Learning Patterns of Mobile Interface Design

George C. GUVERNATOR V
The College of William and Mary
Williamsburg, Virginia
gcguvernator@email.wm.edu

***Abstract*—Nothing for now. We'll write our abstract last.**

I. INTRODUCTION

Interface design is a crucial element in any software project. Graphical interfaces allow for more intuitive human-computer interaction but can challenge even the most skilled developers as they take on the additional responsibilities of a designer. In the world of mobile applications, where many competing implementations of an idea are available to users, design can play a key role in a user's choice of application. Additionally, limited and varying screen sizes, touch-based interfaces, and limited resources all challenge the mobile interface designer. It is our observation that, unfortunately, some developers fail to spend ample time designing Graphical User Interfaces (GUIs). This is especially true in academia, where many of the most technically correct and well-implemented software projects falter in this area, considering GUI design an overly costly afterthought. In mobile applications, this can mean that not all screen sizes, input methods, accessibility features, and other mobile-exclusive considerations are accounted for. As a result, developers lose many potential users from otherwise well-written and well-executed projects.

Mobile application development presents unique challenges beyond those faced when developing software with more traditional keyboard and mouse interfaces. On mobile platforms, user interface design poses the unique challenge of restricted screen space with respect to more conventional desktop or console platforms and therefore has a greater influence on the overall usability of the application. Additionally, developers must keep in mind the variety of devices of different size that their applications will run on. An interface built for a 15 by 10 cm tablet, for example, may not scale well to a 9 by 5 cm smartphone. Elements designed to be tapped on the larger screen by human fingers or styli would become more difficult to accurately activate on the smaller screen. Another complication to developers are extensions to the platform's standard user interface, such as those used for accessibility or universal access by users with physical disabilities. Finally, fragmentation on the Android platform causes design and development bugs, both from devices running different versions of the Android platform which support different graphical layout components, and from a variety of vendors building their devices differently. The problem of Android device and version fragmentation are discussed by Han et. al. [1] and Degusta [2].

This research funded by a grant from the National Science Foundation (1359275).

Facing these additional challenges of mobile development, it is therefore important to understand both how mobile applications are designed and which identifiable design patterns users prefer. The former can be accomplished given access to an application's source code, and the latter is theoretically possible by scraping user ratings (both long-form text reviews and one- to five-star numeric ratings) from Google Play, the officially supported Android application repository. Given the unique challenges of the mobile environment discussed above, we assert that design is a significant factor reviewers consider when rating an application. While many reviews center around program functionality and stability, we believe there is enough weight placed on design to show clear trends and allow for correlative measurement.

In this research, we analyze the GUI design of a variety of Android applications, allowing us to gather data and create a characteristic model. We evaluate correlations between the gathered design data and reported user experiences, such as comments and ratings, as well as other potentially confounding variables found in the application's metadata from the application repository.

To accomplish this, we scrape Android source code from the F-Droid Web repository[1] using a tool we developed called *fdscrape*.[2] Package names found with the source code on F-Droid are also searched on Google Play, a non-free Android application repository, and metadata such as user ratings, comments, popularity, and size are scraped and associated with each application. The combined source code and metadata, collectively the *F-Droid corpus*, is run through a program we have developed called AGUILLE.[3] This tool analyzes the structure of GUI markup language in the source code and extracts and counts the individual elements used to construct the interface, analyzing and combining data points to prepare for machine learning analysis.

Finally, the analysis of AGUILLE and metadata found with fdscrape are combined in a machine learning workflow in Weka. The workflow leverages the power of the M5 model tree to generate explainable branches and decision points that are applied in an ordered hierarchical structure to determine a potential rating for future applications. This is improved by analyzing each application category separately and building separate models for those categories with enough applications to make valid predictions. This categorical discretization is a

[1] F-Droid can be accessed on the Web at `https://f-droid.org/`.

[2] Fdscrape is licensed under the GNU General Public License (version 3) and is available on the Web at `https://github.com/qguv/fdscrape`.

[3] AGUILLE is licensed by the GNU General Public License (Version 3) and is available on the Web at `https://github.com/qguv/aguille`.

key part of our experiment, see section IV on the following page.

A summary of specific results should go here. It will mirror the summary that will end up in the conclusion.

Further development could turn the predictive model into a suggestive one. The models generated by the framework described in this research could be a crucial addition to "Interface Builders," [3] graphical applications designed to help developers create graphical interfaces. Developers would have a new, powerful tool suggesting subtle changes to their design in order to better emulate the most popular and successful graphical interfaces available today.

The main contributions of this research are as follows:

- Development of a framework of tools to gather layout information of Android applications (section III)
- An empirical study correlating Android GUI design patterns and the reported quality user experiences (section IV on the following page)
- Analysis of recurring design patterns and trends in Android GUI design (section V on page 4)
- Development of a predictive model for mobile GUI design (section IV-B on page 4)

II. Background

This section will be expanded to better explain where our research fits in the field of related work discussed in section VI on page 5.

Research on learning design patterns [4], [5] proves useful when designing a predictive machine learning workflow. Both Neural Networks and vanilla Decision Trees are discussed and implemented in [5].

We began by correlating specific features and ratings manually using straightforward linear regression in order to test different weightings of features. Next, random forests were used to gain insight on the sorts of decision points that regression and model trees produce. We eventually settled on to the M5 model tree to firm up final results and to allow trends to be explained and described in human-friendly decision points rather than difficult-to-describe coëfficient models or more opaque random models.

A. Choosing Android

Though the concepts presented in this paper are applicable to any graphical environment, we have chosen to work with the Android mobile platform due to both the unique challenges of a mobile environment discussed in section I on the preceding page and the uniformity and availability of application source code.

We feel the concepts presented in this paper would be most advantageous to mobile developers, as user ratings, our evaluative metric, can directly influence an application's ultimate success or failure. Android users must often choose between similar implementations of the same tool. The Google Play store in turn provides a system with which users can rate applications and post feedback for developers and other potential users. These user ratings help Android users to narrow down their choices in a vastly competitive market.

Android is also ubiquitous among mobile device users. The popularity of the platform continues to grow as new users and developers adopt Android as their primary mobile platform. With over 1.3 million[4] Android applications on the Google Play store at time of writing, the popularity of the Android platform has provided us and will continue to provide other research teams with ample data to search for significant correlations and generalizations.

Finally, the somewhat constrained GUI design framework in the Android platform (Android XML) allows for relatively straightforward parsing of the application's graphical layout without needing to peek into the application's logic.

Because of these unique properties of the Android platform, we believe mobile applications will benefit most from the preliminary results presented in this paper.

III. Extracting Design Elements

Before we can begin evaluating and correlating patterns, we must first collect information on Android GUI design. Since Android developers define graphical layouts in source code, we decided to gather and interpret source code in order to gather data about Android GUIs. We chose the free and open-source Android software repository F-Droid as a source for Android source code.

After gathering source data, we must extract the parts of the source code pertaining to graphical layouts. We then analyze those layouts to determine what built-in graphical elements the developer chose to use in designing the application and in what quantity and proportion.

After all layouts of all available applications in the repository have been analyzed, the results are fed to a machine learning algorithm to make generalizations about which elements affect others, which best predict ratings, and which carry little meaning in the context of this study.

Finally, the performance of this machine learning system is analyzed, and the workflow is tweaked to attempt to improve prediction and correlation both between elements and against user ratings.

A. Dataset Acquisition with fdscrape

We have developed a program (in Python) to enable mass retrieval of Android source code to mine. We use F-Droid, a software repository containing binaries and source code for 1,145 free and open-source Android applications. The majority of these applications are also available on the official Android application repository, the Google Play store.[5] Because of this, we have downloaded all available applications and their source code from F-Droid as well as Google Play ratings and metadata for the same applications, storing the data for analysis in the machine learning step.[6] This data is stored with the source code of each application.

[4] According to *Appbrain Stats*, a Google Play metrics service. Visit `http://www.appbrain.com/stats/number-of-android-apps` for the latest statistic.

[5] The Google Play store can be accessed on the Web at `https://play.google.com/store`.

[6] To accomplish this, we have cross-checked Java package names against both F-Droid and the Google Play store.

We originally chose to scrape only rating information from Google Play. It was decided, however, that by saving more of the metadata provided by Google Play and developers, better predictions could be made by accounting for variables beyond the scope of design. This permits meta-analysis of our hypothesis, i.e. we can decide how much design affects ratings and how much predictive accuracy to expect from our model. We gather this data in order to compensate for any confounding correlation that Google Play metadata may have on determining rating.

Specifically, the developer-chosen application category (e.g. Weather Application, Productivity Application, Action Game, Puzzle Game, and others in table I) provides an effective way to analyze groups of applications at a time. It is our hypothesis in RQ 1 that separate analysis within application categories will yield more meaningful results. This has the potential to greatly improve the accuracy of our predictive algorithm and allows us to better understand what "good design" entails in certain domains. For example, the same elements that constitute good design for an action game might exemplify bad design for a news application.

After omitting applications that were not on the Google Play store, had no ratings, or did not host source on the main F-Droid website, we collected the source code of 894 applications to build our dataset.

B. Tag Lexing & Extraction with AGUILLE

We have developed AGUILLE, the Android Graphical User Interface Lazy LExer, to perform the Android source analysis we originally hoped GUITAR would accomplish. The tool takes in an application's source code, finds the relevant Android XML structure, and parses that structure into native Python objects. Using these objects, AGUILLE calculates the frequency with which each XML tag, or element, occurs in the application. The graphical design of the application, therefore, is reflected in the developer's choice of graphical elements.

These frequencies are collected in a CSV file, along with the metadata gathered with fdscrape. Lots of the scraped information can be cached to speed up parses of entire repositories.

The tool is designed such that, should more sophisticated calculations prove necessary, separate sub-commands could easily be added. AGUILLE is open-source; anyone may extend it by adding further subcommands or modifying its current behavior.

C. Machine Learning with Weka

We make use of the Weka 3.7 *Knowledge Flow* environment to create a machine learning workflow. Data from AGUILLE is loaded separately by category. We drop categories which contain less than one percent of all applications mined, leaving the 20 categories described in table I.

	Applications	
Category Name	Number	Percent
'Tools'	278	33.3%
'Productivity'	88	10.5%
'Communication'	67	8.0%
'Personalization'	34	4.1%
'Books and Reference'	32	3.8%
'Game Puzzle'	30	3.6%
'Education'	29	3.5%
'Media and Video'	29	3.5%
'Music and Audio'	25	3.0%
'Entertainment'	24	2.9%
'Transportation'	18	2.2%
'Travel and Local'	18	2.2%
'Finance'	17	2.0%
'Game Arcade'	17	2.0%
'Health and Fitness'	17	2.0%
'Lifestyle'	15	1.8%
'News and Magazines'	15	1.8%
'Social'	15	1.8%
'Photography'	13	1.6%
'Libraries and Demo'	11	1.3%
Total: 20 categories	792	94.7%

TABLE I. APPLICATIONS IN EACH MINED CATEGORY

IV. EMPIRICAL EVALUATION

The design of our experiment is such that two chief research questions (RQs) may be addressed:

1) What sort of design do applications have in common? What sort of trends emerge when analyzing entire repositories of applications?
2) Does separate analysis of applications by Google Play category improve the quality of our predictive algorithm? Does such separation give more meaningful generalizations when explaining the output of our decision tree?

Our goal in evaluating the accuracy and quality of our predictions is twofold: to attempt to improve the ability of our algorithm to predict user ratings and to make conclusions how about individual elements affect user perception of an application.

When evaluating the performance of our machine learning workflow, we are looking for statistically significant correlations with performance better than the 5–10% correlation we see with naïve sample algorithms.

We cannot expect anything near perfect prediction, as more goes into a user's choice of rating than design. We must therefore focus on explainable output, such as that from a decision tree, to try to explain the influence of design on ratings.

A. Experiment Design

To be able to learn which design elements lead to the best applications, we need a group of factors, or *heuristic*,

to evaluate, as well as a metric for determining what constitutes a "good" application. This study uses the frequency of use of Android XML tags in graphical layout source code as a heuristic. Because the Android framework comes with a rigidly-defined set of elements, XML tag (and therefore element) frequency allows us to point to very specific design choices to explain findings when learning correlations.

Android also allows developers to define their own tag elements, but because scraping these developer-defined, application-specific tags and properties involves parsing Java logic and rendering the elements, this added complexity is somewhat beyond the scope of this project. The application-specific nature of these tags also means that they will most likely not be of use when attempting to make correlations between applications.

Although it may be possible to learn a more complex heuristic over time, it would increase overhead and likely introduce excessive complexity into our system. At this stage of research, it is wiser to rely on the pre-designed elements available to all Android applications to potentially determine the quality of GUI design. We have found that element frequency is sufficient to establish a statistically significant correlation between the elements themselves and the evaluative metric, Google Play ratings.

Potential alternative evaluative metrics could include uninstall rate and frequency, certain statistical functions on the cumulative body of Google Play user ratings, or cross-referenced reviews from established news sources. Google Play ratings were trivial to scrape and analyze, as the data is publicly available, so these public ratings serve as an initial metric we use to establish the overall quality of an application. In our algorithm, we can weigh the rating's relevance depending on how many users rated the application, a metric we also obtained from the Google Play store. We might be more confident in a metric to which many users contributed.

In early models, we found that the M5 model tree first branched based on the amount of user ratings on Google Play. The sub-trees after this split did not clearly resemble each other; the branch reached by applications with few ratings gave a counter-intuitive model, while the branch for applications with a more substantial amount of ratings weighted elements as we would expect. This suggests that the model's predictive accuracy increases substantially when more rating data is available, as we would expect.

Our group of independent variables, the frequency of each Android graphical element, will reflect the different proportions of interactive Android *widgets* with respect to each other. These interactive widgets are built-in to the Android platform and include buttons, check-boxes, radio buttons, images, and text. Additionally, these widgets can appear in different *views*, all of which have different ways to specify how the widgets will be laid out on the screen. All of these views and widgets are part of our element count.

Of course, GUI design is hardly the only factor users consider when rating a program. It is important to consider major confounding variables (viz. quality of functionality, stability, the ability of the program to solve a real problem) and integrate Google Play's qualitative long-form paragraph review system. A naïve but effective way to acknowledge applications with known performance issues (and therefore identify those applications whose low ratings have little to do with design) involves searching for key terms occurring abnormally frequently in text reviews. See table II for a list of key words and phrases that could indicate poor performance rather than poor design. Such key words and phrases are likely to indicate that low ratings are due to factors outside the realm of interface design. After gathering other metadata, fdscrape counts the frequency of these key words and phrases with respect to the available body of reviews. The calculated frequencies of these words are fed into the machine learning algorithm along with the tag frequency count from AGUILLE and the metadata from fdscrape.

If our algorithm were to put significant weight on these terms rather than the intended features in our heuristic, we can safely chalk these up to poor application performance. This gives a decision tree the option to discard obviously poor-quality applications in an early decision node in order to focus on the design factors we are interested in analyzing. If a more sophisticated method than calculating tag frequency proves necessary, we could take a naïve Baysean approach, analyzing the probability rather than the frequency of key phrases in known or exemplary good and bad applications.

By acknowledging applications which have known issues unrelated to design, observed ratings of the remaining applications in our dataset will better reflect design quality.

Key Phrase	Possible Conclusion
'incompatible'	Could indicate versioning or device compatibility issues for certain users.
'uninstall'	Could indicate frustration with the application or the inability for certain users to un-install pre-installed software.
'crash'	Could indicate stability issues.
'slow', 'lag'	Could indicate resource overloading, frequent Internet requests, or poor data structure implementation.
'black screen', 'white screen', 'blank screen'	Could indicate initialization problems.

TABLE II. KEY WORDS AND PHRASES SUGGESTING POOR PERFORMANCE

B. Experiment Results

Results would go here, once we decide what output of which machine learning workflows to include.

V. DISCUSSION & CHALLENGES

The single most significant setback to this project has been the failure of the Android fork of the GUITAR tool. We have been forced to develop an in-house tool from scratch in its place. It has taken weeks to develop AGUILLE to a usable and dependable state. While new developments such as those detailed in section IV-B show promising correlation

and prediction, preliminary results with primitive data did not show expected trends. Specifically, before AGUILLE and our machine-learning workflow became capable of more sophisticated data transformations, the sample heuristic (viz. mean amount of buttons per layout in each application) correlated against average rating showed no statistically significant results.

After improvement to AGUILLE and the addition of metadata and tag phrases, statistically significant results surfaced. Category discretization provided even better results, as discussed in section IV-B on the preceding page. We believe further probing into consequential design decisions and further sophistication of AGUILLE and the design heuristic will continue to render reportable, statistically significant results.

For example, a further step up in sophistication involves a report of what percentage of all available screen space is occupied by widgets.

VI. RELATED WORK

Available research into the overlap of the machine learning and user experience fields tends to concentrate on either GUI testing or programming interface (API) design rather than using machine learning to gain insight on the GUI design patterns users favor. Much available research that does indeed combine machine learning and user interface design aims to design front-end applications for the non-statistician that enable powerful data mining with little knowledge of the implementation of machine learning algorithms.

Arlt et. al. [6] have written a chapter on various different methods of parsing and testing GUIs. The research of Nguyen et. al. [7] presents a tool called GUITAR to parse the structure of an application's GUI in order to generated automated tests for that application. We were originally hopeful that GUITAR or one of its derivatives could prove invaluable in gathering GUI data to mine. Unfortunately, the Android-specific fork of GUITAR has not been updated since the release of Android 2.2 and is therefore not compatible with the majority of applications on F-Droid. Although much existing research [6], [8], [9] makes use of GUITAR, our GUI-parsing tool had to be developed from scratch as discussed in section III-B on page 3.

The approach posited by Yang et. al. [9] to programmatically generate GUI models in mobile applications does not use GUITAR in its entirely; rather, it analyzes GUI events using GUITAR and calls these events directly on the application. Similarly, Amalfitano et. al. [10] showcase "an automated technique that tests Android apps via their [GUI]." Although writing a GUI parser from scratch may have slowed down development, using just one tool to rip the GUI of an Android application where other teams have used many has helped to simplify the process of gathering GUI data.

The research of Shi et. al. [4] and Ferenc et. al. [5] discusses ways to better understand source code design patterns in Java and C++, respectively. Our research aims to discover design patterns in graphical interfaces, not implementation patterns in source code, setting our research apart from other pattern-based learning research.

Lieberman's research [3] discusses the concept of an "Interface Builder," a graphical tool assisting a developer in designing a user interface. He discusses *Programming by Example,* where the developer "takes on the role of operating the user interface in the same manner as the intended end-user would, interacting with the on-screen interface components to demonstrate concrete examples of how to use the interface." The application then learns and generalizes the developer's input to guess at the desired functionality. This research may prove invaluable to future research where the "smart interface builder" discussed above is being designed.

Papatheodorou's research [11] focuses chiefly on using machine learning to learn over time the sort of interface the user may expect and to adapt to that knowledge. It may be possible to use aspects of [11] to generalize the expectations of a range of users or potential users during the design process rather than after deployment, saving developers valuable time to test and improve their software. Our work, however, proposes a different approach to interface learning, requiring no such lengthy data collection process from users. We learn from freely available data, speeding up empirical research and eschewing the technical challenges and privacy concerns inherent in collecting data directly from users.

A promising, unique opportunity to combine the machine learning and user experience fields is missing in available research. We hope to open the door for future research to use machine learning and data mining to analyze a wealth of existing information about user interfaces. This will help developers of all platforms to better understand their users' preferences and peeves not only in graphical or mobile environment design but also in the design of overall user experience. With more intuition on user propensity and preference, developers can design more natural, intuitive software.

VII. FUTURE WORK

Our model would determine the likely rating for a graphical interface given the analyzed source of the application. Given a prototype GUI designed by a developer in an interface builder, the model could guess at a rating for the design based on trends it found in other applications in the same category.

It could also be possible to use a genetic algorithm to determine better positions and attributes for the interface elements in the GUI. The algorithm would weigh a high rating (as determined by the model) against changing the developer's design as little as possible, creating incremental changes to generate potentially optimized versions of the developer's original layout.

VIII. CONCLUSION

We have discussed a method for learning the correlation between certain elements of GUI design, which we will analyze with AGUILLE, and Google Play ratings, which we have mined from the Web. This is improved when factoring in Google Play metadata and discretizing applications by category. Because of the importance of optimized, intuitive interface on smaller devices, developers will benefit from insight from a model that attempts to explain user behavior and preference.

A summary of specific results should go here. It will mirror the summary that will end up in the introduction.

We are confident that further (perhaps automated) probing will continue to reveal interesting relationships between different elements. After learning our model, we could predict the quality of future interfaces. With future refinement, the algorithm could suggest interface improvements by means of a genetic process in an interactive interface builder.

REFERENCES

[1] D. Han, C. Zhang, X. Fan, A. Hindle, K. Wong, and E. Stroulia, "Understanding android fragmentation with topic analysis of vendor-specific bugs," in *Reverse Engineering (WCRE), 2012 19th Working Conference on*. IEEE, 2012, pp. 83–92.

[2] M. DeGusta, "Android orphans: Visualizing a sad history of support," 2011.

[3] H. Lieberman, "Computer-aided design of user interfaces by example," in *Computer-Aided Design of User Interfaces III*. Springer, 2002, pp. 1–12.

[4] N. Shi and R. A. Olsson, "Reverse engineering of design patterns from java source code," in *Automated Software Engineering, 2006. ASE'06. 21st IEEE/ACM International Conference on*. IEEE, 2006, pp. 123–134.

[5] R. Ferenc, A. Beszédes, L. Fülöp, and J. Lele, "Design pattern mining enhanced by machine learning," in *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*. IEEE, 2005, pp. 295–304.

[6] S. Arlt, S. Pahl, C. Bertolini, and M. Schäf, "Trends in model-based gui testing," *Advances in Computers*, vol. 86, pp. 183–222, 2012.

[7] B. N. Nguyen, B. Robbins, I. Banerjee, and A. Memon, "Guitar: an innovative tool for automated testing of gui-driven software," *Automated Software Engineering*, pp. 1–41, 2013.

[8] C. Hu and I. Neamtiu, "Automating gui testing for android applications," in *Proceedings of the 6th International Workshop on Automation of Software Test*. ACM, 2011, pp. 77–83.

[9] W. Yang, M. R. Prasad, and T. Xie, "A grey-box approach for automated gui-model generation of mobile applications," in *Fundamental Approaches to Software Engineering*. Springer, 2013, pp. 250–265.

[10] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon, "Using gui ripping for automated testing of android applications," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2012, pp. 258–261.

[11] C. Papatheodorou, "Machine learning in user modeling," in *Machine Learning and Its Applications*. Springer, 2001, pp. 286–294.