Informatics Bulletin, Faculty of Computers and Artificial Intelligence, Helwan University
Published Online ***Vol 6 Issue 2, July 2024***
(https://fcihib.journals.ekb.eg)

A Survey of techniques for Automatic Code Generation from User Interface Designs with Various Fidelities

Michel Samir Zaki Gad[1], Mohamed Marie[2], Ahmed El Sayed Yakoub[2]

[1]Software Engineering, Faculty of Computers and Artificial Intelligence, Helwan University, Cairo, Egypt.
[2]Information Systems department. Faculty of Computers and Artificial Intelligence, Helwan University, Cairo, Egypt.
michelsamir1996@gmail.com, mohamedmarie@yahoo.com, eng_ahmedyakoup@yahoo.com

***Abstract*—Graphical User Interface (GUI) visual nature makes it the most commonly used type of User Interface (UI), as it enables direct manipulation and interaction with software. Mockup-based design is a popular method for creating GUIs. This approach involves several steps, ultimately resulting in the development of a more detailed mockup and its subsequent implementation into code. Due to changing requirements, the design process often requires repeating these steps, which can be tedious and necessary modifications to the GUI code. Furthermore, converting a design into GUI code is a time-consuming task that can prevent developers from focusing on implementing the software's functionality and logic, making it costly. To address these challenges and allocate more time in building the application's functionality, automating the code generation process using sketches or GUI design images can be a viable solution. This comprehensive literature review presents an overview of various existing techniques and approaches that facilitate the automatic generation of source code from hand-drawn, high or low fidelity mockups and wireframes, utilizing diverse methods such as deep learning or computer vision.**

***Index Terms*— Code Generation, Graphical user interfaces, Deep Learning, Computer Vision, Mockups**

I. Introduction

In interactive software, user interfaces (UIs) are the means by which users interact with and operate the system's capabilities. Graphical User Interfaces (GUIs) are the most commonly used type of UI due to their visual nature, allowing direct manipulation of the software. However, creating GUIs for applications is often a manual and time-consuming process. A survey of over 5,000 developers revealed that 51% reported working on app UI design tasks on a daily basis, which is more frequent compared to other development tasks [1]. Another study found that, on average, 48% of software code size relates to the user interface, and 50% of implementation time is dedicated to the user interface portion [2].

Mockup-based design is a prevalent workflow for building user interfaces [3]. In this approach, a graphic designer creates a basic illustration of the intended UI design, typically starting with a digital or sketched wireframe [4]. The wireframe outlines the fundamental structure of the application but lacks specific details like colors. As the design progresses, the wireframe is refined and more details are added, turning it into a higher-fidelity mockup [5]. Once the design is finalized, the implementation process begins. The prototype is evaluated for usability and any design issues, and this iterative process continues until the prototype is deemed satisfactory. However, due to changing requirements, this design process can become repetitive, requiring modifications to the GUI code.

Developers are responsible for implementing client-side software based on GUI mockups. However, converting a design into GUI code is time-consuming and prevents developers from focusing on implementing the actual functionality and logic of the software, making it costly. Generating GUI code from mockups also requires extensive experience due to the complexity involved in extracting visible elements, defining their relationships, selecting appropriate UI components, and generating source code. Another challenge arises when generating front-end code from GUI images because the computer languages used to implement such GUIs are specific to each target runtime system. This can result in tedious and repetitive work when the software needs to run on multiple platforms using native technologies [19].

To address these challenges and allocate more time to developing the core functionality of an application, automating the generation of front-end code becomes necessary. Developers need a way to visually understand UI elements and their spatial arrangement within an image and translate this understanding into appropriate GUI components and compositions. Automating this process of visual understanding and translation would greatly facilitate the initial stages of GUI implementation. However, this task is challenging due to the wide range of UI designs and the complexity of generating GUI code.

The task of comprehending digital mockups presented as images by a machine falls under the domain of Computer Vision. It involves the machine making deductions, understanding the mockups, and extracting meaningful information from them. Computer Vision has made significant advancements, with deep learning methods, particularly Convolutional Neural Networks (CNNs), showing promise in

various vision-related problems [6, 7].

Detecting objects in UI screenshots poses a unique visual recognition challenge that requires a specialized solution. This systematic literature review compares current studies to evaluate the effectiveness of neural networks, computer vision, and other approaches for identifying objects in UI mockups or sketches, regardless of their fidelity or whether they are hand-drawn or digitally created. The review also assesses their ability to automate the generation of front-end source code.

This paper is organized as follows. Section 2 provides background information, followed by the literature review in Section 3. Section 4 discusses the methodologies employed, and Section 5 presents the results and limitations. Finally, Section 6 presents the conclusions and recommendations.

II. Background

There is some confusion surrounding the definitions of wireframes and mockups and how they differ from each other. It is important to provide a precise clarification and clearly distinguish these concepts. Typically, the design process follows a sequential progression consisting of three stages: wireframes, mockups, and prototypes. However, it's worth noting that variations may exist depending on the designer, team, and project, and not all stages may be included. For the purpose of this discussion, we will focus on wireframes and mockups.

A. Wireframes

A wireframe, also known as a screen blueprint, is a document that outlines the basic structure and layout of a page or screen in applications. It visually represents the interface elements that will be present on key pages. Wireframes are considered low-fidelity design documents characterized by their simplicity and lack of visual styles and branding elements. They do not include specific details such as colors, images, or finalized content. Instead, their purpose is to provide a basic visual understanding of a page in the early stages of a project, facilitating stakeholder and team approval before moving to the creative phase.

Wireframes can be classified into two types: digital and hand-drawn wireframes. Hand-drawn wireframes, also referred to as sketches, are particularly useful during the initial design stages and for quick iterations. They allow designers to visualize rough ideas and create initial models for the overall layout in a simplified format. On the other hand, digital wireframes are more detailed while still maintaining simplicity. These wireframes are typically created using digital wireframing tools available online. Although they may not include specific components like images or complete text, they provide more detailed representations compared to hand-drawn wireframes.

Despite the availability of digital wireframing tools, many designers still prefer to start the wireframing process by sketching on paper using a pen (hand-drawn wireframe). This preference can be attributed to the fact that designers often have an artistic background and may feel more comfortable and unrestricted when using traditional tools. While there is no universally established standard, wireframe sketches typically utilize a set of symbols that have widely recognized meanings. Fig. 1 visually illustrates some of these elements.

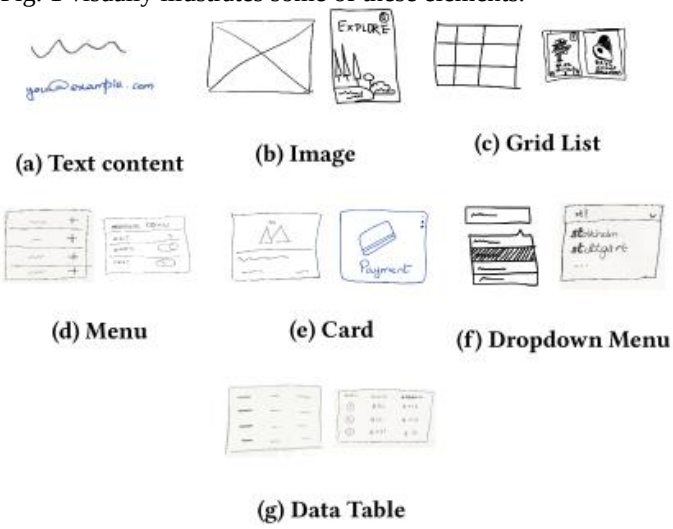


Fig. 1. Examples of elements commonly used to represent UI elements in wireframes

B. Mockups

A mockup is a design document that demonstrates a high level of fidelity, closely resembling the desired final product. It can be compared to a graphical user interface (GUI) screenshot of an application. Mockups showcase the visual appearance of the design and are typically created after the wireframing stage but before prototyping. Wireframes, on the other hand, focus on presenting the structure of the design, and UI elements are later incorporated into the mockups. Essentially, mockups enhance wireframes by adding visual design elements such as images, colors, and typography. Fig. 2 provides a visual comparison to illustrate the differences between wireframes and mockups.

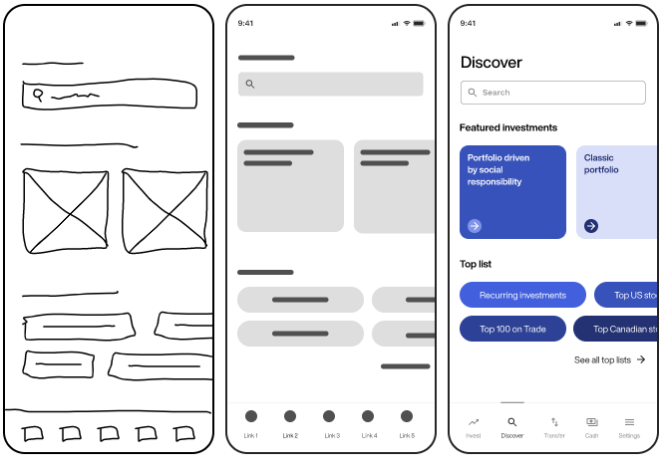


Fig. 2. The difference between Hand-drawn wireframe and Digital wireframe and Mockup.

Furthermore, these concepts can also be categorized based

on three different levels: (i) low-fidelity, which resembles hand-drawn wireframes and emphasizes outlining the basic page structure; (ii) mid-fidelity, which resembles digital wireframes and represents the initial stages of interface creation; and (iii) high-fidelity, which refers to mockups with high-quality visuals and content.

Once the final design document is completed, designers pass their work to front-end developers for implementation in code. The process of implementing user interfaces involves translating the graphical designs created by the designers into functional software code. While developers typically prioritize implementing core functionalities, they often find themselves spending a significant amount of time coding user interfaces.

III. LITERATURE REVIEW

Recently, there has been an increasing emphasis on utilizing deep learning and computer vision techniques to automatically generate UI code, which is a relatively new area of research. The purpose of this literature review is to examine and evaluate the current methods and approaches employed by deep learning and computer vision in classifying UI components within wireframes or mockups presented as images. The related works in this section are categorized into those that deal with wireframes and sketches, and those that focus on mockups and screenshots. Despite the difference in focus, both categories share a common goal of automatically translating a design into application code.

A. Wireframe-Based Techniques

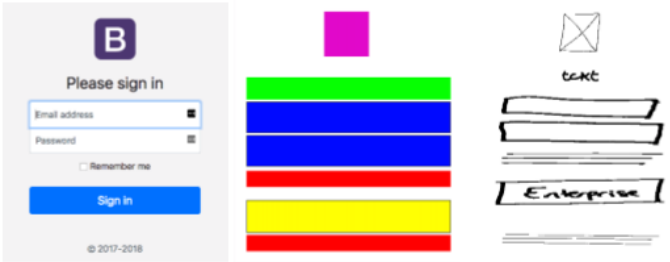
In [9], the authors employed a simple object detection technique that takes a low-fidelity hand-drawn image of a web page as input. The input image undergoes several stages of processing, including morphological transformations and contour detection, to enable object cropping and facilitate the detection of individual elements in the sketched image. By leveraging deep learning, the algorithm can determine whether the detected element is a button, text, image, or another component. The identified elements are then used to construct an HTML page.

In the initial stage of this study, object detection is performed. Gaussian Blur is applied, followed by rounds of erosion and dilation, and contour detection to identify separate rectangles corresponding to each element. At this point, the system can only detect the presence of an element without determining its classification or identity. The identified rectangles are subsequently cropped and used as input for the object recognition system. A Convolutional Neural Network (CNN) is employed in this system to classify the cropped objects. The identified objects are then coded into an HTML page using a bootstrap framework.

In [10], the authors introduce a novel application called Sketch2Code, which focuses on translating wireframes drawn on paper into application code. As shown in Fig. 3, the dataset includes both wireframes and their corresponding normalized images, which represent the structure of the websites in a standardized format. Sketch2Code utilizes an Artificial Neural Network (ANN) to learn the relationship between a wireframe image and its normalized image. This enables the detection and classification of different UI elements and containers within the wireframe. The approach employs a deep semantic segmentation network technique, which falls under the category of image segmentation.

The pre-processing for Sketch2Code involves noise removal from the input image and contour detection to outline the wireframe. After segmentation, the elements are detected, and a Domain Specific Language (DSL) code is generated. This DSL code represents the structure of the wireframe in a JSON tree-like structure, which can be directly translated into HTML.

The Microsoft AI Lab has developed Sketch2Code [11], an application that utilizes artificial intelligence (AI) to transform hand-drawn web page sketches, represented as images, into valid HTML code. While there is no detailed literature available on their work, the authors have made their code and dataset openly accessible to the public. At the core of this system is an image recognition model trained on datasets of hand-drawn images. The model can identify essential HTML elements like buttons, labels, and text boxes. Additionally, it can recognize handwritten text and incorporate it into the generated HTML code for the website.



(a) Original website (b) Normalised version (c) Sketched version

Fig. 3. The original website (a) is normalized in (b). Wireframe sketch version in (c)

In the work by Suleri et al. [12], they developed a software prototyping workbench that employs object detectors to detect elements within a sketch. The object detection model used is a RetinaNet based Single-Shot MultiBox Detection (SSD) network with a Resnet backbone. To train the model, the authors created a synthetic dataset using real hand-drawn wireframes. They gathered 5906 sketches of 19 Google's Material Design based UI elements by engaging 350 participants and asking them to draw and annotate the wireframes. To generate the synthetic dataset, the authors randomly filled an image by sampling the UI elements and placing them in random positions. This process resulted in an annotated dataset of 125,000 images.

The authors categorized their system interactions into three fidelity representations, each offering different levels of interaction. At the low fidelity level, wireframes are

represented in their original form, allowing users to create or modify existing wireframes and their interactions. The medium fidelity level is represented by the output of the object detector and a generated medium fidelity design. In this phase, users have control over the conversion process and can modify the properties of the detected elements. They also have the option to manually detect elements in case of missed detections. At the high fidelity level, the final user interface (UI) is generated, and users can define different themes and generate the final code.

Jain et al. [13] introduced the Sketch2Code software, focusing on the real-time transformation of hand-drawn sketches into a coded UI application. They utilized a deep neural network trained on a customized database of sketches to identify UI elements. The dataset used in this study is relatively small, consisting of only 149 sketches, which include a total of 2,001 samples of GUI elements. Furthermore, the GUI element sketches were created by individuals.

To address the first sub-problem of recognizing UI components in the image, the authors employed the RetinaNet model to generate boundary box coordinates and component classes in a CSV format without performing additional pre-processing on the input image. The RetinaNet model used had 50-Resnet layers pre-trained on the ImageNet dataset and incorporated the Feature Pyramid Network (FPN) to create multi-scale feature maps. To tackle the second sub-problem of overlapping bounding boxes, the authors developed an overlapping classes filtering algorithm. This algorithm assigned individual priorities to the prediction classes and utilized a priority-score hierarchy to choose between overlapping bounding boxes when the overlap exceeded 50%.

Finally, to address the third sub-problem of converting predicted classes and bounding boxes into functional UI code, the authors developed a UI parser. This parser translated the UI representation object produced by the model into a coded application. The UI representation object is platform-independent, allowing the generated UI code to be used across multiple platforms.

In contrast, Kim et al. [14] employed a different approach to detect the underlying layout of a wireframe. Their methodology involved two main steps: layout detection and UI element recognition. For UI element recognition, they utilized the Faster R-CNN object detector. For layout detection, computer vision techniques were employed to connect disconnected edges. Since the layouts were constrained to the x and y-axis, a slope filtering technique was utilized to remove slopes that were not horizontally or vertically aligned. Finally, a correspondence line algorithm was applied to determine the layout, and this algorithm was executed for each detected layout until no more layouts were found.

Yun et al. [15] employed YOLO, an object detection network, in their study. The network was trained using transfer learning from an existing network to learn the features of UI elements. When provided with a hand-drawn mockup, the mockup is passed through the YOLO network, which detects the UI elements and provides their corresponding confidence levels. The output is then transformed into a hierarchical structure, which is used to generate code specifically for the desired platform.

In the paper referenced as [16], the authors describe the creation and implementation of deep learning models for translating GUI sketches, created using the Balsamiq Mockups application, into HTML, CSS, and Bootstrap code. A comprehensive dataset was developed, consisting of graphical user interface (GUI) sketches of web pages and corresponding captions. The dataset includes 1,100 images, with 1,000 images for training and 100 for testing. It encompasses the most commonly used Bootstrap components.

The study involved constructing two deep learning models, each utilizing a different approach to integrate into the web application. The first approach combines a convolutional neural network (CNN) with two recurrent networks (RNNs) in a hybrid architecture, following the encoder-decoder architecture commonly used in papers like pix2code [19]. However, this approach is applied to sketches rather than GUI screenshots. The second approach, which is a significant contribution of the paper, involves utilizing YOLO to detect and localize HTML elements. Additionally, a layout algorithm was developed to convert the YOLO output into code. The layout algorithm maps each object recognized by YOLO into HTML and CSS code based on the bounding box coordinates.

The paper [17] presents a four-step pipeline for generating a website in real-time from a hand-drawn sketch image. The pipeline includes image acquisition, object and container detection, building the object hierarchy, and HTML code generation. During image acquisition, the sketch image is converted to black and white, and an adaptive threshold is applied to define the lines. Morphological operations, specifically closing and erosion, are then used to remove any noise. The object and container detection step identifies the various elements depicted in the sketch using two modules: one for detecting individual atomic elements and another for identifying containers. Containers are defined as boxes that encompass multiple atomic elements, such as buttons or text elements.

For element detection, the YOLO object detector is employed to identify elements along with their boundaries, but without a hierarchical structure. The object hierarchy is built using a sequence of modules that extract hierarchical information. This involves merging elements and containers to establish the hierarchy. The final stage is HTML code generation, where the completed hierarchy is inputted into the HTML/CSS generators to generate the corresponding code. To train the YOLO models, a diverse and large collection of examples is required. The paper also introduces a mechanism for generating a large dataset of digital hand-drawn-like sketches, commonly known as synthetic sketches.

In their study [18], the authors employed YOLOv5, a precise and efficient deep learning framework, to automate the conversion of hand-drawn GUI mockups into Android-based GUI prototypes. The approach proposed in this paper consists of three main phases: detection and classification, alignment of GUI components, and construction of the GUI layout.

In the detection and classification phase, a pre-trained YOLO model is utilized to accelerate the training process and improve performance. The custom dataset is then used to fine-tune the pre-trained model, enabling the detection and classification of hand-drawn components in the input image. During the alignment of GUI components phase, the detected GUI components are aligned using heuristic methods to ensure accurate positioning and sizing. The output is a JSON file that provides a comprehensive description for each GUI component in the processed image.

Lastly, in the construction of the GUI layout phase, the JSON file is converted into an output GUI prototype that aligns with the target platform, as shown in Fig. 4. The custom dataset consists of images, each containing the same number of GUI components, covering all the listed components. The dataset comprises 390 training images, with 90% used for training and 10% for validation, along with 100 images designed for testing.

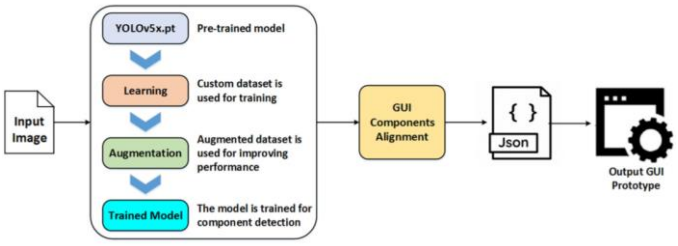


Fig. 4. Architecture of the proposed methodology [18]

B. Mockup-Based Techniques

The authors of a particular paper [19] introduce Pix2code, an application that converts high-fidelity GUI screenshots into computer code. They utilize a Deep Learning framework to perform this conversion for web-based, Android, and iOS platforms. To create the Pix2code dataset, they map bootstrap-based websites into a Domain-specific language (DSL) consisting of 18 vocabulary tokens that describe the website's layout and components. The dataset includes 3,500 pairs of GUI images and their corresponding DSL code markup.

The core idea behind Pix2code is training a model to learn the mapping between a GUI screenshot and the code that produces the corresponding GUI. The model comprises two main components. First, a Convolutional Neural Network (CNN) extracts high-level visual features from the GUI image, which are then transformed into a fixed-length feature vector using a fully connected layer. Second, a Recurrent Neural Network (RNN) with Long Short-Term Memory (LSTM) architecture performs language modeling on the DSL code related to the input GUI image. Through training, the LSTM network grasps the syntax and semantics of the source code, generating a language-encoded vector that represents a sequence of one-hot encoded tokens corresponding to the DSL code.

To solve the problem, the authors propose a three-step approach. Firstly, a CNN-based image encoder extracts high-level visual features from the GUI screenshot and converts them into a fixed-length feature vector. Secondly, an LSTM network, an RNN architecture, is trained to perform language modeling on the DSL code associated with the GUI image. This results in the LSTM network understanding the syntax and semantics of the code, generating a language-encoded vector representing a sequence of one-hot encoded tokens. Lastly, an LSTM-based code decoder is used. The vectors from the previous steps are concatenated and fed into this decoder, which generates accurate code that reflects the layout and components of the input GUI image. The LSTM decoder learns the relationship between objects in the GUI image and the corresponding tokens in the DSL code.

Another methodology described in reference [20] shares similarities with the previous method discussed in reference [19]. In this approach, a CNN processes the UI image representation, which is then encoded by an LSTM into an intermediate representation vector. This vector is further decoded by a final LSTM to generate the final intermediate code. This methodology features a more straightforward training strategy as it does not require contextual information as input to the network, making it more accessible, as illustrated in Fig. 5.

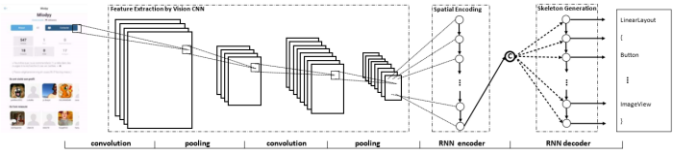


Fig. 5. Architecture of Neural Machine Translator for UI-Image-to-GUI-Skeleton Generation

Nguyen et al. [21] introduced the pioneering concept of automatic reverse engineering of mobile application user interfaces (REMAUI). By analyzing screenshots of a mobile application's user interface, REMAUI identifies various components such as buttons, textboxes, and images, and generates the corresponding code. This study marked the first instance of employing computer vision and optical character recognition techniques, along with mobile-specific heuristics, to facilitate the conversion of screen images into code for mobile platforms. These applications not only capture the structural elements but also consider the style aspects, including images, colors, and fonts, of the designs. However, despite the successful functionality of the REMAUI method, it also exhibits certain limitations.

Moran et al. [22] introduced ReDraw as an extension of REMAUI. ReDraw is an algorithm that takes mockups of

mobile application screens and generates structured XML code for them. The paper presents a three-stage approach to automate the conversion of GUI designs into code, comprising the Detection, Classification, and Assembly steps. The initial stage of their approach involves employing computer vision techniques to identify the individual components of the GUI. In the second stage, these identified components are classified according to their functionality, such as toggle-button, text-area, and more. Deep convolutional neural networks (CNN) are used for this classification task. In the final stage, the XML code is generated by combining the outcomes of the previous stages with the K-nearest neighbor (KNN) algorithm, which organizes the code based on the hierarchy of web programming.

It is noteworthy that the authors of this paper have also contributed to the development of a dataset. This dataset encompasses 14,382 GUI images, containing 191,300 annotated GUI segments. It includes 15 classifications, such as RadioButton, ProgressBar, Switch, Button, and Checkbox. The CNN model mentioned earlier relies on this dataset for training and evaluation purposes.

Chen et al. [23] proposed a framework that takes UI pages as input and generates the corresponding GUI code for Android or iOS as output. The authors initially employ traditional image processing techniques, including edge detection, to locate the UI elements within the pages. Subsequently, they utilize CNN-based classification to determine the semantics of the UI elements, such as their types.

The proposed framework comprises three distinct phases: component identification, component type mapping, and GUI code generation. In the component identification phase, components are extracted from the UI pages using image processing techniques. Then, a deep learning algorithm based on CNN classification is employed to identify the component types, such as Button or TextView. The component type mapping phase involves mapping the identified component types to their respective counterparts in the target platform. Finally, the GUI code generation phase generates the final implementation code based on the component types and their attributes obtained from the previous phases. Notably, the component type mapping phase plays a crucial role in the framework, utilizing a large map and heuristic rules to generate the final code.

Hassan et al. [24] adopted a top-down approach in their study to gather information from an image. The first step involves extracting and masking the text elements from the image using the Canny edge detection algorithm. This method detects all the edges in the image, including both the text and other UI elements. To eliminate the outlines and boundaries of the other UI elements, they applied a median blur technique. After the application of dilation, a contour detection algorithm is utilized to calculate the bounding box for each text element.

Next, the original image is masked to preserve only the UI elements, and a pre-processing step is employed to extract these elements. This step includes resizing the image, converting it to grayscale, applying Gaussian blur, binarizing it, and thresholding the image. Once the image is transformed and thresholded, the contours of the UI elements inside the image are identified. The output of this process is a set of detected elements segmented into individual images. These images are then subjected to a classification step to predict the type of UI elements.

By employing transfer learning, a classification model that has been trained on large datasets is used. This pre-trained model is then retrained using their own dataset. The model produces an output for each element, which includes the element's UI type, bounding box information, and extracted features. These details are stored in a hierarchical JSON format.

In the method proposed by [25], image processing technology is employed to detect UI components in the application screenshot. Subsequently, the detected components are classified using a customized CNN. To train the CNN, a ReDraw dataset was randomly sampled. To identify UI components in the screenshot image, a series of image processing steps are applied, including grayscale conversion, filtering, thresholding, dilation, and closing. The Flood-Fill algorithm is then utilized to differentiate between distinct sections in the GUI. These detected GUI sections are then analyzed to determine the hierarchical relationships between them. Next, the UI components and GUI sections are segmented and separated based on the size of the detected areas. Finally, the UI components are classified into generic classes that correspond to the ReDraw dataset. The dataset was constructed by randomly sampling 2,500 images from each class from ReDraw dataset, which were then divided into training (70%), validation (20%), and test (10%) datasets.

Introduced in 2020, UIED [26] is a GUI element detection toolkit designed to detect GUI elements using an image-based approach. It provides users with a platform where they can upload their GUIs and automatically detect and identify the elements present within them. The toolkit offers a web interface for user convenience. The approach proposed by [26] divides the detection task into two main parts: non-text element detection and text detection. Traditional computer vision algorithms are employed to extract non-text regions, while deep learning models are utilized for classification and text detection.

To detect non-text elements, the approach makes use of the Flood-Fill and Sklansky algorithms to identify potential layout blocks. The image is then subjected to edge detection and transformed into a binary map representation. The binary map is further segmented into block segments based on the previously detected blocks, and the connected component labeling algorithm is applied to detect GUI elements within each block. These detected elements are subsequently classified using a ResNet-50 model that has been trained on a dataset consisting of 90,000 GUI elements divided into 15 distinct classes. For text detection, the approach employs the

advanced EAST OCR (Optical Character Recognition), which is a deep learning-based scene text detector capable of accurately identifying text within the screenshot image.

Screen Recognition [27] is a system that generates metadata describing UI components based on a single GUI image. This metadata is then utilized by iOS VoiceOver to enhance accessibility. The system is specifically optimized for mobile devices, ensuring efficient memory usage and fast performance. To achieve this, deep learning techniques are employed, leveraging a dataset of iPhone applications. The authors of the study created a comprehensive dataset by manually downloading the top 200 most popular applications from each of the 23 categories (excluding games). Screenshots of visited UIs, along with their associated metadata such as tree structure and properties of UI elements, were collected. However, due to incomplete data, manual annotation was necessary. A group of 40 individuals annotated all UI elements in the collected screenshots using bounding boxes and identifiers, resulting in a dataset comprising 77,637 annotated UI screens.

The UI detection model within the system is designed to extract elements from the GUI and classify them accordingly. To accomplish this, an SSD (Single Shot MultiBox Detector) model with a MobileNetV1 backbone is employed. After the inference process, the output undergoes post-processing to eliminate unnecessary detections. Additionally, the system utilizes a built-in OCR (Optical Character Recognition) service to identify any missing elements. However, since the detector generates separate bounding boxes for each element, it is necessary to group the UI elements. This grouping task is achieved using hard-coded heuristics that have been empirically acquired from a randomly selected sample of 300 cases.

IV. Methodologies

This section classifies the primary methodologies used to convert wireframes or mockups into source code. At the time of writing, these techniques have been categorized into five distinct methodologies.

A. End to End Methodologies

This methodology involves a complete end-to-end approach, where a deep learning model is used to process the mockup or wireframe and generate source code that can be transformed into a user interface. This approach takes inspiration from the way deep neural networks (DNNs) are interconnected to generate textual descriptions (DSL code) from an image. Successful implementation of this technique often relies on having access to sizable datasets with specific characteristics needed to train the models. Therefore, it is often necessary to develop or utilize datasets that meet these criteria. These datasets typically consist of a large collection of wireframes or mockups in image form, along with their corresponding code.

Broadly speaking, this methodology can be divided into three sub-problems. The first is a computer vision problem, as it needs to understand an image and infer the identified objects and their properties. The second is a language modeling problem, as it needs to understand text and generate syntactically and semantically correct code. Finally, the system utilizes solutions from the previous two sub-problems to link the identified objects with their corresponding textual descriptions. This allows the system to be trained and generate DSL code for the GUI image.

Beltramelli's work [19] is considered the first to utilize this methodology, which has since inspired numerous authors to propose various approaches for reverse engineering UI designs into code. Chen [20] improves upon Beltramelli's [19] work by not requiring contextual information as input to the network. In one of its two approaches, [16] employs the same technique as [19], but applied to wireframes instead of mockups.

B. Object Detection Methodologies

Object detection involves identifying, labeling, and defining the boundaries of objects in an image to improve their recognition. This methodology enables a computer to locate and identify objects within an image and gather data about their positions. Unlike image classification, which only labels images containing a specific object (such as a button), object detection creates bounding boxes for each detected object in the image. This means that an image with two buttons would have separate boxes and labels for each button. Fig. 6 illustrates that an image may contain multiple boxes.

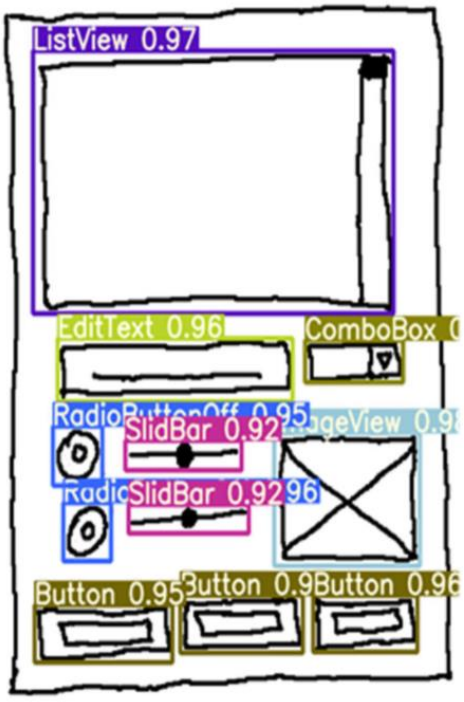


Fig. 6. Detected and recognized elements with bounding boxes.

Convolutional neural networks (CNNs) are well-suited for

object detection and are considered highly effective algorithms for this purpose. Deep learning methods provide advanced approaches to accurately recognize objects. CNN-based object detection approaches can be broadly categorized into two types: two-stage detectors, exemplified by Region-based CNN (R-CNN) and its various adaptations (such as Faster R-CNN), and one-stage detectors, including SSD, RetinaNet, and YOLO.

The study presented in [14] employed a two-stage detector in their approach, as they are known for their higher accuracy rates despite being significantly slower and unsuitable for real-time applications. In contrast, studies [12, 13, 15, 16, 17, 18, 27] utilized one-stage detectors in their approaches, as they offer faster processing speeds and can be used in real-time applications, albeit at the cost of lower accuracy rates. It is worth noting that approaches using object detectors to detect elements in an image often require heuristic methods to determine the hierarchy and layout of these elements.

C. Heuristic Based Methodologies

These methodologies achieve the extraction of constituent elements in a wireframe or mockup through the iterative execution of a sequence of procedures. These procedures involve the utilization of "classic" computer vision algorithms, including traditional techniques such as edge/contour detection, image resizing, grayscale conversion, Gaussian blur application, filtering, thresholding, and morphological operations like erosion, dilation, opening, closing, and boundary extraction. Fig. 7 provides an example of these computer vision techniques.

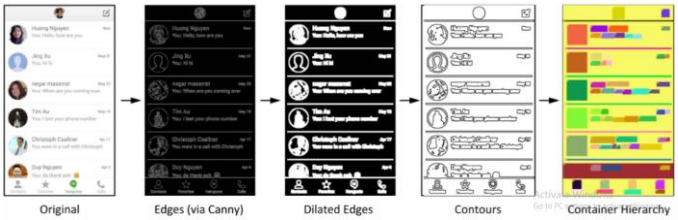


Fig. 7. Example of the computer vision techniques applied on a given mockup by Nguyen et al. [21].

Nguyen et al. [21] were the pioneers in utilizing computer vision and optical character recognition techniques to convert GUI screenshots into application code. Similar techniques were also employed in other studies, such as [8, 28, 29]. While effective for simple GUIs, these techniques may have limitations when dealing with complex GUI layouts or images with gradient backgrounds or photographs. It is important to note that some of these techniques are used as an initial step in other segmentation approaches to facilitate the process of UI element classification.

D. Hybrid Based Methodologies

These methodologies typically begin by using traditional computer vision techniques (Heuristic-Based Methodologies) to extract the location of UI elements. Subsequently, CNN-based classification is utilized to determine the type (class) and other semantic attributes of the UI elements.

Broadly speaking, this methodology can be divided into two primary principles: detection and classification. In the detection stage, computer vision techniques such as image processing, morphological operations, and contour detection are employed to identify rectangles that correspond to each UI element in the GUI image. At this stage, the system can only detect the presence of an element without being able to classify it. These rectangles (UI elements) are then segmented and cropped for the classification stage. After the detection step, the detected elements are subjected to a classification phase, where each element is classified using a pre-trained CNN on a dataset of real-world examples.

This approach can be referred to as "Data-Driven Methodologies" since it heavily relies on data for UI element classification. Having a large and diverse dataset is crucial for successfully training CNN models to classify UI elements. The quality and quantity of the training data significantly impact the accuracy of the classification results, as CNNs rely heavily on data for learning. Additionally, data augmentation or the creation of synthetic datasets can also be beneficial for training CNN models.

Several studies, as cited in [9, 22, 23, 24, 25, 26], have adopted this approach. However, grouping and linking UI elements to determine hierarchy and layout are not automatically obtained and require the use of heuristic rules.

V. Evaluating Methodologies: Results and Limitations

This section consists of two parts: a discussion of the results obtained from the methodologies and a discussion of their limitations.

A. Limitations

Our discussion will focus on the weaknesses and limitations of the approaches utilized in previous related works. Hassan et al.'s study [24] has certain limitations, including a text identification step that may produce false positives, and these false positives cannot be eliminated with median blur. Additionally, the approach is sensitive to gradient variations, which can have a negative impact on contour detection. In Nguyen et al.'s study [21], the use of computer vision techniques also has limitations. For instance, the process of extending the techniques to identify new elements is time-consuming, and programmers need to manually engineer features to classify all new elements.

Beltramelli's study [21] identifies the main drawback of their approach as the continuous maintenance required for the DSL. This adds complexity and effort to the practical utilization of the approach. Furthermore, the DSL relies on a fixed set of UI components and a limited set of style properties, such as color. Consequently, it is not designed to handle the wide variety of component types, styles, and

arrangements present in many example screenshots. Both Aşıroğlu et al. [9] and Robinson [10] limit their consideration of GUI elements to a small number of types. Aşıroğlu et al. only consider TextBox, Dropdown, Button, and Checkbox, while Robinson focuses on images, paragraphs, titles, inputs, and buttons. This narrow focus restricts the practical applicability of their proposed models, as effective approaches should be capable of detecting all types of GUI elements found in the relevant interface.

Chen et al. [20] observe a strength in their study, which is the reliable detection of text elements in GUI images, even when the texts are written in different languages. However, the model has weaknesses that can result in reduced accuracy when dealing with very simple GUI skeletons (those with 10 or fewer components, 3 or fewer containers, and/or 3 or fewer levels of depth). Additionally, the model may struggle to distinguish small UI elements positioned on top of complex images. In the study [23], it was noted that the detected GUI elements were not identified, and applying the heuristic rules required significant manual effort, without offering direct code generation. The study [23] also highlights that only four components (textbox, textarea, checkbox, and button) were used to detect UI elements, which can limit the usefulness of the approach. Moreover, all UI components were assumed to be left-aligned.

One limitation of the approach presented in this study [14] is the variable inference time, ranging from 0.2 seconds to 1.7 seconds, which increases with the number of elements present on a page. Additionally, images captured under low-light conditions, where pixels are darker, can further increase the inference time. The study [10] identifies only five UI elements (Button, RadioButton, CheckBox, Textbox, and Text) in the wireframes, which restricts the scope of their work. Furthermore, the literature does not prioritize the detection of containers in hand-drawn mockups, despite the fact that even with human-drawn wireframes, there may be small misalignments or gaps in the final design, making the extraction of containers challenging.

The study [15] utilized a small dataset comprising only 50 sketches, with a total of 600 GUI elements, which can impact the accuracy of the results. Moreover, the study lacked detailed information regarding the results and the evaluation of accuracy. In the work presented in [25], the precision of the proposed classifier is suboptimal, considering the number of classes in the dataset. Additionally, the detection of UI components relies on traditional computer vision techniques, and the detected contours require post-processing for consistent results. In [26], UIED achieves a low accuracy F1 score of 52% on 5,000 UI images from the Rico dataset. Furthermore, the element detection algorithm is not suitable for noisy images, as it relies on clean input images. Additionally, GUIs with open designs (not perfectly closed regions) may be incorrectly identified as one large region.

The study [16] does not include recognition of internal elements in the side navigation bar, limiting the approach's scope. Furthermore, only a subset of Bootstrap components, including images, videos, buttons, navigation bars, and tables, are covered due to the large number of sketches required to support all components, further restricting the work. The study [17] discovered that the final system is vulnerable to changes in camera perspective, which can result in misaligned boundaries and incorrectly rotated images. Additionally, in the system evaluation, certain components, such as checkboxes and annotation elements, still require further improvement to achieve an acceptable log-average miss rate (LAMR).

Regarding Microsoft AI Lab Sketch2Code [11], the quality of the input sketches was found to significantly impact the generated output. It is also important to note that Sketch2Code has predefined icons and options it can recognize, meaning designers must adhere to a specific syntax when creating sketches. Deviating from this syntax may lead to undesired results.

The approach proposed in [18] is unable to handle text in hand-drawn wireframes. In [12], the Eve tool does not support the use of pen and paper for sketching. Instead, it provides a digital canvas for sketch creation, which may influence the design process. In screen recognition [27], the elements are grouped using hard-coded heuristics that require continual improvement to enhance accuracy. Additionally, approaches like Screen Recognition and others that rely on heuristics are not capable of generating "deep" trees or producing new and complex structures. REDRAW [16] can only detect and assemble a specific set of stylistic details from mockup artifacts, such as background colors, font colors, and font sizes. Therefore, there is a need to expand the range of stylistic details that can be inferred from a target mockup artifact.

B. Methodologies Results

The approaches primarily focus on developing interfaces for web and mobile platforms. Most of these approaches are specifically tailored for designing web-based GUIs, followed by GUIs for the Android platform. A smaller number of approaches are dedicated to developing GUIs for iOS applications [27]. Additionally, some research studies have proposed a multi-platform approach for designing interfaces that can be used across all three platforms [13, 19]. The number of GUI elements detected in interface design varies significantly across different approaches. Some approaches detect a small set of 4 to 5 elements, as observed in [9, 10, 14], while others handle more extensive sets like the 15 GUI elements in [22, 26] or the 19 GUI elements of Material Design, as described in [12]. The majority of existing approaches primarily focus on identifying the type and location of GUI elements and do not attempt to comprehend handwritten text, as seen in [19, 10, 18]. However, approaches that do extract handwritten content from text-based GUI elements often employ Optical Character Recognition (OCR)

techniques as demonstrated in [26, 13, 22].

Regarding datasets, most studies have developed their own datasets as an integral part of their research, as evidenced by [10, 12, 20, 23, 27]. In contrast, a smaller number of studies have utilized pre-existing datasets as [1, 25, 26]. The majority of research studies crawl online stores or websites to collect web pages or mobile applications and then automate the process of capturing screenshots. In some cases, GUI images are synthetically generated by randomly populating an image with sampled UI elements placed at random locations [12]. Alternatively, some studies use GUI screenshots to automatically generate sketches.

TABLE I
OVERVIEW OF THE COVERED STUDIES IN THIS PAPER

Reference	Platform	Input type	Output type	Dataset	Detected elements	Technique utilized	Evaluation results
[9]	Web	Sketch	HTML	Images from Sketch2Code [11]	4 elements	Hybrid Based	The model achieves 96% method accuracy and 73% validation accuracy.
[10]	Web	Sketch	GUI hierarchical skeleton JSON	Screenshots from 1,750 URLs	5 elements	Hybrid Based	F1 score varies from 0.548 (paragraph) to 0.811 (image)
[12]	Android	Sketch	Markup-like DSL XML code	UISketch dataset and Syn-dataset	19 Material Design elements	Object Detection	84.9% mAP with 72.7% AR
[13]	Android, iOS, and Web	Sketch	HTML	149 sketches, 2,001 of GUI elements	10 elements	Object Detection	The inference time ranges from 0.2 sec to 1.7 sec increasing with the number of elements on a page.
[14]	Web	Sketch	HTML/CSS	NA	5 elements	Object Detection	The accuracy is 91% and recall rate 86% of GUI object detection
[15]	NA	Sketch	Markup-like DSL XML code	50 sketch images including ~ 600 elements	NA	Object Detection	NA
[16]	Web	Sketch	HTML/CSS	1100 images, 1000 for training and 100 for testing.	5 elements from bootstrap	Object Detection	Yolo achieved an 88.28% accuracy in the test set
[17]	Web	Sketch	HTML/CSS	8400 mockups image Then, fine-tuning it using 100 real hand-drawn images	9 atomic elements and containers	Object Detection	The detection performance of our approach achieved a mAP score of 95.37%,
[18]	Android	Sketch	XML script	390 images for training (90% training and 10% validation) and 100 images for testing.	13 element	Object Detection	recognition accuracy of 98.54% when tested on various hand-drawn GUI structures designed by five developers
[19]	Android, iOS, and Web	GUI screenshot	Markup-like DSL Code	Pix2code dataset (1,750 Android, 1,750 iOS, 1,750 Web)	NA	End-to-End	Pix2code can automatically generate code from a single input image with over 77% accuracy for three different platforms
[20]	Android	GUI screenshot	Markup-like DSL	185,277 pairs of GUI images and GUI skeletons	NA	End-to-End	Accuracy: 60.28% , The average BLEU score is 79.09
[21]	Android, iOS	GUI screenshot	mobile application	NA	text or images	Heuristic Based	488 screenshots of third-party applications showed that the UIs generated by REMAUI were similar to the original ones.
[22]	Mobile	GUI screenshot	GUI hierarchical skeleton	REDRAW - 14,382 GUI screenshots and 191,300 labeled GUI elements	15 element	Hybrid Based	CNN precision is 91.1%, outperformed both REMAUI and pix2code in MAE
[23]	Android, iOS	GUI screenshot	Markup-like DSL code (Android or iOS)	1,842,580 unique Android screenshots	NA	Hybrid Based	The CNN classification achieving more than 85% accuracy
[24]	mobile and web-based	GUI screenshot	hierarchical JSON format	NA	6 elements and text	Hybrid Based	validation accuracy of 90.2%
[25]	Mobile	GUI screenshot	Detected component marked on the image	ReDraw Dataset (2500 images)	14 element	Hybrid Based	The classification accuracy is up to 96.97%, precision rate is 86.4%, and recall rate is 86.4%
[26]	Web	GUI screenshot	Stored in a JSON file	Rico - 90,000 GUI elements	15 element	Hybrid Based	F1 score of 52% on 5,000 UI images from the Rico dataset
[27]	Mobile (iOS)	GUI screenshot	UI elements with apple voice over	GUIs from 4,239 iPhone applications (77,637 UI screens)	12 element	Object Detection	The model's weighted mAP (IOU > 0.5) is 87.5%.

The evaluation methods employed by each approach differ significantly in terms of metrics and criteria. Most approaches utilize metrics that prioritize the performance of object detection, focusing primarily on accuracy, followed by precision, recall, and F1 value. This differs from the conventional notion that mean Average Precision (mAP) is the most appropriate performance metric for multi-class object detection, a metric only used by a small number of studies [12,

17, 27]. Some approaches also assess the similarity between the generated GUI and the original GUI, either in terms of visual similarity at the pixel level or structural similarity by comparing the hierarchical tree structure. In certain cases, similarity is evaluated manually by the researchers or by GUI designers/developers. Overall, the reported accuracy varies across studies, ranging from the mid-70s to the upper 90s on a percentage scale. However, there can be significant variations in the accuracy of detecting different types of GUI elements. For example, in a study [10], the accuracy for paragraphs was found to be 0.562, while the accuracy for images was 0.896.

Accuracy is a metric that measures how correct the model's predictions are. It is calculated by dividing the number of correct predictions by the total number of predictions. Precision, on the other hand, focuses on accuracy by indicating the proportion of true positives to the total number of predicted positives. False positives occur when an object is incorrectly identified as present in an image when it is not. Recall measures completeness by indicating the proportion of true positives to the total number of actual positives. False negatives occur when an object that is present in an image is not identified. The F1 score provides a balance between precision and recall by calculating their harmonic mean. A high F1 score indicates both precision and completeness. It is used when both precision and recall are important.

Average Precision (AP) is a commonly used metric for evaluating object detection models. It calculates the average precision at different levels of recall and is typically assessed for each object class individually. To compare performance across all object classes, the mean Average Precision (mAP) is often used as the final metric, which calculates the average AP across all classes.

Visual similarity between generated applications and mockups can be evaluated using metrics like mean squared error (MSE) and mean absolute error (MAE) by comparing the pixel values in screenshots of the generated applications with the original mockup screenshots. Minimizing MSE and MAE on test examples indicates high visual similarity. BLEU (Bilingual Evaluation Understudy) is another metric used to evaluate the similarity between machine-generated translations and human-created reference translations.

Table 1 provides a concise summary of the studies covered in this paper, including information on the platform, input type, output type, dataset, detected elements, technique used, and evaluation results for each study.

VI. Conclusion

Generating frontend code from image designs, such as wireframes or mockups, is a challenging task that requires a visual understanding of the images to detect UI elements and their hierarchy. This literature review provides an overview of various techniques and approaches that employ different methods, such as deep learning or computer vision, to automatically generate source code and accelerate the UI design process. Deep learning approaches proved to be well-suited for this task and achieved higher accuracy compared to relying solely on computer vision techniques. However, the use of diverse evaluation measures in this research indicates a lack of a standardized evaluation framework. Furthermore, the absence of standardized and high-quality datasets hinders effective comparison of approaches and future work in this field.

References

[1] IDC. 2015. Mobile Trends Report. https://www.appcelerator.com/resource-center/research/2015-mobile-trends-report/ Accessed: 15 February 2018.

[2] B. A. Myers and M. B. Rosson, “Survey on user interface programming,” in Proceedings of the SIGCHI conference on Human factors in computing systems - CHI ’92, New York, New York, USA: ACM Press, 1992, pp. 195–202. doi: 10.1145/142750.142789.

[3] M. W. Newman and J. A. Landay, “Sitemaps, storyboards, and specifications,” in Proceedings of the 3rd conference on Designing interactive systems: processes, practices, methods, and techniques, New York, NY, USA: ACM, Aug. 2000, pp. 263–274. doi: 10.1145/347642.347758.

[4] P. Campos and N. Nunes, “Practitioner Tools and Workstyles for User-Interface Design,” IEEE Softw, vol. 24, no. 1, pp. 73–80, Jan. 2007, doi: 10.1109/MS.2007.24.

[5] T. Silva da Silva, A. Martin, F. Maurer, and M. Silveira, “User-Centered Design and Agile Methods: A Systematic Review,” in 2011 AGILE Conference, IEEE, Aug. 2011, pp. 77–86. doi: 10.1109/AGILE.2011.24.

[6] C. Dong, C. C. Loy, K. He, and X. Tang, “Image Super-Resolution Using Deep Convolutional Networks,” Dec. 2014, [Online]. Available: http://arxiv.org/abs/1501.00092

[7] B. Varadarajan, G. Toderici, S. Vijayanarasimhan, and A. Natsev, “Efficient Large Scale Video Classification,” May 2015, [Online]. Available: http://arxiv.org/abs/1505.06250

[8] J. Seifert, B. Pfleging, E. del Carmen Valderrama Bahamóndez, M. Hermes, E. Rukzio, and A. Schmidt, “Mobidev,” in Proceedings of the 13th International Conference on Human Computer Interaction with Mobile Devices and Services, New York, NY, USA: ACM, Aug. 2011, pp. 109–112. doi: 10.1145/2037373.2037392.

[9] B. Asiroglu et al., “Automatic HTML Code Generation from Mock-Up Images Using Machine Learning Techniques,” in 2019 Scientific Meeting on Electrical-Electronics & Biomedical Engineering and Computer Science (EBBT), IEEE, Apr. 2019, pp. 1–4. doi: 10.1109/EBBT.2019.8741736.

[10] A. Robinson, “Sketch2code: Generating a website from a paper mockup,” May 2019, [Online]. Available: http://arxiv.org/abs/1905.13750

[11] Microsoft, ‘Microsoft AI lab’, Aug. 23, 2018. https://www.ailab.microsoft.com/ (accessed Nov. 20, 2018).

[12] S. Suleri, V. P. Sermuga Pandian, S. Shishkovets, and M. Jarke, “Eve,” in Extended Abstracts of the 2019 CHI Conference on Human Factors in Computing Systems, New York, NY, USA: ACM, May 2019, pp. 1–6. doi: 10.1145/3290607.3312994.

[13] V. Jain, P. Agrawal, S. Banga, R. Kapoor, and S. Gulyani, “Sketch2Code: Transformation of Sketches to UI in Real-time Using Deep Neural Network,” Oct. 2019.

[14] B. Kim, S. Park, T. Won, J. Heo, and B. Kim, "Deep-learning based web UI automatic programming," in Proceedings of the 2018 Conference on Research in Adaptive and Convergent Systems, New York, NY, USA: ACM, Oct. 2018, pp. 64–65. doi: 10.1145/3264746.3264807.

[15] Y. S. Yun, J. Jung, S. Eun, S. S. So, and J. Heo, "Detection of GUI elements on sketch images using object detector based on deep neural networks," in Lecture Notes in Electrical Engineering, Springer Verlag, 2019, pp. 86–90. doi: 10.1007/978-981-13-0311-1_16.

[16] T. Bouças and A. Esteves, "Converting Web Pages Mockups to HTML using Machine Learning," in Proceedings of the 16th International Conference on Web Information Systems and Technologies, SCITEPRESS - Science and Technology Publications, 2020, pp. 217–224. doi: 10.5220/0010116302170224.

[17] J. Ferreira, A. Restivo, and H. Ferreira, "Automatically Generating Websites from Hand-drawn Mockups," in Proceedings of the 16th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications, SCITEPRESS - Science and Technology Publications, 2021, pp. 48–58. doi: 10.5220/0010193600480058.

[18] A. A. Abdelhamid, S. R. Alotaibi, and A. Mousa, "Deep learning-based prototyping of android GUI from hand-drawn mockups," IET Software, vol. 14, no. 7, pp. 816–824, Dec. 2020, doi: 10.1049/iet-sen.2019.0378.

[19] T. Beltramelli, "pix2code," in Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems, New York, NY, USA: ACM, Jun. 2018, pp. 1–6. doi: 10.1145/3220134.3220135.

[20] C. Chen, T. Su, G. Meng, Z. Xing, and Y. Liu, "From UI design image to GUI skeleton," in Proceedings of the 40th International Conference on Software Engineering, New York, NY, USA: ACM, May 2018, pp. 665–676. doi: 10.1145/3180155.3180240.

[21] T. A. Nguyen and C. Csallner, "Reverse Engineering Mobile Application User Interfaces with REMAUI (T)," in 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, Nov. 2015, pp. 248–259. doi: 10.1109/ASE.2015.32.

[22] K. Moran, C. Bernal-Cardenas, M. Curcio, R. Bonett, and D. Poshyvanyk, "Machine Learning-Based Prototyping of Graphical User Interfaces for Mobile Apps," IEEE Transactions on Software Engineering, vol. 46, no. 2, pp. 196–221, Feb. 2020, doi: 10.1109/TSE.2018.2844788.

[23] S. Chen, L. Fan, T. Su, L. Ma, Y. Liu, and L. Xu, "Automated Cross-Platform GUI Code Generation for Mobile Apps," in 2019 IEEE 1st International Workshop on Artificial Intelligence for Mobile (AI4Mobile), IEEE, Feb. 2019, pp. 13–16. doi: 10.1109/AI4Mobile.2019.8672718.

[24] S. Hassan, M. Arya, U. Bhardwaj, and S. Kole, "Extraction and Classification of User Interface Components from an Image," International Journal of Pure and Applied Mathematics, vol. 118, no. 24, 2018.

[25] X. Sun, T. Li, and J. Xu, "UI Components Recognition System Based On Image Understanding," in 2020 IEEE 20th International Conference on Software Quality, Reliability and Security Companion (QRS-C), IEEE, Dec. 2020, pp. 65–71. doi: 10.1109/QRS-C51114.2020.00022.

[26] M. Xie, S. Feng, Z. Xing, J. Chen, and C. Chen, "UIED: a hybrid tool for GUI element detection," in Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, New York, NY, USA: ACM, Nov. 2020, pp. 1655–1659. doi: 10.1145/3368089.3417940.

[27] X. Zhang, L. De Greef, and S. White, "Screen Recognition: Creating Accessibility Metadata for Mobile Applications from Pixels," in Conference on Human Factors in Computing Systems - Proceedings, Association for Computing Machinery, May 2021. doi: 10.1145/3411764.3445186.

[28] Seoyeon Kim, Jisu Park, Jinman Jung, Seongbae Eun, Young-Sun Yun, Sunsup So, Bongjae Kim, Hong Min and Junyoung Heo, 2018. Identifying UI Widgets of Mobile Applications from Sketch Images. Journal of Engineering and Applied Sciences, 13: 1561-1566.

[29] A. Swearngin, M. Dontcheva, W. Li, J. Brandt, M. Dixon, and A. J. Ko, "Rewire," in Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems, New York, NY, USA: ACM, Apr. 2018, pp. 1–12. doi: 10.1145/3173574.3174078.