The Design Smells Breaking the Boundary between Android Variants and AOSP

Wuxia Jin
Xi'an Jiaotong University
Xi'an, China
jinwuxia@mail.xjtu.edu.cn

Jiaowei Shang
Xi'an Jiaotong University
Xi'an, China
jwshang@stu.xjtu.edu.cn

Jianguo Zheng
Xi'an Jiaotong University
Xi'an, China
Zz_Nut@stu.xjtu.edu.cn

Mengjie Sun
Xi'an Jiaotong University
Xi'an, China
jamnsun@stu.xjtu.edu.cn

Zhenyu Huang
Honor Device Co., Ltd.
Xi'an, China
huangzhenyu1@hihonor.com

Ming Fan
Xi'an Jiaotong University
Xi'an, China
mingfan@mail.xjtu.edu.cn

Ting Liu
Xi'an Jiaotong University
Xi'an, China
tingliu@mail.xjtu.edu.cn

***Abstract*—Phone vendors customize their Android variants to enhance system functionalities based on the Android Open Source Project (AOSP). While independent development, Android variants have to periodically evolve with the upstream AOSP and merge code changes from AOSP. Vendors have invested great effort to maintain their variants and resolve merging conflicts. In this paper, we characterize the design smells with recurring patterns that break the design boundary between Android variants and AOSP. These smells are manifested as problematic dependencies across the boundary, hindering Android variants' maintainability and co-evolution with AOSP. We propose the DroidDS for automatically detecting design smells. We collect 22 Android variant versions and 22 corresponding AOSP versions, involving 4 open-source projects and 1 industrial project. Our results demonstrate that: files involved in design smells consume higher maintenance costs than other files; these infected files are not merely the files with large code size, increased complexity, and object-oriented smells; the infected files have been involved in more than half of code conflicts induced by re-applying AOSP's changes to Android variants; a substantial portion of design issues could be *mitigable*. Practitioners can utilize our DroidDS to pinpoint and prioritize design problems for Android variants. Refactoring these problems will help keep a healthy coupling between diverse variants and AOSP, potentially improving maintainability and reducing conflict risks.**

***Index Terms*—design smell, software maintenance, Android**

I. INTRODUCTION

In the Android ecosystem, mobile device vendors develop and maintain their Android variants based on Google's open-source mobile operating system (i.e., Android Open Source Project, AOSP [1]). For example, Honor MagicOS, Xiaomi MiUI, and Samsung One UI are built upon AOSP to enhance and customize features [2, 3]. While independent development, Android variants need to periodically keep in sync with the AOSP version, merging code changes from AOSP updates. Since Google releases new versions of AOSP frequently, Android variant updates have been routine activities. As revealed by recent studies [4, 5], half of the merge commits incur conflicts averaged on their investigated variants. It is a time-consuming and manually intensive task to re-apply AOSP changes to variants [6]. Android vendors have invested great efforts in resolving conflicts and fixing incompatible code to maintain their versions.

To facilitate the maintainability and evolvability of an Android variant, its customized source code should unidirectionally depend on AOSP code via stable APIs, while keeping *a clear design boundary* between them [5]. However, diverse customization and limited update periods present challenges for Android variants to uphold this clear boundary. Design smells [7, 8] are the symptoms of problematic designs with software evolution, typically involving particular patterns of undesired dependencies among code elements. A plethora of work [9–11] has identified design smells in individual systems; however, the design smells in the Android ecosystem are distinct, particularly in breaking the boundary between AOSP maintained by Google and Android variants maintained by vendors. Despite being managed separately, an Android variant must be updated periodically with AOSP changes. In this ecosystem, design smells can disrupt the boundary between the AOSP and downstream variants, impeding the co-evolution of Android variants with AOSP and leading to code conflicts.

In this paper, we formally define the design smells and propose a detection approach namely DroidDS for Android variants. First, in terms of the code element composition (termed *ownership* [5]) of an Android variant, we design an Ownership-attributed Dependency Graph (ODG) to characterize the Android variant architecture co-evolving with AOSP. This architecture can be regarded as *a structure of two high-level modules that are typically managed separately*, i.e., the upstream module managed by AOSP and the downstream module managed by an Android vendor. Next, we summarize 3 design smells with 13 recurring patterns that erode the boundary between the upstream and downstream modules, including Inline Dependency (ID), Unstable Dependency (UD), and Cyclic Dependency (CD). They violate fundamental design principles [12], blurring, enlarging, and twining the module boundary. Third, we propose the DroidDS to detect these violations. DroidDS creates an ODG from the code facts of the Android variant repository and then it searches smell

1558-1225/25/$31.00 ©2025 IEEE
DOI 10.1109/ICSE55347.2025.00062

subgraphs from the ODG after graph pruning.

To evaluate design smells, we investigate four research questions.

RQ1: Do entities involved in design smells consume more maintenance costs than other entities in Android variants?

RQ2: To what extent do the entities involved in design smells capture conflicts due to Android variants merging with AOSP changes?

RQ3: To what extent can design smell instances be mitigable that require minor code changes for fixing?

RQ4: Are entities involved in Android design smells not merely the files with larger code size, increased complexity, and object-oriented smells?

We collected 22 Android variant versions and 22 corresponding AOSP versions for study, analyzing 197 million lines of code and 27 million history commits. Our results show that design smells are strongly associated with high maintenance costs, i.e., the files involved in design smells are 9.42 times more change-prone and bug-prone than other files in Android variants. We also observed that the design smells can capture 68.87% merge conflict files and 72.26% conflict blocks. Moreover, our results indicate that 52.37% design problems can be *mitigable*. Our collaborators responsible for the industrial project confirmed our results and planned to address ID and UD issues with priority.

In summary, our work makes the following contributions:

- Our work is the first to formally propose and characterize the design smells that break the boundary between Android variants and AOSP.
- We implement the DroidDS to detect these design smells. Practitioners can utilize our DroidDS to monitor the boundary between their Android variants and AOSP.
- Our empirical study reveals and evaluates the impact of design smells on maintainability and conflicts due to merging with AOSP updates into Android variants.
- We contribute our datasets [13], the largest size of the Android OS dataset as far as we know.

II. Methodology

A. Software Architecture of Android Variants

According to the code composition of an Android variant, we regard the Android variant architecture co-evolving with AOSP as a structure of two high-level *separately managed* modules: the *upstream module* reusing the code from AOSP and the *downstream module* extended by this variant, while the *downstream module* has to periodically keep in sync with upstream changes. To formalize this architecture, we define an *Ownership-attributed Dependency Graph (ODG)* extracted from code facts. Each node denotes an entity that has an attribute named *ownership* to distinguish whether its source code is newly developed by the variant or reuses the AOSP.

Ownership-attributed Dependency Graph (ODG). An ODG is a graph structure extracted from code facts of an Android variant, i.e., $G =< V, D, O >$. V is a set of entities and $V = \{e_i\}$. An entity e_i can be a kind of *variable*, *method*, or *class*. $e_i.t$ labels the entity kinds. D is a set of dependencies between entities. A dependency $d = e_i \rightarrow e_j$ denotes the relation from e_i to e_j like *include*, *call*, *inherit* and *implement*. O is a set of *ownership attributes* that an entity carries.

We classify three kinds of *ownership attributes* for code entities, including *actively native*, *extensive*, and *intrusively native*. Concretely, the code of a variant is typically composed of three parts [5]: 1) the code cloned from the AOSP version (i.e., *actively native*); the code that is newly developed by the variant version (i.e., *extensive*); the code that is originally from AOSP while further intrusively modified by the variant (i.e., *intrusively native*). That is, $O = \{\text{“}actively\ \ native\text{”}, \text{“}intrusively\ \ native\text{”}, \text{“}extensive\text{”}\}$, and $e_i.o \in O$.

Figure 1 illustrates a code snippet in the AOSP version (Android-11.0.0_r46) and the corresponding variant version (LineageOS-18.1). LineageOS-18.1 extends from Android-11.0.0_r46. In LineageOS-18.1, the ownership attribute of `setRebootProgress` is *“actively native”* because its source code is completely the same as that of `setRebootProgress` defined in the AOSP version; the ownership attribute of `showShutDownDialog` is *“intrusively native”* since the AOSP declares this method while the variant modifies it; the ownership attribute of `rebootCustom` is *“extensive”* since the variant version newly defines this method.

File : services/core/java/com/android/server/power/ShutdownThread.java

Line	AOSP version : Android-11.0.0_r46	Android variant version : LineageOS-18.1
1	private void setRebootProgress(final int progress, final CharSequence message) {	private void setRebootProgress(final int progress, final CharSequence message) {
2	mHandler.post(new Runnable() {	mHandler.post(new Runnable() {
3	…	… “actively native”
4	}	}
5	private static ProgressDialog showShutdownDialog (Context context) {	private static ProgressDialog showShutDownDialog (Context context) {
6	if(showSysuiReboot()){	+ boolean mRebootCustom = false;
7	return null;	- if (showSysuiReboot()) {
8	}	+ if (mRebootCustom && showSysuiReboot()) {
9	}	return null;
10		} “intrusively native”
11		}
12		+ public static void rebootCustom(final Context context, String reason, boolean confirm) {
13		+ mReboot = true;
14		… “extensive”
15		+ }

Fig. 1: An example of entity ownership attributes

Software Architecture based on ODG. We define the software architecture of an Android variant co-evolving with AOSP as a structure of two high-level *separately managed* modules, including an *upstream* module (M_{up}) and a *downstream* module (M_{down}). M_{up} is a subgraph of ODG that only contains (*actively/intrusively*) *native* entities as nodes and their relationships as edges; M_{down} is a subgraph of ODG that only contains *extensive* entities along with their dependencies; the dependencies between *native* entities and *extensive* entities link the two modules. In the Android ecosystem, the source code of M_{up} and M_{down} are independently developed by the Google AOSP and Android vendors in separate codebases. As a result, once a new version of M_{up} is released by AOSP and can be accessible, M_{down} has to upgrade and merge with new changes from M_{up}. It is inevitable to incur code conflicts during code merging. Android vendors invest substantial effort to maintain their customized variants and resolve conflicts [6, 14, 15].

B. Design Smells in Android Variants

For a well-designed architecture of an Android variant, M_{down} should depend on M_{up} through stable APIs and keep a clear design boundary between them [5] (Figure 2(a)). A clear boundary will facilitate the maintainability of customized Android code and the mitigation of conflict risks. However, the special software architecture (i.e., it is a structure of two modules managed by independent organizations) and customization requirements make it difficult for an Android variant to persist in a clear design boundary. Prior empirical studies [4–6] reported a tight coupling between Android variants and the AOSP. However, it is still unclear about the design smells severely eroding the boundary between M_{up} and M_{down} and hindering Android variant maintainability.

We adhere to the well-recognized definition of design smell—"a structure in a design that indicates a violation of fundamental design principles, and which can negatively impact the project's quality." [9, 16–18] A design smell is typically embodied as a group of undesired dependencies among code elements [8]. Lots of work studied design smells while rarely considering software domains. It is necessary to characterize and detect the potentially problematic dependencies for design smells that hinder the co-evolution of Android variants with AOSP. After scrutinizing large-scale Android OS and engaging in discussions with Android architects from a leading mobile vendor which has maintained its Android variants for more than 10 years, we summarized 3 kinds of design smells with 13 recurring patterns. The architects have confirmed that these adverse patterns are major concerns to be addressed. Table I describes them.

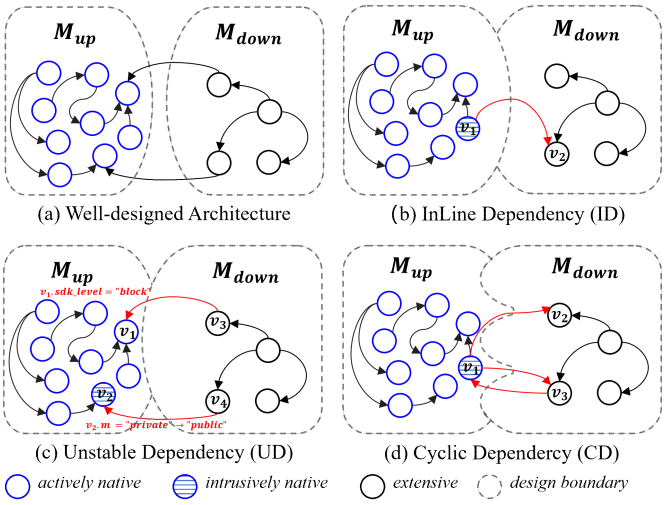


Fig. 2: Design smells for Android variants

We referenced existing work [8] for general definition frameworks, upon which we introduced the design smells for Android in terms of *rationale*, *description*, *adverse impact*, *dependency patterns*, and *formalization*. For each design smell, *rationale* illustrates its general definition as well as the design principles it violates, which is typically software domain-agnostic; *description* explains the module-level manifestation of this design smell specifically for Android variants; *adverse impact* shows how this smell erodes boundary between the two separately-managed modules, leading to heavy maintenance cost and code conflicts; *dependency pattern* illustrates kinds of recurring patterns for this smell; *formalization* expresses these patterns based on the ODG.

1. Inline Dependency (ID).

Rationale: This design smell refers to that the elements, supposed to be in one module, are embedded into another module. It violates the Single Responsibility Principle [16].

Description: In Android variants, this smell means that the code implementations conducted by an Android variant are inlined to the code scope originally defined by AOSP. That is, the entities inside M_{up} are intrusively modified by M_{down} and embed code from M_{down}.

Adverse Impact: the intrusive modifications to M_{up} would *blur the design boundary* between an Android variant and its upstream AOSP code, bringing in *textual* or *syntactic* conflicts [6, 19] during Android update. In Figure 2(b), M_{down} modifies v_1 inside M_{up} to inline v_2 that was developed by M_{down}. If v_1 is updated by the AOSP new version while the embedded v_2 is updated by the Android variant, a merge commit will incur inside the code block of v_1.

Dependency Patterns: As listed in Table I, we summarized 4 recurring patterns (i.e., *ID1–ID4*) for this smell. *ID1* describes that an *intrusive native* class in M_{up} embeds the field or method members that are newly developed by M_{down}.

ID2, *ID3*, and *ID4* take into account different entity kinds involved in inline dependencies.

Formalization:

$$
\begin{aligned}
&\texttt{Pre}: \exists\ e_i \rightarrow e_j \in D \wedge e_i.o = \text{“}intrusively\ native\text{”} \wedge e_j.o = \text{“}extensive\text{”} \\
&\texttt{T} = \{\text{“}inner\ class\text{”}, \text{“}anonymous\ class\text{”}, \text{“}anonymous\ method\text{”}\} \\
&ID1: \texttt{Pre} \wedge e_i.t = \text{“}class\text{”} \wedge e_j.t \in \{\text{“}field\text{”}, \text{“}method\text{”}\} \\
&ID2: \texttt{Pre} \wedge e_i.t = \text{“}method\text{”} \wedge e_j.t = \text{“}parameter\text{”} \\
&ID3: \texttt{Pre} \wedge e_i.t \in \texttt{T} \\
&ID4: \texttt{Pre} \wedge e_i.t = \text{“}method\text{”} \wedge e_j.t \in \{\text{“}class\text{”}, \text{“}method\text{”}, \text{“}variable\text{”}\}
\end{aligned}
\tag{1}
$$

2. Unstable Dependency (UD).

Rationale: As defined by existing work [20, 21], this design smell refers to an element that depends on other elements that are less stable than itself. The affected element might cause a ripple effect of changes in a software system. This smell violates the Stable Dependency Principle [16].

Description: In Android variants, this smell indicates that entities in M_{down} depend on less stable entities provided by the upstream M_{up}. Unstable entities can be either claimed by Android non-SDK restrictions [22] or non-public APIs indicated by modifiers.

Adverse Impact: The dependencies in this smell would *enlarge the design boundary* between M_{down} and M_{up}, excessively exposing hidden entities of M_{up} to M_{down}. The AOSP would often update unstable entities in M_{up}, leading to M_{down} having to change with them frequently and hence introducing *syntactic conflict* risks. Figure 2 (c) presents two unstable dependencies. $v_3 \rightarrow v_1$, where v_1 is a unstable non-SDK API ($v_1.sdk\_level$ = "*block*") encapsulated in M_{up} while $v_3 \in M_{down}$ accesses it. Another dependency is $v_4 \rightarrow v_2$,

TABLE I: Dependency patterns for design smells in Android variants

Design Smell	Dependency Patterns in Android Variants
Inline Dependency (ID)	ID1: An *intrusively native* class embeds *extensive* class fields or method members.
	ID2: An *intrusively native* method embeds *extensive* parameters.
	ID3: An *intrusively native* entity embeds *extensive* inner classes or *extensive* anonymous entities.
	ID4: An *intrusively native* method embeds *extensive* local code blocks.
Unstable Dependency (UD)	UD1: An *extensive* entity depends on *native* non-SDK APIs;
	UD2: An *extensive* entity depends on *native* non-SDK API of which API level is promoted through intrusive modifications.
	UD3: An *extensive* entity depends on *native* entities of which accessibility level is promoted through intrusive modifications.
	UD4: An *extensive* entity inherits or rewrites final *native* entities through intrusive modifications.
	UD5: An *extensive* entity dynamically depends on *native* entities through reflection mechanism.
Cyclic Dependency (CD)	CD1: An *intrusively native* entity depends on *extensive* base classes.
	CD2: An *intrusively native* entity depends on *extensive* interface entities.
	CD3: An *intrusively native* entity depends on *extensive* object entities.
	CD4: There exists an cyclic shape between *intrusively native* entities and *extensive* entities through transitive dependencies.

where v_2 is a *private* API originally defined in M_{up} while is further modified by M_{down} into *public* for access.

Dependency Patterns: There are three manners to indicate whether a code entity of AOSP can be stably and safely depended upon by Android variants: (1) the SDK level proposed by Google non-SDK restriction policy, i.e., `sdk`,`unsupported`,`max-target-'X'` , and `blocked` [22]; (2) the accessibility level, i.e., `public`, `protected`, `private` and default; and (3) the modifier keyword like `final`. The order of stability is "`sdk` > `unsupported` > `max-target-'X'` > `block`" and "`public` > `protected` > default > `private`" for an entity. The `final` classes cannot be inherited by other classes, and `final` methods cannot be rewritten by others. The code developed by Android variants sometimes makes intrusive modifications to promote the API accessibility initially claimed by AOSP for extensive reuse. For example, downstream Android code might change the `private` APIs of AOSP into `public` and subsequently depend on them. As listed in Table I, the *UD1*, *UD2* and *UD5* break the design contracts indicated by non-SDK claims, *UD3* violates the design contracts implied by the accessibility level, and *UD4* breaks the constraints imposed by the modifier keywords.

Formalization:

$$
\begin{aligned}
&\texttt{Pre1}: \exists\, e_i \rightarrow e_j \in D \wedge e_i.o = \text{“}extensive\text{”} \wedge e_j.o = \text{“}actively\ native\text{”}\\
&\texttt{Pre2}: \exists\, e_i \rightarrow e_j \in D \wedge e_i.o = \text{“}extensive\text{”} \wedge e_j.o = \text{“}intrusive\ native\text{”}\\
&\texttt{Pre3}: T = \{\text{“}class\text{”}, \text{“}method\text{”}\} \wedge e_i.t \in T \wedge e_j.t \in T \wedge e_j.sdk\_level\\
&\qquad \in \{\text{“}blocked\text{”}, \text{“}unsupported\text{”}, \text{“}max-target-'X'\text{”}\}\\
&\texttt{Pre4}: T = \{\text{“}class\text{”}, \text{“}method\text{”}, \text{“}field\text{”}\} \wedge e_i.t \in T \wedge e_j.t \in T\\
&UD1: \texttt{Pre1} \wedge \texttt{Pre3}\\
&UD2: \texttt{Pre2} \wedge \texttt{Pre3} \wedge e_j.sdk\_level > map(e_j).sdk\_level\\
&UD3: \texttt{Pre2} \wedge \texttt{Pre4} \wedge e_j.m > map(e_j).m\\
&UD4: \texttt{Pre2} \wedge d.t \in \{\text{“}inherit\text{”}, \text{“}override\text{”}\}\\
&\qquad \wedge e_j.m \neq \text{“}final\text{”} \wedge map(e_j).m = \text{“}final\text{”}\\
&UD5: \texttt{Pre2} \wedge \texttt{Pre4} \wedge d.t = \text{“}reflect\text{”}
\end{aligned}
\tag{2}
$$

3. Cyclic Dependency (CD).

Rationale: CD describes that elements directly or indirectly depend on each other to function correctly, as defined in prior work [23–25]. The chain of dependencies among elements breaks the desirable acyclic nature of a sub-system's dependency structure. Code elements involved in a cycle can be hard to maintain in isolation.

Description: In Android variants, this smell indicates that M_{down} first modifies the *native* entities in M_{up} and then makes these entities depend on *extensive* code in M_{down}. Such dependencies from M_{up} to M_{down} lead to a dependency cycle at the design level between M_{up} and M_{down}.

Adverse Impact: If any entities in M_{down} were changed, the upstream entities in M_{up} have to be adapted accordingly, and vice versa. The dependencies in this smell would *twine the design boundary* between Android variants and AOSP. Consequently, the change to either party (Android variants or AOSP) would cause *semantic* conflicts manifested as build or test failures[19, 26] when merging changes from AOSP. Figure 2(d) shows that both $\{v_1 \rightarrow v_3, v_3 \rightarrow v_1\}$ and $\{v_1 \rightarrow v_2, v_3 \rightarrow v_1\}$ result in dependency cycles between M_{up} and M_{down}.

Dependency Patterns: As listed in Table I, *CD1–CD3* presents four major reasons for the unhealthy reverse dependencies from M_{up} to M_{down} in terms of dependency kinds and entity kinds.

CD4 additionally considers indirect dependencies that are calculated by transitive closure [27].

Formalization:

$$
\begin{aligned}
&\texttt{Pre1}: \exists\, e_i \rightarrow e_j \in D \wedge e_i.o = \text{“}intrusive\ native\text{”} \wedge e_j.o = \text{“}extensive\text{”}\\
&\texttt{Pre2}: \exists\, (e_i \rightarrow e_j \in D \wedge e_j \rightarrow e_i \in D) \wedge e_i.o = \text{“}intrusive\ native\text{”}\\
&\qquad \wedge e_j.o = \text{“}extensive\text{”}\\
&\texttt{Pre3}: T = \{\text{“}class\text{”}, \text{“}method\text{”}, \text{“}field\text{”}\} \wedge e_i.t \in T \wedge e_j.t \in T\\
&CD1: \texttt{Pre1} \wedge d.t = \text{“}inherit\text{”}\\
&CD2: \texttt{Pre1} \wedge d.t = \text{“}implement\text{”}\\
&CD3: \texttt{Pre1} \wedge \texttt{Pre3} \wedge d.t \notin \{\text{“}inherit\text{”}, \text{“}implement\text{”}\}\\
&CD4: \texttt{Pre2} \wedge \texttt{Pre3}
\end{aligned}
\tag{3}
$$

C. Detection Approach

We implement an approach named DroidDS to reveal design smells. Figure 3 illustrates the approach overview. The input includes the source code (Src) and commit records (Cmt) of an Android variant, and the source code (Src_u) and commit history (Cmt_u) of the corresponding AOSP version that the Android variant is developed based on. DroidDS first creates an ODG for an Android variant, and then it detects design smells from the ODG after pruning. The output is a set of design smell instances labeled Res.

1) Ownership-Attributed Dependency Graph Creation: This module extracts the ODG for an Android variant.

Dependency Graph Extraction. First, we employ a static code analysis tool named *ENRE* [28] to extract a dependency graph (i.e., G) from Src and a dependency graph (i.e., G_u)

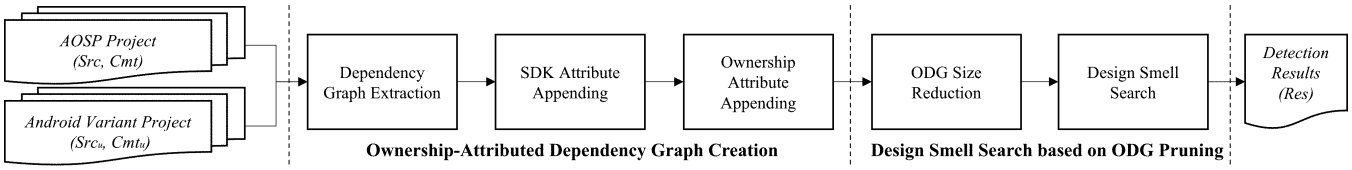


Fig. 3: The overview of DroidDS

from Src_u, respectively for the Android variant and its corresponding AOSP version. $G =< V, D >$, $G_u =< V_u, D_u >$. Each node in $V \cup V_u$ denotes an entity e; each edge (i.e., $d = e_i \rightarrow e_j$) in $D \cup D_u$ denotes one dependency from e_i to e_j. e carries several attributes: $e.t$ denotes the entity kind like `method` and `class`; $e.qualifiedName$ denotes its qualified name; $e.m$ records a set of modifiers that decorate this entity, such as `public`, `private`, and `final`; $e.loc$ denotes the line number where this entity is defined in source code. For one dependency $d = e_i \rightarrow e_j$, $d.t$ denotes the dependency kind such as `call`, `inherit`, and `contain`.

SDK Attribute Appending. As proposed by Google, each release of Android (either AOSP or variants) should claim the non-SDK API lists since Android 9 [22]. The non-SDK API lists show the restriction levels of API entities. As introduced in Section II-B, `sdk` means an API entity can be publicly accessed; `blocked` indicates that an entity should not be accessed; `unsupported` indicates that an entity can be exploited by developers while would be changed anytime; and `max-target-'X'` means that an entity can be accessed in Android versions not later than the "X" version [22]. We adopt the process in prior work [5] to export non-SDK information and append them to the attributes of entities in G and G_u. $e.sdk\_level \in \{$"$blocked$", "sdk", "$unsupported$", "$max-target-'X'$"$\}$.

Ownership Attribute Appending. We employ the method proposed by Jin et al. [5] to identify the ownership attribute for entities in G, producing $G' =< V, D, O >$ from $G =< V, D >$ of an Android variant. For each $e \in V$, this step leverages *git blame* [29] to track the commit sets cmt_e from Cmt. If cmt_e has intersections with Cmt_u and also has intersections with $Cmt \setminus Cmt_u$, we set the ownership attribute of e as $e.o =$*"intrusively native"*; if cmt_e only has intersections with Cmt_u, $e.o =$*"actively native"*; otherwise, $e.o =$*"extensive"*. After processing each $e \in V$ for G, it finally generates the ODG for an Android variant, i.e., $G' =< V, D, O >$. Please note that we use *git blame* solely to serve the ownership attribute identification.

2) Design Smell Search based on ODG Pruning: Using the ODG (G') of an Android variant, we transform the smell detection as a sub-graph search on G'.

ODG Size Reduction. We prune G' to reduce the search scope of smell graph detection. In terms of the ownership attributes (i.e., $e.o$), we obtain the upstream module namely M_{up} and downstream module namely M_{down} for G' using the formalization in Section II-A. Since we focus on design smells across the boundary between M_{up} and M_{down} along with associated dependencies, we employ two prune strategies to reduce G' as follows.

The first strategy prunes the dependencies between *native* entities except for `define` dependencies. The basic *define* dependencies are retained since they often act as candidate edges to form a design smell sub-graph. The second strategy filters out the dependencies between *extensive* entities. That is, if one dependency only connects *extensive* entities and there is no *native* entity connecting the two entities, this dependency will be removed. After traversing each edge $d \in D$ in G', it produces a smaller-sized G'^s.

Design Smell Search. We use the dependency patterns (denoted as $\{G\_smell_i\}$) of design smells formalized in Section II-B to search sub-graphs from G'^s. Besides entity kinds and dependency kinds, the design smells also concern the attribute changes of *native* entities like modifier ($e.m$) and non-sdk restriction level ($e.sdk\_level$). As a result, this step first builds a set of entity pairs $\{(e, e_u)\}$ (where $e.o =$*"intrusively native"*), indicating that $e \in V$ in G'^s is derived from $e_u \in V_u$ in G_u via code modifications made by the Android variant into AOSP. Next, it runs a depth-first search algorithm to retrieve isomorphic instances of G_smell_i from G'^s. Finally, we store detected smell instances into Res.

III. Research Questions and Subject Collection

A. Research Questions

We study four research questions to evaluate the impact of the design smells on the maintenance of Android variants.

RQ1: *Do entities involved in design smells consume more maintenance costs than other entities in Android variants?* We will compare the change-proneness and bug-proneness of the file entities involved in design smells with those of other files. The positive answer will imply the impact of design smells on the maintainability of Android variants.

RQ2: *To what extent do the entities involved in design smells capture conflicts due to Android variants merging with AOSP changes?* One major maintenance activity is to handle conflicts due to AOSP updates [5, 6]. We will explore the correlation between design smells and conflicts, and evaluate the precision and recall using design smells to capture conflicts.

RQ3: *To what extent can design smells be mitigable that require minor code changes for fixing?* Based on industrial collaborators' experience, some design smell instances would be *mitigable* if their *descriptive* dependencies diverge from *prescriptive* ones. The *prescriptive* dependencies indicate the intended dependencies supposed to be introduced by a smell; the *descriptive* dependencies indicate the defacto dependencies that have been associated with this smell.

TABLE II: The collected project versions for study

Project	Downstream Version					Upstream (AOSP) Version				Merge Commits	
	Version	Version Abbr.	#File	#LoC	#Commit	Version	#File	#LoC	#Commit	#MerCmt	#ConfCmt
AOSPA[30]	Topaz	A-T	30,551	5,267,551	795,029	android13.0.0_r32	30,312	5,218,902	782,268	10	6
	Sapphire	A-S	27,539	4,575,525	674,953	android12-gsi	27,308	4,526,922	661,807	46	37
	Ruby	A-R	26,497	4,165,240	501,248	android11-qpr1-d-release	26,429	4,147,331	650,983	64	47
	Quartz	A-Q	19,939	3,188,444	420,479	android11-d2-release	19,913	3,175,862	498,028	99	52
CalyxOS[31]	android13	C-13	30,549	5,226,916	784,306	android-13.0.0_r35	30,312	5,218,909	782,288	8	3
	android12	C-12	27,384	4,530,132	651,640	android-12.0.0_r29	27,308	4,526,923	649,754	5	2
	android11	C-11	26,699	4,169,261	482,229	android-11.0.0_r46	26,463	4,155,454	497,949	10	1
LineageOS[32]	LineageOS-20.0	L-20.0	30,544	5,232,233	784,381	android-13.0.0_r35	30,312	5,218,909	782,288	8	4
	LineageOS-19.1	L-19.1	27,685	4,576,171	663,065	android-12.1.0_r7	27,469	4,558,550	659,793	3	1
	LineageOS-18.1	L-18.1	26,713	4,190,090	499,427	android-11.0.0_r46	22,534	3,669,603	505,590	17	8
	LineageOS-17.1	L-17.1	22,951	3,711,094	426,543	android-10.0.0_r41	22,455	3,652,993	571,185	28	12
	LineageOS-16.0	L-16.0	2,0293	3,212,780	371,425	android-9.0.0_r61	19,871	3,171,811	369,065	50	12
OmniROM[33]	android-13.0	O-13.0	30,357	5,223,536	784,163	android-13.0.0_r35	30,312	5,218,909	782,288	5	4
	android-12.0	O-12.0	27,351	4,530,005	650,684	android-12.0.0_r28	27,308	4,526,922	650,983	4	2
	android-11	O-11	25,866	4,147,440	502,111	android-11.0.0_r38	25,796	4,131,630	687,952	178	106
	android-10	O-10	22,744	3,696,397	426,463	android-10.0.0_r41	22,558	3,677,983	650,983	10	3
	android-9	O-9	19,930	3,186,526	371,214	android-9.0.0_r34	19,824	3,173,299	423,947	43	3
IndustrialX	E	I-E	38,108	6,277,516	818,119	android-13.0.0_r1	29,007	4,810,366	762,679	4	4
	D	I-D	38,118	6,267,146	816,998	android-13.0.0_r1	29,007	4,810,366	762,679	4	4
	C	I-C	37,974	6,235,618	815,815	android-13.0.0_r1	29,007	4,810,366	762,679	4	4
	B	I-B	35,546	5,866,790	661,243	android-12.0.0_r10	27,161	4,511,991	648,567	2	2
	A	I-A	31,181	4,909,996	491,418	android-11.0.0_r35	26,454	4,152,622	418,833	14	14
Summary	22 versions		624,519	102,386,407	13,392,953	22 versions	577,120	95,066,623	13,962,588	616	331

RQ4: *Are entities involved in Android design smells not merely the files with larger code size, increased complexity, and object-oriented smells?* Code size, complexity, and object-oriented (OO) smells impact change- and fault-proneness [34–37]. We will evaluate whether the identified design smells are not dominated by large files (measured by size and complexity) and OO smells. The answer will imply that our smells are new factors correlated with Android maintenance costs.

B. Subject Collection

Our study subjects include open-source Android variants and industrial projects for diversity. Concretely, we followed existing work [4, 5] to collect open-source Android variant projects. Our selection criteria consider three aspects. First, a project has been actively maintained and managed by the Version Control System like Git [29]. Second, a project has been evolving with the Android framework of AOSP to merge changes from AOSP. Third, the version information of a project has been well-recorded through git branches or tags. We collected an industrial project anonymously named IndustrialX from a famous Android device vendor. For the past three years, the vendor has continuously taken the top 3 ranking in domestic Android market share. In summary, we collected 5 Android variant projects, including 4 open-source ones and 1 industrial project. For each project, we selected the latest versions. We also obtained the AOSP version corresponding to each variant based on the version information.

Finally, we collected 22 Android variant versions and 22 upstream AOSP versions accordingly for investigation. Similar to existing work [4, 5], our study also focused on the Java code of the Android *framework base*. The major reason is that Android variants usually customize the framework base for extension [6]. Table II lists collection results. The *Summary* row shows that our work analyzed 197 million lines of Android code and 27 million commits.

IV. Evaluation

A. Maintenance Cost Verification (RQ1)

We investigate whether the files involved in design smells are more change-prone and bug-prone in Android variants.

1) Study Setup: Prior work [8, 38, 39] employed the maintenance history managed by a version control system to verify design smells. They assumed that if the files in design smells are error-prone and change-prone as recorded in the revision history, the connections of these files would be problematic and significantly impact architecture maintenance. Similarly, we also investigate whether the files infected by design smells are more error-prone and change-prone in Android variants. The positive results will imply the adverse effects of design smells on Android maintenance.

2) Measures: Following existing work [8, 38, 39], we calculated six measures including $\#issue$, $\#author$, $\#changeCmt$, $\#changeLoc$, $\#issueCmt$, and $\#issueLoc$ to evaluate the change-proneness and error-proneness of a file. $\#issue$ counts the issues that a file participated in the commit history; $\#author$ counts the developers who contributed to the revision of a file; $\#changeCmt$ and $\#changeLoc$ denote the number of commit records and the lines of code where a file was modified; $\#issueCmt$ and $\#issueLoc$ considers the commits and lines of code related to issue fix commits.

TABLE III: A summary of the detected design smells

Project	#Ins	#a_File%	#b_File%	Project	#Ins	#a_File%	#b_File%
A-T	3,073	2.76%	4.84%	L-16.0	1,196	2.80%	5.40%
A-S	10,141	10.82%	15.31%	O-13.0	305	0.75%	1.43%
A-R	2,287	2.79%	3.41%	O-12.0	307	0.75%	1.43%
A-Q	1,061	2.33%	3.99%	O-11	2,564	3.49%	5.70%
C-13	444	0.84%	1.49%	O-10	1,231	2.34%	4.98%
C-12	260	0.56%	1.32%	O-9	959	2.19%	4.64%
C-11	366	0.75%	1.78%	I-E	36,278	14.02%	46.34%
L-20.0	399	0.81%	1.57%	I-D	38,215	13.78%	51.81%
L-19.1	1,216	1.94%	3.48%	I-C	36,777	14.75%	46.43%
L-18.1	1,388	2.30%	4.20%	I-B	36,931	15.21%	46.19%
L-17.1	1,384	2.56%	5.14%	I-A	28,327	21.59%	47.60%
Summary 22 versions	Summary #Ins 205,111		Average #a_File% 5.46%			Average #b_File% 14.02%	

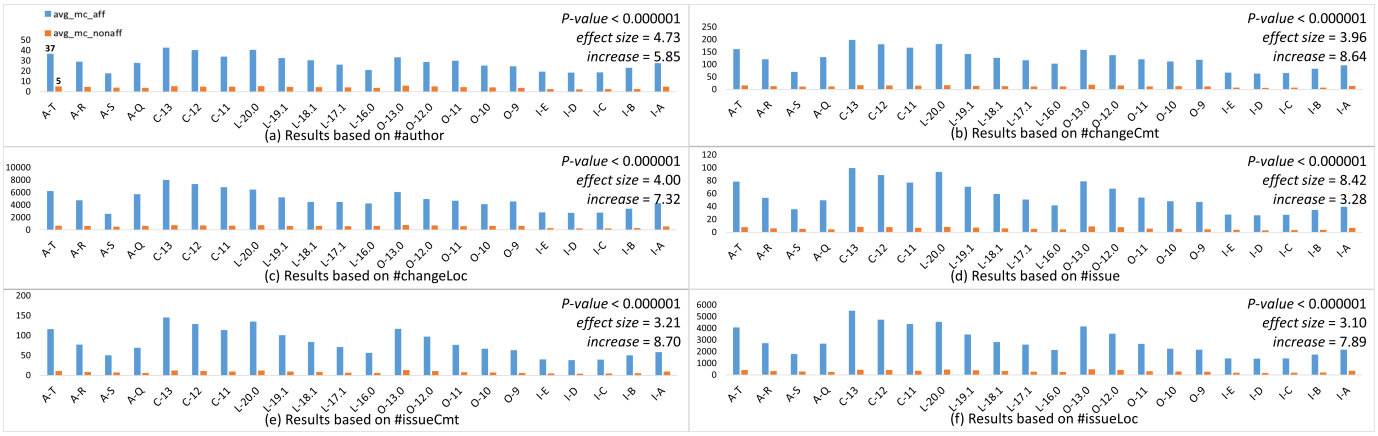


Fig. 4: The results based on all commit history in RQ1

These six measures are calculated based on commit history. We considered two approaches to determine the scope of the commit history. Firstly, similar to previous research [8, 27, 40], we took into account all changes and issues in the revision history to calculate maintenance costs. This is because large-scale projects typically undergo long-term evolution and maintenance. Secondly, following the findings of research [41, 42], we assessed the impact of design smells on maintenance costs after their introduction. For a specific software version where design smells are identified, we selected the commit history after this version to calculate maintenance costs.

For each investigated subject, we computed the average maintenance cost (i.e., *avg_mc_aff*) of the files involved in design smells and the average values (i.e., *avg_mc_nonaff*) of other files in terms of each measure. To rigorously study RQ1, we employed the Wilcoxon Sign-Rank tests [43] to test whether the values of *avg_mc_aff* are significantly higher than those of *avg_mc_nonaff* in all subjects. We defined the *Null Hypothesis (H_0)– there is no difference between the population of avg_mc_aff and the population of avg_mc_nonaff.* This hypothesis test reports *P-value* and *effect size*. If the evidence rejects H_0 hypothesis, we further calculated the relative increase of *avg_mc_aff* compared to *avg_mc_nonaff*, i.e., $increase = \frac{avg\_mc\_aff - avg\_mc\_nonaff}{avg\_mc\_nonaff}$.

3) Results: Table III shows the design smells detected from all project versions. *#Ins* counts the design smell instances, *#a_File%* counts the percentage of the distinct files involved in design smells to all files in a project, and *#b_File%* denotes their percentage to the files that link the upstream module and downstream module in a project. We can observe that the design smells indeed occurred in all Android variants we studied. Since the detected design smells concern the problematic dependencies across an Android variant's upstream and downstream modules, the file percentage calculated by *#b_File%* is always larger than *#a_File%*. On average, the files participating in design smells account for 14.02% of the files across the module boundary.

Figure 4 visualizes the measurement results when using all changes and issues in revision history to calculate maintenance costs. Regarding each maintenance cost measure, blue-rendered bars and orange-rendered bars in Figure 4 correspond to the average value for files infected by design smells and that for non-infected files, i.e., *avg_mc_aff* and *avg_mc_nonaff*. Considering the project A-T (i.e., AOSPA Topaz) in Figure 4(a), the two bars show that the files involved in design smells required 6.4 times (i.e., *increase*=(37-5)/5) more contributors in maintenance. Each sub-figure is labeled with three values, i.e., *P-value*, *effect size* and the average of *increase*. *P-value* and *effect size* are reported for the Wilcoxon Sign-Rank tests based on each group of (*avg_mc_aff, avg_mc_nonaff*). The labeled *increase* computes the average of *increase* on all projects. All P-values are smaller than 0.000001, meaning that the maintenance measures for infected files are significantly greater than those for other files. All *effect size* values are larger than 0.8 [44], demonstrating that the difference between the maintenance costs of files affected by design smells and those of other files is significant. Moreover, the average of *increase* is 5.85, 8.64, 7.32, 8.42, 8.70, and 7.89 in terms of six maintenance measures, respectively. In general, all sub-figures present consistent results.

TABLE IV: The results based on the commit history after the version where design smells are detected in RQ1

	#author	*#changeCmt*	*#changeLoc*	*#issue*	*#issueCmt*	*#issueLoc*
P-value	< 0.0002	< 0.0002	< 0.0002	< 0.0002	< 0.0002	< 0.0002
effect size	2	2.37	2.69	2.36	2.41	2.53
increase	6.45	10.56	15.27	9.99	10.48	13.45

Table IV summarizes the measurement results, which uses the commit history after the version where design smells are detected to calculate maintenance costs. We also made Wilcoxon Sign-Rank tests and reported P-values and *effect size*. Table IV indicates that the maintenance costs of infected files are significantly greater than those of other files. The average of *increase* is 6.45, 10.56, 15.27, 9.99, 10.48, and 13.45 in terms of six maintenance measures. The observation is consistent with that of Figure 4.

RQ1 Summary. The experiments demonstrated that the files involved in design smells consume 9.42 times higher maintenance costs than non-infected files, averaged on all Android variants. It suggests that design smells are strongly correlated with heavy maintenance costs.

B. Merge Conflict Analysis (RQ2)

We study the relation between the design smells and conflicts that are caused by re-applying AOSP's changes to Android variants.

1) Study Setup: We followed existing work [4, 5] to collect code conflicts due to Android variants merging the updates from AOSP. First, we employed the *git log* to identify merge commits from the revision history of an Android variant. Next, we performed a series of git commands to simulate these merge commits locally. It will produce conflict logs if a conflict occurs. The last two columns of Table II list the merge commits we collected. *#MerCmt* counts the merge commits, and *#ConfCmt* counts the conflict commits. We then evaluated the extent to which the files in design smells are correlated with merge conflicts during Android updates. If most conflicting files and blocks are prone to be involved in design smells, we assume that design smells strongly hinder Android maintenance activities.

We further identified recurring conflict blocks. First, we computed the similarity of each pair of conflict blocks based on the Longest Common Subsequence [45]. This algorithm has been commonly adopted by *diff* utility for textual merging and conflict report [19]. We calculated the similarity using $Sim = R/N$, where R counts the text length of a conflict block, and N denotes the text length of the matched text. If the Sim is larger than the threshold σ, the two conflict blocks would be considered recurring ones.

We conducted a group of experiments under σ=[10%, 20%, 30%, 40%, 50%, 60%, 70%, 80%, 90%, 100%]. We observed that the detection results produced by $\sigma = 50\%$ are closest to manual results. Therefore, we used this configuration.

2) Measures: We measured *avg_confB_aff* and *avg_confB_nonaff* to count the average conflict blocks within design smell files and those outside design smell files. We defined the *Null Hypothesis (H_0)— there is no difference between the population of avg_confB_aff and the population of avg_confB_nonaff*. This hypothesis test reports *P-value* and *effect size*. If the evidence rejects H_0, it indicates that conflict blocks are significantly correlated with design smells.

If the evidence rejects H_0, we further calculated *Precision* = $\frac{|ConfF\_cap|}{|CapF|}$, *Recall* = $\frac{|ConfF\_cap|}{|ConfF|}$ and *P_ConfB_ds* = $\frac{|ConfB\_ds|}{|ConfB|}$. *Precision* measures the percentage of conflict files (i.e., *ConfF_cap*) involved in design smells to all design smell files (i.e., $CapF$). *Recall* measures the percentage of conflict files (i.e., *ConfF_cap*) involved in design smells to all conflict files (i.e., *ConfF*). *P_ConfB_ds* measures the percentage of conflict blocks (i.e., *ConfB_ds*) captured by design smells. Larger values of these measures will indicate the more likely that conflicts are involved in design smells.

We also measured *ConfB_recur*, the recurring conflict blocks that not only occurred more than one time but also were captured by design smells. Out of *ConfB_cap*, we computed the percentage of *ConfB_com*, *ConfB_ver*, and *ConfB_pro* that occurred repeatedly in different commits of the same versions, different versions of the same projects, and different projects.

TABLE V: Results of Wilcoxon sign-rank test in RQ2

P-value	*effect size*	*times*
0.002	0.974	2

3) Results: Table V presents the Wilcoxon Sign-Rank Test results with *P-value* of 0.002 and *effect size* of 0.974. It indicates that the number of code conflict blocks involved with design smell files is significantly larger than that of other files. On average, the files affected by design smells have been involved in 2 times (indicated by $times = 2$) more conflict blocks than the non-affected files.

Table VI lists the *Precision* and *Recall* results of conflicts involved with design smells. Using the project A-T as an example, among 329 files affected by design smells, 60 files (i.e., $Precision$=12.16%) are involved in merge conflicts; 40 out of conflicting files (i.e., $Recall$=66.67%) are involved in design smells.

Averaged on all projects, the $Recall$ is 68.87%; the $Precision$ is smaller than $Recall$ due to that our experiments only collected direct code conflicts during the merging process of Android variants with AOSP. In addition, Table VII showed that 72.26% conflict blocks have been involved in design smells averaged on all versions.

TABLE VI: The precision and recall results in RQ2

Project	*Precision*	*Recall*	Project	*Precision*	*Recall*
A-T	12.16%	66.67%	L-16.0	6.27%	69.57%
A-S	9.23%	73.77%	O-13.0	27.08%	61.90%
A-R	33.03%	64.70%	O-12.0	2.00%	22.22%
A-Q	44.81%	70.90%	O-11	35.08%	58.43%
C-13	22.00%	50.00%	O-10	10.53%	72.22%
C-12	1.33%	5.56%	O-9	3.52%	70.00%
C-11	1.11%	100%	I-E	22.60%	90.82%
L-20.0	29.55%	67.53%	I-D	21.75%	94.01%
L-19.1	0.76%	100%	I-C	22.35%	94.01%
L-18.1	3.64%	55.56%	I-B	20.44%	90.80%
L-17.1	7.78%	53.49%	I-A	14.64%	82.87%
Average	*Precision*: 15.98%			*Recall*: 68.87%	

TABLE VII: The conflict blocks involved with design smells

Project	|*ConfB*|	|*ConfB_ds*|	*P_ConfB_ds*	Project	|*ConfB*|	|*ConfB_ds*|	*P_ConfB_ds*
A-T	140	78	55.71%	L-16.0	37	29	78.38%
A-S	677	509	75.18%	O-13.0	111	56	50.45%
A-R	710	555	78.17%	O-12.0	15	4	26.67%
A-Q	479	362	75.57%	O-11	1311	972	74.14%
C-13	95	74	77.89%	O-10	65	47	72.31%
C-12	57	1	1.75%	O-9	16	12	75.00%
C-11	1	1	100%	I-E	2359	2261	95.85%
L-20.0	190	128	67.37%	I-D	2359	2304	97.67%
L-19.1	5	5	100%	I-C	2359	2304	97.67%
L-18.1	32	15	46.88%	I-B	2846	2754	96.77%
L-17.1	97	56	57.73%	I-A	1322	1170	88.50%
Average	|*ConfB*|: 695			|*ConfB_ds*|: 623			*P_ConfB_ds*: 72.26%

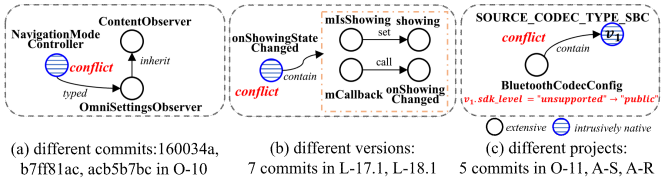
TABLE VIII: Recurring conflicts captured by design smells

|*ConfB_ds*|	|*ConfB_recur*| (%)	|*ConfB_com*| (%)	|*ConfB_ver*| (%)	|*ConfB_pro*| (%)
13,697	2,686 (**19.61%**)	1,093 (**7.98%**)	1,251 (**9.13%**)	1,433 (**10.46%**)

Table VIII summarizes the recurring conflict blocks that both were captured by design smells and occurred repeatedly. Among all 13,697 conflict blocks captured by design smells, 19.61% of them belong to *ConfB_recur*, recurring blocks that occurred more than one time in merge conflicts.

ConfB_com, *ConfB_ver*, and *ConfB_pro* columns indicate that 7.98%, 9.13%, and 10.46% of conflict blocks involved in design smells frequently occurred across commits, versions and projects respectively.

Figure 5 presents three cases of the recurring conflicts captured by design smells. The sub-figure (a) shows the CD smell-induced conflict, occurring repeatedly in 3 different commits of the O-10 project. A *native* `NavigationModeController` (`NMC` for short) in the upstream reversely depends on `OmniSettingsObserver`, causing a dependency cycle between upstream and downstream. When merging changes of `NMC` made by AOSP, conflicts occurred and their impact propagated into both modules via cycles. The sub-figure (b) depicts the ID smell-induced conflicts recurring in 7 commits across two versions of LineageOS. A *native* `onShowStateChanged` was modified to inline *extensive* code blocks. During every merging AOSP change, the same conflict occurred. Sub-figure (c) is the UD smell-induced conflict recurring in 5 commits across two projects (i.e., OmniROM and AOSPA). An *extensive* `BluetoothCodecConfig` relies on *native* `SOURCE_CODEC_TYPE_SBC` that is an unstable API initially decorated by *unsupported*. Due to its instability, AOSP often changed this API, causing conflicts frequently.



(a) different commits:160034a, b7ff81ac, acb5b7bc in O-10 (b) different versions: 7 commits in L-17.1, L-18.1 (c) different projects: 5 commits in O-11, A-S, A-R

Fig. 5: Three recurring conflict cases

RQ2 Summary. Our hypothesis test shows that the number of conflict blocks within design smell files is significantly larger than that of other files. Moreover, 72.26% of conflicts are involved in design smells; 19.61% of them are recurring conflicts across commits, versions, and projects. These observations imply that addressing design smells has the potential to mitigate Android update conflicts, especially recurring ones.

C. Mitigable Smell Analysis (RQ3)

We explore mitigable design smell instances that present an inconsistency between their intended dependencies and their defacto associated dependencies.

1) Study Setup: Industrial collaborators discussed that some design smell instances could be easily eradicated because their associated dependencies present redundant dependencies or lack intended dependencies. We term such instances as readily *mitigable* smells.

We summarized the *isMitigable* predicates and possible resolution in Table IX for several design smells based on the ODG defined in Section II-A.

We identified mitigable instances based on ODG automatically. Concretely, for each smell instance, we traversed each dependency, identifying missing dependencies or redundant dependencies as specified by *isMitigable* predicates. If the detection results are not null for this smell instance, we consider it a mitigable smell. We then observed the distribution of these smells across different types and projects.

In addition, we conducted a random sampling of 24 instances of mitigable design smell eliminations in the commit history for our case studies. We manually scrutinized the modifications made by developers in the commits, while observing the number of lines of code that were altered to address a mitigable design smell. If the number of modified code lines is relatively small, it suggests that mitigable smells can be easily resolved with minor code modifications.

TABLE IX: Mitigable design smells

An ID2 instance is mitigable if it misses *call* dependencies from other *extensive* entities to the *intrusively native* entities involved in this smell. **Solution:** adding *call* to remove intrusive modifications or add dependencies from *extensive* entites to *intrusively native* entities.
A UD1 instance is mitigable if it presents redundant dependencies of non-SDK API entities involved in this smell. **Solution:** removing the dependencies of non-SDK API.
A UD3 instance is mitigable if it presents redundant promotion of accessibility where the *actively native* entity is originally accessible. **Solution:** removing the intrusive modification to undo this promotion.
A UD4 instance is mitigable if it misses *inherit* or *override* dependencies from other *extensive* entities to the *intrusively native* entity. **Solution:** removing the intrusive modification of deleting *final*.
A UD5 instance is mitigable if it presents redundant *reflect* dependencies from *extensive* entities to *intrusively native* entity with *sdk_level="sdk"*. **Solution:** removing *reflect* dependencies and then depending on the *intrusively native* entity directly.
A CD1 instance is mitigable if it misses *inherit* dependencies from other *extensive* entities to the *intrusively native* entity. **Solution:** removing *inherit* dependency from the *actively native* entity to *extensive* entity.

2) Measures: We calculated MP_i (i denotes the design smell kind) to observe the percentage of mitigable instances to the total number of detected instances. A larger MP_i would suggest that a substantial portion of smell instances could be resolved easily and hence should be avoidable. Refactoring them is cost-effective to reduce the maintenance costs of the Android variants in the short term.

3) Results: Table X shows the distribution of *MP_i* in terms of design smell kinds. We can see that among all ID2 detection results in all project versions, 85.70% of instances are easily mitigable. The percentage for mitigable smells is the largest for ID2, followed by UD5 and UD3. On average, the mitigable instances take a portion of 52.37% among all smells that have *isMitigable* predicates.

Table XI lists the distribution of MP_i in terms of project versions. Considering the six kinds of smells that have *isMitigable* predicates, 17 out of 22 Android variants present more than 50% mitigable instances. Our case studies examined 24 instances of mitigable design smell eradicated in the commit history. Our findings revealed that 16 instances were rectified by developers through a single-line code modification, while 8 necessitated two lines of code adjustments. This indicates that mitigable smells could be fixed with minor code alterations.

The high percentage of mitigable smells and our case study results suggest that a significant number of design issues can be readily resolved. Our forthcoming research will entail analyzing a broader range of cases.

TABLE X: Mitigable design smells categorized by kinds

Smell	ID2	UD1	UD3	UD4	UD5	CD1	Average
MP_i	85.70%	30.71%	61.07%	47.58%	65.24%	23.93%	52.37%

TABLE XI: Mitigable design smells categorized by projects

Project	A-Q	A-R	A-S	A-T	C-13	C-12	C-11	L-20.0
MP_i	87.50%	81.82%	75.36%	88.89%	100%	93.33%	80.00%	93.67%
Project	L-19.1	L-18.1	L-17.1	L-16.0	O-13.0	O-12.0	O-11	O-10
MP_i	67.66%	70.33%	78.13%	76.39%	100%	83.33%	77.94%	68.75%
Project	O-9	I-E	I-D	I-C	I-B	I-A		
MP_i	62.16%	32.45%	32.40%	32.94%	33.13%	35.24%		

Figure 6 illustrates mitigable design smell cases in *IndustrialX* project: UD3 and UD4 smells about `ActivityManagerService` (i.e., `AMS`). In I-B version, the accessibility of `cleanUpApplicationRecordLocked` (i.e., `cleanUp`) is promoted from *default* to *protected* and its *final* modifier is also removed, producing UD3 and UD4 smells. Developers introduced them to extend `AMS` with a sub-class namely `XXActivityManagerService` and overwrote `cleanUp` method. Therefore, UD3 and UD4 instances are non-mitigable, i.e., the defacto dependencies associated with them are consistent with their intended dependencies. In a later I-C version, developers optimized the class inheritance for `AMS` by deprecating the base class and sub-class without rolling back the obsolete changes of accessibility and modifier. Consequently, UD3 and UD4 become mitigable ones. Developers eradicated these smells by readily removing those obsolete changes.

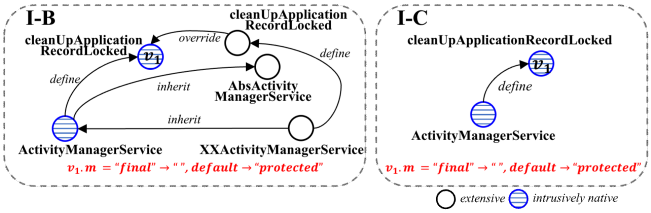


Fig. 6: A mitigable design smell case in IndustrialX project

RQ3 Summary. On average, 52.37% of design smell instances under investigation are mitigable. These mitigable smells might be resolved by minor code changes. The mitigable portion is the largest for ID2, followed by UD5 and UD3. The results are encouraging that a substantial portion of design smells can be mitigated.

D. Confounding Factors Analysis (RQ4)

Many studies revealed that other factors like code size and complexity are correlated with maintenance costs. In this RQ4, we investigated whether the detected design smells are not merely a set of files with larger code size, increased complexity, or object-oriented (OO) smells.

TABLE XII: P_{ol} results in RQ4

	Top 1%		Top 5%		Top 10%		Average	
Metric	SLOC	CC	SLOC	CC	SLOC	CC	SLOC	CC
P_{ol}	13.33%	6.52%	32.06%	31.06%	49.46%	49.86%	31.62%	29.15%
Smell	AWD	CH	FE	MH	SS		Average	
P_{ol}	44.60%	1.46%	20.89%	3.12%	10.30%		16.07%	

1) Study Setup: We utilized a Python module called Lizard [46] to calculate the Cyclomatic Complexity (CC) [47] and code size (SLOC) for each file. Next, we used GAP [48] to identify five types of OO smells, including Abstraction without Decoupling (AWD) [49], Cyclic Hierarchy (CH) [49], Feature Envy (FE) [50], Multipath Hierarchy (MH) [49], and Shotgun Surgery (SS) [50]. We chose these five OO smells because they have been well-studied in existing work [51, 52] and involve problematic dependencies.

2) Measures: We computed P_{ol}, similar to previous work [8], to assess the extent of overlap between files identified as having design smells and those with high values (i.e., top 1%, 5%, 10%) of SLOC, CC, or those affected by OO smells. Given the large scale of the Android projects we studied, averaging 4.65 million SLOC, we used three thresholds (1%, 5%, and 10%) to select files with the highest SLOC and CC. A lower P_{ol} value would suggest that our design smells are not dominated by large files and OO smells. This would imply that our design smells are new factors correlated with higher maintenance costs.

3) Results: The first three rows in Table XII illustrate P_{ol} in terms of SLOC and CC measures under different settings. Each cell denotes the value of P_{ol} averaged on all studied projects. Considering the top 5% configuration, we observed that 32.06% of design smell instances are found in the top 5% files with the largest size, and 31.06% of design smell instances are found in the top 5% files with the largest complexity. On average, P_{ol} is 31.62% and 29.15% in terms of SLOC and CC, respectively.

Similarly, the last two rows in Table XII list P_{ol} values regarding five different OO smells. Using FE (Feature Envy) as an example, $P_{ol} = 20.89\%$ indicates that only 20.89% of design smell files we detected are also affected by FE smells. We achieved consistent results for all OO smells across all projects, with an average value of $P_{ol} = 16.07\%$. In summary, our results indicate that the detected design smells are not dominated by large files or OO Smells.

RQ4 Summary. The results indicate that the design smell instances we detected are not dominated by large files (in terms of SLOC and CC) or OO smells. On average, only 31.62% of our design smell files are found in large files, 29.15% of design smells are found in complex files, and 16.07% of design smells we detected are found in OO smells.

V. Discussion

RQ1 and RQ4 results have demonstrated that files involved in design smells consume higher maintenance costs than other files and these design issues are new factors correlated with increased maintenance costs. In the practice of the industrial project, the architects have admitted these problematic dependencies are pain points to maintain. We advise that **practi-**

tioners can utilize DroidDS to localize problematic designs that hinder the maintainability of Android variants.

Merging AOSP changes into Android variants is a routine maintenance activity of Android updates, inevitably leading to code conflicts. Our RQ2 study revealed that 72.26% of conflicts can be captured by design smells. 19.61% of such conflicts are recurring ones across different versions and projects. The observations suggest that **fixing design smells has the potential to mitigate conflict risks.**

In our RQ3 results, we found that 33.23% of design smell instances in the five versions of the *IndustrialX* project are *mitigable*. We shared these results with the industrial developers working on the project. The developers admitted that they sometimes intentionally introduce design smells to expedite version releases. In response to our findings on mitigable smells, they are now **prioritizing refactoring efforts to address *mitigable* issues related to ID and UD** since these smells take a substantial portion.

The maintenance and co-evolution issues can also be observed in other software ecosystems. For example, OpenHarmony is a new open-source mobile platform at an earlier stage compared to Android and iOS [53]. In the web browser domain, popular browsers like Microsoft Edge [54, 55] and Brave Browser [56] are developed upon and evolve with the open-source Chromium [57]. **Our work will facilitate the studies of other ecosystems that consist of an original base software and divergent variants.**

VI. Related Work

Design Smell Detection. Lippert and Roock [17] provided an initial category of smells. Much work [7, 51, 58] followed it to enrich the categories. Cai et al. [8] conveyed a similar notion through the term "architecture anti-patterns", highlighting that such issues lead to heavy maintenance costs. Recent work focused on Cyclic Dependency[59, 60] and Unstable Dependency [20, 21, 61]. Some work [62–64] detected and refactored smells in Android applications (i.e., APPs). Various techniques have been designed to detect smells, such as metrics-based [65–67], optimization-based [68, 69], graph-based [70, 71], and learning-based ones [72, 73]. The work of [74] identified smells by predicting package-level dependencies. Jafari et al. [75] investigated dependency smells, i.e., library dependencies and configuration dependencies, in JavaScript. Unlike them, our work concerns the design smells eroding the boundary between Android variants and AOSP. We define these smells and propose DroidDS to reveal them.

Android Evolution Studies. The rapid evolution of the Android OS and the diverse customizations by manufacturers have brought many challenges such as compatibility issues [76, 77], complexity [5], deprecated API updates [78, 79], and security problems [80, 81]. Prior work proposed diverse solutions to address the issue of Android updates. Haryono et al.[82, 83] developed tools to update deprecated Android APIs for mobile APPs. Researchers [4, 6] empirically studied merge conflicts due to updates. Jin et al. [5] demystified the coupling between the AOSP and its various customizations. Our work is the first to formally define design smells for Android customizations, evaluating their severe impact on maintenance costs and merge conflicts.

VII. Threats to Validity

The first threat is the accuracy of our DroidDS. It is impossible to verify all design smells due to their large number of associated dependencies. To mitigate this, we randomly sampled 947 instances for manual checking, taking a portion of 4.6‰ from detection results. The recall and the precision achieved 100% and 98.94%. Second, we cannot claim that our work covers all patterns and smells that erode the design boundary between Android customizations and AOSP. We also can not claim that our results are generalizable across all Android projects. To reduce it, we collected 22 Android variants and 22 AOSP versions for an extensive study, including one industrial project. As far as we can know, our data set is the largest compared to related work. Third, Like previous studies [8, 40, 59], we are not claiming that we have considered all confounding factors (e.g., code refactoring) that can affect change-proneness and bug-proneness. Fourth, similar to previous research [8, 40], we conducted a correlation analysis between the presence of design smells and increased maintenance costs. We are not asserting a causal relationship between them. Another threat is the *isMitigable* predicates in Table IX. To be rigorous, we only reported the predicates that our industrial collaborators confirmed consistently. We will explore other *mitigable* smells.

VIII. Conclusion

We have formally defined the design smells that break the clear design boundary between Android variants and AOSP. We have designed DroidDS to detect these violations based on an Ownership-attributed Dependency Graph. Our results on 22 Android variant versions and 22 corresponding AOSP versions demonstrate the severe impact of design smells on Android variant maintenance. Files involved in design smells are change-prone and bug-prone, capturing more than half of update conflicts. Industrial collaborators responsible for the investigated commercial project have confirmed our results and embarked upon integrating DroidDS into their software engineering platform to alert erosion. Our work provides a foundation for developers to pinpoint and prioritize design issues that hinder the co-evolution of their Android customization with AOSP. We envision our research will also benefit other software ecosystems that consist of one original project and divergent variants.

Acknowledgment

This work was supported by National Key R&D Program of China (2022YFB2703500), National Natural Science Foundation of China (62232014, 62372368, 62272377, 62293501, 62293502, 62032010,62372367), the Fundamental Research Funds for the Central Universities, the Honor Corporation, and Young Talent Fund of Association for Science and Technology in Shaanxi, China.

REFERENCES

[1] Google. https://source.android.google.cn/.

[2] Z. Zhang, H. Zhang, Z. Qian, and B. Lau, "An investigation of the android kernel patch ecosystem," in *30th USENIX Security Symposium (USENIX Security 21)*, pp. 3649–3666, USENIX Association, Aug. 2021.

[3] Q. Hou, W. Diao, Y. Wang, X. Liu, S. Liu, L. Ying, S. Guo, Y. Li, M. Nie, and H. Duan, "Large-scale security measurements on the android firmware ecosystem," in *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, pp. 1257–1268, 2022.

[4] P. Liu, M. Fazzini, J. Grundy, and L. Li, "Do customized android frameworks keep pace with android?," in *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*, pp. 376–387, 2022.

[5] W. Jin, Y. Dai, J. Zheng, Y. Qu, M. Fan, Z. Huang, D. Huang, and T. Liu, "Dependency facade: The coupling and conflicts between android framework and its customization," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pp. 1674–1686, IEEE, 2023.

[6] M. Mahmoudi and S. Nadi, "The android update problem: An empirical study," in *Proceedings of the 15th International Conference on Mining Software Repositories (MSR)*, pp. 220–230, 2018.

[7] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic, "Toward a catalogue of architectural bad smells," in *Architectures for Adaptive Software Systems: 5th International Conference on the Quality of Software Architectures, QoSA 2009, East Stroudsburg, PA, USA, June 24-26, 2009 Proceedings 5*, pp. 146–162, Springer, 2009.

[8] R. Mo, Y. Cai, R. Kazman, L. Xiao, and Q. Feng, "Architecture anti-patterns: Automatically detectable violations of design principles," *IEEE Transactions on Software Engineering (TSE)*, vol. 47, no. 5, pp. 1008–1028, 2019.

[9] N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F. Le Meur, "Decor: A method for the specification and detection of code and design smells," *IEEE Transactions on Software Engineering (TSE)*, vol. 36, no. 1, pp. 20–36, 2009.

[10] A. Blouin, V. Lelli, B. Baudry, and F. Coulon, "User interface design smell: Automatic detection and refactoring of blob listeners," *Information and Software Technology*, vol. 102, pp. 49–64, 2018.

[11] H. Zhu, Y. Li, J. Li, and X. Zhang, "An efficient design smell detection approach with inter-class relation," in *Proceedings of the International Conference on Software Engineering and Knowledge Engineering (SEKE)*, pp. 181–186, 2023.

[12] R. C. Martin, "Design principles and design patterns," *Object Mentor*, vol. 1, no. 34, p. 597, 2000.

[13] Ours. https://github.com/xjtu-enre/ICSE2025DroidDS.

[14] F. Thung, S. A. Haryono, L. Serrano, G. Muller, J. Lawall, D. Lo, and L. Jiang, "Automated deprecated-api usage update for android apps: How far are we?," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 602–611, IEEE, 2020.

[15] Y. Zhao, L. Li, K. Liu, and J. Grundy, "Towards automatically repairing compatibility issues in published android apps," in *Proceedings of the 44th International Conference on Software Engineering (ICSE)*, pp. 2142–2153, 2022.

[16] R. C. Martin, *Agile software development: principles, patterns, and practices*. Prentice Hall PTR, 2003.

[17] M. Lippert and S. Roock, *Refactoring in large software projects: performing complex restructurings successfully*. John Wiley & Sons, 2006.

[18] M. Fowler and K. Beck, *Refactoring: improving the design of existing code*. USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[19] T. Mens, "A state-of-the-art survey on software merging," *IEEE Transactions on Software Engineering (TSE)*, vol. 28, no. 5, pp. 449–462, 2002.

[20] F. A. Fontana, I. Pigazzini, R. Roveda, and M. Zanoni, "Automatic detection of instability architectural smells," in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 433–437, IEEE, 2016.

[21] F. A. Fontana, I. Pigazzini, R. Roveda, D. Tamburri, M. Zanoni, and E. Di Nitto, "Arcan: A tool for architectural smells detection," in *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pp. 282–285, IEEE, 2017.

[22] "Restrictions on non-sdk interfaces." https://developer.android.com/guide/app-compatibility/restrictions-non-sdk-interfaces.

[23] J. Lakos, "Large-scale c++ software design," *Reading, MA*, vol. 173, pp. 217–271, 1996.

[24] H. Melton and E. Tempero, "An empirical study of cycles among classes in java," *Empirical Software Engineering*, vol. 12, pp. 389–415, 2007.

[25] T. D. Oyetoyan, D. S. Cruzes, and R. Conradi, "A study of cyclic dependencies on defect profile of software components," *Journal of Systems and Software*, vol. 86, no. 12, pp. 3162–3182, 2013.

[26] H. V. Nguyen, M. H. Nguyen, S. C. Dang, C. Kästner, and T. N. Nguyen, "Detecting semantic merge conflicts with variability-aware execution," *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (FSE)*, 2015.

[27] R. Mo, Y. Cai, R. Kazman, L. Xiao, and Q. Feng, "Decoupling level: a new metric for architectural maintenance complexity," in *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, pp. 499–510, IEEE, 2016.

[28] W. Jin, Y. Cai, R. Kazman, Q. Zheng, D. Cui, and T. Liu, "Enre: a tool framework for extensible entity relation extraction," in *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings*, pp. 67–70, IEEE Press, 2019.

[29] Git. https://git-scm.com/doc.

[30] AOSPA. https://github.com/AOSPA/android_frameworks_base/.

[31] CalyxOS. https://gitlab.com/CalyxOS/platform_frameworks_base/.

[32] LineageOS. https://github.com/LineageOS/android_frameworks_base/.

[33] OmniROM. https://github.com/omnirom/android_frameworks_base/.

[34] K. El Emam, S. Benlarbi, N. Goel, and S. N. Rai, "The confounding effect of class size on the validity of object-oriented metrics," *IEEE Transactions on Software Engineering*, vol. 27, no. 7, pp. 630–650, 2001.

[35] S. M. Olbrich, D. S. Cruzes, and D. I. Sjøberg, "Are all code smells harmful? a study of god classes and brain classes in the evolution of three open source systems," in *2010 IEEE international conference on software maintenance*, pp. 1–10, IEEE, 2010.

[36] D. Landman, A. Serebrenik, and J. Vinju, "Empirical analysis of the relationship between cc and sloc in a large corpus of java methods," in *2014 IEEE International Conference on Software Maintenance and Evolution*, pp. 221–230, IEEE, 2014.

[37] D. Port, B. Taber, and L. Huang, "Investigating a nasa cyclomatic complexity policy on maintenance risk of a critical system," in *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pp. 211–221, IEEE, 2023.

[38] W. Jin, Y. Cai, R. Kazman, G. Zhang, Q. Zheng, and T. Liu, "Exploring the architectural impact of possible dependencies in python software," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 758–770, 2020.

[39] W. Jin, D. Zhong, Y. Cai, R. Kazman, and T. Liu, "Evaluating the impact of possible dependencies on architecture-level maintainability," *IEEE Transactions on Software Engineering (TSE)*, vol. 49, no. 3, pp. 1064–1085, 2022.

[40] M. Nayebi, Y. Cai, R. Kazman, G. Ruhe, Q. Feng, C. Carlson, and 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)*, pp. 171–180, IEEE, 2019.

[41] H. Liu, B. Li, Y. Yang, W. Ma, and R. Jia, "Exploring the impact of code smells on fine-grained structural change-proneness," *International Journal of Software Engineering and Knowledge Engineering*, vol. 28, no. 10, pp. 1487–1516, 2018.

[42] D. Sas, P. Avgeriou, and U. Uyumaz, "On the evolution and impact of architectural smells—an industrial case study," *Empirical Software Engineering*, vol. 27, no. 4, p. 86, 2022.

[43] F. Wilcoxon, "Individual comparisons by ranking methods," in *Breakthroughs in statistics: Methodology and distribution*, pp. 196–202, Springer, 1992.

[44] J. Cohen, *Statistical power analysis for the behavioral sciences*. routledge, 2013.

[45] J. W. Hunt and T. G. Szymanski, "A fast algorithm for computing longest common subsequences," *Communications of the ACM*, vol. 20, no. 5, pp. 350–353, 1977.

[46] Lizard. https://github.com/terryyin/lizard.

[47] T. J. McCabe, "A complexity measure," *IEEE Transactions on software Engineering*, no. 4, pp. 308–320, 1976.

[48] GAP. https://github.com/gaptools/GAP.

[49] J. Dietrich, C. McCartin, E. Tempero, and S. M. A. Shah, "Barriers to modularity-an empirical study to assess the potential for modularisation

of java programs," in *International Conference on the Quality of Software Architectures*, pp. 135–150, Springer, 2010.
[50] M. Fowler, *Refactoring - Improving the Design of Existing Code*. Addison Wesley object technology series, Addison-Wesley, 1999.
[51] U. Azadi, F. A. Fontana, and D. Taibi, "Architectural smells detected by tools: a catalogue proposal," in *2019 IEEE/ACM International Conference on Technical Debt (TechDebt)*, pp. 88–97, IEEE, 2019.
[52] H. Mumtaz, P. Singh, and K. Blincoe, "Analyzing the relationship between community and design smells in open-source software projects: An empirical study," in *Proceedings of the 16th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pp. 23–33, 2022.
[53] L. Li, X. Gao, H. Sun, C. Hu, X. Sun, H. Wang, H. Cai, T. Su, X. Luo, T. F. Bissyandé, J. Klein, J. C. Grundy, T. Xie, H. Chen, and H. Wang, "Software engineering for openharmony: A research roadmap," *CoRR*, vol. abs/2311.01311, 2023.
[54] M. Edge. https://www.microsoft.com/en-us/edge.
[55] C. Sung, S. K. Lahiri, M. Kaufman, P. Choudhury, and C. Wang, "Towards understanding and fixing upstream merge induced conflicts in divergent forks: An industrial case study," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*, pp. 172–181, 2020.
[56] B. Browser. https://github.com/brave/brave-browser.
[57] Chromium. https://github.com/chromium/chromium.
[58] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic, "Identifying architectural bad smells," in *2009 13th European Conference on Software Maintenance and Reengineering*, pp. 255–258, IEEE, 2009.
[59] L. Xiao, Y. Cai, R. Kazman, R. Mo, and Q. Feng, "Detecting the locations and predicting the maintenance costs of compound architectural debts," *IEEE Transactions on Software Engineering (TSE)*, vol. 48, no. 9, pp. 3686–3715, 2021.
[60] Q. Feng, S. Liu, H. Ji, X. Ma, and P. Liang, "An empirical study of untangling patterns of two-class dependency cycles," *Empirical Software Engineering*, vol. 29, no. 2, p. 56, 2024.
[61] A. Agrawal, X. Yang, R. Agrawal, R. Yedida, X. Shen, and T. Menzies, "Simpler hyperparameter optimization for software analytics: Why, how, when?," *IEEE Transactions on Software Engineering (TSE)*, vol. 48, no. 8, pp. 2939–2954, 2021.
[62] J. Reimann, M. Brylski, and U. Aßmann, "A tool-supported quality smell catalogue for android developers," in *Proc. of the conference Modellierung 2014 in the Workshop Modellbasierte und modellgetriebene Softwaremodernisierung–MMSM*, vol. 2014, 2014.
[63] S. Habchi, R. Rouvoy, and N. Moha, "On the survival of android code smells in the wild," in *2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pp. 87–98, IEEE, 2019.
[64] F. Palomba, D. Di Nucci, A. Panichella, A. Zaidman, and A. De Lucia, "On the impact of code smells on the energy consumption of mobile applications," *Information and Software Technology*, vol. 105, pp. 43–55, 2019.
[65] N. Tsantalis and A. Chatzigeorgiou, "Identification of move method refactoring opportunities," *IEEE Transactions on Software Engineering (TSE)*, vol. 35, no. 3, pp. 347–367, 2009.
[66] S. A. Vidal, C. Marcos, and J. A. Díaz-Pace, "An approach to prioritize code smells for refactoring," *Automated Software Engineering*, vol. 23, pp. 501–532, 2016.
[67] D. Tiwari, H. Washizaki, Y. Fukazawa, T. Fukuoka, J. Tamaki, N. Hosotani, and M. Kohama, "Metrics driven architectural analysis using dependency graphs for c language projects," in *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, vol. 1, pp. 117–122, IEEE, 2019.
[68] W. Kessentini, M. Kessentini, H. Sahraoui, S. Bechikh, and A. Ouni, "A cooperative parallel search-based software engineering approach for code-smells detection," *IEEE Transactions on Software Engineering (TSE)*, vol. 40, no. 9, pp. 841–861, 2014.
[69] S. M. Al Khatib, K. Alkharabsheh, and S. Alawadi, "Selection of human evaluators for design smell detection using dragonfly optimization algorithm: An empirical study," *Information and Software Technology*, vol. 155, p. 107120, 2023.
[70] A. Tommasel, "Applying social network analysis techniques to architectural smell prediction," in *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*, pp. 254–261, IEEE, 2019.
[71] I. Pigazzini, "Automatic detection of architectural bad smells through semantic representation of code," in *Proceedings of the 13th European Conference on Software Architecture (ECSA)*, pp. 59–62, 2019.
[72] H. Liu, J. Jin, Z. Xu, Y. Zou, Y. Bu, and L. Zhang, "Deep learning based code smell detection," *IEEE Transactions on Software Engineering (TSE)*, vol. 47, no. 9, pp. 1811–1837, 2019.
[73] K. Alkharabsheh, S. Alawadi, V. R. Kebande, Y. Crespo, M. Fernández-Delgado, and J. A. Taboada, "A comparison of machine learning algorithms on design smell detection using balanced and imbalanced dataset: A study of god class," *Information and Software Technology*, vol. 143, p. 106736, 2022.
[74] A. Tommasel and J. A. Diaz-Pace, "Identifying emerging smells in software designs based on predicting package dependencies," *Engineering Applications of Artificial Intelligence*, vol. 115, p. 105209, 2022.
[75] A. J. Jafari, D. E. Costa, R. Abdalkareem, E. Shihab, and N. Tsantalis, "Dependency smells in javascript projects," *IEEE Transactions on Software Engineering (TSE)*, vol. 48, no. 10, pp. 3790–3807, 2022.
[76] D. Han, C. Zhang, X. Fan, A. Hindle, K. Wong, and E. Stroulia, "Understanding android fragmentation with topic analysis of vendor-specific bugs," in *2012 19th Working Conference on Reverse Engineering*, pp. 83–92, IEEE, 2012.
[77] P. Liu, Y. Zhao, M. Fazzini, H. Cai, J. Grundy, and L. Li, "Automatically detecting incompatible android apis," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 33, no. 1, pp. 1–33, 2023.
[78] T. McDonnell, B. Ray, and M. Kim, "An empirical study of api stability and adoption in the android ecosystem," in *2013 IEEE International Conference on Software Maintenance*, pp. 70–79, IEEE, 2013.
[79] L. Li, J. Gao, T. F. Bissyandé, L. Ma, X. Xia, and J. Klein, "Cda: Characterising deprecated android apis," *Empirical Software Engineering*, vol. 25, pp. 2058–2098, 2020.
[80] S. Yang, R. Li, J. Chen, W. Diao, and S. Guo, "Demystifying android non-sdk apls: Measurement and understanding," in *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, pp. 647–658, 2022.
[81] Y. He, Y. Gu, P. Su, K. Sun, Y. Zhou, Z. Wang, and Q. Li, "A systematic study of android non-sdk (hidden) service api security," *IEEE Transactions on Dependable and Secure Computing*, 2022.
[82] S. A. Haryono, F. Thung, H. J. Kang, L. Serrano, G. Muller, J. Lawall, D. Lo, and L. Jiang, "Automatic android deprecated-api usage update by learning from single updated example," in *Proceedings of the 28th international conference on program comprehension (ICPC)*, pp. 401–405, 2020.
[83] S. A. Haryono, F. Thung, D. Lo, L. Jiang, J. Lawall, H. J. Kang, L. Serrano, and G. Muller, "Androevolve: automated android api update with data flow analysis and variable denormalization," *Empirical Software Engineering*, vol. 27, no. 3, p. 73, 2022.