Interaction Proxy Manager: Semantic Model Generation and Run-time Support for Reconstructing Ubiquitous User Interfaces of Mobile Services

TIAN HUANG, Department of Computer Science and Technology, Tsinghua University, China
CHUN YU*, Department of Computer Science and Technology, Tsinghua University, China
WEINAN SHI, Department of Computer Science and Technology, Tsinghua University, China
BOWEN WANG, Department of Computer Science and Technology, Tsinghua University, China
DAVID YANG, Department of Computer Science and Technology, Tsinghua University, China
YIHAO ZHU, Department of Computer Science and Technology, Tsinghua University, China
ZHAOHENG LI, Department of Computer Science and Technology, Tsinghua University, China
YUANCHUN SHI, Department of Computer Science and Technology, Tsinghua University, China

Emerging terminals, such as smartwatches, true wireless earphones, in-vehicle computers, etc., are complementing our portals to ubiquitous information services. However, the current ecology of information services, encapsulated into millions of mobile apps, is largely restricted to smartphones; accommodating them to new devices requires tremendous and almost unbearable engineering efforts. Interaction Proxy, firstly proposed as an accessible technique, is a potential solution to mitigate this problem. Rather than re-building an entire application, Interaction Proxy constructs an alternative user interface that intercepts and translates interaction events and states between users and the original app's interface. However, in such a system, one key challenge is how to robustly and efficiently "communicate" with the original interface given the instability and dynamicity of mobile apps (e.g., dynamic application status and unstable layout). To handle this, we first define UI-Independent Application Description (UIAD), a reverse-engineered semantic model of mobile services, and then propose Interaction Proxy Manager (IPManager), which is responsible for synchronizing and managing the original apps' interface, and providing a concise programming interface that exposes information and method entries of the concerned mobile services. In this way, developers can build alternative interfaces without dealing with the complexity of communicating with the original app's interfaces. In this paper, we elaborate the design and implementation of our IPManager, and demonstrate its effectiveness by developing three typical proxies, mobile-smartwatch, mobile-vehicle and mobile-voice. We conclude by discussing the value of our approach to promote ubiquitous computing, as well as its limitations.

CCS Concepts: • **Human-centered computing → Ubiquitous and mobile computing systems and tools**; **Systems and tools for interaction design**; *User interface management systems*; Graphical user interfaces.

*Corresponding author.

Authors' addresses: Tian Huang, ht20@mails.tsinghua.edu.cn, Department of Computer Science and Technology, Tsinghua University, Beijing, China; Chun Yu, Department of Computer Science and Technology, Tsinghua University, Beijing, China, chunyu@tsinghua.edu.cn; Weinan Shi, Department of Computer Science and Technology, Tsinghua University, Beijing, China, swn@mail.tsinghua.edu.cn; Bowen Wang, Department of Computer Science and Technology, Tsinghua University, Beijing, China, wbw20@mails.tsinghua.edu.cn; David Yang, Department of Computer Science and Technology, Tsinghua University, Beijing, China, ydw22@mails.tsinghua.edu.cn; Yihao Zhu, Department of Computer Science and Technology, Tsinghua University, Beijing, China, zhuyh22@mails.tsinghua.edu.cn; Zhaoheng Li, Department of Computer Science and Technology, Tsinghua University, Beijing, China, zhaohengli.thu@gmail.com; Yuanchun Shi, Department of Computer Science and Technology, Tsinghua University, Beijing, China, shiyc@tsinghua.edu.cn.

2474-9567/2023/9-ART99
https://doi.org/10.1145/3610929

Additional Key Words and Phrases: GUI semantics, mobile applications, multi-device user interfaces, model-based user interface description languages, UI reconstruction

ACM Reference Format:
Tian Huang, Chun Yu, Weinan Shi, Bowen Wang, David Yang, Yihao Zhu, Zhaoheng Li, and Yuanchun Shi. 2023. Interaction Proxy Manager: Semantic Model Generation and Run-time Support for Reconstructing Ubiquitous User Interfaces of Mobile Services. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 7, 3, Article 99 (September 2023), 39 pages. https://doi.org/10.1145/3610929

1 INTRODUCTION

Smartphones, with the most abundant application resources, are currently mostly accessible by touching the phone screen alone. With the rise of AIoT, people tend to choose different devices and interactions depending on the use scenario [13], expecting applications to offer services in various forms [4, 30, 31, 46]. Just imagine a user has a new unread message. He/she may prefer reading it on a wearable device like a smartwatch or glasses while walking, on their PC at work for efficiency, or through a car screen or voice interaction while driving for convenience and safety. While these alternative devices offer the potential to further enrich user experiences, the challenge of developing differentiated user interfaces comes to our attention.

To present existing mobile applications to various terminals, there are two approaches in mainstream practice. The first involves screen-sharing technology based on video streaming, such as Google Cast [1], Miracast [2]. However, this solution is limited to visual content and does not adapt to other modalities like voice or gestures. The second approach leverages the system-level framework. Examples include CarPlay [3] and Android Auto [4], enabling the deployment of applications to car screens. Whereas, it is unable to assist third-party developers in cross-device interface construction, necessitating code adaptation from application developers.

Inspired by the previous work for enhancing accessibility [61], we consider Interaction Proxy as a solution to the aforementioned issues. This technology offers an interaction remapping mechanism that intercepts and forwards page content and events on the phone, enabling a flexible construction of new user interfaces without altering the code of the original application. However, despite demonstrating the potential for accessibility improvements, Interaction Proxy is currently at the proof-of-concept stage, with limited practical reliability [61]. Figure 1 highlights several challenges associated with Interaction Proxy, including unstable or missing mobile page data, difficulties locating GUI widgets, and issues with synchronizing the state of each interface [32, 43]. Consequently, this technique may result in failed function execution or incomplete content of the new user interface [6], leading to high development and debugging costs.

The challenges mentioned above primarily stem from the dynamic nature of user interfaces [29]. To eliminate the instability brought by the interface, this paper proposes Interaction Proxy Manager (IPManager), a software module that functions between interfaces, responsible for decoupling the application's *interface implementation* from the *service semantics* it contains. In the IPManager, we define UI-Independent Application Description (UIAD), a reversed-engineered semantic model, to synchronize and manage the original application's interface. The model organizes the application's *information* and *methods* in a hierarchical structure, making it widely applicable due to its alignment with human cognition and the principles of the underlying object-oriented implementation of the application.

As shown in Figure 2, the IPManager operates in two phases. In the offline phase, the registration system aids developers in model design and establishes the relationship with the original GUI using low-effort interactive annotations. This process formulates recognition strategies, maintains precise mapping relationships, and

[1]https://developers.google.com/cast
[2]https://www.wi-fi.org/discover-wi-fi/miracast
[3]https://www.apple.com/ios/carplay
[4]https://www.android.com/auto

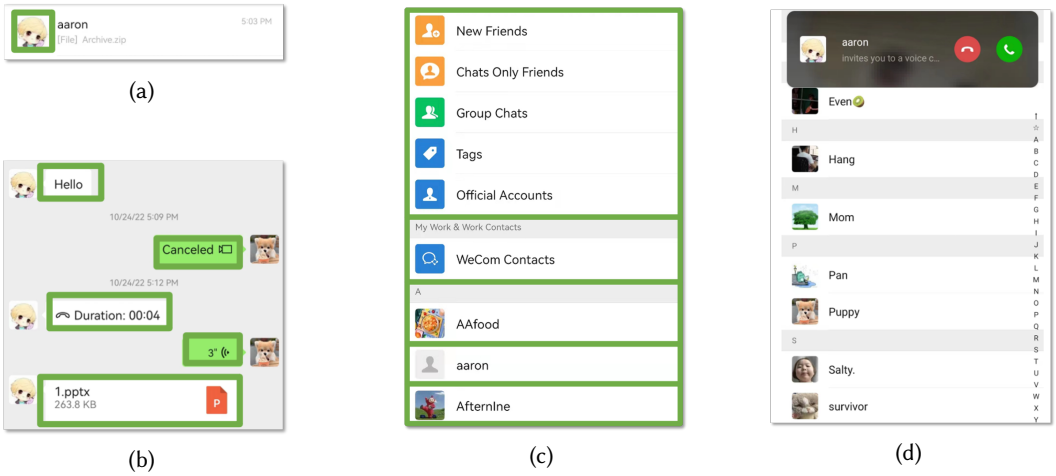


Fig. 1. Challenges posed by the dynamicity of the interface. In this example of WeChat, green boxes indicate problematic widgets. (a) The avatar widget is not available on the layout. (b) There are a wide variety of widgets involved in the chat. (c) The layout hierarchy is confusing. (d) A situation of unrecognizable page.

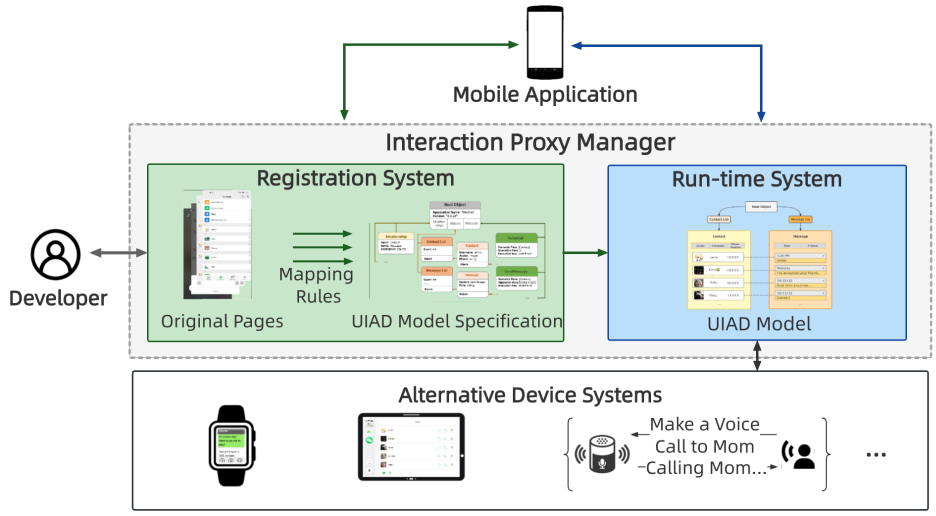


Fig. 2. How the Interaction Proxy Manager works. It extracts a UI-Independent Application Description Model from an existing mobile application and provides information and method API for alternative interface systems to construct GUIs on the smartwatch, car display, and VUI.

provides model APIs for new interfaces. In the run-time phase, the model responds to API calls from new interfaces, delivering the requested data or controlling the phone to execute the necessary actions. Thereby, the new interfaces can indirectly access the original GUI data, whose dynamicity is directly addressed by the model.

Moreover, we implement techniques such as the maintenance of the run-time relationship between the model and the original GUI, cache and route optimizations for stable and efficient communication.

Interaction Proxy Manager overcomes the limitations of existing solutions and aims to achieve the following goals: 1) **Robustness**. Decoupling the interface from the service allows for a more accurate extraction of application semantics and shields new interface systems from the original interface's instability, leading to more reliable operation. 2) **Generality**. The model can be both generated from a wide range of applications and invoked by various devices to construct novel user interfaces. 3) **Economy**. Model generation can be a one-shot process, enabling swift deployment to different devices and scenarios without burdensome modifications. 4) **Flexibility**. IPManager transcends the physical limitations of a single GUI, allowing for alternative modalities beyond visual feedback. 5) **Incrementability**. The model can be continuously updated and adapted to new semantic concepts based on user requirements.

The specific contributions of our work therefore include:

- We propose Interaction Proxy Manager to eliminate the impacts of the original interface, which fulfills semantics-based reverse engineering and provides reliable interaction mapping.
- We introduce the Interactive Annotation Mechanism for registering with the original interface, which supports dynamic learning strategies for page classification and widget recognition, streamlines the annotation process, and guarantees robust model generation.
- We provide a series of run-time support such as cache and routing optimization to properly handle the dynamicity of the interface and efficiently respond to new interfaces' requests.

2 RELATED WORK

This section reviews the literature relevant to our work, including prior Interaction Proxy systems, mobile GUI semantic analysis, model-based UI development, and annotation systems with interactive machine learning.

2.1 Prior Interaction Proxy Systems

The concept of Interaction Proxy has been previously utilized in designing interactions based on existing user interfaces for various purposes. Therein, it has been used for run-time repair and enhancement of the accessibility of mobile applications [47, 61–63] or adapting user interfaces to wearable devices [58, 64]. SUGILITE [33], KITE [37], and PUMICE [36] propose to program by demonstration or natural language inputs on smartphones. SOVITE [34] helps users discover conversational breakdowns using the existing mobile GUIs. Rataplan [53] is pixel-based approach for linking multi-modal proxies to automated sequences of actions in GUIs.

These systems rely on the direct remapping of interface elements or events, resulting in case-by-case connections. More importantly, they lack robust mobile page recognition, usually using pre-defined rules [61], which leads to unreliable generation results, especially for content-rich UIs like large-screen interfaces.

2.2 Semantic Analysis for Mobile GUI

Mobile GUI layout obtained through Android's Accessibility Service API [5] only presents the interface rendering without in-depth semantics. Semantic analysis addresses GUI irregularities [32, 43] and assists in reverse engineering of application logic. We define the design space of mobile GUI semantic analysis in 4 progressively deeper levels: *Item*, *Label*, *Concept Hierarchy*, and *Concept Graph*.

Level 1: Item. Identify all meaningful items on the GUI page. REMAUI [44] and GUI skeleton [14] recognize GUI elements using computer vision or OCR, and extract visual features in screenshots.

[5] https://developer.android.com/referenc/android/accessibilityservice/AccessibilityService

Level 2: Label. Currently, semantic labeling of GUI widgets is the focus of most research, including widget object detection [16], UI embeddings [10, 28], natural language description of pages [56], and furthermore, UX concept assignment to buttons and icons [15, 39, 41].

Level 3: Concept Hierarchy. The logical relationships between components are normally described in a tree, in which the components are not GUI widgets but semantic concepts. Screen Recognition[57, 60] comes close to this level, describing the logical relationships via hierarchical segmentation. However, the semantic information is shallow, the components remain at the widget level, and the edges of the tree are not differentiated according to a deeper level of semantics.

Level 4: Concept Graph. This level represents the deepest analysis, illustrating relationships between all semantic concepts and necessitating integration of semantics across pages to restore the application's workflow and data management. Studies address mapping natural language instructions [38] or voice commands [45] to GUI action sequences, defining concepts for GUI widgets and operation paths [36], characterizing inter-screen semantics [35], identifying interaction trace events [21], and various UI tasks [55]. However, current research is limited in capturing shallow cross-page semantics, necessitating further efforts to develop practical concept graphs.

Our work is at the intersection between Level 3 and Level 4. We build the semantic hierarchy, integrate the involved content of the multiple pages, and describe the relationships between them.

2.3 Model-Based UI Development

Model-Based UI Development (MBUID) simplifies UI development by generating code from models that define data structure, behavior, and relationships [42, 49, 51]. The Camelon Reference Framework [23] outlines the MBUID process, detailing layers of abstraction and their relationships: the Task and Concepts level, Abstract UI (AUI) [52], Concrete UI (CUI), and Final UI (FUI). Various MBUID software tools have emerged [40], supporting cross-toolkit development and element reuse. MBUID also enables reusable model creation [51] and offers design assistance features [25].

Recent MBUID advancements focus on UX role exploration [2], enhancing interface adaptation via new frameworks [1, 3], developing domain-specific languages for user interfaces [5, 48], developing multi-platform applications [12] and leveraging low-code platforms for swift web app development [11, 50].

While MBUID expedites code generation and eases front-end development, it neglects the complex development requirements associated with data and functionality sources in the UI. Our approach seeks to decouple information services from legacy smartphone applications through reverse engineering, subsequently offering APIs for new UI development and facilitating the integration of data and functionality in new UIs.

2.4 Annotation Systems with Interactive Machine Learning

Interactive Machine Learning (IML) enables users to iteratively build and refine mathematical models through input and review cycles, without extensive background knowledge [8, 18]. This makes IML suitable for annotation systems, reducing modeling expenses and expertise requirements for annotators.

IML enables iterative improvements based on manual evaluations. Examples include Crayons [20], CueFlik [7, 22], Abstrackr [54], and medical image clustering [26]. IML-based techniques also enhance user comprehension of system functionality in text analysis [19], abstract annotation [59], and sentiment analysis [27]. Collaborative semantic inference [24] further improves human-system understanding, as recently implemented in Google Translate to prevent gender discrimination.

In our work, we use IML to help the system learn classification strategies for pages and widgets. This approach is suitable for our task with dynamical granularity and significantly reduces annotators' expertise and workload.

3 UIAD MODEL SPECIFICATION

The stability and reusability of an application's semantics far surpass that of its interface, as countless GUI element combinations can represent the same functionality. Therefore, we propose a UIAD Model that organizes the remaining semantic knowledge after removing the UI implementation. The model aims to provide a consistent structure for representing application semantics while adapting to multi-modal interfaces.

In this section, we introduce the UIAD Model Specification using WeChat, a popular instant messaging application, as an example. As the model's foundation, it defines the structure and APIs, while the specific content becomes available only after registration with the original interface.

3.1 Model Structure

Drawing inspiration from epistemology, which divides knowledge into descriptive "knowing-that" and procedural "knowing-how" [9], the model organizes application content into **information** and **methods** within a hierarchical structure. This representation aligns with human perception of application semantics, reducing learning costs for developers. Moreover, it is consistent with object-oriented programming principles in application source code, enhancing the feasibility of semantic description.

The UIAD Model can describe entire applications, multiple sub-functions, or even a single GUI page by representing the semantic structure as a tree. Tree nodes, defined as semantic elements, exemplify the hierarchical order of the semantics via parent-child relationships. In this arrangement, child nodes embody components of their parent nodes' semantics.

For example, Figure 3 illustrates part of the model structure for WeChat. Once the model establishes a correspondence with the application content, the semantic elements can derive values from multiple pages of the original GUI, as shown in Figure 5, enabling the model to integrate semantic information from diverse sources.

To guide the design of the model structure for organizing **information** and **methods**, we will subsequently provide detailed definitions for each type of semantic element. First, the **information** is described through the following semantic elements:

- *Root Object*: The root represents the entire application.
- *Object*: A subtree with such an element at the root describes all the information of an *Object*. Different types of *Objects* are distinguished by their names, such as *Contact* and *Message*.
- *Object List*: Elements of this class consist of several *Object* children sharing the same name, indicating that these *Objects* exist simultaneously in the same list.
- *Property*: A property of an *Object* or *Object List*, consisting of the property's name, type, and value. The value, represented as a string, must be parsed based on the type. The value may be initially set, or left empty during the design phase and later populated with data from the original GUI.

Relationships between semantic elements occasionally extend beyond the conventional parent-child hierarchy found in tree structures, thereby requiring representations for lateral or cross-branch linkages. To accommodate these unconventional relationships within the overall tree structure, we introduce a distinct semantic element, the *Relationship*. As a child of the *Root Object*, a *Relationship* embodies a "link" between two previously defined semantic elements (*Object* or *Object List*). Figure 3 illustrates a *Relationship* where a *Contact* is the "source" of a *Message*.

Second, the **methods** are flattened in the model, differing from multipage jumps in the mobile GUI. They are organized by the following classes of semantic elements:

- *Method*: Different *Methods* are distinguished by their names, such as *"VoiceCall"* and *"SendMessage"*. As a child of the *Root Object*, a *Method* comprises three properties: *Semantic Parameters*, *Operation Parameters*, and *Execution Sequence*.

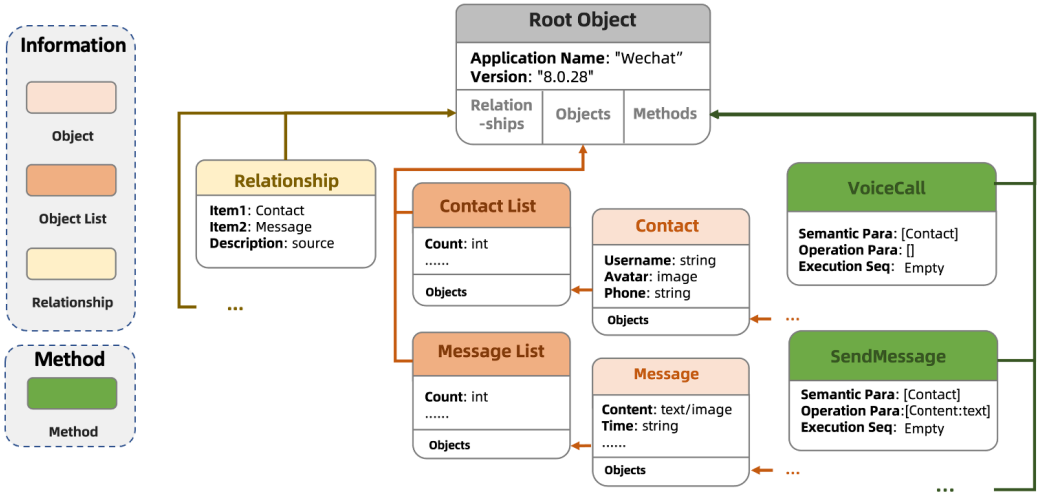


Fig. 3. Part of the UIAD Model structure of WeChat. As the starting point, *Root Object* has 2 fixed-value properties. It has two *Object List* children: *Contact List* and *Message List*. *Contact List* contains *Contact* objects, describing all properties of each contact. Similarly, *Message List* contains *Message* objects, organizing all data related to each message. Additionally, two methods, *"VoiceCall"* and *"SendMessage"*, are defined with detailed parameters. At this stage, most of the *Properties*' values and all the *Execution Sequences* are still empty, awaiting data from the original interface.

- *Semantic Parameters*: The list of semantic parameters of the method. Each parameter refers to a previously defined element (*Object List* or *Object*), such as the *Contact* involved in the *"SendMessage" Method*.
- *Operation Parameters*: The list of the method's parameters, excluding the semantic ones mentioned above. They are typically required to ensure the completeness of the operation, such as the content for the *"SendMessage" Method*.
- *Execution Sequence*: The list of operations in the original interface, i.e., operable GUI widgets that can accept *Operation Parameters* or potentially initiate the *Method*. Initially, the sequence is empty, as shown in Figure 3. Only after registering with the original interface, the edit box and the "Send" button will be added to the *Execution Sequence* of the *"SendMessage" Method*, as shown in Figure 5.

The initial model structure needs to be built manually according to the functional requirements, ensuring it includes all relevant semantic elements. For example, when deploying the function of sending messages, the list of contacts and messages, and the parameters involved in "SendMessage" are necessary. The definitions of these semantic elements, including their names, types, and parameters, must be specified manually before the registration process. However, the specific values (e.g. username "Lee") and *Execution Sequence* will be derived from the original GUI widgets after registration.

3.2 Model API

To access the data in the model, two types of model APIs are supported: to get information and to use methods. As shown in Figure 4, new interface systems can simply call these APIs to access the content of the original GUI, and thus fill it into alternative interfaces. Particularly, Figure 10 demonstrates how a car display calls these APIs during runtime. The three parameters required to call the API are defined as follows.

- *API Name*: Denotes the API to call.
- *Condition*: Defines constraints for the model's subtree (or forest) selection.
- *Target*: Identifies desired information or method.

Table 5 in the Appendix lists in detail the main supported APIs and their usages. The following subsections introduce the two categories of APIs.

3.2.1 To Get Information. We can access the content of any node or subtree within the model. Given the *Condition* and *Target*, the model locates the relevant information in the tree and returns the result. The model supports *get_property* and *get_related_info* for object information queries, and associated APIs for list queries.

- *get_property*: The properties can belong to any subtree or forest, as determined by the *Condition*. The *Target* specifies which properties to return. In particular, if the *target* is an *Object* name, all properties of the object are returned.
- *get_related_info*: Get the information of *Object* B that is related to *Object* A. The description of *Object* A and their relationship are set in the *Condition*, and the *Target* specifies which properties of *Object* B are required.
- *list related*: Supports specific operations such as retrieving list items in a set interval (*get_list*) and finding an object's position (*index_of*). Considering the list's versatility, additional APIs like *filter* and *sort* can be implemented.

3.2.2 To Use Methods. We can use any method defined in the model. By assigning values to the required parameters in the *Condition* and specifying the method name in the *Target*, the model can trigger the method by performing operations according to the *Execution Sequence*.

4 INTERACTION PROXY MANAGER

In this section, we present the IPManager, responsible for generating a UIAD Model from the original mobile GUI and managing requests from the new interface system.

To achieve this, we first establish the mapping relationship between the original pages and the model, enabling run-time model generation based on the current page. Therefore, the IPManager consists of an offline registration system and a run-time system, depicted in Figure 4. During the offline phase, the system registers the model with the original pages, storing the results in the Model Registration File. In the run-time phase, the system employs this file to generate the corresponding model instance from the current GUI.

To enhance the reliability and efficiency of the run-time system, we introduce the model manager, overseeing data exchange with new interface systems. The model manager dynamically maintains mapping relationships between the model and the original GUI, executes simulated actions on the original GUI, manages previously generated model instances, and preserves the latest version of the UIAD model.

4.1 Registration System

The registration system, depicted in the dashed box of Figure 4, establishes the mapping relationships between the model and the original GUI. These relationships are integrated into the Model Registration File, which lays the foundation for subsequent run-time semantic analysis.

Based on the functional requirements of a given application, developers organize all semantic elements and establish the corresponding Model Specification, as described in Section 3. The system then identifies the original GUI widgets linked to each semantic element. To achieve this, the system records three components: (1) *page classification*, (2) *widget recognition*, and (3) *page jumping graph*. The first two components support widget recognition within a single page, while the third enables cross-page searches. The process requires human annotation, for which we employ an interactive annotation mechanism to reduce costs, further detailed in Section 5.

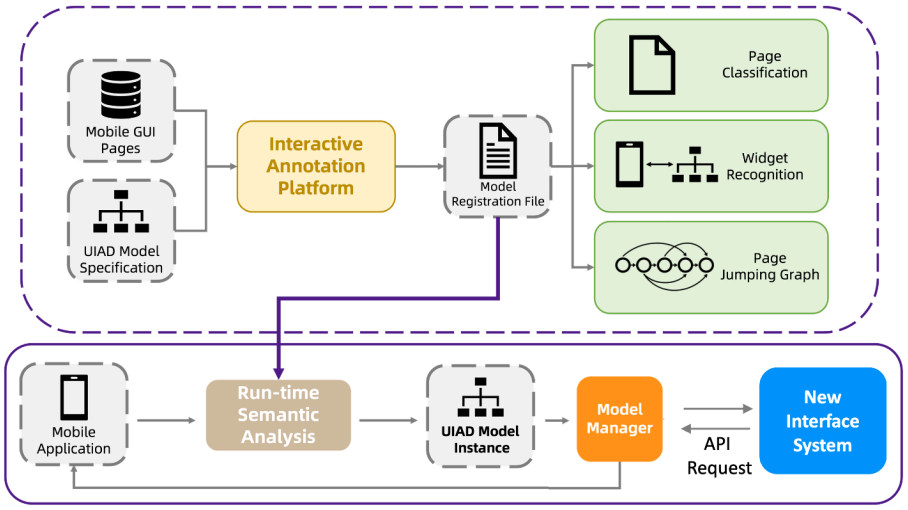
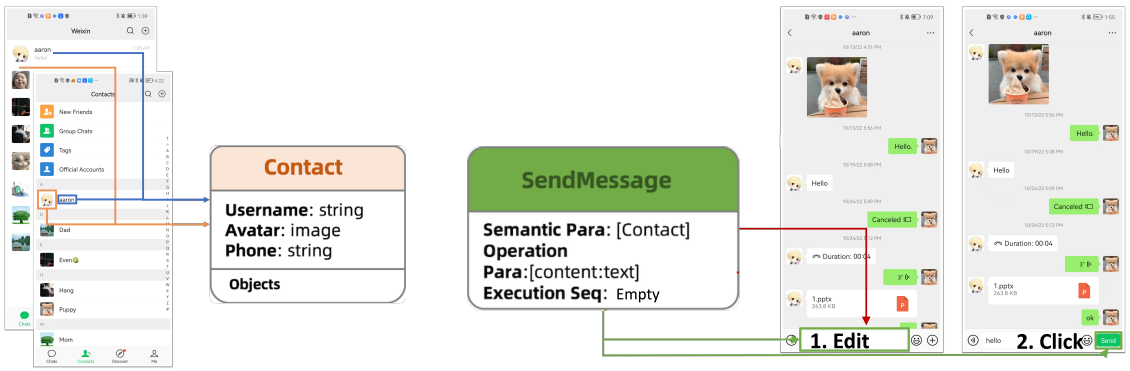


Fig. 4. IPManager is in charge of the generation and run-time support of the UIAD Model. The dashed box area is the registration system in the offline phase, and the solid box area is the run-time system.



(a) (b)

Fig. 5. Two examples of mapping rules that establish the correspondence between application content and semantic elements. (a) A contact's username and avatar can be extracted from two classes of pages. Based on the GUI widget being mapped, the username is taken from its text, and the avatar is taken from a screenshot at a given offset relative to it. (b) By mapping the corresponding two widgets, the *"SendMessage" Method* is triggered by one edit and one click.

4.1.1 Page Classification. Mobile pages are categorized based on their topics. Page classification serves as the first feature for GUI widget recognition and forms the foundation of the page jumping graph, helping the run-time system identify the current state and locate the correct routes. The classification criteria can dynamically change. When pages within the same class create ambiguity in widget recognition or necessitate a page jumping between them, we can divide the class into subclasses.

4.1.2 Widget Recognition. As shown in Figure 5, a mapping rule specifies the property of specific GUI widgets (e.g., image, text, action) corresponding to a semantic element in the model tree, indicating that the GUI widgets assign value to the semantic element (Figure 5a) or clarify its triggering mechanism (Figure 5b). Different mapping rules apply to different GUI widgets; hence, when using a rule, it is necessary to recognize all applicable widgets on the run-time GUI. The process involves classifying widgets according to the semantic elements they map to. Similarly to page classification, the criteria of widget recognition can dynamically change, which depends on the model structure.

4.1.3 Page Jumping Graph. While the previous sections cover the semantics within a single mobile page, the page jumping graph represents the semantic relationships between pages. It helps the run-time system find the correct routes when the required widget is not present on the currently displayed page. The graph encapsulates both the **route** and the **context cursor**.

Route: A basic jumping graph is a directed graph generated automatically, with nodes as pages of the same class. Edges specify the widget and its action (typically "click") triggering the jump. With this graph, a route between any two pages can be found, ensuring at least one route from the start page to the homepage and then to the end page.

Context cursor: Page jumping is required not only when page classes differ, but also when semantic differences exist among pages within the same class. For instance, if the current mobile page is for chatting with Contact A and the target is Contact B, page jumping is needed even though both pages are chat pages. Therefore, we define the contextual semantics of a single page as a cursor in the model, with the subtree below representing the content covered by the page. In the example above, there are two subtrees, Contact A and Contact B, in the run-time generated model. Since the cursor is on Contact A, page jumping is required to reach Contact B.

4.2 Run-time Semantic Analysis

After registering with the original GUI, the run-time system analyzes the semantics of the displayed page on the phone using the Model Registration File (derived in Section 4.1). As depicted in the solid box of Figure 4, each parsed UIAD Model at runtime contains information and methods for a single page, known as a *model instance* (e.g., Figure 6). The system re-analyzes semantics whenever the mobile GUI updates.

Run-time semantic analysis follows the same two-step process as the offline phase: page classification and widget recognition. Necessary widgets are identified and assigned to corresponding semantic elements based on pre-designed mapping rules, adhering to a top-down layout tree order. Algorithm 2, detailed in the Appendix, is utilized for this purpose.

4.3 Model Manager

The model manager is responsible for ensuring the UIAD Model's availability on demand. As depicted in Figure 7, the manager continuously listens to requests from new user interfaces and generates responses through the *Processor*. Since the model instances generated in Section 4.2 can only describe the current single page, the manager relies on the *Route* and *Cache* to record the required UIAD Model instances for data integration. If an error occurs at any stage, the *Error Recovery* is triggered to resolve the issue.

The following paragraphs detail the functionality of each module.

4.3.1 Processor. The Processor identifies the target subtrees in the model based on the received API request. For getting information requests, the corresponding part of the model tree is directly returned, while using methods requests prompt automatic phone operation according to the *Execution Sequence*. If data is missing, the manager invokes the Route module to perform page jumping and retries the process.

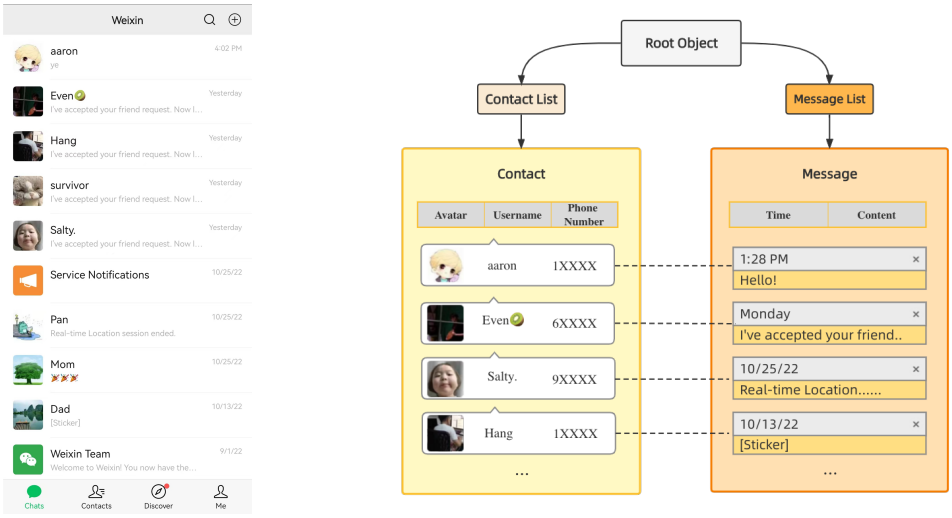


Fig. 6. (a) The current page displayed on the phone and (b) its corresponding UIAD Model instance.

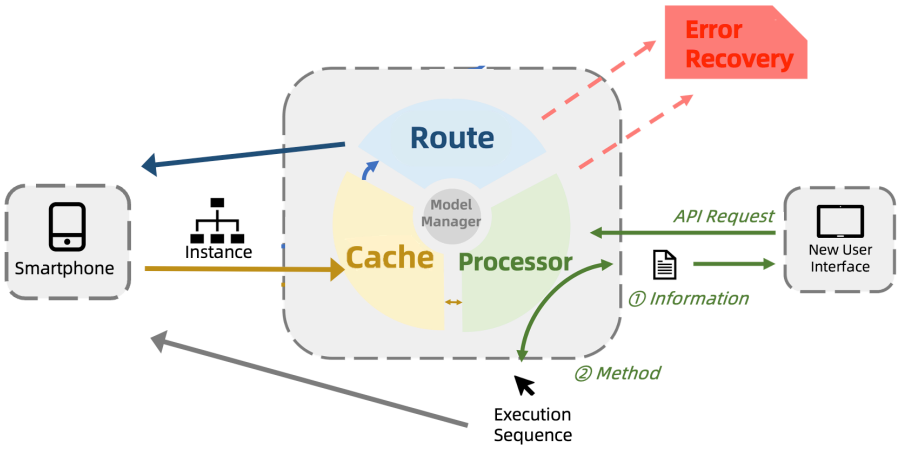


Fig. 7. Overview of how the model manager works.

4.3.2 Route. Data missing occurs when the state of the new UI and the original mobile GUI are not synchronized. The Route module resolves this issue in two cases:

- If the current context cursor conflicts with the target (e.g. the target is Contact B but the cursor is on Contact A), the manager first performs a global back until the cursor leaves its original location and resolves the conflict. It then initiates the second case's processing.

- Without context conflicts, the manager identifies all destination page classes containing the requested information or methods, based on the Model Registration File. Given the current displayed page class and the destination page classes, the manager determines the shortest route and operates the phone accordingly. If the route is not found or the operation fails, the Error Recovery (described below) is triggered.

4.3.3 Cache. To optimize efficiency, a cache merges all historical model instances to form a complete UIAD Model, reducing overhead from page jumping caused by Route module triggers. If a new instance conflicts with an old one, the cache is directly overwritten. Conflicts arise only when the same semantic element has different values or matches different GUI widgets. Model instances generated from pages of different classes are all preserved. The similar structures of all model instances facilitate merging or overwriting.

Additionally, a cache for lists is implemented. Synchronizing the mobile GUI and the new interface may be challenging when list lengths differ. The manager merges list items from multiple model instances into a larger list during phone scrolling, storing the expanded list in the cache. Cache data is prioritized when the API is called; if insufficient, the list is controlled to scroll and obtain more data.

4.3.4 Error Recovery. Run-time interruptions due to errors include the following two cases:

- The Route module failure: On the first failure, the manager performs global backs to return to the homepage and re-calls the Route module. If the second attempt also fails, the destination page might be missing from the preset dataset, prompting users to add page data using the registration system.
- Inability to locate requested data even after reaching the destination page: It can stem from either the data's absence, requiring model adjustment manually for alternate solutions, or widget recognition errors, which can be addressed by annotating unrecognized widgets and retraining a new recognition strategy.

5 INTERACTIVE ANNOTATION MECHANISM

Manual annotation is required during model registration with the original UI (described in Section 4.1). Among the three components mentioned, the page jumping graph can be primarily automated [17], and the context cursor is well-defined and easy to annotate. However, page classification and widget recognition pose greater challenges. Since classification criteria dynamically update based on model modifications, pre-defined heuristic rules or pre-trained data-driven models are not applicable. In this section, we propose the Interactive Annotation Mechanism (IAM) to efficiently carry out the annotation task.

5.1 Workflow of the IAM

Since GUIs originate from human design logic and follow established design principles, the variations in GUIs are enumerable. By extracting sufficient features from page layout data and images, it is possible to generate a comprehensive set of rules that cover all cases. The challenge lies in refining a classification strategy for each class. Directly proposing strategies involving complex feature combinations is impractical for annotators. Furthermore, without a comprehensive understanding of mobile GUI design, it is impossible to confidently establish a robust strategy based on sample interfaces. Additionally, pre-defined strategies cannot be dynamically adjusted when false cases occur.

In contrast to abstract classification strategies, human perception of the class to which each instance belongs is clear. To this end, we propose a human-computer collaborative mechanism that constructs the classification strategy iteratively, allowing for continuous improvements based on simple human input. Our interactive annotation mechanism, shown in Figure 8, enables continuous refinement of the classification strategy through iterative annotation and inspection.

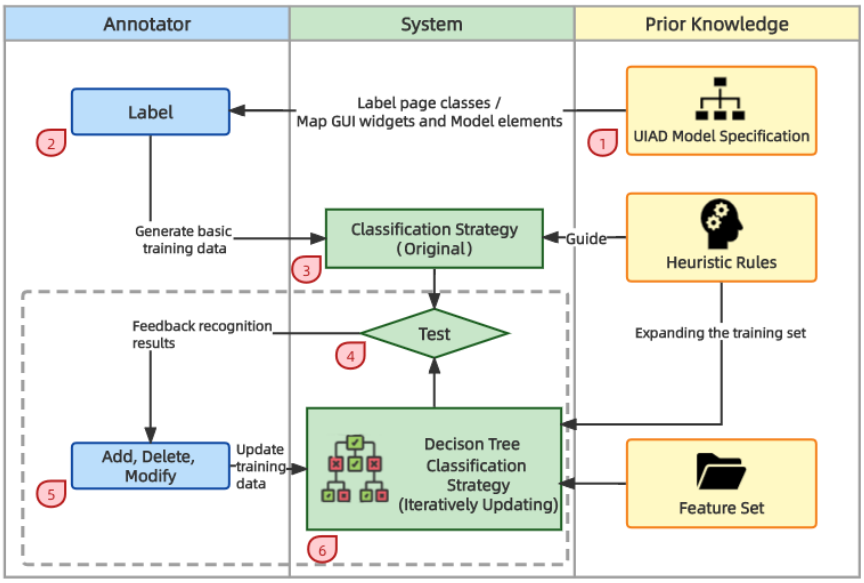


Fig. 8. Workflow of the interactive annotation mechanism. (1) UIAD Model Specification is designed. (2) Annotators label the class to which each page or widget belongs. (3) The system applies pre-defined heuristic rules as the original classification strategy. (4) Test with the dataset. (5) Annotators modify or confirm some of the results. (6) The system uses new annotations for decision tree training. Steps (4), (5) and (6) iterate until annotators no longer make new annotations.

5.1.1 Heuristic Rules. In the first round, we construct a fixed similarity function using a predefined set of features to determine whether two instances belong to the same class, with only one annotation to start the classification. The feature set and similarity function are available in Appendix B.

5.1.2 Iterative Update. Heuristic rules, used as the original classification strategy, cannot guarantee correctness (Table 1). These rules are tested on the dataset, and the results are visualized for annotators to correct errors. We use decision trees—an interpretable algorithm conducive to manual modification—trained based on new labeling or confirmation data, and refine them until satisfactory.

5.1.3 Expanding the Training Set. To further reduce the annotation workload, a series of heuristic rules are employed to automatically generate more negative examples based on the existing annotated data, resulting in an expanded training set.

5.1.4 Summary. Figure 9 demonstrates an example of training a widget recognition strategy through interactive annotation. Ideally, only one annotation per class is needed. To prevent redundancy, only unconfirmed results are sent to annotators. The strategy is tested on new pages and adjusted with additional examples when required. For similar cases, a single example suffices to adapt the strategy, avoiding exhaustive enumeration, as demonstrated in Figures 9(b) and 9(c), where the incorrectly recognized message time widgets are removed.

While both pages and widgets follow the workflow above, their implementations differ significantly, as detailed in Table 6.

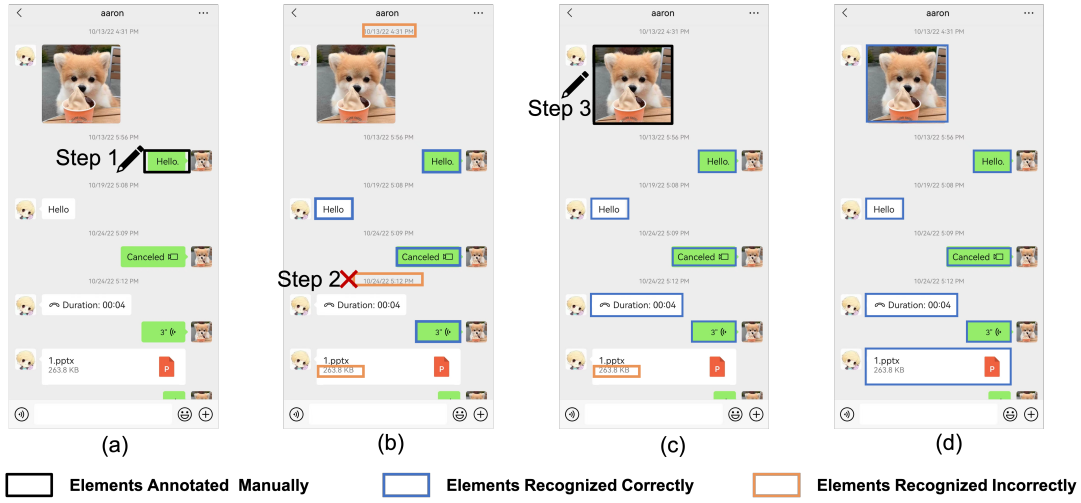


Fig. 9. The annotation process for recognizing all widgets of message content, using the feature set available in Appendix B.1.2. (a) The "Hello" widget was initially annotated. (b) The system recognized several widgets based on the annotated example. The annotator removed the "10/24/22 5:12 PM" widget, as it represents the message's time, not content. (c) The system corrected the misrecognition of message time and added recognition of the "Duration: 00:04" widget. The puppy image widget was not recognized and was manually added. (d) The system added recognition for the "1.pptx" widget, successfully recognizing all message content.

5.2 Offline Dataset Validation

We established an offline dataset to evaluate the performance of two existing methodologies and the IAM. We developed UIAD Models for 12 applications across 9 categories, as shown in Table 7. Our selection criteria aimed to include applications with rich data, diverse functionality, high user engagement, and strong cross-device interaction potential. Corresponding to these models' needs, we collected 6,413 pages and 116,516 widgets, which needed to be classified into 339 and 7,981 classes, respectively.

5.2.1 Performance of Existing Methodologies. We evaluated the following two existing methodologies.

- **Pre-defined heuristic rules**: A detailed implementation is available in Appendix B. This method has been utilized by prior interaction proxies [61]. We employed a five-fold cross-validation technique on our dataset.
- **Pre-trained data-driven models**: We employed Screen2Vec [35] with Euclidean distance for page classification, and the semantic-icon-classifier [41] for widget recognition. These two models were trained on the Rico dataset [17] and tested on our dataset.

As the results in Table 1 demonstrate, both pre-defined heuristic rules and pre-trained data-driven models exhibited limitations: the former's uniform rules couldn't cover every case, while the latter's classification criteria differed from the models' specifications.

5.2.2 Performance of the IAM. We evaluated the performance of the annotators who used the **IAM**. Unlike the above two methodologies, the IAM is a human-machine hybrid system for data annotation, rather than a stand-alone model. Its effectiveness is assessed through the following metrics:

Table 1. Classification Results.

Methodology	Page Classification			Widget Recognition		
	Precision	Recall	F1	Precision	Recall	F1
Pre-defined heuristic rules	0.720	0.958	0.822	0.612	0.790	0.689
Pre-trained data-driven models	0.628	0.978	0.765	0.570	0.473	0.517

- **Annotation Completeness**: Annotators are required to traverse all data and annotate necessary cases until 100% precision and recall are achieved, signifying the achievement of complete annotation accuracy through iterative human-machine interactions.
- **Annotation Efficiency**: It is reflected by the number of annotations needed to achieve complete accuracy.

It is important to note that the performance of these metrics is only influenced by the order of the cases in the dataset, and is independent of the individual annotators (assuming no errors are made during the annotation process). Consequently, we invited 10 annotators to individually handle all cases in the dataset. For each annotator, the data was shuffled randomly to ensure an unbiased evaluation.

The experimental results validate that by employing the IAM, multiple rounds of verification and annotation ensured that the recognition of all classes could be achieved with 100% precision and recall. Furthermore, IAM offers the following advantages.

- **Simplicity in generating strategies**: The decision trees exhibit a complex structure, averaging 11.27 nodes (sd=5.06) and a depth of 5.18 (sd=2.26). Manually deriving such a rule set is challenging, but by the IAM, it can be automatically extracted by adding a few instances.
- **High single annotation gain**: The average number of extra annotations for pages was 1.58 (sd=1.31) times and for widgets, it was 1.95 (sd=1.60) times. Interactive annotation leverages human decision-making to provide more representative examples, effectively avoiding the repeated annotation of similar cases that may occur in traditional machine learning. For instance, as shown in Figure 9(c), adding one single image is sufficient to learn a more refined classification strategy.
- **Dynamic adjustability**: Subdivision and merging frequently occur during model editing. Figure 9 illustrates an example of merging images and text into one class, which can be further subdivided based on the message sender. This level of flexibility is not supported by pre-defined rules and pre-trained models.

6 ENHANCED DESIGN FLEXIBILITY AND DIVERSE APPLICATIONS WITH THE IPMANAGER

In this section, we highlight the enhanced design flexibility and diverse applications provided by the IPManager. We demonstrate the capabilities of IPManager by creating three distinct user interfaces for WeChat.

6.1 Innovative Design Pattern

The innovation encompasses the following three main aspects.

6.1.1 Flexible Mapping Options. The reorganization of UI elements and various mapping options facilitate the development of flexible and versatile design strategies for new UIs. Many-to-one pages enable data delivery from small to large screens, one-to-many pages suitably project large to small screens, and many-to-many pages are commonly employed in constructing multi-modal distributed interfaces in AIoT scenarios.

6.1.2 Widget Customization. Widgets can be customized with interesting trigger patterns. Many-to-one widgets bind a new UI widget to multiple original widgets with similar semantics across different mobile applications, enabling a single action to trigger multiple application functions. One-to-many widgets allow the same original

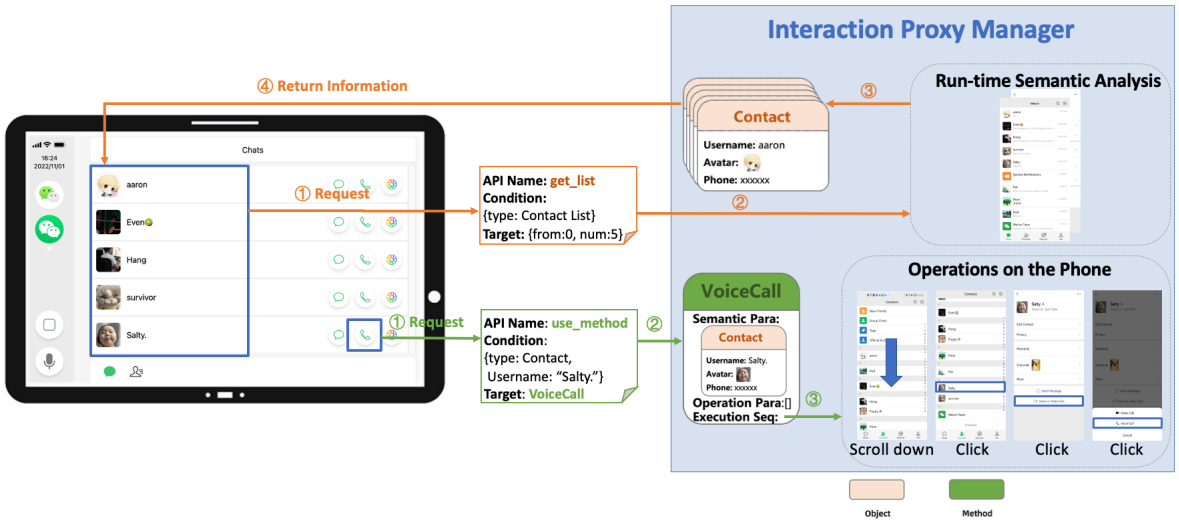


Fig. 10. Interaction Proxy system on the car display. The car display sends *get_list* and *use_method* API requests to the IPManager during runtime.

widgets to be triggered in multiple ways or modalities, such as initiating phone calls via touch, voice, or physical shortcut keys. Many-to-many widgets combine the features of the previous two, offering the potential for service combinations.

6.1.3 Tailoring Applications for Specific User Groups. The flat management of methods in the UIAD Model enables developers to create simplified application versions tailored for specific user groups, such as accessibility services and services for the elderly. This approach ensures that the applications address the unique needs and preferences of these user groups, improving usability and user experience.

6.1.4 Distributed Interfaces and Data Integration. The UIAD Model's support for distributed interfaces across devices enables users to leverage mobile phones as data sources and integrate personal data into different UIs for various scenarios. This feature allows for seamless data flow and user experience across multiple devices.

6.2 Application Examples

To demonstrate the three aspects mentioned above, we collaborated with the WeChat mobile application and invited three development teams to create three distinct user interfaces based on the UIAD Model: the tablet-sized car display GUI, the smartwatch GUI, and the voice user interface.

GUI reconstruction on the car display helps drivers interact with minimal attention cost. The car GUI system's layout is designed specifically for car displays, with the model APIs directly rendering the GUI. Casting from the phone to the car display exemplifies a "small to big" transformation, allowing content from multiple phone pages to be displayed on a single car display page, as shown in Figure 10.

GUI reconstruction on the watch improves user experience in high-mobility scenarios, such as checking messages while running. Casting to watches demonstrates a "big to small" transformation. The parameters required for API requests may originate from multiple pages on the watch, as shown in Figure 11.

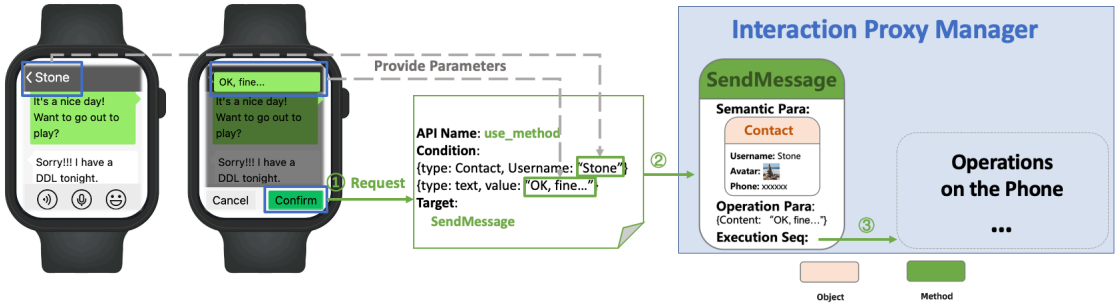


Fig. 11. Interaction Proxy system on the smartwatch. The smartwatch sends a *use_method* API request to the IPManager during runtime.

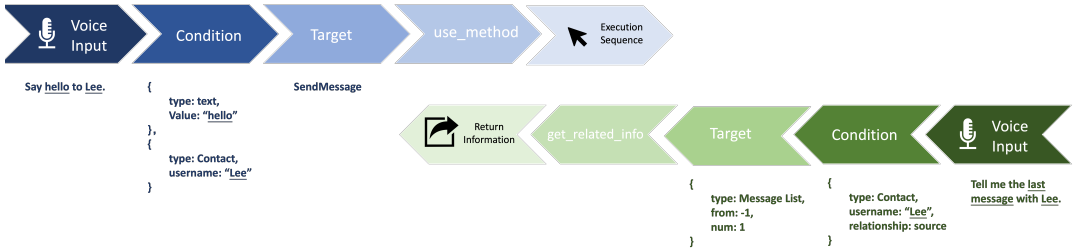


Fig. 12. Interaction Proxy system on the voice user interface.

The voice user interface is built on a modality where the machine completes tasks based on users' natural language commands. The developers employed Large Language Models to parse the commands into UIAD Model APIs and their corresponding parameters, enabling the creation of a simple voice user interface.

These examples show that our approach enables developers to create user interfaces that cater to different screen sizes, modalities, and user needs while supporting seamless data integration across devices.

7 WORKFLOW PERFORMANCE EVALUATION

Based on the aforementioned system design, we implemented the system and designed a user study to demonstrate the effectiveness of our approach. This study has been reviewed and approved by an appropriate Institutional Review Board (IRB).

7.1 Implementation

We implemented the IPManager, comprising the offline registration system and the run-time system, and developed a UI deployment tool to validate model usability.

7.1.1 Registration System. We developed an interactive annotation platform using Vue for model design and registration with the original GUI (Figure 13). A phone application obtains GUI layouts and captures screenshots using Android's Accessibility Service API and MediaProjection[6], while a Flask server manages data transfer and decision tree training for classification.

[6]https://developer.android.com/reference/android/media/projection/MediaProjection

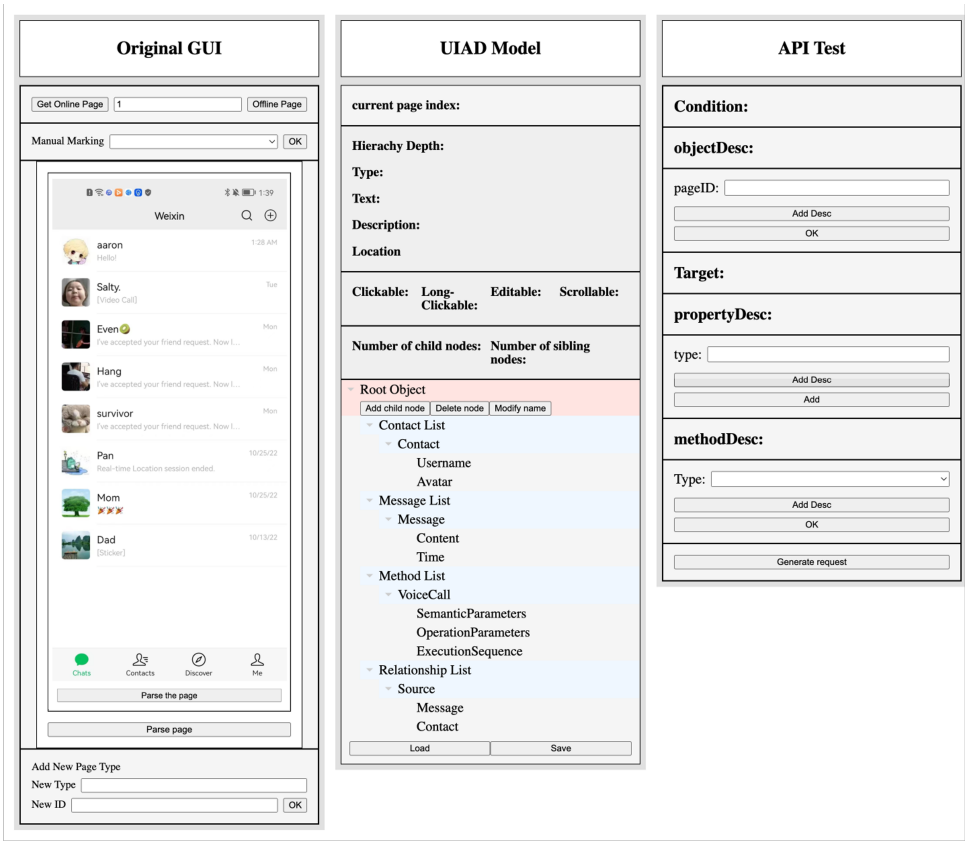


Fig. 13. The interactive annotation platform implementation.

7.1.2 Run-time System. A Flask server provides real-time semantic analysis and the model manager. It generates the UIAD Model and sends page-jumping instructions when necessary. The phone application sends page data to the server and receives instructions, which guide it to simulate on-screen operations and complete the jump via Android's Accessibility Service API.

7.1.3 Target-device UI Deployment. Given the designed target-device UI's HTML file, the deployment tool specifies the content source of UI elements by binding model API (Figure 14). This generates executable front-end code, allowing the GUI to run as web pages and support voice command input.

7.2 Procedure

The user study involved 4 distinct study groups corresponding to the following 4 steps.

7.2.1 Identifying Use Cases. We interviewed 3 experienced AIoT product managers (P1-P3) to identify functional requirements for various use cases. Based on their opinions, we designed 4 use cases and their corresponding applications.

7.2.2 Developing UIAD Models. We engaged 8 participants, including 4 junior programmers (with less than 2 years of programming experience) (P4-P7) and 4 non-programming students (P8-P11), to create UIAD models

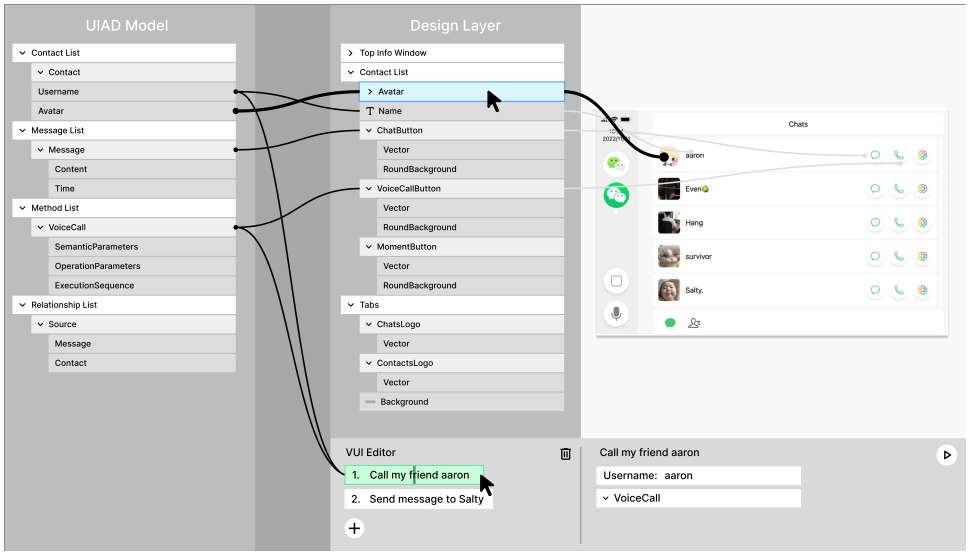


Fig. 14. The target-device UI deployment tool implementation.

for the 4 use cases. Each participant was presented with the use cases using a Latin square design sequence for counterbalancing. This resulted in a total of 32 models. Participants utilized our IPManager to design Model Specifications, register with the mobile interface, and test Model APIs.

7.2.3 Developing Target-Device UIs. We hired 8 professional designers (P12-P19) to create the required UI for each use case, tailored for the target devices. Using a Latin square design sequence, each designer used 4 UIAD models developed in the previous step. Then they bound each UI element to the corresponding model API and automatically deployed the new interfaces.

7.2.4 Experiencing Target-Device UIs. We invited 8 end-users (P20-P27), aged 18 to 26, to use 4 developed UIs in different use cases with their various devices. They provided feedback to assess the usability and effectiveness of the implemented applications on the target devices. The experiment lasted for two weeks, with each participant spending over 5 hours using each UI.

7.3 Results

We will analyze the results of each step of the user study.

7.3.1 Identifying Use Cases. During our discussions with product managers, our approach was validated, with P2 and P3 noting the model's adaptability based on personalized needs. P1 stated that the deployment on various devices can be achieved through our approach. They also expressed further expectations. P1 said *"I hope to support multi-device input and output when playing games and watching movies. However, the current solution seems to have limited ability for video streams."* P3 suggested incorporating data from other devices, such as health data from a smartwatch.

Additionally, they noted that our new approach would bring significant changes to product design ideas. P2 stated *"[This approach] clearly excels in multi-device adaptation, prompting us to focus on designing products with this aspect in mind, instead of targeting specific devices as in the case of existing smart homes and smart cockpits."* P1 added, *"We can easily consider separating information presentation and input, reducing user operation*

complexity, especially when user interaction is limited, such as while driving." P3 mentioned that in a home setting, a smartphone could be used for control while a larger screen displays a 3D representation of the home's status.

In the end, we selected 4 representative use cases from our interview to carry out the next steps of the user study, as shown in Table 2. The details of these use cases are provided in Appendix F.1.

Table 2. An overview of use cases.

Use Case ID	Description	Target-devices
C1	Home audio system	Smart speakers, large-screen TVs
C2	Map applications for vehicle head-up displays	HUDs
C3	Food ordering in mobile scenarios	Smartwatches
C4	Schedule and Express Delivery App Integration	Smartphones

Table 3. Subjective feedback from UIAD model developers. Full results are available in Table 13 in Appendix.

Statements	**Median**
1. You can understand how the UIAD Model works.	6
2. You think the annotation platform is easy to use.	5
3. You can design a good UIAD Model.	6.5
4. You can register the model with the original GUI.	6
5. You find the operation efficient.	6.5
6. You think the annotation platform is smart.	6.5
7. You think the UIAD model is not complicated.	5
8. You are willing to use our system.	6

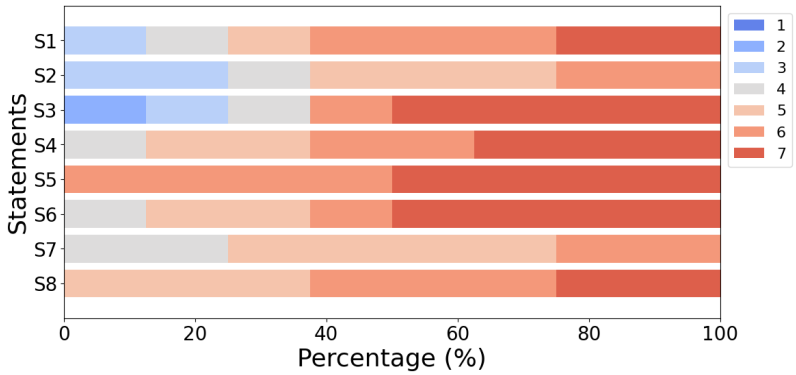


Fig. 15. Distribution of participant ratings for each statement in Table 3. Ratings range from 1 (strongly disagree) to 7 (strongly agree).

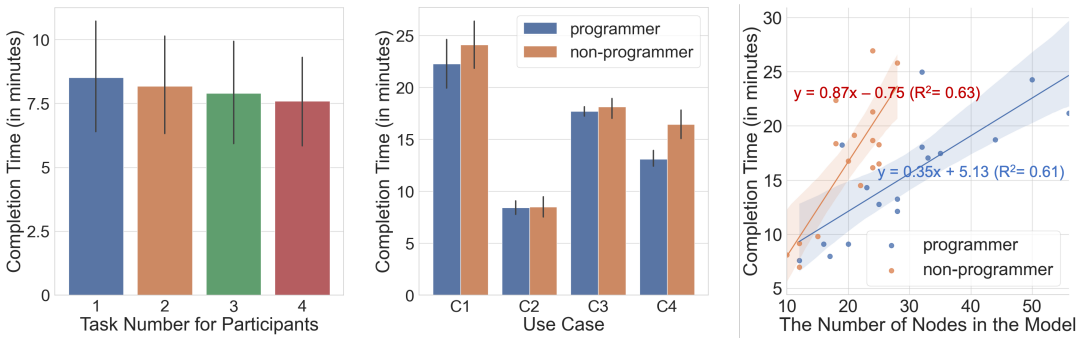


Fig. 16. Average time spent by participants on developing UIAD models. (a) Completion time of each task attempt. (b) Completion time of each use case. (c) The relationship between completion time and the number of nodes in the model. Error bars in (a) and (b) indicate standard deviation. The details of the statistical evaluation are provided in Appendix F.3.1.

7.3.2 Developing UIAD Models. We present the evaluation results of the participants' performance, as well as their subjective feedback using a 7-point Likert scale in Table 3 and Figure 15.

Participants reported that the learning cost was within tolerable limits (Statements 1, 2, 3, 4 and 7), with an average training time per participant of 35.97 minutes (sd = 2.79), using WeChat as the example. The success rate of independently completing model development was 90.63%, with the remaining 3 models being completed with minimal guidance. The programmers had a 100% independent completion rate, while the non-programmers had an 81.25% rate.

Most participants acknowledged that the model effectively reduced development costs (Statement 5) and were willing to use the system for multi-device interface deployment (Statement 8). As shown in Figure 16a, participants' proficiency improved with an increase in model development instances, resulting in reduced completion time (p-value = 0.006). Figure 16b shows a difference in completion time between programmers and non-programmers (p-value = 0.017), attributed to the model aligning more with programmers' thinking. P5 stated, *"The model is easy to understand from an Object-Oriented Programming (OOP) perspective."* Additionally, as seen in Figure 16c, the time spent developing a model positively correlated with its complexity, related to the requirements of the use case. Notably, the slopes of the fitted lines for programmers and non-programmers exhibited significant differences (p-value = 0.010), indicating that programmers could adapt in less time as the model complexity increased.

Notably, compared to the existing workflows where senior programmers' estimates of development time would require at least dozens of hours, our approach significantly reduces development time and even supports non-programmers in developing.

Participants appreciated our approach. P6 believed that the model is released from the specific settings of different scenarios and is configured as a more universal framework. P9 took it as a more concise, abstract, and well-designed outer layer of applications. Participants also thought the annotation platform was very smart (Statement 6). P10 found the overall experience smooth and the configuration steps interconnected. However, P8 still hoped for further improvements: *"Currently, the program strictly follows a step-by-step configuration, and each attribute match must be exact... More user-friendly interaction methods can be considered."* Similarly, P10 expressed a desire for more intelligent fuzzy matching.

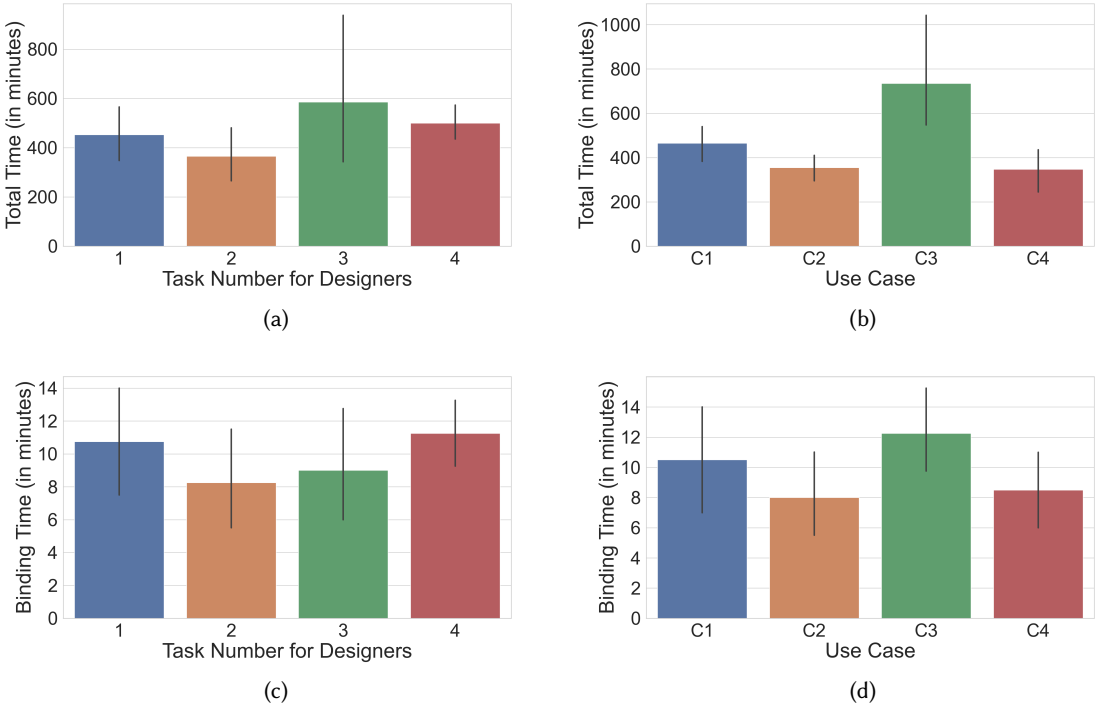


Fig. 17. Average time spent by designers on developing target-device UIs. (a) Total time spent on each task attempt. (b) Total time spent on different use cases. (c) Time spent on binding the API for each task attempt. (d) Time spent on binding the API for different use cases. Error bars indicate standard deviation. The details of the statistical evaluation are provided in Appendix F.3.2.

Participants also identified the potential of our approach. P4 said *"Constructing applications for various devices with the model as the kernel will become a new paradigm."* P5 suggested that it could be extended to gesture-oriented user interfaces, which can save a lot of code. P6 expected that *"This would enable parallel operation across different devices."* P7 expressed a strong demand for porting more applications (such as their frequently used running app) to the watch and interconnecting multiple AIoT devices at home through one terminal.

However, participants sometimes encountered ambiguities. P7 said *"I'm not sure whether a concept should be further split out."* P6 expressed concerns about *"whether the pairing and annotation are comprehensive and complete."* Actually, at the current stage, participants can make these decisions freely. When new requirements or issues arise later, they can come back and modify accordingly.

7.3.3 Developing Target-Device UIs. We evaluated the designers' development costs in the study. The average training time per participant was 12.13 minutes (sd = 1.83), using WeChat as the example. The success rate of completing tasks independently was 90.63%, with the remaining 3 failures resolved after clarification of requirements.

The total time (Figure 17a) and the time for binding the model APIs (Figure 17c) did not change significantly with the number of tasks the participants have completed (p-value = 0.204 and 0.749, respectively). Instead, it is strongly correlated with use cases (Figure 17b (p-value = 0.006) and Figure17d) (p-value = 0.039). The average

time spent on API binding was 4.91 (sd = 2.37) minutes, only accounting for 2.06% of the total time, indicating that most time was spent on UI design.

Participants reported reduced time compared to existing workflows. P12 said *"It reduced the delivery cost with the front-end development department and there was no need for the original work on organizing the information hierarchy of the page."* P14 said *"The tree is better understood than the product manager documents."* P17 and P19 noted significant time savings when the information structure and function were well-defined. P13 said *"The design work is almost the same as before. The main learning cost is in learning the principle of the model but completing the binding is very efficient."* P15 appreciated the run-time access to data via APIs for UI validation, saving them from manually inputting data.

Some participants sometimes encountered confusion, but resolved issues easily. P15 said *"Some nodes [in the model] appear somewhat ambiguous and need further explanation."* P18 said *"Node names [in the model] can be unclear, but understanding improves when related to the original interface."* P14 stated that designers still needed to build a design hierarchy upon understanding the model.

Participants found our approach met their design requirements, but it did impact designers' thinking and behavior to some extent. Unlike the traditional workflow, our approach encourages independent work, reducing interaction between designers and product managers, which can lead to occasional uncertainty in design decisions. For instance, P17 expressed uncertainty regarding the display of ratings, *"whether they should be stars or specific scores"*. P16 expressed confusion on whether the term "find" should be interpreted as a search box or a search icon. P18 said *"Sometimes I am unsure if my confusion is because I have not fully understood the model."* While our approach permits design freedom, participants still sought confirmation from product managers due to unfamiliarity with the new workflow.

7.3.4 Experiencing Target-Device UIs. The target-device UI examples corresponding to each use case are illustrated in Figures 21, 22, 23, 24 in the Appendix. In this study, we evaluated the performance of the developed 32 UIs and the participants' subjective feedback using a 7-point Likert scale, as shown in Table 4 and Figure 18.

Table 4. Subjective feedback from end-users. Full results are available in Table 14 in Appendix.

Statements	**Median**
1. You are satisfied with the target-device UIs.	6
2. The target-device UIs are easy to navigate and use.	7
3. The design of the target-device UIs are clear and understandable.	6
4. The target-device UIs are responsive and fast.	7
5. There are no errors or issues while using the UIs.	6
6. You are willing to use these target-device UIs.	6

From the usage logs of participants (P20-P27), we collected 16,704 API call logs, boasting a success rate of 99.78%, with failures primarily due to network issues. The average response time for each API call was 413.73 milliseconds (sd = 353.05), varying across different use cases (Figure 19a, p-value < 0.001).

As shown in Figure 19b, the total time spent comprises five segments, each showing significant differences from the others (p-value < 0.001). Page loading time accounted for the largest proportion (56.48%), followed by network transfer time (24.63%). Semantic analysis time averaged a mere 36.36 milliseconds, with relatively small fluctuations (sd = 26.59). Waiting time, the duration required for a new page to stabilize with a ready layout and image, depends on factors like phone performance, network conditions, page content, and application optimization. In our experiment, we assigned a fixed empirical value as the waiting time after each operation. With a 50-millisecond waiting time, task success rates reached 100%; without it, they dropped to 67.64%.

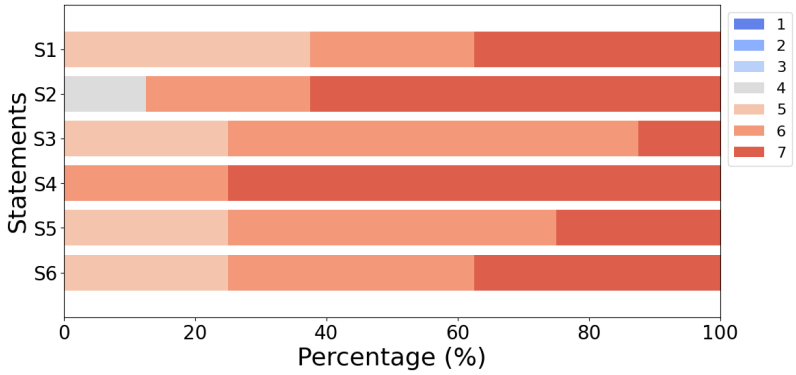
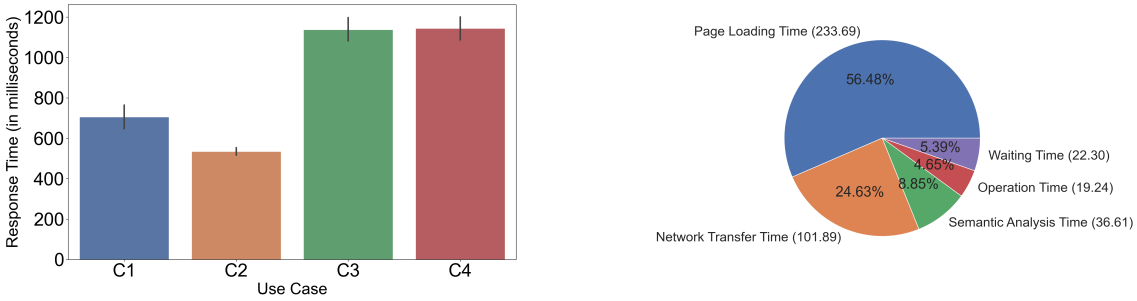


Fig. 18. Distribution of participant ratings for each statement in Table 3. Ratings range from 1 (strongly disagree) to 7 (strongly agree).



(a) Average response time for each API call in different use cases. Error bars indicate standard deviation

(b) The distribution of the response time.

Fig. 19. The average response time for each API call. The details of the statistical evaluation are provided in Appendix F.3.3.

In reality, due to the multi-thread, cache, and background updates, a considerable portion of the time is not user-perceived. This is particularly evident in Use Case 2 and Case 4, featuring many background automatic updates not requiring user intervention.

Table 4 shows that most participants found the developed target-device UIs satisfactory in terms of ease of use (Statements 2 and 3), robustness (Statement 5), and perceived delay (Statement 4). The enthusiasm to use these UIs was evident among participants (Statements 1 and 6). For example, P22 stated *"The ordering app wasn't previously compatible with watches or voice, but now I can order while I am riding bikes. It is amazing."* P25 said *"[Use Case 4] The combination of the functions of two apps is fantastic."* P21 and P24 liked the ability to use their familiar navigation software on the HUD. Interestingly, P27 said *"Removing ads is great."* Despite these positive remarks, there were suggestions and concerns. P27 expressed a desire for faster startup times. Privacy and security issues were also noted. P26 stated *"I'm worried about having my phone content collected."*. P23 said *"If I accidentally display private information on the big screen, it wouldn't be good."*

Additionally, new requirements also surfaced that were not considered during the development phase. These emerged in two areas: the first pertained to overlooked original phone data, as exemplified by P20's observation that certain food combination options weren't included in the ordering interface. The second area was user-generated requirements. A case in point is P24's expectation for a one-click feature to copy a song from chat, paste it into a music app, and play it directly. To address these issues, we informed participants that our approach allows for incremental updates and low-cost modifications to accommodate new requirements, thereby enhancing functionality.

8 DISCUSSION

In this section, we discuss our approach based on the findings from the user study conducted above.

8.1 Design Guidelines

To enhance speed and robustness, we propose the following design guidelines for utilizing the model:

- Minimize the number of page jumps on the phone, as they significantly increase time usage and the risk of unstable page loading. Try to avoid any cache miss when calling the API. For instance, the progression of the new UI should align as closely as possible with the natural flow of context switching, mitigating cross-level or frequent jumps of the context cursor on the model tree.
- Be aware of spontaneous updates to the original GUI - changes that are not user-driven, such as "receiving a new message". Such updates may not always be tracked by the system, especially if the phone isn't displaying the "new message found" page at that time. To counter this, the new interface system should proactively call the API to monitor updates.
- Keep in mind that mapping new UI elements and APIs primarily supports basic cases. If the new UI requires more complex display logic, such as a multistep API with nested parameters, developers will need to manage the new interface state and relevant parameter variables within their own code.

8.2 Capabilities of the UIAD Model

Compared to the preliminary Interaction Proxy[61], which is still in the proof-of-concept stage and not fully implemented, our proposed UIAD model offers several capabilities by efficiently decoupling a mobile application's UI and service:

Understandable Description and Efficient APIs: Unlike existing proxies [58, 62] that directly map the original UI to the target one, leading to repetitive handling of the original UI's complexity and dynamicity, the UIAD model offers a more holistic solution. It generates a structured and understandable description of an application's service semantics that is comprehensible to both programmers and non-programmers. This model, applicable across different target devices, offers efficient APIs that allow developers to focus on application functionality, unencumbered by the original UI's intricacies. This approach simplifies the development of new interface systems. Further, with the corresponding run-time support, these APIs guarantee the high performance of new interface systems.

Robust UI Reconstruction: Existing interface mapping algorithms, which use heuristic rules to compute layout signatures based on developers' specifications, are rudimentary. As Zhang et al. pointed out [61], *"It can therefore be challenging to reason about the state of an app."* To overcome this challenge, we employ the IAM to build the model that ensures a low annotation burden while maintaining high recognition accuracy for pages and widgets. This approach guarantees the robustness of new interface systems.

Complex Functions Management: In comparison to single-task-oriented programming by demonstration[33], the UIAD Model encompasses the full range of an application's functionality. It empowers developers to freely organize and invoke diverse APIs, such as chaining and nesting, to support complex tasks. Given its adeptness

at handling intricate semantic relationships, the model is particularly suited for the heavily engineered task of UI reconstruction.

Crowd Engagement: As evidenced by our user study, the UIAD Model's ease of comprehension extends to both programmers and non-programmers. This accessibility broadens user engagement, reduces the development threshold and cost, and thereby enhances its adoption and effectiveness.

8.3 Feasibility of the UIAD Model in the Future

We discuss the future feasibility of the proposed model from three aspects:

Technical Feasibility: The maturity of Robotic Process Automation (RPA) in PC platforms validates the potential of automating user interface interactions. Coupled with AI research into web page understanding, extending these technologies to smartphones supports the prospects for automated model construction.

Implementation Feasibility: Our user study results confirm that our model meets various departmental needs and effectively curtails development costs. Primarily aimed at functional migration, it can support currently incompatible applications such as video streaming or gaming with the simple addition of video transmission technology.

Market Feasibility: The limited number of applications for AIoT devices like smartwatches presents a challenge to ecosystem development. Our model can expedite the migration of existing applications to these platforms, thereby fostering long-term growth and diversity.

9 LIMITATIONS & FUTURE WORK

This section highlights our work's limitations and proposes potential future directions.

Enhancing Data Loading: Our current reliance on Android's Accessibility Service API results in slower data retrieval and limited availability. Future work could explore acquiring data directly from the phone system, which could potentially reduce page loading time and enhance the semantic richness of the model.

Automating Model Generation: Our goal is to automate the process of generating UIAD Models from existing mobile GUIs through reverse engineering. The main challenge is the subjective nature of the semantic analysis, which complicates controlling the model structure without human intervention, especially when handling complex semantics in various applications. Accurately representing intricate nested structures and relationships is crucial for optimal UI generation. We also need to take into account privacy rights, data importance, and additional functions when dealing with advanced semantics. The integration of these factors may help automated model generation to better cope with real-world complexities, leading to more efficient and user-friendly interfaces.

Automating User Interface Construction: Our current application requires human input to determine the new UI layout and which APIs to call. We aim to generate UIAD Models from existing device interfaces, learn the transformation from models to different interfaces, and utilize context modeling [28] along with existing model-based UI deployment tools to automatically generate optimal user interfaces, potentially utilizing large models for lower development costs.

10 CONCLUSION

This paper presents Interaction Proxy Manager, which decouples interface and semantics for reliable UI reconstruction. Therein, the UIAD Model is defined based on the semantic analysis of existing mobile applications and offers APIs for alternative device systems. IPManager functions through two distinct phases: the offline phase and the run-time phase. In the offline phase, the Interactive Annotation Mechanism assists annotators in registering with mobile pages and learns the classification strategies for pages or widgets. It effectively reduces annotation difficulty and cost, while ensuring recognition accuracy, as validated by an offline dataset.

Subsequently, the run-time system generates the UIAD Model using learned strategies, providing information or invoking methods based on requests received from new interface systems. We showcased IPManager's design flexibility and application diversity by developing three applications that extended phone services to car displays, smartwatches, and voice-oriented interfaces. We conducted a user study to evaluate the entire workflow, involving product managers, developers, designers, and end-users. The result validates the usability, efficiency, comprehensibility, and robustness of our approach. We anticipate that our work will contribute to the broader field of UI reconstruction, facilitating the deployment of more services across diverse devices and ultimately enriching the user experience.

ACKNOWLEDGMENTS

This work is supported by the Natural Science Foundation of China under Grant No. 62132010, Beijing Key Lab of Networked Multimedia, the Institute for Guo Qiang, Tsinghua University, the Institute for Artificial Intelligence, Tsinghua University (THUAI), the 2025 Key Technological Innovation Program of Ningbo City under Grant No. 2022Z080, as well as by the Beijing Municipal Science & Technology Commission and the Administrative Commission of Zhongguancun Science Park under No. Z221100006722018.

REFERENCES

[1] Silvia Abrahão, Emilio Insfran, Arthur Sluÿters, and Jean Vanderdonckt. 2021. Model-based intelligent user interface adaptation: challenges and future directions. *Software and Systems Modeling* 20, 5 (2021), 1335–1349. https://doi.org/10.1007/s10270-021-00909-7

[2] Silvia Abrahão, Francis Bourdeleau, Betty Cheng, Sahar Kokaly, Richard Paige, Harald Stöerrle, and Jon Whittle. 2017. User Experience for Model-Driven Engineering: Challenges and Future Directions. In *2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. 229–236. https://doi.org/10.1109/MODELS.2017.5

[3] Pierre A. Akiki, Arosha K. Bandara, and Yijun Yu. 2016. Engineering Adaptive Model-Driven User Interfaces. *IEEE Transactions on Software Engineering* 42, 12 (2016), 1118–1147. https://doi.org/10.1109/TSE.2016.2553035

[4] Pedro Albuquerque Santos, Rui Pedro da Costa Porfírio, Rui Neves Madeira, and Nuno Correia. 2021. Computational Framework to Support Development of Applications Running on Multiple Co-Located Devices. In *Companion of the 2021 ACM SIGCHI Symposium on Engineering Interactive Computing Systems* (Virtual Event, Netherlands) *(EICS '21)*. Association for Computing Machinery, New York, NY, USA, 63–69. https://doi.org/10.1145/3459926.3464758

[5] Lyan Alwakeel and Kevin Lano. 2020. *Model Driven Development of Mobile Applications.*

[6] Domenico Amalfitano, Vincenzo Riccio, Ana C. R. Paiva, and Anna Rita Fasolino. 2018. Why does the orientation change mess up my Android application? From GUI failures to code faults. *Software Testing, Verification and Reliability* 28, 1 (Jan 2018), e1654. https://doi.org/10.1002/stvr.1654

[7] Saleema Amershi. 2011. Designing for Effective End-User Interaction with Machine Learning. In *Proceedings of the 24th Annual ACM Symposium Adjunct on User Interface Software and Technology* (Santa Barbara, California, USA) *(UIST '11 Adjunct)*. Association for Computing Machinery, New York, NY, USA, 47–50. https://doi.org/10.1145/2046396.2046416

[8] Saleema Amershi, Maya Cakmak, William Bradley Knox, and Todd Kulesza. 2014. Power to the People: The Role of Humans in Interactive Machine Learning. *AI Magazine* 35, 4 (Dec. 2014), 105–120. https://doi.org/10.1609/aimag.v35i4.2513

[9] J. R. Anderson. 1996. Act: a Simple Theory Of Complex Cognition. *American Psychologist* 51 (1996), 355–365. Issue 4. https://doi.org/10.1037/0003-066x.51.4.355

[10] Gary Ang and Ee-Peng Lim. 2022. Learning and Understanding User Interface Semantics from Heterogeneous Networks with Multimodal and Positional Attributes. *ACM Trans. Interact. Intell. Syst.* (dec 2022). https://doi.org/10.1145/3578522 Just Accepted.

[11] Carlo Bernaschina, Sara Comai, and Piero Fraternali. 2017. Online Model Editing, Simulation and Code Generation for Web and Mobile Applications. In *2017 IEEE/ACM 9th International Workshop on Modelling in Software Engineering (MiSE)*. 33–39. https://doi.org/10.1109/MiSE.2017.1

[12] G. Botturi, E. Ebeid, F. Fummi, and D. Quaglia. 2013. Model-driven design for the development of multi-platform smartphone applications. In *Proceedings of the 2013 Forum on specification and Design Languages (FDL)*. 1–8.

[13] Frederik Brudy, Christian Holz, Roman Rädle, Chi-Jui Wu, Steven Houben, Clemens Nylandsted Klokmose, and Nicolai Marquardt. 2019. Cross-Device Taxonomy: Survey, Opportunities and Challenges of Interactions Spanning Across Multiple Devices. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems* (Glasgow, Scotland Uk) *(CHI '19)*. Association for Computing Machinery, New York, NY, USA, 1–28. https://doi.org/10.1145/3290605.3300792

[14] 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

[15] Jieshan Chen, Chunyang Chen, Zhenchang Xing, Xiwei Xu, Liming Zhu, Guoqiang Li, and Jinshui Wang. 2020. 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, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 322–334. https://doi.org/10.1145/3377811.3380327

[16] 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?. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 1202–1214. https://doi.org/10.1145/3368089.3409691

[17] 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 ACM Symposium on User Interface Software and Technology* (Québec City, QC, Canada) *(UIST '17)*. Association for Computing Machinery, New York, NY, USA, 845–854. https://doi.org/10.1145/3126594.3126651

[18] John J. Dudley and Per Ola Kristensson. 2018. A Review of User Interface Design for Interactive Machine Learning. *ACM Trans. Interact. Intell. Syst.* 8, 2, Article 8 (jun 2018), 37 pages. https://doi.org/10.1145/3185517

[19] Alex Endert, Patrick Fiaux, and Chris North. 2012. Semantic Interaction for Visual Text Analytics. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Austin, Texas, USA) *(CHI '12)*. Association for Computing Machinery, New York, NY, USA, 473–482. https://doi.org/10.1145/2207676.2207741

[20] Jerry Alan Fails and Dan R. Olsen. 2003. Interactive Machine Learning. In *Proceedings of the 8th International Conference on Intelligent User Interfaces* (Miami, Florida, USA) *(IUI '03)*. Association for Computing Machinery, New York, NY, USA, 39–45. https://doi.org/10.1145/604045.604056

[21] Shirin Feiz, Jason Wu, Xiaoyi Zhang, Amanda Swearngin, Titus Barik, and Jeffrey Nichols. 2022. Understanding Screen Relationships from Screenshots of Smartphone Applications. In *27th International Conference on Intelligent User Interfaces* (Helsinki, Finland) *(IUI '22)*. Association for Computing Machinery, New York, NY, USA, 447–458. https://doi.org/10.1145/3490099.3511109

[22] James Fogarty, Desney Tan, Ashish Kapoor, and Simon Winder. 2008. CueFlik: Interactive Concept Learning in Image Search. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Florence, Italy) *(CHI '08)*. Association for Computing Machinery, New York, NY, USA, 29–38. https://doi.org/10.1145/1357054.1357061

[23] D. Thevenin G. Calvary, J. Coutaz. 2002. The CAMELEON Reference Framework, CAMELEON Project. September 2002, available at http://giove.isti.cnr.it/projects/cameleon/pdf/cameleon%20d1.1refframework.pdf.

[24] Sebastian Gehrmann, Hendrik Strobelt, Robert Krüger, Hanspeter Pfister, and Alexander M. Rush. 2020. Visual Interaction with Deep Learning Models through Collaborative Semantic Inference. *IEEE Transactions on Visualization and Computer Graphics* 26, 1 (2020), 884–894. https://doi.org/10.1109/TVCG.2019.2934595

[25] Mohammed Gomaa, Akram Salah, and Syed Rahman. 2005. Towards a better model based user interface development environment: A comprehensive survey. *Proceedings of MICS* 5 (2005), 2.

[26] Xuan Guo, Qi Yu, Rui Li, Cecilia Ovesdotter Alm, Cara Calvelli, Pengcheng Shi, and Anne Haake. 2016. an expert-in-the-loop paradigm for learning medical image grouping. (2016), 477–488. https://doi.org/10.1007/978-3-319-31753-3_38

[27] Shih-Wen Huang, Pei-Fen Tu, Wai-Tat Fu, and Mohammad Amanzadeh. 2013. Leveraging the Crowd to Improve Feature-Sentiment Analysis of User Reviews. In *Proceedings of the 2013 International Conference on Intelligent User Interfaces* (Santa Monica, California, USA) *(IUI '13)*. Association for Computing Machinery, New York, NY, USA, 3–14. https://doi.org/10.1145/2449396.2449400

[28] Muhammad Waseem Iqbal, Nadeem Ahmad Ch, Syed Khuram Shahzad, Muhammad Raza Naqvi, Babar Ayub Khan, and Zulfiqar Ali. 2021. User Context Ontology for Adaptive Mobile-Phone Interfaces. *IEEE Access* 9 (2021), 96751–96762. https://doi.org/10.1109/ACCESS.2021.3095300

[29] Ajay Kumar Jha, Sunghee Lee, and Woo Jin Lee. 2019. An empirical study of configuration changes and adoption in Android apps. *Journal of Systems and Software* 156 (2019), 164–180. https://doi.org/10.1016/j.jss.2019.06.095

[30] Sunjae Lee, Hayeon Lee, Hoyoung Kim, Sangmin Lee, Jeong Woon Choi, Yuseung Lee, Seono Lee, Ahyeon Kim, Jean Young Song, Sangeun Oh, et al. 2021. FLUID-XP: flexible user interface distribution for cross-platform experience. In *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking*. 762–774.

[31] Sunjae Lee, Hayeon Lee, Hoyoung Kim, Sangmin Lee, Jeong Woon Choi, Yuseung Lee, Seono Lee, Ahyeon Kim, Jean Young Song, Sangeun Oh, Steven Y. Ko, and Insik Shin. 2021. FLUID-XP: Flexible User Interface Distribution for Cross-Platform Experience. In *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking* (New Orleans, Louisiana) *(MobiCom '21)*. Association for Computing Machinery, New York, NY, USA, 762–774. https://doi.org/10.1145/3447993.3483245

[32] Valeria Lelli, Arnaud Blouin, and Benoit Baudry. 2015. Classifying and Qualifying GUI Defects. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. 1–10. https://doi.org/10.1109/ICST.2015.7102582

[33] Toby Jia-Jun Li, Amos Azaria, and Brad A. Myers. 2017. SUGILITE: Creating Multimodal Smartphone Automation by Demonstration. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems* (Denver, Colorado, USA) *(CHI '17)*. Association for Computing Machinery, New York, NY, USA, 6038–6049. https://doi.org/10.1145/3025453.3025483

[34] Toby Jia-Jun Li, Jingya Chen, Haijun Xia, Tom M. Mitchell, and Brad A. Myers. 2020. Multi-Modal Repairs of Conversational Breakdowns in Task-Oriented Dialogs. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology* (Virtual Event, USA) *(UIST '20)*. Association for Computing Machinery, New York, NY, USA, 1094–1107. https://doi.org/10.1145/3379337.3415820

[35] Toby Jia-Jun Li, Lindsay Popowski, Tom Mitchell, and Brad A Myers. 2021. Screen2Vec: Semantic Embedding of GUI Screens and GUI Components. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems* (Yokohama, Japan) *(CHI '21)*. Association for Computing Machinery, New York, NY, USA, Article 578, 15 pages. https://doi.org/10.1145/3411764.3445049

[36] Toby Jia-Jun Li, Marissa Radensky, Justin Jia, Kirielle Singarajah, Tom M. Mitchell, and Brad A. Myers. 2019. PUMICE: A Multi-Modal Agent That Learns Concepts and Conditionals from Natural Language and Demonstrations. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology* (New Orleans, LA, USA) *(UIST '19)*. Association for Computing Machinery, New York, NY, USA, 577–589. https://doi.org/10.1145/3332165.3347899

[37] Toby Jia-Jun Li and Oriana Riva. 2018. Kite: Building Conversational Bots from Mobile Apps. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services* (Munich, Germany) *(MobiSys '18)*. Association for Computing Machinery, New York, NY, USA, 96–109. https://doi.org/10.1145/3210240.3210339

[38] Yang Li, Jiacong He, Xin Zhou, Yuan Zhang, and Jason Baldridge. 2020. mapping natural language instructions to mobile ui action sequences. (2020). https://doi.org/10.18653/v1/2020.acl-main.729

[39] Yang Li, Gang Li, Luheng He, Jingjie Zheng, Hong Li, and Zhiwei Guan. 2020. widget captioning: generating natural language Description For Mobile User Interface Elements. (2020). https://doi.org/10.18653/v1/2020.emnlp-main.443

[40] Quentin Limbourg, Jean Vanderdonckt, Benjamin Michotte, Laurent Bouillon, and Murielle Florins. 2004. USIXML: A User Interface Description Language Supporting Multiple Levels of Independence.. In *ICWE Workshops*. 325–338.

[41] Thomas F. Liu, Mark Craft, Jason Situ, Ersin Yumer, Radomir Mech, and Ranjitha Kumar. 2018. Learning Design Semantics for Mobile Apps. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology* (Berlin, Germany) *(UIST '18)*. Association for Computing Machinery, New York, NY, USA, 569–579. https://doi.org/10.1145/3242587.3242650

[42] Gerrit Meixner, Fabio Paternò, and Jean Vanderdonckt. 2011. Past, Present, and Future of Model-Based User Interface Development. *i-com* 10, 3 (2011), 2–11. https://doi.org/10.1524/icom.2011.0026

[43] Kevin Moran, Boyang Li, Carlos Bernal-Cárdenas, Dan Jelf, and Denys Poshyvanyk. 2018. Automated Reporting of GUI Design Violations for Mobile Apps. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 165–175. https://doi.org/10.1145/3180155.3180246

[44] 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

[45] Lihang Pan, Chun Yu, JiaHui Li, Tian Huang, Xiaojun Bi, and Yuanchun Shi. 2022. Automatically Generating and Improving Voice Command Interface from Operation Sequences on Smartphones. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems* (New Orleans, LA, USA) *(CHI '22)*. Association for Computing Machinery, New York, NY, USA, Article 208, 21 pages. https://doi.org/10.1145/3491102.3517459

[46] Fabio Paternò. 2020. Concepts and Design Space for a Better Understanding of Multi-Device User Interfaces. *Univers. Access Inf. Soc.* 19, 2 (jun 2020), 409–432. https://doi.org/10.1007/s10209-019-00650-5

[47] André Rodrigues, André Santos, Kyle Montague, and Tiago Guerreiro. 2017. Improving Smartphone Accessibility with Personalizable Static Overlays. In *Proceedings of the 19th International ACM SIGACCESS Conference on Computers and Accessibility* (Baltimore, Maryland, USA) *(ASSETS '17)*. Association for Computing Machinery, New York, NY, USA, 37–41. https://doi.org/10.1145/3132525.3132558

[48] Ayoub Sabraoui, Anas Abouzahra, Karim Afdel, and Mustapha Machkour. 2019. MDD Approach for Mobile Applications Based On DSL. In *2019 International Conference of Computer Science and Renewable Energies (ICCSRE)*. 1–6. https://doi.org/10.1109/ICCSRE.2019.8807572

[49] Egbert Schlungbaum. 1996. *Model-based user interface software tools-current state of declarative models.* Technical Report. Georgia Institute of Technology.

[50] Henoc Soude and Kefil Koussonda. 2022. A Model Driven Approach for Unifying user Interfaces Development. *International Journal of Advanced Computer Science and Applications* 13, 7 (2022). https://doi.org/10.14569/IJACSA.2022.01307107

[51] Hallvard Trætteberg. 2002. *Model-based User Interface Design.* Ph. D. Dissertation. http://hdl.handle.net/11250/249669

[52] Jean M. Vanderdonckt and François Bodart. 1993. Encapsulating Knowledge for Intelligent Automatic Interaction Objects Selection. In *Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems* (Amsterdam, The Netherlands) *(CHI '93)*. Association for Computing Machinery, New York, NY, USA, 424–429. https://doi.org/10.1145/169059.169340

[53] Tom Veuskens, Kris Luyten, and Raf Ramakers. 2020. Rataplan: Resilient Automation of User Interface Actions with Multi-Modal Proxies. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 4, 2, Article 60 (jun 2020), 23 pages. https://doi.org/10.1145/3397329

[54] Byron C. Wallace, Kevin Small, Carla E. Brodley, Joseph Lau, and Thomas A. Trikalinos. 2012. Deploying an Interactive Machine Learning System in an Evidence-Based Practice Center: Abstrackr. In *Proceedings of the 2nd ACM SIGHIT International Health Informatics Symposium* (Miami, Florida, USA) *(IHI '12)*. Association for Computing Machinery, New York, NY, USA, 819–824. https://doi.org/10.1145/2110363.2110464

[55] Bryan Wang, Gang Li, and Yang Li. 2023. Enabling Conversational Interaction with Mobile UI Using Large Language Models. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems* (Hamburg, Germany) *(CHI '23)*. Association for Computing Machinery, New York, NY, USA, Article 432, 17 pages. https://doi.org/10.1145/3544548.3580895

[56] Bryan Wang, Gang Li, Xin Zhou, Zhourong Chen, Tovi Grossman, and Yang Li. 2021. Screen2Words: Automatic Mobile UI Summarization with Multimodal Learning. In *The 34th Annual ACM Symposium on User Interface Software and Technology* (Virtual Event, USA) *(UIST '21)*. Association for Computing Machinery, New York, NY, USA, 498–510. https://doi.org/10.1145/3472749.3474765

[57] Jason Wu, Xiaoyi Zhang, Jeff Nichols, and Jeffrey P Bigham. 2021. Screen Parsing: Towards Reverse Engineering of UI Models from Screenshots. In *The 34th Annual ACM Symposium on User Interface Software and Technology* (Virtual Event, USA) *(UIST '21)*. Association for Computing Machinery, New York, NY, USA, 470–483. https://doi.org/10.1145/3472749.3474763

[58] Jian Xu, Qingqing Cao, Aditya Prakash, Aruna Balasubramanian, and Donald E Porter. 2017. Uiwear: Easily adapting user interfaces for wearable devices. In *Proceedings of the 23rd annual international conference on mobile computing and networking*. 369–382. https://doi.org/10.1145/3117811.3117819

[59] Seid Muhie Yimam, Chris Biemann, Ljiljana Majnaric, Šefket Šabanović, and Andreas Holzinger. 2015. Interactive and Iterative Annotation for Biomedical Entity Recognition. In *Brain Informatics and Health,Lecture Notes in Computer Science*. 347–357. https://doi.org/10.1007/978-3-319-23344-4_34

[60] 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. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. https://doi.org/10.1145/3411764.3445186

[61] Xiaoyi Zhang, Anne Spencer Ross, Anat Caspi, James Fogarty, and Jacob O Wobbrock. 2017. Interaction proxies for runtime repair and enhancement of mobile application accessibility. In *Proceedings of the 2017 CHI conference on human factors in computing systems*. 6024–6037. https://doi.org/10.1145/3025453.3025846

[62] Xiaoyi Zhang, Anne Spencer Ross, and James Fogarty. 2018. Robust Annotation of Mobile Application Interfaces in Methods for Accessibility Repair and Enhancement. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*. https://doi.org/10.1145/3242587.3242616

[63] Xiaoyi Zhang, Tracy Tran, Yuqian Sun, Ian Culhane, Shobhit Jain, James Fogarty, and Jennifer Mankoff. 2018. Interactiles: 3D Printed Tactile Interfaces to Enhance Mobile Touchscreen Accessibility. In *Proceedings of the 20th International ACM SIGACCESS Conference on Computers and Accessibility*. https://doi.org/10.1145/3234695.3236349

[64] Zhilan Zhou, Jian Xu, Aruna Balasubramanian, and Donald E. Porter. 2020. A Survey of Patterns for Adapting Smartphone App UIs to Smart Watches. In *22nd International Conference on Human-Computer Interaction with Mobile Devices and Services*. https://doi.org/10.1145/3379503.3403564

A UIAD MODEL API

Table 5 shows the UIAD Models APIs and their usages.

B PRE-DEFINED HEURISTIC RULES

B.1 Feature Set

B.1.1 Page Classification. The text location rule set describes several texts and their corresponding screen coordinate position. Each class of pages has a unique rule set. For example, Figure 20 shows the text location rule set corresponding to the WeChat contact list page.

B.1.2 Widget Recognition. We present a total of 26 features for widget recognition, which can be divided into three categories:

(1) Style and position on images:
Coordinates relative to the entire screen's left edge, right edge, top edge, and bottom edge; coordinates relative to the list item's left edge, right edge, top edge, and bottom edge; X-axis and Y-axis centerlines

Table 5. Model APIs and their usages.

Type	API Name	Example	Condition	Target	Result
To Get Information	get_property	Get all information about Lee.	{type:Contact, Username:“Lee”}	Contact	Phone:“xxxxxxxx”, Avatar: Base64 encoded, Recent_message:“Hi!” .....
	get_related_info	Get the chat history with Lee.	{type:Contact, Username:“Lee”, Relationship:source}	Message List	[Message0, Message1, ...]
	get_list	Get the first five contacts.	{type:Contact List}	{from: 0, num: 5}	[Contact0, Contact1, ..., Contact4]
	index_of	Get the index of Lee in the Contact List.	{type:Contact, Username:“Lee”}	index	0
To Use Methods	use_method	Make a video call with Lee.	{type: Contact, Username:“Lee”}	VideoCall	(The smartphone automates the task.)
		Say hello to Lee.	{type: Contact, Username:“Lee”}, {type: text, value:“Hello!”}	SendMessage	(The smartphone automates the task.)

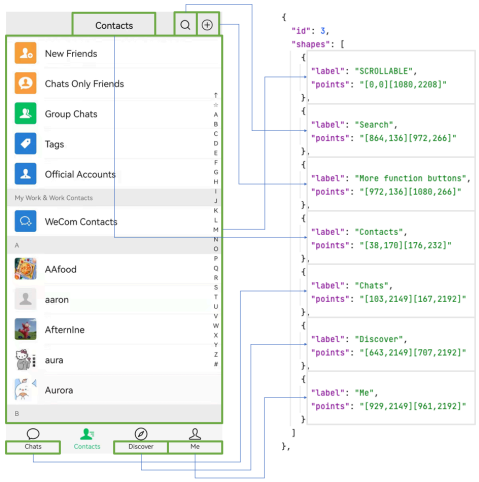


Fig. 20. Text Location rule set corresponding to the WeChat contact list page.

relative to the entire screen; X-axis and Y-axis centerlines relative to the list item; average RGB within the region; maximum color block RGB within the region.

(2) Style and position on layouts:
Depth, node class, resource ID, clickable, editable, scrollable, path in the layout tree, path in the layout tree (ignoring subtree numbering), path in the layout tree (ignoring list item numbering), text, content description

(3) Page segmentation:
The segmentation scheme from the start of the entire page to the widget, consists of a sequence composed only of Horizon (H) and Vertical (V).

B.2 Similarity Function

Using our own dataset, we determined whether two items belonged to the same class through a similarity function (Algorithm 1). We established thresholds by leveraging the Receiver Operating Characteristic (ROC) curve.

Algorithm 1: Similarity Function

Input: two pages or widgets, A and B
Output: a real number indicating the similarity between the two pages or widgets

$sim \leftarrow 0$;
foreach *feature* $\in$ *featureSet* **do**
    $sim \leftarrow sim + \text{match}(feature, A, B) \times \text{weight}(feature)$; `// Calculate the matching ratio of each feature of` A `and` B
end
return *sim*

C RUN-TIME SEMANTIC ANALYSIS ALGORITHM

Algorithm 2 is utilized for run-time semantic analysis.

Algorithm 2: Run-time Semantic Analysis Algorithm

Input: the current layout tree's root *crtRoot* and screenshot *img*.
Output: a UIAD Model instance.

pageId ← `PageClassification.getPageId(`*crtRoot, img*`)`;
rootStrategy ← `WidgetRecognition.getStrategy(`*pageId*`)`;
modelInstance ← `match(`*rootStrategy, crtRoot*`)`; `// Matching is done in a top-down order.`
return *modelInstance*;
Function `match(`*crtStrategy, crtRoot*`)`**:**
    matchedNodes ← *crtStrategy*.`findNodes(`*crtRoot*`)`;
    crtModel ← `package(`*matchedNodes*`)`; `// Extract the corresponding data from` *matcheNodes*.
    foreach *childStrategy* $\in$ *crtStrategy.children* **do**
        foreach *node* $\in$ *matchedNodes* **do**
            crtModel.children ← `match(`*childStrategy, node*`)`;
        end
    end
    return *crtModel*;
return

D THE IMPLEMENTATION OF THE IAM

While both pages and widgets follow the workflow of IAM, their implementations differ significantly, as detailed in Table 6.

Table 6. Differences between the page classification and the widget recognition.

	Page Classification	**Widget Recognition**
Purpose	Each page corresponds to a unique class.	A widget can be mapped to multiple semantic elements in the model.
Model Implementation	A multi-class decision tree.	A bi-class decision tree for each semantic element.
Annotation for Each Iteration	Modify or confirm the classes of pages.	Remove False-Positive widgets and add False-Negative ones.
Feature Set	Text location rules detailed in B.1.1.	Three categories of widget features detailed in B.1.2, forming 26-dimensional vectors.
Expansion of the Train Set	Classification with conservative heuristic rules, such as identifying the resource-id and the title.	Heuristically expand the set of negative examples, which are near positive examples.

E OFFLINE DATASET VALIDATION

Aiming to validate the IAM, we constructed an offline dataset by developing UIAD Models for 12 applications across 9 categories, as shown in Table 7.

Table 7. An overview of applications with corresponding page and widget count evaluated.

Application	Category	Pages	Page Classes	Widgets (remove meaningless ones)	Widget Classes (page classes× semantic elements)	Model Size (Registration File) (in kBs)
WeChat	Message	123	17	2364	238	931
Taobao	Shopping	232	22	4921	528	1276
JingDong	Shopping	560	51	11264	1632	1651
Amap	Map	1238	33	19391	363	786
NetEase Music	Music	1598	45	28943	945	716
QQ Music	Music	508	42	10216	798	614
MeiTuan	Dilivery	462	29	12110	899	988
Eleme	Dilivery	376	26	9626	780	854
Himalaya	Audio	234	12	4199	228	561
Alipay	Pay	554	26	6094	416	603
TikTok	Social	152	17	1778	527	920
iQIYI	Video	376	19	5610	627	1066
All	-	6413	339	116516	7981	10966

F WORKFLOW PERFORMANCE EVALUATION

F.1 Use Cases for Workflow Performance Evaluation

We listed the following use cases based on the requirements from the AIoT product management to evaluate the performance of our workflow:

F.1.1 Use Case 1: Home Audio System. In a home environment, this use case employs voice control for music apps, combining rich information displayed on a large screen to address the limitations of Voice User Interface (VUI) output capabilities. The application synchronizes smartphone screen information with smart speakers and large-screen TVs. The functional requirements are as follows:

(1) Access smartphone music app album lists, playback lists, and song details on a large screen.
(2) Trigger music app functions on a large screen, such as play, pause, select playlists, discover, podcasts, and subscriptions.
(3) Support using voice to access the above information or trigger the above functions.

F.1.2 Use Case 2: Map Applications for Vehicle Head-up Displays. In a driving scenario, this use case extracts key road information from smartphone map apps like Amap and projects it to a HUD in vehicles, realizing information dimensionality reduction from smartphone to HUD. Functional requirements include smartphone Amap key road information extraction, such as text descriptions, road sign guidance, real-time maps, speed, and other vehicle status data.

F.1.3 Use Case 3: Food Ordering in Mobile Scenarios. In mobile scenarios (such as running, walking, or cycling), this use case involves ordering food using smartwatches. The functional requirements are as follows:

(1) Support Voice activation of the food order system.
(2) Access Store recommendation list.
(3) Access Food recommendation list.
(4) Support search, selection, browsing, and other functions.
(5) Support Payment and confirmation options.

F.1.4 Use Case 4: Schedule and Express Delivery App Integration. This use case explores multi-app integration by combining a Schedule and an Express Delivery App, integrating the information in a new interface to conveniently remind users to pick up their deliveries in a timely manner. The functional requirements are as follows:

- Display existing ToDo list.
- Automatically add express delivery information to the ToDo list.
- Support common operations for each ToDo item, such as marking it as complete, modifying, searching, etc.

F.2 Target-Device UI Examples

The target-device UI examples corresponding to each use case are illustrated in Figures 21, 22, 23, 24.

F.3 Statistical Evaluation

F.3.1 Developing UIAD Models. We provide an in-depth statistical analysis of our findings, as previously mentioned in Section 7.3.2. To elucidate the factors influencing the time required for participants to develop UIAD models, we employed a Generalized Estimating Equations (GEE) model. The dependent variable in our analysis was the time taken by participants, while the independent variables encompassed the participant group (categorized as either programmer or non-programmer), the count of task attempts each participant undertook, and the specific use case guiding the task's execution.

We transformed the categorical "Use case" variable into binary dummy variables: UseCase_1, UseCase_2, UseCase_3, and UseCase_4. To prevent multicollinearity, we dropped UseCase_4, retaining only UseCase_1, UseCase_2 and UseCase_3.

As shown in Table 8, a participant's programming background, the count of task attempts, and specific use case significantly affect the time to develop UIAD models. Non-programmers tend to take longer, with an increase in

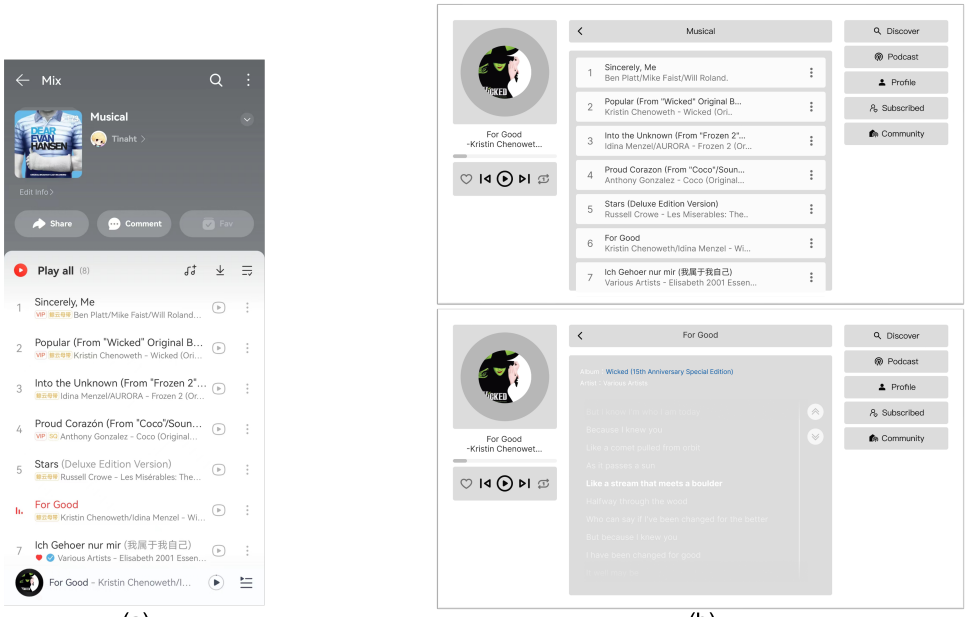


Fig. 21. A Target-device UI example for Use Case 1. (a) The original mobile phone interface of NetEase Music. (b) New deployed UI.

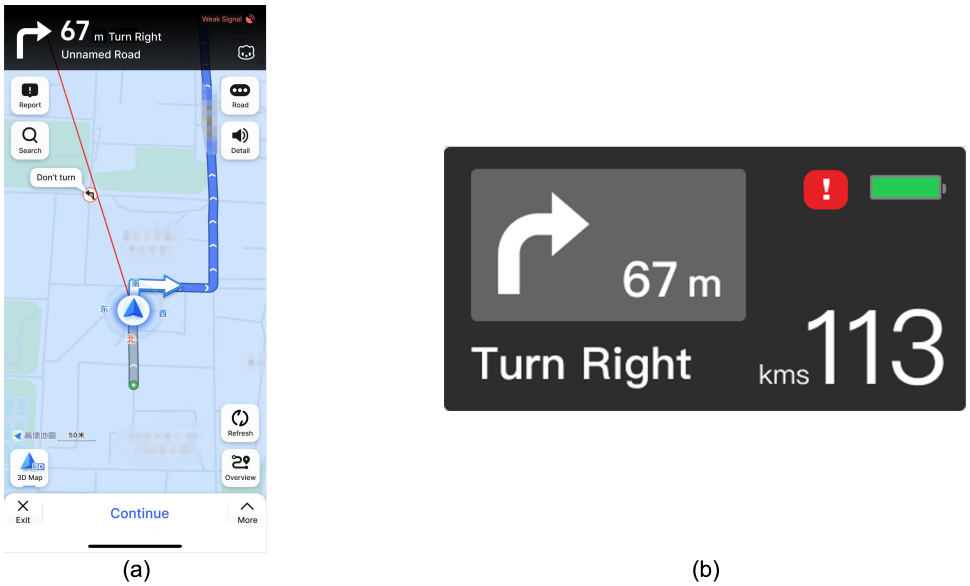


Fig. 22. A Target-device UI example for Use Case 2. (a) The original mobile phone interface of Amap. (b) New deployed UI.

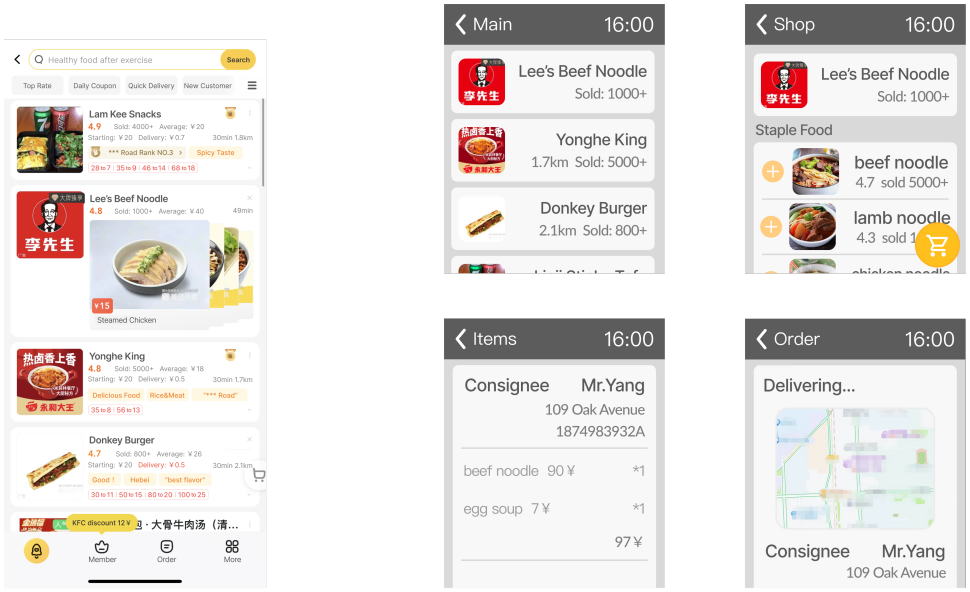


Fig. 23. A Target-device UI example for Use Case 3. (a) The original mobile phone interface of Meituan. (b) New deployed UI. The images have been post-processed to change the text into English.

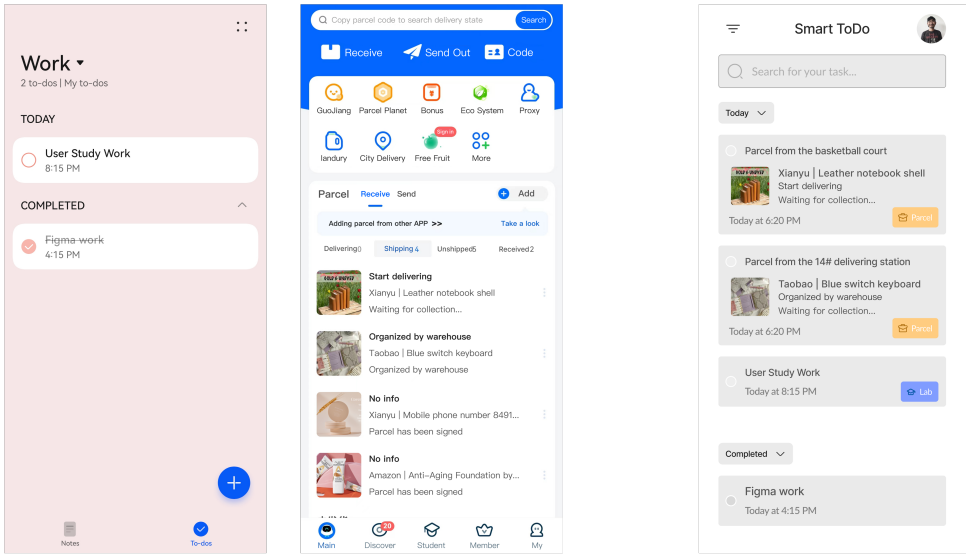


Fig. 24. A Target-device UI example for Use Case 4. (a) The original mobile phone interface of Schedule Application. (b) The original mobile phone interface of Express Delivery Application. (c) New deployed UI. The images have been post-processed to change the text into English.

Table 8. GEE model results evaluating the impact of various independent variables on the completion time of participants for developing UIAD models. Coefficient estimates, standard errors, z-values, p-values, and 95% confidence intervals are listed.

Independent variables	coef	std err	z	P>|z|	[0.025	0.975]
Intercept	15.588	0.896	17.405	< 0.001	13.832	17.343
Group(Programmer=0, Non-programmer=1)	1.418	0.595	2.382	0.017	0.251	2.585
The count of task attempts	-0.607	0.219	-2.771	0.006	-1.037	-0.178
UseCase_1	8.417	0.623	13.519	< 0.001	7.197	9.637
UseCase_2	-6.314	0.744	-8.487	< 0.001	-7.773	-4.856
UseCase_3	3.142	0.648	4.849	< 0.001	1.872	4.412

Table 9. OLS regression results evaluating the difference in slopes of the regression lines between programmers and non-programmers. Coefficient estimates, standard errors, z-values, p-values, and 95% confidence intervals are listed.

Independent variables	coef	std err	z	P>|z|	[0.025	0.975]
Intercept	-0.752	3.609	-0.208	0.836	-8.145	6.641
x	0.872	0.174	5.026	< 0.001	0.517	1.227
Group(Programmer=0, Non-programmer=1)	-5.884	4.344	-1.354	0.186	-3.016	14.783
x × Group	0.523	0.190	2.761	0.010	-0.911	-0.135

time by 1.418 units. As participants attempt more tasks, they become more efficient, decreasing the completion time. Additionally, the results suggest that the characteristics of different use cases profoundly affect the time to develop UIAD models. In particular, tasks under UseCase_2 appear to be easier or more straightforward, leading to quicker completion times.

To test whether there is a significant difference in the slopes of the regression lines of the two groups in Figure 16c, we constructed a model that includes an interaction term: $y = a \times x + b \times Group + c \times x \times Group + d$. The Ordinary Least Squares (OLS) results are shown in Table 9. The coefficient of $x \times Group$ is 0.523 with a p-value of 0.010, proving that the difference in the slopes of the two lines is significant.

F.3.2 Developing Target-Device UIs. Similarly, we also employed a GEE model to elucidate the factors influencing the time required for participants to develop target-device UIs (previously mentioned in Section 7.3.3). The dependent variable in our analysis was the total and binding time taken by participants, while the independent variables encompassed the count of task attempts each participant undertook, and the specific use case guiding the task's execution.

Consistent with the prior approach, we transformed the categorical "Use Case" variable into binary dummy variables: UseCase_1, UseCase_2, UseCase_3 (UseCase_4 was removed).

According to the GEE model results in Table 10 and 11, UseCase_3 has a significant impact on both the total (p-value=0.006) and binding time (p-value=0.039), indicating that this particular use case might involve higher complexity or demands. Additionally, the count of tasks each participant attempts does not have a substantial effect on the completion time in this context (p-value=0.204 for the total time and 0.749 for the binding time).

F.3.3 Experiencing Target-Device UIs. We utilized an OLS model to analyze whether a specific use case would influence the response time for each API call, as previously discussed in Section 7.3.4. Consistent with the prior approach, we transformed the categorical "Use Case" variable into binary dummy variables: UseCase_1, UseCase_2, UseCase_3 (UseCase_4 was removed).

Table 10. GEE model results evaluating the impact of various independent variables on the **total** time of participants for developing target-device UIs. Coefficient estimates, standard errors, z-values, p-values, and 95% confidence intervals are listed.

Independent variables	coef	std err	z	P>|z|	[0.025	0.975]
Intercept	128.438	49.571	2.591	0.010	31.280	225.595
The count of task attempts	18.125	14.280	1.269	0.204	-9.863	46.113
UseCase_1	58.750	38.998	1.506	0.132	-17.685	135.185
UseCase_2	3.750	23.006	0.163	0.871	-41.342	48.842
UseCase_3	193.750	69.833	2.774	0.006	56.879	330.621

Table 11. GEE model results evaluating the impact of various independent variables on the **binding** time of participants for developing target-device UIs. Coefficient estimates, standard errors, z-values, p-values, and 95% confidence intervals are listed.

Independent variables	coef	std err	z	P>|z|	[0.025	0.975]
Intercept	3.9688	1.015	3.908	< 0.000	1.979	5.959
The count of task attempts	0.1125	0.351	0.320	0.749	-0.576	0.801
UseCase_1	1.0000	1.170	0.855	0.393	-1.293	3.293
UseCase_2	-0.2500	0.985	-0.254	0.800	-2.181	1.681
UseCase_3	1.8750	0.908	2.065	0.039	0.096	3.654

Table 12. OLS regression results evaluating the impact of use cases on the response time for each API call. Coefficient estimates, standard errors, z-values, p-values, and 95% confidence intervals are listed.

Independent variables	coef	std err	z	P>|z|	[0.025	0.975]
Intercept	572.667	19.306	29.663	0.000	534.805	610.529
UseCase_1	-219.583	22.499	-9.760	< 0.001	-263.706	-175.459
UseCase_2	-304.696	24.857	-12.258	< 0.001	-353.444	-255.947
UseCase_3	-3.155	24.958	-0.126	0.899	-52.102	45.791

According to the OLS model results in Table 12, UseCase_1 and UseCase_2 significantly decrease the response time for each API call. On the other hand, UseCase_3 does not significantly affect the response time, suggesting that its performance is similar to the omitted category (UseCase_4 in this case). This highlights that the specific use case does indeed have an influence on the API call's response time.

Additionally, to evaluate whether there were significant differences among the five segments of response time illustrated in Figure 19, we conducted a Friedman test. The results with a statistic of 3575.456 and a p-value less than 0.001 suggest that there is a statistically significant difference among these five segments.

F.4 Subject Feedback

The subjective feedback details are shown in Table 13 and Table 14.

Table 13. Full results of the subject feedback from UIAD Model developers on a Likert Scale from 1 (strongly disagree) to 7 (strongly agree). Statement numbers refer to Table 3.

Participant	Statements							
	S1	**S2**	**S3**	**S4**	**S5**	**S6**	**S7**	**S8**
P4	6	5	7	7	6	7	5	6
P5	7	6	7	6	7	7	6	7
P6	7	6	7	5	6	7	6	7
P7	6	5	7	7	6	7	5	5
P8	5	4	3	4	6	5	4	6
P9	4	3	2	6	7	5	4	5
P10	6	5	4	7	7	6	5	5
P11	3	3	6	5	7	4	5	6

Table 14. Full results of the subject feedback from end-users on a Likert Scale from 1 (strongly disagree) to 7 (strongly agree). Statement numbers refer to Table 4.

Participant	Statements					
	S1	**S2**	**S3**	**S4**	**S5**	**S6**
P20	6	7	6	7	6	6
P21	7	6	7	7	7	7
P22	5	4	5	6	5	5
P23	7	7	6	7	6	7
P24	5	6	5	6	5	5
P25	6	7	6	7	6	6
P26	7	7	6	7	7	7
P27	5	7	6	7	6	6