Psychologically-Inspired, Unsupervised Inference of Perceptual Groups of GUI Widgets from GUI Images

Mulong Xie
mulong.xie@anu.edu.au
Australian National University
Australia

Zhenchang Xing
zhenchang.xing@anu.edu.au
CSIRO's Data61 & ANU
Australia

Sidong Feng
sidong.feng@monash.edu
Monash University
Australia

Xiwei Xu
xiwei.xu@data61.csiro.au
CSIRO's Data61
Australia

Liming Zhu
liming.zhu@data61.csiro.au
CSIRO's Data61 & UNSW
Australia

Chunyang Chen
chunyang.chen@monash.edu
Monash University
Australia

ABSTRACT

Graphical User Interface (GUI) is not merely a collection of individual and unrelated widgets, but rather partitions discrete widgets into groups by various visual cues, thus forming higher-order perceptual units such as tab, menu, card or list. The ability to automatically segment a GUI into perceptual groups of widgets constitutes a fundamental component of visual intelligence to automate GUI design, implementation and automation tasks. Although humans can partition a GUI into meaningful perceptual groups of widgets in a highly reliable way, perceptual grouping is still an open challenge for computational approaches. Existing methods rely on ad-hoc heuristics or supervised machine learning that is dependent on specific GUI implementations and runtime information. Research in psychology and biological vision has formulated a set of principles (i.e., Gestalt theory of perception) that describe how humans group elements in visual scenes based on visual cues like connectivity, similarity, proximity and continuity. These principles are domain-independent and have been widely adopted by practitioners to structure content on GUIs to improve aesthetic pleasantness and usability. Inspired by these principles, we present a novel unsupervised image-based method for inferring perceptual groups of GUI widgets. Our method requires only GUI pixel images, is independent of GUI implementation, and does not require any training data. The evaluation on a dataset of 1,091 GUIs collected from 772 mobile apps and 20 UI design mockups shows that our method significantly outperforms the state-of-the-art ad-hoc heuristics-based baseline. Our perceptual grouping method creates opportunities for improving UI-related software engineering tasks.

CCS CONCEPTS

• **Software and its engineering**;

KEYWORDS

Graphical User Interface, Widget Grouping, Perceptual Grouping

ESEC/FSE '22, November 14–18, 2022, Singapore, Singapore

ACM ISBN 978-1-4503-9413-0/22/11.
https://doi.org/10.1145/3540250.3549138

ACM Reference Format:
Mulong Xie, Zhenchang Xing, Sidong Feng, Xiwei Xu, Liming Zhu, and Chunyang Chen. 2022. Psychologically-Inspired, Unsupervised Inference of Perceptual Groups of GUI Widgets from GUI Images. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '22), November 14–18, 2022, Singapore, Singapore.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3540250.3549138

1 INTRODUCTION

We do not just see a collection of separated texts, images, buttons, etc., on GUIs. Instead, we see perceptual groups of GUI widgets, such as card, list, tab and menu shown in Figure 1. Forming perceptual groups is an essential step towards visual intelligence. For example, it helps us decide which actions are most applicable to certain GUI parts, such as, clicking a navigation tab, expanding a card, scroll the list. This would enable more efficient automatic GUI testing [22, 35]. As another example, screen readers [3, 8] help visually impaired users access applications by reading out content on GUI. Recognizing perceptual groups would allow screen readers to navigate the GUI at higher-order perceptual units (e.g., sections) efficiently [58]. Last but not least, GUI requirements, designs and implementations are much more volatile than business logic and functional algorithms. With perceptual grouping, modular, reusable GUI code can be automatically generated from GUI design images, which would expedite rapid GUI prototyping and evolution [37, 38].

Although humans can intuitively see perceptual groups of GUI widgets, current computational approaches are limited in partitioning a GUI into meaningful groups of widgets. Some recent work [12, 16] relies on supervised deep learning methods (e.g., image captioning [34, 51]) to generate a view hierarchy for a GUI image. This type of method is heavily dependent on GUI data availability and quality. To obtain sufficient GUI data for model training, they use GUI screenshots and view hierarchies obtained at application runtime. A critical quality issue of such runtime GUI data is that runtime view hierarchies often do not correspond to intuitive perceptual groups due to many implementation-level tricks. For example, in the left GUI in Figure 2, the two ListItems in a ListView has no visual similarity (a large image versus some texts), so they do not form a perceptual group. In the right GUI, a grid of cards form a perceptual group but is implemented as individual FrameLayouts. Such inconsistencies between the implemented widget groups and

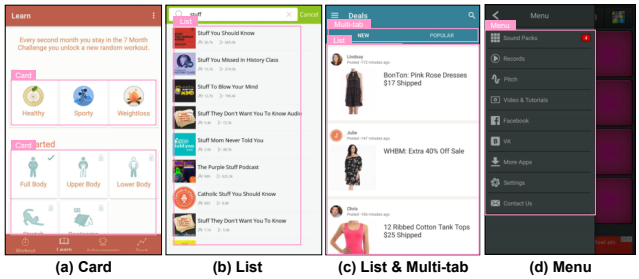

Figure 1: Examples of perceptual groups of GUI widgets (perceptual groups are highlighted in pink box in this paper)

the human's perceptual groups make the trained models unreliable to detect perceptual groups of GUI widgets.

Decades of psychology and biological vision research have formulated the Gestalt theory of perception that explains how humans see the whole rather than individual and unrelated parts. It includes a set of principles of grouping, among which *connectedness*, *similarity*, *proximity* and *continuity* are the most essential ones [7, 45]. Although these principles and other related UI design principles such as CRAP [42] greatly influence how designers and developers structure GUI widgets [48], they have never been systematically used to automatically infer perceptual groups from GUI images. Rather, current approaches [38, 58] rely on ad-hoc and case-specific rules and thus are hard to generalize on diverse GUI designs.

In this work, we systematically explore the Gestalt principles of grouping and design the first psychologically-inspired method for visual inference of perceptual groups of GUI widgets. Our method requires only GUI pixel images and is independent of GUI implementations. Our method is unsupervised, thus removing the dependence on problematic GUI runtime data. As shwon in Figure 3, our method enhances the state-of-the-art GUI widget detection method (UIED [21, 53]) to detect elementary GUI widgets. Following the Gestalt principles, the method first detects containers (e.g., card, list item) with complex widgets by the connectedness principle. It then clusters visually similar texts (or non-text widgets) by the similarity principle and further groups clusters of widgets by the proximity principles. Finally, based on the widget clusters, our method corrects erroneously detected or missing GUI widgets by the continuity principle (not illustrated in Figure 3, but can be seen in Figure 5).

At the right end of Figure 3, we show the grouping result by the state-of-the-art heuristic-based method Screen Recognition [58]. Screen Recognition incorrectly partitions many widgets into groups, such as the bottom navigation bar and the four widgets above the bar, the card on the left and the text above the card. It also fails to detect higher-order perceptual groups, such as groups of cards. In contrast, our approach correctly recognizes the bottom navigation bar and the top and middle row of cards. Although the text label above the left card is very close to the card, our approach correctly recognizes the text labels as separate widgets rather than as a part of the left card. Our approach does not recognize the two cards just above the bottom navigation bar because these two cards are partially occluded by the bottom bar. However, it correctly recognizes the two blocks of image and text and detects them as a group.

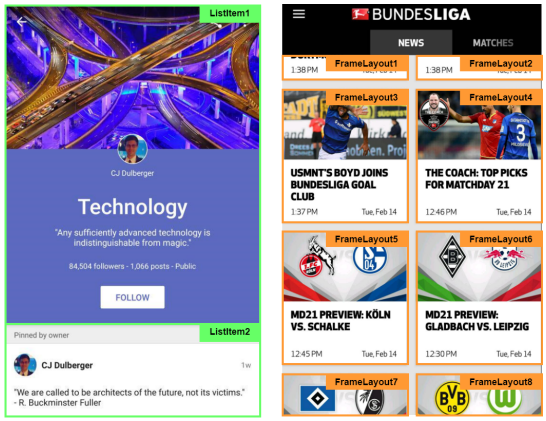

Figure 2: Implemented view hierarchy does not necessarily correspond to perceptual groups

Clearly, the grouping results by our approach correspond more intuitively to human perception than those by Screen Recognition.

For the evaluation, we construct two datasets: one contains 1,091 GUI screenshots from 772 Android apps, and the other contains 20 UI prototypes from a popular design tool Figma [4]. To ensure the validity of ground-truth widget groups, we manually check all these GUIs and confirm that none of these GUIs has the perception-implementation misalignment issues shown in Figure 2. We first examine our enhanced version of UIED and observe that the enhanced version reaches a 0.626 F1 score for GUI widget detection, which is much higher than the original version (0.524 F1). With the detected GUI widgets, our perceptual grouping approach achieves the F1-score of 0.593 on the 1,091 app GUI screenshots and 0.783 F1-score on the 20 UI design prototypes. To understand the impact of GUI widget misdetections on perceptual grouping, we extract the GUI widgets directly from the Android app's runtime metadata (i.e., ground-truth widgets) and use the ground-truth widgets as the inputs to perceptual grouping. With such "perfectly-detected" GUI widgets, our grouping approach achieves a 0.672 F1-score on app GUIs. In contrast, Screen Recognition [58] performs very poorly: 0.123 F1 on the ground-truth widgets and 0.092 F1 on the detected widgets for app screenshots, and 0.232 F1 on the detected widgets for UI design prototypes. Although our grouping results sometimes do not exactly match the ground-truth groups, our analysis shows that some of our grouping results still comply with how humans perceive the widget groups because there can be diverse ways to structure GUI widgets in some cases.

To summarize, this paper makes the following contributions:

- A robust, psychologically-inspired, unsupervised visual inference method for detecting perceptual groups of GUI widgets on GUI images, the code is released on GitHub[1].
- A comprehensive evaluation of the proposed approach and the in-depth analysis of the performance with examples.
- An analysis of how our perceptual grouping method can improve UI-related SE tasks, such as UI design, implementation and automation.

[1]https://github.com/MulongXie/GUI-Perceptual-Grouping

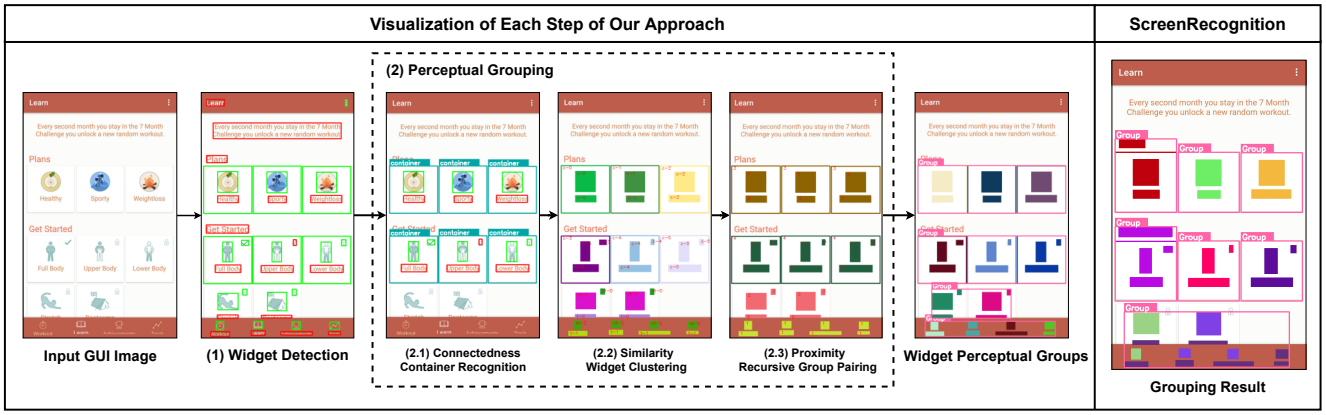

Figure 3: Left: Our approach overview: (1) Enhanced UIED [21] for GUI widget detection; (2) Gestalt-principles inspired perceptual grouping. Right: Grouping result of the state of of the art heuristic-based approach ScreenRecognition [58]

2 GUI WIDGET DETECTION

Our approach is a pixel-only approach. It does not assume any GUI metadata or GUI implementation about GUI widgets. Instead, our approach detects text and non-text GUI widgets directly from GUI images. To obtain the widget information from pixels, it enhances the state-of-the-art GUI widget detection technique UIED [53]. In order to fit with subsequent perceptual grouping, our approach mitigates the incorrect merging of GUI widgets in the containers by UIED and simplifies the widget classification of UIED.

2.1 UIED-Based GUI Widget Detection

UIED comprises three steps: (1) GUI text detection, (2) non-text GUI widget detection and (3) merging of text and non-text widgets. UIED uses an off-the-shelf scene text detector EAST [60] to identify text regions in the GUI images. EAST is designed for handling nature scene images that differ from GUI images, such as figure-background complexity and lighting effects. Although EAST outperforms traditional OCR tool Tesseract [47], we find the latest OCR tool developed by Google [6] achieves the highest accuracy of GUI text recognition (see Section 4.1). Therefore, in our use of UIED, we replace EAST with the Google OCR tool.

For locating non-text widgets, our approach adopts the design of UIED that uses a series of traditional, unsupervised image processing algorithms rather than deep-learning models (e.g., Faster RCNN [31] or YOLO [40]). This design removes the data dependence on GUI implementation or runtime information while accurately detecting GUI widgets. UIED then merges the text and non-text detection results. The purpose of this merging step is not only to integrate the identified GUI widgets but also to cross-check the results. Because non-text widget detection inevitably extracts some text regions, UIED counts on the OCR results to remove these false-positive non-text widgets. Specifically, this step checks the bounding box of all candidate non-text widget regions and removes those intersected with text regions resulting from the OCR.

2.2 Improvement and Simplification of UIED

We find that the UIED detection results often miss some widgets in a container (e.g. card). The reason is that, in order to filter out invalid non-text regions and mitigate over-segmentation that wrongly segments a GUI widget into several parts, UIED checks the widgets' bounding boxes and merges all intersected widgets regions into a big widget region. This operation may cause the wrong neglection of valid GUI widgets that are enclosed in some containers. Therefore, we equip our GUI widget detector with a container recognition algorithm (see Section 3.2) to mitigate the issue. If a widget is recognized as a container, then all its contained widgets are kept and regarded as proper GUI widgets rather than noises.

The original UIED classifies non-text GUI widgets as specific widget categories (e.g., image, button, checkbox). In contrast, our GUI widget detector only distinguishes text from non-text widgets. Although GUI involves many types of non-text widgets, there is no need of distinguishing actual classes of non-text widgets for perceptual grouping. GUI widget classes indicate different user interactions and GUI functionalities, but widgets with different classes can form a perceptual group as long as they have similar visual properties, such as size, shape, relative position and alignment with other widgets. Therefore, we do not distinguish different classes of non-text widgets. However, we need to distinguish non-text widgets from text widgets, as they have very different visual properties and need to be clustered by different strategies (see Section 3).

3 GUI WIDGET PERCEPTUAL GROUPING

After obtaining text and non-text widgets on a GUI image, the next step is to partition GUI widgets into perceptual groups (or blocks of items) according to their visual and perceptual properties.

3.1 Gestalt Laws and Approach Overview

Our approach is inspired by psychology and biological-vision research. Perceptual grouping is a cognitive process in which our minds leap from comprehending all of the objects as individuals to recognizing visual patterns through grouping visually related elements as a whole [26]. This process affects the way we design GUI layouts [42] from alignment, spacing and grouping tool support [4, 44] to UI design templates [24] and GUI frameworks [2] It also explains how we perceive GUI layouts. For instance, in the examples in Figure 1, we subconsciously observe that some visually

similar widgets are placed in a spatially similar way and identify them as in a group (e.g. a card, list, multitab or menu).

Previous studies rely on ad-hoc, rigid heuristics to infer UI structure without a systematic theoretical foundation. Our approach is the first attempt to tackle the perceptual grouping of GUI widgets guided by an influential psychological theory (named Gestalt psychology [7]) that explains how the human brain perceives objects and patterns. Gestalt psychology's core proposition is that human understands external stimuli as wholes rather than as the sums of their parts [46]. Based on the proposition, the Gestaltists studied perceptual grouping [26] systematically and summarized a set of "gestalt laws of grouping" [45]. In our work, we adopt the four most effective principles which greatly influence GUI design [48] in practice as the guideline for our approach design: (1) connectedness (2) similarity (3) proximity and (4) continuity.

We define a group of related GUI widgets as a *layout block* of *items*. A typical example is a *list* of *list items* in the GUI, or a *card* displaying an image and some texts. The fundamental intuition is: if a set of widgets have similar visual properties and are placed in alignment with similar space between each other, they will be "perceived" as in the same *layout block* by our approach according to the Gestalt principles. In detail, our approach consists of four grouping steps in accordance with four Gestalt principles. First, it identifies containers along with their contained widgets that fulfil the connectedness law. Second, it uses an unsupervised clustering method DBSCAN [25] to cluster text or non-text GUI widgets based on their spatial and size similarities. Next, it groups proximate and spatially aligned clusters to form a larger layout block following the proximity law. Finally, in line with the continuity principle, our approach corrects some mistakes of GUI widget detection by checking the consistency of the groups' compositions.

3.2 Connectedness - Container Recognition

In Gestalt psychology, the principle of uniform connectedness is the strongest principle concerned with relatedness [41]. It implies that we perceive elements connected by uniform visual properties as being more related than those not connected. The forms of the connection can be either a line connecting several elements or a shape boundary that encloses a group of related elements. In the GUI, the presentation of the connectedness is usually a box container that contains multiple widgets within it, and all the enclosed widgets are perceived as in the same group. Thus, the first step of our grouping approach is to recognize the containers in a GUI.

In particular, we observe that a container is visually a (round) rectangular wireframe enclosing several children widgets. The card is a typical example of such containers, as shown in Figure 1(a). Therefore, with the detected non-text widgets, the algorithm first checks if a widget is of rectangle shape by counting how many straight lines its boundary comprises and how they compose. Specifically, we apply the geometrical rule that a rectangle's sides are made of 4 straight lines perpendicular to each other. Subsequently, our approach determines if the widget's boundary is a wireframe border by checking if it is connected with any widgets inside its boundary. If a widget satisfies the above criteria, it will be identified as a container, and all widgets contained within it are partitioned into the same perceptual group.

Figure 4: Widget clustering, cluster conflict resolving and final resulting groups in which we use the same color to paint the widgets in the same subgroup and highlight higher-order groups in pink boxes

3.3 Similarity - Widget Clustering

The principle of similarity suggests that elements are perceptually grouped together if they are similar to each other [9]. Generally, similarity can be observed in aspects of various visual cues, such as size, color, shape or position. For example, in the second GUI of Figure 1, the image widgets are of the same size and aligned with each other in the same way (i.e., same direction and spacing), so we visually perceive them as a group. Likewise, the text pieces on the right of the image widgets are perceptually similar even though they have different font styles and lengths because they have the same alignment with neighbouring texts.

3.3.1 Initial Widget Clustering. Our approach identifies similar GUI widgets by their spatial and visual properties and aggregates similar GUI widgets into blocks by similarity-based clustering. It clusters texts and non-text widgets through different strategies. In general, similar non-text widgets in the same block (e.g. a *list*) usually have similar sizes and align to one another vertically or horizontally with the same spacing. Texts in the same block are always left-justified or top-justified (assume left-to-right text orientation), but their sizes and shapes can vary significantly because of different lengths of text contents. Thus, the approach clusters the non-text widgets by their center points $(Center_X, Center_Y)$ and areas, and it clusters texts by their top-left corner $(Top, Left)$

Our approach uses the DBSCAN (Density-Based Spatial Clustering of Applications with Noise) algorithm [25] to implement the clustering. Intuitively, DBSCAN groups the points closely packed together (points with many nearby neighbors), while marking points whose distance from the nearest neighbor is greater than the maximum threshold as outliers. In the GUI widget clustering context, the point is the GUI widget, and the distance is the difference between the values of the widgets' attribute that the clustering is based on. Figure 4 illustrates the clustering process. For non-text widgets, our approach performs the clustering three times based on three attributes respectively. It first clusters the widgets by $Center_x$ for the horizontal alignment, then by $Center_Y$ for the vertical alignment and finally by *area*. These operations produce three clusters: $Cluster^{non-text}_{horizontal}$, $Cluster^{non-text}_{vertical}$ and $Cluster^{non-text}_{area}$. Our approach then clusters the text widgets twice based on their top left

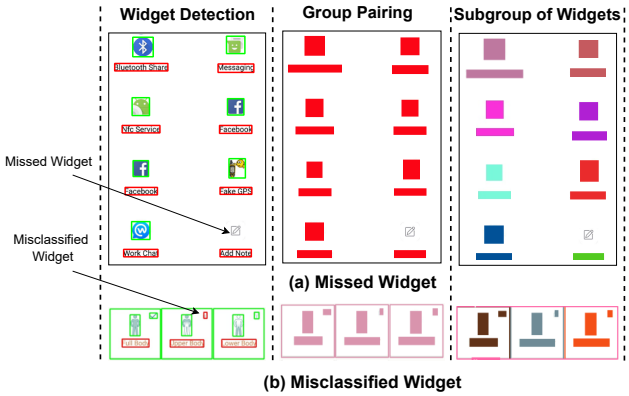

Figure 5: Examples of widget detection error correction. (1st column - green box: non-text; red-box: text; 2nd column - same color: higher-order perceptual group; 3rd column - same color: subgroup of widgets)

corner point $(Top, Left)$ for left-justified (vertical) and top-justified (horizontal) alignment. It produces the $Cluster^{text}_{horizontal}$ based on the texts' Top, and the $Cluster^{text}_{vertical}$ based on the texts' $Left$. The resulting clusters are highlighted by different colors and numbers in Figure 4. We only keep the clusters with at least two widgets and discard those with only one widget.

3.3.2 Cluster Conflicts Resolving. It is common that some widgets can be clustered into different clusters by different attributes, which causes cluster conflicts. For example, as illustrated in Figure 4, several non-text widgets (e.g., the bottom-left image) are both in a vertical cluster (marked in blue) and a horizontal cluster (marked in red). The intersection of clusters illustrates the conflict. The approach shall resolve such cluster conflicts to determine to which group the widget belongs. This conflict-resolving step also complies with the similarity principle that suggests the widgets in the same perceptual group should share more similar properties.

The conflict resolving step first calculates the average widget areas of the groups to which the conflicting widget has been assigned. In accordance with the similarity principle, the widget is more likely to be in a group whose average widget area is similar to the conflicting widget's area. In addition, another observation is that repetitive widgets in a group have similar spacing between each other. So for a widget that is clustered into multiple candidate groups, the approach checks the difference between the spacing of this widget and its neighboring widgets in a group and the average spacing between other widgets in that group. It keeps the widget in the group where the conflicting widget has the largest widget-area similarity and the smallest spacing difference compared with other widgets in the group. For example, the bottom-left image widget will be assigned to the vertical cluster rather than the horizontal one according to our conflict resolving criteria. After conflict resolving, our approach produces the final widget clustering results as shown in the right part of Figure 4. We use different colors and indices to illustrate the resulting non-text (nt) and text (t) clusters.

3.4 Proximity - Iterative Group Pairing

So far, GUI widgets are aggregated into groups as per the connectedness and similarity principles. Some groups are close to each other and similar in terms of the number and layout of the contained widgets, which may further form a larger perceptual group even though these groups may contain different types of widgets. For example, in the clustering result of Figure 4, we can observe that the clusters *nt-0*, *t-0*, *t-0-1* and *nt-2* are proximate and have the same or similar number of widgets aligned in the same way. We can see this feature intentionally or subconsciously and perceive them as in the same large group as a whole. Gestalt psychology states that when people see an assortment of objects, they tend to perceive objects that are close to each other as a group [9]. The close distance, also known as proximity, of elements is so powerful that it can override widget similarity and other factors that might differentiate a group of objects [50]. Thus, the next step is based on widget clusters' proximity and composition similarity to pair the clusters into a larger group (i.e., layout block).

If two groups $Group_a$ and $Group_b$ are next (proximate) to each other (i.e., no other groups in between), and they contain the same number of widgets and the widgets in the $Group_a$ and the $Group_b$ share the same orientation (vertical or horizontal), our approach combines $Group_a$ and $Group_b$ into a larger block. A widget in $Group_a$ and its closet widget in $Group_b$ will be paired and form a subgroup of widgets. Our approach first combines the two proximate groups containing the same type of widgets, and then the groups containing different types of widgets. The formed larger block can be iteratively combined with the proximate groups until no more proximate groups are available.

Sometimes there are different numbers of widgets in the two proximate groups but the two groups may still form one larger perceptual block. For example, the cluster *nt-2* in Figure 4 has one less widget compared to *nt-0*, *t-0* and *t-0-1* because the bottom widget in the right column is occluded by the float action button and thus missed by the detector. Another common reason for the widget number difference is that widgets in a group may be set as invisible in some situations, and thus they do not appear visually. Therefore, if the difference between the number of widgets in the two proximate groups is less than 4 (empirically determined from the ground-truth groups in our dataset), our approach also combines the two groups into a larger block.

As shown in the final groups in Figure 4, our approach identifies a set of perceptual groups (blocks), including the multitab at the top and the list in the main area. Each list item is a combined widget of some non-text and text widgets (highlighted in the same color). These perceptual groups encode the comprehension of the GUI structure into higher-order layout blocks that can be used in further processing and applications.

3.5 Continuity - Detection Error Correction

The GUI widget detector may make two types of detection errors - missed widgets and misclassified widgets. Missed widgets means that the detector fails to detect some GUI elements on the GUI (e.g., the bottom-right icon in Figure 5(a)). Misclassified widgets refer to the widgets that the detector reports the wrong type, for example,

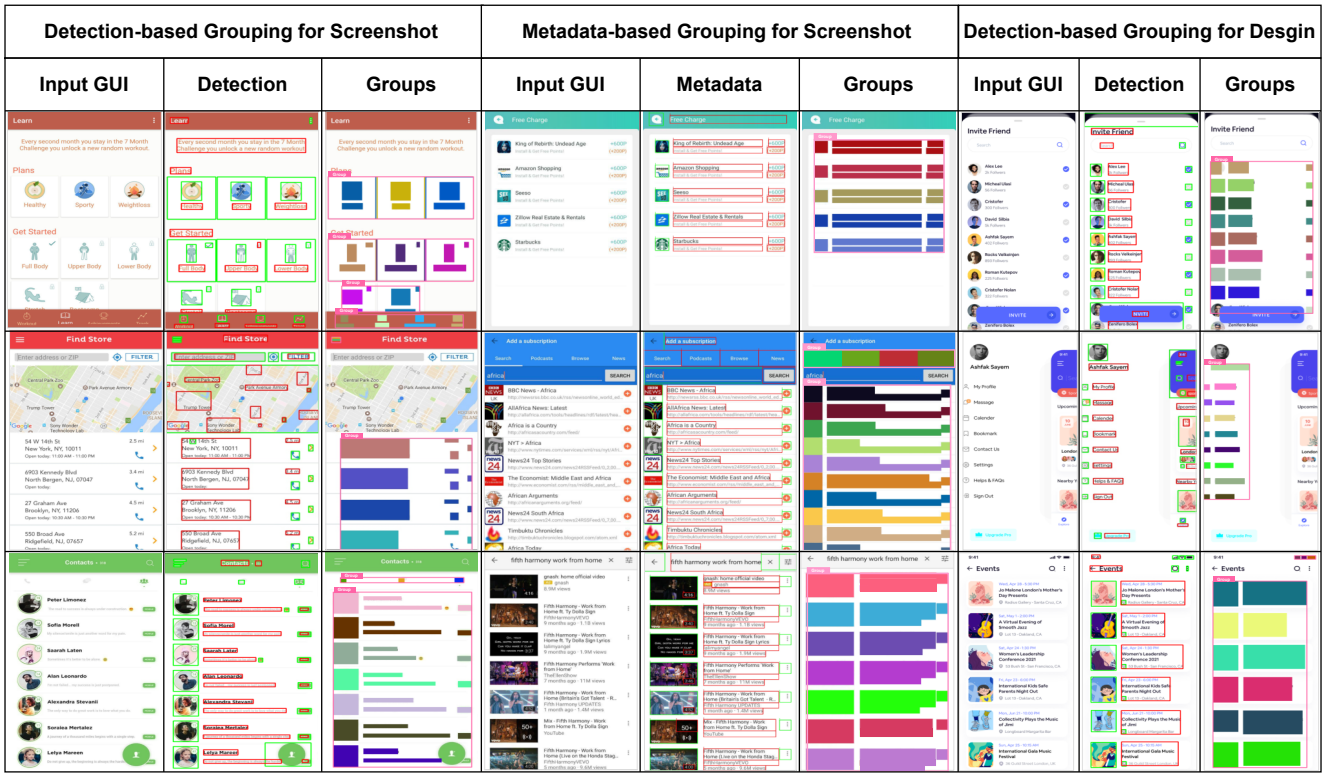

Figure 6: Examples of GUI widget detection and perceptual grouping results (red box - text widget, green box - non-text widget, pink box - perceptual group). Metadata-based means grouping the ground-truth widgets directly from GUI metadata.

the top-right small icon (i.e., a non-text widget) in the middle card in Figure 5(b) is misclassified as a text widget due to an OCR error.

It is hard to recognize and correct these detection errors from the individual widget perspective, but applying the Gestalt continuity principle to expose such widget detection errors by contrasting widgets in the same perceptual groups can mitigate the issue. The continuity principle states that elements arranged in a line or curve are perceived to be more related than elements not in a line or curve [41]. Thus, some detection errors are likely to be spotted if a GUI area or a widget aligns with all the widgets in a perceptual group in a line but is not gathered into that group.

Our approach tries to identify and fix missed widgets as follows. It first inspects the subgroups of widgets in a perceptual group and checks if the widgets in the subgroups are consistent in terms of the number and relative position of the contained widgets. If a subgroup contains fewer widgets than its counterparts, then the approach locates the inconsistent regions by checking the relative positions and areas of other subgroups' widgets. Next, the approach crops the located UI regions and uses the widget detector upon the cropped regions with relaxed parameters (i.e. double of the minimum area threshold for valid widgets) to try to identify the missed widget, if any. For example, the tiny icon at the bottom right in Figure 5(a) is missed because its area is so small that the detector regards it as a noisy region and hence discards it in the initial detection. By analyzing the resulting perceptual group and its composition, our approach finds that seven of the eight subgroups have two widgets (marked in the same color), while the subgroup at the bottom right has only one widget. It crops the area that may contain the missed widget according to the average sizes and average relative positions of the two widgets in the other seven subgroups. The missed tiny icon can be recovered by detecting the widget with the relaxed valid-widget minimum area threshold in the missing area.

Our approach uses the exact mechanism that contrasts the subgroups to identify the misclassified widgets, but here it focuses on widget type consistency. As shown in Figure 5(b), our approach groups the three cards in a perceptual group. By contrasting the widgets in the three cards, it detects that the middle card has text widgets at the top right corner, while the other two cards have a non-text widget at the same relative positions. Based on the continuity principle, our approach re-classifies the top-right widget in the middle card as non-text with a majority-win strategy.

4 EVALUATION

We evaluate our approach in two steps: (1) examine the accuracy of our enhanced version of UIED and compare it with the original UIED [21]; (2) examine the accuracy of our widget perceptual grouping approach and compare it with the state-of-the-art heuristic-based method Screen Recognition [58].

4.1 Accuracy of GUI Widget Detection

Compared with the original UIED [21], our GUI widget detector uses the latest Google OCR tool and improve the text and non-text

Table 1: Overall results of widget detection (IoU > 0.9)

	Our Enhanced Revision			*Original UIED*		
Type	**Precision**	**Recall**	**F1**	**Precision**	**Recall**	**F1**
Non-Text	**0.589**	**0.823**	**0.687**	0.431	0.469	0.449
Text	**0.678**	0.693	**0.686**	0.402	**0.720**	0.516
All Widgets	**0.580**	**0.680**	**0.626**	0.490	0.557	0.524

widget merging by container analysis. We evaluate GUI widget detection from the three perspectives: text widget detection, non-text widget detection and the final widget results after merging. To be consistent with the evaluation setting in the UIED paper [21], we run experiments on the same Rico dataset of Android app GUIs [36] and regard the detected widgets whose intersection over union (IoU) with the ground truth widget is over 0.9 as true positive. The ground-truth widgets are the leaf widgets (i.e., non-layout classes) extracted from the GUI's runtime view hierarchy.

Table 1 shows the widget detection performance of the enhanced and the original UIED. Our enhanced version achieves a much higher recall (0.823) for non-text widgets than the original UIED (0.469), and meanwhile, it also improves the precision (0.589 over 0.431). This significant improvement is due to the more intelligent container-aware merging of text and non-text widgets by our enhanced version. As the original UIED is container-agnostic, it erroneously discards many valid widgets contained in other widgets as noise. For GUI text, the Google OCR tool used in the enhanced version achieves much higher precision (0.678) than the EAST model used in the original UIED (0.402), with a slight decrease in recall (0.693 versus 0.720). The improvements in both text and non-text widgets result in a much better overall performance (0.626 F1 by the enhanced version versus 0.524 by the original UIED).

4.2 Perceptual Grouping Performance

We evaluate our perceptual grouping approach on both Android app GUIs and UI design prototypes. Figure 6 shows some perceptual grouping results by our approach. These results show our approach can reliably detect GUI widgets and infer perceptual groups for diverse visual and layout designs.

4.2.1 Datasets. Our approach simulates how humans perceive the GUI structure and segment a GUI into blocks of widgets according to the Gestalt principles of grouping. To validate the recognized blocks, we build the ground-truth dataset from two sources: Android apps and UI design prototypes. The ground truth annotates the widget groups according to the GUI layout and the widget styles and properties as shown in Figure 7.

Android App GUI Dataset The ground truth of widget groups can be obtained by examining the layout classes used to group other widgets in the implementations. However, as shown in Figure 2, the layout classes do not always correspond to the perceptual groups of GUI widgets. Therefore, we cannot use the GUI layout classes directly as the ground truth. Instead, we first search the GUIs in the Rico dataset of Android app GUIs [36] that use certain Android layout classes that may contain a group of widgets (e.g., ListView, FrameLayout, Card, TabLayout). Then we manually examine the candidate GUIs to filter out those whose use of layout classes has obvious violations against the Gestalt principles. Furthermore, the Rico dataset contains many highly-similar GUI screenshots for an application. To increase the visual and layout diversity of GUIs in our dataset, we limit up to three distinct GUI screenshots per application. Distinction is determined by the number and type of GUI widgets and the GUI structure. We obtain 1091 GUI screenshots from 772 Android applications. Using this dataset, we evaluate both detection-based and metadata-based grouping. Detection-based grouping processes the detected widgets, while metadata-based grouping uses the widgets obtained from the GUI metadata (i.e., assumes the perfect widget detection).

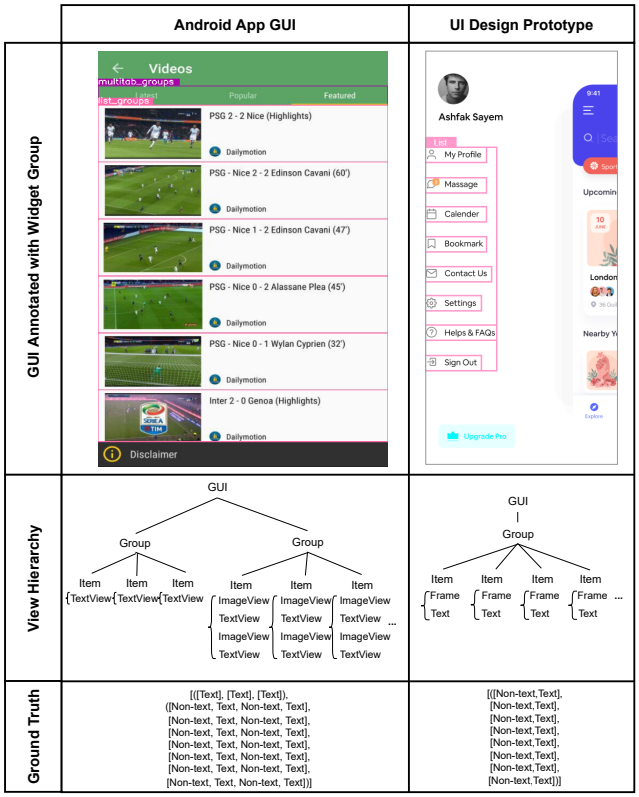

Figure 7: Examples of Android app GUI and UI design prototype, view hierarchy and ground truth

UI Design Prototypes We collect 20 UI design prototypes shared on a popular UI design website (Figma [4]). These UI design prototypes are created by professional designers for various kinds of apps and receive more than 200 likes. This small set of UI design prototypes demonstrates how professional designers structure the GUIs and group GUI widgets from the design rather than the implementation perspective. As a domain-independent tool, Figma supports only elementary visual elements (i.e., text, image and shape). Designers can create any widgets using these elementary visual elements. Due to the lack of explicit and uniform widget metadata in the Figma designs, we evaluate only the detection-based grouping on these UI design prototypes.

4.2.2 Metrics. The left part of Figure 7 shows an example in our Android app GUI dataset. We see that the layout classes (e.g., ListView, TabLayout) in the view hierarchy map to the corresponding perceptual groups. In our dataset, specific layout classes are generalized

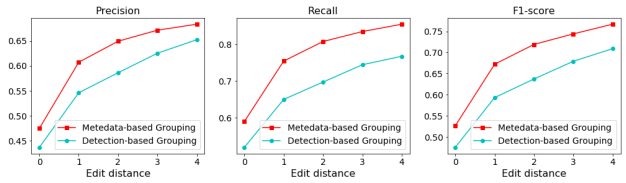

Figure 8: Performance at different edit distance thresholds

to blocks, as we only care about generic perceptual groups in this work. Following the work [16] for generating GUI component hierarchy from UI image, we adopt the sequence representation of a GUI component hierarchy. Through depth-first traversal, a view hierarchy can be transformed into a string of GUI widget names and brackets ("[]" and "()" corresponding to the blocks). This string represents the ground-truth perceptual groups of an app GUI. As discussed in Section 2.1, perceptual grouping is based on the widgets' positional and visual properties while the actual classes of non-text widgets are not necessary. Thus, the ground-truth string only has two widget types: Text and Non-text. Specifically, it converts TextView in the view hierarchy to text and all other classes as Non-text. Similarly, as shown in the right part of Figure 7, the designers organize texts and non-text widgets (images or shape compositions referred to as frames) into a hierarchy of groups. Based on the design's group hierarchy, we output the ground-truth string of perceptual groups. The perceptual groups "perceived" by our approach are output in the same format for comparison.

We compute the Levenshtein edit distance between the two strings of a ground-truth block and a perceived block. The Levenshtein edits inform us of the specific mismatches between the two blocks, which is important to understand and analyze grouping mistakes. If the edit distance between the ground-truth block and the perceived block is less than a threshold, we regard the two blocks as a candidate match. We determine the optimal matching between the string of ground-truth blocks and the string of the perceived blocks by minimizing the overall edit distance among all candidate matches. If a perceived group matches a ground-truth group, it is a true positive (TP), otherwise a false positive (FP). If a ground-truth group does not match any perceived group, it is a false negative (FN). Based on the matching results, we compute: (1) precision (*TP/(TP+FP)*) (2) recall (*TP/(TP+FN)*), and (3) F1-score (*(2*precision*recall) / (precision+recall)*)

4.2.3 Performance on Android App GUIs. We experiment with five edit distance thresholds (0-4). The distance 0 means the two blocks have the perfect match, and the distance 4 means as long as the unmatched widgets in the two blocks are no more than 4, the two blocks can be regarded as a candidate match. As shown in Figure 8, for detection-based grouping, the precision, recall and F1-score is 0.437, 0.520 and 0.475 at the distance 0. As the distance threshold increases (i.e., the matching criterion relaxes), the precision, recall and F1-score keeps increasing to 0.652, 0.776 and 0.709 at the distance threshold 4. As shown in Figure 8 and Table 2, the metadata-based grouping with the ground-truth GUI widgets achieves a noticeable improvement over the detection-based grouping with the detected

Table 2: Performance comparison (edit distance≤1)

Widgets	Approach	#Bock	Precision	Recall	F1
Metadata	Our Approach	1,465	**0.607**	**0.754**	**0.672**
	Screen Recog	1,038	0.131	0.116	0.123
Detection	Our Approach	1,260	0.546	0.650	0.593
	Screen Recog	992	0.103	0.083	0.092

widgets, in terms of all three metrics, especially for recall. This suggests that improving GUI widget detection will positively improve the subsequent perceptual grouping.

As the examples in Figure 6 show, our approach can not only accurately process GUIs with clear structures (e.g., the first row), but it can also process GUIs with large numbers of widgets that are placed in a packed way (e.g., the second and third rows). Furthermore, our approach is fault-tolerant to GUI widget detection errors to a certain extent, for example, the second row of detection-based grouping for screenshot and design. The map and the pushed-aside partial GUI result in many inaccurately detected GUI widgets in these two cases. However, our approach still robustly recognizes the proper perceptual groups.

We compare our approach with the heuristic-based grouping method (Screen Recognition) proposed in Zhang et.al. [58] (which received the distinguished paper award at CHI2021). The results in Table 2 shows that Screen Recognition can hardly handle visually and structurally complicated GUIs based on a few ad-hoc and rigid heuristics. Its F1 score is only 0.092 on the detected widgets and 0.123 on the ground-truth widgets. This is because its heuristics are designed for only some fixed grouping styles such as cards and multi-tabs. In contrast, our approach is designed to fulfil generic Gestalt principles of grouping.

We manually inspect the grouping results by our approach against the ground-truth groups to identify the potential improvements. Figure 9 presents four typical cases that cause the perceived groups to be regarded as incorrect. For the detection-based grouping, the major issue is GUI widget over-segmentation (a widget is detected as several widgets) or under-segmentation (several widgets are detected as one widget). In the first row, the detector segments the texts on the right side of the GUI into several text and non-text widgets. As indicated by the same color in the Grouping Result column, our approach still successfully partitions the widgets on the same row into a block, and recognizes the large group containing these row blocks. But as shown in the Group Comparison column, one widget in the second, third and fourth detected blocks do not match those in the corresponding ground-truth blocks. In the second row, the GUI widget detector merges close-by multi-line texts as a single text widget, while these text widgets are separate widgets in the ground truth. Again, our approach recognizes the overall perceptual groups correctly, but the widgets in the corresponding blocks do not completely match.

While using the ground-truth widgets from the GUI metadata to mitigate the GUI widget misdetection, the grouping results see the improvement but suffer from two other problems. First, the widgets in the metadata contain some widgets that are visually occluded or hidden. The third row in Figure 9 illustrates this problem, where some widgets are actually occluded behind the menu on the left, but they are still available in the runtime metadata and are

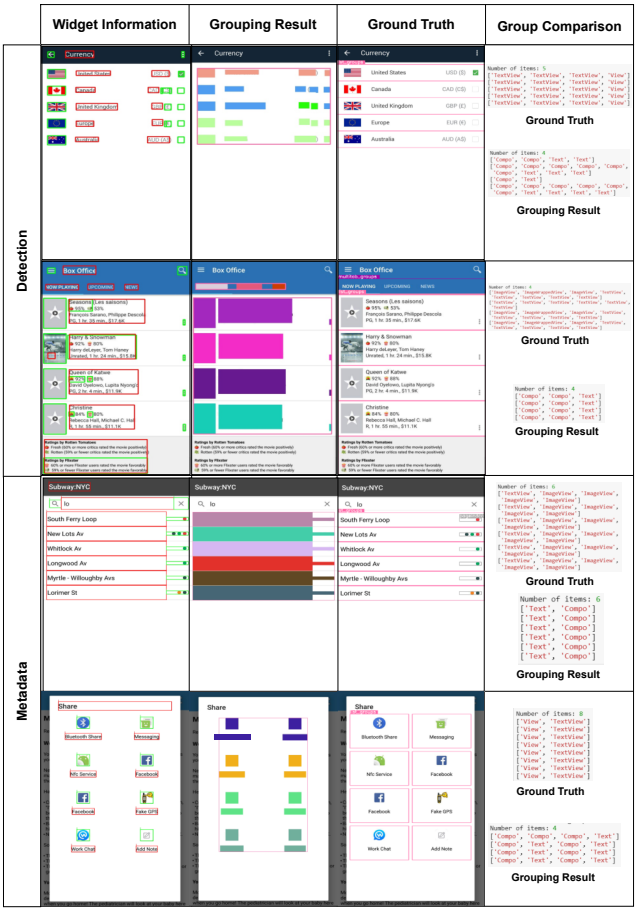

Figure 9: Typical causes of grouping mistakes (red box - text widget, green box - non-text widget, pink box - perceptual group, red dashed box - unmatched ground-truth widget)

extracted as the ground-truth widgets. This results in a completely incorrect grouping. The issue of widget occlusion or modal window could be mitigated as follows: train an image classifier to predict the presence of widget occlusion or modal window, then follow the figure-ground principle [1] to separate foreground modal window from the background, and finally detect the perceptual groups on the separated model window. Second, alternative ways exist to partition the widgets into groups. For example, for the GUI in the fourth row, the ground truth contains eight blocks, each of which has one image and one text while our grouping approach partitions these blocks into four rows of a large group, and each row contains two blocks (as indicated by the same color in Grouping Result). Perceptually, both ways are acceptable but the group differences cause the grouping result by our approach to be regarded as incorrect.

4.2.4 Performance on UI Design Prototypes. Tested on the 20 UI design prototypes, our approach achieves the precision of 0.750, the recall 0.818 and the F1-score 0.783. The third column in Figure 6 shows some results of our grouping approach for the UI design prototypes, where we see it is able to infer the widget groups well for different styles of GUI designs. GUI widget detection is more robust on UI design prototypes due to the more accurate GUI widget detection, which leads to the improvement of the subsequent grouping of detected widgets. As shown in Figure 6, the widgets in a UI design prototype is usually scattered, while the real app GUIs are packed. Both GUI widget detection and perceptual grouping become relatively easier on less packed GUIs.

4.2.5 Processing Time. As the GUI widget grouping can be used as a part of various automation tasks such as automated testing, the runtime performance can be a concern. We record the processing time while running our approach over the dataset to get a sense of its efficiency. Our experiments run on a machine with Windows 10 OS, Intel i7-7700HQ CPU, and 8GB memory. Our approach comprises two major steps: widget detection and perceptual grouping. We improved and refactored the original UIED to boost the run performance of the widget detection, and now it takes an average of 1.1s to detect the widgets in a GUI, which significantly exceeds the original UIED that takes on average 9s per GUI. The grouping process is also efficient, which takes an average of 0.6s to process a GUI. In total, the average processing time of the entire approach is 1.7s per GUI image. Furthermore, as our approach does not involve any deep learning techniques, it does not require advanced computing support such as GPU.

5 RELATED WORK

Our work falls into the area of reverse-engineering the hidden attributes of GUIs from pixels. There are two lines of closely related work: GUI widget detection and GUI-to-text/code generation.

GUI widget detection is a special case of object detection [31, 40, 49]. Earlier work [38] uses classic computer vision (CV) algorithms (e.g., Canny edge and contour analysis) to detect GUI widgets. Recently, White et al. [52] apply a popular object detection model YOLOv2 [40] to detect GUI widgets in GUI images for random GUI testing. Feng et al. [15] apply Faster RCNN [31] to obtain GUI widgets from app screenshots and construct a searchable GUI widget gallery. Chen et al. [20] proposed an approach to complete icon labeling in mobile applications. A recent study by Xie et al. [21] shows that both classic CV algorithms and recent deep learning models have limitations when applied to GUI widget detection, which has different visual characteristics and detection goals from natural scene object detection. They design a hybrid method UIED inspired by the figure-ground [1] characteristic of GUI, which achieves the start-of-the-art performance for GUI widget detection.

GUI-to-text/code generation also receives much attention. To improve GUI accessibility, Chen et al. [18] propose a transformer-based image captioning model for producing labels for icons. To implement GUI view hierarchy, REMAUI [38] infers three Android-specific layouts (LinearLayout, FrameLayout and ListView) based on hand-craft rules to group widgets. Recently, Screen Recognition [58] develops some heuristics for inferring tabs and bars. However, these heuristic-based widget grouping methods cannot handle visually and structurally complicated GUI designs (e.g., nested perceptual groups like a grid of cards). Alternatively, image captioning models [34, 51] have been used to generate GUI view hierarchy from GUI images [12, 16]. Although these image-captioning based methods get rid of hard-coded heuristics, they suffer from GUI data availability and quality issues (as discussed in Introduction

and illustrated in Figure 2). These methods also suffer from code redundancy and no explicit image-code traceability issues (see Section 6.2). The perceptual groups recognized by our approach could help to address these issues.

None of the existing GUI widget detection and GUI-to-code approaches solve the perceptual grouping problem in a systematic way as our approach does. ReDraw [37] and FaceOff [59] solves the layout problem by finding in the codebase the layouts containing similar GUI widgets. Some other methods rely on source code or specific layout algorithm (e.g., Android RelativeLayout) to synthesize modular GUI code or layout [11, 13] or infer GUI duplication [54]. All these methods are GUI implementation-oriented, and hard to generalize for other application scenarios such as UI design search, UI automation, robotic GUI testing or accessibility enhancement. In contrast, our approach is based on domain-independent Gestalt principles and is application-independent, so it can support different downstream SE tasks (see Section 6).

In the computer vision community, some machine learning techniques [33, 55, 57] have been proposed to predict structure in the visual scene, i.e., so-called scene graphs. These techniques can infer the relationships between objects detected in an image and describe these relationships by triplets (<subject, relation, object>). However, such relationship triplets cannot represent complex GUI widget relations in perceptual groups. Furthermore, these techniques also require sufficient high-quality data for model training, which is a challenging issue for GUIs.

6 PERCEPTUAL GROUPING APPLICATIONS

Our perceptual grouping method fills in an important gap for automatic UI understanding. Perceptual groups, together with elementary widget information, would open the door to some innovative applications in software engineering domain.

6.1 UI Design Search

UI design is a highly creative activity. The proliferation of UI design data on the Internet enables data-driven methods to learn UI designs and obtain design inspirations [14, 23, 36]. However, this demands effective UI design search engines. Existing methods often rely on the GUI metadata, which limits their applicability as most GUI designs exist in only pixel format. GalleryDC [15] builds a gallery of GUI widgets and infer elementary widget information (e.g., size, primary color) to help widget search. Unfortunately, this solution does not apply to the whole and complex UIs. Chen et al. [17] and Rico [36] use image auto-encoder to extract image features through self-supervised learning, which can be used to find visually similar GUI images. However, the image auto-encoder encodes only pixel-level features, but is unaware of GUI structure, which is very critical to model and understand GUIs. As such, given the GUI in the left of Figure 2, these auto-encoder based methods may return a GUI like the one on the right of Figure 2, because both GUIs have rich graphic features and some textural features. Unfortunately, such search results are meaningless, because they bear no similarity in terms of GUI structure and perceptual groups of GUI widgets. Our approach can accurately infer perceptual groups of GUI widgets from pixels. Based on its perceptual grouping results, a UI design search would become structure-aware, and finds not only visually but also structurally similar GUIs. For example, a structure-aware UI design search would return a GUI like the one in 2nd-row-1st-column of Figure 6 for the left GUI in Figure 2.

6.2 Modular GUI-to-Code Generation

Existing methods for GUI-to-code generation either use hand-craft rules or specific layout algorithms to infer some specific implementation layout [13, 38], or assume the availability of a codebase to search layout implementations [37, 59]. Image-captioning based GUI-to-Code methods [12, 16, 29, 30] are more flexible as they learn how to generate GUI view hierarchy from GUI metadata (if available). However, the nature of image captioning is to just describe the image content, but it is completely unaware of GUI structure during the code generation. As such, the generated GUI code is highly redundant for repetitive GUI blocks. For example, for the card-based GUI design in Figure 1(a), it will generate eight pieces of repetitive code, one for each of the eight cards. This type of generated code is nothing like the modular GUI code developers write. So it has little practicality. Another significant limitation of image captioning is that the generated GUI layouts and widgets have no connection to the corresponding parts in the GUI image. For a GUI with many widgets (e.g., those in the 2nd and 3rd rows in Figure 6), it would be hard to understand how the generated code implements the GUI. With the support of our perceptual grouping, GUI-to-code generation can encapsulate the widget grouping information into the code generation process and produce much less redundant and more modular, reusable GUI code (e.g., extensible card component).

6.3 UI Automation

Automating UI understanding from pixels can support many UI automation tasks. A particular application of UI automation in software engineering is automatic GUI testing. Most existing methods for automatic GUI testing rely on OS or debugging infrastructure [5, 10, 32, 43]. In recent years, computer vision methods have also been used to support non-intrusive GUI testing [27, 28, 39, 52]. However, these methods only work at the GUI widget level through either traditional widget detection [38] or deep learning models like Yolo [40]. Furthermore, they only support random testing, i.e., random interactions with some widgets. Some studies [19, 22, 56] show that GUI testing would be more effective if the testing methods were aware of more likely interactions. They propose deep learning methods to predict such likely interactions. However, the learning is a completely black box. That is, they can predict where on the GUI some actions could be applied, but they do not know what will be operated and why so. Our approach can inform the learning with higher-order perceptual groups of GUI widgets so that the model could make an explainable prediction, for example, scrolling is appropriate because this part of GUI displays a list of repetitive blocks. It may also guide the testing methods to interact with the blocks in a perceptual group in an orderly manner, and ensure all blocks are tested without unnecessary repetitions. Such support for UI automation would also enhance the effectiveness of screen readers which currently heavily rely on accessibility metadata and use mostly elementary widget information.

7 CONCLUSION AND FUTURE WORK

This paper presents a novel approach for recognizing perceptual groups of GUI widgets in GUI images. The approach is designed around the four psychological principles of grouping - connectedness, similarity, proximity and continuity. To the best of our knowledge, this is the first unsupervised, automatic UI understanding approach with a systematic theoretical foundation, rather than relying on ad-hoc heuristics or model training with GUI metadata. Through the evaluation of both mobile app GUIs and UI design prototypes, we confirm the high accuracy of our perceptual grouping method for visually and structurally diverse GUIs. Our approach fills the gap of visual intelligence between the current widget-level detection and the whole-UI level GUI-to-code generation. As a pixel-only and application-independent approach, we envision our approach could enhance many downstream software engineering tasks with the visual understanding of GUI structure and perceptual groups, such as structure-aware UI design search, modular and reusable GUI-to-code generation, and layout-sensitive UI automation for GUI testing and screen reader. Although our current approach achieves very promising performance, it can be further improved by dealing with widget occlusion or modal window. Moreover, we will investigate semantic grouping that aims to recognize both interaction and content semantics of perceptual groups.

ACKNOWLEDGEMENTS

This research was partially funded by Data61-ANU Collaborative Research Project. Chunyang is partially supported by Facebook/Meta Gift grant.

REFERENCES

[1] [n.d.]. 7 Gestalt Principles of Visual Perception: Cognitive Psychology for UX: UserTesting Blog. https://www.usertesting.com/blog/gestalt-principles#figure
[2] [n.d.]. Free website builder: Create a free website. http://www.wix.com/
[3] [n.d.]. Get started on Android with TalkBack - Android Accessibility Help. https://support.google.com/accessibility/android/answer/6283677?hl=en
[4] [n.d.]. the collaborative interface design tool. https://www.figma.com/
[5] [n.d.]. UI/Application Exerciser Monkey : Android Developers. https://developer.android.com/studio/test/monkey#:~:text=TheMonkeyisaprogram,arandomyetrepeatablemanner.
[6] [n.d.]. Vision AI | Derive Image Insights via ML | Cloud Vision API. https://cloud.google.com/vision/
[7] 2006. *"Gestalt psychology". Britannica concise encyclopedia.* Britannica Digital Learning.
[8] 2021. Accessibility - Vision. https://www.apple.com/accessibility/vision/
[9] 2021. Gestalt psychology. https://en.wikipedia.org/wiki/Gestalt_psychology#cite_note-1
[10] UIAutomator, 2021. https://developer.android.com/training/testing/ui-automator.
[11] Mohammad Bajammal, Davood Mazinanian, and Ali Mesbah. 2018. Generating Reusable Web Components from Mockups. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE 2018)*. New York, NY, USA, 601–611. https://doi.org/10.1145/3238147.3238194
[12] Tony Beltramelli. 2018. pix2code: Generating code from a graphical user interface screenshot. In *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems*. 1–6.
[13] Pavol Bielik, Marc Fischer, and Martin Vechev. 2018. Robust Relational Layout Synthesis from Examples for Android. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 156 (Oct. 2018), 29 pages. https://doi.org/10.1145/3276526
[14] Chunyang Chen, Sidong Feng, Zhengyang Liu, Zhenchang Xing, and Shengdong Zhao. 2020. From lost to found: Discover missing ui design semantics through recovering missing tags. *Proceedings of the ACM on Human-Computer Interaction* 4, CSCW2 (2020), 1–22.
[15] Chunyang Chen, Sidong Feng, Zhenchang Xing, Linda Liu, Shengdong Zhao, and Jinshui Wang. 2019. Gallery D.C.: Design Search and Knowledge Discovery through Auto-Created GUI Component Gallery. *Proc. ACM Hum.-Comput. Interact.* 3, CSCW, Article 180 (Nov. 2019), 22 pages. https://doi.org/10.1145/3359282
[16] Chunyang Chen, Ting Su, Guozhu Meng, Zhenchang Xing, and Yang Liu. 2018. From UI Design Image to GUI Skeleton: A Neural Machine Translator to Bootstrap Mobile GUI Implementation. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 665–676. https://doi.org/10.1145/3180155.3180240
[17] Jieshan Chen, Chunyang Chen, Zhenchang Xing, Xin Xia, Liming Zhu, John Grundy, and Jinshui Wang. 2020. Wireframe-based UI Design Search through Image Autoencoder. *ACM Transactions on Software Engineering and Methodology* 29, 3 (Jul 2020), 1–31. https://doi.org/10.1145/3391613
[18] Jieshan Chen, Chunyang Chen, Zhenchang Xing, Xiwei Xu, Liming Zhut, Guoqiang Li, and Jinshui Wang. 2020. Unblind Your Apps: Predicting Natural-Language Labels for Mobile GUI Components by Deep Learning. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. 322–334.
[19] Jieshan Chen, Amanda Swearngin, Jason Wu, Titus Barik, Jeffrey Nichols, and Xiaoyi Zhang. 2022. Extracting Replayable Interactions from Videos of Mobile App Usage. *arXiv preprint arXiv:2207.04165* (2022).
[20] Jieshan Chen, Amanda Swearngin, Jason Wu, Titus Barik, Jeffrey Nichols, and Xiaoyi Zhang. 2022. Towards Complete Icon Labeling in Mobile Applications. In *CHI Conference on Human Factors in Computing Systems*. 1–14.
[21] Jieshan Chen, Mulong Xie, Zhenchang Xing, Chunyang Chen, Xiwei Xu, Liming Zhu, and Guoqiang Li. 2020. Object detection for graphical user interface: old fashioned or deep learning or a combination? *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Nov 2020). https://doi.org/10.1145/3368089.3409691
[22] Christian Degott, Nataniel P. Borges Jr., and Andreas Zeller. 2019. Learning User Interface Element Interactions. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 296–306. https://doi.org/10.1145/3293882.3330569
[23] Biplab Deka, Zifeng Huang, Chad Franzen, Joshua Hibschman, Daniel Afergan, Yang Li, Jeffrey Nichols, and Ranjitha Kumar. 2017. Rico: A Mobile App Dataset for Building Data-Driven Design Applications. In *Proceedings of the 30th Annual Symposium on User Interface Software and Technology (UIST '17)*.
[24] Dribbble. [n.d.]. Discover the world's Top Designers & Creatives. https://dribbble.com/
[25] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. 1996. A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining* (Portland, Oregon) *(KDD'96)*. AAAI Press, 226–231.
[26] Michael W. Eysenck and Marc Brysbaert. 2018. Fundamentals of Cognition. (2018). https://doi.org/10.4324/9781315617633
[27] Sidong Feng and Chunyang Chen. 2022. GIFdroid: An Automated Light-weight Tool for Replaying Visual Bug Reports. (2022).
[28] Sidong Feng and Chunyang Chen. 2022. GIFdroid: automated replay of visual bug reports for Android apps. In *Proceedings of the 44th International Conference on Software Engineering*. 1045–1057.
[29] Sidong Feng, Minmin Jiang, Tingting Zhou, Yankun Zhen, and Chunyang Chen. 2022. Auto-Icon+: An Automated End-to-End Code Generation Tool for Icon Designs in UI Development. *ACM Transactions on Interactive Intelligent Systems (TiiS)* (2022).
[30] Sidong Feng, Suyu Ma, Jinzhong Yu, Chunyang Chen, TingTing Zhou, and Yankun Zhen. 2021. Auto-icon: An automated code generation tool for icon designs assisting in ui development. In *26th International Conference on Intelligent User Interfaces*. 59–69.
[31] Ross Girshick. 2015. Fast R-CNN. In *The IEEE International Conference on Computer Vision (ICCV)*.
[32] Jiaqi Guo, Shuyue Li, Jian-Guang Lou, Zijiang Yang, and Ting Liu. 2019. SARA: self-replay augmented record and replay for Android in industrial cases. In *Proceedings of the 28th acm sigsoft international symposium on software testing and analysis*. 90–100.
[33] Boris Knyazev, Harm de Vries, Cătălina Cangea, Graham W. Taylor, Aaron Courville, and Eugene Belilovsky. 2020. Graph Density-Aware Losses for Novel Compositions in Scene Graph Generation. arXiv:2005.08230 [cs.CV]
[34] Yann Lecun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *Nature* 521, 7553 (2015), 436–444. https://doi.org/10.1038/nature14539
[35] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2019. Humanoid: A Deep Learning-Based Approach to Automated Black-box Android App Testing. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1070–1073. https://doi.org/10.1109/ASE.2019.00104
[36] Thomas F. Liu, Mark Craft, Jason Situ, Ersin Yumer, Radomir Mech, and Ranjitha Kumar. 2018. Learning Design Semantics for Mobile Apps. In *The 31st Annual ACM Symposium on User Interface Software and Technology* (Berlin, Germany) *(UIST '18)*. ACM, New York, NY, USA, 569–579. https://doi.org/10.1145/3242587.3242650

[37] Kevin Moran, Carlos Bernal-Cárdenas, Michael Curcio, Richard Bonett, and Denys Poshyvanyk. 2020. Machine Learning-Based Prototyping of Graphical User Interfaces for Mobile Apps. *IEEE Transactions on Software Engineering* 46, 2 (2020), 196–221. https://doi.org/10.1109/TSE.2018.2844788

[38] Tuan Anh Nguyen and Christoph Csallner. 2015. Reverse Engineering Mobile Application User Interfaces with REMAUI. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering* (Lincoln, Nebraska) *(ASE '15)*. IEEE Press, 248–259. https://doi.org/10.1109/ASE.2015.32

[39] Ju Qian, Zhengyu Shang, Shuoyan Yan, Yan Wang, and Lin Chen. 2020. RoScript: A Visual Script Driven Truly Non-Intrusive Robotic Testing System for Touch Screen Applications. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 297–308. https://doi.org/10.1145/3377811.3380431

[40] Joseph Redmon and Ali Farhadi. 2018. YOLOv3: An Incremental Improvement. *CoRR* abs/1804.02767 (2018). arXiv:1804.02767 http://arxiv.org/abs/1804.02767

[41] Andy Rutledge. 2009. Gestalt Principles of Perception - 3: Proximity, Uniform Connectedness, and Good Continuation. http://andyrutledge.com/gestalt-principles-3.html

[42] S1T2. 2021. Apply crap to design: S1T2 blog. https://s1t2.com/blog/step-1-generously-apply-crap-to-design

[43] Onur Sahin, Assel Aliyeva, Hariharan Mathavan, Ayse Coskun, and Manuel Egele. 2019. Randr: Record and replay for android applications via targeted runtime instrumentation. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 128–138.

[44] Sketch. [n.d.]. https://www.sketch.com/

[45] Sternberg and Robert. 2003. *Cognitive Psychology Third Edition*. Thomson Wadsworth.

[46] Herb Stevenson. [n.d.]. Emergence: The Gestalt Approach to Change. http://www.clevelandconsultinggroup.com/articles/emergence-gestalt-approach-to-change.php

[47] Tesseract-Ocr. [n.d.]. tesseract-ocr/tesseract: Tesseract Open Source OCR Engine (main repository). https://github.com/tesseract-ocr/tesseract

[48] Thalion. 2020. Ui Design in practice: Gestalt principles. https://uxmisfit.com/2019/04/23/ui-design-in-practice-gestalt-principles/

[49] J. R. R. Uijlings, K. E. A. van de Sande, T. Gevers, and A. W. M. Smeulders. 2013. Selective Search for Object Recognition. *International Journal of Computer Vision* 104, 2 (01 Sep 2013), 154–171. https://doi.org/10.1007/s11263-013-0620-5

[50] UserTesting. 2019. 7 Gestalt Principles of Visual Perception: Cognitive Psychology for UX: UserTesting Blog. https://www.usertesting.com/blog/gestalt-principles#proximity

[51] Oriol Vinyals, Alexander Toshev, Samy Bengio, and Dumitru Erhan. 2015. Show and Tell: A Neural Image Caption Generator. arXiv:1411.4555 [cs.CV]

[52] Thomas D. White, Gordon Fraser, and Guy J. Brown. 2019. Improving Random GUI Testing with Image-based Widget Detection. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. ACM, New York, NY, USA, 307–317. https://doi.org/10.1145/3293882.3330551

[53] Mulong Xie, Sidong Feng, Zhenchang Xing, Jieshan Chen, and Chunyang Chen. 2020. UIED: a hybrid tool for GUI element detection. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1655–1659.

[54] Rahulkrishna Yandrapally, Andrea Stocco, and Ali Mesbah. 2020. Near-Duplicate Detection in Web App Model Inference. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 186–197. https://doi.org/10.1145/3377811.3380416

[55] Jianwei Yang, Jiasen Lu, Stefan Lee, Dhruv Batra, and Devi Parikh. 2018. Graph R-CNN for Scene Graph Generation. arXiv:1808.00191 [cs.CV]

[56] YazdaniBanafsheDaragh. [n.d.]. Deep-GUI: Towards Platform-Independent UI Input Generation with Deep Reinforcement Learning. *UC Irvine* ([n. d.]). https://escholarship.org/uc/item/3kv1n3qk

[57] Rowan Zellers, Mark Yatskar, Sam Thomson, and Yejin Choi. 2018. Neural Motifs: Scene Graph Parsing with Global Context. arXiv:1711.06640 [cs.CV]

[58] Xiaoyi Zhang, Lilian de Greef, Amanda Swearngin, Samuel White, Kyle Murray, Lisa Yu, Qi Shan, Jeffrey Nichols, Jason Wu, Chris Fleizach, Aaron Everitt, and Jeffrey P. Bigham. 2021. Screen Recognition: Creating Accessibility Metadata for Mobile Applications from Pixels. arXiv:2101.04893 [cs.HC]

[59] Shuyu Zheng, Ziniu Hu, and Yun Ma. 2019. FaceOff: Assisting the Manifestation Design of Web Graphical User Interface. In *Proceedings of the Twelfth ACM International Conference on Web Search and Data Mining* (Melbourne VIC, Australia) *(WSDM '19)*. Association for Computing Machinery, New York, NY, USA, 774–777. https://doi.org/10.1145/3289600.3290610

[60] Xinyu Zhou, Cong Yao, He Wen, Yuzhi Wang, Shuchang Zhou, Weiran He, and Jiajun Liang. 2017. EAST: an efficient and accurate scene text detector. 5551–5560.