

## Article

# UIGuider: Detecting Implicit Design Guidelines Using a Domain Knowledge Graph Approach

Bo Yang \*  and Shanping Li

College of Computer Science and Technology, Zhejiang University, Hangzhou 310012, China; shan@zju.edu.cn

\* Correspondence: imyb@zju.edu.cn

**Abstract:** Mobile application developers must adhere to a set of design guidelines to ensure consistency in graphical user interfaces (GUIs) and strive for best practices. Violating these widely accepted design guidelines can negatively impact user experience and diminish an application's market value. Although a few explicit design guidelines outline specific user interface layouts to avoid, substantial design knowledge remains scattered across various web pages. This implicit design knowledge includes design guidelines for UI component usage scenarios, anatomy, fine-grained categorizations, etc. Manually inspecting design guideline violations is knowledge-intensive and time-consuming, demanding high levels of expertise. To address this, we propose UIGuider, a novel approach to automatically constructing a component-based design knowledge graph using multimodal data. UIGuider can discern valuable text containing design guidelines, extract component and concept entities, and establish relationships among them. Our experiments demonstrate the effectiveness and usefulness of UIGuider in automatically detecting implicit design guidelines and constructing domain knowledge graphs. Additionally, the result presentation of a design violation detection tool is optimized. The results of user studies confirm that the use of the knowledge graph and real-world app datasets enhances the overall usability of the tool.

**Keywords:** knowledge graph; GUI testing; violation detection



**Citation:** Yang, B.; Li, S. UIGuider: Detecting Implicit Design Guidelines Using a Domain Knowledge Graph Approach. *Electronics* **2024**, *13*, 1210. <https://doi.org/10.3390/electronics13071210>

Academic Editor: Enrique Romero-Cadaval

Received: 16 January 2024

Revised: 10 March 2024

Accepted: 22 March 2024

Published: 26 March 2024



**Copyright:** © 2024 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

## 1. Introduction

The graphical user interface (GUI), as the entry point for users to interact with mobile applications, is one of the most crucial features of apps. Effective GUI design directly influences app quality and user experience (UX), both of which are essential for user retention and successful software promotion [1].

The iterative process of GUI design and development presents enduring challenges, even for experienced designers and developers. Designers must adhere to numerous design principles [2], ensuring that the resulting prototype conforms to standards of consistency, aesthetics, and efficacy [3]. Rapid advancements in front-end technologies [4] have introduced various visual effects to GUI design, including the integration of streaming media and dynamic particle animations, which enhance UX but also pose challenges for developers. Developers also face challenges from the abundance of development frameworks and conflicting development specifications [5], further increasing demand for developers with specialized expertise.

These pressures on developers and designers make the occurrence of UI design smells in mobile application GUIs inevitable. A UI design smell is any visual characteristic within the GUI, indicating issues that violate UI design principles and subsequently impact the user experience. The definition of design smell is inspired by that of code smell [6]. Common design smells are shown in Figure 1 (with smells marked in red boxes). In Figure 1a, the text superimposed upon the background image makes the title in the top bar indiscernible. In Figure 1b, applying tabs and bottom navigation simultaneously may confuse users, as it remains unclear which tab controls the current content. In Figure 1c,

some destinations in the navigation drawer have icons whereas others do not. Icons should be used for all destinations or none. In Figure 1d, 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 in Google Material Design documentation [7].

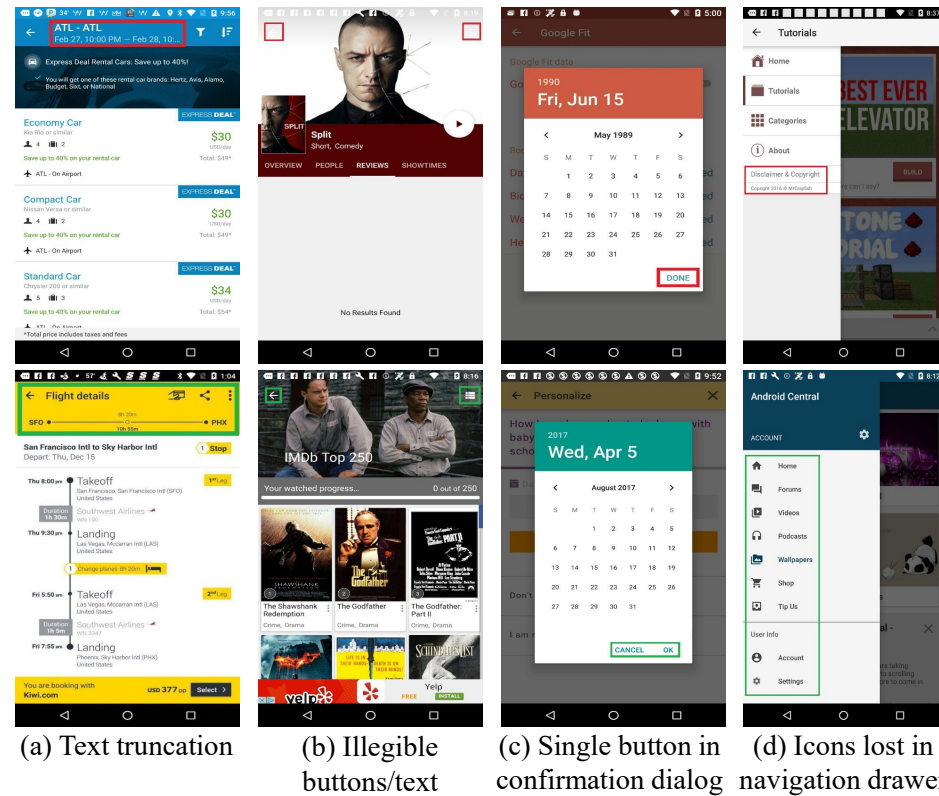


**Figure 1.** UI design smells (1st row) vs. non-smell UIs (2nd row) (issues highlighted in red boxes).

Some studies, such as Zhao et al. [8], Yang et al. [9], and Zhang et al. [10], have proposed detecting UI violations by analyzing the “do not” examples explicitly provided in the Material Design guidelines. In contrast to the explicit design knowledge provided through “do not” guidelines and the corresponding examples (as shown above the dashed line in Figure 2), Material Design still encompasses a large amount of implicit design knowledge. This includes UI component usage scenarios, anatomy, and design knowledge within fine-grained classifications of UI components (as shown below the dashed line in Figure 2). They effectively guide UI design (e.g., avoiding the use of bottom navigation in user preferences), but they are not explicitly labeled as “do not”; hence, they are referred to as implicit design guidelines. The results of a survey by Gao et al. [11] indicated that implicit design guidelines strongly impact on the competitiveness of applications. Developers have to check their UIs to analyze the potential risks of violating the design guidelines.

There are three key challenges in detecting UI violations of the design guidelines. The first challenge is collecting and extracting implicit design knowledge from guideline documents. These documents often lack standardized formatting and writing conventions, so extracting relevant information is difficult. Furthermore, these documents typically contain complex multimodal semantic information, adding to the complexity of the task. Second, implicit design knowledge sometimes includes principled dictums (e.g., “banners communicate a succinct message”). This can cause ambiguity (e.g., “how long must a message be to be considered succinct?”), confusing designers and developers when examining the design guidelines. Third, determining whether design violations exist requires a systematic approach to evaluating the alignment between the UI and the design guidelines. The design examples used to explain the design guidelines (e.g., Figure 2) are

often quite different from the actual UI designs to be checked (e.g., Figure 1), hindering their direct comparison to determine guideline violations.

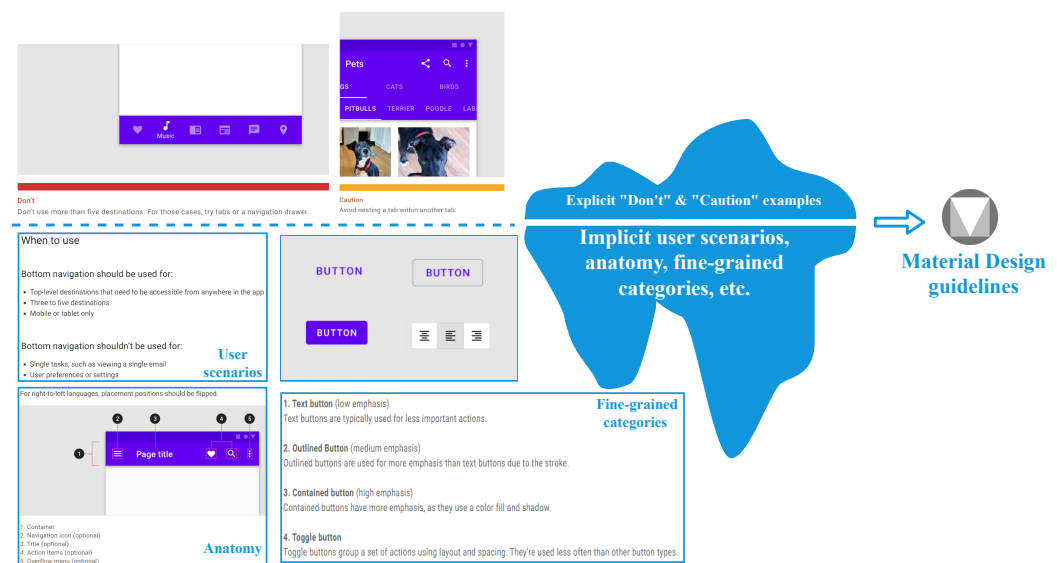


**Figure 2.** Examples of explicit and implicit design guidelines in Material Design. The part above the dashed line shows explicit design guidelines labeled as “do not do that” and “caution” for using the bottom navigation and tab, respectively. The part below the dashed line shows implicit design guidelines without labels, such as the usage scenarios for using bottom navigation, the anatomy of a top bar, and the fine-grained classifications of buttons.

With this paper, we provide the following contributions:

1. To the best of our knowledge, we are the first to propose a keyword- and learning-based approach to automatically capturing and associating knowledge entities from design guidelines.
2. We constructed a design knowledge graph based on Google Material Design and developed a method for detecting violations of design guidelines using this graph.
3. Our evaluation demonstrates high accuracy for building the graph. User studies confirmed that the effectiveness of the knowledge graph and that real app datasets enhance the overall usability of the tool.

The rest of the paper is structured as follows: Section 2 provides a review of the related studies. Section 3 details our technical framework and methodology. Section 4 discusses the study questions and the results of our quantitative and usefulness evaluation. Section 5 presents our conclusions and recommendations for future work.

## 2. Related Studies

In this section, we discuss (1) knowledge graphs for software engineering and (2) design violation detection.

### 2.1. Knowledge Graphs for Software Engineering

Knowledge graphs (KGs) in software engineering have three main application categories: (1) KG-based Q&A and conversational systems; (2) KG-powered search and recommendation systems; (3) KG-driven domain-specific applications [12]. The major knowledge sources span the entire software development lifecycle, including requirement documents, test cases, bug reports, log data, Stack Overflow posts, source code, etc. Recently, researchers [13,14] have constructed KGs using software documentation and existing test cases. With these KGs, searches can be performed according to the domain language of software requirements, and corresponding test cases can be automatically generated or

recommended. Therefore, KGs can enhance the efficiency of software testing [15] and the reuse rate of software test cases [16,17] to a certain extent.

For bug localization, deep-learning-based models [18] are used to extract semantic information from code. KG-based methods [19] are used to extract interrelations of source code because code is more structural and logical than natural language. KGs can mine deep semantic and structural relationships from multisource software data, assisting developers in organizing and understanding bug-related knowledge [20].

In the era of large language models (LLMs), the utility and significance of KGs have become more apparent. LLMs are known for their lack of factual knowledge, often hallucinating factual incorrectness [21,22]. In contrast to black-box models such as LLMs, meticulously crafted KGs offer more dependable, structured, and explainable knowledge [12,23].

These KG-based methods rely on documents with established writing conventions and standards. To the best of our knowledge, we are the first to construct a design knowledge graph from a design system documentation and apply it to downstream tasks.

## 2.2. Design Violation Detection

Many studies have focused on checking compatibility [24], presentation failure [25], usability [26], and accessibility [27] in GUI testing.

Approaches related to our study involve the examination and report presentation failures on websites [25,28,29], such as X-PERT [30], Crosscheck [31], and WebDiff [32], which aim to check for cross-browser compatibility and consistency. GUI testing techniques generate test cases by simulating real user interactions to trigger app functions. However, they cannot be used to evaluate the visual effects of mobile applications. Early studies such as that by Mahajan and Shneiderman [33] indicated touchstone consistency as one factor affecting the usability of an application. Their study revealed that an inconsistent GUI can decrease user performance by 10% to 25%. Issa et al. [34] reported that visual defects represent 16–33% of reported defects in four open-source systems. Despite the strong impact of GUI on end-user experience and acceptance, it has received limited attention in the GUI-testing literature.

As with any manual testing, testing using checklists of visual properties is time-consuming and error-prone. In addition, the visual effects in apps tend to be more subjective and mutable as catering to all tastes is difficult. As such, visual defects are more difficult to detect than nonvisual defects. Guideline documents from Apple [35], Google [7], or Microsoft [36] (also known as design systems) allow developers to use consistent formatting to direct GUI designs. Consequently, studies [8,37,38] have recently paid considerable attention to visual GUI testing and violation detection.

Moren et al. [37] proposed GVT, an automated method for reporting GUI design violations in mobile apps. They also analyzed the distribution of different industrial design violations. The main difference between GVT and our approach is that they have different purposes and usages. GVT takes mockups as its input and verifies whether the GUI of a mobile app aligns with its intended design, assisting developers in fixing errors that deviate from the original design.

Zhao et al. [8] proposed a deep-learning-based computer vision technique to lint GUI animation effects against material designs' "do not" guidelines. They trained a GUI animation feature extractor to learn the temporal-spatial features of GUI animation and used k-nearest neighbor (KNN) to identify the most similar violation of GUI animation. They considered nine "do not" guidelines and labeled 1000 GUI screenshots for each guideline.

In comparison to the above methods, our approach begins with design documents, extracts UI component design knowledge, and lints the design of mobile apps. Designers, developers, and product managers can use our approach for design linting throughout the mobile application development process.

### 3. Approach

Our approach consists of four main steps, as shown in Figure 3: basic graph construction (Section 3.1), textual design knowledge graph construction (Section 3.2), visual design knowledge graph construction (Section 3.3), and knowledge fusion of the two knowledge graphs (Section 3.4).

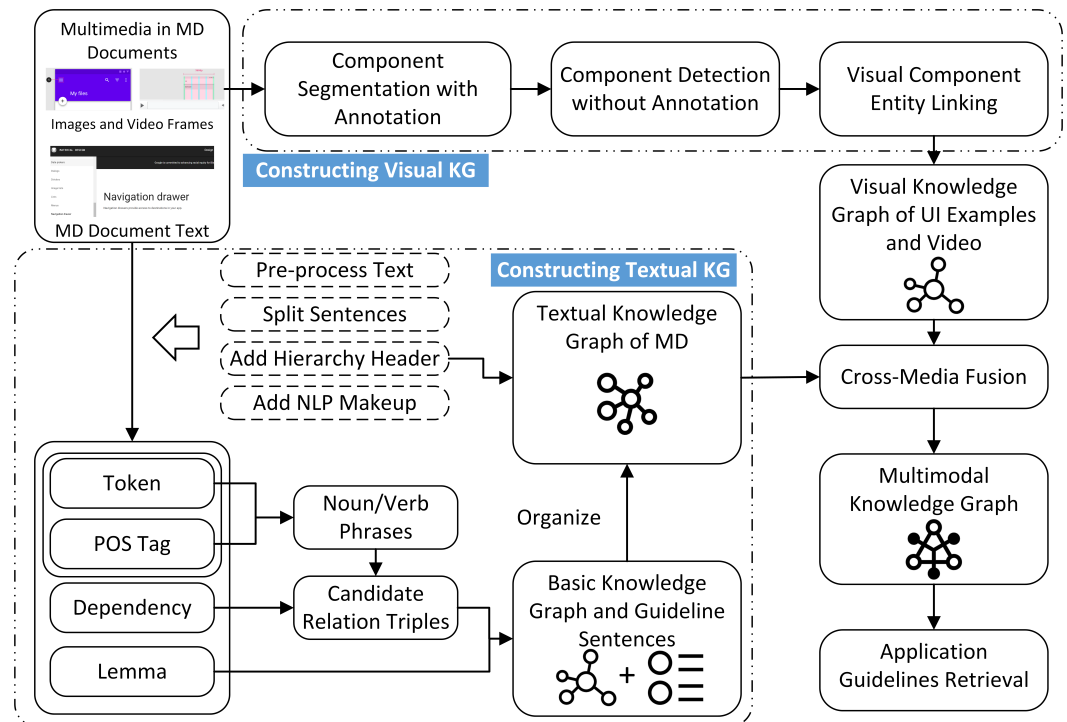


**Figure 3.** Overview of proposed approach to constructing multimodal design knowledge graph.

#### 3.1. Basic Graph Construction

The purpose of constructing a basic knowledge graph is to preprocess the textual data and establish the foundational structure of the knowledge graph, leveraging the inherent hierarchy of web pages.

The Material Design documentation [7] organizes design knowledge in web pages. Using EasySpider [39], we extracted all textual contents from the “Components” section of web pages and filtered content formats not considered in this study, such as hyperlinks and code snippets. We ruled out the contents under the “Resources”, “Development”, “Component-Theme”, and “Component-Standard” sections as they are irrelevant to the UI component design knowledge under our study. In the preprocessing stage, we split the text content into sentences and enriched them in two steps:

1. Recording the source information for each sentence, such as the section title of the source page from which the sentence is extracted, and the sequence of sentences in the source web page;
2. Making the sentences readable without context by replacing the pronouns at the beginning of the sentences (e.g., it, they, and this) with the title of the source web page.

In the knowledge entity extraction phase of the basic knowledge graph, we use the chapter titles in the Material Design documentation as the entity nodes. Based on semantics, these are further divided into component (COM) nodes (e.g., button, top bar, and navigation drawer) and concept (CONC) nodes (e.g., usage, anatomy, and behavior). Further knowledge entity categories and classification methods are introduced in Section 3.2.3. Table 1 lists the knowledge entity categories, abbreviations, and examples. In the relationship extraction phase of the basic knowledge graph, we adopt a bottom-up, depth-first text processing strategy. When processing the content of a chapter, the node extracted from the

chapter title serves as the parent node, and the child nodes and relationships derived from this content are linked to the parent node. The reason is that the hierarchical structure of the documentation itself is also a semantic classification based on expert knowledge.

**Table 1.** Category of knowledge entity in the textual design knowledge graph.

| Category        | Abbreviation | Example                         |
|-----------------|--------------|---------------------------------|
| Component       | COM          | button, app bar, dialog, etc.   |
| Concept         | CON          | usage, anatomy, placement, etc. |
| Style/Attribute | SA           | padding, stroke, fill, etc.     |
| Behavior        | BE           | scale, click, etc.              |
| Status          | ST           | enable, disabled, focused, etc. |
| Other           | DO           | device, etc.                    |

### 3.2. Textual Design Knowledge Graph Construction

#### 3.2.1. Guideline Sentences Identification

The process of identifying guideline sentences is a critical step in distilling design knowledge. We aimed to sift through the Material Design documentation, setting aside descriptive and directional text, and extract only the sentences with substantial design guidance significance. For instance, a directive sentence such as *“Buttons communicate actions that users can take”* is filtered out, whereas a sentence like *“By default, Material Design uses capitalized button text labels”* is considered as a design guideline. An expert keyword dictionary specific to the UI design field was compiled. This compilation leveraged hierarchical headers and keywords observed in a prior study [9].

For the purpose of classifying text, we employed TF-IDF [40] (Term Frequency-Inverse Document Frequency), an unsupervised method celebrated for its simplicity and effectiveness. By comparing the frequency of words within a document against their frequency across a corpus of documents, TF-IDF identifies the most distinctively frequent and important words. Moreover, when evaluating computational demands, TF-IDF proves more efficient than methods such as LLMs.

Term Frequency (TF) is a measure of how often a word appears in a document. If a word occurs multiple times, it is likely to be more significant. However, using term frequency alone could be misleading as some words might appear frequently but across all documents in the corpus, providing little discriminative power. Inverse Document Frequency (IDF) counterbalances the limitations of term frequency. It diminishes the weight of terms that occur very frequently in the corpus and increases the weight of terms that occur rarely. IDF is calculated as the logarithmically scaled inverse of the fraction of documents that contain the word. The rationale is straightforward: if a word appears in many documents, it is not a good differentiator and likely not as relevant to the particular meaning of any one document. By combining TF and IDF, the TF-IDF score for a term increases proportionally to the frequency of times a word appears in the document and is offset by the frequency of the word in the corpus. Words that are unique to a document will have a higher score; hence, the TF-IDF algorithm considers them as more important.

If a sentence shares a high TF-IDF score with a previously identified guideline sentence, it is also considered as a guideline. We drew our training dataset directly from the examples section of the Material Design documentation, as shown in Figure 2.

#### 3.2.2. NLP Preprocessing

This step transforms the natural text into a computer-understandable format, an operation known as NLP “markup” [41]. We employed widely-known tools, Stanford CoreNLP [42] and spaCy [43], integrated into our workflow to handle a set of NLP preprocessing tasks. Tokenization, part-of-speech (POS) tagging, dependency parsing, and lemmatization were meticulously performed to prepare the data for further processing.

Specifically, tokenization breaks down text into its constituent parts (words, phrases, symbols), enabling a more granular analysis. POS tagging assigns grammatical categories

(e.g., NNP for “Proper noun, singular”, VBZ for “Verb, third person singular present”, VBN for “Verb, past participle”) to each lexical unit within a sentence. This tagging facilitates the recognition of the syntactic functions of words. Figure 4 shows the POS tagging result of the definition sentence of the navigation drawer using spaCy. Dependency parsing is the task of analyzing the grammatical structure of a sentence and assigning syntactic relationships between words. The dependency parser provides token properties to navigate the generated dependency parse tree. The dependency attribute specifies the syntactic dependency relationship between the head token and its child token. Lemmatization, then, reduces words to their base or dictionary form, stripping away inflectional endings and delivering a more stripped-down data set for analysis. This step is pivotal in normalizing the linguistic variety for better consistency in subsequent NLP tasks.

The meticulous application of these NLP preprocessing techniques ensured that the text data were optimized for the ultimate goal, constructing a reliable and robust design knowledge graph that represents the distilled wisdom embedded in the Material Design documentation.



**Figure 4.** POS tagging results for the usage sentence of bottom navigation.

### 3.2.3. Noun/Verb Phrases Chunking and Classification

The aim of this step is to extract knowledge entities embedded within guideline sentences. By extracting noun phrases (NPs) and verb phrases (VPs), we classified them into six predefined categories: component, concept, style, behavior, state, and others. These categories with their abbreviations and examples are listed in Table 1.

We employed “tree parsing” [44], a full-text parsing technique heralded for its efficiency in extracting NPs and VPs. Table 2 shows a set of regular expressions used to identify the NPs and VPs, where “MD” is an abbreviation for “modal”. The symbol “?” indicates whether such a determinant exists; “\*” means zero or more determinants; “+” means must have such a determinant; “—” means continue to next row.

Using rule-based chunking, we slice the sentences’ tokens into the respective NPs and VPs. For example, from the sentence “Each destination is represented by an icon and an optional text label”, we extract three NPs (“each destination”, “an icon”, and “an optional text label”) and one VP (“is represented”), as shown in Figure 5.

Then, we identify the knowledge entity categories from the extracted NPs and VPs. Knowledge entity classification is crucial for associating the knowledge graph with understandable design guidelines. To ensure classification precision, three experienced workers were responsible for this work. They all had at least three years of experience in front-end development or software testing. They simultaneously labeled the same entities and compared the results afterward. If the same determination was reached, the result was considered final; if discrepancies arose, the knowledge entity was reviewed and discussed until all three annotators reached a consensus. Based on the labeling results, we created a set of rule-based knowledge entity rulers (EntityRuler) instructions in spaCy for identifying the categories of entities in the knowledge graph.

**Table 2.** Regular expression of chunks.

| Name | Regular Expression                                                                                                 |
|------|--------------------------------------------------------------------------------------------------------------------|
| NP   | (CD)* (DT)?(CD)*(JJ)*(CD)*(VBD   VBG)*(NN.)*(POS)*(CD)*—<br>(VBD   VBG)*(NN.)*(VBD   VBG)*(NN.)*(POS)*(CD)*(NN.)*+ |
| VP   | (MD)*(VB.)*+(CD)*(JJ)*(RB)*(JJ)*(VB.)*?(DT)?(IN*   TO)*+                                                           |

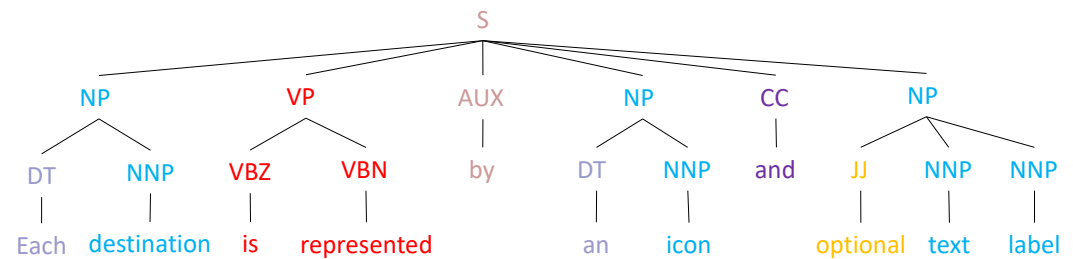


**Figure 5.** Example of chunking NPs and VPs from the usage sentence of bottom navigation. Different colors represent different tags.

### 3.2.4. Dependency Parsing and Relationship Triples Candidating

The subsequent phase enabled us to formulate semantic relationship triples such as (concept, relation VP, concept). Traditional rule-based chunking narrows in on morphologically structured chunks such as (NP, VP, NP); however, this approach misses a large amount of information and introduces noise into the relationship triples. To address the problem, we adopted the solution suggested in HDSKG chunking [41], which combines the rule-based chunking with the dependency parser. Based on the pre-chunked NPs and VPs, we used dependency parsing to determine the dependencies among all terms. To address the challenge of different expressions that convey the same meaning, we analyze the dataset and proposed various scenarios to deal with sentence variations. Further details can be found in Zhao et al. [41].

Through these advanced NLP techniques—combining rule-based chunking, tree parsing, and dependency parsing—our methodology improves the accuracy and integrity of the designed knowledge graph. We enable the structured representation of knowledge and its fluid integration into downstream NLP applications.

## 3.3. Visual Design Knowledge Graph Construction

The visual design knowledge graph branch of our approach takes images and video keyframes as its input and creates a single, coherent visual knowledge base. This branch relies on the same ontology as the textual design knowledge extraction branch. Similarly, the visual design knowledge graph consists of entity extraction, linking, and coreference modules.

### 3.3.1. Visual Entity Extraction and Linking

Once knowledge entities are added to the visual knowledge base, an attempt is made to link each knowledge entity to real-world knowledge entities from a curated background knowledge base. For UI components with annotation, we directly link the segmented image to the node corresponding to the annotation. To recognize UI components without annotation, manual refinement is performed based on the coarse-grained classification results, and the segmented images are linked to the fine-classified nodes in the knowledge graph.

**(1) Component detection with annotation.** In this step, we extend the bounding box given by the annotation and identify the text and primary color within the cropped box.

The edge extractor aims to detect the rendered edges of a UI component or the divider lines within it. Although the coordinates of a UI component always form a rectangle, its actual edge may not be a rectangle. We adopt Canny edge detection [45] to detect the edges surrounding or with a UI component. First, we preprocess the input UI image using image binarization. Next, we adopt a  $5 \times 5$  Gaussian filter to reduce noise. We filter the image with a Sobel kernel [46] in the horizontal and vertical directions to find the intensity gradient of the image. Finally, we use nonmaximum suppression [47] to refine the edge detection results. We require the pixels to have sufficient horizontal and vertical connectivity. The edge extractor outputs the start and end coordinates of the detected edges.

The color extractor aims to identify the primary color of the UI components and their constituent parts. To determine the primary color of the UI components, we adopt the HSV color space [38]. Unlike the RGB color model, which is hardware-oriented, the HSV model is user-oriented [48], based on the more intuitive appeal of combining hue, saturation, and

value elements to create a color. This method first converts an RGB color to the HSV color space. Each RGB color has a range of HSV values. The lower range is the minimum shade of the color that can be detected by the human eye, and the upper range is the maximum shade. For example, black is in the range of (0, 0, 0)–(184, 254, 80). Then, a mask is created for each primary color (i.e., black, blue, cyan, green, lime, magenta, red, and white). The mask is the area where the HSV value of the pixels matches the color between the lower and upper ranges of a primary color. Finally, the area of the mask in each color and the corresponding image occupancy ratio are calculated. The color with the maximum ratio is identified as the primary color of the UI component.

**(2) Component detection without annotation.** Our approach separately detects non-text UI components and UI texts. For UI text detection, we use the pre-trained state-of-the-art scene text detector EAST [49]. For non-text UI components' detection, we adopt the UI widget detection tool [50] to detect non-text UI widget regions (e.g., icons and buttons).

### 3.4. Cross-Media Knowledge Fusion

Given a set of multimodal documents consisting of textual data (e.g., UI design conventions and concepts) and visual data (e.g., UI examples and video keyframes), the textual and visual branches of the system take their respective modality data as input, extract knowledge elements, and create separate knowledge bases. These textual and visual knowledge bases share the same ontology but contain complementary information. Even coreferential knowledge elements that exist in both knowledge bases are not completely redundant because each modality has its own unique granularity. To leverage the complementary nature of the two modalities, we combine the two modality-specific knowledge bases into a single, coherent, multimodal knowledge base, where each knowledge element can be grounded in either or both modalities. Then, we link the matching textual and visual knowledge entities using a NIL cluster. Additionally, with visual linking (Section 3.3.1), we corefer cross-modal knowledge entities linked to the same background KG node.

### 3.5. Demonstration of UIGuider

A proof-of-concept prototype of UIGuider v1.0, combining our method with a previous method [51], was implemented. Figure 6 shows the detection of the three design guidelines of the navigation drawer. Given a UI screen, UIGuider conducts UI data parsing and extraction of atomic UI information; then, it decides which design guidelines to retrieve from the textual KG for validation. By generating a UI design smell report, UIGuider summarizes the list of design guidelines being violated and highlights the corresponding component area(s) on the input UI in a red or orange box, depending on the severity of the design guideline violation. For each guideline, UIGuider provides further details, recommends related design guidelines, and visualizes knowledge based on queries to the textual KG. The visual KG presents conformance and violation UI examples for each reported guideline violation. The corresponding UI component areas on the violation UI examples are also marked to draw attention. In our prior study [51], we conducted a demographic survey on Google Material Design guidelines. The results showed that nonanimation design guidelines accounted for the majority (86%). Therefore, UIGuider excluded design guidelines related to sound, motion, and interaction. Regarding the methods for motion, Zhao et al. [8] have dealt with nine items. The integration of animation-related design guideline detection, and the support for more detectable design guidelines, are our future work directions. Based on the design knowledge graph, UIGuider enhances detection reports through an interactive graph view of UI components, concepts, and design guidelines.

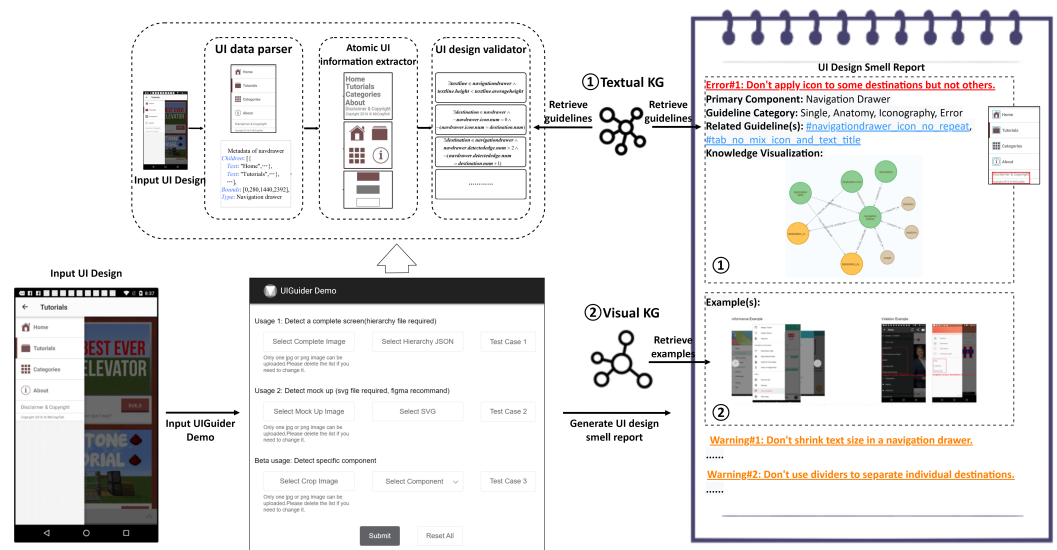


**Figure 6.** Overview of UIGuider demonstration (illustrating the detection of a navigation drawer as an example).

#### 4. Evaluation

We used EasySpider [39], a web crawler, to extract textual and images contents from the “Components” section of the Google Material Design website [7]. We ruled out the contents under the “Resources”, “Development”, “Component-Theme”, and “Component-Standard” sections as they are irrelevant to the UI component design knowledge in our study. The month of access was August 2023. Finally, we obtained about 3390 textual entries and 450 images from 30 web pages as the data source. The resulting design knowledge graph consisted of 571 design knowledge entities, including 116 COM entities and 455 CON entities. Our design knowledge graph contained 246 guideline relationships, containing all 126 explicit “do not” guidelines manually extracted in a prior study [9]. Our aim in this section was to answer two research questions (RQs):

1. **RQ1:** How effectively can UIGuider extract implicit design guidelines?
2. **RQ2:** To what extent can a design domain knowledge graph improve the effectiveness of design violation detection?

##### 4.1. Quantitative Evaluation: RQ1

In this section, we describe our evaluation of all our KG steps including guideline sentence identification, phrase chunking, relationship triple candidating, and KG fusion. We recruited three graduate students (unaffiliated with this study) with at least three years of front-end design and development experience to independently annotate the instances. We applied Cohen’s kappa [52] to assess the inter-rater agreement. For any instance where the two students were assigned different labels, the third student was assigned to supply an additional label to reconcile the conflict using a majority-vote strategy. Using the definitive labels and k-fold cross-validation technique [53], we evaluated the performance of each module by comparing the results to those of popular benchmark methods. All experiments were conducted in a computer with an Intel Xeon Gold 6226 2.7 GHz CPU, with 64 GB memory, and four 8 GB NVIDIA GeForce RTX 2080 Ti GPUs. The precision, recall, and F1 scores, as shown in Table 3, were calculated to quantify their performance.

##### 4.1.1. Guideline Sentences Identification

To assess the effectiveness of the TF-IDF module in guideline sentence classification, we conducted experiments and compared its performance against that of the SVM [54] algorithm. Two students determined whether the 261 sentences constituted guideline sentences based on the established definitions. Inter-rater agreement was assessed using Cohen’s kappa, which was 0.941, indicating near-perfect consensus. Given the limited

sample size, we employed a four-fold cross-validation technique to maximize data use and minimize overfitting. Based on the final labels, we found that our method outperformed SVM by approximately 20%, and identification precision was **0.94**. These errors primarily stemmed from triggering keywords in explanatory sentences, such as “text buttons do not distract from nearby content”. Such errors can be mitigated in the model by setting a minimum word co-occurrence threshold.

**Table 3.** Evaluation of results at each step of design knowledge graph construction.

| Task                               | Approach         | Precision | Recall | F1-Score |
|------------------------------------|------------------|-----------|--------|----------|
| Guideline sentences identification | Our approach     | 0.94      | 0.89   | 0.91     |
|                                    | SVM [54]         | 0.77      | 0.69   | 0.73     |
| Entity chunking                    | Our approach     | 0.90      | 0.90   | 0.90     |
|                                    | TaskNav [55]     | 0.51      | 0.78   | 0.62     |
| Relationship triple candidating    | Our approach     | 0.82      | 0.76   | 0.79     |
|                                    | TaskKG [56]      | 0.67      | 0.69   | 0.68     |
| KG fusion (UI component detection) | Our approach     | 0.87      | 0.92   | 0.89     |
|                                    | Faster RCNN [57] | 0.44      | 0.44   | 0.44     |

#### 4.1.2. Chunking and Knowledge Entity Classification

For NP and VP chunking, we compared our method to the rule-based phrase extraction method in TaskNav [55] via manual inspection. Our textual design knowledge graph contained 3390 phrases extracted from guideline sentences. Manually checking all phrases would have required substantial time and effort. Therefore, we adopted a statistical sampling method [58] to determine the minimum number (*MIN*) of phrases that needed to be analyzed containing a given noun or verb phrase. We set  $e = 0.05$  at a 95% confidence level for determining *MIN*. A total of 346 phrases were sampled. The main source of error was incorrect sentence input, resulting in meaningless extracted phrases. For correct sentence inputs, our method achieved a **0.90** precision for phrase extraction.

As a baseline, we input the original guidance sentences into TaskNav [55] and extracted noun and verb phrases, achieving a precision of **0.51**. TaskNav performed poorly compared with our method because it enforces rigid rules and treats some guidance sentences as irrelevant.

#### 4.1.3. Relationship Triple Candidating

For relationship triple candidating, we asked annotators to verify if all three slots of a triple were correct, only regarding those with fully accurate slots as correct. Similarly, we employed the statistical sampling method described in Section 4.1.2 for each attribute type to estimate extraction accuracy with a 5% error margin at 95% confidence. Two authors independently examined each sampled attribute to assess accuracy. They then discussed any discrepancies in their independent judgments to reach a consensus. The Cohen’s kappa between the two annotators was 0.952 (almost perfect agreement). Based on the final labels, the precision of guideline properties was **0.82**. Errors primarily stemmed from inaccurate UI component affiliation relationships and UI component names. For example, the pronoun in a sentence referred to a text button or container button that strongly impacts relationship extraction. Such errors could be avoided in preprocessing by employing more precise pronoun substitution rules.

#### 4.1.4. Visual Knowledge Entity Extraction and Knowledge Graph Fusion

Table 4 shows the overall visual knowledge entity detection performance for the non-text elements and all elements in the design examples with annotations and without annotations.

For KG fusion, we extracted 684 UI components and asked the annotators to verify if each visual knowledge entity linked to the correct textual knowledge entity in the textual

design KG. The Cohen's kappa between the two annotators was 0.897 (almost perfect agreement). Based on the final labels, the accuracy of KG fusion was **0.87**. The results of the error analysis showed that the errors were mainly due to inaccurate affiliation relationships between components and imprecise UI component aliases. For example, the app bar was incorrectly linked to the top app bar as we had improperly expanded the bar to the app bar. Such errors could be avoided by more precise coreference resolution rules. The results demonstrate that leveraging object detection algorithms can provide an appropriate foundation for the high-quality fusion of the textual design and visual design KGs.

**Table 4.** Statistics of visual knowledge entity detection based on role.

| Task               | Non-text Components |        | All Components |        |
|--------------------|---------------------|--------|----------------|--------|
|                    | Precision           | Recall | Precision      | Recall |
| With annotation    | 0.93                | 1      | 0.86           | 0.91   |
| Without annotation | 0.43                | 0.47   | 0.49           | 0.56   |

#### 4.2. Usefulness Evaluation: RQ2

Having confirmed the quantitative performance of our approach, we further investigated two aspects of the utility of UIGuider: **RQ2.1:** Can UIGuider help developers better understand design guideline violations? Does the performance differ with different task complexities? **RQ2.2:** How do developers rate the explanations, and how do these explanations affect user behavior when searching?

**Participants:** We recruited 12 front-end developers from different backgrounds, comprising undergraduate students, Ph.D. students, and developers from IT companies. These developers had at least 1 year of web and/or mobile app development experience. All participants used Google regularly. None of the developers were involved in the development of our study. According to their proficiency level and background, the participants were randomly halved to G1 and G2 (six developers per group).

**Dataset:** To evaluate UIGuider on real-world apps, we used the open-source Rico dataset [59]. The Rico dataset contains 64,759 different UI images from 9384 real Android applications. For each UI image, Rico provides a screen capture, a view hierarchy represented in JSON files, and various metadata of UI components, such as text, location, color, clickability, focus, or scrollability.

According to a prior study [9], 12.3% of Rico's UIs contain at least one confirmed design violation of Material Design guidelines. We selected violation examples from the design violation dataset according to the following criteria. First, we aimed to cover as diverse UI components as possible. The selected questions covered at least one UI component. Second, we aimed to cover diverse types of questions. Finally, We also needed to control the number of questions to be examined to avoid the fatigue of human workers. We constructed 15 independent tasks based on our observation of the design violation dataset, as shown in Table 5.

**Baseline:** To evaluate the usefulness of UIGuider, we used two baselines: Google, one of the largest search engines, and GPT-4 with Vision [60,61] (GPT-4V), a state-of-the-art multimodal LLM with vision capabilities. For knowledge usability, Google is the benchmark due to its widespread use and robustness as a global search engine. We used the accuracy of retrieving relevant information and the time cost as measure metrics.

Given that UIGuider is the first to automate knowledge extraction and detect design guidelines' violations, it was difficult to find a consistent and usable tool as a baseline. For example, Zhao et al. [8] considered some animation design guidelines, and the code of Zhang et al. [10] is not available and considered a few text-related design guidelines. Therefore, to find a capable baseline and explore possible future work directions, we chose GPT-4V as the baseline. GPT-4V is a multimodal-capability model developed by OpenAI. It can understand pictures, analyze the content of the pictures for users, and answer questions related to the pictures. The measure metric is whether GPT-4V could handle multimodal UI

violation detection tasks. Typically, to achieve the best performance, LLMs require prompt engineering. We also considered GPT-4V with prompt engineering (GPT-4VP) as a baseline. The prompt engineering formulates the role of GPT (professional UI reviewer) and the task (analyze UI and detect guideline violations); and the input includes not only UI screens but also the specific design guideline text to be checked. In GPT-4V without the prompt engineering, the design guideline text was replaced by the specific documentation link. The model we used was specifically GPT-4V (8K), released on 28 February 2024.

**Table 5.** The tasks used for the usefulness evaluation and the results of GPT-4V and GPT-4V with prompt engineering detecting the design violation in the task UI.

| No. | Task Description                                                                | Difficulty | GPT-4V | GPT-4VP |
|-----|---------------------------------------------------------------------------------|------------|--------|---------|
| 1   | Using too long a text label in a text button                                    | Easy       | ✗      | ✓       |
| 2   | An outlined button's width should not be narrower than the button's text length | Medium     | ✗      | ✓       |
| 3   | Truncating text in a top bar                                                    | Easy       | ✗      | ✗       |
| 4   | Wrapping text in a regular top bar                                              | Medium     | ✗      | ✗       |
| 5   | Applying icons to some destinations but not others in a navigation drawer       | Hard       | ✗      | ✗       |
| 6   | Shrinking text size in a navigation drawer                                      | Medium     | ✗      | ✗       |
| 7   | Mixing tabs that contain only text with tabs that contain only icons            | Easy       | ✗      | ✓       |
| 8   | Truncating labels in a tab                                                      | Easy       | ✗      | ✗       |
| 9   | Using a bottom navigation and tab together                                      | Easy       | ✗      | ✓       |
| 10  | Using a bottom navigation bar for fewer than three destinations                 | Medium     | ✗      | ✗       |
| 11  | Using dialog titles that pose an ambiguous question                             | Hard       | ✗      | ✗       |
| 12  | Using a single prominent button in a banner                                     | Hard       | ✗      | ✗       |
| 13  | Displaying multiple FABs on a single screen                                     | Medium     | ✗      | ✓       |
| 14  | Including less than two options in a speed dial of FAB                          | Hard       | ✗      | ✗       |
| 15  | Using primary color as the background color of text fields                      | Hard       | ✗      | ✗       |

**Procedure:** In the user study, we compared UIGuider against Google (the baseline). UIGuider was configured to not explicitly state the violated design guidelines but instead present them by highlighting nodes and edges in the knowledge graph. Participants were asked to judge whether and how many design violations existed in given UIs using different systems. Figure 7 shows an example of the procedure.

The first step of the experiment was an introduction to our study. Second, participants were provided an example task to familiarize themselves with the procedure and features of our systems. Third, each participant was asked to independently finish all 15 tasks. G1 and G2 acted as the experimental approach (Google + UIGuider) and the control approach (only Google) to separately finish each task, respectively. To prevent the impacts of participants' learning and fatigue, the order of tasks was rotated based on the Latin square [62].

Fourth, after completing the tasks, participants were asked to fill in the System Usability Scale (SUS) questionnaire [63], a quick and reliable tool to quantify the usability of a system. Finally, we conducted an open interview with all participants and collected their usage experiences and feedback.

**Result of RQ2.1:** Figure 8 shows the average answer accuracy and answer time of the six participants for each task. If the six participants answered all 19 violations, we would expect 114 ( $6 \times 19$ ) answered violations. The six participants reported a total of 117 violations, among which only 71 of the answered violations were true violations. Therefore, the overall answer accuracy was approximately 0.61. The experimental group (Google + UIGuider) outperformed the baseline group (Google only) in both answer accuracy (0.78 vs. 0.61)

and answer time (248 s vs. 307 s). UIGuider helped developers to more effectively find violations, saving an average of one minute of answer time.

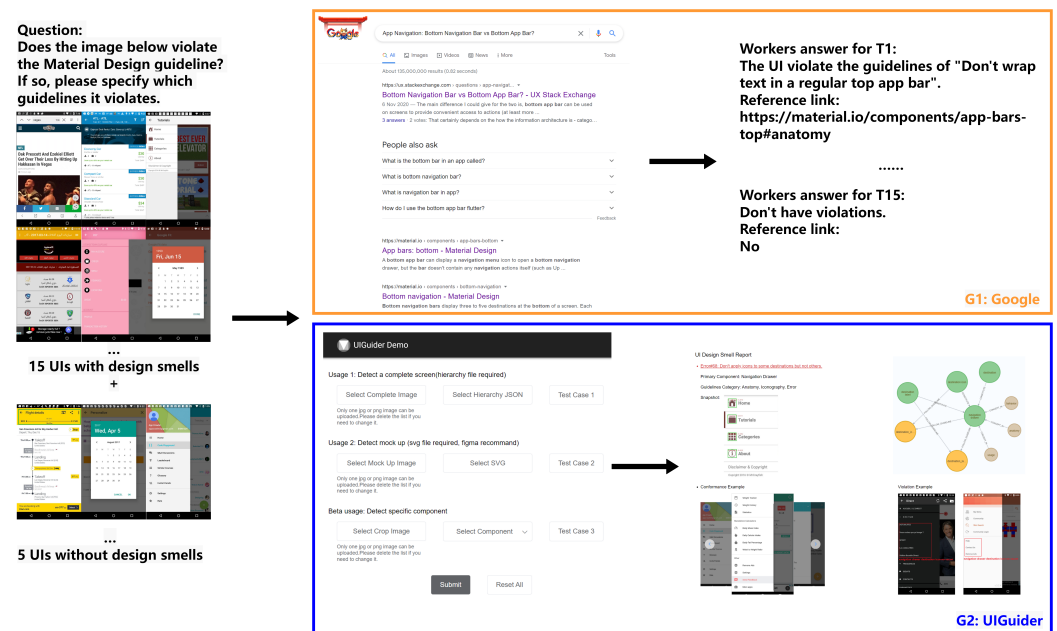


**Figure 7.** Comparison between UIGuider demonstration and Google Search for design knowledge.

Among the 15 tasks, only 1 task (T6) had all violations answered by all participants (i.e., accuracy = 1). Five tasks (T6, T7, T8, T9, and T11) had the same accuracy in both groups. Four of them (T7–T9 and T11) were constraint questions. Only T10 had a higher accuracy (0.8) in the baseline group (Google only). When topic UI components were distinct UI components (e.g., FAB) or widgets (e.g., button or text title), their design guidelines were easier to search manually, for example, using FAB with progress indicator (T7), having a two-line title in the top bar (T11), and having the same size of text field and button (T10).

For nine tasks (T1–T5 and T12–T15), the majority of the participants failed to give a more useful answer than our approach. In other words, the accuracy in the experimental group (Google + UIGuider) was higher than that in the baseline group (Google only). For T2, T12, T13, and T14, the accuracy of the baseline group was approximately 0.2 lower than that of the experimental group. Compared with these easy-to-answer tasks, these four questions required the examination of multiple design guidelines (e.g., T2) or detailed constraints (e.g., T13). For T1, T3, T4, T5, and T15, the gap in the accuracy between the baseline group and the experimental group was more than 0.4. This indicated that UIGuider effectively increased search performance. For example, T4 required comparing the bottom navigation and bottom bars. Our approach could generate a comparison table in terms of design guidelines about usage, anatomy, platform, motion, etc. T15 was a question about whether the bottom navigation and tab could be used at the same time. Our approach clearly stated that combining bottom navigation and tabs may cause confusion, as their relationship to the content may be unclear.

Regarding the answer time, UIGuider helped users save an average of one minute by reducing the time spent on acquiring and understanding design knowledge. Notably, T9, T10, and T11 required more answer time in the experimental group (Google + UIGuider) than in the baseline group (Google only). Among the three tasks, the accuracy for tasks T9 and T11 was the same; T10 was more precisely answered in the baseline group. This result demonstrates that humans are more likely to notice distinct UI components (e.g., FAB) and text styles. For example, T9 asked for advice on covering a tab with a snackbar, and T10 asked about having the same size of text field and button. For the other 12 tasks, UIGuider reduced answer time by 10 to 112 s.

In addition, the results revealed that GPT-4V using design documentation links as input could not detect any violations. Even though it gives some semantic explanations about the UI screen and design principles of UI components, most of the responses are unhelpful for violation detection tasks. The GPT-4V with prompt engineering succeeded in T1, T2, T7, and T9. These are all moderate and easy detection tasks. However, for the rest of the tasks, it responded that the violation could not be detected and required further clarification. Moreover, although it successfully completed the detection task (such as T2) in some instances, its performance varied in repeated experiments: sometimes violations could not be detected.

In conclusion, GPT-4V still has many shortcomings in detecting design violations, whereas UIGuider can already assist developers in finding violations more effectively, saving an average of one minute of response time.



**Figure 8.** Answer accuracy and answer time for each task.

**Result of RQ2.2:** Figure 9 summarizes the responses to the five usability questions and five learnability questions from the SUS questionnaire by the participants. Usability scores ranged from 4 to 4.5, with an average score of 4.28, indicating that participants agreed that UIGuider was easy to use and well-integrated. Learnability scores ranged from 1.75 to 2.75, with an average score of 2.10, further confirming that our tool is simple and consistent, without the need for high learning costs. In summary, the SUS evaluation results indicated that the participants appreciated the help provided by UIGuider with the tasks.

**User Feedback:** In the interviews after the study, participants noted that using UIGuider v1.0 saved them time when searching for and locating original design guideline pages at the beginning of the task, especially when the question was clear (T6). Concerning useful features, participants found the suggestions on conformance and violation examples to be the most useful because such comparisons intuitively illustrated the impact of violations on UX and potential directions for correction. However, participants found switching between Google and UIGuider to be inconvenient. The participants suggested that the graphical view of knowledge was moderately useful. The primary limitation was that it only displayed the general relationships between components, potentially limited by the density of information display, inhibiting a more detailed semantic explanation of the usage scenarios among them.

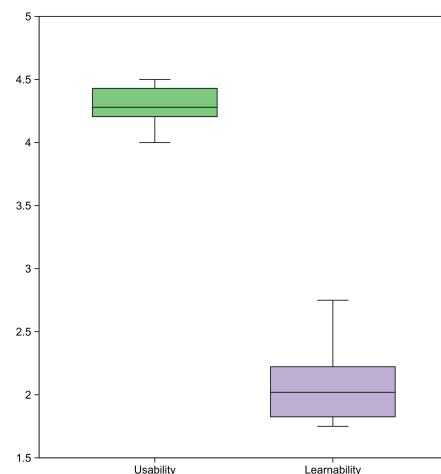


**Figure 9.** Distribution and average score of SUS results.

## 5. Conclusions and Future Work

In this paper, we present a framework for structuring multimodal knowledge such as design system documentation. We selected Google Material Design as a case study and developed a design violation detection tool named UIGuider, which can help recommend UI design guidelines and generate answer explanations. The results of our quantitative evaluation and user studies demonstrated that UIGuider can facilitate the answering of questions and reduce the time required for these tasks through summarizing explanations.

In the future, one of our primary directions is to broaden our method's accessibility and ease of use by developing practical tools like browser plugins. These plugins would operate as an interface layer between the developers and the underlying complexity of the design guidelines, offering instant recommendations and explanations directly within the design environment. As developers work on their projects, the plugin could detect potential design violations in real time and suggest improvements, effectively facilitate the design process, and enhance efficiency.

Another key area of future work is implementing a knowledge extractor based on LLMs. Leveraging the powerful abilities of LLMs in understanding context, inferring intent, and distinguishing subtle differences to identify and extract fragmented and challenging knowledge concealed in various design documents. By training these LLMs using specific design-related data corpora, the resulting knowledge extractor can provide more precise and context-relevant guidance, thereby elevating the functionality of UIGuider to a new level.

The integration of LLMs into UIGuider would allow for complex dissections of design paradigms, extending knowledge sources to diverse documentation beyond Google Material Design, such as Apple HIG [35] and blogs. Moreover, by analyzing design patterns and design guidelines that have received positive feedback across the design community, the system could offer more personalized and community-validated recommendations. This evolution would not only enhance the tool's recommendation engine with deeper insights but also refine its explanatory answers to accommodate a broader range of design queries.

In conclusion, built on the foundation laid by our existing multimodal knowledge structure framework and UIGuider, these advancements would assist designers and developers in creating aesthetically pleasing, fully functional, and user-centric designs with the aid of design guidelines.

**Author Contributions:** Conceptualization, B.Y. and S.L.; methodology, B.Y.; software, B.Y.; validation, B.Y.; formal analysis, B.Y.; investigation, B.Y.; resources, B.Y.; data curation, B.Y.; writing—original draft preparation, B.Y.; writing—review and editing, B.Y. and S.L.; visualization, B.Y.; supervision, S.L.; project administration, B.Y.; funding acquisition, S.L. All authors have read and agreed to the published version of the manuscript.

**Funding:** This research received no external funding.

**Institutional Review Board Statement:** Not applicable.

**Informed Consent Statement:** Not applicable.

**Data Availability Statement:** Data are contained within the article.

**Acknowledgments:** Research work mentioned in this paper is supported by State Street Zhejiang University Technology Center.

**Conflicts of Interest:** The authors declare no conflicts of interest.

## References

- Chen, J.; Chen, C.; Xing, Z.; Xia, X.; Zhu, L.; Grundy, J.; Wang, J. Wireframe-based UI design search through image autoencoder. *ACM Trans. Softw. Eng. Methodol.* **2020**, *29*, 1–31. [CrossRef]
- Nielsen, J. 10 Usability Heuristics for User Interface Design. 1994. Available online: <https://www.nngroup.com/articles/ten-usability-heuristics/> (accessed on 7 March 2024).
- Galitz, W.O. *The Essential Guide to User Interface Design: An Introduction to GUI Design Principles and Techniques*, 3rd ed.; Wiley: Hoboken, NJ, USA, 2007.
- 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, Perth, Australia, 13–14 July 2019.
- Technica, A. HTML5 Specification Finalized, Squabbling over Specs Continues. 2014. Available online: <https://arstechnica.com/information-technology/2014/10/html5-specification-finalized-squabbling-over-who-writes-the-specs-continues/> (accessed on 7 March 2024).
- Fowler, M.; Beck, K.; Brant, J.; Opdyke, W.; Roberts, D. *Refactoring: Improving the Design of Existing Code*; Addison Wesley: Boston, MA, USA, 1999.
- Google. Google Material Design. Available online: <https://m2.material.io/components/> (accessed on 7 March 2024).
- Zhao, D.; Xing, Z.; Chen, C.; Xu, X.; Zhu, L.; Li, G.; Wang, J. Seenomaly: Vision-based linting of GUI animation effects against design-do not guidelines. In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, Seoul, Republic of Korea, 23–29 May 2020.
- Yang, B.; Xing, Z.; Xia, X.; Chen, C.; Ye, D.; Li, S. Do not do that! Hunting down visual design smells in complex UIs against design guidelines. In Proceedings of the 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), Madrid, Spain, 22–30 May 2021.
- 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, New York, NY, USA, 23–28 August 2021.
- Gao, S.; Wang, Y.; Liu, H. UiAnalyzer: Evaluating whether the UI of apps is at the risk of violating the design conventions in terms of function layout. *Expert Syst. Appl.* **2024**, *239*, 122408. [CrossRef]
- Ji, S.; Pan, S.; Cambria, E.; Marttinen, P.; Yu, P.S. A survey on knowledge graphs: Representation, acquisition, and applications. *IEEE Trans. Neural Netw. Learn. Syst.* **2021**, *33*, 494–514. [CrossRef] [PubMed]
- Nayak, A.; Kesri, V.; Dubey, R.K. Knowledge graph based automated generation of test cases in software engineering. In Proceedings of the 7th ACM IKDD CoDS and 25th COMAD, Hyderabad, India, 5–7 January 2020.
- Ke, W.; Wu, C.; Fu, X.; Gao, C.; Song, Y. Interpretable test case recommendation based on knowledge graph. In Proceedings of the 2020 IEEE 20th International Conference on Software Quality, Reliability and Security (QRS), Macau, China, 11–14 December 2020.
- Wang, Y.; Bai, X.; Li, J.; Huang, R. Ontology-based test case generation for testing web services. In Proceedings of the Eighth International Symposium on Autonomous Decentralized Systems (ISADS'07), Sedona, AZ, USA, 21–23 March 2007.
- Yang, W.; Deng, F.; Ma, S.; Wu, L.; Sun, Z.; Hu, C. Test case reuse based on software testing knowledge graph and collaborative filtering recommendation algorithm. In Proceedings of the 2021 IEEE 21st International Conference on Software Quality, Reliability and Security Companion (QRS-C), Haikou, China, 6–10 December 2021.
- Chen, P.; Xi, A. Research on industrial software testing knowledge database based on ontology. In Proceedings of the 2019 6th International Conference on Dependable Systems and Their Applications (DSA), Harbin, China, 3–6 January 2020.
- Liu, G.; Lu, Y.; Shi, K.; Chang, J.; Wei, X. Convolutional neural networks-based locating relevant buggy code files for bug reports affected by data imbalance. *IEEE Access* **2019**, *7*, 131304–131316. [CrossRef]
- Zhang, J.; Xie, R.; Ye, W.; Zhang, Y.; Zhang, S. Exploiting code knowledge graph for bug localization via bi-directional attention. In Proceedings of the 28th International Conference on Program Comprehension, Seoul, Republic of Korea, 13–15 July 2020.
- Zhou, C. Intelligent bug fixing with software bug knowledge graph. In Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Lake Buena Vista, FL, USA, 4–9 November 2018.

21. Ji, Z.; Lee, N.; Frieske, R.; Yu, T.; Su, D.; Xu, Y.; Ishii, E.; Bang, Y.; Madotto, A.; Fung, P. Survey of hallucination in natural Language Generation. *ACM Comput. Surv.* **2022**, *55*, 1–38. [CrossRef]
22. Pan, S.; Luo, L.; Wang, Y.; Chen, C.; Wang, J.; Wu, X. Unifying large language models and Knowledge Graphs: A roadmap. *arXiv* **2023**, arXiv:2306.08302.
23. Mitchell, T.; Cohen, W.; Hruschka, E.; Talukdar, P.; Yang, B.; Betteridge, J.; Carlson, A.; Dalvi, B.; Gardner, M.; Kisiel, B.; et al. Never-ending learning. *Commun. ACM* **2018**, *61*, 103–115. [CrossRef]
24. Mesbah, A.; Prasad, M.R. Automated cross-browser compatibility testing. In Proceedings of the 33rd International Conference on Software Engineering, Honolulu, HI, USA, 21–28 May 2011.
25. Mahajan, S.; Halfond, W.G.J. Detection and localization of HTML presentation failures using computer vision-based techniques. In Proceedings of the 2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST), Graz, Austria, 13–17 April 2015.
26. Swearngin, A.; Li, Y. Modeling mobile interface tappability using crowdsourcing and deep learning. In Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems, Glasgow, UK, 4–9 May 2019.
27. Chen, J.; Chen, C.; Xing, Z.; Xu, X.; Zhu, L.; Li, G.; Wang, J. Unblind your apps: Predicting natural-language labels for mobile GUI components by deep learning. In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, Seoul, Republic of Korea, 27 June–19 July 2020.
28. Mahajan, S.; Li, B.; Behnamghader, P.; Halfond, W.G.J. Using visual symptoms for debugging presentation failures in web applications. In Proceedings of the 2016 IEEE International Conference on Software Testing, Verification and Validation (ICST), Chicago, IL, USA, 11–15 April 2016.
29. Mahajan, S.; Alameer, A.; McMin, P.; Halfond, W.G.J. Automated repair of layout cross browser issues using search-based techniques. In Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, 10–14 July 2017.
30. Choudhary, S.R.; Prasad, M.R.; Orso, A. X-PERT: Accurate identification of cross-browser issues in web applications. In Proceedings of the 2013 35th International Conference on Software Engineering (ICSE), San Francisco, CA, USA, 18–26 May 2013.
31. Choudhary, S.R.; Prasad, M.R.; Orso, A. CrossCheck: Combining crawling and differencing to better detect cross-browser incompatibilities in web applications. In Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation, Montreal, QC, Canada, 17–21 April 2012.
32. Roy Choudhary, S.; Versee, H.; Orso, A. WEBDIFF: Automated identification of cross-browser issues in web applications. In Proceedings of the 2010 IEEE International Conference on Software Maintenance, Timi oara, Romania, 12–18 September 2010.
33. Mahajan, R.; Shneiderman, B. Visual and textual consistency checking tools for graphical user interfaces. *IEEE Trans. Softw. Eng.* **1997**, *23*, 722–735. [CrossRef]
34. Issa, A.; Sillito, J.; Garousi, V. Visual testing of Graphical User Interfaces: An exploratory study towards systematic definitions and approaches. In Proceedings of the 2012 14th IEEE International Symposium on Web Systems Evolution (WSE), Trento, Italy, 28 September 2012.
35. Apple. Human Interface Guidelines. Available online: <https://developer.apple.com/design/human-interface-guidelines/> (accessed on 10 March 2024).
36. Microsoft. Microsoft Interface Definition Language 3.0 Reference. Available online: <https://learn.microsoft.com/en-us/uwp/midl-3> (accessed on 10 March 2024).
37. 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, Gothenburg, Sweden, 27 May–3 June 2018.
38. Chen, C.; Feng, S.; Xing, Z.; Liu, L.; Zhao, S.; Wang, J. Gallery D.C.: Design search and knowledge discovery through auto-created GUI component gallery. *Proc. ACM Hum. Comput. Interact.* **2019**, *3*, 1–22. [CrossRef]
39. Wang, N.; Feng, W.; Yin, J.; Ng, S.K. EasySpider: A no-code visual system for crawling the web. In Proceedings of the Companion ACM Web Conference 2023, Austin, TX, USA, 30 April–4 May 2023.
40. Salton, G.; Buckley, C. Term-weighting approaches in automatic text retrieval. *Inf. Process. Manag.* **1988**, *24*, 513–523. [CrossRef]
41. Zhao, X.; Xing, Z.; Kabir, M.A.; Sawada, N.; Li, J.; Lin, S.W. HDSKG: Harvesting domain specific knowledge graph from content of webpages. In Proceedings of the 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), Klagenfurt, Austria, 20–24 February 2017.
42. Stanford NLP Group. Stanford CoreNLP. Available online: <https://stanfordnlp.github.io/CoreNLP/> (accessed on 10 March 2024).
43. Explosion. spaCy. Available online: <https://spacy.io/> (accessed on 10 March 2024).
44. Grover, C.; Tobin, R. Rule-Based Chunking and Reusability. In Proceedings of the Fifth International Conference on Language Resources and Evaluation (LREC'06), Genoa, Italy, 22–28 May 2006; Calzolari, N., Choukri, K., Gangemi, A., Maegaard, B., Mariani, J., Odijk, J., Tapias, D., Eds.; European Language Resources Association (ELRA): Genoa, Italy, 2006.
45. Canny, J. A computational approach to edge detection. *IEEE Trans. Pattern Anal. Mach. Intell.* **1986**, *PAMI-8*, 679–698.
46. Sobel, I.; Feldman, G. A 3×3 isotropic gradient operator for image processing. *Pattern Classif. Scene Anal.* **1973**, *1*, 271–272.
47. Burel, G.; Carel, D. Detection and localization of faces on digital images. *Pattern Recognit. Lett.* **1994**, *15*, 963–967. [CrossRef]
48. Douglas, S.; Kirkpatrick, T. Do color models really make a difference? In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems Common Ground—CHI '96, Vancouver, BC, Canada, 13–18 April 1996.

49. Zhou, X.; Yao, C.; Wen, H.; Wang, Y.; Zhou, S.; He, W.; Liang, J. EAST: An efficient and accurate scene text detector. In Proceedings of the 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Honolulu, HI, USA, 21–26 July 2017.
50. Chen, J.; Xie, M.; Xing, Z.; Chen, C.; Xu, X.; Zhu, L.; Li, G. 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, Virtual, 8–13 November 2020.
51. Yang, B.; Xing, Z.; Xia, X.; Chen, C.; Ye, D.; Li, S. UIS-Hunter: Detecting UI design smells in android apps. In Proceedings of the 2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), Madrid, Spain, 25–28 May 2021.
52. Landis, J.R.; Koch, G.G. An application of hierarchical kappa-type statistics in the assessment of majority agreement among multiple observers. *Biometrics* **1977**, *33*, 363–374. [[CrossRef](#)] [[PubMed](#)]
53. Stone, M. Cross-validatory choice and assessment of statistical predictions. *J. R. Stat. Soc.* **1974**, *36*, 111–133. [[CrossRef](#)]
54. Cortes, C.; Vapnik, V. Support-vector networks. *Mach. Learn.* **1995**, *20*, 273–297. [[CrossRef](#)]
55. Treude, C.; Robillard, M.P.; Dagenais, B. Extracting development tasks to navigate software documentation. *IEEE Trans. Softw. Eng.* **2015**, *41*, 565–581. [[CrossRef](#)]
56. Sun, J.; Xing, Z.; Chu, R.; Bai, H.; Wang, J.; Peng, X. Know-how in programming tasks: From textual tutorials to task-oriented knowledge graph. In Proceedings of the 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME), Cleveland, OH, USA, 30 September–4 October 2019.
57. Ren, S.; He, K.; Girshick, R.; Sun, J. Faster R-CNN: Towards real-time object detection with region proposal networks. *IEEE Trans. Pattern Anal. Mach. Intell.* **2017**, *39*, 1137–1149. [[CrossRef](#)] [[PubMed](#)]
58. Singh, R.; Mangat, N.S. *Elements of Survey Sampling*; Springer: Dordrecht, The Netherlands, 2010.
59. Deka, B.; Huang, Z.; Franzen, C.; Hibschan, J.; Afergan, D.; Li, Y.; Nichols, J.; Kumar, R. 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, Québec City, QC, Canada, 22–25 October 2017.
60. OpenAI. GPT-4 Technical Report. *arXiv* **2023**, arXiv:2303.08774.
61. OpenAI. GPT-4V(ision) System Card. Available online: <https://openai.com/research/gpt-4v-system-card> (accessed on 3 March 2024).
62. Winer, B.J. *Statistical Principles in Experimental Design*; McGraw-Hill Book Company: New York, NY, USA, 1962.
63. Brooke, J. SUS: A ‘quick and dirty’ usability scale. In *Usability Evaluation in Industry*; CRC Press: Boca Raton, FL, USA, 1996; pp. 207–212.

**Disclaimer/Publisher’s Note:** The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.