# An Evolutionary Analysis of Software-Architecture Smells

Philipp Gnoyke
*Otto-von-Guericke University Magdeburg*
Magdeburg, Germany
philipp.gnoyke@t-online.de

Sandro Schulze
*University of Potsdam*
Potsdam, Germany
sandro.schulze@uni-potsdam.de

Jacob Krüger
*Ruhr-University Bochum*
Bochum, Germany
Jacob.Krueger@rub.de

***Abstract*—If software quality assurance is postponed or even abandoned for a software system, maintenance and evolution become harder or even impossible. One widely known symptom for the degradation of system quality are Architecture Smells (ASs), which violate fundamental principles of software design. In this paper, we present a study on the evolution of ASs as well as on how and when they foster system degradation. Thus, we provide valuable insights regarding what ASs are meaningful to assure system quality. To this end, we analyzed the evolution of three types of ASs in 14 open-source systems with a total of 485 versions. We adapted indicators used in previous studies to assess the severity of ASs (e.g., growth, lifetime), and relate ASs to technical debt as another established indicator. Our results indicate that 1) ASs remain mostly stable compared to the code size of a system, 2) certain types of ASs, such as cyclic dependencies, have a greater impact on system degradation, and 3) certain properties determine how much an AS contributes to software degradation. These findings are valuable for practitioners to identify and tackle system degeneration, as well as for researchers to scope new research on managing ASs and technical debt.**

***Index Terms*—software maintenance, software evolution, architecture smells, software quality, technical debt, empirical study**

## I. Introduction

Software degradation (a. k. a. software aging [39] or software decay [1], [22]) is a common phenomenon that can be observed for long-living software systems. Essentially, software degradation describes that evolving a system by changing its code, adding features, or fixing bugs is likely to diminish the original design and code structure [10]. As a result, the system may become less maintainable, changeable, or even reliable, and thus future evolution eventually becomes a tedious and time-consuming task [25], [37], [58]. Recently, the process of software degradation has increasingly been described by the metaphor of *Technical Debt* (TD) [13], [26], considering the aforementioned negative effects as *interest rate* to be paid for postponing important design decisions.

*Code Smells* (CSs) [16] and *Architectural Smells* (ASs) [17], [18] have been proposed as symptoms for software degradation that help to locate the aforementioned effects in a system; thus allowing developers to perform countermeasures, such as refactoring [16], [29]. For CSs, a vast amount of research on the longevity, maintenance effort, evolution, and impact on software degradation as well as TD has been conducted [11], [27], [47], [59]. Similarly, research on the detection, impact, and refactoring of ASs has recently been performed [5], [8], [17], [18], [44]. However, few studies have been concerned with the evolution of ASs and their impact on software degradation [42], [46], [48]. So, we are missing reliable empirical insights on how and when ASs contribute to software degradation, which are needed to conclude actionable recommendations.

In this paper, we contribute to understanding the evolution of ASs and their impact on software degradation, especially with respect to TD. To this end, we built on previous work by Sas et al. [48] and Roveda et al. [46], which we extended as follows. First, we present an improved technique for tracking ASs through a system's evolution. In particular, we propose the notions of *intra-version* (i.e., smell instances in a single version) and *inter-version* smells (i.e., smell instances over multiple versions). By employing our adaptations, we argue that we can more precisely distinguish smells that evolve over time, and can also reuse properties of single smell instances by aggregating them as part of inter-version smells (cf. Sec. III). Second, we present a novel technique for tracking *cyclic dependencies* [29] (CDs) by considering overlapping cycles as part of larger *supercycles*. Moreover, we take into account that cycles may be split and merged during a system's evolution, which allows for a more precise tracking of the corresponding ASs. Finally, we relate ASs with TD to reason about the impact of ASs on software degradation and their evolution.

Using our improved tracking technique, we conducted a case study on three types of ASs (i.e., cyclic, hub-like, and unstable dependencies) in 14 open-source systems encompassing 485 versions. While the case study also serves as an assessment of our technique, our main goal was to gain empirical insights to address the overarching question: **To what extent and under what circumstances do ASs contribute to software degradation?** By answering this question, we aim to provide more precise insights into the nature and severity of ASs, and thus support developers in identifying and removing ASs that actually cause software degradation.

In summary, we make the following contributions:

- We propose an improved technique for tracking ASs during system evolution (cf. Sec. III). In particular, we introduce a novel *evolution graph* for CDs that allows for a more fine-grained analysis, including the occurrences of splitting and merging of intra-version smells. Based on this graph, we propose new properties to characterize CDs.

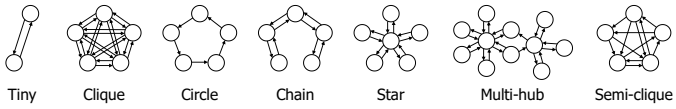


Fig. 1. Different topologies for CDs based on Al-Mutawa et al. [2].

- We relate the evolution of ASs to TD (cf. Sec. III-D). Thus, we are able to assess how and when a particular type of ASs impacts TD over time, which has not been done in previous research.
- We evaluate the previous points through a case study on 485 versions of 14 open-source systems.
- We publish an open-access repository with the implementation of our technique and our case-study results.[1]

Our findings show that our adaptations to the tracking technique help reveal novel insights, particularly that specific types of ASs seem to cause more TD (e.g., class-level CDs), and thus impair software degradation more than others. Such findings help to guide practitioners in their daily work and researchers in designing novel techniques to tackle ASs and TD during system evolution.

## II. Background

In this section, we introduce the notion of ASs with a particular focus on the smells we consider in this paper. Moreover, we briefly introduce the concept of TD.

### A. Architectural Smells

A vast number of design principles exist that aim at keeping a system vital, thus mitigating software degradation if these principles are applied properly [29], [31], [34]. Design decisions that violate these principles on an architectural level, and thus have a negative impact on software quality, are referred to as ASs [17]. In previous work, a variety of ASs has been proposed and classified [18], [28], [29], [35], [52]. For this paper, we focus on three types of these ASs: *cyclic dependency*, *hub-like dependency*, and *unstable dependency*. We chose these types of smells because they occur frequently, have been shown to negatively impact software quality, and tools are available to detect them automatically [6], [7], [21], [32], [33], [38], [45], [48]. Next, we describe these smells in greater detail.

**Cyclic Dependency (CD).** This AS describes components (e.g., classes or packages) that depend on each other, so that their dependency graph forms a cycle. In graph theory, this means that all components of a CD are strongly connected. As a result, these components exhibit a high coupling, and thus decrease testability, hinder maintenance, challenge (isolated) reuse, and violate the acyclic dependency principle [29], [52]. According to previous studies, we consider CDs on two levels of granularity: on class level (i.e., CD exists between classes) and on package level (i.e., CD exists between packages) [2], [5]. Besides their size, CDs can be distinguished by their shape. To this end, Al-Mutawa et al. [2] introduced seven commonly observed topologies, which we show in Figure 1.

[1] https://figshare.com/s/fa17e81cf4f27c84d059

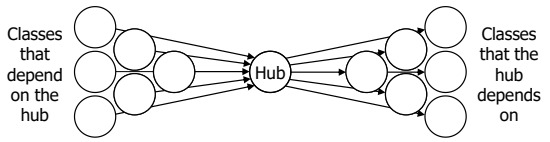


Fig. 2. A conceptual example for a HD based on Roveda et al. [45].

**Hub-Like Dependency (HD).** This AS describes a component (e.g., class, package) that exhibits a high number of incoming and outoing dependencies (i.e., edges in a dependency graph). In Figure 2, we show a conceptual example for this AS. Usually detected on class level (scope of this study), this smell indicates violations of the *few interfaces* principle [34]—which leads to a higher coupling, and thus increased effort during software evolution, for example, due to a large number of changes that must be propagated to the dependent components [52]. Moreover, previous research has shown that practitioners perceive this smell as having the most negative impact [32].

**Unstable Dependency (UD).** This AS describes a component that depends on other components that are less stable than itself. As a result, the stable component suffers from extensive change propagation (due to ripple effects from the dependent components) and increased maintenance effort [6], [35]. Different notions of stability exist, but in this paper we adopt Martin's [30] *instability metric* and focus on UDs on package level. This metric is used in AS detection tools, such as Arcan [6], which we used for our own tooling.

Martin's instability metric assumes that a component with many incoming dependencies (called *afferent coupling;* $C_a$) is considered stable (i.e., less change-prone). A component without any incoming, but potentially outgoing dependencies (called *efferent coupling;* $C_e$), is considered to be unstable, since it can be easily changed. This is also reflected by the metric's formula: $I = \frac{C_e}{C_e + C_a}$, which covers a range from 0 for maximally stable to 1 for maximally unstable. We show a simplified example in Figure 3, where according to the instability metric, component A is stable (I = 0) and component B is unstable (I = 1). Based on the instability metric, an UD can be detected by identifying a component that depends on a considerable number of other components that are *less* stable than itself. In practice, the *degree of unstable dependency (DoUD)* metric can be used to compute the ratio of outgoing unstable dependencies to all dependencies [6]. With this metric, a higher value is considered more severe, and thus indicates a greater need for removal (e.g., by refactoring).

### B. Technical Debt

Initially defined by Cunningham [13], TD is a metaphor to describe that for certain (business) reasons, such as new features or urgent adaptations, a debt is caused that negatively impacts a software system regarding its design and (code) structure [7]. This debt usually hinders maintainability and evolvability, and thus should be reimbursed regularly; for instance, by code refactoring or design adaptations [13]. If this is neglected, the TD grows and may even cause interest rates, that is, ripple effects that amplify the negative impacts, and thus lead to software erosion. TD can be distinguished by the parts of a

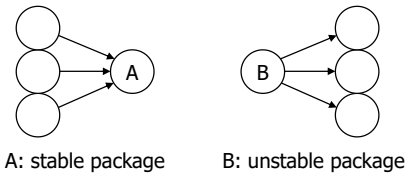


Fig. 3. Examples for stable and unstable components based on Martin [30].

system that are affected, leading to different types of TD, such as for code, design, architecture, or tests [3], [52]. In this paper, we focus on and always refer to *architectural technical debt*.

## III. Tracking and Measuring ASs

In this section, we present central concepts of the technique we implemented to conduct our case study. This includes the definition of intra- and inter-version smells, tracking strategies for the three types of ASs, and the definition of properties to measure in the case study. To summarize our contributions on tracking ASs, we compare our technique with Sas et al. [48] in Table I. Finally, we present our implementation.

### A. Intra- and Inter-Version Smells

To analyze the evolution of a system, we have to examine a series of its versions. Consequently, we can identify a set of ASs that exist in each analyzed version. We refer to ASs in a single version as *intra-version smells*. Similar to how adjacent versions usually represent gradual changes to a system, a smell can exist in several consecutive versions—either completely unchanged or evolved. Therefore, we refer to a set of intra-version smells that are related to one another and span multiple versions as *inter-version smells*.

To identify which intra-version smells constitute inter-version smells, and thus analyze their evolution, we must apply a tracking technique to all detected intra-version smells in a system. In a related study, Sas et al. [48] proposed and applied a greedy strategy to track smells based on the Jaccard set similarity of components. For this purpose, each intra-version smell was compared to each other intra-version smell of the same type of ASs in the following version. Then, the most similar pairs were iteratively matched if a predefined similarity threshold was exceeded, which resulted in one-dimensional inter-version smells. In this paper, we basically rely on the tracking technique of Sas et al., but propose improvements that aim at optimizing the runtime and achieving a more accurate tracking over time. In the following, we present the details of our improvements, separated by the type of ASs.

### B. Tracking CDs

We can analyze CDs using two strategies: First, we can define CDs as dependency graphs that represent *strongly connected components*, that is, each vertex can be reached from every other vertex with any number of edges between the vertices [33]. For instance, Al-Mutawa et al. [2] rely on this definition, and thus all graphs in Figure 1 constitute strongly connected components. Second, we can define CDs as *simple cycles*, which are cyclic paths with no repeated vertices [49].

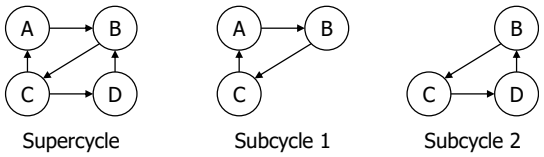


Fig. 4. Conceptual example for two subcycles that form a supercycle.

We propose to combine both definitions when analyzing CDs. So, we refer to the former case as *supercycles* and to the latter case as *subcycles*. By definition, each subcycle is part of exactly one supercycle, but supercycles can comprise an arbitrary number of subcycles. In Figure 4, we show an example of a supercycle that consists of two subcycles, which also indicates that subcycles may overlap to a certain degree. Consequently, we focus on tracking supercycle CDs, since subcycles may overlap or disappear over time. By contrast, the usually larger and more complex supercycles are unique over several versions, allowing us to track them more accurately. By analyzing the source code of Arcan [5], we found that it treats each subcycle as an individual CD instance. Given the aforementioned limitations of tracking subcycles, we recognized opportunities for improving the accuracy of our tracking technique by tracking supercycles in contrast to Sas et al. [48], who employed Arcan.

We show another phenomenon that we had to address to accurately track CDs in Figure 5: The possibility of two or more supercycles *merging* into one another, or one supercycle *splitting* into two or more distinct supercycles. Merging occurs when new dependencies are introduced between components of different CDs (i.e., supercycles). Similarly, a code restructuring that results in the removal of components or dependencies can split a CD. In Figure 5, dependencies are added (merging) or removed (splitting) between components C and E.

This phenomenon causes that CDs cannot be tracked as a sequence of intra-version smells in a simple, one-dimensional manner, because we would loose information about merging and splitting. As a solution, we propose to represent the evolution of the CDs in a system through a *CD evolution graph*, which is a layered, directed, acyclic, and not necessarily connected graph [51]. In Figure 6, we display such a graph for an exemplary system with nine components and five versions. At the top, we show the resulting CD evolution graph for the entire system, with each vertex representing an intra-version CD and an edge between two vertices indicating a relationship, namely that the same CD occurred in two consecutive versions.

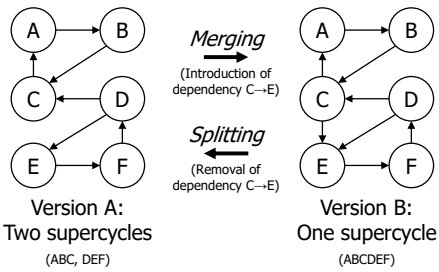


Fig. 5. Conceptual example for the merging and splitting of CDs.

TABLE I
COMPARISON OF OUR TRACKING TECHNIQUE WITH SAS ET AL. [48].

| tracking characteristic | CDs | | HDs, UDs | |
|---|---|---|---|---|
| | Sas et al. | This study | Sas et al. | This study |
| topology of tracked instances | path | graph | path | path |
| matching criterion | Jaccard | $\geq 2$ shared components | Jaccard | 1. same central component, 2. Jaccard |
| object of consideration | affected components in subcycles | affected components in supercycles | affected components | 1. central component, 2. affected components |
| determination of component identity | name comparison | name comparison | name comparison | name comparison |

At the bottom, we show the dependency graphs for each version, with each vertex representing a component.

Building on the evolution and dependency graphs, we defined our tracking technique as follows:

1) We define two CDs in consecutive versions of the evolution graph as related if they share at least two vertices in the dependency graph, since this is also the minimum number of vertices required for a CD to exist. For instance, in Figure 6, CDs A2 and B2 are related, because they share the vertices D and E in the dependency graph. To decide whether vertices are shared between CDs, we use a text-based comparison of their names.
2) We check for all pairs of CDs in consecutive versions whether they are related. As a result, we obtain one or more *connected subgraphs* (including single vertices) over all considered versions in our evolution graph that eventually constitute inter-version smells. We refer to such a subgraph as a *family*, because it may include multiple CDs of one version. Our example shows three families, encompassing one (B3), two (D3, E2), and nine (all other vertices) CDs, respectively.

In summary, the bottom part of Figure 6 represents CDs (i.e., supercycles) and the top part relations (i.e., merging, splitting) between these—as our technique would identify.

### C. Tracking HDs and UDs

Similar to Sas et al. [48], we consider the evolution of HDs and UDs as one-dimensional paths, because both types of ASs contain a central component that is essential to the respective smell (cf. Sec. II-A). For HDs, this is the hub class, while it is the stable package for UDs. However, we argue that if two intra-version smells of consecutive versions share the same central component, they constitute the same smell (i.e., they form an inter-version smell), disregarding their potential growth or shrinkage. Thus, we modify the tracking technique by introducing an additional step: We initially match all HDs and all UDs (of consecutive versions) that share the same central component. This step can be performed in log-linear time, since it only requires to sort intra-version smells by their central component and to linearly compare them. Since we observed that most versions do neither remove nor introduce large shares of ASs, this strategy increases the tracking efficiency. Additionally, false-positive and false-negative mappings are reduced, because ASs whose central component was not renamed between versions are correctly matched.

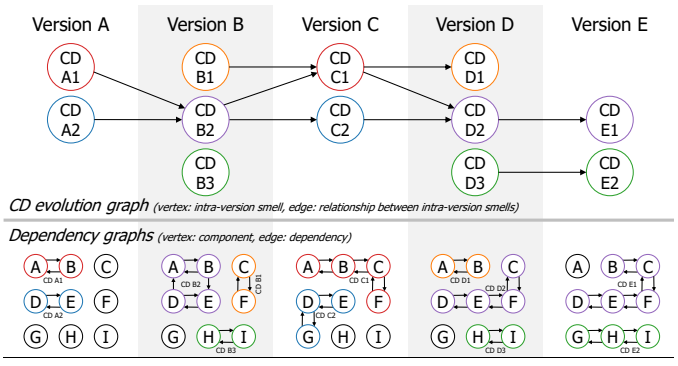


Fig. 6. Conceptual example of a system-wide CD evolution graph.

As a result, we only map the remaining ASs (for which the central component has been renamed) to each other by using the Jaccard set similarity proposed by Sas et al. For this comparison, we excluded the central component, since it would always differ between pairs of smells. Similar to Sas et al., we use a Jaccard index of at least 0.6 as the minimum threshold for any matching, which means that sets like {A, B, C, D} and {A, B, C, E} are considered to represent the same smell. Moreover, for HDs, we compute separate Jaccard indexes for the sets of afferent and efferent classes, because the same class may appear on both sides of an HD, and thus may otherwise distort our results. Afterwards, we aggregate the two Jaccard values via their weighted harmonic mean, using the respective sizes of the unions of afferent and efferent classes as weights. This reduces false-positive matching in situations in which one side of two HDs with few components is very similar, while the other side with many components is very dissimilar [48].

### D. Properties of ASs

Besides employing the tracking technique on evolving ASs, we also compute a variety of metrics related to the three types of ASs we consider. With these metrics, we can gain more detailed insights on properties and effects of the ASs, allowing us to reason on the impact and severity of each AS. In particular, we compute metrics regarding intra-version and inter-version ASs, but also for particular versions of the system. Some of the metrics have been proposed in previous research, but we defined some new metrics specifically for our study. In Table II, we show an excerpt (due to space restrictions) of these metrics with a brief explanation (cf. Sec. IV). The full list of metrics can be found in our replication package.

With intra-version smell metrics, we gain insights on properties that can help to estimate to what extent a smell manifests itself in a system, such as the size or complexity of ASs. Moreover, by considering these metrics over time (i.e., for an inter-version smell), we can derive a time series, and thus analyze the evolution of ASs regarding certain patterns (e.g., growth rate). Of particular interest for inter-version smells are

TABLE II
EXCERPT OF THE METRICS FOR OUR STUDY.

| AS | metric | description |
|---|---|---|
| | *Intra-version smells* | |
| all | order [2], [45], [48] | number of affected components (vertices) |
| all | size [48] | number of dependencies between the affected components (edges) |
| all | size overcomplexity | share of edges between affected components that are not necessary to form the smell |
| all | centrality [45], [48] | PageRank of the smell in the dependency structure (CD: highest PageRank of the affected components; HD, UD: PageRank of the central component) |
| all | overlap ratio [48] | share of components in the smell affected by at least one other smell |
| CD | shape [2], [48] | one of the seven shapes in Figure 1 or unknown |
| CD | number of subcycles | number of simple cycles in the supercycle |
| UD | DoUD (degree of unstable dependency) [45], [48] | ratio of less stable referenced packages to all referenced packages of the main package |
| UD | instability gap quartiles [48] | quartiles of the distribution of the instability difference to less stable referenced packages |
| | *Inter-version smells* | |
| all | lifespan in versions [48] | number of versions the smell is present in |
| CD | family order | total number of intra-version instances in the CD family |
| CD | median family width | median of the number of co-existing intra-version instances in the CD family |
| CD | maximum family width | maximum of the number of co-existing intra-version instances in the CD family |
| | *System versions* | |
| | TDI [45] | amount of TD in the version |
| | TD per LOC | TDI normalized by LOC |
| | #ASs | number of intra-version smells in the version |
| | #ASs per LOC | number of ASs normalized by LOC |
| | #AS introductions | number of introduced ASs in the version |
| | #AS introductions per LOC | #AS introductions normalized by LOC |
| | #AS removals | number of removed smells in the version |
| | #AS removals per LOC | #AS removals normalized by LOC |

lifespan properties, since they allow us to reason about when and why certain smells appear or disappear.

For CDs, we propose a set of new metrics to account for our tracking technique. For instance, we describe the number of co-existing intra-version CDs of the same family in a version as the *family width*. To describe the whole graph, statistical reference points like the median and maximum family width can be used. Furthermore, we refer to the total number of intra-version CDs in a family as its *family order*. If a merge or split occurs, multiple intra-version CDs of the same family exist in the same version(s). In such a case, the family order is higher than the total lifespan in terms of versions. Computing the delta of intra-version smells per version in a family (i.e., growth or shrinkage) can be misleading. For example, merging reduces the number of CDs, while not alleviating the underlying complexity. Therefore, when counting smell introductions and removals in CDs, we distinguish between 1) an entire family appearing/disappearing in a certain version and 2) one ore more intra-version smells joining in or disappearing from an already existing family.

We also compute several metrics for each version of an analyzed system (cf. Table II) that allow us to reason about smells in the whole system, independent of the type of ASs. Finally, since we wanted to study the impact of ASs on TD, we needed to quantify the latter. To this end, we adapted a metric

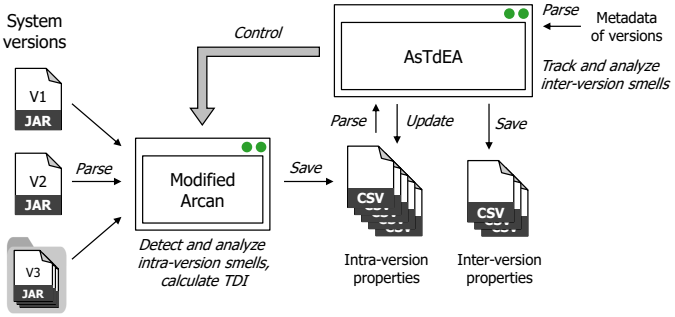


Fig. 7. Workflow of our tracking technique.

from Roveda [45]: Each smell is assigned a severity score, which depends on the type of ASs. For example, the number of less stable packages is counted for UDs, which equals the order minus 1. Then, the respective value is related to a reference dataset extracted from the Qualitas Corpus, and normalized (range: [0,1]) to obtain a severity score. In combination with its centrality and order, a TD value is calculated for each AS and summarized as a TD index (TDI) for the whole system in each version. Given the shortcomings pointed out by Sas et al. [48], we excluded the "History" information from the TDI calculation, which doubles the TD of smells that increase in affected components and halves the TD of shrinking smells.

### E. Implementation

We implemented our tracking technique in a tool called ASTDEA ("Architecture Smell and Technical Debt Evolution Analyzer"), for which we show the overall workflow in Figure 7. Conceptually, ASTDEA controls a tool for detecting and measuring ASs, using its output and additional metadata to track and analyze the evolution of ASs. As detection tool, we used a modified version of Arcan [6]. Next, we highlight crucial modifications, which are mainly caused by our novel concept for tracking the evolution of CDs (cf. Sec. III-B).

First, we slightly changed the TD computation for CDs proposed by Roveda et al. [45]. While they computed the TD for each subcycle, we consider supercycles as intra-version smell instances. So, we compute the TD value of an intra-version CD as the sum of its subcycles' TD values.

Second, for Arcan, we added the TD computation, which was not implemented in the open-source version 1.2.1. Next, we added the detection of supercycles. Since we still needed to detect subcycles, we used Arcan's original detection algorithm for this purpose, but complemented it with an algorithm for supercycle detection. To this end, we employ Tarjan's [53] algorithm for strongly connected components. For the same reason (i.e., use of supercycles for CD detection), we replaced the CD shape detection technique of Arcan with the algorithm of Al-Mutawa et al. [2]. Finally, we added the metrics that were not contained in Arcan (cf. Sec. III-D) and implemented some optimizations, such as multi-threading.

## IV. CASE STUDY

In the following, we report the design of our case study and describe as well as discuss our findings.

TABLE III
OVERVIEW OF OUR SUBJECT SYSTEMS FROM THE QUALITAS CORPUS DATASET [54].

| system | domain | # versions | | first version | | | last version | | |
|---|---|---|---|---|---|---|---|---|---|
| | | original | included | id | date | loc | id | date | loc |
| Ant | build system | 23 | 23 | 1.1 | 2000-07-18 | 7,837 | 1.8.4 | 2012-05-23 | 105,007 |
| ANTLR | parser generator | 22 | 22 | 2.4.0 | 1998-09-18 | 2,834 | 4.0 | 2013-01-22 | 21,919 |
| ArgoUML | diagram application | 16 | 16 | 0.16.1 | 2004-09-04 | 106,500 | 0.34 | 2011-12-15 | 192,410 |
| Azureus/Vuze | database | 63 | 63 | 2.0.8.2 | 2004-03-14 | 62,388 | 4.8.1.2 | 2012-12-17 | 484,739 |
| Freecol | game | 32 | 32 | 0.3.0 | 2004-09-30 | 21,309 | 0.10.7 | 2013-01-07 | 100,748 |
| FreeMind | diagram application | 16 | 16 | 0.0.2 | 2000-06-27 | 2,712 | 0.9.0 | 2011-02-19 | 50,198 |
| Hibernate | database | 115 | 106 | 0.8.1 | 2001-11-30 | 3,555 | 4.2.2 | 2013-05-23 | 217,163 |
| JGraph | diagram application | 39 | 37 | 5.4.4-java1.4 | 2005-03-28 | 10,780 | 5.13.0.0 | 2009-09-28 | 22,758 |
| JMeter | software testing | 24 | 24 | 1.8.1 | 2003-02-03 | 34,170 | 2.9 | 2013-01-28 | 90,612 |
| JStock | stock trading | 31 | 31 | 1.0.6 | 2011-03-29 | 43,811 | 1.0.7c | 2013-06-20 | 48,842 |
| Jung | diagram application | 23 | 23 | 1.0.0 | 2003-07-31 | 7,206 | 2.0.1 | 2010-01-25 | 37,989 |
| JUnit | software testing | 24 | 23 | 2.0 | 1998-01-08 | 1,346 | 4.11 | 2012-11-16 | 7,428 |
| Lucene | text search | 36 | 31 | 1.2-final | 2003-09-10 | 6,505 | 4.3.0 | 2013-04-27 | 285,804 |
| Weka | machine learning | 63 | 38 | 3.1.7 | 2000-02-22 | 57,194 | 3.7.9 | 2013-02-21 | 247,805 |

### A. Subject Systems

To conduct our case study, we required a dataset of established systems with diversity regarding the involved developers, domains, number of versions, and sizes. Ensuring such properties helps to improve the external validity of our findings. Since we adapted Arcan [5], we considered only Java systems and required compiled `jars` of each version. For this purpose, we built on the Qualitas Corpus dataset [54] that fulfills these criteria and has been used in related studies [2], [46], [48]. We provide an overview of the systems and versions we selected in Table III.

As we can see, the 14 systems are well-established, cover various domains, span up to around 15 years of evolution (involving between 16 to 115 versions), and vary considerably in size as well as growth. We had to exclude 42 of the 527 versions provided. Namely, we excluded versions that refer to different branches of a system, which partly overlap regarding the periods they cover. Analyzing such versions would bias our results, for instance, by invalidating time-related metrics (e.g., JGraph 5.4.4 was released for Java 1.3 and 1.4 on the same day; thus, we used only the version for Java 1.4).

### B. Research Questions

The overall goal of our study was to gain insights into how ASs and their evolution contribute to software degradation. To this end, we formulated three research questions (RQs), each addressing a certain aspect of AS evolution.

**RQ$_1$** *How does the number of ASs evolve over time?*
We wanted to investigate how ASs in general evolve over time (i.e., increase/decrease), and how this relates to the evolution of other system properties, such as size, number of classes/packages, or TD. *Rationale:* A growing number of ASs indicates that system quality decreases, and thus the system becomes less maintainable over time.

**RQ$_2$** *What is the impact of ASs on TD?*
Since TD is an indicator for software degradation, our focus with this RQ was to analyze whether and how ASs impact TD over time, and thus cause an interest rate. *Rationale:* If we find a relation between TD and ASs (or certain types of ASs), corresponding cases should be considered more urgent for removal, since they have a greater impact on software degradation. As a result, finding such relations can help developers identify refactoring candidates among ASs earlier in a system's lifetime.

**RQ$_3$** *What influences the lifespan of ASs?*
A longer lifespan of an AS indicates that it manifests itself in the system, and thus continuously contributes to system degradation and may be more difficult to remove, since it is tightly integrated in the system. *Rationale:* We wanted to understand what properties mainly influence the lifespan of ASs. The results can help to identify and remove potentially long-living ASs early on.

By addressing these RQs, we provide insights for researchers and practitioners into how ASs manifest in a system, and which ASs may cause more software degradation.

### C. Analyses

We performed multiple analysis steps on our dataset. First, we executed several queries to elicit different metrics-based properties of each system's evolution (cf. Table II). Second, we performed a regression analysis and statistical tests to study the evolution of ASs and TD in the systems. Finally, we analyzed which of the properties have a major influence on the lifespan of ASs. For this purpose, we determined the number of versions for each intra-version smell until its last successor was removed or the analyzed time period was fully covered. We aggregated the results of all 14 systems, but looked at each type of ASs individually. Each property was divided into bins, so that a similar number of ASs fell into each bin, to allow aggregated statistical analyses. As minimum bin size for UDs and class-level CDs, we used $1/20$ of their total number. For HDs and package-level CDs, we used $1/10$ instead. We found these bin sizes to be appropriate through manual tests, since they ensure a fine enough granularity for differentiated analyses and a high enough aggregation to minimize noise in the results.

### D. Results & Discussion

To address our RQs, we measured different properties of our 14 subject systems. Since we cannot visualize all of them in this paper, we exemplify five systems in Figure 8 that represent the most interesting findings and are also representative for the other systems. We provide detailed visualizations and data

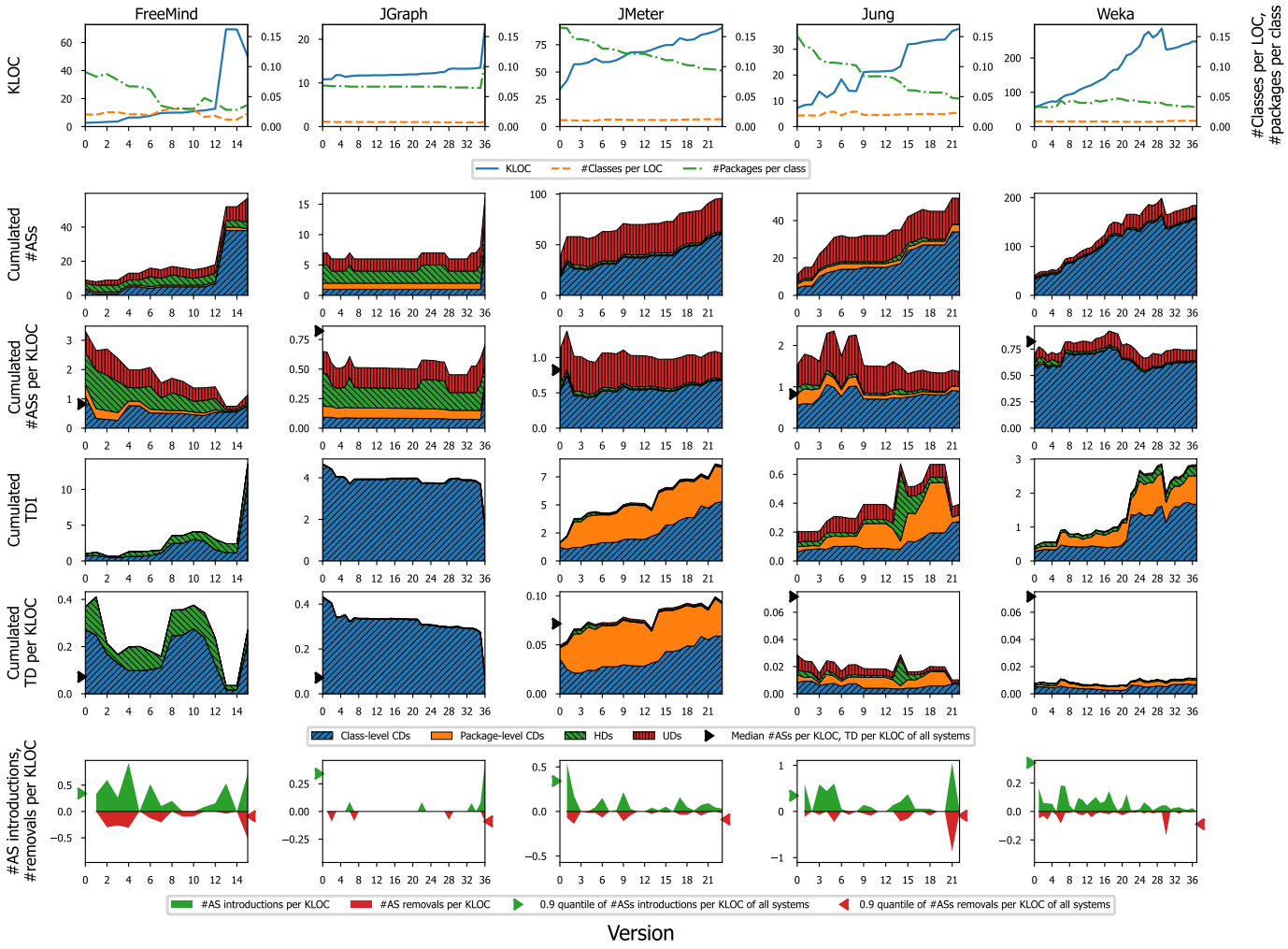


Fig. 8. Measurement results for five exemplary subject systems, for each from top to bottom row: (1) system size, (2) absolute ASs per type, (3) relative ASs per type, (4) absolute TD per type of ASs, (5) relative TD per type of ASs, and (6) introductions and removal of ASs.

for all systems in our replication package. From top to bottom, we display in each row the evolution of:

1) the system's size in thousand lines of code (KLOC) (left axis) as well as ratios of classes and packages (right axis);
2) the system's absolute number of ASs per type;
3) the relative number of ASs per type;
4) the system's absolute TD per type of ASs;
5) the relative TD per type of ASs; and
6) AS-introducing and AS-removing changes.

Below, we describe and discuss the results for each RQ.

### *E. **RQ$_1$**: Evolution of ASs*

**Results.** Unsurprisingly, we see in Table III and the first row of Figure 8 that most systems grew considerably in size (KLOC). Only JStock remained relatively stable in the observed period. Thus, most systems exhibit similar trends to JMeter, Jung, or Weka. However, we can also see that a few drastic increases in code size have been caused by a single version, for instance, in FreeMind and JGraph. Moreover, most systems have decreased in code size between a few versions, potentially caused by refactoring source code or architectural redesign. Still, the ratio of classes per LOC varies only slightly and the ratio of packages per class typically decreases (JGraph is an exception here, too).

Since most systems increased in size, it is also not surprising that the absolute number of ASs (second row in Figure 8) also increases. Particularly for FreeMind and JGraph, both values seem to be directly related. Interestingly, the total number of ASs rarely decreases throughout the versions we considered, and does not always correspond to a decreased code size (e.g., for FreeMind). Similarly, the total amount of TD is increasing the more ASs are incorporated in the system. An exception is JGraph, in which the TD decreased drastically, even though the code size and number of ASs increased.

In the third and fifth rows, we relate ASs and TD with the code size to mitigate the impact of sheer size from our analysis. We can see that the ratio of ASs and TD are mostly stable throughout a system's evolution, but they can vary considerably between different systems. FreeMind is an exception, since the relative number of ASs and the amount of TD decrease over time, independently from the change in code size. Also, in JGraph, the pattern of ASs increasing while TD decreased remains.

**RQ$_1$: How does the number of ASs evolve over time?**

*Most systems increased in code size throughout their versions, which relates to more ASs and TD. However, considering the relative number of ASs and amount of TD, they remained mostly stable and even decreased in some systems, and thus do not exhibit an exponential growth. Still, our data indicates that once ASs have infected a system, they won't disappear but rather manifest themselves, and thus contribute to software degradation.*

**Discussion.** Considering the total number of ASs, our findings are similar to those of Sas et al. [48], who found that the absolute number of ASs mostly increased during a system's evolution. We observed that the relative number of ASs stays mostly stable, and sometimes decreases, which somewhat contradicts the previous findings of Sas et al., who found increasing relative levels of package-level CDs. A possible reason could be the different perspective on CDs (i.e., supercycles vs. subycles, cf. Sec. III-B). However, similar results to our study have been found for CSs [50]. Reflecting on such findings, we argue that ASs also seem to be caused more by the sheer amount of source code rather than architectural properties. While we require further studies to strengthen such evidence, this finding has important, direct implications for researchers and practitioners: we should first focus on understanding to what extent ASs are actually a problem. Otherwise, we may run into the same problems as with CSs [50], for which considerable research efforts have been spent on specific refactorings and tools that may not address an actual problem. Similar issues have been identified and have partly caused considerable paradigm shifts for other areas, such as code clones [24], refactoring in general [55], or preprocessor use [15]. Consequently, our findings can help researchers scope their work and avoid costly endeavors into directions that could be misguided—instead focusing on understanding the real problem first.

Interestingly, but out of the scope of this paper, is the decrease in ASs and TD that we found for a few systems (e.g., FreeMind). Exploring whether this has been caused by a shift in the systems' architecture (and not just the size change) could help to identify architectural designs that prevent ASs. Similarly, it may indicate that the developers of these systems are aware of ASs and try to avoid them. Consequently, comparing between such and the remaining systems more qualitatively may help unveil to what extent ASs are an actual problem in practice, and how to avoid them.

### *F. **RQ$_2$**: ASs and TD*

**Results.** To answer this RQ, we focus on the diagrams in the second to fifth rows (absolute and relative ASs, and TD compositions) of Figure 8. Our results reveal that there are considerable differences in the ratio of each type of ASs to the overall number of ASs. Particularly, class-level CDs constitute the type that is mostly present for the whole period of the systems' evolution (e.g., for JMeter, Jung, Weka) with only few exceptions (i.e., JGraph, Freecol, partially Hibernate). Moreover, FreeMind shows a special evolution pattern, since its class-level CDs have a low share, but then increase sharply in version 14. By contrast, the share of package-level CDs is almost neglectable (exceptions: ANTLR, JGraph, Jung). Apart from class-level CDs, UDs constitute the second most present type of ASs in all systems (except JGraph). Finally, HDs occur only to a minor extent, but with a higher share than package-level CDs. For some systems (i.e., Weka, JUnit, ArgoUML), this smell type is neglectable or even disappears during the systems' evolution.

For the impact of ASs on TD (i.e., how much a certain smell type contributes to the TDI), the most prevalent type of ASs (class-level CD) has also the greatest impact on TD except for three systems (i.e., JMeter, Jung, Hibernate). Interestingly, for JGraph, the share of class-level CDs is rather low, whereas this type almost solely accounts for the TD. Next, even though having low shares of the overall number of ASs, package-level CDs contribute considerably (second most) to TD for almost half of the systems. For this type of ASs, we also observe cases (e.g., JMeter, Weka) in which the difference between its share of the total number of ASs and its impact on TD is considerable. Eventually, both HDs and UDs do not substantially amount for TD in most systems. In particular, UDs have an impact on three systems (i.e., ANTLR, Jung, JUnit), and HDs on five systems (e.g., JStock, Freemind).

In the last row of Figure 8, we show how AS introductions and removals evolve over time to evaluate whether this has an influence on TD. Our results indicate that ASs are introduced and removed simultaneously and in an irregular fashion. We assume that this is due to introduced smells subsuming existing ones (e.g., due to excessive growth or renaming), which is then interpreted as removal by our tracking technique. We also performed a correlation analysis for various combinations (e.g., AS level vs. TD increase rate, AS removal rate vs. TD increase rate; see replication package), but could not find any correlation that is significant for the majority of systems.

**RQ$_2$: What is the impact of ASs on TD?**

*We found considerable differences regarding the shares of types of ASs, with class-level CDs being the most prevalent type. These differences are also reflected by the types' impact on TD. However, our results show that for half of the systems, certain types of ASs (package-level CDs, HDs) have considerable impact on TD, even though they constitute only a small share. So, we conclude that the number of occurrences of a certain type does not necessarily impact software degradation. Rather, other properties (e.g., the complexity of ASs) seem to play a pivotal role.*

**Discussion.** Overall, we obtained several interesting insights for this RQ. First of all, the prevalence of class-level CDs is a complementary finding that extends on Sas et al. Interestingly, despite having a considerably lower relative number than for Sas et al.; as a result of our different definition (supercycles); class-level CDs remained the most prevalent type of ASs in our study. A more in-depth analysis is required to explain this result.

Second, and maybe the most important finding: The share of the types of ASs does not necessarily determine their impact on

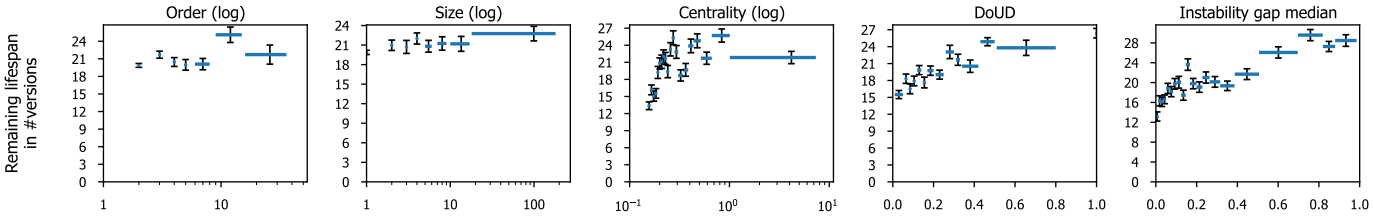

Fig. 9. Impact of different properties of UDs on the remaining lifespan. We show confidence intervals ($\alpha = 5\,\%$). Horizontal lines indicate bin ranges.

TD, and thus on software degradation. Possible reasons could be that some ASs are complex and tangled with the underlying system, thus affecting many components and contributing considerably to the overall TD. These observations are also stable for the whole evolution period, indicating that once such an intricating AS has entered a system, it is unlikely to disappear or be removed by chance. We argue that this constitutes a highly valuable result, since it allows practitioners to focus on specific, rather complex ASs with a high impact on TD as candidates to be removed. However, we are still analyzing what properties may be good proxies for such intricating smells.

### G. *RQ₃: Lifespans of ASs*

**Results.** By studying changes that introduced or removed ASs (cf. sixth row in Figure 8), we found that ASs are often introduced into, but rarely removed from, a system. So, we aimed to understand for what reasons (i.e., properties) ASs may persist throughout many versions. In the lifespan analysis, we found no clear correlation between most properties and the lifespan of ASs. However, depending on their type, ASs with increasing values for specific properties showed longer lifespans (with some fluctuation). As an example, we show the corresponding data for UDs in Figure 9 (explained shortly).

For class-level CDs, such properties include the order, size, size overcomplexity, severity score, and number of subcycles. The size overcomplexity exhibited the highest difference in average lifespans between the lowest and the highest bin of values (ca. 15 vs. 23 versions). For properties like the order and size, the mean lifespan ranges from around 15 to 20 versions.

Package-level CDs showed less clear tendencies and higher fluctuations. However, much more than for class-level CDs, the shape correlated with the lifespan. On average, cliques persisted for the highest number of versions (ca. 25), followed by multi-hubs. By contrast, chains were the least persistent shape (ca. 6 versions), followed by unknown shapes and stars.

Similarly, HDs exhibited no clear relationships between any property and the lifespan. For several ones, such as the order or size, bins with midsize values showed the highest mean lifespans. However, surrounding bins had mostly lower lifespans.

On average, the lifespan of UDs (cf. Figure 9) remained mostly similar for different orders, sizes, and size overcomplexities. By contrast, it showed a clear correlation with the centrality (first bin: ca. 13 vs. penultimate bin: ca. 26 versions), the DoUD (first bin: ca. 16 vs. last bin: ca. 26 versions), and the median of the instability gap (first bin: ca. 13 vs. last bin: ca. 28 versions). Furthermore, UDs that fully overlapped with other ASs typically persisted longer (ca. 21 versions) than those with lower overlap ratios (ca. 16 versions).

**RQ₃: What influences the lifespan of ASs?**

*We found no universal correlations between the properties of all types of ASs with their lifespans. However, increasingly complex class-level CDs persisted considerably longer, while UDs showed high lifespans if they are central, overlapping, or have high instability gaps.*

**Discussion.** While a naive assumption would be that especially large ASs persist longer, our findings show that this only applies for class-level CDs. Given their high shares on both the number of ASs and the TDI in many systems (cf. Figure 8), our results suggest that developers should target large class-level CDs first. This finding contradicts Sas et al. [48], who found tiny CDs (i.e., an order and size of two) to be the most persistent shape. A potential reason is the different tracking technique, since they tracked subcycle CDs, which can be expected to be less stable for larger instances than supercycle CDs (cf. Sec. III-B). We can apply a similar reasoning for the shapes of package-level CDs, for which our results also differ from Sas et al. (albeit they did not separate between class- and package-level in their survival analysis regarding shapes).

For UDs, it could be expected that the DoUD and instability gap correlate with the lifespan: UDs with only few less stable packages and a low difference in instability can be assumed to more quickly loose their status as a smell even without targeted refactoring. For example, the addition of a new dependency changes by definition the instability metric of both affected components. If more persistent ASs are considered more severe, our findings support the suggestion of Roveda et al. [46] to concentrate on UDs with a high DoUD for refactoring. Finally, the observation that more central UDs persist longer can be particularly interesting for practitioners, since such UDs potentially propagate changes to more parts of a system, and thus should be given higher priority during refactoring.

### H. Tracking Evaluation

Given that our tracking of CDs as graphs (cf. Sec. III-B) introduces additional complexity, we checked whether it increases the accuracy when analyzing the evolution of ASs. In our case study, we found that 23.3 % of intra-version class-level CDs were part of a CD family in which at least one split or merge occurred. For package-level CDs, this number amounted to 30.0 %. We observed the highest maximum family width of 109 parallel branches in Azureus with a family order of 2,500 smells, which alone represents nearly 30 % of all intra-version class-level CDs in Azureus. These results indicate that not considering CD evolution as graphs would

lead to a considerable loss in information when analyzing the relationship between smell instances in consecutive versions.

### I. Threats to Validity

A threat to the internal validity is that we implemented our own tooling to track ASs and to collect metrics on them. Thus, our results may be biased by bugs or a misconception in the design of our technique (or metrics). Since it is not possible to ensure that an implementation works as intended, we cannot overcome this threat. Nonetheless, to mitigate potential biases, we tested our implementation extensively and performed sanity checks during its design (e.g., inspecting ASs manually) to improve our confidence that our tooling works as intended.

A threat to the external validity of our case study are the selected subject systems. Some involve rather few versions, are comparably small, or involve parallel versions. Thus, our results may not be generalizable for other systems. We aimed to mitigate this problem by building on an established dataset of well-known open-source systems and ensuring the quality of our data (e.g., excluding parallel versions). Our results reveal different patterns regarding the evolution of ASs and TD, which we would expect for a wider range of systems. So, we argue that the potential threat caused by the selected systems is small.

## V. Related Work

**Evolution of CSs.** First studies [58] on the evolution of CSs involved only a few subject systems. For instance, Vaucher et al. [57] studied the evolution of *god classes* in Eclipse JDT (21 versions) and Xerces (34 versions), Olbrich et al. [37] god classes as well as *shotgun surgery* in Xerces (51 versions) and Lucene (25 versions), or Chatzigeorgiou and Manakos [12] *long method*, *feature envy*, *state checking*, and god class in JFlex (10 versions) and JFreeChart (14 versions). Interestingly, these studies obtained similar findings, namely that the ratios of different CSs can vary heavily between systems, but remain relatively stable within each system. Moreover, many CSs are introduced when creating the code, and most stay in a system for a long time. Recent studies [12], [40], [56] have been performed on larger numbers of systems, versions, and domains, for instance, for SQL [36], Android [19], [20], JavaScript [23], or PHP [43]. Not surprisingly, these studies suggest that some types of CSs are more actively removed in some systems, and thus different evolutionary patterns exist. Still, they largely agree on the previous finding, providing extensive empirical evidence regarding the evolution of CSs. However, the empirical evidence on the impact of CSs on software degradation is somewhat inconclusive [47], [50], [59]. Even though some results of these studies are comparable to ours, we had a different focus. Namely, we focused on ASs (not CSs) and advance on pure evolutionary analyses by also considering the impact ASs have on software degradation. Consequently, our work is complementary to, and advances upon, existing research on CSs.

**Evolution of ASs.** As we described, we built upon two existing works, which are among few investigating the evolution of ASs empirically. Sas et al. [48] conducted a study on three types of ASs on 524 versions of 14 systems, revealing that ASs differ regarding their growth and life-span. Since our study is inspired by Sas et al., the findings are partly similar. Nonetheless, we advance on this study and reveal new insights by considering TD, involving different ASs, and extending as well as improving the technique for tracking ASs. Similarly, we built on the work of Roveda et al. [46] who propose a so-called architectural debt index to measure the TD caused by ASs—also involving evolutionary aspects. However, the index itself is not derived from empirical findings, which we elicited with our analysis. Finally, Rangnau [42] studied six open-source systems to understand why developers introduce ASs. The results suggest that ASs are mostly resolved when developers fix a bug, and developers are unaware of introducing ASs. We complement existing studies on ASs, and provide a means to improve or validate proposed metrics based on real-world data.

**Evolution of TD.** Similarly to ASs, the evolution of TD has not been extensively studied, but is often associated with CSs. Potdar and Shihab [41] analyzed four open-source systems, which Bavota and Russo [9] extended to 159 systems—with both studies revealing similar findings. Most importantly in the context of this work, that TD becomes more expensive over time, but survives for a long time (similar to CSs). Similarly, Amanatidis et al. [4] found that it is more expensive to correct higher degrees of TD. Related to our findings, Digkas et al. [14] found that the TD in 66 Java systems of the Apache ecosystem increases over time, but it actually decreases when normalized to the systems' sizes. Since we built on the concept of TD, it is not surprising that we found partly overlapping results. However, we complement previous works, since we studied TD in relation to ASs, and thus revealed new insights that have not been reported previously.

## VI. Conclusion

In this paper, we presented considerable extensions to an established technique for tracking ASs through a system's evolution. Using the improved technique, we performed a case study in which we analyzed 485 versions of 14 open-source systems to study the evolution of ASs. Advancing further on previous works, we related ASs to TD to also study the impact of each type of ASs on software degradation. Our results indicate that our improved technique helps to uncover novel insights that are relevant for practitioners and researchers. For instance, we found that certain types of ASs seem to have far more impact than others, since they occur less often but still cause most TD. In the end, we argue that researchers first have to explore ASs in more detail to understand whether they are an actual problem and how to tackle them. Practitioners can benefit from our results by identifying, reviewing, and resolving potentially problematic ASs (i.e., class- and package-level CDs) to improve the design of their system. We aim to explore our findings in more detail in the future, particularly why ASs survive for such a long time and whether developers actually care to resolve ASs for what reasons. Using our insights, we intend to derive actionable recommendations and to potentially design new tool support for developers.

## REFERENCES


[1] I. Ahmed, U. A. Mannan, R. Gopinath, and C. Jensen, "An empirical study of design degradation: How software projects get worse over time," in *Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 2015.

[2] H. A. Al-Mutawa, J. Dietrich, S. Marsland, and C. McCartin, "On the shape of circular dependencies in Java programs," in *Proceedings of the Australian Software Engineering Conference (ASWEC)*. IEEE, 2014.

[3] N. S. R. Alves, T. S. Mendes, M. G. de Mendonça Neto, R. O. Spínola, F. Shull, and C. B. Seaman, "Identification and management of technical debt: A systematic mapping study," *Information and Software Technology*, vol. 70, 2016.

[4] T. Amanatidis, A. Chatzigeorgiou, and A. Ampatzoglou, "The relation between technical debt and corrective maintenance in PHP web applications," *Information and Software Technology*, vol. 90, 2017.

[5] F. Arcelli Fontana, I. Pigazzini, R. Roveda, D. A. Tamburri, M. Zanoni, and E. D. Nitto, "Arcan: A tool for architectural smells detection," in *Proceedings of the International Conference on Software Architecture (ICSA)*. IEEE, 2017.

[6] F. Arcelli Fontana, I. Pigazzini, R. Roveda, and M. Zanoni, "Automatic detection of instability architectural smells," in *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2016.

[7] P. Avgeriou, P. Kruchten, I. Ozkaya, and C. B. Seaman, "Managing technical debt in software engineering (Dagstuhl Seminar 16162)," *Dagstuhl Reports*, vol. 6, no. 4, 2016.

[8] U. Azadi, F. Arcelli Fontana, and D. Taibi, "Architectural smells detected by tools: A catalogue proposal," in *Proceedings of the International Conference on Technical Debt (TechDebt)*. IEEE, 2019.

[9] G. Bavota and B. Russo, "A large-scale empirical study on self-admitted technical debt," in *Proceedings of the International Conference on Mining Software Repositories (MSR)*. ACM, 2016.

[10] L. A. Belady and M. M. Lehman, "A model of large program development," *IBM Systems Journal*, vol. 15, no. 3, 1976.

[11] A. S. Cairo, G. de F. Carneiro, and M. P. Monteiro, "The impact of code smells on software bugs: A systematic literature review," *Information*, vol. 9, no. 11, 2018.

[12] A. Chatzigeorgiou and A. Manakos, "Investigating the evolution of code smells in object-oriented systems," *Innovations in Systems and Software Engineering*, vol. 10, no. 1, 2014.

[13] W. Cunningham, "The WyCash portfolio management system," *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, vol. 4, no. 2, 1992.

[14] G. Digkas, M. Lungu, A. Chatzigeorgiou, and P. Avgeriou, "The evolution of technical debt in the apache ecosystem," in *Proceedings of the European Conference on Software Architecture (ECSA)*. Springer, 2017.

[15] W. Fenske, J. Krüger, M. Kanyshkova, and S. Schulze, "#ifdef directives and program comprehension: The dilemma between correctness and preference," in *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2020.

[16] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 2019.

[17] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic, "Identifying architectural bad smells," in *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE, 2009.

[18] ——, "Toward a catalogue of architectural bad smells," in *Proceedings of the International Conference on the Quality of Software Architectures (QoSA)*. Springer, 2009.

[19] S. Habchi, N. Moha, and R. Rouvoy, "Android code smells: From introduction to refactoring," *Journal of Systems and Software*, vol. 177, 2021.

[20] S. Habchi, R. Rouvoy, and N. Moha, "On the survival of Android code smells in the wild," in *International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. IEEE, 2019.

[21] S. Herold, "An initial study on the association between architectural smells and degradation," in *Proceedings of the European Conference on Software Architecture (ECSA)*. Springer, 2020.

[22] C. Izurieta and J. M. Bieman, "How software designs decay: A pilot study of pattern evolution," in *Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 2007.

[23] D. Johannes, F. Khomh, and G. Antoniol, "A large-scale empirical study of code smells in JavaScript projects," *Software Quality Journal*, vol. 27, no. 3, 2019.

[24] C. J. Kapser and M. W. Godfrey, ""Cloning considered harmful" considered harmful: Patterns of cloning in software," *Empirical Software Engineering*, vol. 13, no. 6, 2008.

[25] F. Khomh, M. D. Penta, and Y. Guéhéneuc, "An exploratory study of the impact of code smells on software change-proneness," in *Proceedings of the Working Conference on Reverse Engineering (WCRE)*. IEEE, 2009.

[26] P. Kruchten, R. L. Nord, and I. Ozkaya, "Technical debt: From metaphor to theory and practice," *IEEE Software*, vol. 29, no. 6, 2012.

[27] G. Lacerda, F. Petrillo, M. Pimenta, and Y. G. Guéhéneuc, "Code smells and refactoring: A tertiary systematic review of challenges and observations," *Journal of Systems and Software*, vol. 167, 2020.

[28] D. M. Le and N. Medvidovic, "Architectural-based speculative analysis to predict bugs in a software system," in *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM, 2016.

[29] M. Lippert and S. Roock, *Refactoring in Large Software Projects: Performing Complex Restructurings Successfully*. Wiley, 2006.

[30] R. C. Martin, "OO design quality metrics: An analysis of dependencies," 1994.

[31] ——, "Design principles and design patterns," Object Mentor, 2000.

[32] A. Martini, F. Arcelli Fontana, A. Biaggi, and R. Roveda, "Identifying and prioritizing architectural debt through architectural smells: A case study in a large software company," in *Proceedings of the European Conference on Software Architecture (ECSA)*. Springer, 2018.

[33] H. Melton and E. D. Tempero, "An empirical study of cycles among classes in Java," *Empirical Software Engineering*, vol. 12, no. 4, 2007.

[34] B. Meyer, *Object-Oriented Software Construction*. Prentice Hall, 1997.

[35] R. Mo, Y. Cai, R. Kazman, and L. Xiao, "Hotspot patterns: The formal definition and automatic detection of architecture smells," in *Proceedings of the Working Conference on Software Architecture (WICSA)*. IEEE, 2015.

[36] B. A. Muse, M. M. Rahman, C. Nagy, A. Cleve, F. Khomh, and G. Antoniol, "On the prevalence, impact, and evolution of SQL code smells in data-intensive systems," in *Proceedings of the International Conference on Mining Software Repositories (MSR)*. ACM, 2020.

[37] S. M. Olbrich, D. S. Cruzes, V. R. Basili, and N. Zazworka, "The evolution and impact of code smells: A case study of two open source systems," in *Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 2009.

[38] J. A. D. Pace, A. Tommasel, and D. Godoy, "Towards anticipation of architectural smells using link prediction techniques," in *Proceedings of the Working Conference on Source Code Manipulation and Analysis (SCAM)*. IEEE, 2018.

[39] D. L. Parnas, "Software aging," in *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 1994.

[40] R. Peters and A. Zaidman, "Evaluating the lifespan of code smells using software repository mining," in *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE, 2012.

[41] A. Potdar and E. Shihab, "An exploratory study on self-admitted technical debt," in *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2014.

[42] T. Rangnau, "Determining the rationale of architectural smells from issue trackers," Master's thesis, University of Groningen, 2020.

[43] A. Rio and F. B. e Abreu, "PHP code smells in web apps: Survival and anomalies," 2020. [Online]. Available: https://arxiv.org/abs/2101.00090v1

[44] L. Rizzi, F. Arcelli Fontana, and R. Roveda, "Support for architectural smell refactoring," in *Proceedings of the International Workshop on Refactoring Tools (WRT)*. ACM, 2018.

[45] R. Roveda, "Identifying and evaluating software architecture erosion," Ph.D. dissertation, University of Milano-Bicocca, 2018.

[46] R. Roveda, F. Arcelli Fontana, I. Pigazzini, and M. Zanoni, "Towards an architectural debt index," in *Proceedings of the International Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, 2018.

[47] J. A. M. Santos, J. B. Rocha-Junior, L. C. L. Prates, R. S. do Nascimento, M. F. Freitas, and M. G. de Mendonça, "A systematic review on the code smell effect," *Journal of Systems and Software*, vol. 144, 2018.

[48] D. Sas, P. Avgeriou, and F. Arcelli Fontana, "Investigating instability architectural smells evolution: An exploratory case study," in *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2019.

[49] R. Sedgewick and K. Wayne, *Algorithms*. Addison-Wesley, 2011.

[50] D. I. K. Sjøberg, A. Yamashita, B. C. D. Anda, A. Mockus, and T. Dybå, "Quantifying the effect of code smells on maintenance effort," *IEEE Transactions on Software Engineering (TSE)*, vol. 39, no. 8, 2012.

[51] K. Sugiyama, S. Tagawa, and M. Toda, "Methods for visual understanding of hierarchical system structures," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 11, no. 2, 1981.

[52] G. Suryanarayana, G. Samarthyam, and T. Sharma, *Refactoring for Software Design Smells: Managing Technical Debt*. Elsevier, 2014.

[53] R. E. Tarjan, "Depth-first search and linear graph algorithms," *SIAM Journal on Computing*, vol. 1, no. 2, 1972.

[54] E. D. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble, "The Qualitas Corpus: A curated collection of Java code for empirical studies," in *Proceedings of the Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2010.

[55] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk, "When and why your code starts to smell bad," in *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 2015.

[56] ——, "When and why your code starts to smell bad (and whether the smells go away)," *IEEE Transactions on Software Engineering (TSE)*, vol. 43, no. 11, 2017.

[57] S. Vaucher, F. Khomh, N. Moha, and Y. Guéhéneuc, "Tracking design smells: Lessons from a study of god classes," in *Proceedings of the Working Conference on Reverse Engineering (WCRE)*. IEEE, 2009.

[58] N. Zazworka, M. A. Shaw, F. Shull, and C. Seaman, "Investigating the impact of design debt on software quality," in *Proceedings of the International Workshop on Managing Technical Debt (MTD)*. ACM, 2011.

[59] M. Zhang, T. Hall, and N. Baddoo, "Code bad smells: A review of current knowledge," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 23, no. 3, 2011.