellen definiert und gemessen werden kann, https://www.enterprise-design.eu/files/images/downloads-wissen/modelqualitaet_v2.0.pdf
27. Koschmider, A., Laue, R., Fellmann, M.: Business Process Model anti-Patterns: a Bibliography and Taxonomy of published Work. In: ECIS (2019)
28. Fontana, F.A., Ferme, V., Marino, A., Walter, B., Martenka, P.: Investigating the Impact of Code Smells on System's Quality: An Empirical Study on Systems of Different Application Domains. In: 2013 IEEE International Conference on Software Maintenance, pp. 260–269 (2013)

29. Sharma, T., Fragkoulis, M., Spinellis, D.: Does Your Configuration Code Smell? In: 2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR), pp. 189–200 (2016)
30. Schwarz, J., Steffens, A., Lichter, H.: Code Smells in Infrastructure as Code. In: 2018 11th International Conference on the Quality of Information and Communications Technology (QUATIC), pp. 220–228 (2018)
31. van der Bent, E., Hage, J., Visser, J., Gousios, G.: How good is your puppet? An empirically defined and validated quality model for puppet. In: 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 164–174 (2018)
32. Rahman, A., Williams, L.: Characterizing Defective Configuration Scripts Used for Continuous Deployment. In: 2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST), pp. 34–45 (2018)
33. Cito, J., Schermann, G., Wittern, J.E., Leitner, P., Zumberi, S., Gall, H.C.: An Empirical Analysis of the Docker Container Ecosystem on GitHub. In: 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), pp. 323–333 (2017)
34. Jha, A.K., Lee, S., Lee, W.J.: Developer Mistakes in Writing Android Manifests: An Empirical Study of Configuration Errors. In: 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), pp. 25–36 (2017)
35. Bogner, J., Boceck, T., Popp, M., Tschechlov, D., Wagner, S., Zimmermann, A.: Towards a Collaborative Repository for the Documentation of Service-Based Antipatterns and Bad Smells. In: (2019)
36. Sabir, F., Palma, F., Rasool, G., Guéhéneuc, Y.-G., Moha, N.: A systematic literature review on the detection of smells and their evolution in object-oriented and service-oriented systems. Software: Practice and Experience 49, 3–39 (2019)
37. Alexander, C.: A pattern language: towns, buildings, construction. Oxford university press (1977)
38. Wake, W.C.: Refactoring Workbook. Addison-Wesley Longman Publishing Co., Inc, Boston, MA, USA (2003)
39. Shvets, A., Frey, G. and Pavlova, M.: AntiPatterns, https://sourcemaking.com/antipatterns
40. Kerievsky, J.: Refactoring to Patterns. Pearson Higher Education (2004)
41. The Open Group: ArchiMate Model Exchange File Format. Version 2 (2015)
42. Greefhorst, D., Proper, E.: Architecture principles: the cornerstones of enterprise architecture. Springer Science & Business Media (2011)
43. Stelzer, D.: Enterprise Architecture Principles: Literature Review and Research Directions. In: Dan, A., Gittler, F., Toumani, F. (eds.) Service-Oriented Computing. ICSOC/ServiceWave 2009 Workshops, pp. 12–21. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
44. Hacks, S., Steffens, A., Hansen, P., Rajashekar, N.: A Continuous Delivery Pipeline for EA Model Evolution. In: Reinhartz-Berger, I., Zdravkovic, J., Gulden, J., Schmidt, R. (eds.) Enterprise, Business-Process and Information Systems Modeling. BPMDS 2019, EMMSAD 2019, pp. 141–155. Springer International Publishing (2019)