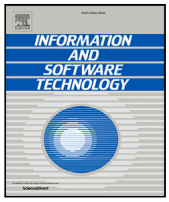
Information and Software Technology 170 (2024) 107429

Contents lists available at ScienceDirect

Information and Software Technology

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



Prevalence and severity of design anti-patterns in open source programs—A large-scale study

Alan Liu [a,*], Jason Lefever [b], Yi Han [c], Yuanfang Cai [b]

[a] Germantown Academy, United States of America
[b] Drexel University, United States of America
[c] Xi'an Jiaotong University, China

ARTICLE INFO

Keywords:
Software design
Anti-pattern
Technical debt
Software evolution
Software maintenance

ABSTRACT

Context: Design anti-patterns can be symptoms of problems that lead to long-term maintenance difficulty. How should development teams prioritize their treatment? Which ones are more severe and deserve more attention? Does the impact of anti-patterns and general maintenance efforts differ with different programming languages?

Objective: In this study, we assess the prevalence and severity of anti-patterns in different programming languages and the impact of dynamic typing in Python, as well as the impact scopes of prevalent anti-patterns that manifest the violation of design principles.

Method: We conducted a large-scale study of anti-patterns using 1717 open-source projects written in Java, C/C++, and Python. For the 288 Python projects, we extracted both explicit and dynamic dependencies and compared how the detected anti-patterns and maintenance costs changed. Finally, we removed anti-patterns involving five or fewer files to assess the impact of trivial anti-patterns.

Results: The results reveal that 99.55% of these projects contain anti-patterns. Modularity Violation – frequent co-changes among seemingly unrelated files – is most prevalent (detected in 83.54% of all projects) and costly (incurred 61.55% of maintenance effort on average). Unstable Interface and Crossing, caused by influential but unstable files, although not as prevalent, tend to incur severe maintenance costs. Duck typing in Python incurs more anti-patterns, and the churn spent on Python files multiplies that of C/C++ and Java files. Several prevalent anti-patterns have a large portion of trivial instances, meaning that these common symptoms are usually not harmful.

Conclusion: Implicit and visible dependencies are the most expensive to maintain, and dynamic typing in Python exacerbates the issue. Influential but unstable files need to be monitored and rectified early to prevent the accumulation of high maintenance costs. The violations of design principles are widespread, but many are not high-maintenance.

1. Introduction

The software industry has long recognized that sub-optimal structures and relations in source code, often defined as "*smells*" or "*anti-patterns*", may not introduce bugs or defects immediately but can significantly reduce the quality and productivity of the project in the long run. Cunningham [1] coined this phenomenon as *technical debt*. In the past decades, researchers have proposed definitions of smells and anti-patterns at various levels: code smell/anti-patterns indicate improper relations among program entities within source files, design smells/anti-patterns concern the relation among files, and architecture smells/anti-patterns represent symptoms among components. These definitions are usually based on the violations of design principles and supported by various tools, such as SonarQube [2], Designite [3], Arcan [4], and DV8 [5].

Fowler's code smells [6], such as god class, long methods and feature envy, are most well-known, and researchers have studied their prevalence in a large number of open source projects [7,8] and in various domains [9–11]. Researchers have also reported that prevailing tools often detect a large number of code smells, many of which are false positives and ignored by developers [12]. Multiple studies

* Corresponding author.
E-mail address: alanl200511@gmail.com (A. Liu).

https://doi.org/10.1016/j.infsof.2024.107429
Received 3 September 2023; Received in revised form 27 February 2024; Accepted 28 February 2024
Available online 1 March 2024

revealed that issues in higher-level design and architecture structures are the primary source of code-level technical debt [13–15]. However, there is no large-scale prevalence and severity analysis for these architecture and design-level smells/anti-patterns. Researchers often evaluate their definitions and detection techniques using about a dozen open source projects [16–18] primarily written in one programming language. They have also reported various industrial case studies [19–24] where these anti-patterns can be detected.

The question is: considering design anti-patterns as symptoms of design debt, how prevalent are these symptoms in software systems written in different programming languages? Are some of them more severe than others and deserve more attention? Are prevalent anti-patterns always worth treatment? Anti-patterns are usually defined based on the violation of design principles, and we notice that certain types of violations are extremely common, such as cyclical dependencies among files and packages. The question is, are these violations inevitable, and are they always harmful? These questions motivated our large-scale anti-pattern prevalence and severity analysis.

Another motivation is that dynamic typing is the main feature of popular programming languages, such as Python. While previous studies on smaller projects have indicated that files involved in implicit dependencies in Python projects tend to be more expensive to maintain than Python files without implicit dependencies [25,26], the absence of large-scale research necessitated a broader investigation to confirm these findings in larger, long-lived Python projects, and avoid potential biases inherent in smaller scope studies on smaller projects. Most interestingly, since duck typing is the main feature of Python, the question is, compared with traditional programming languages, such as Java and C/C++, do Python projects, in general, more or less subject to anti-patterns and more or less expensive to maintain?

In this paper, we study the prevalence and severity of a suite of anti-patterns defined by Mo et al. [16]: Unstable Interface (UIF), Modularity Violation Group (MVG), Crossing (CRS), Unhealthy Inheritance Hierarchy (UIH), Clique (CLQ), and Package Cycle (PKC). Different from other architectural and design level smells and anti-patterns, three out of the six anti-patterns, UIF, MVG, and UIH, are defined using both structural information and evolution history and are more likely to identify real design debt that incurs high-maintenance costs. Moreover, the tool that detects these anti-patterns, DV8, can also report the maintenance costs of each instance of each anti-pattern, in the form of file percentage, churn, and churn percentage, making it possible to assess the severity of each type of these anti-patterns. DV8 uses Depends [27], a multi-language source code dependency extraction tool, to obtain file relations so that we can investigate and compare the prevalence and severity of anti-patterns of projects written in different languages. In this study, we analyzed 1717 open-source projects written in Java, C/C++, or Python. These projects have at least 100 files and evolved for more than one year. Concretely, we investigated the following research questions:

RQ1: *Which anti-pattern is the most prevalent, and does the prevalence differ for different programming languages?*

Since anti-patterns manifest the violation of design principles, the answer to this question will help us understand how well these design principles are followed, if at all.

RQ2: *Which anti-pattern incurs the most severe cost measured as the amount of maintenance churn, and does the severity differ for different programming languages?*

The answer to this question will help us understand the differences among anti-patterns and whether different programming languages present different symptoms: Is one programming language subject to certain symptoms more than others? It is also possible that certain anti-patterns are prevalent, but are not expensive to maintain, and hence not a significant problem.

RQ3: *How does dynamic typing support by newer languages affect the prevalence and severity of anti-patterns, as well as the overall maintenance costs?*

In particular, popular programming languages, such as Python, support duck typing; that is, the type of an entity will be inferred at run-time so that the developers can write fewer LOCs and make the programs more concise and easier to understand. The question is, can dynamic typing really make a program easier to maintain? To answer this question, for the 288 Python projects we studied, we analyzed their anti-patterns using both explicit dependencies and *possible dependencies* [25], that is, dependencies statistically inferred from duck typing, and compared how the prevalence and severity of anti-patterns, as well as general maintenance costs, change with and without possible dependencies.

RQ4: *For highly prevalent anti-patterns, do they always impact a large number of files?*

Since an anti-pattern manifests the violation of design principles, a prevalent anti-pattern implies that the underlying design principles are rarely strictly followed. In addition to assessing the consequence of such violations in terms of churn, this question assesses their severity based on their impact scopes: if a significant portion of such anti-patterns only affect a small number of files, it implies that such violations are usually confined within a small scope, and should not be a significant concern.

To answer this question, we remove *trivial* anti-patterns and recount their prevalence among the 1717 projects. The rationale is that if there is a significant portion of projects that are only infected by trivial anti-patterns, it implies that although such violations are inevitable, these projects can manage to limit their impact scope.

Here we consider an anti-pattern instance as *trivial* if it influences five files or fewer, based on Gobet and Clarkson's study on cognitive complexity [28]: people can easily process approximately 5 "chunks" of information at a time. Similarly, in his famous paper, Miller [29] proposed a law of human cognition and information processing: *"Humans can effectively process no more than seven units, or chunks, of information, plus or minus two pieces of information, at any given time"*. Accordingly, it is safe to state that if an anti-pattern only influences five or fewer files, it will not significantly impair the developer's ability to understand and maintain them.

To the best of our knowledge, this is the first large-scale, multiple-language prevalence and severity study of design-level anti-patterns. The results reveal that the violation of design principles is very common but often without severe consequences, such as Package Cycle, Clique, and Unhealthy Inheritance. By contrast, anti-patterns caused by implicit dependencies, manifested as Modularity Violations, are both prevalent and high-maintenance. The widely used duck typing in Python makes more dependencies invisible, and the costs of maintaining Python programs can be much higher than those of C/C++ and Java projects. Unstable Interface and Crossing, caused by influential but change-prone files, are less prevalent but can incur a significant amount of maintenance costs. The results are consistent among all three programming languages.

We also notice that many instances of Unstable Interface and Crossing also contain large cycles or highly influential base classes, implying that even if the violations of design principles are not avoidable due to trade-offs for other quality attributes, such violations in larger scopes with growing impact and activities (which may transform into an UIF or CRS) should be monitored and avoided. Another implication is that the convenience of dynamic typing in Python may not be free. For long-lived, larger projects, Python projects could be more expensive to maintain. To prevent the accumulation of high maintenance costs, the development team should prioritize the treatment of these severe symptoms, reduce invisible dependencies, and monitor the emergence and growth of unstable but influential files.

2. Background

Technical debt detection tools aim to identify, locate, and measure concrete problematic instances found within a project's source

		1	2	3	4	5	6	7	8	9	10	11
1	Volume.h	–	0,2	0,2	2,2					0,2	0,2	
2	WbfsBlob.h	0,2	–	0,3	4,4	0,1	0,2	0,3	0,2		0,4	
3	CISOBlob.h	0,2	0,3	–	4,3	0,1	0,2	0,2	0,2		0,3	
4	Blob.h				–							
5	ISOFile.h	13,3	0,1	0,1	6,1	–	0,1	0,2	0,2		0,1	
6	VolumeGC.h	10,5	0,2	0,2	4,2	0,1	–	0,6	0,6		0,2	
7	VolumeWiiCrypted.h	6,8	0,3	0,2	4,3	0,2	0,6	–	0,8	0,3	0,3	0,1
8	VolumeWad.h	6,7	0,2	0,2	4,2	0,2	0,6	0,8	–	0,1	0,2	
9	WII_IPC_HLE_Device_es.cpp	0,2						0,3	0,1	–		0,1
10	DriveBlob.h	0,2	0,4	0,3	4,5	0,1	0,2	0,3	0,2		–	
11	Boot.cpp	11,2						0,1		0,1		–

Fig. 1. DSM of an Unstable Interface. Note: The file in red font is the leading file of the Unstable Interface, that is, it has many dependents and changes often with them.

code. Prominent commercial and academic examples include SonarQube [2], Designite [3], DV8 [5], Structure101 [30], Arcan [18], and Archinaut [31]. Prevailing tools are mostly limited to only searching for code-level issues: issues that are contained within a single file. Researchers have reported that problems at the design or architecture level are the major source of code-level technical debt [13–16]: most problems observed within a program entity stem from how program entities are related to one another.

Researchers have proposed various design and architecture smell definitions and detection techniques [32,33,33–35]. Mo et al. [16] introduce a collection of high-level issues called *architecture anti-patterns* that they have empirically shown to be indicators of excessive maintenance. These anti-patterns are defined in terms of the design rule theory of Baldwin and Clark [36] and a suite of well-known design principles. The rationale of their definition is to detect influential abstractions that can be seen as potential design rules [36], which may decouple the system into a collection of independent modules and leverage revision history to determine if these modules are truly independent. We chose to study the prevalence and severity of anti-patterns based on their definitions first because these concepts and the associated detection techniques have been applied and validated in multiple industrial settings [19–21,37,38]. Another reason is that these anti-patterns incorporate co-change history, more likely to identify real design debt that incurs extra maintenance penalty [39]. Their supporting tool, DV8 [5], reports the maintenance costs of each anti-pattern instance, in terms of file count, file percentage, churn, and churn percentage, making it possible to assess their severity.

An anti-pattern can be visualized using a *Design Structure Matrix* (DSM), a complementary visualization technique that can be used to display both structural and historical relationships between files as an adjacency matrix. The columns and rows of this matrix are labeled with the same set of files in the same order, and a marked cell indicates that the file on the row *depends* on or *co-changes* with the file on the column. For example, the ‘‘13, 3’’ in the cell at the intersection of the fifth row and first column in Fig. 1 means `ISOFile.h` has 13 structural dependencies, such as *Call* and *Use*, to `Volumn.h`, and these two files co-changed together 3 times as recorded in their revision history. The ‘‘0, 2’’ in the cell at the intersection of the first row and second column in Fig. 1 means `WbfsBlob.h` has zero structural dependencies on `Volume.h` but the two files have changed together twice. The co-change count has been omitted in some figures when not relevant. For each anti-pattern proposed by Mo et al. we provide a brief description and an example of a DSM.

- An *Unstable Interface* (UIF) is a file with a large number of dependent files that also changes frequently with those dependent files. In Fig. 1, `Volume.h` has 5 structural dependents while also changing frequently with all the other 10 files. Such influential but unstable files, which we call *leading files*, usually are the focal points of design debt, making all their relatives unstable and high-maintenance.
- A *Modularity Violation Group* (MVG) is a group of files where they frequently co-change with structurally unrelated peers, which we call *evolutionary coupling*. The higher the co-change frequency, the stronger the coupling. In Fig. 2, most pairs of files have a strong evolutionary relationship but without any structural connection—especially those involving `PowerPC.cpp`: none of the other 13 files explicitly depend on it, but they co-changed frequently, e.g., `Interpreter_LoadStore.cpp` changed 53 times with it. These frequent co-changes indicate the existence of implicit assumptions or hidden dependencies among them.
- A *Crossing* (CRS) is a file with a large number of dependent and dependee files that also changes frequently with those dependent and dependee files. In Fig. 3, `EmulationActivity.java` is at the center of many incoming and outgoing relationships as evidenced by row 10 and column 10 both having high structure counts and change frequencies. Similar to UIF, these *center files* in a CRS are usually the core of design debt.
- An *Unhealthy Inheritance Hierarchy* (UIH) is a group of files with one of the following two structures: a parent class depends on its sub-classes, or the client of the hierarchy uses both the parent and the children, which are empirically validated to be high-maintenance [16,20,37,40–42]. In Fig. 4, files 2 through 6 inherit from file 1. However, files 7 through 12 all depend on both the base class, file 1, and at least one of its child classes. For projects written in C, DV8 does not detect this anti-pattern.
- A *Clique* (CLQ) is a set of files where each file directly or indirectly depends on every other file, and the whole file group forms a strongly connected graph. In Fig. 5, all these 12 files form a clique, and changing any one of them may potentially cause all other files to change.
- A *Package Cycle* (PKC) is a pair of mutually dependent packages. In Fig. 6, the files 6–10 are defined in a separate package from the rest, yet there are dependencies going in both directions. While CLQ detects cycles among multiple files, PKC detects cycles between pair-wise packages, which violates the principle of forming a hierarchical structure to ease software extension and evolution [43,44].

Extracting various dependencies from source files is the first step before these anti-patterns can be detected. DV8 uses Depends [27] as a default preprocessor to extract dependency information from various programming languages. In a strongly typed language, such as Java or C/C++, the dependency structure can be determined at compile time. In a language that does not require data type declaration at compile time, such as Python, the exact dependency among files can only manifest at run time. Depends can also be configured to extract *possible dependencies* among Python files [25,26], that is, using type inference techniques to determine most dynamic dependencies at compile time. DV8 and Depends are the major tools we used to conduct the prevalence and severity analysis of these anti-patterns for projects written in multiple programming languages.

		1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	JitlLBase_LoadStore.cpp	–													
2	PowerPC.cpp		–												
3	Interpreter_Branch.cpp		0,2	–											
4	Interpreter_LoadStore.cpp		0,53		–										
5	CachedInterpreter.cpp		0,6	0,1		–									
6	Jit_Util.cpp						–								
7	Jit_LoadStore.cpp							–							
8	Jit_SystemRegisters.cpp		0,1						–						
9	Jit.cpp		0,8							–					0,1
10	JitArm64_LoadStore.cpp										–				
11	JitlL.cpp		0,9									–			
12	IR_X86.cpp		0,4				0,4					0,11	–		
13	Movie.cpp													–	0,30
14	Core.cpp		0,6											0,13	–

Fig. 2. DSM of a Modularity Violation. Note: There are no syntactical dependencies among these files, and they changed together frequently.

		1	2	3	4	5	6	7	8	9	10	11	12	13
1	LoadStateFragment.java	–					0,3	0,2	0,3		4,3			
2	NativeLibrary.java		–								5,11	0,5		0,2
3	Java_WlimoteAdapter.java		5,1	–				0,1		0,1				
4	Animations.java				–									
5	MainPresenter.java		3,2			–		0,2		0,2		0,5		
6	MenuFragment.java	0,3					–	0,3	0,3		4,5			
7	EmulationFragment.java	0,2	15,5	0,1		0,2	0,3	–	0,2	0,1	0,2			
8	SaveStateFragment.java	0,3					0,3	0,3	–		4,3			
9	Java_GCAdapter.java		9,0	0,1		0,2		0,1		–			0,2	
10	EmulationActivity.java	9,3	45,11	3,2	11,2	3,2	4,5	16,9	9,3	3,2	–			
11	GameAdapter.java		0,5			0,5		0,2			3,12	–	0,5	3,8
12	TvMainActivity.java					7,4				0,2	3,5	0,5	–	0,5
13	GameDetailsDialog.java		0,2								3,6		0,5	–

Fig. 3. DSM of a Crossing. Note: The file in red font is the center file of the Crossing, that is, it has high fan-in, high fan-out, and changed often with its relatives.

		1	2	3	4	5	6	7	8	9	10	11	12
1	SettingsItem.java	–											
2	SliderSetting.java	14	–										
3	CheckBoxSetting.java	10		–									
4	SingleChoiceSetting.java	10			–								
5	SubmenuSetting.java	3				–							
6	HeaderSetting.java	3					–						
7	SubmenuViewHolder.java	5				4		–	3				
8	SettingsAdapter.java	25	11	3	13	3		3	–		3	3	3
9	SettingsFragmentPresenter.java	5	3	31	19	7	9			–			
10	SliderViewHolder.java	5	4						3		–		
11	CheckBoxSettingViewHolder.java	5		5					3			–	
12	SingleChoiceViewHolder.java	5			4				3				–

Fig. 4. DSM of an Unhealthy Inheritance. Note: The file in red font is the base class with many subclasses, and they are all used by the client files in blue font.

3. Large-scale anti-pattern prevalence and severity study

In this section, we elaborate on the subjects of our study, the process and measures employed to investigate each of the research questions proposed in Section 1, as well as the results and answers to these questions. The replication package [45] with all the scripts and data are provided.[1]

3.1. Subjects

We first selected the most popular projects in GitHub based on the number of stars they received, and manually removed repeated projects, forming the initial projects list with 3111 projects in total: 752 in C/C++, 1884 in Java, and 475 in Python. We analyzed all of them using DV8 and posted the results on a demo website.[2] Since Depends,[3]

[1] https://zenodo.org/records/10472153.

[2] http://demo.archtimize.io/discover.

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

		1	2	3	4	5	6	7	8	9	10	11	12
1	authentication.cpp	–	17										
2	remote_actor.cpp	14	–									2	
3	test		5	–					5	5	5		
4	protobuf_broker.cpp		15		–				15	15	15		
5	simple_broker.cpp		15			–			15	15	15		
6	ping_pong.cpp			1	4	4	–					4	4
7	request_timeout.cpp				15	15	15	–				15	15
8	core-test.cpp			4					–				
9	drr_cached_queue.cpp		30						30	–	30		
10	drr_queue.cpp		10						10	10	–		
11	remote_actor.cpp		2		14	14	14	10				–	14
12	broker.cpp				9	9	9	5				9	–

Fig. 5. DSM of a Clique. Note: The dependency relations among these files form a strongly connected graph.

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	TextureInfo.h	–														
2	XFMemory.h		–													
3	VideoEvents.h			–												
4	TextureDecoder.h				–											
5	VideoConfig.h					–										
6	FBInfo.h				3		–									
7	GraphicsModActionData.h							–								
8	GraphicsModAction.h							7	–							
9	GraphicsModManager.h	1	5	1			11		33	–						
10	GraphicsModManager.cpp	1	2			14	10	6	17	14	–			7		
11	VertexManagerBase.cpp	5	40			34		2		2	1	–	3	14		
12	TextureCacheBase.cpp	103			54	111	5	4		5	4	9	–	40		
13	VideoConfig.cpp					52				2	1	2	1	–		
14	VideoBackendBase.cpp				1	20				3	2	3	5	13	–	
15	VertexShaderManager.cpp		117			31		2	2	3	2	2		15		–

Fig. 6. DSM of a Package Cycle. Note: The files in blue font belong to a different package from the rest of the files, but there are cyclical dependencies across packages.

the multi-language source code dependency extraction tool used by DV8, can only reliably process three types of programming languages, C/C++, Java, and Python, in this study, we only analyzed projects written in these languages. Since anti-patterns are only meaningful for non-trivial projects, and their impacts are long-term, in this study, we filtered out projects with 100 or fewer files and those with less than 1 year of revision history. Table 1 presents the basic information of the 1717 projects used as the subjects of this study, which include 487 C/C++ projects, 942 Java projects, and 288 Python projects. These projects are all non-trivial and long-lived, with at least 1954 LOC.

For each project, we first extract different types of structural dependencies from source files, such as `call` and `implement`, and export these dependency information into a JSON file that can be accepted by DV8 [5]. In order to detect the three types of anti-patterns that require co-change information, for each project, we also extracted its 1-year revision history and exported it into a log file as the second input of DV8. Using the JSON structural dependency file and the revision history log, DV8 calculates all the anti-patterns for each project on the backend and presents the results on the demo website. These results can be downloaded for further processing and analysis. Next, we will explain the measures, process, and answers to these research questions.

3.2. RQ1: General anti-pattern prevalence

To investigate the prevalence of these anti-patterns, we first counted the percentage of projects in which these anti-patterns are detected and the percentage of anti-pattern infected projects written in different programming languages, as presented in Table 2. This table reveals

Table 1
Subjects.

Language	# Projects	LOC range	Files range	Evolution period
C++	487	1954–2 289 195	101–584 216	1–37 years
Java	942	1694–2 816 629	101–50 318	1–25 years
Python	288	3679–1 062 453	101–7811	1–22 years

Table 2
Percentage of projects with anti-patterns.

Language	CLQ (%)	CRS (%)	MVG (%)	PKC (%)	UIH (%)	UIF (%)	Any (%)
All	82.00	39.00	83.54	84.79	91.62	34.26	99.55
C/C++	49.28	43.12	83.16	93.02	91.17	39.84	98.77
Java	92.46	21.66	73.67	94.06	98.09	19.64	99.68
Python	92.01	48.96	100	29.51	75.69	35.76	100

that within the 1717 projects analyzed, 99.55% of them contain some instances of anti-patterns.

Four out of the six anti-patterns are detected in more than 80% of the 1717 projects: Unhealthy Inheritance (UIH) (91.62%), Package Cycle (PKC) (84.79%), Modularity Violation (MVG) (83.54%), and Clique (CLQ) (82%). The results indicate that the violation of well-known design principles is very common. For example, although it has long been recognized that cyclical dependencies among program entities may increase maintenance costs, and a complex system should form a hierarchical structure to ease software extension and evolution [43,44], most software projects do have cycles, in the form of PKC at the package level, or CLQ at the file level. The prevalence of

MVG indicates that co-changes happen extremely often between files that do not refer to each other explicitly. The high prevalence of these anti-patterns implies that these well-known design principles are rarely strictly followed, which motivated our investigation of their severity and impact scope in RQ2 and RQ4, respectively.

The results do not differ much for different programming languages. Java projects have the least occurrence of Unstable Interface (19.64%) and Crossing (21.66%), while C/C++ projects have the least occurrence of Clique (49.28%). The other noticeable observation is that, for Python projects, Package Cycles is least common (29.51%), and the occurrence of Unhealthy Inheritance is also the least (75.69%) compared with C/C++ and Java. However, MVG instances are detected in ALL Python projects. One possibility is that the anti-pattern data published on the website are all calculated based on explicit dependencies only, that is, not including implicit dependencies caused by Python duck typing. This observation motivated our investigation of RQ3.

The other two anti-patterns are not as prevalent: Unstable Interface is detected in 34.26% of projects, and Crossing is found in 39% of projects. These two anti-patterns reflect that in these projects, there exist one or a few highly influential files, such as key abstractions or widely used utility files, but these influential files are error-prone or change-prone, making all their relative files unstable. It is interesting to note that these two anti-patterns are least common for Java projects, only about 20% of the 942 Java projects contain them, while prior work has reported that [16], for Java projects, Unstable Interface and Crossing have the most significant contributions to error-proneness and change-proneness. The implication is that these less prevalent anti-patterns are not necessarily less important, which motivated our investigation of RQ2.

Answer to RQ1: Of all the 6 types of anti-patterns, PKC, CLQ, UIF, and MVG are most prevalent, while UIF and CRS do not happen often. These results are consistent across programming languages, sizes, and project domains.

3.3. RQ2: General anti-pattern severity

To assess the severity of these anti-patterns, we consider the following three measures:

1. File percentage: the percentage of files involved in these anti-patterns (Table 8a, b, c, d);
2. Churn and churn percentage: maintenance costs measured by the total amount of churn (lines of Code (LOC) added or deleted to maintain source files), and the percentage of churn spent on files involved in anti-patterns (Table 9a, b, c, and d);
3. Churn per file: maintenance costs per file measured by the amount of churn spent per anti-pattern-involved file (Table 10 a, b, c, and d).

For each project, we collect the total amount of churn spent in the 1-year evolution period from its GitHub log as the basis and calculate these measures accordingly. For example, the Median rows and MVG columns in Tables 8a, 9a, and 10a present that, without differentiating programming languages, the median percentage of files influenced by MVG is 10.30% (Table 8a), but these files costs as much as 68.27% of the total churn (Table 9a). If we consider the costs per file, files involved in MVG spent 150.38 LOC to maintain (Table 10), ranked the third after UIF and CRS.

These data confirm that Modularity Violation is not only the most prevalent but costly: files involved in MVG cost the majority of churn. Unstable Interface (UIF) and Crossing (CRS) are also expensive anti-patterns: although their median file percentages are small (7.10% and 4.29% respectively), their median churn percentages are non-trivial (36.88% and 17.79% respectively). Considering the median value of Churn per File, files involved in these two anti-patterns are the most expensive: they can cost 80% more churn than MVG and about 10 times more than that of CLQ, PKC, and UIH. The implication is that although influential but unstable files do not happen often and do not necessarily influence many files, once they occur, they can be very expensive to maintain, becoming the focal point of technical debt. We consider the three anti-patterns, UIF, CRS, and MVG, to be severe.

For the other three prevalent anti-patterns, Clique is the least costly in terms of churn and file percentages, only consuming 7.98% of the total churn and impacting 6.31% of all the files (Tables 9a, 8a), while PKC's Churn per File median score is the least: files involved in PKC only consumes 16.72 churn LOC (Table 10a).

The implication is that the most common anti-patterns are not necessarily the most harmful, while less common anti-patterns could be the most expensive to maintain.

Now, we examine how this result differs for different programming languages. Tables 8b–10b, 8c–10c, and 8d–10d represent the measures for C/C++, Java, and Python programs respectively. These data reveal that, regardless of the programming language in use, MVG is the number one costly anti-pattern in terms of churn percentage, even though it does not necessarily influence most files. It is noticeable that ALL Python projects are infected with MVG, and the median score of churn percentage is 66%, much higher than that of C/C++ and Java projects. The implication is that developers have to spend most effort changing files that appear to be independent of each other, because their dependencies are invisible, implicit, but not absent. For both C/C++ and Python projects, Unstable Interface is the second most costly in terms of churn percentage. For Java projects, Unhealthy Inheritance (UIH) is the second in terms of median churn percentage (46.53%), almost as high as Modularity Violation (46.59%). For all three programming languages, Cliques incurred the least (C/C++ and Java) or second to the least (Python) churn percentage, meaning that although this anti-pattern is prevalent, its impact is limited.

The Churn per File data also reveal a consistent story: for all three languages, the median amount of churn spent on files with severe anti-patterns, UIF, CRS, and MVG, are in the order of hundreds of LOC: from 107 (MVG in Java) to 442 (UIF in Python). Among these three anti-patterns, files involved in UIF and CRS consume significantly more churn than MVG. By contrast, these numbers on prevalent anti-patterns are in the order of tens: from 9 (PKC in Java) to 61 (CLQ in Python). These anti-patterns are not mutually exclusive, however. We notice that many instances of Unstable Interface and Crossing also overlap with Unhealthy Inheritance and Clique instances. In particular, if an influential base class has many subclasses and changes together with them and their clients often, this base class will also be detected as an unstable interface. Similarly, the center file of a Crossing has both high fan-in and high fan-out, and their relatives are more likely to have cycles. The implication is that although the violations of well-known design principles are very common, most of them do not incur severe penalties as long as they are not also part of severe anti-patterns.

Answer to RQ2: Of all the 6 types of anti-patterns, MVG files consume the majority of maintenance churn; files infected with UIF and CRS are the most costly to maintain, and they can be the focal point of high maintenance. The majority of PKC, CLQ, and UIH instances, although very prevalent, are not as harmful. These results are consistent across programming languages, sizes, and project domains.

3.4. RQ3: The impact of dynamic typing

From Table 2, we can observe that compared with C/C++ and Java projects, much fewer Python projects contain Package Cycles (29.51%) and Unhealthy inheritance (75.69%). Except for Modularity Violation and Unstable Interface, the file percentage and churn percentage of

Table 3
Percentage of python projects with anti-patterns: Detected with possible dependencies.

Language	CLQ (%)	CRS (%)	MVG (%)	PKC (%)	UIH (%)	UIF (%)	Any (%)
Python (Explicit)	92.01	48.96	100	29.51	75.69	35.76	100.00
Python (Both)	**93.06**	**78.72**	**100**	**95.83**	**87.15**	**71.18**	**100.00**
Java	92.46	21.66	73.67	94.06	98.09	19.64	99.68

all other types of anti-patterns are exceptionally low. Is it really the case that Python projects are less infected by these other types of anti-patterns? Our hypothesis is that this is caused by the widely used duck typing in Python: the data type of a variable does not need to be declared at compile time. Instead, its type will be inferred at run-time based on how the variable is used so that a variable can be assigned to a different data type dynamically.

Jin et al. [25,26] first investigated the possibility of making these dynamic dependencies explicit at compile time. Since a variable can be bounded to more than one data type, they named such dependencies as *possible dependencies* and revealed that more than 90% of possible dependencies can be uniquely determined statically. They have also demonstrated that a Python file involved in possible dependencies consumes 32% more maintenance effort than a Python file without possible dependencies. The question is, *if a file with implicit dependencies can be more difficult to maintain than a file without, is it possible that a Python project, in general, could be more expensive to maintain than a C/C++ or Java project?* To answer this question, we compare (1) how the number of anti-patterns differs with and without possible dependencies, and (2) how maintenance efforts spent on Python projects, measured by Churn per File and Churn per LOC, differ from programs written in C/C++ and Java. Here we only consider the possible dependencies that can be uniquely determined.

3.4.1. Anti-pattern costs with and without possible dependency

To understand the impact of possible dependencies on anti-pattern detection, we make these possible dependencies caused by duck typing explicit and rerun DV8, and assess how the number of anti-pattern instances and their costs would change. For this purpose, we downloaded the source files and revision history of these 288 Python projects and re-generated the dependency information using Depends, which is configured to extract possible dependencies using type inference techniques. Using these new dependency files, we re-ran DV8 on these projects and repeated the prevalence and severity study.

Table 3 presents the percentage of Python projects infected by each type of anti-pattern using both explicit and possible dependencies. We also copied the data for Python projects analyzed using explicit dependencies only and that of Java projects as references. The table shows that the occurrence of Package Cycle increases from 29.51% to 95.83%, meaning that Python projects are even less likely to form a hierarchical structure than C/C++ and Java projects. All other types of anti-patterns increased as well. In particular, the occurrence of Crossing increased from 48.96% to 78.72%, and the Unhealthy Inheritance percentage is now 87.15%, comparable to that of C/C++ and Java.

Tables 8e and 9e respectively present the file percentage and churn percentage measures for Python projects with both explicit and possible dependencies. Except for that of Modularity Violation, file and churn percentages for the other five types of anti-patterns all increased (Table 8d vs. Table 8e, and Table 9d vs. Table 9e), which is expected because a large portion of previously implicit dependencies become explicit. As shown in Table 8(d) and 8(e): the percentage of anti-pattern files increased from 27.17% to 44.81%, indicating that a significant portion of anti-patterns were hidden by duck typing. In particular, the PKC file percentage increased from 3.7% to 26.12%, and the CRS and UIH file percentages almost doubled. The implication is that many syntactical dependency-based anti-patterns still exist but become implicit. Then the question is, compared with anti-patterns detected in C/C++ and Java, are these implicit anti-patterns in Python more expensive to maintain?

Table 4
Churn per File & Churn per LOC.

	Churn per File			Churn per LOC		
	C++	Java	Python	C++	Java	Python
MEAN	106.18	17.70	118.25	0.42	0.20	2.19
MIN	0.00	0.00	0.04	0.00	0.00	0.00
MAX	9598.27	220.49	2591.82	31.03	2.76	96.88
Q1	6.34	1.90	21.86	0.04	0.02	0.29
MEDIAN	**27.85**	**7.72**	**58.88**	**0.14**	**0.10**	**0.81**
Q3	79.48	22.39	130.51	0.37	0.27	1.71
IQR	73.13	20.49	108.65	0.33	0.24	1.42
SD	502.82	26.83	229.06	1.77	0.30	6.80

Comparing the Churn per File data in Table 10b (C/C++), Table 10c (Java), and Table 10e (Python), it becomes clear that in term of the mean and median Churn per File measures, Java files involved in anti-patterns consume much less churn than that of C/C++ and Python anti-pattern files, regardless of the anti-pattern types. The Churn per File costs for C/C++ and Python are comparable. We also notice that the median total churn LOC for Python MVG (19k) is much higher than that of C/C++ (11k), and Java (9k), which implies that implicit dependencies in Python, mostly in the form of duck typing, could be more expensive to maintain than that of C/C++ and Java. Next, we further explore the overall maintenance cost differences among projects written in these three programming languages.

3.4.2. Overall maintenance costs with and without possible dependency

To test the hypothesis that files involved in Python duck typing are more expensive to maintain, hence Python projects could be more costly to maintain than that of C/C++ and Java, we now calculate the maintenance costs of *all* the source files in a project, regardless of their involvement in anti-patterns. Specifically, we calculate the Churn per File and Churn per LOC using the 1-year evolution history of a project and compare how these measures differ for different programming languages. Since the amount of churn consumed for Python projects does not change regardless of how we detected anti-patterns, with or without possible dependencies, here we only use the data extracted using both explicit and possible dependencies.

Table 4 presents these data for all three languages. First of all, a Java file consumes much less churn, with a median value of 7.72 per file and 0.10 per LOC. Although the Churn per File data for anti-pattern files in C/C++ and Python are similar, when we consider *all* files, the median score of Python (58.88) doubles that of C/C++ (27.85). Similarly, Python has the largest median Churn per LOC, 0.81, which is about 6 times that of C/C++ (0.14) and 8 times that of Java (0.10).

The result is somewhat counter-intuitive in that Python is often considered easier to use. We would like to test the hypothesis that Python files are more expensive to maintain in a more rigorous way. To assess which models should be used to assess the differences among the three measures, we first apply the Shapiro–Wilk test[4] to determine that most of this data are not normally distributed. Accordingly, we apply the non-parametric 1-tailed Mann–Whitney U Test[5] to determine

[4] https://www.statskingdom.com/shapiro-wilk-test-calculator.html.
[5] https://www.statskingdom.com/170median_mann_whitney.html.

Table 5
Churn per File & Churn per LOC.

	Churn per File			Churn per LOC		
	C++	Java	Python	C++	Java	Python
C/C++	–	–	–	–	–	–
Java	2.13E−4	–	–	0	–	–
Python	0	0	–	4.44E−16	0	–

if the maintenance costs for the three programming languages are truly different from each other.

Table 5 presents the p-values of the pairwise Mann–Whitney tests for the Churn per File and Churn per LOC data. Each cell is a p-value of the test between the data set on the column and the one on the row. For example, $2.13E-04$ in row Java, column C/C++ in Table 5 means that the Cost per File in Java is significantly lower than that of C++ since the p-value is significantly lower than 0.05. Since all the p-values are significantly lower than 0.05, it confirms that the maintenance costs for source files in different programming languages are significantly different: Java files consume the least churn to maintain, followed by C/C++ files. Python files, however, contrary to our intuition, are most expensive to maintain.

Answer to RQ3: When we take into account possible dependencies, the prevalence of structural anti-patterns increases significantly; because of implicit dependencies, maintaining a Python file can be multiple times more costly than that of a C/C++ or Java file.

3.5. RQ4: The impact of non-trivial anti-patterns

Based on the observation that most prevalent anti-patterns may not be costly, we would like to further assess the severity of these prevalent anti-patterns based on their impact scopes: if an anti-pattern instance is trivial, e.g., only involving *five* or fewer files, it is highly likely that its existence is not very harmful. As we mentioned in Section 1, our choice of using *five* as the threshold to determine if an anti-pattern instance is *"trivial"* is based on Gobet and Clarkson's study on cognitive complexity [28], and Miller's [29] law of human cognition and information processing: people can easily process approximately 5 "chunks" of information at a time.

To assess the difference between trivial and non-trivial anti-patterns, for each project, we excluded these trivial instances and recalculated the percentages of projects in which these anti-patterns are detected, as well as their file percentages. Based on DV8's default thresholds of anti-patterns, there should not be any trivial Unstable Interface instances: for a group of files to be detected as a UIF, the leading file needs to influence at least 10% of all the files, which should always be more than 5 because all the projects we selected have more than 100 files. There should be few trivial cases for Crossing as well: the center file of a Crossing should have at least four dependents and depend on four other files. For PKC, since DV8 only detects pair-wise cyclical dependencies among packages, the total number of instances does not change either. Accordingly, in this section, we only consider three prevalent anti-patterns: UIH, CLQ, and MVG. We are particularly interested in UIH and CLQ because they violate most well-known design principles.

Table 6 presents the percentage of projects with all vs. non-trivial anti-patterns. Overall, Clique had the biggest difference when trivial instances were removed: the percentage of projects infected with Clique decreased 33%, from 82% to 54.71%. Projects with Modularity Violations decreased 19%, from 83.54% to 67.98%, and projects with Unhealthy Inheritance decreased 15%, from 91.62% to 77.66%. The implication is that although Cliques are prevalent, many of these instances are small, especially for those in C/C++ projects: the percentage

Table 6
Percentage of projects with All vs. Non-trivial anti-patterns.

Language	CLQ (%)	MVG (%)	UIH (%)
All (>5 files)	54.71	67.98	77.66
All	82.00	83.54	91.62
Reduction	**−33%**	**−19%**	**−15%**
C/C++ (>5 files)	18.69	66.94	80.08
C/C++	49.28	83.16	91.17
Reduction	**−62%**	**−20%**	**−12%**
Java (>5 files)	68.79	49.36	90.98
Java	92.46	73.67	98.09
Reduction	**−26%**	**−33%**	**−7%**
Python (>5 files)	68.40	99.00	67.36
Python	93.06	100.00	87.15
Reduction	**−26%**	**−1%**	**−23%**

Table 7
File percentage of anti-patterns: All vs. Non-trivial.

Language	CLQ (%)	MVG (%)	UIF (%)
All (>5 files)	6.09	20.86	16.28
All	8.5	25.13	22.55
Reduction	**−28%**	**−17%**	**−28%**
C/C++ (>5 files)	1.34	11.28	13.67
C/C++	2.35	14.63	18.82
Reduction	**−43%**	**−23%**	**−27%**
Java (>5 files)	9.79	5.83	24.60
Java	13.23	8.71	34.03
Reduction	**−26%**	**−33%**	**−28%**
Python (>5 files)	2.91	45.56	3.50
Python	4.19	51.95	4.72
Reduction	**−31%**	**−12%**	**−26%**

of C/C++ projects with Cliques decreases 62%, from 49.28% to 18.69% when trivial Cliques are excluded. Table 7 presents the file percentage differences with all vs. non-trivial anti-patterns, from which we also observed a significant reduction. In particular, the file percentage of CLQ and UIH both dropped 28%. The implication is that although Clique and Unhealthy Inheritance are very common, a large portion of projects only have trivial instances.

Answer to RQ4: From the three most prevalent anti-patterns, a large portion of CLQ and UIH instances are trivial, neither influential nor high-maintenance.

4. Result summary

Our study obtained several interesting results. The answer to RQ1 and RQ2 revealed that, of all the 1717 non-trivial and long-lived projects written in C/C++, Java, or Python, as many as 99.55% of them have at least one of the 6 types of anti-patterns. Among all these anti-patterns, Modularity Violation – frequent co-changes among seemingly unrelated files – turns out to be the most prevalent and costly, incurring, on average, 61.55% of all the LOC spent to modify a codebase. By contrast, less than 40% of the projects have Unstable Interface or Crossing—revealing influential but unstable files. However, once these anti-patterns occur, their maintenance cost could be most significant. We consider Modularity Violation (MVG), Unstable Interface (UIF) and Crossing (CRS) as three severe anti-patterns. The other three most prevalent anti-patterns, Package Cycle (PKC), Unhealthy Inheritance (UIH), and Clique (CLQ), are not as high-maintenance. These results are consistent regardless of the programming languages, sizes, and project domains.

The answer to RQ3 revealed that if we only consider explicit dependencies without dynamic typing, the presence of certain types of anti-patterns is significantly lower in Python projects: only 29.51% of

Python projects have Package Cycle, and the numbers for Java and C/C++ projects are 94.06% and 93.02% respectively. However, after we make possible dependencies explicit, the percentage of Package Cycle infected projects increased to 95.83%, indicating that using duck typing in Python does not reduce the prevalence of anti-patterns but just makes them invisible. As a matter of fact, considering the Churn per File and Churn per LOC spent on all the source files, the cost of maintaining Python files multiplies that of C/C++ and Java projects, and the majority of these costs stem from implicit dependencies. This result strengthens the observation that invisible dependencies are most high-maintenance, and Python, as a programming language known as being easier to use, may incur high maintenance costs in the long run.

Finally, the study reveals that after removing anti-patterns that influence 5 or fewer files, the prevalence of several anti-patterns reduced significantly. For example, the Clique infected projects reduced from 82% to 54.71%. The instances of Unhealthy Inheritance and Modularity Violations and their File Percentages were also noticeably reduced. The implication is that although cyclical dependencies are considered harmful in general, and the violations of well-known principles appear to be very common, file-level cycles are usually small and do not incur severe maintenance costs. However, if such a violation extends to a large number of files, a Clique or Unhealthy Inheritance could become a severe Unstable Interface or Crossing, which may cause severe technical debt.

5. Related work

Researchers have proposed many definitions and tools to detect problems in code, design, and architecture. Fowler [6]'s definition of "bad smell"—a heuristic for a number of design symptoms at the code level, such as god method, spaghetti code, code clones, and feature envy, are most widely studied. Researchers have proposed various methods to detect these code smells [46–49,49,50,50,51], and these tools tend to report a large number of code smells, making it hard to determine which ones are false positives, and often ignored by developers [52]. It has been recognized that many of these code-level problems are caused by higher-level problems in design or architecture [15]. In this study, we focus on design-level anti-patterns, that is, problematic relations among files.

Design and architecture smells are also widely studied [32,33,33–35]. Different researchers proposed different definitions of architectural smells. For example, Garcia et al. [33] proposed a categorization of architecture smells based on components and connectors; Sharma et al. [53] presented a tool named Designite[6] that detects architectural-level problems such as package cycles. Fontana et al. [4,54] also proposed a suite of architectural smells, such as HubLike structure that is similar to Crossing in DV8, supported by their tool named Arcan. There are other commercial tools available to support the detection of design or architecture smells, such as AiReviewer,[7] SonarGraph,[8] and Structure 101.[9] In this study, we used DV8 because in addition to structural problems such as package cycles, its definitions of Modularity Violation, Unstable Interface, and Crossing include evolution history information, more likely to identify real design problems leading to extra maintenance costs [39]. Moreover, DV8 is the only tool that defines Unhealthy Inheritance and reports maintenance costs in terms of churn and file percentages as part of the output. We will explore how the results may differ for other types of anti-patterns defined in other tools in the future.

There are also many empirical studies investigating the practical impact of these smells and anti-patterns. In particular, Palomba et al. [7,8] have studied the co-occurrence and impact of code smells in 395 releases of 30 software systems. Johannes et al. [55] performed a large-scale study of JavaScript code smells from 15 JavaScript applications. Researchers also studied the prevalence of code smells in specific contexts or domains, such as code smells in Android code [10], in SQL code [9], and in machine learning code [11]. However, there is no large-scale prevalence study for architecture and design-level smells or anti-patterns. Mo et al. [16] only studied 15 Java projects. Xiao el al.'s [17] recent study investigated 18 Java projects. Jin et al. [56] studied the impact of possible dependencies using 499 projects, but only those written in Python, and a large portion of them are small projects with just a few files. Different from these prior studies, we studied the prevalence of anti-patterns for Java, Python, and C/C++ from 1717 open-source projects that are long-lived and non-trivial. To study the severity of these anti-patterns, we also removed anti-patterns that impacted 5 or fewer files. To the best of our knowledge, this is the first large-scale, multiple-language prevalence and severity study of design anti-patterns.

6. Discussion

In this section, we discuss the implications, limitations, and future work of this research.

Implications. In the comprehensive analysis of myriad open-source projects, certain architectural anti-patterns have shown their pronounced presence, with distinct implications for software maintenance. Notably, Modularity Violation stands out as both prevalent and most costly to maintain: it is detected in 83.54% of all projects and inflicts the gravest toll on software projects, implying that invisible dependencies are most difficult to maintain. Corroborating previous research, the adverse effects of Crossing and Unstable Interface anti-patterns come to the fore. Such affirmation aligns with the existing body of knowledge that these two anti-patterns contribute most to error-proneness and change-proneness [16], which is not only true for Java but also for C/C++ and Python projects. It becomes evident, then, that the most resource-consuming anti-patterns, in terms of code modifications, are Modularity Violation, Unstable Interface, and Crossing. This gravitas is not merely in numbers; it translates to the tangible, escalated demands on time, effort, and resources—a palpable manifestation of accruing technical debt.

Delineating further into the specifics of the Python language and its characteristic feature of duck typing, an intriguing observation emerges. The overall prevalence of anti-pattern diminishes significantly when potential dependencies brought about by duck typing are factored out. This implies that a substantial fraction of anti-patterns in Python arises due to invisible dependencies instigated by duck typing. The LOC spent on maintaining a Python file can be many times more than that of a C/C++ or Java file, meaning that the widely applied duck typing only makes Python programs more expensive to maintain.

When the lens narrows down to anti-pattern instances influencing a minimum of five files, a drastic decline of Clique and Unhealthy Inheritance instances is observed, suggesting that a considerable portion of these prevalent anti-patterns has limited sprawl across files and may pose minimal immediate threats to software maintenance. The implication is that, well-known design principles underlying these anti-patterns are often violated, but are kept in a small scope. As long as these violations do not grow into more severe anti-patterns, their impacts are limited.

In essence, these findings are not just statistical revelations. They underscore the imperative for vigilant monitoring and timely intervention in the realm of software development. By recognizing and rectifying anti-patterns early on, the industry stands to enhance its efficiency and productivity, thereby ensuring robust, sustainable software ecosystems.

Limitations and Future Work. First of all, we only studied open-source projects written in three programming languages, and we cannot

[6] http://www.designite-tools.com/.
[7] http://www.aireviewer.com/.
[8] https://www.hello2morrow.com/products/sonargraph.
[9] https://structure101.com/.

claim that the results will be similar for other programming languages. In this study, we used DV8's default parameters to detect anti-patterns. For example, to detect Cliques, DV8 only considers *Call* and *Use* relations; to detect Modularity Violations, the minimal co-change frequency threshold is 2. If we use other types of relations or a different co-change frequency threshold, the results would be different. We plan to conduct sensitive analysis using various threshold settings to assess the prevalence and severity of these anti-patterns in the future.

Another limitation is that the accuracy of our results depends on the accuracy of the dependencies extracted from Depends. We have noticed that the dependency information extracted using different tools, such as Understand [57] or ENER [26], can be slightly different because different tools define and report dependency information differently. One of our future works is to compare and contrast the dependencies extracted from these tools and build a benchmark to unify the detection and representation of dependencies.

Duck typing is only one type of implicit dependencies. Other types of implicit dependencies exist for various reasons, such as the usage of third-party frameworks. Our understanding of these implicit dependencies and their consequences is still limited. We plan to investigate further and categorize these different types of implicit dependencies, their root causes, and their extraction methods so that designers can be made aware of their impact before they incur more severe consequences.

This research also reveals that instead of reporting all instances of all anti-patterns, technical debt detection tools should focus on reporting the most severe instances so that the designers can prioritize their efforts in fixing these design debts. Our next step is to study the relations among these anti-pattern instances and devise more effective refactoring recommendations.

In this paper, we use the amount of churn to approximate maintenance costs. As Shihab et al. [58] pointed out, using churn alone, similar to using LOC and complexity alone, may not be sufficient to predict the actual maintenance effort. Even if we combine the three dimensions, the correlation with real effort, measured in time, for example, can still be impacted by various factors. Given the scale and nature of this study, churn is the best objective approximation that we can leverage, similar to a number of prior publications that also used churn as effort approximation [59–62].

We are also aware of the fact that not all the churn spent on an anti-pattern-related file is caused by the anti-pattern. Multiple prior works have proved that the maintenance costs of files involved in anti-patterns are significantly higher than those without anti-patterns [16,40]. Here, our main objective is to compare the prevalence and severity of different types of anti-patterns, and we counted their total churn in the same way for all anti-pattern types so that this limitation would not affect the results.

7. Conclusion

The intricacies of software development often lead to the formation of suboptimal structures, commonly referred to as anti-patterns. These anti-patterns, while not immediately problematic, possess the potential to erode the quality of code and escalate maintenance overhead over time. Our study, the first of its magnitude and scope, conducted an in-depth analysis of anti-pattern prevalence and severity across a significant corpus of 1717 open-source projects in Java, C/C++, and Python. The revelations from this comprehensive study have significant implications for the software development community.

A whopping 99.55% of the projects under our scrutiny were afflicted by anti-patterns, emphasizing the universality of this issue across programming paradigms. While Modularity Violation was identified as one of the most prevalent anti-patterns, its consequent maintenance cost was alarmingly high, accounting for, on average, 61.55% of the entire LOC dedicated to project maintenance. However, Unstable Interface and Crossing, though less frequently occurring, manifested as a disproportionately resource-intensive anti-pattern, demanding almost 40% of all maintenance efforts.

Our exploration into the dynamics of the Python programming language, specifically in the context of its implicit dependencies arising from duck typing, unfolded another layer of complexity. Contrary to prevailing beliefs about dynamic typing enhancing maintainability, our findings suggest that implicit dependencies in Python do not simplify maintenance. Instead, duck typing amplifies the presence of anti-patterns and significantly increases maintenance difficulty: maintaining Python files could be many times more costly than maintaining C/C++ and Java files.

Moreover, our examination of trivial anti-patterns, those affecting 5 or fewer files, underscored the fact that design principle violations are extremely common and almost inevitable. However, a significant portion of the projects are able to control these violations in smaller scopes. It is possible that these violations are the results of the consideration of other tradeoffs. As long as such violations do not grow into more severe anti-patterns, their impacts are limited. This distinction is paramount as it helps developers differentiate between anti-patterns that are widespread but less harmful and those that, despite their rarity, can be detrimental to the maintenance life cycle.

In conclusion, this study sheds light on the silent yet pervasive existence of anti-patterns in software projects. It underscores the need for proactive detection and intervention, especially for the more detrimental anti-patterns like Modularity Violation, Unstable Interface, and Crossing. As the software development landscape continues to evolve, armed with insights from this research, developers, and organizations can adopt more informed strategies to tackle anti-patterns preemptively, optimizing the health, quality, and longevity of their software projects.

Data availability

The replication package with all the scripts and data are provided at [45].

Acknowledgements

This research was supported by the United States National Science Foundation grants 2236824, 2232720, and 2213764.

Appendix. Anti-pattern cost comparison of different languages

See Tables 8–10.

Table 8
Anti-patterns cost comparison: File percentage.

(a) File percentage of anti-patterns: All languages							
	CLQ (%)	CRS (%)	MVG (%)	PKC (%)	UIH (%)	UIF (%)	ALL AP (%)
MEAN	9.83%	6.42%	14.50%	29.88%	23.54%	10.04%	43.89%
MIN	0.03%	0.07%	0.00%	0.10%	0.09%	0.09%	0.00%
MAX	76.77%	53.68%	76.47%	98.60%	87.63%	63.97%	98.94%
Q1	2.50%	1.63%	3.33%	13.55%	7.30%	3.35%	28.05%
MEDIAN	**6.31%**	**4.29%**	**10.30%**	**27.78%**	**20.50%**	**7.10%**	**45.05%**
Q3	13.42%	8.78%	21.80%	42.22%	36.99%	13.10%	59.77%
IQR	10.92%	7.16%	18.47%	28.67%	29.69%	9.75%	31.72%
SD	10.61%	6.98%	13.83%	19.51%	17.81%	9.87%	21.02%
(b) File percentage of anti-patterns: C/C++							
	CLQ (%)	CRS (%)	MVG (%)	PKC (%)	UIH (%)	UIF (%)	ALL AP (%)
MEAN	2.35%	7.88%	14.63%	38.68%	18.82%	12.04%	48.87%
MIN	0.03%	0.22%	0.08%	0.76%	0.23%	0.60%	0.00%
MAX	22.43%	53.68%	72.79%	98.60%	70.83%	63.97%	98.60%
Q1	0.50%	2.73%	3.85%	21.61%	8.07%	4.39%	34.75%
MEDIAN	**1.31%**	**5.75%**	**11.12%**	**39.58%**	**17.64%**	**9.55%**	51.07%
Q3	2.67%	10.51%	21.39%	54.22%	27.53%	15.73%	64.14%
IQR	2.17%	7.78%	17.54%	32.61%	19.46%	11.35%	29.39%
SD	3.06%	7.60%	13.78%	21.03%	12.77%	10.88%	20.94%
(c) File percentage of anti-patterns: Java							
	CLQ (%)	CRS (%)	MVG (%)	PKC (%)	UIH (%)	UIF (%)	ALL AP (%)
MEAN	13.23%	5.49%	8.71%	27.93%	34.03%	7.69%	50.90%
MIN	0.27%	0.07%	0.02%	0.15%	0.09%	0.09%	0.00%
MAX	76.77%	39.22%	67.40%	92.80%	87.63%	50.83%	98.94%
Q1	4.27%	1.02%	1.78%	13.75%	21.92%	1.89%	39.24%
MEDIAN	**9.29%**	**2.93%**	**4.86%**	**25.84%**	**35.29%**	**4.57%**	**52.92%**
Q3	18.30%	6.90%	12.28%	38.50%	45.91%	10.28%	64.70%
IQR	14.03%	5.88%	10.50%	24.76%	23.99%	8.39%	25.46%
SD	12.37%	7.23%	10.06%	17.15%	16.35%	9.12%	18.98%
(d) File percentage of anti-patterns: Python - Explicit dependencies only							
	CLQ (%)	CRS (%)	MVG (%)	PKC (%)	UIH (%)	UIF (%)	ALL AP (%)
MEAN	6.25%	3.72%	22.27%	8.68%	4.32%	9.01%	28.02%
MIN	0.21%	0.26%	0.00%	0.10%	0.24%	0.79%	1.14%
MAX	27.07%	15.79%	76.47%	76.89%	32.57%	31.95%	85.61%
Q1	2.37%	0.75%	10.68%	1.74%	1.39%	4.08%	16.44%
MEDIAN	**4.65%**	**2.41%**	**19.95%**	**3.70%**	**2.65%**	**6.84%**	**27.17%**
Q3	9.28%	4.66%	31.68%	9.81%	5.44%	8.85%	37.37%
IQR	6.91%	3.91%	21.00%	8.07%	4.05%	4.78%	20.93%
SD	5.15%	3.97%	14.65%	12.48%	4.71%	8.05%	15.04%
(e) File percentage of anti-patterns: Python - with possible dependencies							
	CLQ (%)	CRS (%)	MVG (%)	PKC (%)	UIH (%)	UIF (%)	ANY AP (%)
MEAN	8.98%	6.64%	21.35%	28.16%	9.96%	10.92%	45.41%
MIN	0.21%	0.32%	0%	0.41%	0.24%	0.48%	2.29%
MAX	45.10%	27.63%	76.14%	87.58%	60.45%	35.02%	90.85%
Q1	3.26%	2.19%	9.77%	12.51%	3.23%	4.74%	33.65%
MEDIAN	**6.63%**	**4.71%**	**19.05%**	**26.12%**	**7.18%**	**8.06%**	**44.81%**
Q3	13.46%	9.46%	30.67%	39.64%	14.17%	16.55%	58.15%
IQR	10.21%	7.41%	20.91%	27.13%	10.94%	11.81%	24.49%
SD	7.58%	5.72%	14.49%	18.71%	9.03%	8.54%	18.15%

Note: IQR = Inter-quartile range. SD = Standard Deviation.

Table 9
Anti-patterns cost comparison: Churn and churn percentage.

(a) Churn and churn percentage of anti-patterns: All languages

	CLQ (%)	CRS (%)	MVG (%)	PKC (%)	UIH (%)	UIF (%)
MEAN	6K (15.46)	20K (24.22)	42K (61.55)	12K (33.04)	11K (31.98)	31K (39.31)
MIN	0 (0.00)	94 (0.05)	0 (0.00)	0 (0.00)	0 (0.00)	73 (0.11)
MAX	1045K (100.00)	362K (95.49)	2130K (100.00)	703K (100.00)	688K (100.00)	893K (98.40)
Q1	61.75 (1.04)	3K (7.17)	1K (39.60)	226 (7.56)	189 (3.76)	5K (17.78)
MEDIAN	**673.5 (7.98)**	**8K (17.79)**	**8K (68.27)**	**2K (26.06)**	**2K (25.89)**	**12K (36.88)**
Q3	4K (22.34)	22K (37.11)	31K (86.02)	10K (52.60)	8K (55.68)	32K (58.10)
IQR	4K (21.30)	19K (29.94)	30K (46.43)	10K (45.04)	7K (51.92)	26K (40.32)
SD	30K (19.93)	36K (20.79)	147K (28.28)	35K (28.84)	33K (28.86)	64K (24.87)

(b) Churn and churn percentage of anti-patterns: C/C++

	CLQ (%)	CRS (%)	MVG (%)	PKC (%)	UIH (%)	UIF (%)
MEAN	9K (6.94)	27K (35.68)	37K (57.95)	23K (45.36)	18K (32.11)	38K (47.40)
MIN	0 (0.00)	255 (0.43)	19 (0.05)	0 (0.00)	0 (0.00)	73 (0.26)
MAX	1045K (99.93)	362K (95.49)	700K (98.64)	703K (100.00)	688K (100.00)	388K (96.93)
Q1	0 (0.00)	4K (16.89)	3K (34.93)	568 (12.36)	339.75 (5.34)	8K (28.21)
MEDIAN	**197 (1.07)**	**11K (33.72)**	**11K (63.93)**	**5K (45.85)**	**3K (27.80)**	**19K (45.77)**
Q3	2K (6.35)	32K (53.05)	35K (80.17)	20K (75.54)	14K (52.87)	45K (65.86)
IQR	2K (6.35)	28K (36.16)	32K (45.24)	20K (63.18)	13K (47.53)	38K (37.65)
SD	69K (13.81)	46K (23.10)	77K (27.78)	56K (32.62)	53K (27.24)	56K (24.21)

(c) Churn and churn percentage of anti-patterns: Java

	CLQ (%)	CRS (%)	MVG (%)	PKC (%)	UIH (%)	UIF (%)
MEAN	5K (22.80)	13K (27.93)	14K (46.33)	6K (33.69)	9K (44.66)	14K (34.03)
MIN	0 (0.00)	94 (0.55)	11 (0.14)	0 (0.00)	0 (0.00)	155 (0.88)
MAX	215K (100.00)	178K (82.01)	335K (100.00)	206K (100.00)	351K (100.00)	232K (87.11)
Q1	50.5 (3.11)	2K (11.29)	661.5 (25.40)	116.75 (10.86)	180.5 (21.98)	3K (13.80)
MEDIAN	**538 (16.94)**	**6K (23.33)**	**3K (46.59)**	**1K (29.12)**	**1K (46.53)**	**7K (29.04)**
Q3	3K (34.47)	16K (40.35)	12K (67.07)	5K (50.16)	7K (66.64)	16K (51.59)
IQR	3K (31.36)	13K (29.07)	11K (41.67)	5K (39.30)	7K (44.67)	13K (37.79)
SD	13K (23.26)	21K (20.54)	33K (25.48)	16K (27.24)	25K (27.79)	23K (23.86)

(d) Churn and churn percentage of anti-patterns: Python - Explicit dependencies only

	CLQ (%)	CRS (%)	MVG (%)	PKC (%)	UIH (%)	UIF (%)
MEAN	5K (9.28)	11K (10.72)	82K (62.12)	4K (8.67)	3K (6.09)	23K (28.24)
MIN	0K (0.00)	1K (0.32)	0K (0.00)	0K (0.00)	0K (0.00)	1K (0.84)
MAX	143K (73.03)	85K (49.01)	2130K (99.27)	39K (72.20)	50K (56.12)	118K (73.71)
Q1	0K (1.18)	3K (2.55)	4K (45.07)	0K (0.36)	0K (0.31)	10K (13.89)
MEDIAN	**1K (4.79)**	**5K (9.37)**	**19K (66.03)**	**1K (2.84)**	**1K (3.04)**	**18K (25.22)**
Q3	5K (12.46)	16K (14.89)	58K (83.04)	5K (11.22)	3K (7.98)	28K (43.09)
IQR	5K (11.28)	14K (12.34)	54K (37.97)	4K (10.86)	3K (7.67)	18K (29.20)
SD	13K (11.94)	14K (10.55)	239K (24.53)	7K (13.61)	6K (9.19)	23K (19.40)

(e) Churn and churn percentage of anti-patterns: Python - with possible dependencies

	CLQ (%)	CRS (%)	MVG (%)	PKC (%)	UIH (%)	UIF (%)
MEAN	8K (12.05)	21K (20.76)	81K (59.54)	16K (28.02)	9K (13.35)	53K (33.49)
MIN	0K (0.00)	0K (0.35)	0K (0.00)	0K (0.00)	0K (0.00)	1K (1.24)
MAX	233K (79.95)	361K (81.58)	2130K (99.26)	465K (94.32)	250K (69.05)	893K (88.58)
Q1	0K (1.46)	4K (6.75)	4K (41.55)	1K (7.18)	0K (0.86)	9K (16.13)
MEDIAN	**2K (6.94)**	**9K (15.72)**	**19K (61.82)**	**6K (23.99)**	**2K (8.04)**	**19K (29.77)**
Q3	7K (17.61)	22K (29.70)	57K (80.15)	15K (45.42)	7K (19.91)	46K (52.28)
IQR	7K (16.15)	18K (22.95)	53K (38.60)	14K (38.25)	7K (19.05)	38K (36.15)
SD	19K (14.15)	40K (17.94)	239K (25.03)	37K (22.92)	25K (15.12)	120K (22.81)

Note: IQR = Inter-quartile range. SD = Standard Deviation.

Table 10
Anti-patterns cost comparison: Churn Per File.

(a) Churn per file of anti-patterns: All languages

	CLQ	CRS	MVG	PKC	UIH	UIF
MEAN	189	408	260	58	78	409
MIN	0	13	4	0	0	3
MAX	149K	23K	12K	9K	16K	12K
Q1	3	177	83	3	3	180
MEDIAN	**23**	**276**	**150**	**17**	**21**	**276**
Q3	86	443	248	57	77	437
IQR	83	266	165	54	74	257
SD	3.7K	990	643	256	402	796

(b) Churn per file of anti-patterns: C/C++

	CLQ	CRS	MVG	PKC	UIH	UIF
MEAN	2K	655	397	123	187	594
MIN	0	32	10	0	0	3
MAX	149K	23K	12K	9K	16K	12K
Q1	0	200	120	5	7	221
MEDIAN	**21**	**349**	**206**	**28**	**46**	**358**
Q3	159	542	338	91	129	529
IQR	159	342	218	86	122	308
SD	13K	2K	1K	640	1K	1K

(c) Churn per file of anti-patterns: Java

	CLQ	CRS	MVG	PKC	UIH	UIF
MEAN	41	244	133	24	28	221
MIN	0	13	4	0	0	9
MAX	1K	921	2K	499	2K	745
Q1	2	154	64	1	2	138
MEDIAN	**12**	**211**	**107**	**9**	**10**	**196**
Q3	43	306	164	29	32	289
IQR	41	152	100	27	30	152
SD	97	143	134	44	72	125

(d) Churn per file of anti-patterns: Python - Explicit dependencies only

	CLQ	CRS	MVG	PKC	UIH	UIF
MEAN	117	506	340	82	118	476
MIN	0	131	4	0	0	55
MAX	2K	2K	7K	684	1K	1K
Q1	15	228	118	7	8	231
MEDIAN	**61**	**330**	**179**	**44**	**56**	**442**
Q3	154	628	297	130	163	637
IQR	139	400	180	123	154	406
SD	174	444	676	107	157	311

(e) Cost Per File of Anti-patterns: Python - with possible dependencies

	CLQ	CRS	MVG	PKC	UIH	UIF
MEAN	121	384	348	84	100	485
MIN	0	44	4	0	0	47
MAX	2K	2K	8K	1K	870	5K
Q1	14	221	118	15	11	229
MEDIAN	**65**	**297**	**184**	**54**	**58**	**320**
Q3	144	477	315	109	131	491
IQR	130	256	197	94	120	262
SD	240	278	692	117	133	648

Note: IQR = Inter-quartile range. SD = Standard Deviation.

References

[1] 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.

[2] S. SonarSource, Sonarqube, 2013, Capturado em: http://www.sonarqube.org.

[3] Designite. [Online]. Available: https://www.designite-tools.com/.

[4] 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.

[5] Y. Cai, R. Kazman, DV8: Automated architecture analysis tool suites, in: 2019 IEEE/ACM International Conference on Technical Debt (TechDebt), 2019, pp. 53–54, http://dx.doi.org/10.1109/TechDebt.2019.00015.

[6] M. Fowler, Refactoring: Improving the Design of Existing Code, Addison-Wesley, 1999.

[7] 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, http://dx.doi.org/10.1016/j.infsof.2018.02.004, [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584918300211.

[8] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, A. De Lucia, On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation, in: Proceedings of the 40th International Conference on Software Engineering, 2018, pp. 482–482.

[9] 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.

[10] 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) http://dx.doi.org/10.1007/s10664-019-09768-9.

[11] B. van Oort, L. Cruz, M. Aniche, A. van Deursen, The prevalence of code smells in machine learning projects, 2021, arXiv:2103.04146.

[12] A. Yamashita, M. Zanoni, F.A. Fontana, B. Walter, Inter-smell relations in industrial and open source systems: A replication and comparative analysis, in: 2015 IEEE International Conference on Software Maintenance and Evolution, ICSME, 2015, pp. 121–130.

[13] 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.

[14] 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.

[15] L. Xiao, Y. Cai, R. Kazman, Design rule spaces: A new form of architecture insight, in: Proc. 36th International Conference on Software Engineering, 2014.

[16] 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).

[17] L. Xiao, Y. Cai, R. Kazman, R. Mo, Q. Feng, Detecting the locations and predicting the maintenance costs of compound architectural debts, IEEE Trans. Softw. Eng. 48 (9) (2022) 3686–3715, http://dx.doi.org/10.1109/TSE.2021.3102221.

[18] F.A. Fontana, I. Pigazzini, R. Roveda, D.A. Tamburri, M. Zanoni, E. Di 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.

[19] 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.

[20] 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.

[21] 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, pp. 171–180, http://dx.doi.org/10.1109/ICSE-SEIP.2019.00026.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] W. Jin, D. Zhong, Y. Cai, R. Kazman, T. Liu, Evaluating the impact of possible dependencies on architecture-level maintainability, IEEE Trans. Softw. Eng. 49 (3) (2023) 1064–1085, http://dx.doi.org/10.1109/TSE.2022.3171288.

[27] Multi-language code dependency analysis tool. [Online]. Available: https://github.com/multilang-depends/depends.

[28] F. Gobet, G. Clarkson, Chunks in expert memory: Evidence for the magical number four...or is it two? Memory (Hove, England) 12 (2004) 732–747, http://dx.doi.org/10.1080/09658210344000530.

[29] G.A. Miller, The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information, Vol. 63, 1956, pp. 81–97, http://dx.doi.org/10.1037/h0043158.

[30] Structure101. [Online]. Available: https://structure101.com/.

[31] 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.

[32] M. Lippert, S. Roock, Refactoring in Large Software Projects: Performing Complex Restructurings Successfull, Wiley, 2006.

[33] 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.

[34] 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.

[35] 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.

[36] C.Y. Baldwin, K.B. Clark, Design Rules, Vol. 1: The Power of Modularity, MIT Press, 2000.

[37] 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.

[38] 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.

[39] 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, http://dx.doi.org/10.1109/ICSE-SEIP52600.2021.00021.

[40] 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.

[41] 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.

[42] 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.

[43] D.L. Parnas, Designing software for ease of extension and contraction, IEEE Trans. Softw. Eng. 5 (2) (1979) 128–138.

[44] H.A. Simon, The architecture of complexity, in: Facets of Systems Science, Springer US, Boston, MA, 1991, pp. 457–476, http://dx.doi.org/10.1007/978-1-4899-0718-9_31.

[45] A. Liu, J. Lefever, Y. Han, Y. Cai, Prevalence and Severity of Design Anti-Patterns in Open Source Programs—A Large-Scale Study, 2024, http://dx.doi.org/10.5281/zenodo.10472153, [Online]. Available: https://zenodo.org/records/10472153.

[46] 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.

[47] 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.

[48] 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.

[49] 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.

[50] N. Tsantalis, A. Chatzigeorgiou, Identification of move method refactoring opportunities, IEEE Trans. Softw. Eng. 35 (3) (2009) 347–367.
[51] 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.
[52] 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.
[53] 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.
[54] F.A. Fontana, I. Pigazzini, R. Roveda, D. Tamburri, M. Zanoni, E. Di Nitto, Arcan: A tool for architectural smells detection, in: 2017 IEEE International Conference on Software Architecture Workshops, ICSAW, 2017, pp. 282–285.
[55] D. Johannes, F. Khomh, G. Antoniol, A large-scale empirical study of code smells in JavaScript projects, Softw. Qual. J. 27 (2019) 1271–1314.
[56] W. Jin, D. Zhong, Y. Cai, R. Kazman, T. Liu, Evaluating the impact of possible dependencies on architecture-level maintainability, IEEE Trans. Softw. Eng. 49 (3) (2023) 1064–1085, http://dx.doi.org/10.1109/TSE.2022.3171288.
[57] Understand. [Online]. Available: https://scitools.com/.
[58] E. Shihab, Y. Kamei, B. Adams, A.E. Hassan, Is lines of code a good measure of effort in effort-aware models? Inf. Softw. Technol. 55 (11) (2013) 1981–1993, http://dx.doi.org/10.1016/j.infsof.2013.06.002, [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584913001316.
[59] T. Mende, R. Koschke, Revisiting the evaluation of defect prediction models, in: Proceedings of the 5th International Conference on Predictor Models in Software Engineering, PROMISE '09, Association for Computing Machinery, New York, NY, USA, 2009, http://dx.doi.org/10.1145/1540438.1540448.
[60] E. Arisholm, L.C. Briand, E.B. Johannessen, A systematic and comprehensive investigation of methods to build and evaluate fault prediction models, J. Syst. Softw. 83 (1) (2010) 2–17, http://dx.doi.org/10.1016/j.jss.2009.06.055.
[61] S. Karus, M. Dumas, Predicting coding effort in projects containing xml, in: 2012 16th European Conference on Software Maintenance and Reengineering, IEEE, 2012, pp. 203–212.
[62] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, A.B. Bener, Defect prediction from static code features: current results, limitations, new approaches, Autom. Softw. Eng. 17 (2010) 375–407, [Online]. Available: https://api.semanticscholar.org/CorpusID:2782280.