GUI2DSVec: Detecting Visual Design Smells Based on Semantic Embedding of GUI Images and Components

Bo Yang and Shanping Li(✉)

Zhejiang University, Hangzhou, China
{imyb,shan}@zju.edu.cn

Abstract. Visual graphical user interface testing (VGT) can simulate end-user behavior on essentially any device, and data-driven VGT approaches can automatically discern certain visual design imperfections. The visual information representing GUI images and the semantic information of components are crucial for data-driven design defect detection methodologies. The GUI representation techniques in the existing VGT approaches are incapable of unifying multimodal semantic information such as text content, screen images, metadata, et al., but require subsequent manual execution and implementation. To address the dual issues, this paper presents GUI2DSVec (GUI to Design Smell Vector), a novel GUI representation method that automatically extracts semantic information from GUI images and components to embedding vectors without manual exertion. By representing image and component metadata into vector space, this study introduces an automated end-to-end mobile application design smell detection method based on GUI representation. Performance validation on 64,759 real Android UIs demonstrates GUI2DSVec's efficiency in detecting visual design smells (accuracy@20 of 0.77). Additionally, the paper analyzes the impact of discrete representation modules and model variations on the performance of the smell detection method.

Keywords: Mobile GUI testing · GUI representation · Violation detection

1 Introduction

Graphical user interfaces (GUIs) serve as the portal through which end users access information and interact, playing an indispensable role in the mobile internet era. The process of GUI design and development is continuous and poses challenges even for senior designers and developers. Designers must adhere to numerous design guidelines [23], ensuring that the created prototype aligns with standards of consistent, beautiful, and effective [14]. For developers, the advancements in front-end technology [29] have introduced plenty dazzling visual effects to GUI design, such as dense streaming media embedding and dynamic particle animations. While these technologies improve the user experience, they

A. Zaslavsky et al. (Eds.): MobiQuitous 2023, LNICST 594, pp. 445–457, 2024.
https://doi.org/10.1007/978-3-031-63992-0_29

also present significant challenges for developers. Additionally, the abundance of development frameworks [15] and the existence of contentious development specifications [1] have further elevated the demand for developers' expertise.

These pressures on developers and designers render the occurrence of UI design smells in mobile application GUIs inevitable. A UI design smells[1] is any visual characteristic within the GUI, indicating issues that violate UI design principles and subsequently impact the user experience. Common design smells are shown in Fig. 1 (design smells marked in red boxes). In Fig. 1(a), the text superimposed upon the background image renders indistinct the title in the top bar. In Fig. 1(b), applying tabs and bottom navigation at the same time may bewilder users, for it remains confusing which tab controls the current content. In Fig. 1(c), employing icons to some destinations and not others in a navigation drawer. Icons should be used for all destinations, or none. In Fig. 1(d), within a simple dialog, the action button is redundant for the choice itself is actionable when tapped, and users can tap anywhere outside the dialog to close it. These design smells violate the design guidelines developed by Google Material Design documentation [2].

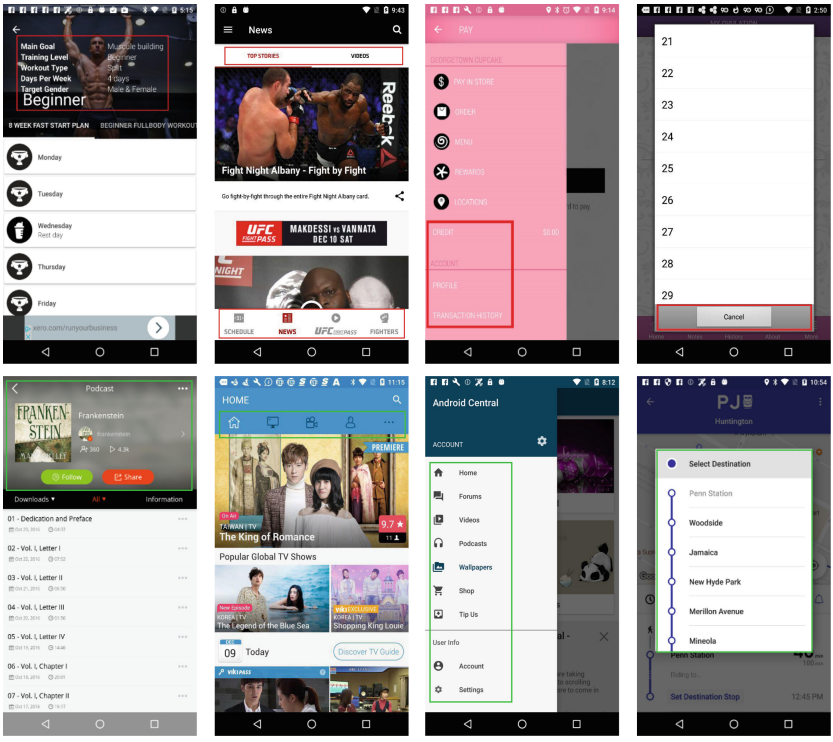
To ensure the correctness of the GUI display and fix design smells, development teams recruit testers or crowdsource GUI testing of applications. Due to the high cost and long duration of manual GUI testing, GUI automation testing technology has become a research focus in academia and industry. Alegroth et al. [4] divided GUI automation testing methods into three generations chronologically. The third-generation testing methods [20,32], namely visual GUI testing (VGT), refer to the utilization of image recognition as the core technology for analyzing and interacting with the front-end of software applications.

However, the current VGT methods are restricted in their ability to identify only a few UI design smells related to text and color [30]. The primary constraints can be summarized as follows: (1) Due to the current lack of design smell data sets, existing machine learning-based visual design smell detection models train predictive models on synthetic data sets [32], often making it difficult to ensure the effectiveness and practicality of data and models; (2) The generalization ability of the model is poor. Different models have different tasks and feature representation structures [22,32], making it difficult to map and learn in the same space; (3) Existing GUI image and component representation methods contain limited semantic information. More details can be found in Sect. 4.

To address the above problems, this paper makes the following contributions:

1. We propose a new GUI representation method GUI2DSVec, that automatically extracts atomic semantic information including text, layout, images, colors, etc. from GUI images and metadata.
2. We implement an automated end-to-end mobile application design smell detection method based on GUI representation.
3. Experiments show that GUI2DSVec achieves 77% accuracy and can automatically detect and classify visual design smells.

[1] The definition of design smell is inspired by code smell [13].



(a) Illegible buttons/text

(b) Tab and bottom navigation together

(c) Icons lost in navigation drawer

(d) No button in simple dialog

Fig. 1. UI design smells (1st row) vs non-smell UIs (2nd row) (issues highlighted in red boxes) (Color figure online)

2 A Semantic Embedding GUI Representation Model

GUI2DSVec is inspired by the word embedding method Word2Vec, that is, predicting the next entity based on the context of the target entity. In Word2Vec, words are used as input, while in GUI2DSVec, GUI component metadata and screenshots are used as input. The generated embedding vectors are used to improve downstream tasks driven by artificial intelligence. Semantic representations of GUIs are meaningful in GUI search [27] and GUI semantic summarization [18]. However, existing GUI representation methods also have limitations. One class of representation methods only focuses on text information on the screen and treats the screen as a bag-of-words model. For example, in SOVITE [16], the exact matching of text labels on the screen is used to summarize the tasks demonstrated by the user. Such representation methods ignore important GUI features such as visual information and layout patterns. Another class of representation methods [28] represents GUIs as sequences of user operations, using Seq2Seq models to map user operation sequences to corresponding

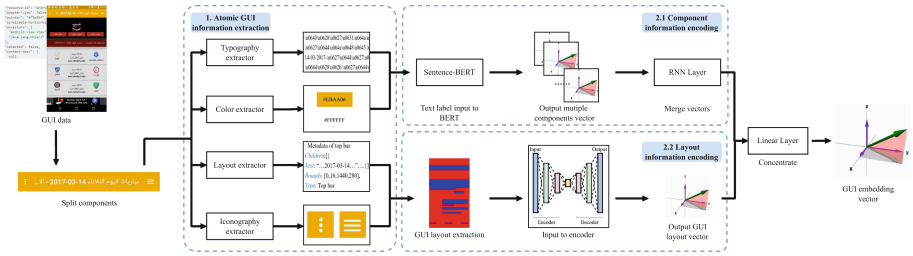


Fig. 2. Flowchart of GUI2DSVec for extracting atomic GUI information and generating GUI embedding vector

GUI components for predicting the user's possible next action or the natural language description of the interaction, etc.

In order to achieve prediction and classification of design smells, GUI2DSVec proposes using GUI hierarchical metadata and GUI images as input to extract component information, layout, and visual information for representation. The feature vectors of this information are extracted through their respective pre-trained models and finally connected as the feature vector of the GUI for subsequent classification tasks.

Figure 2 shows the workflow of GUI2DSVec. GUI2DSVec mainly includes two parts: GUI atomic information extraction and GUI feature information encoding. The goal of GUI atomic information extraction is to glean four crucial atomic GUI information from metadata and input images: text, color, layout, and images. Previous research [30] has demonstrated these four atomic GUI information can be utilized to identify design smells. GUI feature information encoding employs pre-trained models to separately encode the outputs of text and images from the atomic information extractors, separately yielding GUI component feature vectors and GUI layout feature vectors. Specifically, the GUI component feature vector utilizes a pre-trained Sentence-BERT model to calculate, and then merged by a recurrent neural network. The GUI layout feature vector adopts an encoder-decoder architecture for image encoding.

The GUI component and layout feature are then combined using a linear layer to form a 768-dimensional vector, which is GUI embedding vector. The GUI embedding vector can be used as a feature vector for the entire UI in downstream tasks. In this paper, we use visual design smell detection as a case study for demonstration. Our method combines GUI2DSVec and a KNN algorithm-based voter for design smell prediction. The method overview is shown in Fig. 3.

2.1 Atomic GUI Information Extraction

Text Information Extraction. The goal of this step is to crop the area to be processed according to the text area coordinates and screen captures in the metadata. By a scene text detector EAST [33] and text recognition techniques Tesseract [3], the extractor outputs the text information of the area to be

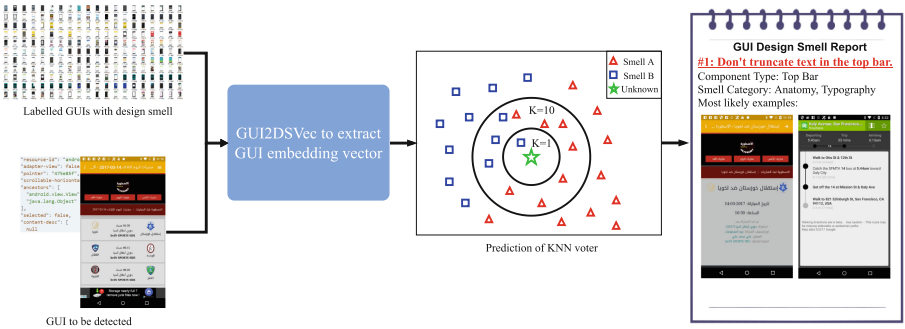


Fig. 3. Overview of design smell prediction

processed, including bounding boxes, number of lines, and strings. Although the area cropped according to the coordinates is basically accurate, it is still necessary to expand the image and perform grayscale processing in the subsequent text detection step to obtain better results. If the input data only contains screen captures, the text extractor has a built-in list for cropping common component areas from the original screen capture.

Color Information Extraction. This step enhances the initial metadata of GUI components by adding attributes to extract the dominant color of GUI components. We introduce the HSV color space for color detection, which is also utilized in recent work [8]. The technique calculates the colored area and corresponding image area percentage of each color. The color with the largest percentage is considered the dominant color of the GUI component.

Image Information Extraction. The image information extractor detects the presence of icons and images in the input UI. Then it attempts to simulate how application users perceive text, components and images in the UI. We uses the UI component detection tool [9] to detect non-text GUI subcomponent areas (such as icons and buttons) and uses EAST to detect text areas. By associating the detectable text and image areas with the component bounding boxes in the metadata, UI design smells related to readability can be determined, such as the two unreadable buttons in Fig. 1(a).

Layout Information Extraction. The layout extractor aims to glean the spatial information of components from the text metadata, such as component composition hierarchy, text content, component bounding box, and component type. Then it organizes these information into a JSON file to represent the GUI's spatial information. To avoid higher-level components in the hierarchy possessing larger bounding boxes causing overlap between components, the extractor

will also extract the edge information presented to the end user for rectification. Specifically, the Canny edge detection [7] is adopted to detect edges of components since the separator lines differ in color from the sides and remain horizontal.

2.2 GUI Components and Images Embedding Vectors

Text Label. The text label of each component is encoded into a word vector by the Sentence-BERT language model [24]. Sentence-BERT is a pre-trained model based on the BERT network, and specifically designed for generating sentence embeddings. Sentence-BERT uses a siamese architecture, which contains 2 BERT architectures that are essentially identical and share the same weights.

GUI Component Vector. In order to merge multiple GUI components into a single fixed-length feature vector, GUI2DSVec uses a recurrent neural network(RNN). The feature vectors of multiple GUI components are fed into the RNN in the preorder traversal order of the hierarchical structure. For each input vector, the hidden state input of the RNN is the output of the previous vector (the hidden state of the initial input vector is 0). This mechanism enables the final output of the RNN to encode all GUI components and can set the GUI feature vectors of different GUI images to a fixed length.

GUI Layout Vector. GUI2DSVec proposes a layout vector extractor for Rico's metadata. First, according to the hierarchical metadata, the bounding boxes of all leaf components are extracted, with different colors to distinguish text and non-text components. Then, an image autoencoder is used to encode each image into a 64-dimensional vector. The autoencoder is trained using a classic encoder-decoder architecture. The input dimension of the encoder is 11200, followed by hidden layers of size 2048 and 256, and the output dimension is 64. That is, $11200 \rightarrow 2048$, $2048 \rightarrow 256$, $256 \rightarrow 64$. These layers apply the ReLU activation function, so the output of each layer is transformed through the activation function to convert any negative input to 0. The decoder has the opposite structure ($64 \rightarrow 256$, $256 \rightarrow 2048$, $2048 \rightarrow 11200$). When the input image passes through the encoder and decoder, the layout vector extractor is trained during the reconstruction process, and the loss function is the mean square error (MSE). The network weights are optimized to produce the best 64-dimensional vector of the original input image to obtain the best reconstruction effect during decoding, that is, the minimum mean square error.

2.3 Concatenating GUI Embedding Vector

Linear layers are used to combine multiple vectors and compress the combined vector into a low-dimensional vector that contains the semantic content of each input. These weights are optimized together with other parameters to minimize the overall loss.

2.4 K-Nearest Neighbor Algorithm Predicts Violation List

The nearest neighbor algorithm [11] is used for searching the most similar patterns in the space. GUI2DSVec adopts cosine similarity to measure the distances between multiple GUI embedding vectors, and employs KNN to find the N screens most similar to a given screen from the dataset.

Specifically, for an input UI, GUI2DSVec first calculate the GUI embedding vector through the steps described above. Then GUI2DSVec find the K closest GUIs to the input GUI embedding vector in the dataset. We use the majority vote of the K nearest GUI cases to determine the type of design smell violated by the input UI. If there is a tie in the majority vote, we compare the average distances and consider the category with the shortest average distance to be the type of design smell violated by the input UI.

3 Experimental Evaluation

To train and evaluate GUI2DSVec in real-world apps, we use the open-source Rico dataset [12] to evaluate the classification effect of design smells. By combining different model variants of information and verifying their performance in classification tasks from different perspectives, this section aims to answer three research questions (RQs):

- RQ 1: What are the performance differences between GUI2DSVec and baselines?
- RQ 2: How do different models perform in smell classification from different perspectives?
- RQ 3: How is the usefulness of GUI2DSVec?

3.1 Experiment Setup

Dataset. The Rico dataset [12] contains 64,759 different GUI images from 9,384 real Android applications. To validate the effectiveness, the first author and another Ph.D. student, who is not the author of the paper, cross-annotated 19,164 UIs with smells and design smell types. Classification methods are based on Bo et al. [30]'s perspective of design smell-related knowledge. Design smells can be divided into anatomy, placement, behavior and usage from the perspective of component knowledge. From the general design knowledge perspective, they can be partitioned into layout, typography, iconography, navigation, communication, and color. Component categories include the 11 most commonly used components: banner, bottom bar, button, bottom navigation, text field, top bar, dialog, list, floating action button (FAB), tab, and navigation drawer.

Baselines. We compare GUI2DSVec to the following baseline models:

1. *Text Embedding Only:* only uses the embedding text as the screen embedding method applied in SOVITE [16]. It can only focus on the text information in the UI.

2. *Layout Embedding Only:* only uses GUI layout information as the screen embedding method applied in RICO [12]. The hierarchy files including type and location information of GUI components are obtained by the autoencoder and represent the entire UI for further prediction.
3. *Screen2Vec* [17]: uses textual content, visual image, and layout patterns to derive the final screen feature. Only the textual content of the GUI components is considered, i.e., the semantic component information contained behind the textual content is not considered.

Training Setup. The model is trained on the cross-entropy loss function using the Adam optimization algorithm. The learning rate is set to 0.001 and the batch size is 256. The model loss function is: $CrossEntropyLoss(t, p) = -\sum_{i \in S} t(i) log(p(i))$, where t denotes the true distribution and p denotes the predicted distribution. In extracting the GUI component information feature vector, the loss is the cross-entropy loss of text prediction. When calculating the cross-entropy loss, each text prediction is compared with each possible text embedding in the vocabulary. In extracting the GUI layout feature vector, the loss is the cross-entropy loss of image reconstruction. However, the layout feature vector does not contain the similarity between the correct prediction and each screen in the dataset. We adopt negative sampling to reduce the overhead of re-calculating each training iteration and avoid overfitting.

Experimental Design. Our experiments adopt the 10-fold cross validation [25] to verify our method. The average of the 10 results is output as the result of one 10-fold cross validation.

For RQ1, since the models of different classification methods are independent, we performed 10-fold cross validation of the model separately for the three classification methods. For RQ2 and RQ3, in addition to the full model that has been verified in RQ1, the remaining four model variants need to perform 10-fold cross validation on the three classification methods respectively.

Evaluation Metrics. In our experiment, we adopt *Accuracy* to measure the performance of our prediction results. Since the smell detection results of the UI to be detected are determined by the majority vote of the K nearest GUI cases, the accuracy of prediction is closely related to the number of k-nearest neighbors searched. Therefore, we use *accuracy*@K to represent the accuracy at different values of K. We use $K = 1, 5, 10, 20$ as developers are unlikely to look through a very long recommendation list.

3.2 RQ1: Performance

Table 1 reports the *accuracy*@20 of design smell prediction for the four methods. The results show that our method has the highest prediciton accuracy in general design and component categories and overall performance. The text-only baseline model has the lowest accuracy, while the layout-only baseline model has

moderate efficiency. This indicates that layout information has a greater impact on performance than text information in the design smell detection task. The Screen2Vec model gets good performance and surpasses our model in component design category, because our model introduces more embedding information and affects the prediction performance in a small number of classification categories.

Table 1. The prediction *accuracy*@20 of design smells for between our model and baselines

Model	Component Design Accuracy (4 types)	General Design Accuracy (6 types)	Component Accuracy (11 types)	Overall Accuracy
Text Only	0.48	0.54	0.59	0.581
Layout Only	0.57	0.60	0.61	0.604
Screen2Vec	**0.72**	0.74	0.74	0.730
GUI2DSVec	0.69	**0.91**	**0.75**	**0.770**

3.3 RQ2: Performance Under Different Classifications

The experiment selects three classification methods to verify the performance of GUI2DSVec. Table 1 reports the *accuracy*@20 of different models within classification methods. The Screen2Vec model (layout + text) performs best in the component knowledge category, even surpassing our model. We analyze that the reason for this is that the GUI embedding vector is too long after full connection with the text embedding vector, resulting in blurred layout embedding vector features and reduced prediction performance. For component category prediction, since similar components have similar layouts and semantic text, and this classification is refined into 11 subcategories, the performance gap of the models under this category is relatively small compared to other categories.

Table 2. Performance of detecting UI design smells for different types in real-app UIs

Typology	*Accuracy*@1	*Accuracy*@5	*Accuracy*@10	*Accuracy*@20
Component Design (4 types)	0.65	0.66	0.67	0.69
General Design (6 types)	0.90	0.90	0.91	0.91
Component (11 types)	0.66	0.69	0.71	0.75
Overall	**0.739**	**0.751**	**0.759**	**0.770**

3.4 RQ3: Usefulness

Table 2 shows the prediction accuracy of GUI2DSVec under different classification methods and different K settings. It can be found that GUI2DSVec achieved

an accuracy of 0.77 in the classification of 21 categories, indicating that for an unknown GUI, GUI2DSVec can efficiently point out problematic components that may have design smells. However, for the component knowledge category and the general design knowledge category, the general design knowledge category with more categories has higher accuracy. We believe the main reason is that the classification and data of general design knowledge are more clear. For example, if it can be pointed out that there are design smells in color, then GUI with similar color layouts are more likely to have design smells; while the too detailed anatomy and usage in the component design knowledge category cannot be clearly distinguished in similar layouts. Discerning comparison of distinct attributes and styles may be requisite for judgment.

4 Related Work

Functional GUI testing [6,26] focuses on GUI display issues that may cause application crashes. This paper focuses more on non-functional GUI testing, i.e., visual smells that will not cause application crashes but have negative impacts on application usability and user experience. The purpose of visual smell detection methods is to enable computers to understand GUIs (i.e., extract GUI features), detect design smells (i.e., identify abnormal visual features from normal GUIs and locate them), and finally provide detection reports.

Zhao et al. [32] first proposed to transform the problem of detecting design smells in GUI animations into a classification problem. They used the encoding part of the GAN as the representation model for GUI animations and used the KNN K-nearest neighbor algorithm to implement the classifier. In the absence of enough labelled data, this method uses GAN and unlabelled real GUI animations to synthesize labelled GUI animation datasets. The result is limited by the classification model, which can only classify the entire UI and cannot locate design defects in a fine-grained manner.

Liu et al. [20] proposed the Owleyes tool for text overlap, image loss, flower screen and other five design smells caused by device compatibility. This method first located design smells on GUI images, but the located areas were too large, and the classification model for these five specific design smells had poor generalization ability. Since the synthesized UIs instead of real UIs were used for training, it could lead to false positives.

The subsequent tools like Nighthawk [21] and Woodpecker [19] are derived from Owleyes. Nighthawk enhances the end-to-end ResNet model, while Woodpecker correlates the target area with the component source code. There are still problems such as poor generalization ability, false positives caused by data enhancement methods and limited smell types.

Alotaibi et al. [5] defined a graph-based component size relationship model for visual defects such as component stacking caused by scaling GUIs. This method requires static analysis of GUIs to extract hierarchical structures and generate size relationship graph models. It is less applicable than methods that only require GUI images as input and detect fewer visual defect categories.

Chen et al. [10] followed the ideas of Zhao et al. [32] to detect smells in image-rich applications (such as games). They not only enhanced the realism of synthesized data by injecting defective code snippets into the code but also extended the types of detectable smells to eight. Similarly, this method requires access to the source code of the application for code injection and has only been verified on game UIs, with generally poor applicability.

Zhang et al. [31] focused on detecting consistency between application programs, GUI policies and legal regulations. Through static analysis and normalization of expert knowledge, they detected four types of design smells that violate the Google Material Design specifications. However, its effectiveness in practice remains to be verified due to the need for application source code and less information involved in these four design smells.

Su et al. [27] proposed dVermin that detect visual defects in GUIs when magnified. This method has achieved good performance on scaling issues but cannot handle other types of design smells.

Previous studies have mainly focused on typesetting (text overlap [19–21] or size anomaly [31]), images (flower screen [10,20]), layout (distortion caused by proportional enlargement [5,27]), and animation (shadow loss after animation [32]) in specific design defects, They have not been able to solve the design defects in text hiding, blurry buttons, interactions and navigation mentioned in Fig. 1.

5 Conclusion and Future Work

This paper proposes a novel GUI representation method called GUI2DSVec to enhance data-driven VGT approaches, implementing an automated end-to-end UI design smell detection methodology. The evaluation of the real-app dataset indicates the utility of our representation modules, and the high detection accuracy of the comprehensive model. In the future, we will extend the method to automatically explore and detect smells throughout the mobile applications at the granularity of source code. Additional studies shall be conducted to validate the usefulness of our methodology and its extensions.

References

1. Html5 specification finalized, squabbling over specs continues (2014). https://tinyurl.com/3ckk6sk2
2. Google material design (2023). https://m2.material.io/components/
3. Tesseract ocr (2023). https://tesseract-ocr.github.io/
4. Alégroth, E., Gao, Z., Oliveira, R., Memon, A.: Conceptualization and evaluation of component-based testing unified with visual GUI testing: an empirical study. In: 2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST), pp. 1–10. IEEE (2015)
5. Alotaibi, A.S., Chiou, P.T., Halfond, W.G.: Automated repair of size-based inaccessibility issues in mobile applications. In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 730–742. IEEE (2021)

6. Amalfitano, D., Fasolino, A.R., Tramontana, P., Ta, B.D., Memon, A.M.: Mobiguitar: automated model-based testing of mobile apps. IEEE Softw. **32**(5), 53–59 (2014)
7. Canny, J.: A computational approach to edge detection. IEEE Trans. Pattern Anal. Mach. Intell. **6**, 679–698 (1986)
8. Chen, C., Feng, S., Xing, Z., Liu, L., Zhao, S., Wang, J.: Gallery dc: design search and knowledge discovery through auto-created GUI component gallery. Proc. ACM Hum.-Comput. Interact. **3**(CSCW), 1–22 (2019)
9. Chen, J., et al.: Object detection for graphical user interface: old fashioned or deep learning or a combination? In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 1202–1214 (2020)
10. Chen, K., Li, Y., Chen, Y., Fan, C., Hu, Z., Yang, W.: Glib: towards automated test oracle for graphically-rich applications. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 1093–1104 (2021)
11. Cover, T., Hart, P.: Nearest neighbor pattern classification. IEEE Trans. Inf. Theory **13**(1), 21–27 (1967)
12. Deka, B., et al.: Rico: a mobile app dataset for building data-driven design applications. In: Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology, pp. 845–854 (2017)
13. Fowler, M., Beck, K.: Refactoring: Improving the design of existing code. In: 11th European Conference, Jyväskylä, Finland (1997)
14. Galitz, W.O.: The Essential Guide to user Interface Design: an Introduction to GUI Design Principles and Techniques. Wiley, Hoboken (2007)
15. Kaluža, M., Troskot, K., Vukelić, B.: Comparison of front-end frameworks for web applications development. Zbornik Veleučilišta u Rijeci **6**(1), 261–282 (2018)
16. Li, T.J.J., Chen, J., Xia, H., Mitchell, T.M., Myers, B.A.: Multi-modal repairs of conversational breakdowns in task-oriented dialogs. In: Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology, pp. 1094–1107 (2020)
17. Li, T.J.J., Popowski, L., Mitchell, T., Myers, B.A.: Screen2vec: Semantic embedding of GUI screens and GUI components. In: Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems, pp. 1–15 (2021)
18. Li, T.J.J., Riva, O.: Kite: Building conversational bots from mobile apps. In: Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services, pp. 96–109 (2018)
19. Liu, Z.: Woodpecker: identifying and fixing android UI display issues. In: Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings, pp. 334–336 (2022)
20. Liu, Z., Chen, C., Wang, J., Huang, Y., Hu, J., Wang, Q.: Owl eyes: Spotting UI display issues via visual understanding. In: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, pp. 398–409 (2020)
21. Liu, Z., Chen, C., Wang, J., Huang, Y., Hu, J., Wang, Q.: Nighthawk: fully automated localizing UI display issues via visual understanding. IEEE Trans. Software Eng. **49**(1), 403–418 (2022)
22. Moran, K., Li, B., Bernal-Cárdenas, C., Jelf, D., Poshyvanyk, D.: Automated reporting of GUI design violations for mobile apps. In: Proceedings of the 40th International Conference on Software Engineering, pp. 165–175 (2018)

23. Nielsen, J., Molich, R.: Heuristic evaluation of user interfaces. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI 1990, pp. 249–256. Association for Computing Machinery, New York, NY, USA (1990)
24. Reimers, N., Gurevych, I.: Sentence-bert: sentence embeddings using siamese bert-networks. arXiv preprint arXiv:1908.10084 (2019)
25. Stone, M.: Cross-validatory choice and assessment of statistical predictions. J. Roy. Stat. Soc. Ser. B Methodol. **36**(2), 111–133 (1974)
26. Su, T., et al.: Guided, stochastic model-based GUI testing of android apps. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, pp. 245–256 (2017)
27. Su, Y., et al.: The metamorphosis: automatic detection of scaling issues for mobile apps. In: 37th IEEE/ACM International Conference on Automated Software Engineering, pp. 1–12 (2022)
28. White, L., Almezen, H.: Generating test cases for GUI responsibilities using complete interaction sequences. In: Proceedings 11th International Symposium on Software Reliability Engineering. ISSRE 2000, pp. 110–121. IEEE (2000)
29. Xing, Y., Huang, J., Lai, Y.: Research and analysis of the front-end frameworks and libraries in e-business development. In: Proceedings of the 2019 11th International Conference on Computer and Automation Engineering, pp. 68–72 (2019)
30. Yang, B., Xing, Z., Xia, X., Chen, C., Ye, D., Li, S.: Don't do that! hunting down visual design smells in complex UIS against design guidelines. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), pp. 761–772. IEEE (2021)
31. Zhang, Z., Feng, Y., Ernst, M.D., Porst, S., Dillig, I.: Checking conformance of applications against GUI policies. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 95–106 (2021)
32. Zhao, D., et al.: Seenomaly: vision-based linting of GUI animation effects against design-don't guidelines. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, pp. 1286–1297 (2020)
33. Zhou, X., et al.: East: an efficient and accurate scene text detector. In: Proceedings of the IEEE conference on Computer Vision and Pattern Recognition, pp. 5551–5560 (2017)