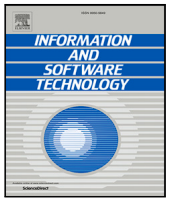
Information and Software Technology 164 (2023) 107322

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof



Software design analysis and technical debt management based on design rule theory

Yuanfang Cai [a,*], Rick Kazman [b]

[a] Department of Computer Science, Drexel University, United States of America
[b] Department of Information Technology Management, University of Hawaii, United States of America

ARTICLE INFO

Keywords:
Software architecture
Software design
Design debt
Automated analysis
Industrial case studies

ABSTRACT

In this paper we reflect on our decade-long journey of creating, evolving, and evaluating a number of software design concepts and technical debt management technologies. These include: a novel maintainability metric, a new model for representing design information, a suite of design anti-patterns, and a formalized model of design debt. All of these concepts are rooted in options theory, and they all share the objective of helping a software project team quantify and visualize major design principles, and address the very real maintainability challenges faced by their organizations in practice. The evolution of our research has been propelled by our continuous interactions with industrial collaborators. For each concept, technology, and supporting tool, we embarked on an ambitious program of empirical validation—in "the lab", with industry partners, and with open source projects. We reflect on the successes of this research and on areas where significant challenges remain. In particular, we observe that improved software design education, both for students and professional developers, is the prerequisite for our research and technology to be widely adopted. During this journey, we also observed a number of gaps: between what we offer in research and what practitioners need, between management and development, and between debt detection and debt reduction. Addressing these challenges motivates our research moving forward.

1. Introduction

In this paper we describe a journey down a long and winding road. This journey was motivated by our attempts to help software developers and managers understand the root causes of, the costs, and the consequences of a pernicious form of technical debt, called design debt. Along this road we created a series of theoretical and technical innovations – theories and tools – to help identify, measure, and (hopefully) fix design debt, based on Baldwin and Clark's design rule theory [1]. Different from technical debt smells and measures defined at the code level [2], we focus on design structure problems that are often the *root causes* of lower level problems manifested in code [3,4]. Baldwin and Clark's theory provides a foundation upon which to analyze and quantify well-known design principles, making these previously informal design principles operable. This journey was propelled by a mix of empirical studies and action research, with follow-up semi-structured interviews. Our goal was to provide convincing, quantitative evidence to both developers and managers of the projects we interacted with that their design debt was real, and that it could be precisely measured and monitored. We wanted to convince these projects that the information provided by our tools was actionable and could be used to guide specific improvements in their software design, saving costs and reducing risk in the long run.

We also reflect on the remaining challenges in this research area, which will direct our future journey, and these are considerable. In particular, we have learned that the most fundamental obstacle to the broad adoption of our research and the resulting technology is the lack of widespread understanding of basic design concepts and design principles. For this reason we make a call to action for the software engineering community: to improve software design education for both students and software practitioners.

The field of software design has produced important innovations for the past five decades, reaching back to the foundational work of Parnas and others [5]. The ideas of abstraction and information hiding and their many manifestations – in the SOLID principles, in design patterns, in frameworks and software architectures – have changed how the software community thinks about, talks about, and envisions software systems. However, while this journey has resulted in substantial numbers of ideas, papers and reports, and while it has influenced many designs in the real world, it has produced little in the way of objective, repeatable, empirically-grounded tools for design and design

* Corresponding author.
E-mail address: yuanfang.cai@drexel.edu (Y. Cai).

https://doi.org/10.1016/j.infsof.2023.107322
Received 16 November 2022; Received in revised form 22 August 2023; Accepted 24 August 2023
Available online 30 August 2023

analysis. Design and design analysis are still, fifty years after Parnas's early publications, more of an art than a science. Good designers tend to be old designers, the ones with decades of hard-won experience. For this reason, there has been a gradual increase in technical debt and, in particular, design debt, in the world-wide software community [6]. These problems are precisely what have driven our research and tool development.

Over the past decade we have embarked on a journey of exploring, evolving, and evaluating a series of innovative design concepts, technologies, and tools to manage software design and technical debt. We were driven by a passion to understand the nature of complex software designs, to visualize their structures, and to quantify their value. We are fortunate in that shortly after the publication of each of our main research innovations there were industrial practitioners who became interested and initiated a collaboration. In our research journey we proposed eight major new design concepts and technologies, evaluated them by carrying out action research in more than a dozen companies. Based on this experience we published five industrial case studies with five different commercial organizations spanning manufacturing, electronics, software services, healthcare, and other domains.

The new design concepts that we proposed build on top of one another, based on the experiences and lessons learned from our extensive industrial collaborations. In other words, it is the continuous evaluation of these ideas in "the lab", working with industry partners, and with open source projects, that has propelled their evolution. This process also drove the development of our first research tool, Titan [7], and its later evolution into DV8 [8]. Armed with these tools we were able to explore their application to software design education. The publications resulting from this line of research have accumulated over 1,000 scholarly citations over the past decade. Upon reflection, however, we realize that there is still a long way ahead for these ideas and technologies to be widely adopted and to benefit software development in practice.

Our research was motivated by Baldwin and Clark's design rule theory [1]. In our very first paper [9] we opened up the possibility of using their theory to visualize design structure, to interpret Parnas's research on information hiding in design [5], and quantify the economic value of information hiding. To bring this theory to practice, we first proposed the *Design Rule Hierarchy* (DRH)—a way of clustering a design structure to manifest the key design decisions of independent modules and design rules. After that, we merged information extracted from design structure and revision history, and published a new way to detect an important architectural flaw, which we called *Modularity Violation*. Soon after this idea was published in 2011, we worked with a real-world industrial partner to determine the consequences of this flaw in their projects. This set the tone for much of the research that we have pursued since, as we will show. We developed a new options-theory based metric, *Decoupling Level* (DL) [10], a new design model, called *design rule space* [3,11], a suite of design anti-patterns (which we also referred to as architecture flaws and hotspot patterns) [12,13], and the tooling to automatically detect them. We developed a number of technologies to determine the overall design complexity of a code base, to detect the technical debt caused by problematic design structures, and to guide project leaders in their decision-making surrounding design debt. We explored various ways to visualize this information, and to report on our results, all the while hoping to "close the gap" between developers and management.

This gap has been one of our significant challenges. Typically a developer knows, or at least feels, when their project's design is degrading, when there is enormous and ever-growing technical debt, when their productivity is dropping. But most developers have a problem determining exactly how and why this is happening, and have even less insight into how to fix it. So it is nearly impossible for them to make the business case with management that the code base should be thoroughly analyzed and refactored. The developers do not have the necessary data and cannot provide convincing evidence that, should they be allowed to refactor, things will improve. For this reasons, most projects plod along, growing increasingly debt-laden. And nothing changes, so nothing improves. Our technologies, especially the return-on-investment calculation [14], help address and alleviate this conflict to some extent, but major challenges remain.

2. Related work

Our work has been motivated by the gaps we observed between software research and practice on technical debt detection and management. As we will elaborate, we have come to believe that the only way to fundamentally change the culture surrounding design and architecture debt is through improved software design education.

2.1. What is offered vs. what is needed

In parallel to our journey of identifying and managing design debt, various definitions and detection techniques of code smells [2,15–18,18,19,19,20], design smells [21,22,22–24], and architecture smells [25–27] have been proposed to identify suboptimal structures or relations that may lead to technical debt. Fowler [28]'s definition of "code smells", e.g., god class, spaghetti code, code clones and feature envy have been very influential. Garcia et al. [29] proposed a suite of architecture smells based on the concept of components and connectors. Fontana et al. [26,27] also defined a set of architecture smells, such as hublike structure and cyclical components. Commercial and research tools such as SonarQube [30], Designite [31], Structure101 [32], Lattix [33], and Arcan [34] are available to detect some smells and anti-patterns. Although most of these definitions are also based on the violation of design principles, their definitions are all different, and so are the symptoms that they can detect [35].

Similar to our work, researchers have evaluated these code smells and architecture smells using both open source and industrial projects. For example, Palomba et al. [36] investigated the co-occurrence of code smells in 395 releases of 30 open source Java systems. Johannes et al. [37] studied code smells in 15 applications. Prevalence of smells in specific contexts or domains, such as code smells in Android code [38], in SQL code [39], and in machine learning code [40] have also been reported. Researchers also conducted various industrial studies to test the applicability's of these definitions and associated tools. Several books [41], [42] have introduced techniques for analyzing and managing technical debt. Ernst et al. [43] surveyed 1,831 software engineers and architects, and revealed that architectural decisions are the major source of technical debt. Sas et al. [44] recently investigated the evolution and impact of architecture smells in an industrial project. Martini et al. [45] reported the relation between architecture smells and architectural-level technical debt (ATD) with an industrial partner. Arcelli et al. [46] reported a similar study, but with a different industrial partner and more types of architecture smells.

The problem is, how to determine which of these symptoms are *real* technical debt? All the companies we collaborated with have adopted one or two of these tools to assess the quality of their source code, which often report a large number of problematic instances, making it hard to determine which ones are most important and worth fixing, and developers [47] tends to ignore the reported issues. Even though most of these symptoms are real problems, it is usually not feasible to fix all of them. Multiple studies claim that architectural issues are the major source of technical debt [3,43,48], but it is still not clear fixing which high-level problems will have the greatest return on investment. These problems were all raised during our collaboration with industrial partners, when we realize how deep the gap is between software design research and practice.

Another prominent example of this gap is software metrics, which have been studied for decades in our research community. McCabe Cyclomatic complexity [49] and Halstead metrics [50] are the most well-known ones to measure program complexity. C&K metrics [51], LK

Metrics [52], and MOOD Metrics [53] are also widely studied to measure object-oriented programs. In the software research community, these metrics are often used for bug prediction and localization [54–57], while our industrial collaborators are searching for metrics that can accurately monitor architecture decay, compare different projects and guide architecture improvement.

Our observation is that developers are often well aware of the existence and location of their problems. Once we received a comment "*We do not need a tool to tell us these classes are error-prone, we fix them every day!*" The real difficulty is to determine the scope and severity the problems, and convince both development and management that the problem is worth fixing. Our research in the past decade has been driven by the need to bridge this gap. Our fundamental idea is to integrate revision history into design debt detection [11,12,58], quantify the maintenance costs of each anti-pattern [13], design a metric suitable for comparison and monitoring [10,59], and enable quantitative return-on-investment (ROI) analysis [60].

2.2. Tools and metrics do not remove design debt

After our industrial collaborators adopted our tools and identified design debt, we were often asked *"How to fix these problems?"* Conversations like this have tended to cover general software design topics, such as design principles [5,61] and design patterns [62], and it is clear that current software design education is inadequate, as many software engineering educators have noted already [63–67]. Even when development teams decided to refactor and remove the detected design debt, it was challenging to assess whether the refactoring strategy was appropriate. And it was also challenging to determine if this refactoring was successful. Removing design debt effectively, just like creating a good design, requires highly qualified designers, and they can only be raised through proper software design education.

Currently, software design education may include teaching object-oriented design, model-based design methodologies, design patterns, design principles, and use-case analysis [68–73]. However, as Hu has pointed out [63], there are no widely accepted learning materials or pedagogy for software design. It is challenging to teach most of these abstract concepts and principles. Students who do not have experience with the challenges of maintaining a large-scale system are unlikely to fully internalize the benefits of patterns, or the importance of making a software system better modularized.

We have also been teaching software design and architecture courses for decades, and we realize that there is no broadly accepted, effective way for us to visualize or quantitatively assess: what a good design should look like, whether and to what extent proper design principles are followed, or which design patterns should be applied to address the identified anti-patterns. Over the past decade we have been trying to leverage our research results and tools to answer these questions, and to improve software design education.

3. On the path of software design and technical debt research

Upon reflection, our research in the past decade follows the process as depicted in Fig. 1. The first step is to identify common problems in existing software design research and practice. Second, we derive a suite of new concepts and metrics, based on Baldwin and Clark's option theory [1], to address these problems. The third step is the creation of supporting tools so that we can evaluate these new concepts and technologies at scale using open source projects. After that, we conduct industrial collaborators and collect feedback from practitioners, which helps to validate our hypotheses and which helps to create still more new concepts and tools. In this process, we have published a number of research papers, tool papers, industrial experience papers, as indicated in Fig. 2.

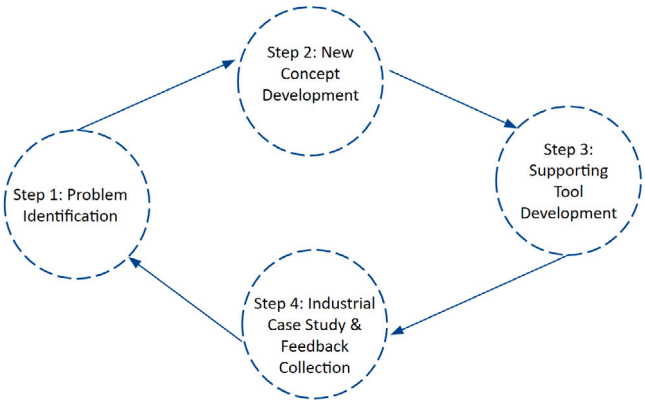


Fig. 1. Research design process.

3.1. Problem identification

As a community of research and practice, our understanding of and ability to manage software design is still immature. Starting from Parnas's seminal paper on Information Hiding [5], software engineering pioneers have proposed a number of key concepts to help reason about software design. The most influential ones include the SOLID design principles [61] and design patterns [62]. Although design principles and patterns are widely recognized, their application mostly depends on the experiences of individual designers and programmers. Designers still cannot say, with confidence, if their designs are applied properly, nor can they quantify the benefits of their design decisions. To visualize and reason about software design, numerous research approaches have been proposed to support software modeling, such as the unified modeling language (UML) [74,75] and various architecture description languages (ADLs) [76–78]. In practice however, almost none of these modeling approaches are regularly applied in real-world software projects [79–81].

In part this is because it is difficult for practitioners to make the link between design principles, patterns, modeling techniques and their actual code. So these concepts remain abstract and disconnected from the concerns of everyday programmers. There has been much work on pattern detection [82–86] and architecture recovery [87–90], which you might think would bridge this gap. But the accuracy of these approaches is insufficient to make them practical, usable, and cost-effective [91]. As a result, given all these advances in software design research, it is still difficult for developers to assess their designs in a disciplined way. It is difficult enough to visualize the design structure of a pattern embedded in a complex code base, or the overall system, let along quantitatively and objectively assess design quality.

Sullivan et al. [9] first introduced Baldwin and Clark's [1] design rule theory into software design modeling and quantitative assessment. This was the first paper that used the *design structure matrix* (DSM) as a way to visualize software designs. It also leveraged options theory to quantity the value of information hiding. They quantified the two designs described in Parnas's seminal paper [5], showing that the information hiding design offered much higher option values. The basic insight was that the higher the *technical potential* of a module, the higher its option value. Here *technical potential* denotes the likelihood that a module can be changed and improved to increase its value. For example, a module might be changed to provide better, faster, or more robust functions. If a module never needs to be changed, it will have low technical potential. Thus, the more independent high *technical potential* modules there are, the higher the value of the overall system.

This research showed that the key step to creating truly independent modules is to create abstract and high-quality *design rules*, e.g., critical architecture decisions. Design rules in software are typically manifested

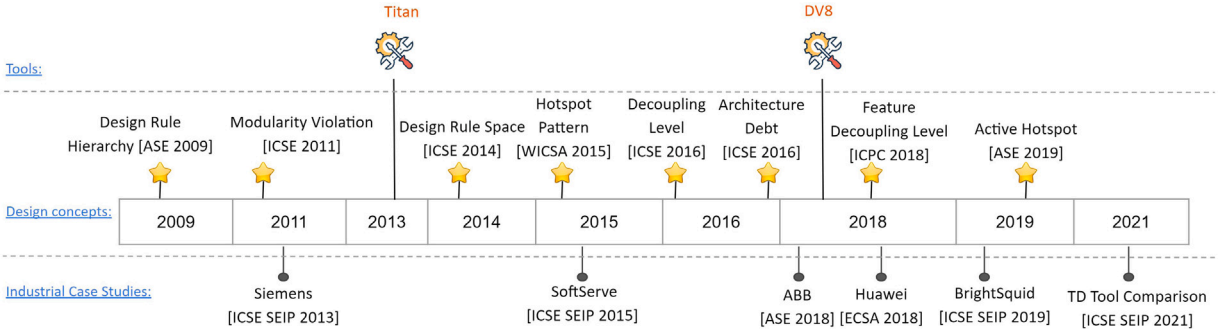


Fig. 2. Publication timeline of major concepts, associated tools, and case studies.

as important abstractions, such as the key interfaces in design patterns, that decouple the other parts of the design. For example, the *Observer* interface in an observer pattern decouples a subject from its concrete observers; the *AbstractFactory* interface decouples clients from concrete factories. The importance of these design rules, as well as their decoupling effects, can be visualized in a DSM model [1,3,9]. Using this representation many well known but previously informal design principles, such as information hiding [5,9] and the SOLID principles [2,12,13,61,92,93] can be visualized and quantified. This combination of options theory and the DSM modeling opened up new possibilities for modeling and assessing software designs with respect to these design principles, and formed the foundation for our research over the next decade.

Sullivan et al.'s paper [9] has, at the time of writing, had an *impact*, at least on the academic community: it has been cited numerous times. But bringing this theory to practice has posed significant challenges. For example, how do we assess if a module has *technical potential*, and how to measure such *potential*? And a more fundamental problem is, how do we define a "module"? In his seminal paper [5], Parnas proposed that "*Each task forms a separate, distinct program module.*". In other words, a module should be an independent task assignment. But how is an independent task assignment manifested in a complex code base? Answering these questions is the first step transforming this theory into something that can be operationalized in real-world software practice.

Starting in 2011, and over the subsequent decade, we proposed a number of concepts, supported by research tools, to advance our ability to both visually and quantitatively assess software designs. Some of the major milestones on this journey are highlighted in Fig. 2. Shortly after each of the proposed concepts was published, one or more industrial collaborators observed its potential value and contacted us. Working with them, we were able to conduct industrial case studies to apply these concepts in practice, and to publish our experiences.

During this process, and motivated by these conversations and these results, our insights deepened and our initial research prototype tool, Titan, was evolved into an industrial grade tool called DV8, with the support of our collaborators. Next we briefly introduce each of the fundamental concepts underlying our analysis approach. As we discuss the concepts we will also present the associated industrial case studies, and the tool capabilities we developed along the way.

3.2. New concepts in software design

Over the past decade we have proposed a number of new design concepts with the objective of visualizing and quantitatively assessing software design. These concepts were based on Baldwin and Clark's design rule theory and were modeled using design structure matrices.

Design Rule Hierarchy. In 2009 we proposed a concept that we called the *Design Rule Hierarchy* (DRH) [94] to explore one of our early fundamental questions, *how to define true modules*. In this hierarchy the most influential design decisions – design rules such as key abstractions – are clustered within the top layer of the hierarchy and are supposed to be kept stable. The elements within subsequent layers depend on design decisions in higher layers, and the elements within each layer are clustered into mutually independent modules. The modules in the bottom layer of a design rule hierarchy are *true modules* because they can be changed, added, or removed without impacting the rest of the system.

Fig. 3 illustrates the concept of a design rule hierarchy. In this example we show a DSM reverse-engineered from the source code implementing the Maze Game from Gamma et al.'s canonical design pattern book [62]. This design applies the abstract factory pattern so that the two concrete factories, red factory with $RedWall.java$, $RedRoom.java$, and $RedMazeFactory.java$ and blue factory with $GreenRoom.java$, $BrownDoor.java$, $BlueWall.java$, and $BlueMazeFactory.java$, can be configured and switched at run-time. This DSM is clustered using our DRH algorithm, which splits the 16 source files into two layers. The first layer contains seven files that are most influential, including key abstractions such as $MapSite.java$, and the key interface of the abstract factory pattern $MazeFatory.java$. The bottom layer contains three true modules decoupled by the design rule files in the first layer, including the two concrete factory modules, and the main control module. Our study [94] also demonstrated that developers working on the same DRH module have more communications than developers working on mutually independent modules, indicating that a DRH module is likely to represent an *independent task assignment*.

Modularity Violation. Given that we have identified a set of modules clustered into a DRH structure, we still have to the answer the following the question: why does it matter? Can the definition of modules and the DRH structure help with design quality assessment and, if so, how? In 2011, we explored the relation between DRH modules and file error-proneness and change-proneness. In our publication titled "*Detecting software modularity violations*" [58], we reported that files that belong to different DRH modules, but which changed together frequently, are more error-prone and change-prone. We called this phenomenon a *modularity violation*, as illustrated in Fig. 4. The DSM in Fig. 4(a) only displays syntactical dependencies among these nine files, showing that both $ProducerImpl.java$ and $ConsumerImpl.java$ depend on $PulsarApi.java$. The other seven files are independent from each other and from these three files. Their co-change history, as depicted in Fig. 4(b), however, revealed that these files changed together frequently. For example, in row 9, column 8 of Fig. 4(b), "$(0, 22)$" means that these two files have no structural dependencies, but were changed together 22 times, indicating that there are strong implicit assumptions coupling these files (Fig. 4(b)).

Design Rule Space. During this process we also applied these emerging ideas, supported by the Titan tool [7] developed by our PhD students, to the teaching of design patterns. In particular, we observed that each implemented design pattern actually forms a design rule hierarchy led by a few key interfaces. If we consider each key interface as a design rule, its DRH, generated automatically by Titan, could identify

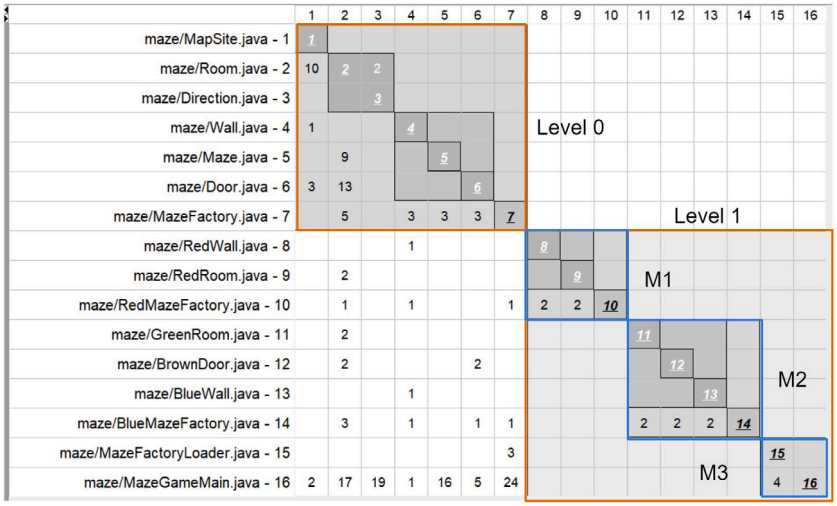
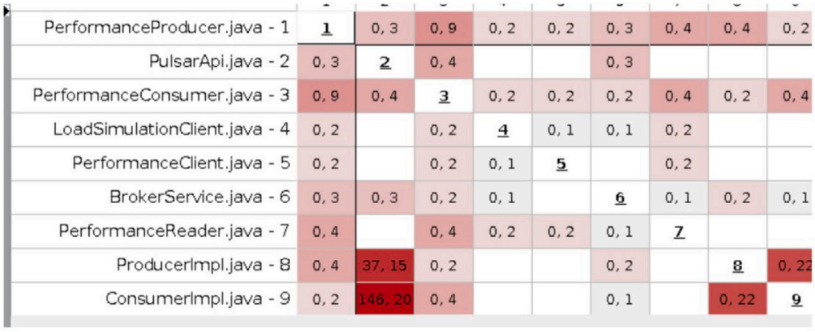


Fig. 3. Abstract factory pattern in Design Rule Hierarchy (DRH).

(a) Structurally Independent Modules



(b) Frequent Co-changes among Modules

Fig. 4. Modularity violation: modules appear to isolated but change frequently together.

the participants of the pattern and their relationships [92,93]. We further tested this idea by analyzing the source code of several open source projects, and we observed that in all of these projects, there are typically just a few key interfaces, each leading a specific design space reflecting one aspect of the design. We thus proposed the idea of a *Design Rule Space* (DRSpace) [3], and that a complex software design could be understood as multiple, overlapping design spaces, each of which can be modeled as a DRSpace.

We also explored the correlation between DRSpaces and error-proneness and change-proneness in source files. This study revealed several interesting results that we published in ICSE 2014 [3], and later extended into a journal paper [4] published in IEEE Transactions on Software Engineering. In particular, this study revealed that most error-prone files in a project can be captured by just a few DRSpaces. Seen another way, it appeared that the more error-prone and change-prone a set of files were, the more likely it was that they were architecturally connected. Another interesting result was that if an influential design rule is error-prone or change-prone, most of the files within its DRSpace are also error-prone and change-prone. Thus DRSpaces could provide some useful insights into software design quality.

Architectural Anti-patterns/Hotspot Patterns. Based on these observations on the relation between DRSpace, modularity violations, and file error-prone and change-proneness, we defined and evaluated a

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
ManagedLedger.java - 1	1				0, 2	0, 3		0, 2	0, 2		0, 3		18, 2	5, 2	0, 2	6, 3		0, 2		0, 2
ManagedLedgerFactoryImpl.java - 2	9, 2	2		0, 8		0, 2	0, 5	2, 2	0, 3	0, 3		27, 8	25, 3			0, 2		0, 1	0, 1	0, 1
ManagedLedgerTest.java - 3	275, 3	137, 3	3	19, 9	0, 1	0, 2	0, 6	20, 5	0, 4	0, 5	0, 2	282, 20	58, 0	6, 1		240, 0	0, 3	0, 1		0, 1
ManagedCursorImpl.java - 4	2, 5	0, 8		4		0, 2	0, 9	0, 4	0, 8		0, 1	219, 28	111, 3	0, 1		14, 14		0, 2	0, 13	0, 3
ReaderImpl.java - 5	0, 2		0, 1		5	0, 3				0, 4	0, 3	0, 3		0, 4			0, 3	14, 12		0, 4
NonPersistentTopic.java - 6	0, 3	0, 2	0, 2	0, 2	0, 3	6	13, 12	0, 1	0, 8	0, 42	4, 25	0, 4		7, 9	0, 2	0, 1	0, 16	0, 8	0, 1	5, 19
BrokerService.java - 7	3, 2	0, 5	0, 6	0, 9		6, 12	7	0, 3	12, 37	27, 21	39, 9	0, 10	3, 0	0, 3		0, 2		0, 1	0, 2	
OpAddEntry.java - 8	0, 2			0, 4		0, 1	0, 3	8	0, 1	0, 1		28, 10	13, 0			0, 1		0, 1		0, 1
ServiceConfiguration.java - 9	0, 2	0, 3	0, 4	0, 8		0, 8		0, 1	9	0, 17	0, 8	0, 9		0, 5		0, 2		0, 1	0, 2	0, 18
PersistentTopic.java - 10	31, 6	0, 3	0, 5	18, 5	0, 4	0, 42	38, 21	0, 1	0, 17	10	14, 26	26, 12	24, 0	17, 10	0, 5	16, 4		0, 9	0, 4	8, 22
Topic.java - 11	0, 3		0, 2	0, 1	0, 3				0, 8		11	0, 3		7, 10	0, 2	0, 1		0, 7	0, 1	1, 16
ManagedLedgerImpl.java - 12	2, 13	9, 8		32, 28	0, 3	0, 4	0, 10	39, 10	0, 9		0, 3	12	90, 4	11, 2	0, 5	34, 6		0, 4	0, 5	
AsyncCallbacks.java - 13	1, 2												13			1, 1				
PulsarApi.java - 14				0, 1	0, 4		0, 3		0, 5					14	0, 1					
TopicReaderTest.java - 15	0, 2				13, 4	0, 2	1, 1			0, 5	0, 2	0, 5		0, 1	15			4, 6		0, 3
ManagedCursor.java - 16	1, 3	0, 2				0, 1	0, 2	0, 1	0, 2		0, 1		21, 1			16				
PersistentTopicTest.java - 17	16, 2	1, 1	0, 3	5, 1	0, 3	0, 16	18, 5		38, 4	180, 22	2, 10	7, 3	81, 0	289, 12		42, 0	17	0, 8		5, 13
ConsumerImpl.java - 18	0, 2	0, 1	0, 1	0, 2		0, 8	0, 1	0, 1	0, 1	0, 9	0, 7	0, 4		146, 20			0, 8	18		0, 17
ManagedCursorContainerTest.java - 19	2, 2	0, 1		0, 13		0, 1	0, 2		0, 2	0, 4	0, 1	0, 5	22, 1			32, 13			19	
ServerCnx.java - 20	0, 2	0, 1	0, 1	0, 3	0, 4		43, 12	0, 1	0, 18	4, 22	20, 16	4, 4	3, 0	376, 33	0, 3			0, 17		20

Fig. 5. Unstable interface: influential files that often change with its dependents.
ManagedLedger.java is an unstable interface because 10 other files syntactically depend on it, and it changed frequently with most of the other 19 files in this cluster.

number of design flaws that we called *anti-patterns* [12]. Each of these anti-patterns represents a problematic design structure that violates one or more design principles and leads to high maintenance costs. Later we further generalized these structures as six types of *architecture anti-patterns*[13]:

- Unstable interface: influential files that changed often with its dependents, as illustrated in Fig. 5.
- Crossing: files with high fan-in and high fan-out but changed with all relatives frequently, as illustrated in Fig. 6.
- Modularity violation: modules that appear to be isolated but changed together frequently, as illustrated in Fig. 4.
- Clique: files that form strongly connected component.
- Package cycle: cyclically dependent folder pairs.
- Unhealthy Inheritance: parent class depends on subclasses, or the client of the hierarchy uses both the parent and the children, detecting the violation of Liskov Substitution Principle [61].

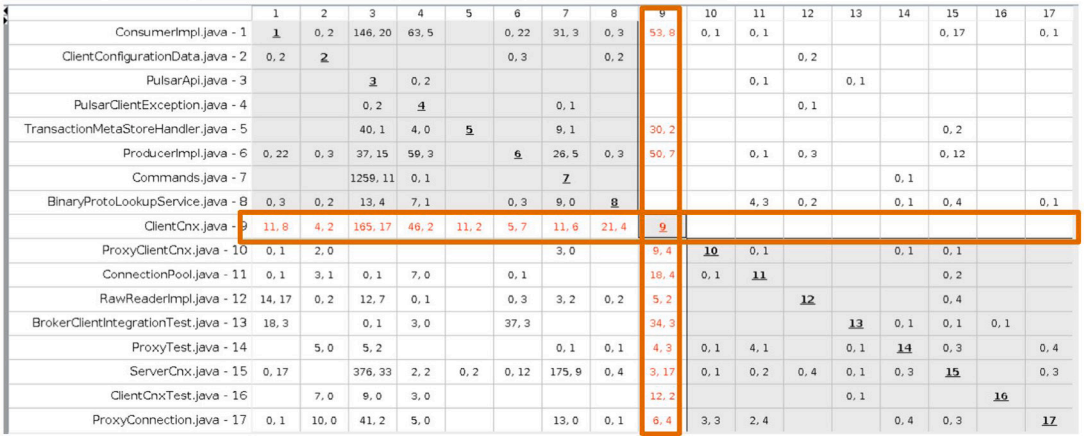
The first three anti-patterns are defined based on both structural dependencies among files and their co-change history. We have analyzed dozens of open source and industrial projects, and demonstrated that files involved in these anti-patterns are much more error-prone and change-prone than other files. These anti-patterns also provide clues on what type of refactoring is needed. For example, to remove a modularity violation, the designer should identify what caused the co-changes, e.g., cloned code or implicit assumptions, and remove these symptoms by modularizing them using more efficient abstractions.

(Compound) Architecture Debt. Anti-patterns and high-maintenance file structures are naturally connected with the concept of technical debt [95]: a problematic design structure that keeps generating extra maintenance costs is a natural analogy of a "debt". Accordingly, we further defined a special type of technical debt, which we called *Architectural Technical Debt* (ATD), referring to sub-optimal architectural design decisions in a software system that incur high maintenance costs—"interest"—over time. To measure such debts, we formally defined them, and demonstrated how to automatically identify and quantify them using a novel history coupling probability matrix [96]. In our follow up research, we further aggregated compound ATDs to capture the complicated relationships among multiple ATD instances. The point of this was that these compound ATDs should be the focus of a project's analysis, if they are seeking effective refactoring solutions [11].

Decoupling Level. Starting in 2011, we have been collaborating with a number of corporations who were interested in these technologies. During our collaboration, the most frequently asked question was: "*Is there a way to measure the design quality of our product?*" At this point, we introduced to our collaborators the numerous software metrics [49, 52,53,97–99] published in our community in the past few decades. However, in doing so we realized that despite decades of research on software metrics, we still cannot reliably measure if one design is "more maintainable" than another. Software managers and architects need to understand whether their software architecture is "good enough", whether it is decaying over time and, if so, by how much. To address these challenges, we contributed a new architecture maintainability metric—*Decoupling Level* (DL) [10]—based on Baldwin and Clark's option theory and our own design rule hierarchy clustering. Instead of measuring how coupled an architecture is, we measure how well the software can be *de*coupled into small and independently replaceable modules. We measured the DL for 108 open source projects and 21 industrial projects, each of which had multiple releases. Our main result showed that the higher the DL score, the better the architecture. By "better" we mean: the more likely bugs and changes can be localized and separated, and the more likely that developers can make changes independently. The DL metric also opened up the possibility of quantifying canonical design principles of single responsibility and separation of concerns, aiding cross-project comparisons and architecture decay monitoring, and establishing design quality benchmarks.

Feature Decoupling Level.

The DL metric itself still could not assess if and to what extent an architecture is "good enough" to support feature addition and evolution – one of the most valuable properties of a design – or to determine if a refactoring effort is successful, such that features can be added more easily. We thus contributed a concept called the *feature space*, and a formal definition of *feature dependency*, derived from a software project's revision history. We captured the dependency relations among the features of a system in a *feature dependency structure matrix* (FDSM), using features as first-class design elements. We also proposed a *Feature Decoupling Level* (FDL) metric that could be used to measure the level of independence among features. Our investigation of 17 open source



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
ConsumerImpl.java - 1	1	0, 2	146, 20	63, 5		0, 22	31, 3	0, 3	53, 8	0, 1	0, 1				0, 17		0, 1
ClientConfigurationData.java - 2	0, 2	2				0, 3		0, 2				0, 2					
PulsarApi.java - 3			3	0, 2							0, 1		0, 1				
PulsarClientException.java - 4			0, 2	4			0, 1					0, 1					
TransactionMetaStoreHandler.java - 5			40, 1	4, 0	5		9, 1		30, 2						0, 2		
ProducerImpl.java - 6	0, 22	0, 3	37, 15	59, 3		6	26, 5	0, 3	50, 7		0, 1	0, 3			0, 12		
Commands.java - 7			1259, 11	0, 1			7							0, 1			
BinaryProtoLookupService.java - 8	0, 3	0, 2	13, 4	7, 1		0, 3	9, 0	8			4, 3	0, 2		0, 1	0, 4		0, 1
ClientCnx.java - 9	11, 8	4, 2	165, 17	46, 2	11, 2	5, 7	11, 6	21, 4	9								
ProxyClientCnx.java - 10	0, 1	2, 0					3, 0		9, 4	10	0, 1			0, 1	0, 1		
ConnectionPool.java - 11	0, 1	3, 1	0, 1	7, 0		0, 1			18, 4	0, 1	11				0, 2		
RawReaderImpl.java - 12	14, 17	0, 2	12, 7	0, 1		0, 3	3, 2	0, 2	5, 2			12			0, 4		
BrokerClientIntegrationTest.java - 13	18, 3		0, 1	3, 0		37, 3			34, 3				13	0, 1	0, 1	0, 1	
ProxyTest.java - 14		5, 0	5, 2				0, 1	0, 1	4, 3	0, 1	4, 1		0, 1	14	0, 3		0, 4
ServerCnx.java - 15	0, 17		376, 33	2, 2	0, 2	0, 12	175, 9	0, 4	3, 17	0, 1	0, 2	0, 4	0, 1	0, 3	15		0, 3
ClientCnxTest.java - 16		7, 0	9, 0	3, 0					12, 2				0, 1			16	
ProxyConnection.java - 17	0, 1	10, 0	41, 2	5, 0			13, 0	0, 1	6, 4	3, 3	2, 4			0, 4	0, 3		17

Fig. 6. Crossing: files with high fan-in and high fan-out but change with all relatives frequently.

projects showed that files within each feature space are much more likely to be changed together [59].

Active Hotspot. At this point, the concepts of design flaws, architectural anti-patterns, and architecture debt had been adopted by several of our industrial collaborators. However, a large scale software system typically has dozens of anti-pattern or debt instances, and many of them overlap with each other since one file can participate in multiple relationships with other files and hence multiple anti-patterns. It takes time to pinpoint the exact patterns and files that need to be refactored and to prioritize refactoring activities. Moreover, certain anti-patterns, such as unstable interface and crossing, cannot be confidently identified until both co-change and dependency count thresholds are reached, which takes time. Severe software degradation does not happen overnight, and it is often too late by the time an anti-pattern is detected. Software evolves continuously through maintenance tasks – primarily fixing bugs and adding new features – and architecture flaws emerge quietly and largely unnoticed until they grow in scope and significance. At some point the system becomes a challenge to maintain. Developers are largely unaware of these flaws and their impacts go unnoticed. To detect these flaws early, we proposed a concept called *Active Hospot* [100], files that are changed frequently to address different issues in a given time range. Using active hotspots, we can monitor software evolution by tracking the interactions among files revised to address issues and so we can identify problematic structures earlier. Our study revealed that there exist just a few dominating active hotspots per project at any given time. Moreover, these dominating active hotspots persist over long time periods, and thus deserve special attention by project personnel.

3.3. Supporting tools

A number of tools have been created that purport to measure and analyze design complexity. Some of these are commercial tools and others are academic. In this section we will first describe the capabilities of our tools DV8 and Titan, as well as their differences. Then we will discuss how these tools compare to other state-of-the-art tools for design analysis.

To calculate DL scores, detect anti-patterns, visualize design structures and anti-patterns, and to be able to conduct large scale analyses, interventions and evaluations, we created two tools over the past decade: Titan [7] and DV8 [8]. The creation of these tools greatly aided our research journey as they enabled automated analyses of ever-increasing scale.

Titan Tool. Titan was created by Dr. Cai's Ph.D. students as a research prototype, to support DSM modeling, basic DSM manipulation operations, DRH clustering, DRSpace extraction, DL calculation, and modularity violation detection. Different from other DSM-based tools, such as Lattix [33], Titan separates a DSM into two types of data models: a dependency matrix and a clustering. Thus, a user could view their source code using different clustering methods. Titan also accepts dependency structures extracted from source code and history co-change information extracted from a project's revision history. As the need to support new concepts and new features grew, as well as the increased number of users we wanted to support, and the ever-increasing scale of systems we wished to analyze, the limitations of Titan, in terms of scalability, extensibility, and usability, became more prominent.

DV8 Tool. Thanks to the support of our industrial collaborators, we created a new industrial-grade tool, called DV8 [8]. Based on, and evolved from Titan, DV8 supports anti-pattern detection, active hotspot detection, and return on investment analysis. It makes it easy to navigate between dependencies in a DSM to the source code from which the DSM was created. DV8 also includes an important new feature, we thought, for communicating with project leaders: the creation of a design debt report that named and quantified the most serious design debts. DV8 also provides better support for the user to analyze different types of dependencies separately, and to configure project-specific thresholds needed to detect anti-patterns. For each detected anti-pattern and hotpot, DV8 exports a detailed spreadsheet with their maintenance costs, as calculated from the project's revision history.

Technical debt tool comparison. We consider DV8 to be a technical debt detection tool, based on our definition of design debt. In fact, many technical debt analysis tools have been developed in recent years – both research tools and industrial products – such as Structure 101 [32], Designite [31], and SonarQube [30]. Each of these tools purports to identify problematic files, and each tool does this using their own definitions and measures. To understand the extent to which these tools agree with each other we conducted an empirical study analyzing 10 projects using multiple tools, including DV8, Structure 101 [32], Designite [31], SonarQube [30], Archinaut [101], and Succinct Code Counter (SCC) [102]. We wanted to see if the top-ranked most problematic files reported by these tools were largely consistent. Our results [35] showed that: (1) these tools report very different results even for the simplest and most common measures, such as file size in lines of code, code complexity, file cycles, and package cycles. (2) These tools also differ dramatically in terms of the set of problematic files they identify, since each implements its own definitions of being "problematic". Our study also revealed that the code-based measures these tools offered, other than file size and complexity, do not even moderately correlate with a file's change-proneness or error-proneness. These tools, in the end, provided no more insight than "big files are bad", and you do not need an expensive tool to tell

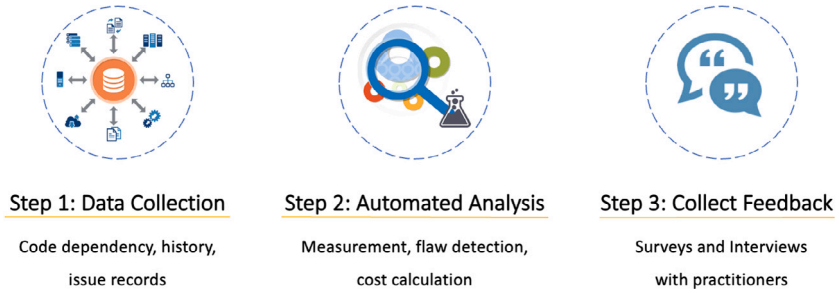


Fig. 7. Research interaction steps.

you that. In contrast, co-change-related measures, such as those DV8 offers, performed better. These measures were positively correlated with productivity measures such as bugs, changes, and churn. This study highlighted the need for the community to create benchmarks and data sets to assess the accuracy of software analysis tools in terms of commonly used measures.

Different from other tools, like Designite, that are also based on design concepts and principles (such as no cycles among packages) DV8 and Titan integrate co-change history with syntactical dependency relations, making it more accurate in identifying debts that have already incurred maintenance penalties. Unlike other tool-supported metrics, such as cyclomatic complexity, the Decoupling Level metric supported by DV8 is independent of project size [10]. Another popular tool that is superficially similar to DV8 is CodeScene [103], which provides visualizations of the most active code, and identifies hotspots based on revision history only. By contrast, DV8 combines structure and co-change information, and detect anti-patterns based on the violation of design principles.

3.4. Industrial case studies

The development and evolution of these concepts and tools are the results of our continuous interaction with our industrial collaborators. We are lucky that, after we presented each of these concepts in conferences, we were contacted each time by industrial practitioners to initiate collaborative projects. New ideas and concepts grew out of these practices, often motivated by the limitations of prior technologies. We have now conducted dozens of analyses of real-world systems over the years, interacting with over 10 commercial organizations.

The process we engaged in with our industrial collaborators involved three major steps, as shown in Fig. 7. In Step 1 we gained access to the project's artifacts – typically code, commits, and issues – and used those to determine the project's architecture flaws in Step 2. We then created a report highlighting the set of architectural flaws, accompanied by data showing the impact of each flaw in terms of the bugs, changes, and churn that had accumulated around the files participating in each flaw. This allowed us to quantify the impact, in terms of lost productivity, accompanying each flaw. This report was given back to the project's key stakeholders, typically a project manager and technical leads (architects and developers). In Step 3 we surveyed and interviewed those key stakeholders, to try and assess whether the problems that we had found were indeed causing pain in the project.

Our first industrial collaboration was with Siemens AG, which the objective of assessing the effectiveness of modularity violation in terms of identifying design problems. In this study, we identified sets of files with modularity violations, that is, files that changed frequently together without being structurally related. Project personnel confirmed that the identified clusters reflected significant architectural violations, and important undocumented assumptions. Given this information a refactoring proposal was made by Siemens developers, accepted by the project manager, and implemented. Based on this experience, in 2013 we published our first case study titled *"Measuring architecture quality by structure plus history analysis"* [104].

During this study, we had to evolve the Titan tool to support the detection of modularity violations, other newly defined anti-patterns, and the extraction of design rule spaces. In 2015 we performed a case study with Softserve, an IT outsourcing company [14]. In this study we analyzed one of Softserve's systems, detecting its architectural anti-patterns as a special type of technical debt—*design debt*—and built an economic model of the costs and benefits of refactoring. Using this model we were able to show a 300% return on investment in the first year alone for the proposed refactorings.

At this time we began additional collaborations with other major multinational companies to further develop and evaluate these ideas and gradually transformed Titan—a research prototype—into DV8—a more fully-featured and more carefully architected industrial-strength tool. DV8 was architected to be more scalable and extensible, allowing us to analyze larger systems and to extend our analyses to new programming languages and new kinds of dependencies.

For example, at this point, we began working with ABB Ltd. the Swedish-Swiss multinational manufacturer of electronics, robotics and automation technology, and Huawei, a Chinese multinational technology corporation. In our ABB case study [105] we used DV8 to measure the DL scores of eight of their projects and detected their architecture flaws. We also collected development process data from the project teams as input to DV8, reported the results back to the practitioners, and followed up with discussions and interviews. Based on these analyses, six of ABB's projects decided to do refactorings. The other two projects were small and received relatively good results from the analysis. In our study with Huawei [106], the development teams used DV8 to determine the severity of their technical debt, and used the identified anti-patterns to guide their refactoring strategy and improve their architectures.

Finally, we did a longitudinal study with BrightSquid, a provider of secure communication tools for the healthcare industry [107]. We analyzed BrightSquid's secure communication platform over a one year period (June, 2016–May, 2017). As a result of this analysis we identified many areas of architecture debt – the "before" state of their architecture – and recommended a refactoring plan to pay down the debt. They did the refactoring over a three month period (January, 2018–March, 2018). We then collected data for the "after" state, from March, 2018 to August 2018. Based on this "after" snapshot, we again analyzed their architecture. The results were dramatic: after refactoring, the numbers of architecture flaws were reduced, and project productivity measures increased dramatically. The average time needed to close issues before and after refactoring was reduced by 72%. The average bug-fixing churn per issue dropped by 2/3: from 102 LOC before refactoring to 34 LOC after refactoring. The average bug-fixing duration reduced 30%, dropping from 10 days before to 7 days. Brighsquid management and architects had intuitively understood that their architecture was sub-optimal, but they were unsure if it is worthwhile to refactor. This analysis quantified the severity of the problem and guided them in their refactoring.

These case studies, taken together, presented (we believed) compelling evidence of the value of this kind of automated analysis. At the "push of a button" a system could be analyzed, unlike previous

architecture analysis methods, such as the ATAM, which required a great deal of planning and human labor [108,109]. We presented analyses backed up by data from the projects themselves, and showed the tremendous value that could be reaped by investing in refactoring to remove design flaws.

4. Reflections and challenges to broad adoption

Although we have had a reasonable number of early adopters, and the evolution of the concepts and tools are all based on continuous industrial interaction, the actual uptake of the research and the supporting tooling in industry has been scant. These early adoptions were all initiated by our collaborators, after our conference presentations, followed by multiple remote or in person meetings. The fact that these companies are attending software engineering conferences and the fact that they contacted us proactively to initiate collaborations suggest that there is an on-going need to seek effective methods to diagnose, manage, and quantify software architectures. During these meetings with our industrial collaborators we explained not only the tools, but also the design principles and rationales behind the metrics and definitions that we had devised, as well as how to interpret the results. After these tools were applied in their projects, we followed up with additional meetings and interviews to get the team's feedback. During this process, we realized that there are still profound challenges to the broad acceptance and adoption of our technologies in industry. We detail some of these challenges below.

4.1. The need for improved software design education

Whenever we have introduced our technologies to practitioners, we have always had to start by introducing basic design principles and terms. We need to teach them about design! After that, our users have had to study our papers to fully understand how the metrics and anti-patterns are related to the design principles they were taught, and to fully grasp the meaning and implications of the report generated from our tool. We have found that we simply could not assume any solid understanding of design concepts, even among seasoned programmers. In addition to the five companies with whom we have published papers, some of our unpublished collaborations were software design education only. We offered senior architect training and graduate level software design courses to several well-known companies including Samsung and Comcast. Our contacts and our trainees from these companies were all experienced architects with years of real-world experience in software design, but the state of knowledge of the development teams and the average developers who would actually use our tools was problematic.

In our experience, garnered from years of interaction with multiple Fortune 500 companies, with small companies, and with open source projects, the average software developer lacks even the most basic knowledge about design concepts. They often do not understand what fan-in and fan-out are and why these matter, nor the benefits of creating a hierarchical structure of dependencies among modules. They often are unaware of the problems associated with large code files, and they almost exclusively use inheritance for code reuse. Design patterns are often poorly understood and wrongly applied. Once we observed a case where one object created over 1,000 other objects, forming a huge and expensive crossing. But the team leader argued that this was a "factory" pattern. To their understanding, a "factory" pattern means that an object should "manufacture" multiple objects. We could not convince the team that this was wrong, because there are online blogs defining "factory" in this way, even though there is not such a "factory" pattern in the seminal "Gang of Four" design pattern book [62]. In another case, there was a file containing numerous other functions, and the team leader said it was a "composite" pattern, meaning that many functions are composed together. This is just a list of the most egregious misunderstandings that we have encountered. In these cases, it was impossible for the team to adopt our tool because the "patterns" they applied will be identified as anti-patterns, and they believed that our tool was wrong. It is only through intensive interaction, and argumentation, often augmented by data collection, that these misunderstandings can be corrected. We have seen, over and over, fundamental failings in the education of software developers.

Others have noted that teaching design concepts is a challenge, e.g. [110]. We have attempted to address shortcomings in software engineering education ourselves, but our results were not encouraging [93]. We recognize that there is no quick fix for this problem.

4.2. The gap between software engineering research and practice

Our industrial collaboration experiences also revealed the surprising gap between software engineering research and practice. Few developers, even team leaders, fully understood (or were even aware of) canonical concepts such as Parnas' information hiding, software families, or hierarchical structure. Even though the UML was developed decades ago and has been intensively promoted, box-and-line style diagrams are still the most popular way for developers to communicate their "architecture", if any notation is used at all. In our recent study at Google, we employed UML component and sequence diagrams to reason about a significant re-architecting decision [111]. In this process, we realized that these basic UML models are still new to most developers. If these foundational concepts in software research have not been accepted or adopted after so many decades and numerous publications and supporting tools, how long and what will it take for practitioners to learn and widely adopt design rule theory, options theory, DRH, DRSpace, DSM, and anti-patterns? We cannot be too hopeful that adoption by "the majority" will happen any time soon. The question is: *what are the major obstacles that prevent the adopt of not only our technologies, but also of the many foundational design concepts that have been proposed by the software engineering research community for decades?* The following are the four gaps that we have determined, based on our experiences.

4.3. The dilemma of software metrics: What we offer and what they need

Through all of our interactions with companies we have heard one common request: *how to quantitatively and continuously measure and monitor design quality?* Our contacts in these companies, who are either senior architects or researchers, were familiar with various metrics proposed in our research community, such as Cyclomatic Complexity [49], C&K Metrics [97], or the maintainability scores provided by commercial tools such as SonarQube. Our communications with our collaborators revealed that developers are usually not convinced that (1) these metrics can faithfully reflect the quality of their design: they cannot be used to compare different projects or monitor the evolution of one project; (2) they cannot provide guidance on how to improve their design.

We proposed the DL metric to address some of these problems at the request our industrial partners. Our early experiments were quite successful: as reported in [10,112], and [106], DL was able to faithfully compare the quality of different projects and monitor the evolution of a project. But this metric was not always useful for all projects from all companies. For example, in the BrightSquid case study [107], although the design before refactoring had a significant number of anti-patterns and suffered from severe technical debt, the DL scores of the "before" and "after" designs were similar. Our experiences revealed that the DL scores can be distorted by the programming languages and frameworks in use. In particular, programs using modern, dynamic languages are more likely to have higher DL scores than programs written in C and C++ [113]. In other words, dynamic typing, polymorphism, and dynamic binding frameworks can cause DL inflation. In these cases, a high DL score does not necessarily mean that the system is easy

to maintain, and those systems often have more modularity violations caused by cloned code or implicit assumptions.

The final challenge is that DV8 users often expect that once the development team reduces the instances of anti-patterns, the DL scores will improve automatically. Some companies even set a target DL and required all the teams to reach it, or set DL score improvement as part of the developers' objectives and key results (OKR). We then explained that DL and anti-patterns are complementary and are defined based on different data sources. We advised them not to use DL, or any other single code-based metric, as the sole maintainability measure because it is defined on structural dependencies only. Instead, the team should continuously monitor productivity and quality changes, together with the reduction of anti-patterns, and variations in DL scores. The team leaders would still ask: *"If so, can DV8 provide concrete refactoring suggestions to improve DL?"* We realized that what the practitioners really needed was not only a metric but also concrete refactoring guidance on how to improve that metric. They also needed evidence that the improved metric score would lead to real improvements in productivity and quality. Our quest for effective software metrics is not over yet.

4.4. The challenge of productivity measures

Throughout our research we have been challenged to provide evidence that the anti-patterns that we have identified really matter, and that better or worse DL scores really do correlate with better or worse architectures. How would we go about gathering evidence to support such hypotheses? Clearly we needed to measure project quality and effort, to see if, for example, the presence of flaws actually matters.

The problem is that few projects capture effort, and even those that do capture effort admit that the measures that they capture are often inaccurate (for example, developers make up effort numbers at the end of the week, for their timesheets, rather than capturing actual effort in real time). For this reason we have turned to objective measures of productivity: bugs, changes, and the churn (committed lines of code) related to the fixing of bugs and the implementation of changes. This data can be mined from a project's commit history and issue tracker.

These measures are clearly not perfect proxies for effort. We can all recall times when a few lines of code required enormous time and effort (e.g. debugging a complex algorithm) and other times when many lines of code required little effort (e.g. creating a large but regular switch statement). And there is no reason to expect that the sizes of commits are anything but random, following a power law distribution [114] and so such differences should wash out over time and many commits. Furthermore, these measures are objective, widely captured by existing software projects, and clearly related to effort, on average.

And so, while high quality project effort data is virtually impossible to come by, we feel confident that our proxy measures of effort – bugs, changes, and churn – give us the insight that we need to analyze design debt. Furthermore, our interactions with developers and management have confirmed this assumption over and over.

4.5. The gap between management and development

Despite what we believe to be a compelling body of evidence on the importance of design, the software industry remains largely indifferent. In our earlier industrial collaborations, our contacts were either in management or research positions. Once they introduced our work to the development team, it was up to the team to execute the tools and interpret the results. In the end, it is the relationship between management and development that determines if the organization will adopt these technologies, and we have observed three different categories of context here:

Case 1: Both management and development felt and acknowledged the maintenance difficulties. In this case, DV8 can be used effectively to quantity the amount and severity of technical debt, helping the organization to make informed decisions on refactoring. Most of our published case studies fall into this category.

Case 2: Management is concerned but the development team is indifferent. While developers may think about the design of their code modules, they rarely think more broadly than that, about architectural design. This is not surprising. Developers are, by and large, not incentivized to create good designs. They are incentivized to pump out features and fix bugs. When DV8 reported poor DL scores or anti-patterns such as cliques or package cycles, the developers would argue that their product had been like that for years, and it should not be a concern. Even if the data reveals that the problematic structure is incurring high maintenance costs, developers insisted that it is the right "pattern" or the architecture is too expensive to refactor, or that the tool is not accurate.

Case 3: The developers are concerned but the management team is indifferent. In this case, the developers experienced the difficulty of maintaining their software, but management had other priorities in mind. Few companies prioritize or incentive removing technical debt over creating new features, or maintaining other quality attributes, such as performance and security. Due to the lack of focus on design and design quality, and due to widespread ignorance about proper design principles, it can be difficult to convince managers to allocate the needed resources to programmers – principally time – for refactoring. In other words, even if they agree that there is design debt, as long as new features can be added and the product keeps running, they often chose to stick with the current design.

Note that these results all stem from case studies. Case study research cannot, by itself, address issues involving populations. And the nature of the intense collaboration required by case study research precludes wide-spread population studies. So there is no way for us to confidently answer the question of how common each of the above cases is. What we can say is that we have now collaborated with dozens of projects and over 10 companies spanning small to very large size, in multiple application domains, operating in very different markets, and we have not observed fundamental differences in their concerns relating technical debt and software design.

Note also that there is a potential threat to validity here. In each case these organizations came to us with their problems, having read our papers or attended a talk. This introduces a selection bias. But this selection bias should, if anything, work in our favor. That is to say that we could expect that this bias means that our partner companies would be more likely to become adopters than a random company. This has not, in fact, been the case.

Finally, we observe that people tend to value and optimize what they measure. To compound this problem people, as a whole, value short-term gains over long-term ones. This is why we tend to sit on the couch and eat sweets rather than getting up and exercising. It is part of human nature. And so, without a change in hearts and minds, perhaps catalyzed by education, we do not see the gap between management and development to close any time soon.

The only solution to this problem is education. If developers and managers are made aware of the negative consequences of their short-term decisions, they may be motivated to changes such practices. The widespread use of practices such as code reviews and unit testing, for example, is evidence that developers practices can and do change over time, with appropriate incentives. When people can see concrete benefits from a practice, they are more likely to adopt it. The challenge with design debt is that the link between cause and effect is less obvious than the link between code reviews or unit testing and reduced bugs.

4.6. The gap between symptom detection and treatment

Even if DV8 reports a significant potential return on investment for a refactoring, and a manager is convinced, there remains the final question: *how to refactor?* It is often the case that even the most well-intentioned programmers do not know where to start, and may

actually cause more harm than good in their refactorings. Most of our collaborators have agreed – once shown the data that we harvested from their projects – with the design debts that our tool has detected. That is, they agree that these are real debts causing real pain. But the tool itself does not explain how to *fix* these problems. More than once we were asked: *"Your tool reported that if refactored successfully, our productivity will increase by this much. But can your tool tell our developers how to refactor?"*

At this point, we have had to explain that the proper refactoring strategy, such as which patterns should be applied to remove which anti-pattern, should be determined based on both domain knowledge and how the system is expected to evolve in the future. There is, therefore, yet another a gap: between the design problems that we detect and existing refactoring tools that a developer might employ to resolve these problems. For example, no effective tools are available to remove unstable interfaces. It is still the developer's responsibility to decide how to refactor. Again, we are back to dealing with the lack of software education and awareness: in the end, as Brooks noted [115], raising good designers is our best hope to create high-quality designs, or high-quality refactoring strategies.

We can offer some useful guidance however. The architectural flaws that DV8 detects are all violations of well-known design principles, such as the SOLID principles. If we detect a modularity violation, this is a typically a violation of the single responsibility principle [61]. If we detect an unhealthy inheritance, this is a violation of Liskov substitutability [116]. And knowing this leads one to an obvious refactoring strategy. If there is a modularity violation, you need to modularize the shared responsibility, perhaps placing it in its own class that other classes can access. If there is an unhealthy inheritance, this is typically resolved by moving a method from a child class to the parent class. Thus while the identification of a flaw does not uniquely determine a refactoring strategy, it does provide clear direction.

4.7. Tool and process challenges

There are a number of other factors that we should enumerate that may have affected our success in transitioning our research ideas and tools to industry. First, we are a small research-focused group. Unlike professional consultants, we have little marketing experience and no marketing budget. This may have hindered awareness and hence adoption.

Second, our tool has an old-fashioned looking user interface, based on Java Swing, and we have certainly paid insufficient attention to tool usability. Perhaps an easier to use, more intuitive user interface would have made a difference in adoption. Moreover, the desktop tool alone is not sufficient to convey the software design knowledge needed to understand the output of DV8. Currently DV8 outputs a number of folders and spreadsheets. Our prior collaborations all started with intensive communications and training. In this training we explained basic design principles, how these principles, or their violations, can be represented in DSMs, and how to interpret the DSM of each anti-pattern. Similar to an MRI or CT scan image that can only be read and interpreted by trained doctors, the current output of DV8 can only be understood by architects who are familiar with DSMs and design principles, which hinders its widespread adoption by developers.

Finally, we rely on third-party reverse engineering tools to extract facts from a code base, such as Depends[1] and Understand.[2] We have observed that different tools report slightly different results from the same project. Thus, while we do not see this as a large threat to validity – these tools are widely used and validated – we do not have 100% confidence in the data that we are extracting from the projects under study. This could erode confidence in the results that we have produced.

[1] https://github.com/multilang-depends

[2] https://scitools.com/

5. Conclusions and the road ahead

Reflecting on our experiences in the past decade, we realize that the most significant hurdle for "the majority" to adopt our research outcome is for the majority to better adopt, understand, and apply design principles and established design strategies. This, in turn, requires improved software design education, which is a critical need. Thus we issue a call to action for the software engineering community: for improved university and professional training in the field of software design. Another challenge that we have noted is to bridge the gaps between software research and practice. One way to bridge this gap is to explore and keep evolving effective software metrics that cater to the needs of practitioners and their managers. In addition, we need to speak the right language – the language of practitioners – focusing on productivity and quality, to foster broad adoption. These reflections lead to the following four directions for our future research:

(1) Augmenting software design education. In the past decade, we have applied Titan and DV8 into our own software design courses. In 2011 [92] and 2013 [93], we studied using DSMs to assess the modularity of student software. We did this by comparing the differences between the DSM representing the intended design and the DSMs of the software actually implemented by the students. The results showed that even though the lab and homework assignments were of small scale and in most cases detailed designs were given to the students in the form of UML class diagrams, as many as 74% of the student submissions, although they fulfilled the required functionality, introduced unexpected dependencies so that the intended modular structure was undermined. Our study shows that even students who understand the importance of modularity and have excellent programming skills may introduce additional harmful dependencies in their implementations, and it is hard for them to detect the existence of these dependencies on their own. We need to educate our students to think about technical debt – how to find it and fix it [42] – and the costs and benefits of refactoring. After applying DV8 to our own classrooms over the past few years, the benefits of tool-support in software design education became evident. It is possible to visualize the traditionally abstract design concepts and patterns, and we can make them quantifiable and operable. We plan to create more teaching materials, examples, and real-world case studies distilled from our industrial collaboration experiences, and better disseminate this new software design teaching methodology.

(2) Bridging the gap between software design research and practice. As we have experienced with our DL metric, given the evolution of new programming languages, frameworks, and paradigms, software metrics (and other research outcomes) need to improve and evolve continuously along with software practice. Otherwise, our research methods and results will soon become obsolete. Moreover, an effective metric should not only support cross-project comparison and continuous monitoring, it should also provide improvement guidance. We plan to further improve DL and develop other complementary measurement algorithms so that a user of the metric suite can be confident that an improvement in the metrics will truly lead to better project outcomes.

(3) Productivity and quality focused design debt detection. Our decade-long set of industrial experiences has revealed that, although management and developers always have different concerns and speak different languages, the common language they share is one of productivity and quality measures. To enable broader adoption of our technologies, we should highlight the design problems that directly impact productivity and quality scores, and quantitatively manifest their impact, rather than merely telling the team that their designs violate design principles.

(4) We intend to go back to the companies that adopted Titan and DV8 and interview them to understand in what ways, if any, their development culture has changed as a result of these interventions.

Finally, we believe that we would benefit from more cross-disciplinary collaboration with, among others, the refactoring, software

project management, and software economics research communities. This will helps us make software design a principled and quantifiable discipline that can actually guide and benefit design practice in the long run.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgments

This research was supported by the United States National Science Foundation grants 1817267, 1514561, 1140300, 1065242, 1835292, 1823177, and 2213764.

We owe an enormous debt of gratitude to our many students and collaborators in this research journey: Humberto Cervantes, Hongzhou Fang, Qiong Feng, Serge Haziyev, Wuxia Jin, Jason Lefever, Ran Mo, Martin Naedele, Maleknaz Nayebi, Will Snipes, Kevin Sullivan, Sunny Wong, and Lu Xiao.

References

[1] C.Y. Baldwin, K.B. Clark, Design Rules, Vol. 1: The Power of Modularity, MIT Press, 2000.
[2] M. Fowler, Refactoring: Improving the Design of Existing Code, Addison-Wesley Longman Publishing Co., Inc., 1999.
[3] L. Xiao, Y. Cai, R. Kazman, Design rule spaces: A new form of architecture insight, in: Proc. 36th International Conference on Software Engineering, 2014.
[4] Y. Cai, L. Xiao, R. Kazman, R. Mo, Q. Feng, Design rule spaces: A new model for representing and analyzing software architecture, IEEE Trans. Softw. Eng. 45 (7) (2019) 657–682.
[5] D.L. Parnas, On the criteria to be used in decomposing systems into modules, Commun. ACM 15 (12) (1972) 1053–1058.
[6] S. Blumberg, R. Das, J. Lansing, N. Motsch, B. Münsterma, R. Patenge, Demystifying digital dark matter: A new standard to tame technical debt, 2022, [Online]. Available: https://www.mckinsey.com/capabilities/mckinsey-digital/our-insights/demystifying-digital-dark-matter-a-new-standard-to-tame-technical-debt.
[7] L. Xiao, Y. Cai, R. Kazman, Titan: A toolset that connects software architecture with quality analysis, in: 22nd ACM SIGSOFT International Symposium on the Foundations of Software Engineering, 2014.
[8] Y. Cai, R. Kazman, DV8: Automated architecture analysis tool suites, in: 2019 IEEE/ACM International Conference on Technical Debt, TechDebt, 2019, pp. 53–54.
[9] K.J. Sullivan, Y. Cai, B. Hallen, W.G. Griswold, The structure and value of modularity in design, ACM SIGSOFT Softw. Eng. Notes 26 (5) (2001) 99–108.
[10] R. Mo, Y. Cai, R. Kazman, L. Xiao, Q. Feng, Decoupling level: A new metric for architectural maintenance complexity, in: Proc. 38rd International Conference on Software Engineering, 2016.
[11] L. Xiao, R. Kazman, Y. Cai, R. Mo, Q. Feng, Detecting the locations and predicting the costs of compound architectural debts, IEEE Trans. Softw. Eng. 48 (9) (2022).
[12] R. Mo, Y. Cai, R. Kazman, L. Xiao, Hotspot patterns: The formal definition and automatic detection of recurring high-maintenance architecture issues, in: Proc. 12th Working IEEE/IFIP International Conference on Software Architecture, 2015.
[13] R. Mo, Y. Cai, L. Xiao, R. Kazman, Q. Feng, Architecture anti-patterns: Automatically detectable violations of design principles, IEEE Trans. Softw. Eng. 47 (5) (2021).
[14] R. Kazman, Y. Cai, R. Mo, Q. Feng, L. Xiao, S. Haziyev, V. Fedak, A. Shapochka, A case study in locating the architectural roots of technical debt, in: Proc. 37th International Conference on Software Engineering, 2015.
[15] M. Mantyla, J. Vanhanen, C. Lassenius, A taxonomy and an initial empirical study of bad smells in code, in: International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings, 2003, pp. 381–384.
[16] W. Kessentini, M. Kessentini, H. Sahraoui, S. Bechikh, A. Ouni, A cooperative parallel search-based software engineering approach for code-smells detection, IEEE Trans. Softw. Eng. 40 (9) (2014) 841–861.
[17] M. Abbes, F. Khomh, Y.-G. Gueheneuc, G. Antoniol, An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension, in: Proc. 15thEuropean Conference on Software Maintenance and Reengineering, 2011, pp. 181–190.
[18] N. Moha, Y.-G. Guéhéneuc, A.-F. Le Meur, L. Duchien, A domain analysis to specify design defects and generate detection algorithms, in: Proc. 11th International Conference on Fundamental Approaches To Software Engineering, 2008, pp. 276–291.
[19] N. Tsantalis, A. Chatzigeorgiou, Identification of move method refactoring opportunities, IEEE Trans. Softw. Eng. 35 (3) (2009) 347–367.
[20] Y. Higo, T. Kamiya, S. Kusumoto, K. Inoue, Refactoring support based on code clone analysis, in: Proc. 5th International Conference on Product Focused Software Development and Process Improvement, 2004, pp. 220–233.
[21] M. Lippert, S. Roock, Refactoring in Large Software Projects: Performing Complex Restructurings Successfull, Wiley, 2006.
[22] J. Garcia, D. Popescu, G. Edwards, N. Medvidovic, Toward a catalogue of architectural bad smells, in: Proceedings of the 5th International Conference on the Quality of Software Architectures: Architectures for Adaptive Software Systems, 2009, pp. 146–162.
[23] J. Garcia, D. Popescu, G. Edwards, N. Medvidovic, Identifying architectural bad smells, in: Proc. 13th European Conference on Software Maintenance and Reengineering, 2009, pp. 255–258.
[24] D. Le, N. Medvidovic, Architectural-based speculative analysis to predict bugs in a software system, in: Proceedings of the 38th International Conference on Software Engineering Companion, 2016, pp. 807–810.
[25] T. Sharma, P. Mishra, R. Tiwari, Designite - A software design quality assessment tool, in: 2016 IEEE/ACM 1st International Workshop on Bringing Architectural Design Thinking Into Developers' Daily Activities, BRIDGE, 2016, pp. 1–4.
[26] F.A. Fontana, I. Pigazzini, R. Roveda, M. Zanoni, Automatic detection of instability architectural smells, in: 2016 IEEE International Conference on Software Maintenance and Evolution, ICSME, 2016, pp. 433–437.
[27] F.A. Fontana, I. Pigazzini, R. Roveda, D. Tamburri, M. Zanoni, E.D. Nitto, Arcan: A tool for architectural smells detection, in: 2017 IEEE International Conference on Software Architecture Workshops, ICSAW, 2017, pp. 282–285.
[28] M. Fowler, Refactoring: Improving the Design of Existing Code, Addison-Wesley, 1999.
[29] J. Garcia, D. Popescu, G. Edwards, N. Medvidovic, Identifying architectural bad smells, in: 2009 13th European Conference on Software Maintenance and Reengineering, IEEE, 2009, pp. 255–258.
[30] S. SonarSource, SonarQube, 2013, Capturado em: http://www.sonarqube.org.
[31] Designite. [Online]. Available: https://www.designite-tools.com/.
[32] Structure101. [Online]. Available: https://structure101.com/.
[33] N. Sangal, E. Jordan, V. Sinha, D. Jackson, Using dependency models to manage complex software architecture, in: Proc. 20th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, 2005, pp. 167–176.
[34] F.A. Fontana, I. Pigazzini, R. Roveda, D.A. Tamburri, M. Zanoni, E.D. Nitto, Arcan: A tool for architectural smells detection, in: 2017 IEEE International Conference on Software Architecture Workshops, ICSA Workshops 2017, Gothenburg, Sweden, April 5-7, 2017, IEEE Computer Society, 2017, pp. 282–285, http://dx.doi.org/10.1109/ICSAW.2017.16.
[35] J. Lefever, Y. Cai, H. Cervantes, R. Kazman, H. Fang, On the lack of consensus among technical debt detection tools, in: 2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice, ICSE-SEIP, 2021, pp. 121–130.
[36] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, A. De Lucia, A large-scale empirical study on the lifecycle of code smell co-occurrences, Inf. Softw. Technol. 99 (2018) 1–10, [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584918300211.
[37] D. Johannes, F. Khomh, G. Antoniol, A large-scale empirical study of code smells in JavaScript projects, Softw. Qual. J. 27 (2019) 1271–1314.
[38] S. Goularte Carvalho, M. Aniche, J. Veríssimo, R. Durelli, M.A. Gerosa, An empirical catalog of code smells for the presentation layer of android apps an empirical catalog of code smells for the presentation layer of android apps, Empir. Softw. Eng. 24 (2019).
[39] B.A. Muse, M.M. Rahman, C. Nagy, A. Cleve, F. Khomh, G. Antoniol, On the prevalence, impact, and evolution of SQL code smells in data-intensive systems, in: Proceedings of the 17th International Conference on Mining Software Repositories, MSR '20, Association for Computing Machinery, New York, NY, USA, 2020, pp. 327–338, http://dx.doi.org/10.1145/3379597.3387467.
[40] B. van Oort, L. Cruz, M. Aniche, A. van Deursen, The prevalence of code smells in machine learning projects, 2021.
[41] I.O. Philippe Kruchten, Managing Technical Debt: Reducing Friction in Software Development, Addison-Wesley, 2005.
[42] N. Ernst, J. Delange, R. Kazman, Technical Debt in Practice—How to Find It and Fix It, MIT Press, 2021.
[43] N.A. Ernst, S. Bellomo, I. Ozkaya, R.L. Nord, I. Gorton, Measure it? Manage it? Ignore it? Software practitioners and technical debt, in: Proceedings of the Joint Meeting on Foundations of Software Engineering, 2015, pp. 50–60.

[44] D. Sas, P. Avgeriou, U. Uyumaz, On the evolution and impact of architectural smells—an industrial case study, Empir. Softw. Eng. 27 (4) (2022) 86.
[45] A. Martini, F.A. Fontana, A. Biaggi, R. Roveda, Identifying and prioritizing architectural debt through architectural smells: a case study in a large software company, in: Software Architecture: 12th European Conference on Software Architecture, ECSA 2018, Madrid, Spain, September 24–28, 2018, Proceedings 12, Springer, 2018, pp. 320–335.
[46] F.A. Fontana, F. Locatelli, I. Pigazzini, P. Mereghetti, Speaker, An architectural smell evaluation in an industrial context, 2020, [Online]. Available: https://api.semanticscholar.org/CorpusID:237560358.
[47] A. Yamashita, L. Moonen, Do developers care about code smells? An exploratory survey, in: 2013 20th Working Conference on Reverse Engineering, WCRE, 2013, pp. 242–251.
[48] R.L. Nord, I. Ozkaya, P. Kruchten, M. Gonzalez-Rojas, In search of a metric for managing architectural technical debt, in: 2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture, IEEE, 2012, pp. 91–100.
[49] T.J. McCabe, A complexity measure, IEEE Trans. Softw. Eng. 2 (4) (1976) 308–320.
[50] M.H. Halstead, Elements of Software Science, in: Operating and Programming Systems Series, Elsevier Science Inc., 1977.
[51] S.R. Chidamber, C.F. Kemerer, A metrics suite for object oriented design, IEEE Trans. Softw. Eng. 20 (6) (1994) 476–493.
[52] M. Lorenz, J. Kidd, Object-Oriented Software Metrics, Prentice Hall, 1994, p. 146.
[53] F.B. e Abreu, The MOOD metrics set, in: Proc. ECOOP'95 Workshop on Metrics, 1995.
[54] S.C. Misra, Modeling design/coding factors that drive maintainability of software systems, Softw. Qual. Control 13 (3) (2005) 297–320.
[55] R. Harrison, S.J. Counsell, R.V. Nithi, An investigation into the applicability and validity of object-oriented design metrics, Empir. Softw. Eng. 3 (3) (1998) 255–273.
[56] W. Li, S. Henry, Object-oriented metrics that predict maintainability, J. Syst. Softw. 23 (2) (1993) 111–122.
[57] M.P. Ware, F.G. Wilkie, M. Shapcott, The application of product measures in directing software maintenance activity, J. Softw. Maint. 19 (2) (2007) 133–154.
[58] S. Wong, Y. Cai, M. Kim, M. Dalton, Detecting software modularity violations, in: Proc. 33rd International Conference on Software Engineering, 2011, pp. 411–420.
[59] R. Mo, Y. Cai, R. Kazman, Q. Feng, Assessing an architecture's ability to support feature evolution, in: Proceedings of the 26th International Conference on Program Comprehension, 2018.
[60] R. Kazman, Y. Cai, R. Mo, Q. Feng, L. Xiao, S. Haziyev, V. Fedak, A. Shapochka, A case study in locating the architectural roots of technical debt, in: Proc. 37th International Conference on Software Engineering, 2015.
[61] R. Martin, Clean Architecture: A Craftsman's Guide to Software Structure and Design, Prentice Hall, 2017.
[62] E. Gamma, R. Helm, R.J. andJohn Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1994.
[63] C. Hu, The nature of software design and its teaching: An exposition, ACM Inroads 4 (2) (2013) 62–72, http://dx.doi.org/10.1145/2465085.2465103.
[64] A. Baker, A. van der Hoek, H. Ossher, M. Petre, Guest editors' introduction: Studying professional software design, IEEE Softw. 29 (1) (2011) 28–33.
[65] D.E. Perry, A.L. Wolf, Foundations for the study of software architecture, ACM SIGSOFT Softw. Eng. Notes 17 (4) (1992) 40–52.
[66] A. Eckerdal, R. McCartney, J.E. Moström, M. Ratcliffe, C. Zander, Can graduating students design software systems? ACM SIGCSE Bull. 38 (1) (2006) 403–407.
[67] C. Loftus, L. Thomas, C. Zander, Can graduating students design: revisited, in: Proceedings of the 42nd ACM Technical Symposium on Computer Science Education, 2011, pp. 105–110.
[68] D. Mazaitis, The object-oriented paradigm in the undergraduate curriculum: a survey of implementations and issues, ACM SIGCSE Bull. 25 (3) (1993) 58–64.
[69] P. Prasad, S. Iyer, VeriSIM: A model-based learning pedagogy for fostering software design evaluation skills in computer science undergraduates, Res. Pract. Technol. Enhanc. Learn. 17 (1) (2022) 1–35.
[70] V.Y. Sien, An investigation of difficulties experienced by students developing unified modelling language (UML) class and sequence diagrams, Comput. Sci. Educ. 21 (4) (2011) 317–342.
[71] S.P. Linder, D.S. Abbott, M.J. Fromberger, An instructional scaffolding approach to teaching software design, J. Comput. Sci. Coll. 21 (2006) 238–250, [Online]. Available: https://api.semanticscholar.org/CorpusID:59812777.
[72] S. Ramnath, B. Dathan, Evolving an integrated curriculum for object-oriented analysis and design, in: Technical Symposium on Computer Science Education, 2008, [Online]. Available: https://api.semanticscholar.org/CorpusID:14668986.
[73] I. Warren, Teaching patterns and software design, in: IFAC Symposium on Advances in Control Education, 2005, [Online]. Available: https://api.semanticscholar.org/CorpusID:5430485.
[74] G. Booch, J. Rumbaugh, I. Jacobson, Unified Modeling Language User Guide, second ed., Addison-Wesley, 2005.
[75] G. Scanniello, C. Gravino, M. Genero, J.A. Cruz-Lemus, G. Tortora, On the impact of UML analysis models on source-code comprehensibility and modifiability, ACM Trans. Softw. Eng. Methodol. 23 (2) (2014) http://dx.doi.org/10.1145/2491912.
[76] P.H. Feiler, B. Lewis, S. Vestal, E. Colbert, An overview of the SAE architecture analysis & design language (AADL) standard: A basis for model-based architecture-driven embedded systems engineering, in: P. Dissaux, M. Filali-Amine, P. Michel, F. Vernadat (Eds.), Archit. Descr. Lang., Springer US, Boston, MA, 2005, pp. 3–15.
[77] D. Garlan, R. Monroe, D. Wile, ACME: An architecture description interchange language, in: CASCON First Decade High Impact Papers, CASCON '10, IBM Corp., USA, 2010, pp. 159–173, http://dx.doi.org/10.1145/1925805.1925814.
[78] N. Medvidovic, R. Taylor, A classification and comparison framework for software architecture description languages, IEEE Trans. Softw. Eng. 26 (1) (2000) 70–93, http://dx.doi.org/10.1109/32.825767.
[79] E. Júnior, K. Farias, B. Silva, A survey on the use of UML in the Brazilian industry, in: Proceedings of the XXXV Brazilian Symposium on Software Engineering, SBES '21, Association for Computing Machinery, New York, NY, USA, 2021, pp. 275–284, http://dx.doi.org/10.1145/3474624.3474632.
[80] M. Petre, UML in practice, in: 2013 35th International Conference on Software Engineering, ICSE, 2013, pp. 722–731.
[81] W.J. Dzidek, E. Arisholm, L.C. Briand, A realistic empirical evaluation of the costs and benefits of UML in software maintenance, IEEE Trans. Softw. Eng. 34 (3) (2008) 407–432, http://dx.doi.org/10.1109/TSE.2008.15.
[82] B. Bafandeh Mayvan, A. Rasoolzadegan, Z. Ghavidel Yazdi, The state of the art on design patterns: A systematic mapping of the literature, J. Syst. Softw. 125 (2017) 93–118, [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121216302321.
[83] M. Hong, T. Xie, F. Yang, Jbooret: an automated tool to recover oo design and source models, in: 25th Annual International Computer Software and Applications Conference, COMPSAC 2001, IEEE, 2001, pp. 71–76.
[84] Z.-X. Zhang, Q.-H. Li, K.-R. Ben, A new method for design pattern mining, in: Proceedings of 2004 International Conference on Machine Learning and Cybernetics (Ieee Cat. No. 04ex826), Vol. 3, IEEE, 2004, pp. 1755–1759.
[85] G. Costagliola, A. De Lucia, V. Deufemia, C. Gravino, M. Risi, Case studies of visual language based design patterns recovery, in: Conference on Software Maintenance and Reengineering, CSMR'06, IEEE, 2006, pp. 10–pp.
[86] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, S.T. Halkidis, Design pattern detection using similarity scoring, IEEE Trans. Softw. Eng. 32 (11) (2006) 896–909.
[87] P. Andritsos, V. Tzerpos, Information-theoretic software clustering, IEEE Trans. Softw. Eng. 31 (2) (2005) 150–165.
[88] S. Mancoridis, B.S. Mitchell, Y. Chen, E.R. Gansner, Bunch: a clustering tool for the recovery and maintenance of software system structures, in: Proc. IEEE Intl. Conf. Software Maintenance, 1999, pp. 50–59.
[89] V. Tzerpos, R.C. Holt, ACCD: an algorithm for comprehension-driven clustering, in: Proc. Seventh Working Conference on Reverse Engineering, 2000, pp. 258–267.
[90] T. Lutellier, D. Chollak, J. Garcia, L. Tan, D. Rayside, N. Medvidović, R. Kroeger, Measuring the impact of code dependencies on software architecture recovery techniques, IEEE Trans. Softw. Eng. 44 (2) (2018) 159–181.
[91] T. Lutellier, D. Chollak, J. Garcia, L. Tan, D. Rayside, N. Medvidovic, R. Kroeger, Comparing software architecture recovery techniques using accurate dependencies, in: Proc. 37th International Conference on Software Engineering, 2015.
[92] Y. Cai, D. Iannuzzi, S. Wong, Leveraging design structure matrices in software design education, in: Proc. 24th Conference on Software Engineering Education and Training, 2011, pp. 179–188.
[93] Y. Cai, R. Kazman, C. Jaspan, J. Aldrich, Introducing tool-supported architecture review into software design education, in: Proc. 26th Conference on Software Engineering Education and Training, 2013.
[94] S. Wong, Y. Cai, G. Valetto, G. Simeonov, K. Sethi, Design rule hierarchies and parallelism in software development tasks, in: Proc. 24th IEEE/ACM International Conference on Automated Software Engineering, 2009, pp. 197–208.
[95] W. Cunningham, The WyCash portfolio management system, in: Addendum to Proc. 7th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, 1992, pp. 29–30.
[96] L. Xiao, Y. Cai, R. Kazman, R. Mo, Q. Feng, Identifying and quantifying architectural debts, in: Proc. 38rd International Conference on Software Engineering, 2016.
[97] S. Chidamber, C. Kemerer, A metrics suite for object oriented design, IEEE Trans. Softw. Eng. 20 (6) (1994) 476–493.
[98] M. Ó Cinnéide, L. Tratt, M. Harman, S. Counsell, I. Hemati Moghadam, Experimental assessment of software metrics using automated refactoring, in: International Symposium on Empirical Software Engineering and Measurement, ESEM, 2012, pp. 49–58.
[99] N.E. Fenton, S.L. Pfleeger, Software Metrics: A Rigorous and Practical Approach, second ed., PWS Publishing Co., Boston, MA, USA, 1998.

[100] Q. Feng, Y. Cai, R. Kazman, D. Cui, T. Liu, H. Fang, Active hotspot: An issue-oriented model to monitor software evolution and degradation, in: 2019 34th IEEE/ACM International Conference on Automated Software Engineering, ASE, 2019, pp. 986–997.
[101] H. Cervantes, R. Kazman, Software archinaut - a tool to understand architecture, identify technical debt hotspots and control its evolution, in: International Conference on Technical Debt, TechDebt'2020, 2020.
[102] Succinct Code Counter. [Online]. Available: https://github.com/boyter/scc.
[103] CodeScene Inc., CodeScene, 2004, https://codescene.io.
[104] R. Schwanke, L. Xiao, Y. Cai, Measuring architecture quality by structure plus history analysis, in: Proc. 35rd International Conference on Software Engineering, 2013, pp. 891–900.
[105] R. Mo, W.S.Y. Cai, S. Ramaswamy, R. Kazman, M. Naedele, Experiences applying automated architecture analysis tool suites, in: Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering, 2018.
[106] W. Wu, Y. Cai, R. Kazman, R. Mo, Z. Liu, R. Chen, Y. Ge, W. Liu, J. Zhang, Software architecture measurement—Experiences from a multinational company, in: C.E. Cuesta, D. Garlan, J. Pérez (Eds.), Software Architecture, Springer International Publishing, 2018, pp. 303–319.
[107] M. Nayebi, Y. Cai, R. Kazman, G. Ruhe, Q. Feng, C. Carlson, F. Chew, A longitudinal study of identifying and paying down architecture debt, in: 2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice, ICSE-SEIP, 2019.
[108] R. Kazman, M. Barbacci, M. Klein, S.J. Carriere, S. Woods, Experience with performing architecture tradeoff analysis, in: Proceedings of the 21st International Conference on Software Engineering, 1999, pp. 54–63.
[109] R. Kazman, L. Bass, Making architecture reviews work in the real world, IEEE Softw. 19 (1) (2002) 67–73, http://dx.doi.org/10.1109/52.976943.
[110] M. Galster, S. Angelov, What makes teaching software architecture difficult? in: Proceedings of the 38th International Conference on Software Engineering Companion, 2016, pp. 356–359.
[111] Q. Jia, Y. Cai, O. Çakmak, A model-based, quality attribute-guided architecture re-design process at google, in: 2023 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice, ICSE-SEIP, 2023.
[112] R. Mo, W. Snipes, Y. Cai, S. Ramaswamy, R. Kazman, M. Naedele, Experiences applying automated architecture analysis tool suites, in: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, in: ASE 2018, Association for Computing Machinery, New York, NY, USA, 2018, pp. 779–789, http://dx.doi.org/10.1145/3238147.3240467.
[113] W. Jin, Y. Cai, R. Kazman, G. Zhang, Q. Zheng, T. Liu, Exploring the architectural impact of possible dependencies in python software, in: 2020 35th IEEE/ACM International Conference on Automated Software Engineering, ASE, 2020, pp. 758–770.
[114] O. Arafat, D. Riehle, The commit size distribution of open source software, in: 2009 42nd Hawaii International Conference on System Sciences, 2009, pp. 1–8.
[115] F. Brooks, No silver bullets: Essence and accidents of software engineering, SIAM J. Comput. 20 (4) (1987) 10–19.
[116] B. Liskov, J. Wing, A behavioral notion of subtyping, ACM Trans. Program. Lang. Syst. 16 (6) (1994) 1811–1841.