Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

ARTICLE TYPE

Empirical Investigation of the Relationship Between Design Smells and Role Stereotypes

Daniel Ogenrwot[1] | **Joyce Nakatumba-Nabende**[2] | **John Businge**[1] | **Michel R.V. Chaudron**[3]

[1]Department of Computer Science, College of Engineering, University of Nevada, Las Vegas, Nevada, United States of America

[2]Department of Computer Science, Makerere University, Uganda

[3]Department of Mathematics and Computer Science, Eindhoven University of Technology, Netherlands

Correspondence
Corresponding author Daniel Ogenrwot, Department of Computer Science, College of Engineering, University of Nevada, Las Vegas, United States of America.
Email: ogenrwot@unlv.nevada.edu

Abstract

During the development of software systems, poor design and implementation choices can have a detrimental impact on the maintainability of the software. Design smells, recurring patterns of poorly designed fragments in software, are indicative of these issues. Role-stereotypes signify the generic responsibilities that classes assume in system design. Although the concepts of role-stereotypes and design smells are inherently different, both significantly contribute to the design and maintenance of software systems. Studying this relation is essential for (i) software maintainability, (ii) code quality improvement, (iii) efficient code review, (iv) guided refactoring, (v) early defect detection and (vi) role-specific metrics design. This paper employs an exploratory approach, combining statistical analysis and unsupervised learning methods, to comprehend the relationship between design smells and role-stereotypes and how this connection varies across different desktop and mobile applications. The study utilizes a dataset comprising 11,350 classes across 30 Java projects mined from GitHub. Overall, the findings reveal several design smells that co-occur more frequently across the entire role-stereotype categories. Specifically, three (3) out of six (6) role-stereotypes considered in this study are more susceptible to design smells. Additionally, the study identifies that design smells are more prevalent in desktop applications than in mobile applications, especially in the Service Provider and Information Holder role-stereotypes. Through unsupervised learning methods, it is observed that certain pairs or groups of role-stereotypes are prone to similar types of design smells compared to others. We believe that these relationships may be associated with the characteristic and collaborative properties between role-stereotypes. Therefore, this study offers crucial insights into previously undisclosed behavior regarding the relationship between design smells and role-stereotypes. The results of this paper can guide software teams in implementing various design smell prevention and correction mechanisms, as well as ensuring the conceptual integrity of classes during their design and maintenance.

KEYWORDS
Software design, class role stereotype, design smells, code smells, clustering

1 | INTRODUCTION

Software design is an essential component of software engineering. A well-designed software system leads to reliable and maintainable software[1,2]. New software applications nowadays are quite complex and constantly adapting to the ever-changing user requirements, which poses a challenge of maintainability. The ever increasing maintenance costs[3] are often a consequence of bad design and development practices commonly observed in software systems. While developing software systems, poor design and implementation choices in source code can negatively affect the maintainability of software systems[4]. The problematic structures identified in source code are known as design smells[5,6,2]. Design smells represent structural anomalies that deviate from established design principles, introducing technical debt and impeding the overall effectiveness of the software. For instance, a “God class" assumes too many responsibilities and usually has very many methods, violating the single-responsibility principle. This design smell is a huge technical debt which affects program readability and comprehension. While the presence

Abbreviations: OSS, open source software; APC, antigen-presenting cells; IRF, interferon regulatory factor.

of design smells does not impede the operation of the software system, it can affect software development, sustainability, and increase the likelihood of software failure[7]. Therefore, mitigating design smells in the source code is imperative for enhancing overall software quality. In the ever-evolving landscape of software development, mobile and desktop applications emerge as predominant software systems. Notably, mobile applications undergo more frequent updates compared to their desktop counterparts[8]. Nevertheless, the need for enhanced software maintenance is important for improving software quality irrespective of the underlying ecosystems.

Responsibility (role) stereotypes indicate generic responsibilities that classes play in the design of a software system. These roles include; Coordinator (CO), Structurer (ST), Controller (CT), Information Holder (IH), Interfacer (IT), and Service Provider (SP) as initially classified by Wirfs-Brock[9]. In this paradigm, classes are characterized according to the type of responsibility they play in the software design[9]. Knowledge about role-stereotypes has proven very helpful in various tasks of software development and maintenance, such as program understanding, program summarization and quality assurance. Role-stereotypes have also been used in creating better layouts of class diagrams which results in improved program comprehension[10,11,12,13]. Besides this, enhancing source code with role-stereotype information helps improve feature location in source code[14]. The benefits of role-stereotypes can be observed as an enrichment of reverse engineering, from the perspective of under-covering software design choices[11,13].

Although the concepts of role-stereotypes and design smells are widely divergent, both are significant contributors to the design and maintenance of software systems. Several studies related to design smells focus on developing detection methods and tools to improve software quality[15,16,17]. On the other hand, studies related to role-stereotypes focuses on classification[18,11] and application of role-stereotypes for example in UML class diagram design[19]. The substantial interest in enhancing software design and maintainability within the software engineering industry reinforces the importance of understanding the correlation between design smells and role-stereotypes.

Building this empirical knowledge can provide the following significance; (1) **software maintainability:** understanding the correlation between design smells and role stereotypes can help in identifying patterns that impact software maintainability. This knowledge aids developers in proactively addressing design issues to prevent long-term software degradation, (2) **improving code quality:** identifying and mitigating design smells contributes to overall code quality improvement. Developers can prioritize and address design issues that are more likely to impact the specific role stereotypes, (3) **efficient code reviews:** guide code reviews. Developers can focus on areas of the codebase that are more likely to be associated with specific roles, leading to more efficient and targeted code inspections, (4) **guided code refactoring:** knowing the impact of design smells on role stereotypes provides valuable guidance for refactoring efforts. Developers can prioritize refactoring activities based on the roles affected, ensuring that improvements align with the intended behavior and responsibilities of each software component, (5) **early defect detection:** this is particularly important in the software development life cycle, where addressing issues early on is more cost-effective than dealing with them in later stages and (6) **role-specific metrics:** the empirical knowledge gained in this area can contribute to the development of role-specific metrics for assessing the health and quality of software components.

Despite the aforementioned significance, the relationship between design smells and role-stereotypes remains unclear due to the structural complexity of source code across diverse software application ecosystems. To this end, we present an exploratory analysis based on statistical and machine learning methods to understand the relation between design smells and role-stereotypes. Machine learning techniques are able to discover local patterns and make inferences from complex feature representations in a given dataset[20]. Recently, unsupervised learning methods have been leveraged to study design smells in static code analysis and have received commendable results[21,1,22,23]. Our study is based on 11,350 Java classes of 30 open source Java-based projects mined from GitHub. To the best of our knowledge, this is the first study on establishing intrinsic relationship between design smells and role stereotypes. As such, we hope to open up the possibility of enhancing design smell metrics with role-stereotype properties as previously observed[11]. The main contributions of the paper are:

(1) The study provides a step-by-step approach to build a fine-grained dataset comprising of a combination of design smells and role-stereotype classification data. As a result, we publish a sizable dataset of 11,350 Java classes which can serve as a resource for other researchers.
(2) Using the perspective of role-stereotypes, we presents a comparison of the occurrence of design smells in desktop versus mobile applications. We show the relevance of role-stereotype information in dealing with design smells.
(3) The study provides a clustering approach to analyze the groupings of role-stereotypes that are prone to similar categories of design smells.
(4) Finally, the paper provides insights to software developers, designers and researchers on previously concealed behavior and relationships between design smells and role-stereotypes.

The rest of this paper proceeds as follows: In Section 2, we present the background of the study, derive terminologies and provide a concrete example to motivate the need for the study. Section 3 outlines related work and identifies study gaps. Next, in Section 4, we provide a comprehensive explanation of the research questions, tools, methods, and analysis performed. The results of our analysis are presented in Section 5. Section 6 focuses on discussions and implications of the study. The threats to validity are discussed in Section 7. Finally, Section 8 concludes the paper and provides direction for future work.

2 | BACKGROUND

In this section, we derive important terminologies used throughout the study and provide a concrete example illustrating the intrinsic relationship between design smells and role-stereotypes, which can be observed through code inspection. The objective of this illustration is to validate the co-occurrence of design smells and role stereotypes in actively maintained codebases, highlighting the necessity for deeper exploration into their co-occurrence dynamics. As previously noted, while role stereotypes and design smells are loosely distinct concepts, both exert significant influence on the design and maintenance of software systems.

2.1 | Terminology

- **Role stereotypes:** Denote generic responsibilities that classes undertake in the design of a software system. We employ terms such as “class role”, “class responsibility”, and “class role stereotypes” interchangeably. It is important to acknowledge that the formal definition is derived from a software design standpoint. However, in this paper, we examine role stereotypes from the implementation perspective.
- **Design smells:** Indication of poor design and implementation choices, leading to problematic source code structures. In the paper, the term “antipattern”, “bad smell”, “code smell” has the same meaning as design smell.
- **class path:** Refers to the fully qualified class name or file name of a Java class. Following[24] guideline, we use Java classes as our unit of analysis.
- **Regex:** Refer to regular expression, used to extract class path from the output of design smell detection.
- **Co-occurrence:** Refers to simultaneous presence of two or more design smells within a single role stereotype. For example; if “LongParameterList“ and “ComplexClass” are detected together in Information Holder role stereotype, then we say “LongParameterList“ and “ComplexClass” co-occur.
- **Cluster**: In this study, a cluuster refers to group of design smells or role-stereotypes with shared characteristics.

2.2 | Concrete Example

To put the problem into context, let us consider a practical example involving code refactoring as a use case. In Listing 1, we examine the `ImapStoreSettings.java` class within the **K9 Mail app**. This class encapsulates IMAP store settings, offering methods for accessing, manipulating, and retrieving additional settings in key-value pairs. We categorize this class under the role stereotype of an `Information Holder`. It is evident that the code is impacted by the `LongParameterList` design smell, highlighted in lines `10-12`. In Listing 2, we present a sample output of the `LongParameterList` design smell detected in the K9 Mail project using the Software Architectural Defects (SAD) tool, with line `19` showcasing the `ImageStoreSettings.java` class.

Assuming a developer aims to refactor the code at lines `10-12` in Listing 1, it becomes essential to understand not only the design smell and the class role stereotype but also how they are interrelated. For instance, the developer might opt to address the issue by implementing the *Builder* design pattern. This involves creating a *Builder* class responsible for constructing an `ImapStoreSettings` object, allowing clients to set specific attributes before building the final object. However, this approach might be more suitable for a `Controller` class role, as the *Builder* pattern is better suited for object construction. Since the `Information Holder` class focuses on data transfer, a more appropriate refactoring strategy would involve implementing a structural design pattern, such as the Data Transfer Object (DTO) pattern.

LISTING 1 Code snippet of `ImapStoreStettings.java` IH class of K9 Mail project. Line `10-12` highlighted indicate a `LongParameterList` design smell

```java
/**
* This class is used to store the decoded contents of an ImapStore URI.
*
* @see ImapStore#decodeUri(String)
*/
public class ImapStoreSettings extends ServerSettings {

  ...

  protected ImapStoreSettings(String host, int port, ConnectionSecurity connectionSecurity,
  AuthType authenticationType, String username, String password, String clientCertificateAlias,
    boolean autodetectNamespace, String pathPrefix){

    super(Type.IMAP, host, port, connectionSecurity, authenticationType, username,
    password, clientCertificateAlias);

    this.autoDetectNamespace = autodetectNamespace;
    this.pathPrefix = pathPrefix;
  }

  ...

}
```

LISTING 2 Example `LongParameterList` design smell detected from K9 Mail project using SAD tool. Line `19` shows `ImageStoreSettings.java` class

```
# Results of the detection

...

# ------>LongParameterList num: 17

17.100.Name = LongParameterList

#LongParameterListClass
17.100.LongParameterListClass-0 = k9mail.src.main.java.com.fsck.k9.activity.compose.RecipientPresenter
17.100.LongParameterListClass-0.NOParam-0 = 9.0
17.100.LongParameterListClass-0.NOParam_MaxBound-0 = {NOParam_MaxBound=6.0}

# ------>LongParameterList num: 18

18.100.Name = LongParameterList

#LongParameterListClass
18.100.LongParameterListClass-0 = k9mail-library.src.main.java.com.fsck.k9.mail.store.imap.ImapStoreSettings
18.100.LongParameterListClass-0.NOParam-0 = 9.0
18.100.LongParameterListClass-0.NOParam_MaxBound-0 = {NOParam_MaxBound=6.0}

...
```

Form the above code snippets, we can ask several questions. For example, are Information Holders more prone to `LongParameterList` then other role-stereotypes? How often does `LongParameterList` occur? Is the occurrence influenced by the type of application - desktop or mobile app? What design choices should be considered for Information Holder role-stereotypes to mitigate occurrence of `LongParameterList`? What refactoring opportunities are possible and how can they be applied? To answer those questions and many other related ones, it is important to build empirical knowledge on the relation between design smells and role-stereotypes.

3 | RELATED WORK

3.1 | Design Smells

Design smells incorporate low-level or local source code problems such as code smells[25]. Code smells are indicators of defects in source code that affect software quality and evolution[26,27,28]. Fowler[29] outlined 22 code smells and their corresponding refactoring techniques. Although code smells do not prevent the software from functioning, their existence in source code can lead to a myriad of problems such as weakening the sustainability of the software and increasing the probability of its failure[7].

Therefore, mitigating design/code smells is critical for highly scalable and maintainable software. Refactoring is recommended as an intervention aimed at improving the internal structure of existing software code without affecting its observable behavior. It enhance software maintainability and generate a more manageable internal architecture[7]. Design smells encompass a broader scope and have a significant impact and hence, refactoring a design smell may result in alterations across a set of classes[30].

Several authors have explored different approaches for detecting code smells from source code. Bafandeh Mayvan et al[31] used a metric-based design smell detection approach to detect bad code smells. This was achieved through cross-referencing each instance of the detected bad smell with the corresponding refactoring technique. Kaur & Singh[16] presented a supervised-based machine learning algorithm for detecting software code smells from design patterns. The authors considered four (4) design patterns and five (5) code smells. Using J48 decision tree classifier, it was observed that there is a relationship between some design patterns with specific code smells. Saranya et al.[32] applied optimization-based approach on model-level code smell detection using Euclidean distance-based Genetic Algorithm and Particle Swarm Optimization (EGAPSO). The study was inspired by the weakness of the metric-based code smell detection approach.

3.2 | Role-Stereotypes

Role-stereotypes indicate generic responsibilities that classes play in the design of a system. Wirfs-Brock[9] proposed six (6) generic categories (shown in Table 1). The classification is intended to convey an important part of the design intention of a class. The benefits of role stereotypes are observed in software development and maintenance tasks, such as program comprehension, program summarization, quality assurance and in the creation of layouts for class diagrams[10,11,12].

TABLE 1 Generic Role-Stereotype Taxonomy[9,13].

Role-Stereotype	Description
Coordinator	An object that does not make many decisions but, in a rote or mechanical way, delegates work to other objects.
Structurer	An object that maintains relationships between objects and information about those relationships.
Controller	An object designed to make decisions and control a complex task.
Information Holder	An object designed to know certain information and provide that information to other objects.
Interfacer	An object that transforms information or requests between distinct parts of a system.
Service Provider	An object that performs specific work and offers services to others on demand.

Based on the benefits that role-stereotypes offer to software design, several researchers have explored the classification of class role-stereotypes in source code. Earlier work by Dragan et al[18] focused on automatic classification of a class's stereotype for C++ source code. The proposed method consists of rules based on both the stereotype and category distributions of the class signature. Moreno & Marcus[33] extended Dragan's work to Java source code. They developed "JStereoCode", a tool that automatically identifies the stereotypes of methods and classes in Java systems. The studies in[18] and[33] are based on expert-designed decision rules that are applied to the syntactical characteristics of the class source code.

Nurwidyantoro et al[11] provides a more recent study in this direction by applying machine learning models for improved automated classification of role-stereotypes in Java classes. Source code classes were classified in one of the six role-stereotypes taxonomy proposed by Wirfs-Brock[9]. The result shows that the Random Forest algorithm enhanced by SMOTE resampling yielded the best performance compared to Multinomial Naive Bayes (MNB) and Support Vector Machine (SVM) models.

3.3 | Research Gaps

In this section, we discuss related work done to understand the relationship between design smells and role-stereotypes. We also try to unveil some gaps in the current body of literature to justify the need for this study. It is important to note that, to the best of our knowledge, no formal in-depth study has been done so far to assess the relationship between design smells and role-stereotypes. Most research attention from both academia and software industry has been directed towards developing various detection and classification methods and tools for design smell and role-stereotype respectively. Some authors have tried to study the relationship between design smells and design patterns[34,35,36,37] but not role-stereotypes.

Jaafar et al[34] analyzed the static relationship between anti-patterns and design patterns. They studied anti-patterns dependencies with other classes within particular design patterns to understand how developers can maintain programs containing those anti-patterns. The study was performed on 1,191 to 3,325 classes of three (3) Java projects (*ArgoUML, JFreeChart, and XercesJ*). It was observed that there is a temporary relationship between anti-pattern and design patterns. However, this study strictly focused on dependencies between anti-patterns and design patterns but not role-stereotypes. Besides, only particular sets of anti-patterns and design patterns were used yet different anti-patterns and design patterns could potentially influence the outcome. The work in[34] did not consider the type of systems (desktop, mobile or web-based) and how system type can impact the dependencies between anti-patterns and design patterns.

Walter & Alkhaeir[35] conducted a study related to that of Jaafar et al[34] to understand how the presence of design patterns impacts the existence of code smells. Based on an exploratory approach, 9 design patterns and 7 code smells were analyzed from two sizeable open-source Java projects. Their findings indicated that classes which participate in design patterns appear to display code smells less frequently than other classes. The observed effect was stronger for some patterns (e.g., Singleton, State-Strategy) and weaker for others (e.g., Composite). However, this study does consider whether role-stereotypes could influence the association of design patterns with code smells. The type of application was also not considered in this study. The impact of design patterns on the presence of code smells might differ between desktop, mobile and web applications.

Mannan et al[38] focused mainly on code smells which tends to affect readability and simplicity. The task of refactoring in this case involves renaming or extracting to methods. Design smells, on the other hand, tend to be more subtle. They usually affect maintainability and flexibility. Next, their study[38] did not look at the general co-occurrence of code smells and how these co-occurrences varies across desktop and android application. In addition, this study did not look at role-stereotypes. Palomba et al[22] studied the co-occurrence of different types of smells on the same code component. It was observed that some code smells frequently co-occur and the occurrence of class-level seems to originate from method-level code smells. The projects selected in this study comprises of mixture of Java desktop applications and libraries, whose internal implementation slightly differs. Furthermore, the study did not look at the association of role-stereotypes and co-occurrence of code smells.

In summary, although some work has been carried out to understand design smells and role-stereotypes, little has been done to disclose the non-innate relation between them and how the type of application - mobile versus desktop influence these relations.

4 | STUDY DESIGN

Overall, we are interesting in building empirical knowledge on the relationship between design smells and role stereotypes. To this end, we outline three research questions detailed in subsection 4.1, followed by data collection and analysis steps.

4.1 | Research Questions

Specifically, our study aims at answering the following research questions:

- **RQ1:** *How do design smells vary across desktop and mobile applications?* The aim is to investigate and understand the differences in the occurrence and distribution of design smells in software systems developed for desktop applications compared to mobile applications. We are also interested in the co-occurrence of these smells. This will give us better understanding of the nature and characteristics of design smells across software ecosystems. Julio et al.,[39] noted the need for further research on the impact of code smells co-occurrences on internal quality attributes.
- **RQ2:** *How do design smells relate with different types of role-stereotypes?* In this research question, the focus is to explore the associations and dependencies between design smells and various role-stereotypes within the context of software systems. For example: Do 'God-classes' appear (significantly) more often with *Controller* role-stereotype than with *Service Provider* role-stereotypes? This will provide valuable insights that can guide software developers in making informed decisions during the design and refactoring activities.
- **RQ3:** *Does the type of application (desktop or mobile) influence the relation between design smells and role-stereotypes?* The focus is on understanding whether the nature of the application platform i.e. desktop or mobile, has an impact on the interplay between design smells and role-stereotypes within software systems. We want to understand the relationship and variations between design smells and role-stereotypes based on the software ecosystem?

4.2 | Data Collection

As shown in Figure 1, *Step 1* describes the data collection and selection strategy. *Step 2* focuses on the detection and preprocessing design smells from the selected projects. *In Step 3*, we preprocess and classify role stereotypes into 6 categories as previously discussed. In *Step 4*, we systematically integrate design smells and role-stereotype data to produce a fine-grained dataset. Using the fine-grained dataset, we perform analysis in *Step 5* and data clustering in *Step 6* to answer the aforementioned research questions.

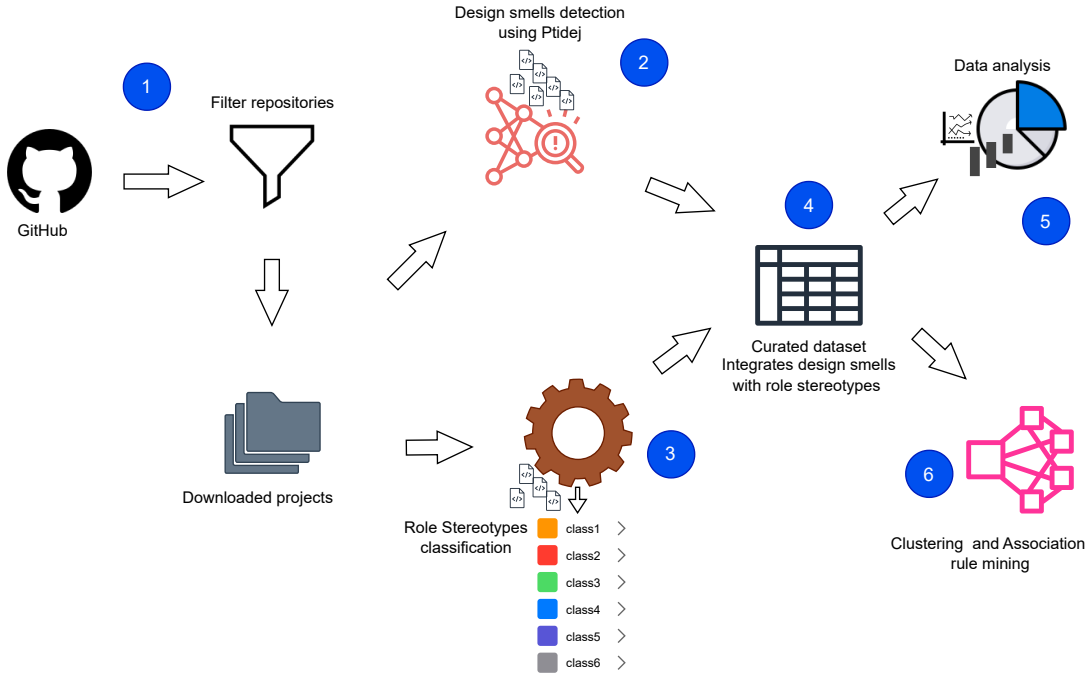

FIGURE 1 The research methodology pipeline. Step 1: data collection; Step 2: design smell detection; Step 3: role-stereotype classification; Step 4: integrate design smell and role-stereotype data to form a fine-grained dataset; Step 5 & Step 6: data analysis and clustering.

Step 1: Project Selection

In the first step, we built a dataset based on 30 active open source Java projects mined from GitHub containing a total of 11,350 number of classes (NOC) and 2,224,258 LOC (∼2,224 KLOC). The selected projects are grouped as either "desktop" or "mobile" software project. Tables 2 and 3 describes the characteristics of projects selected for this study including their domain, version LOC and NOC. The projects selected in this study are commonly studied by previous researchers and predominantly utilize Java as their primary programming language. The choice of Java is justified by two main factors: (1) Java is the only programming language that works well with our selected design smell detection tool mentioned in Section 2 and (2) Java is prevalent in a diverse array of object-oriented open-source projects across various code repositories, enabling the assessment of numerous design smell metrics. As mentioned in[38], Java has more design smell detectors than other languages. Furthermore, Java language is known to support the operation of billions of devices globally. Consequently, the chosen projects are cross-platform compatible (i.e., they can operate on Windows, macOS and Linux operation systems) and depend on Java core libraries for the design of their logical layers. Given these general characteristics, we curated a set of 30 projects from GitHub, provided that the project had at least two contributors and the latest commit was not more than one year. This was to ensure that the projects were still actively maintained (not "toy" projects) and modest in size. We selected GitHub because of its popularity as the largest social coding platform, hosting the development history of millions of collaborative software repositories[40], and offering diverse source code

metadata. The selection of Android-based mobile projects in this study allows for a valid comparison with desktop projects since Android source code is often written in Java or Kotlin. Furthermore, Android stands as the leading mobile operating system, commanding a market share of up to 74.6%[‡]. It also consists of a large community of developers delivering billions of apps on mobile software ecosystem.

TABLE 2 Characteristics of desktop projects analyzed in this study (LOC: Lines of Code; NOC: Number of Classes).

No.	Project	Description	Version	LOC	NOC
1	SweetHome3d	Interior Design Tool	5.6	104,059	546
2	Mars Simulation	Object Modeling	3.1.0	255,459	1109
3	ArgoUML	UML Modeling Tool	0.35.1	177,372	1236
4	JEdit	Text Editor	5.5.0	124,164	577
5	GanttProject	Project Scheduling and Management	2.9.11	66,709	671
6	GoGreen	Carbon Footprint Tracker	0.1.3	4,196	60
7	LiveChatServer	Live Chat Tool	1.4	2,796	23
8	Checkstyle	Java Source Code Validator	8.39	238,255	1008
9	Keystore-explorer	Secret Key Management	5.4.4	55,875	400
10	Angry IP Scanner	Network Scanner	3.7.3	12,701	159
11	JetUML	UML Modeling Tool	3.1	24,965	173
12	JPass	Password Manager	0.1.20	3,442	38
13	LogFX	Log Reader	0.9.1	4,582	44
14	PGP Tool	Easy PGP Decryption and encryption	0.5.9.2	18,673	226
15	Freemind	Mind-mapping Tool	1.0.0	67,287	370

TABLE 3 Characteristics of mobile projects analyzed in this study (LOC: Lines of Code; NOC: Number of Classes).

No.	Project	Description	Version	LOC	NOC
1	K9 Mail	Email Client App	5.600	93,540	779
2	Bitcoin Wallet	Bitcoin Payment App	6.31	18,079	222
3	KeepassDroid	Password Manager	2.5.9	17,916	211
4	Opentrip Planner	Trip Planning and Navigation	2.1.5	9,760	53
5	Tweet Lanes	Twitter Client App	1.4.1	541,694	130
6	Text Secure	Messaging App	4.69.5	25,886	1332
7	Telegram	Messaging App	4.1.1	166,731	679
8	Materialistic	News Reader App	3.3	21,919	131
9	Telecine	Full Resolution Video Recoder	1.6.2	1,410	23
10	AmazeFileManager	File Manager	3.5.2	46,135	265
11	Omni-Notes	Note Taking App	6.0.5	15,932	159
12	AntennaPod	Podcast Manager	2.1.1	53,769	387
13	GnuCash	Expense Tracker	2.4.1	27,837	147
14	Timber	Music Player	1.8	20,562	163
15	SeeWeather	Weather App	2.03	2,553	29

Step 2: Design smells detection

To detect design smells in source code, a number of tools are available. In this study, we used SAD which is part of *Pattern Trace Identification, Detection, and Enhancement in Java* (Ptidej) tool suite [§]. Ptidej is an open-source Java-based reverse engineering tool suite that includes several identification algorithms for idioms, micro-patterns, design patterns, and design defects[41]. This tool can detect 18 different types of design smells described in Table 4. Using Java classes as the unit of analysis, SAD detects occurrence of design smell using a set of rule cards. For example; "LongParameterList" can be detect using the following rule card:

[‡] https://gs.statcounter.com/os-market-share/mobile/worldwide
[§] https://github.com/ptidejteam/v5.2

TABLE 4 Description of the design smells detected using the Ptidej tool.

No.	Design Smell	Description
1	AntiSingleton	Provides mutable class variables, which consequently could be used as global variables.
2	BaseClassKnowsDerivedClass	A class that has many subclasses without being abstract.
3	BaseClassShouldBeAbstract	A class that has many subclasses without being abstract.
4	Blob	Large class declares many fields and methods with a low cohesion.
5	ClassDataShouldBePrivate	A class exposing its fields, violating the principle of data hiding.
6	ComplexClass	A class having at least one method having a high cyclomatic complexity.
7	FunctionalDecomposition	A main class with a procedural name in which inheritance and polymorphism are scarcely used.
8	LargeClass	A class that has grown too large in term of Lines of Code.
9	LazyClass	A class having very small dimension, few methods and low complexity.
10	LongMethod	A method that is unduly long in terms of lines of code.
11	LongParameterList	A method having a long list of parameters, some of which avoidable.
12	ManyFieldAttributesButNotComplex	Declares many attributes but which is not complex. Data class holding values without providing behaviour.
13	MessageChains	A long chain of method invocations performed to implement a class functionality.
14	RefusedParentBequest	A class redefining most of the inherited methods, thus signaling a wrong hierarchy.
15	SpaghettiCode	A class implementing complex methods interacting between them, with no parameters, using global variables.
16	SpeculativeGenerality	A class declared as abstract having very few children classes using its methods.
17	SwissArmyKnife	Complex class that offers a high number of services, such as implementing a high number of interfaces.
18	TraditionBreaker	A class that inherits from a large parent class but that provides little behaviour and without subclasses.

```
RULE_CARD : LongParameterList {
RULE : LongParameterListClass { (METRIC: NOParam, VERY_HIGH, 6) } ;
};
```

Using the rule card provided, the SAD tool performs a scan on Java classes, parsing their methods' signatures. For each method, it checks the number of parameter lists against the threshold specified by the *NOParam* metric (which is set to less than 6). If the count of parameters exceeds or equals 6 (*NOParam* >= 6), the tool identifies a "LongParameterList" occurrence within that class and saves the result in a ".ini" file. A sample output of this process is shown in Listing 2.

Design smells are detected and stored in "`.ini`" files. The file names are tagged with a specific design smell type. For example, in the K9 mail project, `LongParameterList` design smell is stored as `DetectionResults in K9 for LongParameterList.ini`. Our goal is to extract class names and the corresponding design smell detected in that class. Listing 2 shows a sample content of design smell detection file in its raw format. Based on the structure of the *".ini"*, we derived custom regular expressions to extract class names and their corresponding design smells. In the case of our example in Listing 2, we use this regex: `k9mail[a-zA-Z0-9.-]+`. For each design smell, we count the number of its occurrence in a given class, otherwise, the value `0` is assigned[42,43,44]. The summary of this process is illustrated as pseudo-code in Algorithm 1 and the actual implementation can be obtain from our replication package. Table 5 shows a sample output of processed design smells data.

TABLE 5 Sample output of processed design smells dataset. The columns indicate the design smell type and the row indicate Java classes.

index	FullClassPath	Classname	Blob	LongMethod	LazyClass	...
1	k9mail-library.src.main.java.com.fsck.k9.mail.AuthType.java	AuthType	3	1	0	...
2	k9mail-library.src.main.java.com.fsck.k9.mail.Address.java	Andress	1	0	0	...
3	k9mail-library.src.main.java.com.fsck.k9.mail.Body.java	Body	0	0	2	...
4	k9mail-library.src.main.java.com.fsck.k9.mail.Flag.java	Flag	1	0	0	...
5	k9mail-library.src.main.java.com.fsck.k9.mail.Folder.java	Folder	1	0	0	...
6	k9mail-library.src.main.java.com.fsck.k9.mail.K9MailLib.java	K9MailLib	0	3	0	...
7	k9mail-library.src.main.java.com.fsck.k9.mail.Message.java	Message	0	0	1	...
8	k9mail-library.src.main.java.com.fsck.k9.mail.Throttle.java	Throttle	1	2	0	...
9	k9mail-library.src.main.java.com.fsck.k9.mail.K9.java	K9	4	0	0	...
10	k9mail-library.src.main.java.com.fsck.k9.mail.MailService.java	MailService	1	2	0	...

Algorithm 1 Peusocode for Extracting Design Smells

```
procedure ExtractDesignSmells(iniFilePath)
    result ← ()
    for file ∈ iniFilePath do
        for smell ∈ smellList do
            fileContent ← readFileContent(file)
            matches ← findDesignSmell(fileContent, smell)
            if |matches| ≠ 0 then
                columnValue ← columnValue + 1
            else
                columnValue ← 0
            end if
            rowValue ← file
            result.add(rowValue, columnValue)
        end for
    end for
    return result
end procedure
```

Step 3: Role stereotypes classification

The processing of role-stereotype data was based on the replication package offered by Nurwidyantoro et al. [11,13]. First, the selected project source code is passed to srcML[¶], a lightweight, highly scalable, robust, multi-language parsing tool to convert source code into an XML format. Next, we built unlabeled data consisting of 21 features for each project. These classification feature are already published in [11], [13]. In this study, the feature extraction task was carried out as follows;

1. Create a srcML representation of the source code. The output of srcML tool is a list of source code classes in a standardized XML format.
2. We use multiple XPath queries to obtain the features of interest.

The detailed steps of the features extraction are elaborated in previous studies [11,13]. Finally, the unlabeled data was classified to one of the role-stereotype categories i.e. Service Provider, Information Holder, Interfacer, Controller, Coordinator and Structurer. The classification was achieved using the Random Forest classifier [11,13]. Table 6 shows the sample output of the processed role stereotype data.

TABLE 6 Showing sample of processed role-stereotype classification data.

No.	FullClassPath	Classname	loc	numAttr	...	label
1	k9mail-library.src.main.java.com.fsck.k9.mail.AuthType.java	AuthType	33	6	...	Service Provider
2	k9mail-library.src.main.java.com.fsck.k9.mail.Address.java	Andress	331	4	...	Interfacer
3	k9mail-library.src.main.java.com.fsck.k9.mail.Body.java	Body	24	0	...	Interfacer
4	k9mail-library.src.main.java.com.fsck.k9.mail.Flag.java	Flag	64	15	...	Information Holder
5	k9mail-library.src.main.java.com.fsck.k9.mail.Folder.java	Folder	208	5	...	Structurer
6	k9mail-library.src.main.java.com.fsck.k9.mail.K9MailLib.java	K9MailLib	67	7	...	Structurer
7	k9mail-library.src.main.java.com.fsck.k9.mail.Message.java	Message	237	4	...	Coordinator
8	k9mail-library.src.main.java.com.fsck.k9.mail.Throttle.java	Throttle	68	2	...	Controller
9	k9mail-library.src.main.java.com.fsck.k9.mail.K9.java	K9	43	0	...	Controller
10	k9mail-library.src.main.java.com.fsck.k9.mail.MailService.java	MailService	25	2	...	Controller

[¶] https://www.srcml.org/

TABLE 7 Showing a sample of final dataset after integrating design smell and role stereotypes data.

No.	FullClassPath	Classname	label	Blob	LongMethod	LazyClass	...
1	k9mail-library.src.main.java.com.fsck.k9.mail.AuthType.java	AuthType	Service Provider	3	1	0	...
2	k9mail-library.src.main.java.com.fsck.k9.mail.Address.java	Andress	Interfacer	1	0	0	...
3	k9mail-library.src.main.java.com.fsck.k9.mail.Body.java	Body	Interfacer	0	0	2	...
4	k9mail-library.src.main.java.com.fsck.k9.mail.Flag.java	Flag	Information Holder	1	0	0	...
5	k9mail-library.src.main.java.com.fsck.k9.mail.Folder.java	Folder	Structurer	1	0	0	...
6	k9mail-library.src.main.java.com.fsck.k9.mail.K9MailLib.java	K9MailLib	Structurer	0	3	0	...
7	k9mail-library.src.main.java.com.fsck.k9.mail.Message.java	Message	Coordinator	0	0	1	...
8	k9mail-library.src.main.java.com.fsck.k9.mail.Throttle.java	Throttle	Controller	1	2	0	...
9	k9mail-library.src.main.java.com.fsck.k9.mail.K9.java	K9	Controller	4	0	0	...
10	k9mail-library.src.main.java.com.fsck.k9.mail.MailService.java	MailService	Controller	1	2	0	...

Step 4: Data Integration

Our approach to construct a fine-grained dataset involved a systematic integration of the preprocessed design smells and role-stereotype data. This approach is adopted and extended from the work of Ogenrwot et al.[42]. It aims to combine information about design smells and role-stereotypes based on their common *classpaths*. The matching classpaths serve as key identifiers, ensuring that the relevant data from both datasets is correctly associated. It can be summarized as follows:

- The algorithm starts by initializing an empty result set that will store the integrated information.
- It then retrieves *classpaths* from both design smells and role stereotypes data, storing them in separate variables.
- The algorithm iterates over the combined set of classpaths from both datasets. For each iteration, it checks whether the *classpaths* from design smells and role-stereotypes match. If a match is found, it adds the matching *classpath* to the result set, along with the corresponding design smells data and role-stereotypes data. This process continues until all *classpaths* have been examined.
- Finally, the integrated result set is returned.

Table 7 shows an example output of the fine-grained dataset generated by the algorithm.

Algorithm 2 Integrate Design Semlls and Role Stereotype Data

```
procedure IntegrateData(ds_data, rs_data)
    result ← ()                                              ▷ Initialize an empty result set
    ds_classpaths ← ds_data.get(classpath)                   ▷ Get classpaths from design smells data
    rs_classpaths ← rs_data.get(classpath)                   ▷ Get classpaths from role-stereotypes data
    for (ds_classpath or rs_classpath) ∈ (ds_classpaths or rs_classpaths) do
        if ds_classpath = rs_classpath then                  ▷ Check if classpaths match
            result.add(classpath)                            ▷ Add the matching classpath to the result set
            result.add(ds_data[design_smells])               ▷ Add design smells data to the result set
            result.add(rs_data[label])                       ▷ Add role-stereotypes data to the result set
        end if
    end for
    return result                                            ▷ Return the integrated result set
end procedure
```

Step 5: Data analysis

Under the data analysis, the first step involved grouping the data of each selected project based on the role-stereotype classification. After which the percentages of classes with design smells and those without design smells was calculated from the total number

of classes as shown in Table A1 and Table A2. Similarly, for each role-stereotype the total number of design smells was calculated and this data was used to create the stacked-bar chart in Figure 5. The percentage of design smells in each role-stereotype in Figure 4 was obtained by dividing the total number of design smells in each role-stereotype by the total number of design smells found in the entire dataset. The selected projects were also grouped based on the application domain (desktop or mobile application). Table 12 shows the distribution of design smells in each role-stereotype for a given application domain. The data about role-stereotypes and design smells across mobile and desktop applications was used to generated Figure 6.

Step 6: Clustering and association rule mining

In this study, we leveraged the Powered Outer Probabilistic Clustering (POPC) algorithm[45]. The selection of this algorithm was driven by the following considerations: First, numerous clustering algorithms including the popular k-means algorithm, require the number of clusters to be specified in advance which is a huge drawback. Some studies use the silhouette coefficient, elbow method, and other approaches to determine the optimal number of clusters. However, those methods have their limitations, for example: sometimes the elbow method fails to give a clear "elbow point". Second, k-means is not very suitable for a binary or sparse matrix. Our dataset is quite sparse and since k-means depends on a distance measure (e.g euclidean), it becomes difficult to build cluster from a sparse matrix.

With POPC, there is no need to pre-define the number of clusters. The algorithm addresses this challenge through back-propagation i.e. start with many clusters and gradually optimize to obtain the optimal number of clusters. The algorithm is observed to work well on binary datasets and converges to the expected (optimal) number of clusters on theoretical examples as elaborated by Taraba[45]. The algorithm can be summarized as follows:

1. Use k-means clustering to assign each sample s_j to a cluster, denoted as $cl(s_j) = k$, where $k \in 1, ..., N$ and N is chosen to be half the number of data samples.
2. Calculate $J_{r=0}$ to assess the initial clustering, with r indicating the iteration of the algorithm.
3. Increment r to $r := r + 1$, initializing $J_r = J_{r-1}$.
4. For each sample s_j, attempt to assign it to all clusters to which it does not currently belong, denoted as $(cl(s_j) \neq k)$.
   a) If the temporary evaluation score $J_T > J_r$, then assign $J_r := J_T$ and move sample s_j to the new cluster.
5. If J_r is equal to J_{r-1}, then stop the algorithm. Otherwise go back to step 3.

TABLE 8 An example of a table constructed using Figure 7 to build Figure 8. The table shows for each cluster index (CID), the role-stereotypes and design smells present within that cluster. The presence or absence of a role-stereotype in a given cluster is represented with a value of 1 or 0 respectively. The role-stereotypes are abbreviated as follows; Coordinator (CO), Structurer (ST), Service Provider (SP), Information Holder (IH), Controller (CT) and Interfacer (IT).

	Role stereotypes						
CID	**CO**	**ST**	**CT**	**IH**	**IT**	**SP**	**Design Smells**
0	1	1	1	1	1	1	ClassDataShouldBePrivate, ComplexClass, LargeClass
1	0	0	0	1	1	1	LongMethod, ClassDataShouldBePrivate, ComplexClass, LazyClass
2	0	1	1	1	1	1	LongParameterList, LongMethod, ClassDataShouldBePrivate, AntiSingleton, ComplexClass
3	1	1	1	1	1	1	LongParameterList, ClassDataShouldBePrivate, ComplexClass
4	0	1	1	1	1	1	LongParameterList, ClassDataShouldBePrivate, AntiSingleton, ComplexClass
5	1	1	1	1	1	1	LongParameterList, LongMethod, ClassDataShouldBePrivate, Blob, AntiSingleton, ComplexClass
6	0	1	0	1	1	1	LongParameterList, BaseClassShouldBeAbstract, Blob, ComplexClass
7	0	1	0	1	1	1	ClassDataShouldBePrivate
8	0	0	0	1	1	1	ClassDataShouldBePrivate, ManyFieldAttributesButNotComplex
9	1	0	0	1	0	1	SpeculativeGenerality, ComplexClass

We divided the dataset in Table 5 into two groups i.e. desktop and mobile projects respectively for the task of clustering. This is because we want to determine whether there is any observable difference in the cluster formation across the desktop and mobile application. The output of clustering can be observed in Figure 7. To group role-stereotypes based on the design smells that occurred in them, clusters in Figure 7 were reconstructed to produce Figure 8 following the steps below:

(1) Data extracted from Figure 7 for both desktop and mobile projects were used to group clusters based on role-stereotypes.

   a. For each cluster index (CID) within a given role-stereotype, a value 1 was assigned to the role-stereotype's table cell, otherwise, 0 was assigned. Design smells identified in that CID were also recorded. The output of this operation is shown in Table 8.
   b. We repeated step (1a) for all the role-stereotypes.

(2) The role-stereotypes columns were extracted to create a binary matrix. This matrix is as an n-dimensional array passed to a dendrogram creation function[43]. Using the python Plotly package, hierarchical clustering is performed on data to produce the resulting tree shown in Figure 3.

In order to better understand the association of design smells with role-stereotypes, this study also explored an alternative approach to the clustering task. The study applied the well-known Apriori algorithms[46] to construct the association rules. Association rule discovery is an unsupervised learning technique used to detect local patterns which indicates attribute value conditions that occur together in a given dataset[47]. The association rule mining task was carried out as follows;- We defined a set of items $I = i_1, ..., i_n$ which is a binary set of n attributes (design smells) and a set of m transaction $T = t_1, ..., t_m$, which indicate all Java classes analyzed. An association rule $X => Y$ where $X \subseteq I$ and Y is a specific role-stereotype implies that a design smell (or group of design smells) X is associated with Y role-stereotype. We determine the strength of an association rule by its support 1, confidence 2 and lift 3 metrics. Apriori algorithm was applied with the minimum support value empirically set to 0.05.

$$support(X => Y) = P(X, Y) \tag{1}$$

$$confidence = \frac{support(X \cup Y)}{support(X)} \tag{2}$$

$$lift(X => Y) = \frac{support(X \cup Y)}{support(X) \times support(Y)} \tag{3}$$

5 | RESULTS

In this section, we present the results of our study. This is driven by answering the following research questions:

RQ1: How do design smells vary across mobile and desktop applications?

In our prior study[43], we focused on contrasting the prevalence of design smells between desktop and mobile applications. Specifically, we explored how the application type (desktop or mobile) influences the diversity, distribution, and magnitude of design smell occurrences. However, the study was constrained by a relatively small dataset, and the hypothesis testing was conducted considering the entire dataset without a more granular approach. In this study, we build on our prior work by investigating the density of design smells in mobile and desktop applications. This involves comparing the number of design smells per thousand lines of code (KLOC) in the selected projects. Notably, the average number of design smells per KLOC in desktop applications is relatively higher (37.3%) compared to the mobile applications (32.7%). Furthermore, we conducted Welch Two Sample t-test, to assess the statistical significance of these observed differences. The choice of the Welch Two Sample t-test is motivated by the noticeable variations in the sizes of the selected mobile and desktop applications. The result of the Welch's Two Sample t-test indicates that the aforementioned difference is not statistically significant with the t and p values of 0.535 and 0.597 respectively. We also conducted a Spearman rank correlation test and calculated the correlation coefficient (R^2) among design smells, as depicted in Figure 2. The results indicate a generally low correlation between design smells, with the highest observed value being 0.5, which pertains to the relationship between *AntiSingleton* and *ClassDataShouldBePrivate*. Using the POPC clustering algorithm, this study affirms theories pertaining to shared characteristics and similarities among design smells. The hierarchical clustering, as illustrated by the dendrogram shown in Figure 3, unveils the co-occurrence of

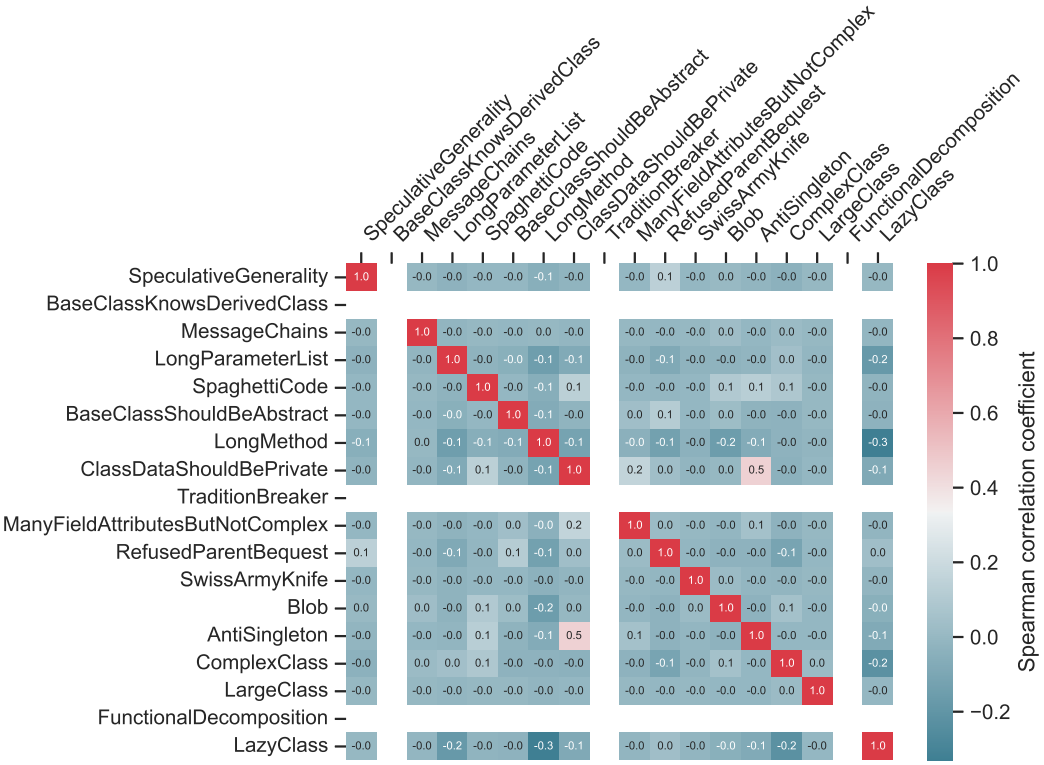

FIGURE 2 Correlation analysis illustrating the co-occurrence patterns of design smells.

design smells in both desktop and mobile applications. The results from Figure 3 are further elaborated in Table 10, presenting an insight into the co-occurrence patterns of design smells in desktop and mobile applications.

To compliment POPC clustering algorithm, the study explored an alternative approach based on association rule mining. Using the apriori algorithm, association rules were established to uncover relationships between design smells and their corresponding degrees of confidence, as detailed in Table 9. The hyper-parameter of the apriori algorithm was fine-tuned by setting the minimum support to 0.05. The results from the association rule mining, presented in Table 9, showcases 12 association rules ordered in descending order of confidence. Notably, the study observed a high-confidence association between *AntiSingleton* and *ClassDataShouldBePrivate* with a confidence level of 0.66. Conversely, the association of *LongMethod* with *LongParameterList* and *ComplexClass* exhibited the lowest confidence, registering at 0.10.

TABLE 9 The association of design smells with each other and their respective degrees of confidence.

No	Rule	Confidence
1.	(AntiSingleton) → (ClassDataShouldBePrivate)	0.66
2.	(LongMethod, LongParameterList) → (ComplexClass)	0.57
3.	(LongParameterList, ComplexClass) → (LongMethod)	0.50
4.	(ComplexClass) → (LongMethod)	0.46
5.	(LongParameterList) → (ComplexClass)	0.41
6.	(LongMethod) → (ComplexClass)	0.37
7.	(ClassDataShouldBePrivate) → (AntiSingleton)	0.49
8.	(LongMethod, ComplexClass) → (LongParameterList)	0.25
9.	(ComplexClass) → (LongParameterList)	0.24
10.	(LongParameterList) → (LongMethod, ComplexClass)	0.20
11.	(ComplexClass) → (LongMethod, LongParameterList)	0.12
12.	(LongMethod) → (LongParameterList, ComplexClass)	0.10

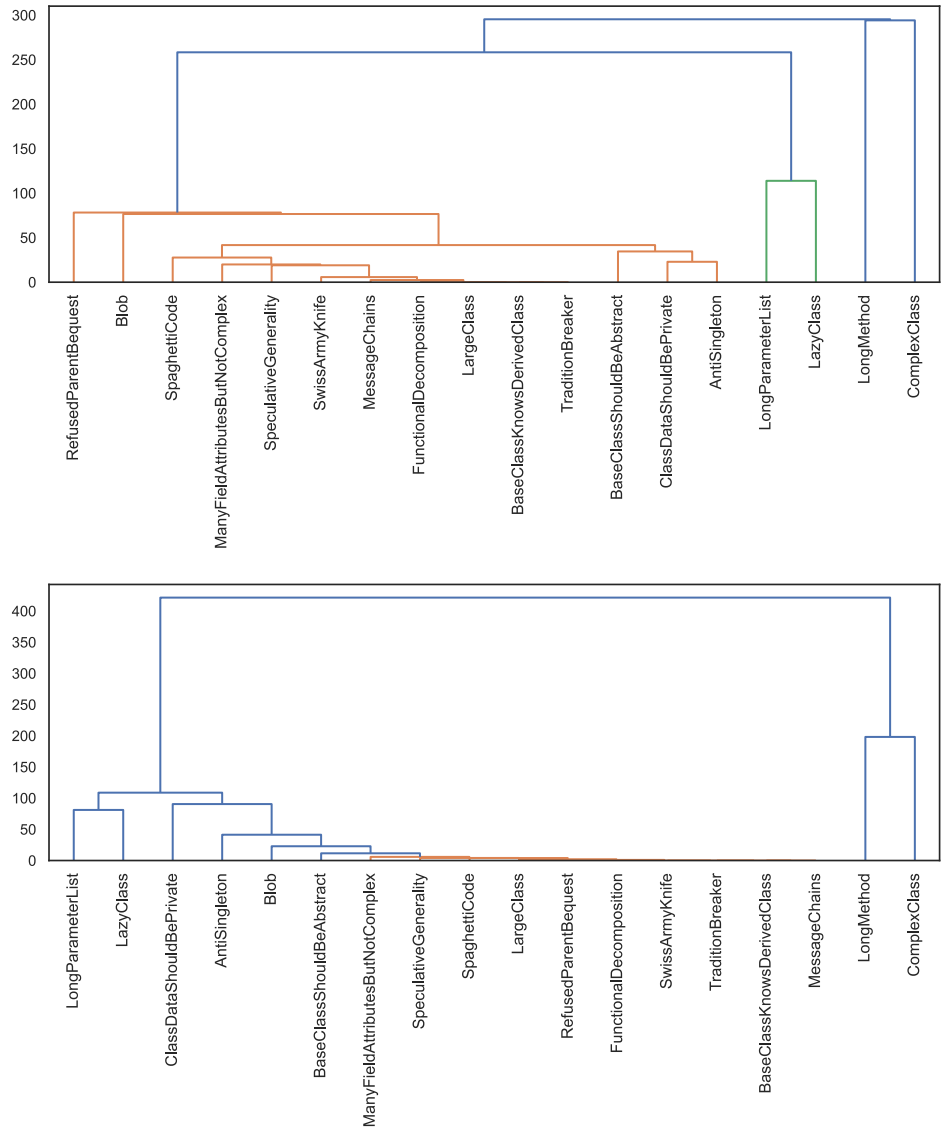

FIGURE 3 Hierarchical cluster visualization of design Smells that frequently co-occur in desktop (top plot) and mobile (bottom plot) applications.

Summary of Results for RQ1

1. On average, design smells are relatively higher in desktop applications than in mobile applications.
2. The average difference in the number of design smells in desktop and mobile applications is not statistically significant.
3. The hierarchical clustering and association rule mining reveals co-occurrence among design smells.

TABLE 10 The co-occurrence of design smells across the desktop and mobile applications extracted from Figure 3.

No	Co-occurrence Set in Desktop
1.	{BaseClassShouldBeAbstract, ManyFieldAttributeButNotComplex}
2.	{Speculative Generality, FunctionalDecomposition, LargeClass, BaseClassKnowsDerivedClass, TraditionalBreaker}
3.	{SwissArmyKnife, MessageChains, SpaghettiCode}
4.	{Blob, ComplexClass}
5.	{ClassDataShouldBePrivate, AntiSingleton}
6.	{LongParameterList, LongMethod}
No	**Co-occurrence Set in Mobile**
1.	{LongMethod, LazyClass}
2.	{BaseClassShouldBeAbstract, Blob}
3.	{FunctionalDecomposition, SwissArmyKnife, RefusedParentBequest, SpaghettiCode, BaseClassKnowsDerivedClass, MessageChains}
4.	{ManyFieldAttributesButNotComplex, LargeClass}
5.	{LongParameterList, AntiSingleton}
6.	{ClassDataShouldBePrivate, ComplexClass}

RQ2: How do design smells relate with different types of role-stereotypes?

In this second research question, the main objective was to understand the relationship between responsibility (role) stereotypes and design smells. To address this research question, Tables A1 and A2 provide insights into the percentages of classes containing design smells and those without, categorized by specific role stereotypes. Notably, the `Service Provider` responsibility stereotype consistently exhibits the highest percentage of classes with design smells in both desktop and mobile applications. This finding is also supported by our aggregated results in Figure 4, where we present the overall percentages of design smells within each role stereotype. Specifically, design smells tend to be more prevalent in `Service Provider` (52.4%), `Interfacer` (25.4%), and `Information Holder` (17.4%) compared to `Controller` (2.7%), `Structurer` (1.6%), and `Coordinator` (0.5%) role stereotypes.

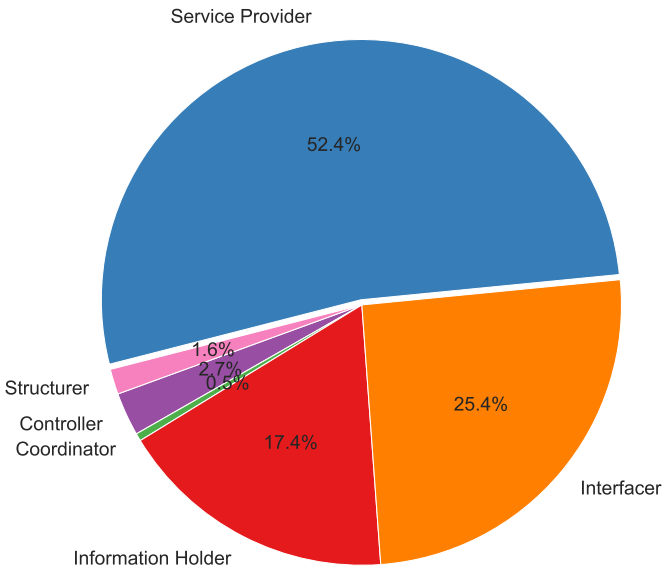

FIGURE 4 Distribution of design smells in each role-stereotype.

The second part of this research question explores the prevalence of specific design smells within each role-stereotype. Figure 5 illustrates (i) the occurrence of particular design smells in the respective role-stereotypes and (ii) the magnitude of their occurrence, depicted by the height of the stacked bar plot. Consistent with earlier observations, design smells are notably frequent

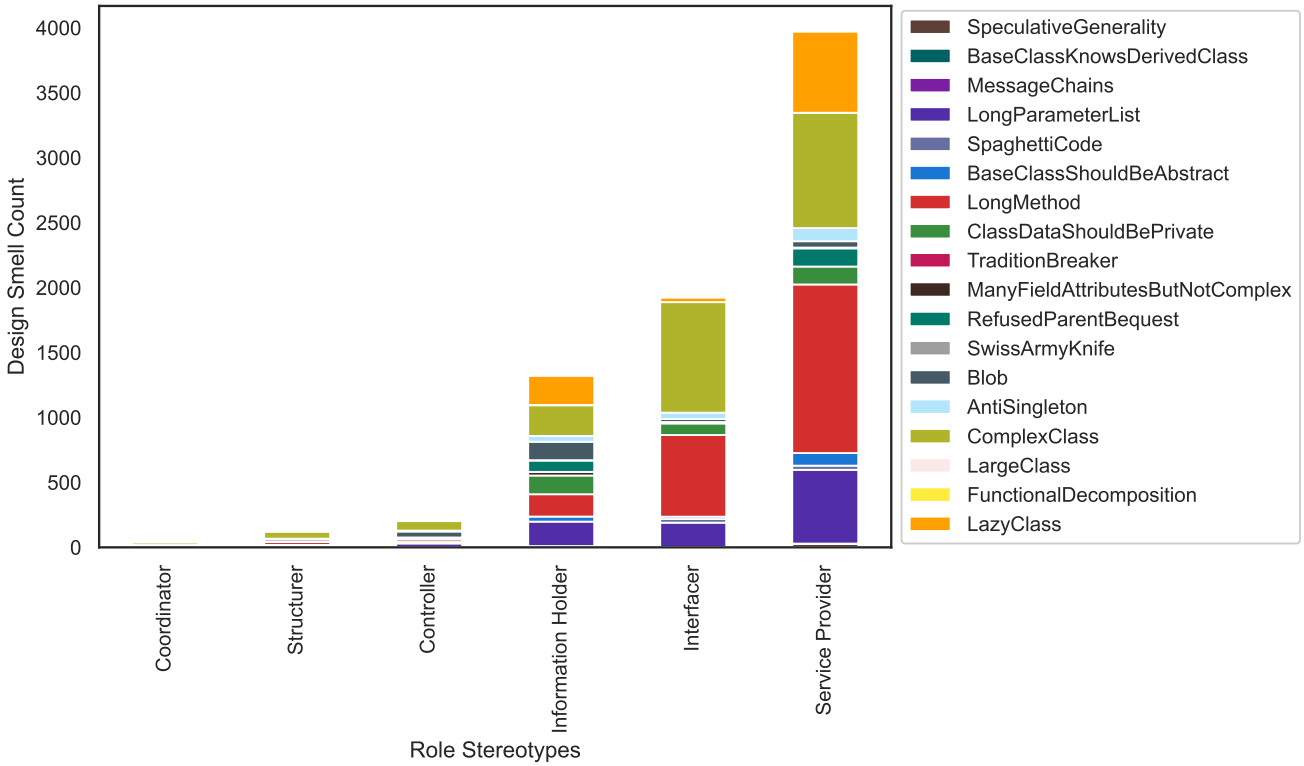

FIGURE 5 Stacked bar chart showing the distribution of design smells in each category role-stereotype. The bar size corresponds to the frequency of occurrence of a specific design smell.

in the `Service Provider`, `Interfacer`, and `Information Holder` role-stereotypes. Furthermore, `LongMethod`, `ComplexClass`, and `LongParameterList` exhibit the highest occurrence frequency across all role-stereotype categories, surpassing `LargeClass`, `MessageChains`, and `SpaghettiCode`. Additionally, it was noted that `Service Provider` and `Information Holder` are susceptible to a broader spectrum of design smells compared to other role-stereotypes, such as `Coordinator`, as depicted in Table 12.

TABLE 11 The association of various design smells with role-stereotypes with the respective degrees of confidence.

No	Rule	Confidence
1.	(ComplexClass) → (Service Provider)	0.72
2.	(ComplexClass) → (Interfacer)	0.63
3.	(LongMethod) → (Interfacer)	0.62
4.	(LongMethod) → (Service Provider)	0.60
5.	(LongMethod, LongParameterList) → (Service Provider)	0.58
6.	(LongParameterList) → (Service Provider)	0.57
7.	(LongMethod, ComplexClass) → (Service Provider)	0.49
8.	(LongParameterList, ComplexClass) → (Service Provider)	0.46
9.	(LazyClass) → (Service Provider)	0.45
10.	(LongMethod, ComplexClass) → (Interfacer)	0.40
11.	(ComplexClass) → (Interfacer)	0.38
12.	(ClassDataShouldBePrivate) → (Information Holder)	0.36
13.	(LongParameterList, ComplexClass) → (Interfacer)	0.36
14.	(LongParameterList) → (Information Holder)	0.26

Association Rule Mining: The results of association rule mining are presented in Table 11. Evidently, associations between design smells and specific role stereotypes are observable. These associations can be reasonably attributed to the inherent

characteristics and collaborative nature of role stereotypes. For instance, the characteristic of a Service Provider role-stereotype is performing a specific task and mostly implemented using "public static" methods. This implies that most implementations of a Service Provider class will be very small in dimension, few methods and low complexity. These properties are attributed to a LazyClass. Additionally, the study uncovers that design smells such as LongMethod, LongParameterList, and ComplexClass are commonly associated with multiple role stereotypes.

Summary of Results for RQ2

1. Design smells tend to occur more often in `Service Provider` (53.4%), `Interfacer` (26.7%) and *Information Holder* (15.1%) than in *Controller* (2.6%), *Structurer* (1.7%) and *Coordinator* (0.5%) role-stereotypes.
2. The study observed that *LongMethod*, *ComplexClass* and *LongParameterList* have the highest frequency of occurrence across the entire role-stereotype categories compared to *LargeClass*, *MessageChains* and *SpaghettiCode*.
3. There exist, association between design smells and some role-stereotypes. It is believed that these associations are attributed to the characteristics and collaborative nature of role-stereotypes.

RQ3: Does the type of application (desktop or mobile) influence the relation between design smells and role-stereotypes?

To address this question, we start with a comparison of the percentage of design smells in mobile and desktop applications within each role stereotype, as shown in Figure 6. This visualization not only underscores the higher prevalence of design smells in desktop applications but also identifies role stereotypes of particular interest. We observed design smells are more prevalent in `Service Provider`, `Information Holder`, `Controller`, and `Structurer` role stereotypes across both desktop and mobile applications. Conversely, `Interfacer` and `Coordinator` role stereotypes exhibit a higher occurrence of design smells in mobile than desktop applications. The detailed breakdown in Table 12 provides insights into the frequency and variety of design smells, highlighting that `Service Provider` and `Information Holder` role stereotypes exhibit the highest diversity of design smells. Additionally, specific design smells are observed to occur exclusively in particular role stereotypes and application types.

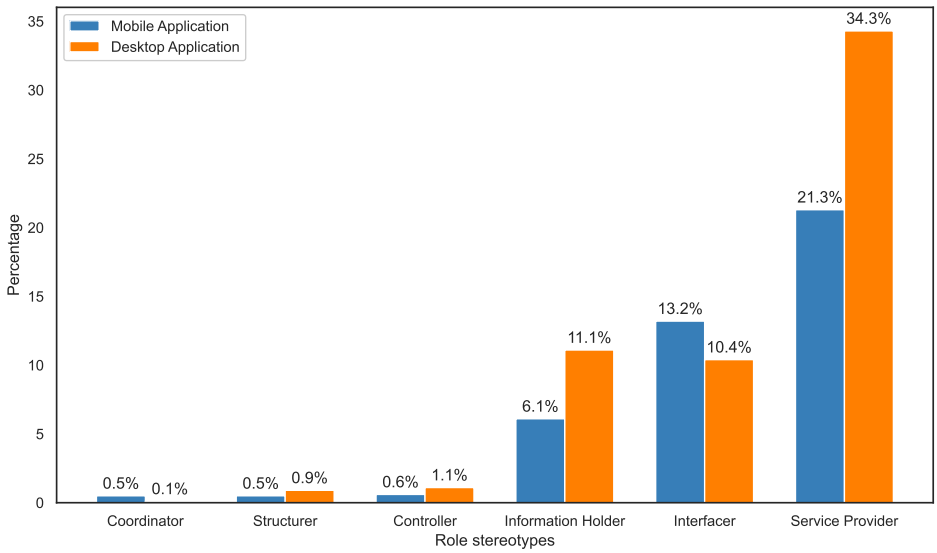

FIGURE 6 Percentage of design smells in mobile vs desktop application for each role-stereotype.

TABLE 12 Distribution of design smells in each role-stereotype across mobile and desktop applications. The highlighted cells indicate the presence of a particular design smell in a given role-stereotype and type of application. The role-stereotypes are abbreviated as follows: Coordinator (CO), Structurer (ST), Service Provider (SP), Information Holder (IH), Controller (CT) and Interfacer (IT).

	CO		ST		CT		IH		IT		SP	
Design Smells	**M**	**D**	**M**	**D**	**M**	**D**	**M**	**D**	**M**	**D**	**M**	**D**
AntiSingleton	-	-	✓	✓	✓	✓	✓	✓	✓	-	✓	✓
BaseClassKnowsDerivedClass	-	-	-	-	-	-	-	-	-	-	-	-
BaseClass ShouldBePrivate	-	-	✓	-	-	✓	✓	✓	✓	✓	✓	✓
Blob	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
ClassDataShouldBePrivate	-	-	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
ComplexClass	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
FunctionalDecomposition	-	-	-	-	-	-	-	-	-	-	-	-
LargeClass	-	-	-	-	-	-	✓	-	-	-	-	-
LazyClass	-	-	-	-	-	✓	✓	✓	✓	✓	✓	✓
LongMethod	-	-	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
LongParameterList	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
MessageChains	-	-	-	✓	-	-	-	-	-	✓	-	✓
ManyFieldAttributesButNotComplex	-	-	-	-	-	-	✓	✓	-	-	-	-
RefusedParentBequest	-	-	-	-	-	-	-	✓	-	✓	-	✓
SpaghettiCode	-	-	-	-	-	✓	-	-	✓	✓	-	✓
SpeculativeGenerality	✓	-	-	-	-	-	✓	✓	-	-	✓	✓
SwissArmyKnife	-	-	-	✓	-	-	-	✓	-	-	-	✓
TraditionBreaker	-	-	-	-	-	-	-	-	-	-	-	-

M: Mobile application.
D: Desktop application.

Finally, we studied the distribution of design smells using unsupervised learning. The objective was to observe the groups/pairs of role-stereotypes which exhibit similar type of design smells and to study whether this relation stretches across mobile and desktop application. Figure 8 indicates that the groups {`Coodinator, Structurer`}, {`Controller, nterfacer`} and {`Servicer Provider, Information Holder`} are quite similar in term of the design smells that occur within the scope of desktop application. The same figure also shows clusters of role-stereotypes: {`Interfacer, Service Provider, Information Holder`} and {`Coordinator, Controller, Structurer`} across mobile applications. We noticed one difference in these groupings across desktop and mobile applications. Specifically, the `Interfacer` role-stereotype has sightly different types of design smells in desktop applications as compared to mobile applications. The fact that interfacers might be designed somewhat differently in the mobile application also surfaced earlier where a hypothetical explanation is the different availability of libraries and frameworks in the mobile application. A similarity across mobile and desktop is that both `Service Provider` and `Information Holder` are in both cases grouped as 'similar' in terms of occurrences of design smells.

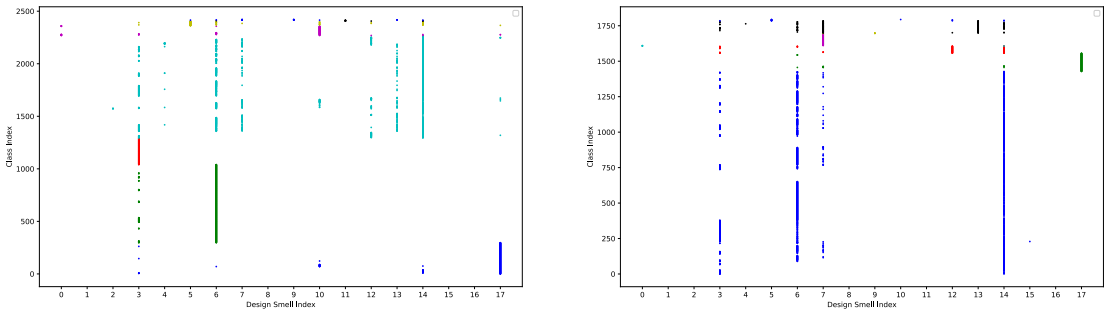

FIGURE 7 Results of clustering created with the POPC algorithm from desktop (left) and mobile (right) projects respectively. The vertical axis represents different samples (classes) belonging to different clusters (indicated in different colours) and the horizontal axis shows different features (design smells).

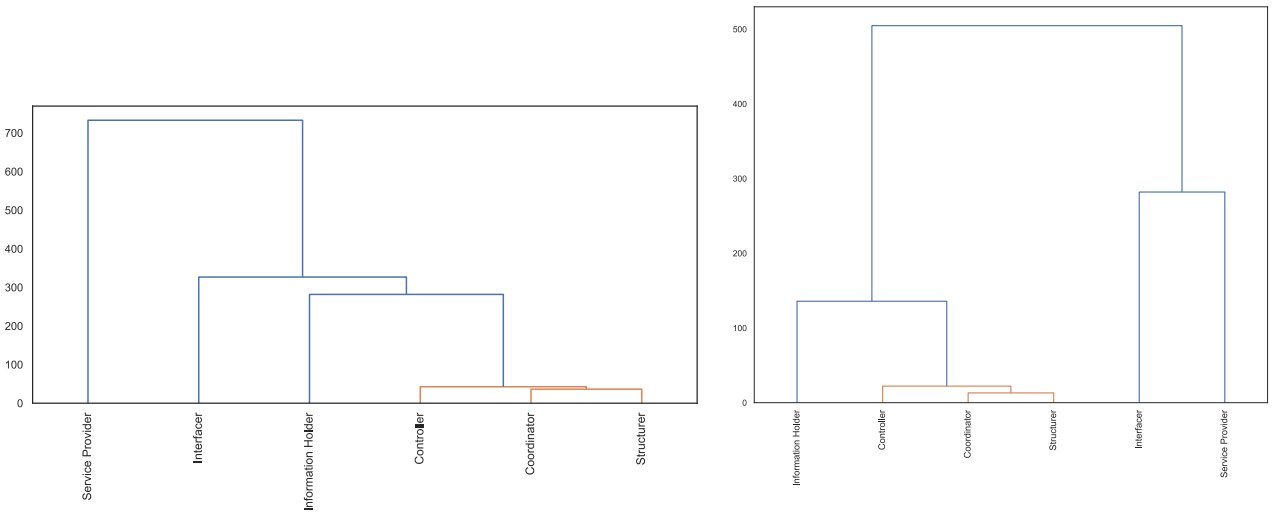

FIGURE 8 Hierarchical clusters represented by a dendrogram showing groups of role-stereotype in desktop (left) and mobile (right) applications with common type of design smells.

Association Rule Mining: The results of association rule mining are presented in Table 13. This finding is relatively consistent with the results of POPC algorithms reflected in Figure 8. Based on association rule mining, we can observe that there are generally strong associations of the different role stereotypes with each other across desktop and mobile applications. For example, the study noticed the following association rule with 100% confidence in desktop applications; {IT → SP}, {ST → SP} and {IT, CT → SP}. Other generated rules such as {CT → SP}, {IT → CT}, {IT, SP → CT}, {SP, CT → IT} and {IT → SP, CT} also indicates high association with confidence range of between 89% - 90%. Similarly, {SP → CT}, {IT → CT}, {IH → SP} and {IT, SP → CT} are observed as the strongest rules with 100% confidence in mobile applications as shown in Table 13. As previously stated, the result of the association is consistent with that of POPC algorithms represented in Figure 8. In Figure 8a (desktop application), we can observe a cluster {SP, IT} which coincides with the association rule {IT → SP} in Table 13.

TABLE 13 The association of the different role-stereotypes with each other across the desktop and mobile applications.

	Desktop		Mobile	
No.	**Rule**	**Confidence**	**Rule**	**Confidence**
1.	IT → SP	1.00	SP → CT	1.00
2.	ST → SP	1.00	IT → CT	1.00
3.	(IT, CT) → SP	1.00	IH → SP	1.00
4.	CT → SP	0.90	(IT, SP) → CT	1.00
5.	IT → CT	0.89	IT → SP	0.80
6.	(IT, SP) → CT	0.89	(IT, CT) ⇒ SP	0.80
7.	(SP, CT) → IT	0.89	(SP, CT) → IT	0.80
8.	IT → (SP, CT)	0.89	IT → (SP, CT)	0.80

Summary of Results for RQ3

1. `Service Provider` and `Information Holder` are prone to a wider range of design smells compared to other role-stereotypes such as `Coordinator` as shown in Table 12.

2. The `Interfacer` role-stereotype has sightly different types of design smells in desktop applications as compared to mobile applications. A hypothetical explanation is the availability of libraries and frameworks in the mobile mobile application development ecosystem.
3. Figure 8 indicates a similarity in the following groups of role-stereotypes; {`Coodinator`, `Structurer`}, {`Controller`, `Interfacer`} and {`Servicer Provider`, `Information Holder`}. The Association rule closely agree with the clustering result.

6 | DISCUSSIONS AND IMPLICATIONS

This section is divided into two subsections. In subsection 6.1, we discuss observations from our empirical study. The implications of this study are presented in subsection 6.2.

6.1 | Discussion of Results

RQ1: How do design smells vary across mobile and desktop applications?

In addressing this research question, we explored the variations in design smells between mobile and desktop applications through two distinct methods. Initially, we compared the number of design smells per thousand lines of code (KLOC) for each selected project. As outlined in the results section, the study notes that the average number of design smells is higher in desktop applications compared to mobile applications. However, this disparity does not reach statistical significance. This finding aligns with earlier research by Mannan et al[38]. It suggests that the prevalence of design smells in both desktop and mobile applications tends to be consistent, despite the distinct characteristics of these software ecosystems. Another reasonable explanation for this statistical result is the influence of developers' experience and coding styles, implying that less experienced developers are prone to introducing design smells in any project.

Design smells co-occurrences have been identified through two unsupervised learning techniques capable of discovering frequent relationships in a dataset i.e. clustering and association rule mining, as illustrated in Figure 3 and Table 9, respectively. The results not only affirm some anticipated relationships but also unveil co-occurrences overlooked by prior research in the field. The clustering outcomes presented in Figure 3 offer practical validation of theories pertaining to shared characteristics and similarities among design smells. For instance, through unsupervised learning, we demonstrated that `Speculative Generality` and `SwissArmKnife` are closely related. Nevertheless, we also found some unexpected relationships or similarities in the clusters which require further research to comprehend and provide recommendations. Therefore, we encourage researchers to explore this direction in future studies. Additionally, our study establishes a solid foundation for software educators to illustrate various design principles to students. This allows learners to practically observe examples of both well-designed and poorly designed systems across a diverse range of software systems.

RQ2: How do design smells relate with different types of role-stereotypes?

The findings in the results section outlined three major observations concerning this research question and this will inform our discussion as follows;-

As observed in Figure 4, we believe that this result is associated with the characteristics and collaborative properties of role-stereotypes. For example, a `Service Provider` is considered an object that performs specific work and offers services to other role-stereotypes on demand Wirfs-Brock[9]. A typical `Service Provider` class cache information and use it to improve performance or give clients more control over their operations Wirfs-Brock[9]. The process of implementing all those operational logic can increase the chance of introducing design smells in source code. Specifically, `LongMethod`, `ComplexClass`, `LongParameterList` and `LazyClass` are identified as the main contributors to high percentage of design smells in the `Service Provider` role-stereotype classes. A hypothetical explanation for the fact that `LongMethod` is largely associated with `Service Provider` is because services are based on long method definitions for API purposes.

The `Interfacer` role acts as a mediator to simplify communication with another system or subsystem. Specifically, It is responsible for handling and transforming requests and information between different parts of a system[9]. It is rare to have isolated code in today's enterprise software development. Software systems will often integrate with other systems such as payment APIs with an online shopping application, banking systems integrated with institutions' payment systems, or even subsystems interaction within your software. Based on the definition of a `Interfacer` role, systems integration is one of the most difficult tasks in software development. As a result, software engineers often resort to performing several "hacks" in order to realise the desired solution while leveraging design smells (e.g. `ComplexClass` and `LongParameterList` as shown in Figure 6) as a trade-off in the `Interfacer` classes.

We also noticed a relatively large distribution (15.1%) of design smells in the `Information Holder` role-stereotype classes. A basic example of Information Holder is the entity and value objects in a rich domain model. The relatively large amount of Information Holder across the selected projects can be related to the notion that an information holder may collaborate with service providers like data access classes or configuration classes to fetch more information on demand.

Overall, the selected projects have somewhat low distribution of design smells in the `Structurer`, `Controller` and `Coordinator` role-stereotypes compared to the previously discussed class roles. This observation can be explained three-fold. First, there are few numbers of `Structurer`, `Controller` and `Coordinator` role-stereotypes in the selected projects. This could imply that most software systems have few implementations of those class roles. However, more investigation should be conducted to support this claim. Secondly, it is likely that some class roles such as Service Provider are assuming too many responsibilities, in which case, a refactoring option should be explored. This could also be coupled with the nature of software systems, where, many functionalities are inclined towards services abstraction for API purpose, transforming information and requests between software layer and data encapsulation. Software engineers and educators should investigate and design role-stereotype based refactoring options. Third, the characteristic of some role-stereotypes safeguards it from design smells. For example, a `Coordinator` delegates work to other objects and it is not involved in a lot of decision making. Therefore, implementing a `Coordinator` object does not involve complex logic and this can limit the possibility of design smells occurrence.

RQ3: Does the type of application (desktop or mobile) influence the relation between design smells and role-stereotypes?

Although class responsibilities as categorized by Wirfs-Brock[9] are generic, there are some key notable differences that exist between mobile and desktop applications. For example, in desktop applications, the entry point is the `main` method - this can be considered as 'centalized flow of control'. In contrast, mobile (Android) applications do not depend on main methods but rather on event-handlers such as `onCreate`, `onResume`, etc. Hence, the overarching notion of behavioural interactions across components of mobile applications is 'event-driven'. Furthermore, desktop applications rely mostly on Java Swing library as their underlying Graphical User Interface (GUI) design library. However, this is not the case in mobile applications since there is a complete separation of the application logic from its presentation i.e. the GUI is mostly designed using eXtensible Markup Language (XML)[38]. Therefore, we believe that these differences could potentially influence the occurrence of design smells in various role-stereotypes.

The findings indicates that desktop applications are prone to design smells compared to mobile applications with the exception of the `interfacer` role-stereotype as shown in Figure 6. It is reasonable to believe that these variations are attributed to the underlying domain infrastructures - i.e. libraries and frameworks that are commonly used for developing applications. When developing mobile applications, certain libraries/frameworks already exist and the developer does not have to rewrite the same logic from scratch hence reducing the possibility of introducing more design smells. Also, the mobile (Android) ecosystem has attracted a large community of developers and code reviewers who ensure high-quality code before release. However, the high amount of design smells in the `Interfacer` role-stereotype of the mobile application is particularly interesting and merits further investigation. We think this can be explained in twofold; first, it indicates that a lot of functionality in mobile applications has to do with making API calls and requests between software layers. This is also supported by the fact that the Hardware Abstraction Layer (HAL) in Android OS provides standard interfaces that expose device hardware capabilities to the higher-level Java API framework[#]. Secondly, the large community of code reviewers and developers can also be disadvantageous. Palomba et al[48] observed that lack of communication or coordination between developers can cause community smells, making the code to

[#] https://developer.android.com/guide/platform

be less maintainable and worse over time. Overall, it is sensible to think that the variety and number of parts that a system has might increase the occurrence of Interfacer-based design smells.

6.2 | Research Implications

In this section, we present the implications of this study to software developers, tool creators and researchers.

To Software Developers: We discuss two fundamental implications for software developers in two main areas; (1) software design and development and (2) software quality assurance. The findings in the study provide important insights for effective software design and development. In essence, software system designers and developers can review class roles not only in term of their responsibility but also as an indicator of how vulnerability they are to certain group of design smells. This knowledge enables developers to pay extra attention to classes assigned with a certain responsibility. For example, Figure 4 and Figure 5 indicate that classes which are classified as `Servicer Provider` are often more prone to a wide range of design smells such as `LongMethod`, `LongParameterList` and `ComplexClass`. Therefore, software developers should incorporate this information during the design-, Quality Assurance-phases and maintenance activities. Understanding which design smells commonly occur in a specific role-stereotype can help developers to look out for such design smells and quickly resolve them to improve the overall software quality and maintainability.

As previously noted, our study revealed association of design smells with role-stereotypes. For example, we observed a high association of `LongMethod` with `Service Provider`. It seems likely that this association is inevitable or unavoidable, given the definition of `Service Provider`. Therefore, developers could ignore this design smell on condition that it is unavoidable and essential to their specific code. Moreover, not every smell is one a developer cares about or would fix. However, developers should be careful of `Speculative Generality` if the code is for a `Controller` role-stereotype.

Software quality assurance is an integral part of any software development and plays a significant role in ensuring that a software system conforms to a standard or predefined requirement. Good quality software eases maintenance and facilitates its evolution. Our results help develop or redefine already existing quality assurance guidelines and reduce the vulnerability of a system to design smells.

To Software Tool Builders: Our finding confirms a recommendation by Nurwidyantoro et al.[11]: it is likely to be beneficial to tailor metrics for detecting design smells to specific role-stereotypes. The variation in type and magnitude noticed in the different role-stereotypes as shown in Figure 5, Figure 6 and Table 12 is a clear indication to design smells detection tool creators that design smells metrics should not be applied uniformly across all classes irrespective of their roles. It would be better to optimize design smell metrics tailoring to a specific project when creating design smell detection tools. As observed by Kuzniarz et al.[19], role-stereotypes are helpful for understanding UML class diagram. We believe that using the results obtained from this study, tool builders can begin to explore ways of automatically inferring design smells from UML class diagram and role-stereotypes information. It may even be possible to come up with good recommendations for refactoring for specific combinations of role-stereotypes and design smells.

To Researchers: The findings in the study indicate that researchers can study design smells from a role-stereotype perspective. This insight is significant for the creation of new knowledge and research direction within the scope of design smells and role-stereotype concepts. For example, researchers could explore the possibility of reverse engineering role-stereotype classification techniques to utilize design smells information observed in a given class. Our results in Figure 5, Figure 8 and Table 12 already sheds light on some of those preliminary features. This also implies that researchers studying role-stereotype can easily benefit from data provided by design smells research experts and vice versa, hence creating synergy between researchers studying design smells and role-stereotypes respectively. From a teaching and learning perspective, our findings provide dependable literature for software educator to demonstrator various theoretical concepts related to role-stereotypes and design smells. As such, learners can easily observe those concepts across a variety of software projects for the distribution of design smells across role-stereotypes as shown in Table 12.

7 | THREATS TO VALIDITY

In this section, we discuss the threats to validity of our work, such as possible faults in the tools employed, the generalizability, and repeatability of the presented results.

Construct validity measures the degree to which tools and metrics measure the properties they are supposed to measure[49]. It aims to ensure that observations and inferences made are appropriate based on the measurements taken during the study. In the context of understanding the relationship between role-stereotypes and design smells, the `Ptidej` tool suite used in this study detects design smells based on some predefined set of metrics[43]. Relying on the outcome of this tool may pose a threat to validity since the metrics are static and assume that all design smells are equal in weight. However, we believe that the result of this tool is still valid for any static code analysis, which is the focus of our study. Another threat to construct validity is the use of role-stereotypes classification replication package shared by the authors[11]. This tool was trained and evaluated on only one project, i.e., K9-Mail (773 Java classes). To mitigate this risk and improve the role-stereotypes classification accuracy, we re-trained the classification model with a larger dataset of over 5,000 Java classes.

Internal validity: The study relied on two significant tools for building our fine-grained data, as previously mentioned. We used the Ptidej tool suite for the detection of design smells and the automated role-stereotypes classification replication package. We believe that the accuracy of our results also depends on the accuracy of those tools. In addition, to mitigate any multiplicity error that might result from the detection of both design smells and role-stereotypes, the study considered only classes with at least one type of design smell and participating in a role-stereotype. This is essential because for any given Java class, you might find a design smell but no/wrong role-stereotype, and vice versa, which can result in a large source of error and affect the overall accuracy.

External validity: This concerns the generalizability and repeatability of the produced results. This study was carried out on software systems (desktop and mobile) written in the Java programming language only. However, various platforms within the mobile and desktop ecosystem exist, which also utilize object-oriented paradigms, and it would be important to further explore the relationship within these systems. Therefore, based on the shared characteristics and program structure of any OOP system, we believe that the methods used in this study can be replicated to benefit other object-oriented-based software. Another external threat to validity is the issue of using a relatively modest-sized dataset consisting of 30 projects (11,350 Java classes). Although we believe that the size of the dataset was modest to answer the research questions and draw conclusions, one could argue that using a larger-sized dataset would give more confidence to the results presented in this paper. Nonetheless, to encourage replication and future extensions of this work, we have developed an open-source replication package (scripts and dataset) available online.

Conclusion validity assesses the degree to which the conclusions we reached about the relationships in our data are reasonable. A low number of samples, which reduces the ability to reveal patterns in the data, is observed as one of the common threats to this type of validity[50]. Therefore, we aimed at achieving statistical reliability as much as possible, given our sample data. As discussed earlier, the Welch's statistical test indicated that there is no significant difference between design smells in desktop and mobile applications in terms of the co-occurrence of design smells. The dataset used in this study consists of 11,350 Java class samples extracted from 30 OSS projects. Although we believe that the size of our dataset is considerable, using a larger-sized dataset would give more confidence to the results. Even so, our finding is still consistent with the work of previous authors[38].

8 | CONCLUSION AND FUTURE WORK

In this paper, we have presented an exploratory study to understand the relation between design smells and role-stereotypes and how this relation varies across desktop and mobile applications. We employed a number of statistical and unsupervised learning methods to report empirical evidence on the aforementioned relationships. Specifically, the study used two unsupervised learning methods i.e. clustering and association rule mining.

Our findings shows that design smells tend to occur more often in `Service Provider`, `Interfacer` and `Information Holder` than in `Controller`, `Structurer` and `Coordinator` role-stereotypes. In addition, we found that design smells are more frequent in desktop than mobile applications especially in `Service Provider` and `Information Holder` role-stereotypes. Using clustering, this paper revealed that the following pairs {`Coordinator`, `Structurer`},{`Controller`, `Interfacer`} and {`Service Provider`, `Information Holder`} are often quite similar in terms of the type of design smells that often occur in them and within the ambit of desktop applications. This is in comparison to {`Interfacer`, `Service Provider`, `Information Holder`} and {`Coordinator`, `Controller`, `Structurer`} role-stereotypes in mobile applications. The results of this paper can guide software teams in their efforts to implement various code/design smell prevention and correction mechanisms, as well as improve and maintain conceptual integrity of classes during their design and maintenance. Moreover, much as we observed that certain design smells are associated to

role-stereotypes (e.g. `LongMethod` and `Service Provider`), it could just as well be inevitable or unavoidable. In which case, we argue that a developer should ignore such design smells, since they could to some extent be unavoidable and essential to the code. Besides, not every design smell is one a developer cares about or would fix.

In the future, we hope to extend this study to include an analysis of the introduction and evolution of design smells relative to role-stereotypes - i.e. are there any patterns over time? Besides, we believe that methodologies for system/architecture-design can benefit by using the lens of design smells across role-stereotypes. Therefore, it would be interesting to observe how different software architectures (microservice vs. monolithic) influence the occurrence of design smells for specific role-stereotypes. Using the findings of this study, It would be great to develop different tools to support developers in their day-to-day activities of software design and maintenance.

AUTHOR CONTRIBUTIONS

Michel R.V. Chauudron: conceptualisation, supervision, review and editing manuscript. **Joyce Nakatumba-Nabende:** writing, review and editing manuscript. **John Businge:** review and editing manuscript. **Daniel Ogenrwot:** data collection, analysis and writing original draft.

DATA AVAILABILITY STATEMENT

The data and script that support the findings of this study are available in

CONFLICT OF INTEREST

The authors declare no potential conflict of interests.

REFERENCES

1. Imran A. Design Smell Detection and Analysis for Open Source Java Software. In: IEEE. 2019:644–648.
2. Imran A, Kosar T. Qualitative analysis of the relationship between design smells and software engineering challenges. In: 2022:48–55.
3. Jha S, Kumar R, Abdel-Basset M, et al. Deep learning approach for software maintainability metrics prediction. *Ieee Access.* 2019;7:61840–61855.
4. Palomba F, Bavota G, Di Penta M, Fasano F, Oliveto R, De Lucia A. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. *Empirical Software Engineering.* 2018;23(3):1188–1221.
5. Kaur S, Singh S. Influence of Anti-Patterns on Software Maintenance: A Review. *International Journal of Computer Applications.* 2015;975:8887.
6. Aversano L, Carpenito U, Iammarino M. An empirical study on the evolution of design smells. *Information.* 2020;11(7):348.
7. Turkistani B, Liu Y. Reducing the Large Class Code Smell by Applying Design Patterns. In: IEEE. 2019:590–595.
8. Mcilroy S, Ali N, Hassan AE. Fresh apps: an empirical study of frequently-updated mobile apps in the Google play store. *Empirical Software Engineering.* 2016;21(3):1346–1370.
9. Wirfs-Brock RJ. Characterizing classes. *IEEE software.* 2006;23(2):9–11.
10. Genero M, Cruz-Lemus JA, Caivano D, Abrahão S, Insfran E, Carsí JA. Does the use of stereotypes improve the comprehension of UML sequence diagrams?. In: 2008:300–302.
11. Nurwidyantoro A, Ho-Quang T, Chaudron MRV. Automated classification of class role-stereotypes via machine learning. In: , , , 2019:79–88.
12. Sharif B, Maletic JI. The effect of layout on the comprehension of UML class diagrams: A controlled experiment. In: IEEE. 2009:11–18.
13. Ho-Quang T, Nurwidyantoro A, Rukmono SA, Chaudron MR, Fröding F, Ngoc DN. Role stereotypes in software designs and their evolution. *Journal of Systems and Software.* 2022;189:111296.
14. Alhindawi N, Dragan N, Collard ML, Maletic JI. Improving feature location by enhancing source code with stereotypes. In: Ieee. 2013:300–309.
15. Barbez A, Khomh F, Guéhéneuc YG. A machine-learning based ensemble method for anti-patterns detection. *Journal of Systems and Software.* 2020;161:110486.
16. Kaur A, Singh S. Detecting Software Bad Smells from Software Design Patterns using Machine Learning Algorithms. *International Journal of Applied Engineering Research.* 2018;13(11):10005–10010.
17. Liu H, Xu Z, Zou Y. Deep learning based feature envy detection. In: 2018:385–396.
18. Dragan N, Collard ML, Maletic JI. Automatic identification of class stereotypes. In: IEEE. 2010:1–10.
19. Kuzniarz L, Staron M, Wohlin C. An empirical study on using stereotypes to improve understanding of UML models. In: IEEE. 2004:14–23.
20. Awad M, Khanna R. Machine learning and knowledge discovery. In: , , Springer, 2015:19–38.
21. Caram FL, Rodrigues BRDO, Campanelli AS, Parreiras FS. Machine learning techniques for code smells detection: a systematic mapping study. *International Journal of Software Engineering and Knowledge Engineering.* 2019;29(02):285–316.
22. Palomba F, Oliveto R, De Lucia A. Investigating code smell co-occurrences using association rule learning: A replicated study. In: IEEE. 2017:8–13.
23. Tahmid A, Nahar N, Sakib K. Understanding the evolution of code smells by observing code smell clusters. In: . 4. IEEE. 2016:8–11.
24. Runeson P, Host M, Rainer A, Regnell B. *Case study research in software engineering: Guidelines and examples.* John Wiley & Sons, 2012.
25. Vaucher S, Khomh F, Moha N, Guéhéneuc YG. Tracking design smells: Lessons from a study of god classes. In: IEEE. 2009:145–154.
26. Soh Z, Yamashita A, Khomh F, Guéhéneuc YG. Do code smells impact the effort of different maintenance programming activities?. In: . 1. IEEE. 2016:393–402.
27. Oizumi W, Sousa L, Oliveira A, et al. On the identification of design problems in stinky code: experiences and tool support. *Journal of the Brazilian Computer Society.* 2018;24(1):13.
28. AbuHassan A, Alshayeb M, Ghouti L. Software smell detection techniques: A systematic literature review. *Journal of Software: Evolution and Process.* 2021;33(3):e2320.
29. Fowler M. *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

30. Sharma T, Singh P, Spinellis D. An empirical investigation on the relationship between design and architecture smells. *Empirical Software Engineering.* 2020;25:4020–4068.
31. Bafandeh Mayvan B, Rasoolzadegan A, Javan Jafari A. Bad smell detection using quality metrics and refactoring opportunities. *Journal of Software: Evolution and Process.* 2020:e2255.
32. Saranya G, Nehemiah HK, Kannan A, Nithya V. Model level code smell detection using EGAPSO based on similarity measures. *Alexandria engineering journal.* 2018;57(3):1631–1642.
33. Moreno L, Marcus A. Jstereocode: automatically identifying method and class stereotypes in java code. In: 2012:358–361.
34. Jaafar F, Guéhéneuc YG, Hamel S, others . Analysing anti-patterns static relationships with design patterns. *Electronic Communications of the EASST.* 2014;59.
35. Walter B, Alkhaeir T. The relationship between design patterns and code smells: An exploratory study. *Information and Software Technology.* 2016;74:127–142.
36. Sousa BL, Bigonha MA, Ferreira KA. A systematic literature mapping on the relationship between design patterns and bad smells. In: 2018:1528–1535.
37. Santos JAM, Petronilo GXA. Building empirical knowledge on the relationship between code smells and design patterns: An exploratory study. *Journal of Software: Evolution and Process.* 2022;34(9):e2487.
38. Mannan UA, Ahmed I, Almurshed RAM, Dig D, Jensen C. Understanding code smells in android applications. In: IEEE. 2016:225–236.
39. Martins J, Bezerra C, Uchôa A, Garcia A. Are code smell co-occurrences harmful to internal quality attributes? a mixed-method study. In: 2020:52–61.
40. Decan A, Mens T, Mazrae PR, Golzadeh M. On the use of GitHub actions in software development repositories. In: IEEE. 2022:235–245.
41. Guéhéneuc YG. Ptidej: A flexible reverse engineering tool suite. In: IEEE. 2007:529–530.
42. Ogenrwot D, Nakatumba-Nabende J, Chaudron MR. Integration of design smells and role-stereotypes classification dataset. *Data in Brief.* 2021;36:107125.
43. Ogenrwot D, Nakatumba-Nabende J, Chaudron MRV. Comparison of Occurrence of Design Smells in Desktop and Mobile Applications. In: Bainomugisha E, Chaudron M, Hebig R., eds. *Proceedings of the 2020 African Conference on Software Engineering (ACSE 2020), Nairobi, Kenya, September 16-17, 2020*. 2689 of *CEUR Workshop Proceedings*. CEUR-WS.org 2020.
44. Ogenrwot D. *Using machine learning to understand the association of design smells with role-stereotypes in software systems*. PhD thesis. Makerere University, 2021.
45. Taraba P. Clustering for Binary Featured Datasets. In: Springer. 2017:127–142.
46. Agrawal R, Mannila H, Srikant R, Toivonen H, Verkamo AI, others . Fast discovery of association rules.. *Advances in knowledge discovery and data mining.* 1996;12(1):307–328.
47. Al-Maolegi M, Arkok B. An improved Apriori algorithm for association rules. *arXiv preprint arXiv:1403.3948.* 2014.
48. Palomba F, Tamburri DAA, Fontana FA, Oliveto R, Zaidman A, Serebrenik A. Beyond technical aspects: How do community smells influence the intensity of code smells?. *IEEE transactions on software engineering.* 2018.
49. Sharma T, Efstathiou V, Louridas P, Spinellis D. Code smell detection by deep direct-learning and transfer-learning. *Journal of Systems and Software.* 2021;176:110936.
50. Wohlin C, Runeson P, Höst M, Ohlsson MC, Regnell B, Wesslén A. *Experimentation in software engineering*. Springer Science & Business Media, 2012.

How to cite this article: Taylor M., Lauritzen P, Erath C, and Mittal R. On simplifying 'incremental remap'-based transport schemes. *J Comput Phys.* 2021;00(00):1–18.

APPENDIX

A ADDITIONAL INFORMATION

In this appendix, we provide additional information and results referenced in the main body of this study.

TABLE A1 Provides a summary of the percentages of classes in **desktop** applications categorized by role-stereotypes. It distinguishes between classes with design smells and those without design smells in relation to the total classes within the project. The highlighted cells draw attention to role-stereotypes exhibiting the highest percentage of classes with design smells for a specific project.

		SP		**CO**		**IH**		**IT**		**CT**		**ST**	
Projects	**NOC**	**a (%)**	**b (%)**	**a (%)**	**b (%)**	**a (%)**	**b (%)**	**a (%)**	**b (%)**	**a (%)**	**b (%)**	**a (%)**	**b (%)**
SweetHome3D	546	2.93	26.19	**5.31**	1.65	3.11	38.46	2.56	8.97	0.73	3.11	1.65	5.31
Mars Simulation	1109	**19.93**	23.81	0.45	-	7.75	23.81	12.35	9.47	-	0.09	0.90	1.44
ArogUML	1236	**28.07**	46.36	0.16	-	6.07	10.76	6.23	1.70	0.08	0.16	0.24	0.16
Edit	577	**46.45**	24.78	0.52	-	13.69	0.32	5.03	0.87	-	-	0.17	0.17
Gantt Project	671	**21.31**	46.05	0.15	-	6.71	14.90	7.60	1.04	-	0.45	1.64	0.15
GoGreen	60	**31.67**	35.00	-	1.67	15	15	-	-	-	1.67	-	-
LiveChat Server	23	**26.09**	21.74	-	-	21.74	13.04	17.39	-	-	-	-	-
Checkstyle	1008	**9.62**	46.33	-	-	7.14	34.42	-	1.79	-	0.10	-	0.60
Keystore explorer	400	**25.75**	29.75	-	-	5.50	20.00	14.00	4.25	-	0.75	-	-
Angry IP Scanner	159	**25.16**	49.06	-	-	5.03	11.95	7.55	0.63	-	-	0.63	-
jetUML	173	**38.73**	24.86	0.58	-	9.83	20.23	4.62	-	-	1.16	-	-
jPass	38	**13.16**	55.26	-	-	5.26	18.42	5.26	2.63	-	-	-	-
LogFX	44	**25**	27.7	-	-	6.82	25.00	9.09	6.82	-	-	-	-
PGP Tool	226	**12.83**	37.17	-	-	6.19	26.55	10.18	7.08	-	-	-	-
Freemind	370	**22.70**	45.95	1.89	0.54	7.30	10.27	6.49	2.97	-	-	1.89	-

a: Percentage of classes containing design smells
b: Percentage of classes that does not contain design smells

TABLE A2 Provides a summary of the percentages of classes in **mobile** applications categorized by role-stereotypes. It distinguishes between classes with design smells and those without design smells in relation to the total classes within the project. The highlighted cells draw attention to role-stereotypes exhibiting the highest percentage of classes with design smells for a specific project.

		SP		**CO**		**IH**		**IT**		**CT**		**ST**	
Projects	**NOC**	**a (%)**	**b (%)**	**a (%)**	**b (%)**	**a (%)**	**b (%)**	**a (%)**	**b (%)**	**a (%)**	**b (%)**	**a (%)**	**b (%)**
K9 Mail	779	4.36	37.10	0.90	1.67	**4.62**	25.03	2.57	7.32	2.05	8.09	1.41	4.88
Bitcoin Wallet	222	2.70	22.97	0.90	1.35	4.05	33.33	**6.31**	21.62	-	0.90	1.8	4.05
Keepassdroid	211	**29.86**	39.34	-	-	9.95	14.22	6.16	0.47	-	-	-	-
OpentripPlanner	53	**24.53**	43.40	-	-	3.77	18.87	5.66	3.77	-	-	-	-
Tweet Lanes	130	**22.31**	33.08	-	-	8.46	15.38	15.38	4.62	-	-	-	-
Text Secure	1332	**23.87**	42.64	0.15	-	3.83	11.41	12.24	4.80	0.08	0.68	0.30	-
Telegram	679	19.15	20.91	2.21	0.29	6.33	9.13	**25.04**	16.05	-	0.15	0.44	0.29
Materialistic	131	**25.19**	42.75	-	-	-	15.27	12.21	3.05	1.53	-	-	-
Telecine	23	**26.09**	26.09	-	-	-	30.43	13.04	4.35	-	-	-	-
Amaze File Manager	265	**23.02**	38.87	-	-	6.79	11.32	16.60	3.40	-	-	-	-
Omni-Notes	159	**22.64**	45.91	-	0.63	8.81	10.69	8.81	2.52	-	-	-	-
AntennaPod	387	**18.60**	43.93	0.26	-	3.62	16.02	10.85	6.20	-	0.52	-	-
GnuCash	147	**30.61**	35.37	-	-	5.44	8.16	15.65	4.76	-	-	-	-
Timber	123	**24.54**	36.20	-	-	11.66	9.82	- 11.66	4.29	-	1.23	-	0.61
SeeWeather	29	**31.03**	31.03	-	-	24.14	3.45	10.34	-	-	-	-	-

a: Percentage of classes containing design smells
b: Percentage of classes that does not contain design smells

TABLE A3 Detailed description of metrics used in the Software Architectural Defect (SAD) tool for the detection of design smells.

No.	Metric	Description
1.	DIT	Depth of inheritance tree of an entity. Uses a recursive algorithm to calculate it.
2.	IR	Inheritance ratio.
3.	LCOM5	Lack of cohesion in methods of an entity.
4.	McCabe	McCabe Complexity: Number of points of decision + 1
5.	MultipleInterface	If a class implements more than one interface.
6.	NAD	Number of attributes declared by an entity.
7.	NMD	Number of methods declared by an entity.
8.	NOC	Number of children of an entity.
9.	NOParam	Maximum number of parameters of the methods of an entity.
10.	NOTI	Highest number of transitive invocations among methods of a class. See the Law of Demeter for a definition.
11.	USELESS	Same value for any entity: 1.
12.	WMC	Weight of an entity as the number of method invocations in each method. (Default constructors are considered even if not explicitly declared).