FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Live Web Prototypes from Hand-Drawn Mockups

João Carlos Silva Ferreira

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: André Restivo

Second Supervisor: Hugo Sereno Ferreira

July 10, 2019

Live Web Prototypes from Hand-Drawn Mockups

João Carlos Silva Ferreira

Mestrado Integrado em Engenharia Informática e Computação

Abstract

Web designers often use physical mockups, such as hand-drawing, to quickly prototype and convey design ideas. Unfortunately, mockups do not reproduce the look and feel of a web page. To achieve this, developers and designers must transform the mockup to the respective *HTML* and *CSS* pages. This transformation is often laborious and the final result may not appeal to the client. To address the issues, designers must return to the mockup phase and repeat this cycle. This back and forward is resource and time-consuming. Shortening this cycle would improve the capability of the designers to deliver more accurate and tangible product in a shorter time.

The goal of *Live Programming* is to shorten the code-compile-run cycle of software applications, providing real-time feedback to the developers. The aim of this thesis is to apply the principle of *Live Programming* to the web design process, by generating the HTML and CSS code necessary from the hand-drawn mockup. This results in a prototype ready to test, where the client and designer can iteratively make changes to the mockup, shortening the feedback time.

The transformation of digital mockups to *HTML/CSS* is something that already exists and is in continuous expansion some implementations even achieve the generation of reactive user interfaces. Notwithstanding the importance and robustness of this well-established tools, we believe that there is room for improvement mainly in the interpretation of hand-drawn mockups. This field is flourishing, and there are already some advancements. Some of the works follow a more traditional approach, using computer vision techniques to extract information from the mockup and on the other side of the spectrum, there are *Deep Learning* techniques to generate code directly from images, this technique is based on the analogy between generating English textual descriptions from images and generating code from mockups. At the time of this writing, these tools do not seem to make use of semantic annotation to define the behavior of the elements.

In light of the aforementioned problem, the proposed solution is the development of software that, based on hand-drawn mockups, extracts the elements and their relative positioning. These elements were then analyzed in an attempt to match their behavior and structure to the mockup. A design language is also produced for the mockups, to facilitate the image processing steps and to enrich the available options with annotations. The development of such a system usually requires a large quantity of data, so the development of a dataset generator is also proposed.

The evaluation aimed to assess the dataset generator feasibility, whose data is used for the training of *Deep Learning* models. When comparing the model trained with only hand-drawn images with another only using synthetic images, similar mean average precision (mAP) performance was measured of 59% and 60% respectively. Using the model that used synthetic data and further fine-tune using hand-drawn data, the mAP value improved to 95%. Therefore, confirming the feasibility of such a generator for atomic element detection when using low quantities of data.

We believe this work can have an impact on the development cycle of user interfaces by providing a *UI* prototype during the mockup phase, thus speeding the development phase.

Resumo

Web designers costumam usar *mockups* físicos, desenhados à mão, para rapidamente prototipar e transmitir as suas ideias de *design*. Intelizmente, os *mockups* não reproduzem a aparência das páginas web. Para isso, os desenvolvedores e designers devem transformar o modelo nas respectivas páginas *HTML* e *CSS*. Essa transformação é geralmente custosa e o resultado final pode não atrair o cliente. Para resolver os problemas, os *designers* devem retornar à fase de extração de requisitos e repetir este ciclo. Esta repetição é custosa quer a nível de tempo e utlização de recursos. A redução deste ciclo melhoraria a capacidade dos *designers* de fornecer produtos mais precisos e tangíveis em menor tempo.

O objetivo de *Live Programming* é encurtar o ciclo de código, compilação e execução de aplicativos de software, fornecendo *feedback* em tempo real para os desenvolvedores. O objetivo desta tese é aplicar o princípio de *Live Programming* ao processo de desenvolvimento de páginas web, gerando o código *HTML* e *CSS* necessário a partir do mockup desenhado à mão. Isso resulta num protótipo pronto para teste, no qual o cliente e o *designer* podem iterativamente fazer alterações no modelo, encurtando o tempo de *feedback*.

A transformação de mockups digitais em *HTML/CSS* é algo que já existe e está em expansão contínua, algumas implementações chegam a atingir a geração de interfaces reativas. Apesar da importância e robustez dessas ferramentas bem estabelecidas, acreditamos que há espaço para melhorias principalmente na interpretação de modelos feitos à mão. Este campo está em crescimento e já existem alguns avanços. Alguns dos trabalhos seguem uma abordagem mais tradicional, usando técnicas de visão computacional para extrair informações do *mockup*, do outro lado do espectro, existem técnicas de *Deep Learning* que geram código diretamente a partir de imagens, esta técnica é baseada na analogia entre, gerar descrições textuais a partir de imagens e gerar código a partir de *mockups*. No momento da escrita deste artigo, essas ferramentas não parecem fazer uso de uma anotação semântica para definir o comportamento dos elementos.

À luz do problema acima mencionado, a solução proposta é o desenvolvimento de software que, baseado em *mockups* desenhados à mão, extraia os elementos e a sua posição relativa. Esses elementos são depois analisados, na tentativa de combinar a sua estrutura com a do *mockup*. Uma linguagem de *design* também foi produzida o desenho de *mockups*, de forma a facilitar as etapas de processamento de imagem e para enriquecer as opções disponíveis com o uso de anotações. O desenvolvimento de tal sistema geralmente requer o uso de grande quantidades de dados, deste modo, o desenvolvimento de um gerador de datasets também é proposto.

A avaliação teve como objetivo medir a viabilidade do uso de um gerador de datasets, cujos dados são utilizados para o treino de modelos beaseados em *Deep Learning*. Quando comparados dois modelos, um treinado usando apenas imagens desenhadas à mão e o outro utilizando apenas imagens sintéticas, notou-se um desempenho semelhante de *mean average precision* (mAP) de 59% e 60% respetivamente. O refinamento, do modelo que utilizou exclusivamente dados sintéticos, com dados reais, melhorou o valor mAP para 95%. Desta forma, confirmando a viablidade do uso de um gerador sintético para a deteção de elementos atómicos, quando o número de exemplos reais é baixo.

Acreditamos que este trabalho pode ter um impacto no ciclo de desenvolvimento de interfaces de utilizador, fornecendo um protótipo da interface durante a fase de levantamento de requisitos, acelerando assim a fase de desenvolvimento.

Acknowledgements

I would like to express my appreciation to my Supervisor, André Restivo and Co-Supervisor, Hugo Sereno Ferreira, for their time, guidance and constructive feedback during the development of this work.

Finally, I wish to thank my family and friends for the support and encouragement throughout my academic course.

João Carlos Silva Ferreira

"Problems that remain persistently insoluble should always be suspected as questions asked in the wrong way."

Allan Watts

Contents

List of Figures

LIST OF FIGURES

List of Tables

Abbreviations

API	Application Programming Interface
CNN	Convolutional Neural Network
CSS	Cascading Style Sheets
DNN	Deep Neural Network
GPU	Graphics Processing Unit
HTML	Hypertext Markup Language
JSON	JavaScript Object Notation
LSTM	Long Short Term Memory
NN	Neural Netowk
OCR	Optical Character Recognition
R-CNN	Region-based Convolutional Neural Network
RNN	Recurrent Neural Network
UI	User Interface
YOLO	You Only Look Once

Chapter 1

Introduction

This chapter provides an overture to this thesis by contextualizing and supporting it through its motivations and goals. Section 1.1 contextualizes the problem on the conception of user interface prototypes. Section 1.2 describes the motivations behind this problem. Section 1.3 better defines the problem, by introducing current limitations and challenges faced by current solutions. Section 1.4 introduces briefly the main contributions that this document aims to achieve. Finally, Section 1.5 explains the structure and organization of this document.

1.1 Context

In web development, there are multiple ways to represent a website. A designer can just use an hand-drawn image to communicate ideas on the spot or use more sophisticated tools. The latter provides the ability to create a digital wireframe and thus convey their idea more effectively at the cost of some extra time. This representation is then used by developers to produce the underlying code implementation, often named as the prototype, that brings the mockup to the target platform. The act of developing a prototype originates challenges, which if the team is not prepared, they might have a hard time to meet the client's requirements and deliver a solution in time. To optimize their process multiple techniques were developed to take the most of the time and the resources available [DD08]. At a higher level, the development of prototypes can follow a three-phase cycle. A requirements phase, where a designer and the client use mockups to communicate their ideas. These mockups are then sent to the prototyping phase, where developers can use frameworks that follows the concept of Live Programming [ARC+19, BFdH+13], in which the code, compilation, and execution are joined together in a single step. This enables the developers to receive immediate feedback during the development process, thus increasing their productivity. And finally, a validation phase where the prototype is evaluated by the client. This cycle repeats until there is an agreement between the prototype's state and the client's requirements.

1.2 Motivation

Designers can make use of free-hand sketches [WS15] to convey their ideas with the client. As stated by Suleri*et al.*“preliminary semi-structured interviews with 18 UI/UX designers, 88% showed an inclination towards starting the design process by sketching their ideas as LoFi prototype” [SSSJ19]. These sketches enable them to quickly express their ideas in a low fidelity representation. Therefore this strategy is attractive for the designer and the client to discussing and communicate different ideas. One drawback of this approach is the representation itself. The final mockup may not represent the look and feel of the final webpage. Furthermore, if the designer does not refine the mockup until it reaches the developers, some miss assumptions might be made during the prototype development phase. Ultimately the prototype may not match the client’s vision. Leading to the repetition of the aforementioned three-phase cycle in order to fine-tune the prototype until an agreement is made with the client. Coupled with this repetition, there is a cost in terms of time and resources associated. This leads to the main motivation of this work, which is reducing this cycle. Not only giving immediate feedback to the client how their prototype might look like but also the ability to developers concentrate on the system’s business logic.

1.3 Problem

Some solutions attempt and minimize the impact of the prototype development cycle, where hand-drawn and digital mockups are automatically parsed to a user interface. Such solutions can be categorized into three main approaches. The solutions that (a) parse a digital mockup, where the image data, and in some cases their structure, is used to generate a preview or the prototype, (b) the detection of elements present on hand-drawn mockups, and finally (c) the general purpose end-to-end models that employ the use of machine learning in order to directly translate mockups into an intermediate code.

There are multiple challenges that emerge from established solutions. The solutions that use digital mockups, work in a controlled environment, where the mockup representation is concisely defined. Therefore, the irregular nature of hand-drawn mockups becomes a challenge for these types of algorithms, and thus, the parsing using these techniques is not trivial without employing more rules and precision during the drawing process. The solutions that are general purpose can be applied to hand-drawn mockups, but the generated result is at the responsibility of the network making the system somewhat rigid, where the output relies heavily on the quality of the training data. Object detection solutions, aim to detect and further define the user interface hierarchy, but it appears that the concept of containers is not clear.

These solutions face numerous challenges and while human nature might translate visual elements into logical representations intuitively, these tasks are not trivial to be performed and solved by machines. The generation of the final code is something that is not trivial, there are multiple parameters that one can suppose to have in the final code such as, (a) should the image’s proportions be the same, (b) should the element’s dimensions be taken into consideration, (c) how should the styles be attributed. The intermediate code representation is also an important decision to be made, therefore challenging the implementation process on (a) what information should be stored, (b) how this information is stored, (c) how the hierarchy is reconstructed. Before having a hierarchy the most fundamental phase is the acquisition or detection of elements, this imposes several questions on (a) what elements should be considered, (b) if there is any special treatment to extract uncommon elements, (c) what strategy should one use. In case of a model is used in this detection phase, what data can be used for training, should this information be conceived by (a) hand, (b) synthesize a dataset using real-world hand-drawings, or (c) use a purely synthetic dataset.

1.4 Main Contributions

This work aims to shorten the prototype development cycle by contributing to the research field of prototype generation from hand-drawn mockups.

In order to achieve this, the objective was the development and organization of the process responsible for (a) acquire information from an image, (b) develop all the necessary logic to generate an intermediate representation and (c) code generation. A dataset is expected to be created, containing all the information necessary to train the models used to detect elements. The system should be able to translate a given image into the final code, by employing the necessary steps and providing attention to the containers defined by the designer. To that end, helping the developers to focus more on the development of the underlying business logic.

1.5 Thesis Structure

Chapter 2 introduces the background that support this thesis. Chapter 3 creates an analysis of the current state of the art related to the development of user interfaces through computer vision. Different proposals were done over the years. To that end, the methodologies found are categorized for better comprehension. Chapter 4 focuses on the issues and limitations of current implementations, complementing with the proposed solution, validation and expected contributions. Chapter 5 contains step by step the reasoning behind each decision taken to solve the challenges that were faced. Chapter 6 introduces the validation process and reflects over the obtained results. Finally, Chapter 7 reflects the current project's state, as well as its goals and future work.

Chapter 2

Background

Up to this point, the context and problem related to the development cycle and the challenges associated with it were introduced .

This chapter contains concepts considered necessary for a good understanding of this thesis. Section 2.1 describes techniques and algorithms which can be used for the pre-processing of images. Section 2.2 introduces fundamental networks which are applied in a wide range of problems, namely object detection, and pattern recognition. Ending with Section 2.3 which summarizes this chapter.

2.1 Computer Vision Techniques

The human vision is a complex system, from which external stimulus can be converted to knowledge, that enables us to understand the world around us. The computer vision support this complexity, by the numerous algorithms and applications related to this field [Sze11]. This section will only cover the necessary computer vision techniques used throughout this document. Subsection 2.1.1 contains information about some of the most popular noise reduction algorithms, their behavior and their usage. Subsection 2.1.2 introduces an widely used edge technique which is used to extract and filter edges from an image. Finally, Subsection 2.1.3 introduces morphological operators and their combinations.

2.1.1 Noise Reduction

When a camera acquires a picture from the ambient, it is possible that noise is also present. The noise of an image is defined by the random variation of pixel intensities, which cause imperfections in form of grains. There are multiple causes for the formation of noise images, such as electrical fluctuations in the sensor, thermal energy, environmental conditions among others [VA13]. Multiple algorithms were proposed to deal with this issue, some of them are reported in the following reviews [VA13, Zha15, CG13]. In this section, only a particular set of de-noising algorithms will be presented, namely Mean, Gaussian, Median filtering.

Before proceeding any further, it is important to define the concept of convolution. The general idea of convolution in computer vision is the repeated application of a kernel to an image in order to reduce a filtering operation.

Mean filtering. The mean filtering is the most basic filtering application, as the name implies it applies an average smoothing to an image. The kernel used has its entries filled with positive value, one example of a 3x3 can be consulted Equation 2.1. Normally the kernel size value is an odd number causing the pixel suffering the filtering being at the center. Only small variations are attenuated and the edges are not preserved using this filter [CG13].

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \tag{2.1}$$

Gaussian filtering. Sometimes a linear kernel is not enough to preserve some aspects. The Gaussian contains a parameter which can be fine-tuned to achieve the required results. Also, the way the kernel is constructed, it takes more into consideration the center pixels than the outside ones. One reason that make this filter popular is that a convolution of a Gaussian with another results in a Gaussian filter, making it possible to combine multiple filters thus reducing the computational cost. It is also a separable filter, this means the 2D matrix can be divided in two 1D vectors (Equation 2.2) which can be applied one after another in order to achieve some performance gains [CG13]. The edges are also not preserved in this algorithm.

$$\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 2 & 1 \end{bmatrix} * \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \tag{2.2}$$

Median filtering. In contrast to the aforementioned filters median filtering can preserve some of the edges in an image. This resilience to outliers makes this filtering one of the very widely used. As its name implies, this filter substitutes a value with the median contained inside the kernel [CG13].

2.1.2 Edge Detection

Detection of edges is a useful operation to simplify an image to an edge level. Edges in computer vision are defined by discontinuities in a subset of pixels, these can occur due to multiple reasons, such as illumination and depth differences. A formal approach for the edge detection is the derivation of the image, this results in the detection of changes and depending on the order of derivative used the edge can be identified. The general stages of an edge detector are *Noise Smoothing/Reduction*, in order to attenuate unwanted noise that could be wrongfully identified as an edge, *Edge*

Enhancement, in other words, the use of a filter to identify the edges (derivative calculation) and *Edge Localization*, which filters wanted from unwanted edges [BM12].

A popular algorithm that follows the aforementioned steps is the Canny Edge Detector. This detector smooths the image using a Gaussian filter, then the gradient is computed and the edge direction is extracted. In order to determine the edge pixels, a non-maxima suppression algorithm is used, this algorithm compares a pixel's value with their neighbor's values (that coincide with the gradient direction), if it is the maximum value then its part of an edge, if not it is discarded. This results in a fine line of pixels that represent the higher variations in gradient. Finally, in order to filter which of these edges will be used, a *hysteresis* thresholding is applied. It uses two values, an upper and lower bound. In short, every value below the lower bound is discarded as an edge and every value above the upper bound is kept, the values in between are only discarded if they are not connected to an upper bound edge [Can86].

2.1.3 Morphological Operations

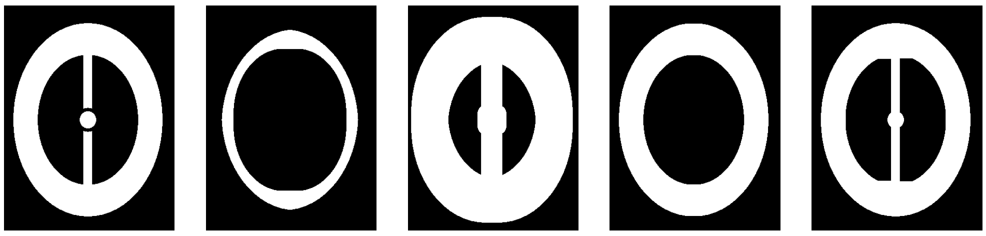

(a) Original (b) Erosion (c) Dilation (d) Opening (e) Closing

Figure 2.1: Morphological operations.

Given a binary image morphological operators can be used in order to clear some outliers from a threshold operation. They fall in the category of non-linear filters. These operators use a structuring element which can act like a normal kernel, or have a different shape depending on the problem to solve. The available operations are dilation, erosion, majority, opening, and closing. Their definition is straight forward, dilation increases the thickness of an object, on the opposite side an erosion shrinks the thickness and the majority sets the value of the majority as the result. The opening and closing operations are compositions of operators. The closing is represented by a dilation followed by an erosion, it is mostly used to remove unwanted segments. The opening is the opposite, an erosion followed by a dilation, it can be used to separate small connections between segments before a counting operation[Sze11]. More than cleaning, these operators, in certain conditions, can be used for segmentation purposes [SS14]. Figure 2.1 illustrates some of the aforementioned operations applied to a binary image.

2.2 Deep Learning

With the advancements in computing hardware, deep learning became a reality and with it, large improvements were made in pattern recognition, object detection among others [LBH15]. Subsections bellow describe at a higher level two types of Neural Networks.

2.2.1 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are a type of networks that has proven to be a good baseline for recognition outperforming state-of-the-art classification systems [RASC14], their use was also explored in sentence classification tasks with great results [Kim14]. Their architecture is what makes them special and useful for classifications tasks. The pioneering work of Yann Lecun *et al.*[LBH98] lead to the creation of LeNet-5, a CNN which was capable of character recognition [LBH98]. And from that moment until know, multiple architectures were developed, better suited for different tasks. Their evolution leads to the development of general object detectors such as Faster R-CNN [RHGS17] and YOLO [RDGF16].

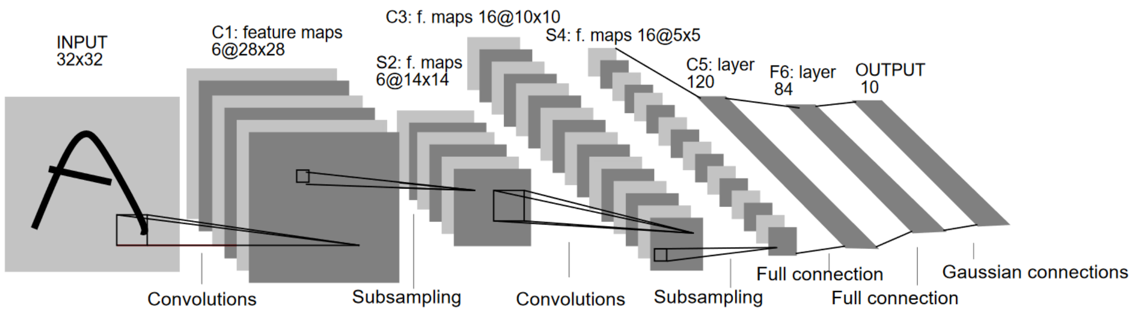


Figure 2.2: LeNet-5 architecture [LBH98].

Figure 2.2 depicts the LeNet-5 CNN's architecture. As the name implies, the most fundamental operation in a CNN is a convolution. As stated before, a convolution applied to an image is represented by a window, which slides through an image and at each location, uses a kernel to calculates a new value for a target pixel. Multiple kernels can be applied to the same image, these kernels are also known as *filters*. The result of a convolution applied to an image often results in a smaller image depending on the procedures used to apply the convolution. CNN's also make use of subsampling to reduce the number of parameters even further. This behavior is the key to improve the performance of a CNN. The two mechanisms used for subsampling are *Strided Convolutions* and *Pooling*. *Strided Convolutions* is the process to skip a certain number of pixels while sliding the aforementioned window, this skipping of pixels results in a smaller image. *Pooling* behavior is similar to a convolution but instead of applying a convolution, for instance, only a simple operator, such as the max value, is applied over the region. For the prediction step, a certain number of fully connected layers, such as softmax, are used at the end of the network [LBH98].

2.2.2 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are a type of networks which receives its last output as an input for a given execution. Figure 2.3 depicts a RNN diagram. This enables the network to use the previous result as a context to infer a new output, updating its state afterward. Their behavior is particularly useful for the processing of vectors which values are somehow related. Their capability of pattern recognition was explored in various fields such as hand-writing[GS09] and speech[GMH13] recognition and textual description generation for image regions [KFF15].

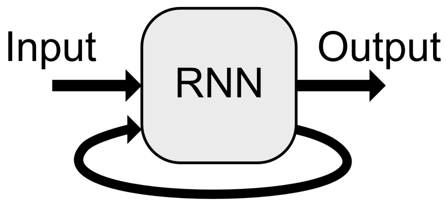


Figure 2.3: Recurrent Neural Network Diagram.

Long Short Term Memory networks (LSTMs) were first proposed by Horchreiter *et al.*[HU97] and from then, it was broadly used and multiple flavors were proposed over the years. What makes LSTMs really special is their capability of learning long term relations. LSTMs achieve this behavior by using its cell state and hidden state to carry information during its processing. The cell state is modified each iteration by a couple of networks called gates, which can remove or add information. The internal architecture of the LSTMs vary in the way their gates, and states interact with each other [LB15].

2.2.3 Object Detectors

One of the great accomplishments in computer vision is object detection, deep learning was responsible for this feat due to its capability of dealing with image gradients, occlusions, and different scales [ZZXW18]. The following subsections describe in a higher level the inner workings of two well-known object detectors, Faster R-CNN [RHGS17] and YOLO [RDGF16].

2.2.3.1 Faster R-CNN

Region-based Convolutional Neural Networks[GDDM14] (R-CNNs) are the basis of Faster R-CNN. Faster R-CNN is an evolution over the Fast R-CNN[Gir15] which aims for real-time object detection. To achieve this, the model is composed of three networks. First features are extracted from a convolutional neural network, the result is then sent to a Region Proposal Network (RPN) which predicts the proposals, finally, both the feature maps and the proposals goes through a Region of Interest pooling phase to define the regions which will be then classified. The process used to calculate the regions of interest is what enables the low processing time hence enabling real-time capabilities[RHGS17]. Faster R-CNN's architecture is represented in Figure 2.4.

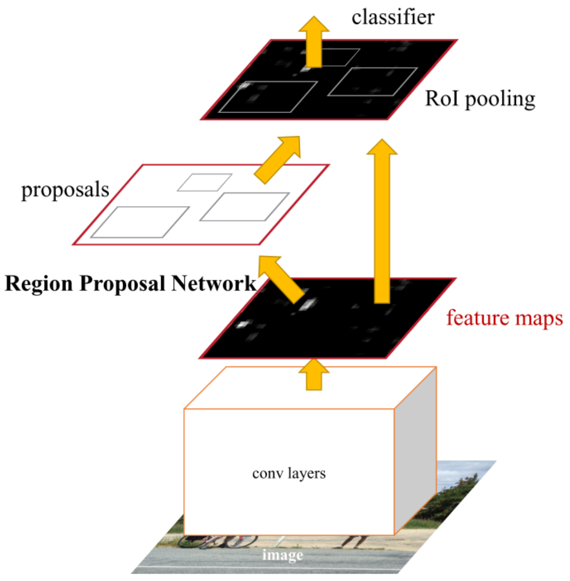


Figure 2.4: Faster R-CNN architecture [RHGS17].

2.2.3.2 YOLO

You Only Look Once (YOLO) distinguishes itself from other R-CNNs by classifying and predicting bounding boxes directly from the image's pixels instead of relying on region proposals. The steps that YOLO uses to accomplish object detection in real-time are the following. When an image is acquired, it is divided in a grid with a fixed size, for each cell, a fixed number of bounding boxes are extracted along with their respective confidence values. In addition, a class probability is calculated from a set of conditional probabilities determined from that particular cell. Finally, the bounding boxes and their respective class probabilities are filtered to determine the final detections [RDGF16].

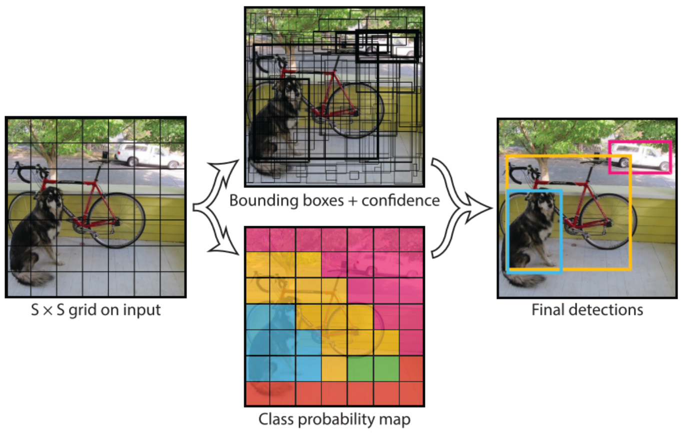


Figure 2.5: YOLO Model [RDGF16].

2.3 Summary

This chapter covered important techniques and concepts to support the comprehension of this dissertation. Some computer vision techniques were covered, that can be used to pre-process an image for a given task. Deep learning fundamentals were also presented using two popular neural networks and their behavior at a higher level. Finally, the knowledge of object detectors was introduced which are capable of real-time object detection.

Chapter 3

State of the Art

The prototype development cycle can be long, as the vision of the client and designers might not be communicated coherently through an hand-drawn medium, the shortening of this cycle is the main goal of this work. It was mentioned the main problem with current solutions and the necessary background introducing concepts of computer vision and deep learning which are considered important for the understanding of this document.

This chapter describes the current state of the art for the conversion of sketches or digital representations to the source code. The main methodologies used to attempt to solve this problem are presented in Section 3.1. Section 3.2 specifies the capabilities of the existing implementations and their possible limitations. Finally 3.3 reflects over the current state of the art.

3.1 Methodologies

The conversion of sketches to a digital form is not something new, [Lan95] is one of the earliest applications in this field of study. At the time of this writing, the techniques found were categorized into five distinct approaches. Subsection 3.1.1 describes a more manual approach where there is a need for the creation of algorithms and heuristics for segmenting the elements. Subsection 3.1.2 contains the end to end methodologies, where the mockup or user interface is fed to a deep learning model and outputs the source code or an intermediate representation which can be translated to a user interface. Subsection 3.1.3 lists all the methodologies that used an object detection technique in order to gather spatial and type information. Subsection 3.1.4 contains the works that rely on the heavy use of data to classify and generate platform-specific code. Finally, Subsection 3.1.5 lists the works that perform structural analysis to the UI using the image and hierarchical layout representation to extract semantic meaning from the elements.

3.1.1 Heuristic Based Methodologies

One technique used to identify elements or widgets from an already drawn user interface is by making use of segmentation techniques in order to detect and extract all said elements (Figure 3.1). This procedure can execute iteratively until all the useful elements are identified. The works related to this paradigm are [HLC16, HABK18, NC15, KPJ$^+$18], each following a different strategy.

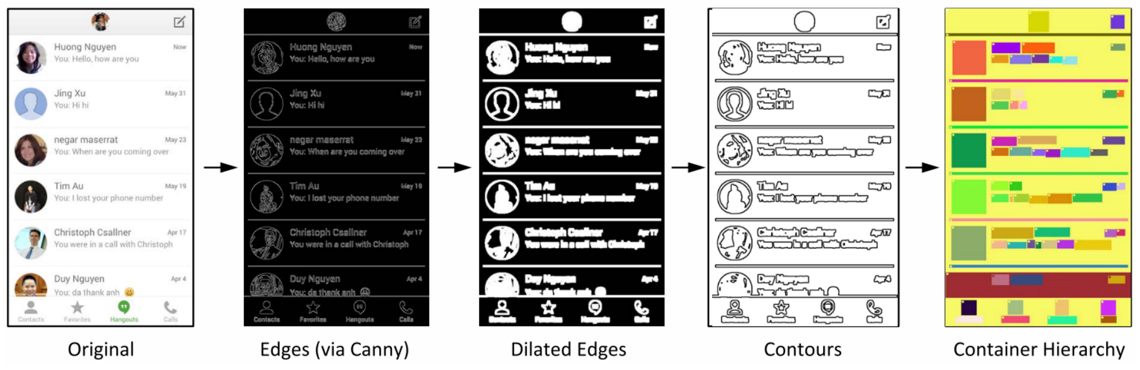


Figure 3.1: Example of the computer vision techniques applied by Nguyen *et al.* [NC15].

The work done by Hassan *et al.*[HABK18] gathers the information from a top to bottom perspective. Firstly, the text elements are extracted and masked from the image. The extraction is done using Canny, an edge detector algorithm. With all the edges detected, the limits of the elements are also present. To remove these limits, they proceed to remove the uniform lines by using a median blur. The result of applying this smoothing filter leaves the text somewhat degraded. Finally, a morphological operation is applied to dilate the remaining pixels, as result, the sections with text represent a higher area than the artifacts. By applying a contour detection algorithm and computing the resulting rectangle the bounding box for each text element is gathered. The text is then masked from the original image to preserve only the UI elements. These elements are then extracted by a pre-processing step and outline detection. The results are the detected elements segmented into separate images which then proceed to a classification step to predict the type of UI component. For each element, the result from the predictor, bounding box and extracted features are then stored in a hierarchical JSON format. This implementation contains some limitations, such as the text identification step which can falsely identify elements which cannot be removed by the median blur. The variation in gradients can affect negatively the detection of contours. The authors suggests possible solutions for these problems, notably the use of alternate methods of text and contour detection and the use of deep neural networks in order to deal with variation in gradients [HABK18].

The procedure followed by Huang *et al.*[HLC16] extracts the elements from the UI from a bottom to top perspective. They accomplish this by detecting the edges of the elements iteratively, all the vertical and horizontal separators are detected, these separators decompose the image in a set of sub-images which then can apply the same separator detection technique. This procedure repeats until no more separators are found. With the elements separated, a classification step is performed to generate tags for the UI elements. Before the classification, visual and structural properties are tied to an element definition. For the classification step first, the leaf nodes are classified using Random Forest and the inner nodes use a bottom-up generating method to determine their tags by adding their children's properties into account. After the classification step, heuristic methods are used to refine the result in case of missing elements or invalid tag match when analyzing the component's children. Finally, the HTML source code can be retrieved using the predicted tag with CSS reflecting the element's visual properties [HLC16].

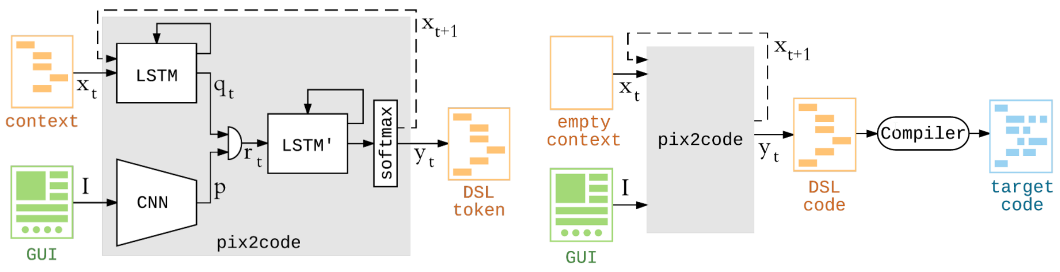
Nguyen *et al.*[NC15], describes an approach to reverse engineering mobile application UIs, as well as the above methodologies, this work uses a single image which is processed through a set of computer vision techniques, it distinguishes itself from the way OCR is used. Given the image, first OCR word detection is performed in order to retrieve all the text information. This technique results sometimes in false positives. In order to filter them, some heuristics are used. From the OCR detection, the bounding boxes of the detected text are extracted, from which it is possible to infer the font size and family. In parallel computer vision techniques are applied to infer the view hierarchy (Figure 3.1). These techniques used are applied in the following order, edge detection using Canny, morphological operation to dilate the edges, and finally contour detection to extract the bounding boxes and producing a hierarchy. With both computer vision and OCR detection steps finalized they are merged. By joining multiple bounding boxes representing the visual words, extracted from the computer vision step, with the OCR lines detected. The final result is then exported to an Android project [NC15].

The work by Kim *et al.*[KPJ+18] purposes methods to disambiguate the detection of UI widgets by studying and relating multiple UI representations and their ambiguities. The formulated statistical rules target 9 types of elements. It uses text OCR analysis to extract keywords which can then be used to disambiguate the detection of elements. As an example, a rectangle with a verb can be considered as a button because it suggests an action. [KPJ+18]

3.1.2 End to End Methodologies

This section includes the works that are based in an end to end approach, where the mockups are directly transformed to code or an intermediate representation. The main inspiration of this technique is closely related to the way DNNs are used to generate textual descriptions from an image[YJW+16, KFF15]. These implementations usually require large amounts of data to train the models thus the use or development of datasets are also required.

The work from Beltramelli [Bel18] seems to be the first using this methodology, which inspired multiple authors to come with different ideas for reverse engineering UI pictures to code. They divided the problem in three sub-problems. First, a computer vision problem related to scene understanding, to detect and infer the properties of the detected objects. Second, a language modeling problem in order to generate syntactically and semantically correct code and finally the junction of the two previous sub-problems to relate the detected objects with their corresponding textual description. To solve these problems the architecture model is formed with three Neural



(a) Training (b) Sampling

Figure 3.2: Proposed *pix2code* architecture [Bel18].

Networks (Figure 3.2), a CNN and two LSTMs. The CNN solves the computer vision problem, it receives as input an image with 256x256, that, after processing, results in a vision-encoded vector related to that image. To solve the language model problem, two LSTM layers, with a size of 128 cells each, are used in order to encode a current context into an intermediate representation language-encoded vector. Both the language-encoded and vision-encoded vectors are concatenated into a single vector, which is fed into an LSTM decoder. This decoder is implemented in a stack of two layers with a size of 512 cells each. The decoder responsibility is to learn how the objects presented in the image are related to the tokens from the DSL code. Finally, the decoder is connected to a softmax layer in order to extract the probability of the next DSL token. To train the model a pair of UI image and its DSL context is needed. To extract a UI from the model, the target UI image and an empty DLS context initialized with the special start attribute are needed [Bel18]. As stated by Zhu *et al*.[ZXY18] this model contains some limitations, some of which are: the need to specify the range which the code is generated; not taking into account the hierarchy of the UI and its code [ZXY18].

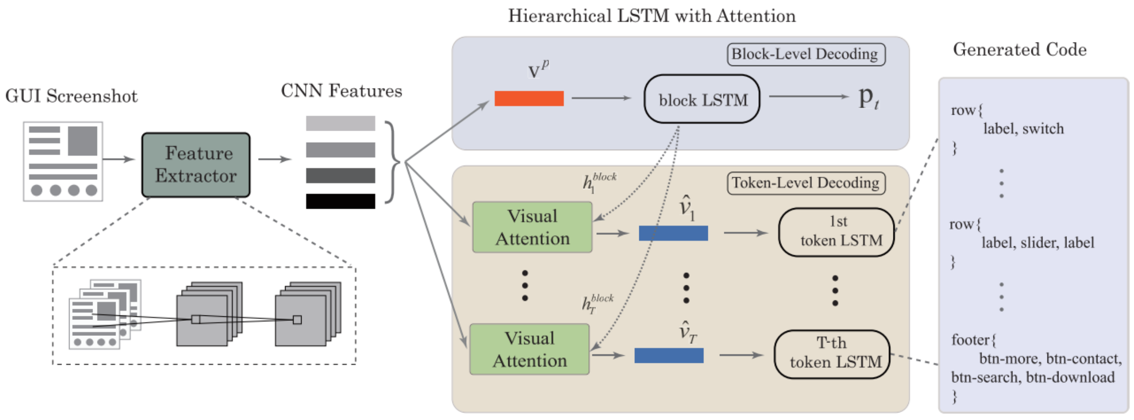


Figure 3.3: Zhu *et al*.proposed architecture [ZXY18].

Zhu *et al*.[ZXY18] aims to improve upon the implementation of Beltramelli [Bel18] by removing the need to feed the network with an initial context. The way they accomplish this is by using an initial CNN to obtain the high-level features of the image and then are fed to a hierarchical LSTM with attention. The hierarchical structure is composed of two levels, a block LSTM, and a token LSTM. The block LSTM receives the high-level features as input and derives the size of code blocks to generate the program, the hidden state is then fed to an attention model, to filter the most important high-level features from the CNN output. The result from the attention model is then forwarded to a token LSTM to generate the corresponding code. Figure 3.3 outlines the proposed architecture. With this approach, it is possible to train the model in an end-to-end process, using just the UI image and the whole DSL code [ZXY18].

The solution provided by Chen *et al*.[CSM+18] contains some similarities to the previous method, in this methodology, after processing a UI image representation using a CNN, an LSTM encodes the CNN's output to an intermediate representation vector which is decoded by a final LSTM in order to generate the final intermediate code. This solution also provides a more accessible training strategy, without the need to provide context as input to the network [CSM+18].

Not related to the generation of UI, Ellis *et al*.[ERSLT17] created a model to convert hand-drawn sketches into the corresponding LaTeXsource code. The model receives the target image and the current rendering of the generated drawing (which is clear in the first iteration), these images are sent to the CNN as a two-channel image, after the processing the results are fed to a multilayer perceptron and a differentiable attention mechanism to filter the most relevant features at the current processing step. The output is code to draw the detected element. This code is then rendered and fed again to the network until all the drawing code is gathered. After retrieving the code specification, it is then processed through a program synthesis engine in order to structure the code [ERSLT17].

3.1.3 Object Detection Methodologies

Object detectors provide a useful ability to identify the bounding boxes of detected objects. This ability makes them a tempting option to identify UI elements in an image. The works in this section reflect the use of such object detectors in order to generate a graphical user interface.

Suleri *et al*.[SSSJ19] developed a software prototyping workbench, which uses object detectors to detect the mockup's elements. To train the model, they used a synthetic dataset which was built using real hand-drawn wireframes. To build the dataset, firstly a selection of 350 participants drawn and annotated 5906 sketches, of 19 UI elements. Then to generate the synthetic dataset, a random image was filled by randomly sampling the aforementioned elements and placing them at random. The result was an annotated dataset with 125000 images. They divided their system interactions in three distinct fidelity representations, in which each one provides different interactions. (1) The low fidelity level represents the mockups in their pure form, in this representation the user can create or modify an existing mockup as well as modify the mockup's interactions. (2) The medium fidelity level is represented by the result of the object detector and a generated medium fidelity design, in this phase, the user has control over the conversion process and enables the user to change the properties of the detected elements as well as the manual detection of elements in case of miss detection. (3) The high fidelity level is defined by the generation of the final UI, where the user can define different themes and generation of the final code.

The work done by Yun *et al.*[YJE$^+$19] makes use of YOLO, one of the object detection networks. By employing the use of transfer learning from an already existing network, this network was then trained to learn the features from the UI elements. Given an hand-drawn mockup, it is then fed to the YOLO network which then outputs the detected elements associated with their respective confidence level. This output is then transformed in a hierarchical structure which is used to generate the code to the target platform [YJE$^+$19, PYE$^+$18].

The methodology followed by Kim *et al.*[KPW$^+$18] is different from the previous based on how the underlying layout of a mockup is detected. Their approach can be divided into two steps, layout detection, and UI element recognition. For the UI element recognition, Faster R-CNN was used. For the layout detection, computer vision techniques were used to merge disconnected edges. Since the layouts are constrained to the x and y-axis, a slope filtering technique is used to avoid slopes which are not horizontally or vertically aligned. Finally, a correspondence line algorithm is executed to determine the layout, for each detected layout the algorithm is executed until no more layouts are found [KPW$^+$18].

3.1.4 Data-Driven Methodologies

The work of Moran *et al.*[MBCC$^+$18] aims to provide automated UI prototyping for mobile applications using a data-driven approach. The dataset is created using data-mining and automated dynamic analysis techniques. The applications are gathered from multiple app-stores and processed to extract the UI hierarchies and UI components. The methodology is distinguished in three main principles: detection, classification, and assembly. The detection step contains two approaches, using computer vision techniques to extract the elements, such as Canny edge detection, morphological operations, and contour detection. The second approach is using an existing structural file where the UI visual properties are already discriminated. After the detection step, the detected elements are submitted to a classification phase, where each element is classified using a previously trained CNN. With the elements classified the assembly phase begins, where a K-Nearest Neighbor algorithm is used for UI determination. To achieve this determination, the UI is compared with a large dataset of previously extracted layouts. The intent with this methodology is to find patterns in the mockup that match real-world examples to generate something that a developer would actually create. Some style-features are also gathered during this process such as background, font color as well as font size. As stated by the authors, this implementation contains some limitations, some of which are the limitation to the screen size, since the K-NN hierarchy construction phase is limited by this factor. The use of multiple screens implies a manual work for the developer to create the navigation flow [MBCC$^+$18].

3.1.5 Structural Information Based Methodologies

This section contains methodologies that do not necessarily generate code but perform structural analysis to the UI using both the image and hierarchical representation. These methodologies are useful in terms of acquiring the semantic relationships between the elements, which can be used in conjunction with an image based code generation algorithm to improve and fix possible semantic errors.

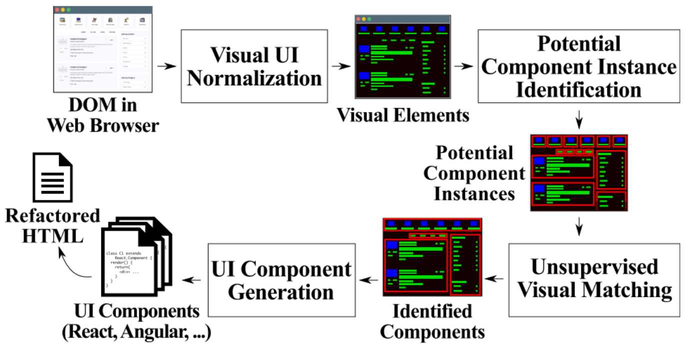

DOM in Web Browser
Visual UI Normalization
Visual Elements
Potential Component Instance Identification
Potential Component Instances
Unsupervised Visual Matching
Identified Components
UI Component Generation
UI Components (React, Angular, ...)
Refactored HTML

Figure 3.4: Bajammal *et al.*approach overview [BMM18].

The search of patterns hierarchically and visually can lead to the identification of reusable web components, this identification is imperative for the organization of a complex project. Bajammal *et al.*[BMM18] aim to automatically generate reusable web components from mockups. Their methodology starts by creating a visual representation from the original DOM code, where the bounding box of the respective elements is drawn in a color-coded format, this format distinguishes two types of visual elements, text, and images. After the visual UI normalization stage, a potential component instance classification is performed by using an approach which maximizes the number of repetitions for each component and their encapsulated component instances. Using the DOM from the webpage a tree is built from a bottom-up perspective where, at each iteration, it is calculated a modularization potential based on the formula for the maximization goals aforementioned. In the final iteration, when the maximal modularization potential is achieved, the results are processed to an unsupervised visual matching phase. This phase uses the image previously calculated, in the visual UI normalization stage, to extract a feature vector for each potential instance. Finally a cosine distance is calculated for each pair. Next, the unsupervised clustering process begins, this process uses variable-density clustering with a hierarchy of densities [CMS13]. With the identified components in place, a UI Component Generation phase is done to generate an intermediate representation of the components which then can be translated to a framework of choice. [BMM18].

Another methodology that makes use of the hierarchical view to understand the semantic properties of UIs is the work from Liu *et al.*[LCS$^+$18]. Their solution is targeted to mobile applications, they created a lexicon of design semantics, which categorizes UI components, text buttons, and icons. Given a screenshot and its respective UI hierarchy the system can then identify UI components, using a code-based and icon classes, using vision-based techniques. For the UI approach where they use heuristics to classify the components, with the UI hierarchy it is possible to infer the class ancestors of a specific component which are used to perform this classification. The UI classification step pre-processes the icons before classifying them using a CNN, the result is then analyzed in an anomaly detection phase which purpose is to distinguish icons and images.

This semantic identification can be made useful to create more robust generative models of UI [LCS+18].

3.2 Implementations

From the above methodologies, some include implementations such as pix2code[1]. Since it is a general solution to convert images to code, there is already an adaptation that supports hand-drawings[2]. In this methodology, the dataset was created by using custom CSS to make websites, from the pix2code dataset, look like hand-drawn images. Then these images were modified by applying operations such as skew, shifts, and rotations to generate more irregularities. This section contains the implementations without a formal document explaining in detailing their methodologies, not found or available at the time of this writing.

Sketch2Code[3] is an implementation from Microsoft[4] which focuses on the understanding of hand drawing with text recognition capabilities to generate HTML or any other markup language used to display the final UI. The analysis is performed at the atomic element level, which makes takes into account the free expression of the designer, it is nonetheless not clear how capable the system is when identifying more complex designs and hierarchies.

teleportHQ[5] supports the development of user interfaces by providing tools to help designers achieve their objectives. One of their experiments is the teleportHQ vision API [6] which uses object detector methodologies in order to identify atomic elements in a mockup. It seems that his solution just provides the output of the object detection algorithm without extraction of a possible hierarchy.

Airbnb[7] implementation[8] instead of aiming for the atomic element level, they train a model to classify sketches, each representing an already existing component in their design stack, this approach seems to work well when it is expected that no other than the already available components fulfill the requirement's needs. Compared to the above methods, this simplifies the tasks needed to extract useful information, yet it can limit the possibility for more complex or custom layouts.

3.3 Conclusions

Due to an increasing interest in this area, a large number of works appeared recently, as a result, multiple methodologies were developed to solve this problem. This section explained the developed methodologies into 5 categories. Heuristic-based, which mainly used computer vision

[1]https://github.com/tonybeltramelli/pix2code
[2]https://github.com/ashnkumar/sketch-code
[3]https://sketch2code.azurewebsites.net/
[4]https://www.microsoft.com
[5]https://teleporthq.io/
[6]https://github.com/teleporthq/teleport-vision-api
[7]https://www.airbnb.com/
[8]https://airbnb.design/sketching-interfaces/

techniques, such as described in Section 2.1, to extract useful information and compose the UI hierarchy. End-to-end based methodologies, that aims to create an deep learning model that receives an image and some code representation to train the model, these techniques offer great flexibility and are proposed as a general solution that could be applied to any UI system. Object Detection based, that use well-established object detectors to extract the elements from a mockup, which outputs the class of the atomic element as well as its position. Data-Driven methodologies, which takes advantage of a large amount of data readily available, categorizing it and using it creatively for the detection of components. Ending with Structural Information methodologies, that uses the layout of an interface to extract useful semantic relationships or detect reusable components. These last methodologies are not meant to generate a user interface from an existing image but can be used in conjunction to other methodologies so to improve the final results. With each methodology the datasets used were also introduced, they varied by the necessities of the strategy in use. Some datasets contained (a) digital representations along with their intermediate code, (b) hand-crafted datasets properly labeled for object detection models, (c) purely synthetic approaches for training end-to-end solutions, and (d) creation of a synthetic dataset using existing hand-drawn elements along with their annotations. Finally, it was also mentioned available implementations which, at the time of this writing, do not contain a formal document available, but are relevant in their approach to solve this problem. One take away of this study is that there is not a bulletproof solution for the resolution of this problem, it mainly depends on what are the system capabilities to be developed versus the time and features available for the implementation of the selected methodology.

Chapter 4

Problem Statement

With the intention of improving the prototype development cycle by providing an prototype preview during the drawing phase. This chapter objective is to identify some faults on the current implementations and proposes a possible solution.

Until this point, the background necessary to understand the underlying concepts was introduced, including topic about computer vision, deep learning and object detectors. It was also discussed the methodologies found during the development of this document such as heuristic, end-to-end, object detection, data-driven, and structural information based methodologies.

Section 4.1 aims to transmit the problem definition. Section 4.2 introduces the fundamental challenges related to the problem of converting mockup representations into prototypes. Section 4.3 presents the thesis statement. Section 4.4 introduces specific challenges associated with to this work. Section 4.5 includes the summary of contributions. Section 4.6 introduces the validation methodology responsible to measure the project's impact. Finally 4.7 reflects over the problem statement and the proposed solution.

4.1 Problem Summary

The process of developing a user interface prototype can be a challenging task. Starting with the designers and the client, both may communicate using some sort of visual representation. This representation while not reflecting the final result is then used by the developers to produce the final prototype. Finally, the client then evaluates the prototype and some adjustments may be made due to the limits presented on the low fidelity representation, used in the initial phase. Repeating this cycle involves the cost of resources and time, thus the act of reducing it pleases both the company and the client. The transformation of images representing mockups into prototypes is something that had gathered a lot of attention recently. Mainly with the aim of creating a shortcut by providing a preview of the final user interface. Multiple proposals appeared over the years, with notable improvements and different strategies. Most of them were originally design for digital mockups, which are a more controlled environment, in terms of item alignment and representation [HLC16, HABK18, NC15, KPJ+18, MBCC+18]. Solutions that target hand-drawn mockups can be end-to-end solutions [Bel18, ZXY18], or using object detection techniques [YJE+19, KPW+18]. The end-to-end solutions, generate intermediate code on the fly, leaving the interpretation of the image and the generation of code on the network's behalf. Therefore, developers have lower control over the produced outcomes, meaning that the quality of the dataset is what determines the final results. The use of object detectors provides a controllable environment, but from the solutions that were found, the detection of the component's hierarchy made with composite elements is not possible or not clear. While a process is clearly defined for an interactive UI approach [SSSJ19], the definition of an internal pipeline could improve the development process.

4.2 Fundamental Challenges

The act of visualizing a given mockup and identifying the elements presented is something that humans take for granted, using machines for this work means multiple challenges must be met and solved. Given the fact that code must then be generated from the image, then the problem starts to become even more challenging, with multiple areas and decisions to take into account.

1. **"When taking the image representation into account, code generation is not trivial."** An image may contain a lot of elements with different sizes and representations when generating such code it could be useful for a designer to reuse these representations. Since it is a hand-drawn representation the elements might not align properly if used as it is. Therefore, the proportions of close elements is something that could be taken into account when trying to represent them. How the hierarchy should be reflected in the final code is also important, it is one of the most fundamental points and introduces to the next point.

2. **"The intermediate representation used by code generators is ambiguous."** Creating an intermediate representation is useful to later be re-used by multiple code generators, and this includes challenges including (a) the information that should be stored for each element, (b) how the hierarchy should be represented and rebuilt, and (c) what types of hierarchical containers must be used. Reconstructing the hierarchy is in itself a whole challenge due to its ambiguous nature, where multiple representations can lead to the same output.

3. **"Acquiring information from an image is not trivial."** When extracting information from an image there are several questions that must be taken into account beforehand, for instance, (a) what elements should be targeted, (b) what information should be extracted. The approach to extract such elements are wide, depending on the mockup representation. While a digital mockup might be more concise thus simpler for the elements to be extracted, when using hand-drawn mockups their unpredictable nature complicates some established methods.

4. **"When opting for a model to extract elements, the dataset to be used can vary."** Techniques that employ the use of deep neural networks usually requires large quantities of data. In many cases, this information (a) might not exists, (b) difficult to adapt, (c) the required labels might not be presented, (d) not appropriate for the project needs. Therefore producing and hand-drawn dataset, or generating a synthetic dataset from an already existing hand-drawn dataset, might seem the easiest approach.

5. **"Notwithstanding the human's drawing capabilities, the representations might be ambiguous."** It is important for these systems to have distinct representations for each type of element. A given element might have multiple representations but they must differ from all the others. Furthermore, the level of coherence and alignment of the items might also impact the results and the level of complexity of the overall system. This implies that a mockup design language must be taken into account before designing or producing the datasets.

4.3 Thesis Statement

This work supports the research done around object detection methodologies and believes in the use of purely synthetic data generators to train these models, therefore serving as a starting point. Moreover, it supports the use of a extensible pipeline where each main module could be enhanced by employing different strategies, contributing to the main goal which is the generation of prototypes from hand-drawn mockups. In light of the aforementioned points, this document proposes the following hypotheses:

H1. The use of synthetic datasets improves the generation of code prototypes from hand-drawn mockups.

H2. The ability to specify and/or infer containers during the drawing process of mockups, improves the layout of the generated prototypes.

4.4 Specific Challenges

From the thesis statements mentioned above, there are some specific challenges that emerge which are:

1. **"How the generation of code can be made and what should it take into account?"** The intermediate code is often used when using end-to-end solutions or other techniques that target multiple UIs. Therefore this representation must be translated to the target code. The solution could be the usage of a language mapping model to generate the code [Bel18, LGH$^+$16], or a creation of a parser [MBCC$^+$18]. The information that such generator must take into account really depends on the richness of the intermediate code representation, which as an example could be, font-size, color or text information [MBCC$^+$18, HLC16].

2. **"What information should be included in the intermediate representation?"** Since the preservation of the spatial relationship between elements is important, usually an intermediate representation is used to express the hierarchy of the detected elements. Their contents depend on what properties are acquired for each element but they can be classified as (1) style properties: which are the color, element's size, and (2) spatial properties such as the hierarchy and global position, the surrounding neighbors, among others.

3. **"How is the hierarchical structure reconstructed?"** Some techniques aim to compare the detected elements with already existing mockups, thus re-constructing the hierarchy based on a real-world example [MBCC+18]. But there are also alternatives that aim for the usage of algorithms, such as the detection of line separators in the mockup [HLC16, KPW+18].

4. **"What techniques are used to extract element related information from a given image?"** Due to the great advancements in deep learning, CNNs are widely used as the basis to perform the detection and classification of the elements in an image [MBCC+18, Bel18, ZXY18, CSM+18, ERSLT17, YJE+19, KPW+18]. When the environment is constrained and regular, it is also possible to make use of traditional computer vision techniques [HABK18, HLC16, NC15].

5. **"How hierarchies are extracted from a given Image?"** While the reconstruction of hierarchies from end-to-end models is not clear, it is possible to use them to extract this information. Another more traditional approach is to use computer vision techniques in order to segment the different elements iteratively to construct the hierarchy. From the analyzed works it is possible to extract a wide amount of information from images such as font size, color, and images. It is also possible to add more information to the images such as custom annotations to assist in the classification phase of an image [YJE+19].

6. **"What elements can be detected by current methodologies?"** Current implementations do not appear to have restrictions when detecting data, the works analyzed aim at different platforms but the identified elements range from buttons, checkboxes, radio buttons, text fields, images, icons, text, among others. Some works provide the use of annotations to help in the classification tasks [YJE+19].

4.5 Summary of Contributions

The expected contributions relate closely to the research questions previously answered. It is expected the development of two main approaches, as alternatives to the current solutions in the literature. Which rely on the evaluation of a purely synthetic dataset generator and the development of a pipeline with attention to the segmentation of containers and provide support for annotations. In more details the expected contributions are:

1. **Container centric segmentation approach.** There are some algorithms proposed in order to extract hierarchical information [KPW+18]. But due to the complex nature of the

hand-drawings, an algorithm to segment and detect only the hand-drawn containers could contribute to more specialization and personalization from the designer's part.

2. **Synthetic mockup generator.** The challenges related to the acquisition of a useful dataset could be minimized by the use of a synthetic generator, where the researchers have control over the elements used, their annotation and even creating the possibility to create segmentation masks. The feasibility of such a generator is also taken into account in cases where the hand-drawn data is in low quantity.

3. **Extensible pipeline.** Such pipeline introduces advantages when comparing to the use of end-to-end solutions, as the results from an intermediate step could be easily debugged. Therefore providing insight over the hierarchy reconstruction. By providing the data flow of the pipeline, new modules could be adapted with the current state of the art methodologies in order to augment the final result.

4. **Proposal of a hierarchy reconstruction algorithm.** Some approaches aim to use algorithms to detect line separators or the use of real-world data. While some algorithms are clear, in other solutions are absent. Therefore an algorithm is proposed, wich aims to recreate an hierarchy given the element's relative positioning and an preferred orientation as the main input.

5. **Proposal of annotations.** This work supports that the notion of annotation could be used to augment the designer's options. By supplying a set of annotations a designer could associate a custom style to a specific annotation, thus controlling the produced prototype.

4.6 Validation Methodology

As mentioned above, the underlying objectives for this work are to develop a synthetic generator dataset and an extensible pipeline with support for annotations and with special attention to the segmentation of containers.

With this in mind, the synthetic dataset must be able to generate the images that represent the mockups and their respective annotation file. Such a tool must be evaluated for how feasible it is when used to train a given model. Therefore, to fulfill this statement, an established model must be trained in three different versions, (1) using real-world data, (2) using synthetic information, (3) using synthetic and then fine-tuned with real-world data.

The development of the container segmentation technique must also be evaluated by comparing the raw performance it has when tested against a set of hand-drawn information. Similarly, by isolating the detection models from the whole pipeline a measurement of its detection capabilities should then reflect on what elements were difficult to be generalized.

The evaluation of the execution times is also something that should be taken into account. Notwithstanding the robustness of the current state of the art models, the image acquisition step can still have an impact over the final result. Naturally, adjustments over the image acquisition steps are quickly iterated when the response time is lower.

4.7 Conclusions

In light of the limitations from the currently available tools, the goal of this project is to develop a technique to segment and identify purposely drawn containers, the proposal of annotations and a hierarchical reconstruction algorithm. Furthermore, it also aims to develop and evaluate the feasibility of a purely synthetic dataset generator when used to train the models used in this system.

From this work, it is expected to provide a new direction of how mockups could be enhanced using annotations, and how a pipeline could ease the development process, enabling for more experiments to be made in future works.

Chapter 5

Implementation

In this document, The prototype development cycle was described along with the problem that arises from this cycle and what solutions currently exist. Then the necessary background related to computer vision techniques and Deep learning was introduced for a better understanding of the vocabulary used throughout this document. The state of the art solutions was then presented and categorized according to their approaches. An evaluation of how the shortcomings and the challenges of these implementations were given along with the proposed solutions and contributions that this work aims to achieve.

Section 5.1 starts by giving a high-level overview of the implementation, by introducing the technologies and strategies used to tackle the problems faced during the development of this project. Section 5.2 describes in detail the development of the synthetic mockup generator. Section 5.3 reports in detail the reasoning and development steps taken for the development of the pipeline, which is responsible for the translation of hand-drawn mockups to code. Section 5.4 reflects on the implementation is also present.

5.1 Overview

When it comes to the generation of websites from mockups, there are multiple strategies to choose from. Recent implementations make use of *DNNs*, which made feasible the interpretation of images to generate different outputs. As aforementioned, examples of outputs of these networks can be a collection of bounding boxes, generated by *Object Detectors* or generation of textual expressions by employing the usage of *CNNs* and *LSTMs*. These networks often require large amounts of examples to train these models. The gathering of this data may not be trivial, it may be (a) non-existent, (b) low-quality or untrustable, (c) missing the required annotations, (d) not appropriate for the project's needs. The solution is often to manually create the dataset [YJE+19] or generate a synthetic dataset based on a previous hand-drawn dataset [SSSJ19]. Inspired by the latter, this work presents the development of a tool capable of generating exclusively synthetic annotated datasets and analyses the feasibility of such dataset when used to pre-train deep neural networks. This tool was developed using *HTML*, *CSS*, *Javascript* and *NodeJS*, thus providing an interactive web interface and a command line interface for the generation of large batches of examples. *RoughJS* was also used to improve the looks of the generated results by achieving a hand-drawing style.

Closely related to the translation of mockups into prototypes a pipeline is also proposed, with the separation of responsibilities in mind contributing to a pluggable research environment. With this system, the generation phases can be modified to experiment with different strategies and thus, providing more control over the generated outputs when comparing to end-to-end solutions. For this implementation python was the programming language of choice due to the abundance of modules that allow ease of use of machine learning resources.

5.2 Mockup Generation

The representation of user interfaces is often done by using two types of elements: (1) atomic elements, such as buttons, radio buttons, and dropdowns, and (2) containers, which aggregates multiple elements. The proposed generator takes both types of elements into account. The developed solution is described and only used one layer of containers, in terms of the hierarchy only a depth of two is achieved by using this system. However, the container generation phase can be applied recursively to allow more complex hierarchies to emerge.

Javascript was used along with with HTML's built-in canvas to output the final result. In order to improve the style of the drawn lines, the library RoughJS[1] was used. It provides the ability to draw basic shapes as well as the tweak of the brush used, which was used to mimic different drawing instruments.

The mockup's generation process is sequentially executed and arranged in the following way: Subsection 5.2.1 describes the establishing process of the working area within an image, which defines the dimensions and position inside a blank image where the drawing will take place. Subsection 5.2.2 proceeds to the definition of a container population algorithm, which places random containers inside the working area. Subsection 5.2.3, likewise, describes a procedure to define the area with a type of element will occupy. This section takes the container's into account and all the available space within the working area is then filled with these element defining areas.

Subsection 5.2.4 uses the aforementioned areas in order to fill them with their respective element, taking into consideration a set of rules. Finally, Subsection 5.2.5 describes the techniques used to add variation when drawing the final elements.

Figure 5.2 depicts a high-level overview of the process followed throughout this section. Figure 5.1 includes an overview and describes the pictographs used.

Examples generated from the current described implementation can be consulted in Appendix A

[1] https://roughjs.com/

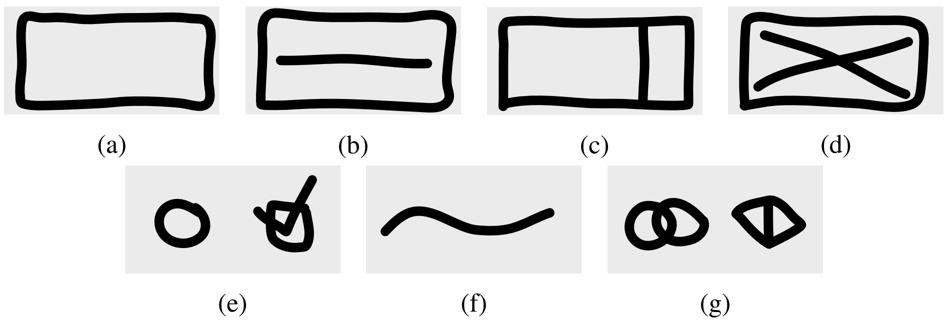


Figure 5.1: Pictographs used to represent different UI elements.

(a) Textfield. (b) Button. (c) Dropdown. (d) Image.
(e) Radio button and checkbox. (f) Text. (g) Annotation symbols.

5.2.1 Boundaries Calculation

This phase is responsible for the definition where the mockup will be drawn. The canvas, which comprises of the available area to draw, is created and predefined with a given width and height. To define the working area, it's dimensions are firstly calculated taking into consideration the canvas's dimensions. The working area is defined by an arbitrary area composed of cells, thus acting as a grid where the elements are placed. In order to calculate or generate this area, a cell size and cell gap variables are initialized. These values are the result given from a pre-defined size times a random value, therefore providing a little variation among different mockups.

The canvas is then divided horizontally and vertically using the grid size and gap. Since the canvas dimensions might not be multiple, there is some remaining value which the cells cannot fully fill. These leftovers are not used to define the working area.

To improve the mockup's representation variety, the aforementioned leftovers are used to calculate a random offset in each direction, and this offset is then used as a starting point for the working area (Figure 5.2b).

For better understanding, these operations are summarized by the calculations presented in Algorithm 1. With this process, given a blank canvas and grid properties, a working area is defined which will be used in the next phases to fill all the mockup's elements.

Algorithm 1 Mockup area initialization

```
1: $grid\_size \leftarrow default\_grid\_size * random(1.0, 1.5)$
2: $grid\_spacing \leftarrow default\_grid\_spacing * random(0.5, 1.0)$
3: $cell \leftarrow grid\_size + grid\_spacing$
4: $leftover_x \leftarrow canvas_{width} \bmod cell$
5: $leftover_y \leftarrow canvas_{height} \bmod cell$
6: $mockup_x \leftarrow leftover_x * random(0, 1)$
7: $mockup_y \leftarrow leftover_y * random(0, 1)$
8: $mockup_{width} \leftarrow canvas_{width} \bmod cell$
9: $mockup_{height} \leftarrow canvas_{height} \bmod cell$
```

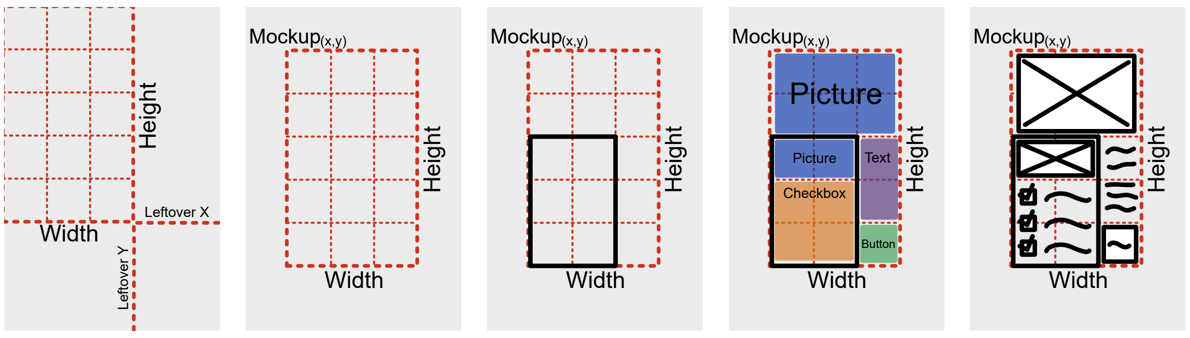


Figure 5.2: High-level process overview, executed in a sequential order.

(a) Mockup dimension and leftover calculation. (b) Mockup translation. (c) Container placement. (d) Element area definition. (e) Element placement.

5.2.2 Container Placement

With the working area defined, the next step of the algorithm is to assign cells to a set of containers. As mentioned before, containers are just aggregation of elements, so their area must be greater than a single cell. The algorithm developed takes the cells that compose the working area and uses them to displace a given container. These cells are also used in the next phase for the placement of elements using the same logic.

In order to place the containers in the grid, each cell is iterated through from top-left to bottom-right. The cells are first initialized empty, without any elements our grids related to them. The algorithm then proceeds to test each cell. If the cell is not assigned to a grid, then it is used as a candidate to start the grid expansion step. Starting from this cell, randomly calculated variables are assigned to the horizontally and vertically expansion of the future container. These variables are defined in the number of cells, which the container will occupy. The calculation of these values takes into account the mockup's boundaries, therefore if the container can occupy this area, it will not overtake the mockup's boundaries.

With the possible expansion dimensions calculated, starting from the current cell, the container boundaries will only coincide with this cell. The neighbor cells will then be explored for the existence of other containers. Each visited neighbor cell will update the current container boundary. And this process is repeated until (a) all neighbor cells are visited and thus the final container is added to the final list of containers or (b) a nearby container is found, consequently, the algorithm stops and the last value used as container's boundary is used. Finally, the container's boundary is evaluated and discarded if the area does not meet a certain criteria. In the current implementation of the algorithm, the area must be greater than two cells.

After the execution of this algorithm, the cells are updated with each indicating the container associated with them and a list of areas representing the containers. This algorithm's pseudo-code can be consulted in Algorithm 2. The result of this phase is represented by Figure 5.2c.

Algorithm 2 Container expansion algorithm

```
Input: $Cell_{1,1} \dots Cell_{N,M}$
Output: $Containers$ list of containers
 1: function CONTAINER PLACEMENT($Cells$)
 2:     $Containers \leftarrow []$
 3:     $Grid \leftarrow 0$
 4:     $N \leftarrow length_x(Cells)$
 5:     $M \leftarrow length_y(Cells)$
 6:     for $j \leftarrow 1$ to $M$ do
 7:         for $i \leftarrow 1$ to $N$ do
 8:             $Cell \leftarrow Cell_{i,j}$
 9:             if $Cell$ belongs to container then
10:                 continue
11:             end if
12:             $Expansion_H \leftarrow random(i,N) - i$
13:             $Expansion_V \leftarrow random(j,M) - j$
14:             if $Expansion$ is not enough then
15:                 continue
16:             end if
17:             $StartingBounds \leftarrow (i,j)$
18:             $EndingBounds \leftarrow StartingBounds$
19:             for $Cell_{Neighbour}$ in $Expansion$ do
20:                 if $Cell_{Neighbour}$ belongs to container then
21:                     break
22:                 end if
23:                 $EndingBounds \leftarrow$Position($Cell_Neighbour$)
24:             end for
25:             SetCellsGrid($StartingBounds$, $EndingBounds$, $Grid$)
26:             $Grid \leftarrow Grid + 1$
27:             $Containers$.append([$StartingBounds, EndingBounds$])
28:         end for
29:     end for
30:     return Containers
31: end function
```

5.2.3 Element Area Definition

This phase is responsible for the definition of areas which represent a kind of element. This process is similar to the one followed by the expansion of containers. The main difference is that information about the cell's grid is taken into account. That is to say, that when expanding from a cell, only cells assigned to the same container are taken into consideration.

Using the cells resulting from the previous phase, they identify the container associated with them. By iterating for each cell, in the same direction as before (from top-left to bottom-right), a check is made to verify if the cell is empty. Then starting at this cell, similarly to the aforementioned phase, horizontal and vertical expansion values are calculated. By verifying the neighbor area, each neighbor is checked if the cell is empty and the container is the same as the starting cell. This step repeats until (a) all neighbor cells inside the expansion area are successfully evaluated or (b) a collision is made, thus implying the cell is not empty or the container is different. Likewise, the algorithm stops and the previous boundary is used as the element's boundary. This boundary and the element's type (selected at random) is then added to a list of elements.

After this step, all the cells inside a grid are covered by areas, each one representing a different UI element. These areas are then used in the next phase to actually be filled with the elements by taking into account a set of rules. The result from this phase is depicted in picture 5.2d.

5.2.4 Element Placement

The main purpose of this phase is to fill the areas defined before with the actual UI elements. Each one of these areas represents a single kind of element. In order to fill the areas, a set of parameters were used in order to have more control over the generated elements. These parameters control the fill behavior, they specify the (1) fill direction: if the element added will fill the area in a given direction, (2) split orientation: to determine if multiple elements are added in sequence in a given direction (3), target height: to define the element's height, when the split is horizontal, and (4) addition of text over the unoccupied space. Table 5.1 contains the rules used for each element.

In the case of pictures, the process is straight forward, only one element is added and it occupies the whole area. But in case of small elements such as checkboxes, they are often small and their shape is limited, therefore not filling the whole area. So the target area is split vertically into rows, with a height of random size. Since this area's height may not be a multiple of row's height, the final row that is not complete is used to offset the other rows. Similarly to the process of boundaries calculation. These rows are then filled with the actual element. In the case of buttons, the work is done, however, a checkbox usually does not fill the row's width. Their aspect ratio is close to 1:1, so a remaining blank space is left to be filled. To enrich the mockup, in cases where this occurs, it is possible to fill the remaining space with a simple text element. This results in a more natural representation of lists, where the checkbox is tied to a textual description. The figure 5.2e depicts the results of element expansion, namely in the areas belonging to text and checkbox.

With this phase, the elements are in place and ready to be drawn in the next phase.

Table 5.1: Example of element's fill parameters.

Element	Expansion	Split	Height	Add Text
Picture	all	none	none	false
Radio Button	vertical	vertical	[40;50]	true
Checkbox	vertical	vertical	[40;50]	true
Dropdown	horizontal	vertical	[40;70]	false
Text field	horizontal	vertical	[40;70]	false
Text block	all	vertical	[40;70]	false
Button	all	vertical	[40;70]	false

5.2.5 Elements Drawing Process

Thus far, the grid is composed of all the elements and containers calculated in the previous phases. If the elements were to be drawn as they are currently defined, it would result in a perfectly drawn wireframe, with straight lines which are are not the norm of hand-drawn mockups. In order to add more variety to the final drawings, the process responsible for the definition of the points, that compose the element's shape, must be subject to some adjustments. With the intent to make the mockups closer to the non-synthetic variant.

The adjustments were separated into two distinct types. First, the points that define the final shape, and are used to create lines, are subject to displacement. This process simply shifts each point by a random value. This might result in a skew effect and the minimization of parallel lines in the final result, thus attempting to mimic inaccuracies found in a hand-drawn image. Another side effect is a slight rotation of the element. Figure 5.3 explains in a visual way the aforementioned process.

After the points are translated, the secound adjustment is applied when drawing the lines. Each line is defined by two points and when drawn only using them, the result are still straight lines. The drawing method then must have the ability to receive a straight line and making it non-uniform. To that end, after receiving a line, while maintaining the start and ending points in their original position, more points are added along the line and displaced randomly. These displacements create small distortions in the final drawing.

When this method is coupled with RougJS library, it is possible to create distinct drawing styles, by defining different brushes to simulate different kinds of writing instruments. Another option is the addition of a custom font to replace the text representation with some characters, as an attempt to mimic handwritten information.

5.3 Pipeline

The generation of prototypes using hand-drawn mockups can be achieved by a large number of techniques. Inspired by the strategies used by object detectors, a small pipeline was developed.

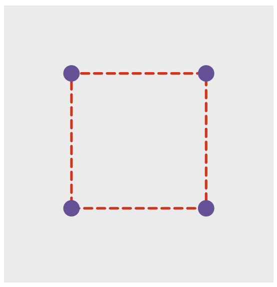
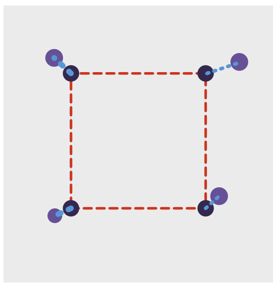
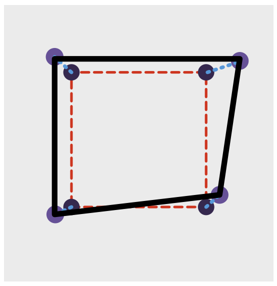
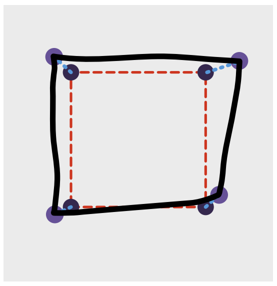
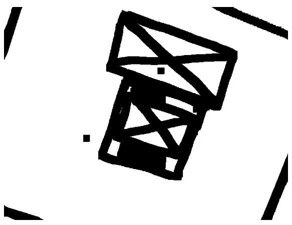
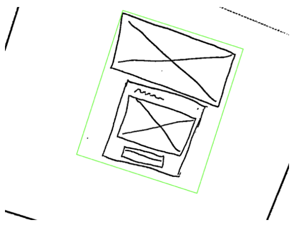
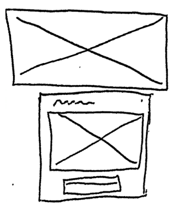
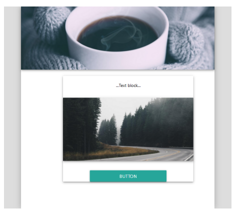

(a)

(b)

(c)

(d)

Figure 5.3: Point Displacement Process.

(a) Original Point Position. (b) Point random displacement. (c) Resulting line. (d) Line with randomness.


(a)

(b)

(c)


(d)

Figure 5.4: Image Acquisition Process and Final Result executed in sequential order.

(a) The resulting image from the adaptive threshold and morphological operations. (b) Small area rectangle detection. (c) Mockup cropped and rotation corrected. (d) Generated website.

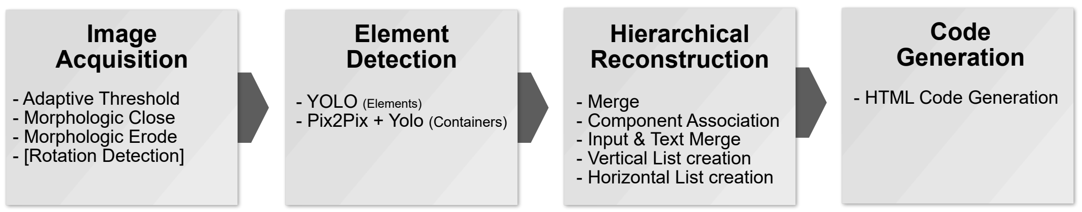


Figure 5.5: Pipeline processing steps executed sequentially.

This approach proposes a pipeline, organized in four main phases, which can be modified independently thus facilitating the development and experiment of different modules, as well as providing more control over the intermediate steps or generated outputs. This section describes the main steps executed sequentially by the pipeline. Each subsection describes a step of the pipeline, describing the expected inputs, operations applied and outputs.

Subsection 5.3.1 describes how image acquisition and pre-processing is applied. It details the reasons for, and operations of this step. Subsection 5.3.2 describes the objective and modules behind the detection phase. This section is responsible for the extraction of information from images. It also proposes an approach to extract and segment containers from a mockup. Subsection 5.3.3 represents the sequence of operations used to build a hierarchy, by using the elements from the detection phase. Subsection 5.3.4 contains information about the generation of code. The use of CSS-Grid functionality that enables developers to create layouts that are not possible using linear layouts.

Figure 5.5 contains a high-level overview of the developed pipeline. Appendix B presents examples of results obtained using the developed tool.

5.3.1 Image Acquisition

Image acquisition from the real world is not trivial, there are multiple factors that can influence the final result. These factors can be labeled as (a) internal factors: that mainly depends on the quality/positioning of the sensor or the lenses used, and (b) external factors: that can be the illumination, weather, among others. This phase is responsible to filter such anomalies, such as noise. By using a set of image operations, transform an acquired image to a high contrast version. The reasoning behind this transformation is due to the way the models, used in the detection phase, where trained. These models were trained using high contrast images, therefore this step is the key to delivering better performance in the end.

To convert a given image to its high contrast version, first, it needs to be converted to black and white, thus working with only one channel, representing the image's intensity levels. Afterward, an adaptive threshold algorithm is applied, this threshold creates high contrast in areas defined by high variance, therefore defining the mockup's lines even when the illumination is irregular. A high contrast image is a result of this operation, but some noise may still appear. Two morphological operations are then applied to remediate this. The structuring element is defined as an ellipse, and the closing and erosion of the operation are applied sequentially. Since each image capture device can produce different results, in order to finetune the final result, the parameters for the adaptive threshold and morphological operations are configurable during run-time.

Another optional step is the detection of the mockup. By considering that the whole mockup is in the image and that it occupies the larger area, all the lines of the high-contrast image are expanded, to the point they overlap with each other. With these lines expanded, then from the search for binary large objects, the largest is selected and the minimum area rectangle is retrieved. By assuming the mockups have a height higher than its width value, the mockup is extracted by cropping and rotating the original image, using this rectangle. Image 5.4 contains the operations applied along with the final generated result.

As mentioned before, this step is required due to how the DNN's where trained. A low quantity of real-world data was available which constrained the final results. With more information available, this step could be ignored.

5.3.2 Element Detection

The detection phase is the most fundamental phase of the system. The main goal is by interpreting the image, provided by the image acquisition phase, output the elements or a subset of elements that compose the mockup. In this phase, each module receives the same image and provides a JSON output of each element found with their corresponding bounding boxes. Two modules compose this phase, being: (1) element detection module, that is responsible for outputting the atomic elements and (2) a container detection module, which extracts boxes that aggregate different atomic elements.

Regarding the element detection module, the model used was a Keras implementation of YOLO [2]. With YOLO, by providing an image as an input it then outputs all the detected elements and their relative information. This data was then translated to a JSON format, comprising of all the elements detected along with their respective bounding boxes.

The module responsible for the extraction of containers used a novel approach, by employing the use of Pix2Pix [IZZE16] and YOLO sequentially. Usually, hand-drawn mockups contain imperfections, (1) lines not connected, (2) ambiguous representations, (3) no straight lines. The use of traditional computer vision techniques, while working well on digital mockups (that are consistent), in hand-drawn environments (a) a higher number of verifications and decisions must be made in order to isolate these containers, or (b) the drawing process must take into account those limitations. The idea behind this approach was to segment only the containers using Pix2Pix (Figure 5.6). The model receives an image, in this case, the mockup and outputs another image, which is the containers. This model is based on conditional adversarial networks and provides a general-purpose solution for this kind of segmentation. Similarly, the output from the YOLO network is then translated to JSON and given as result from this module.

The results provided by these modules are concatenated, this originates a list including all containers and elements with their respective bounding boxes and without any children.

[2]https://github.com/qqwweee/keras-yolo3

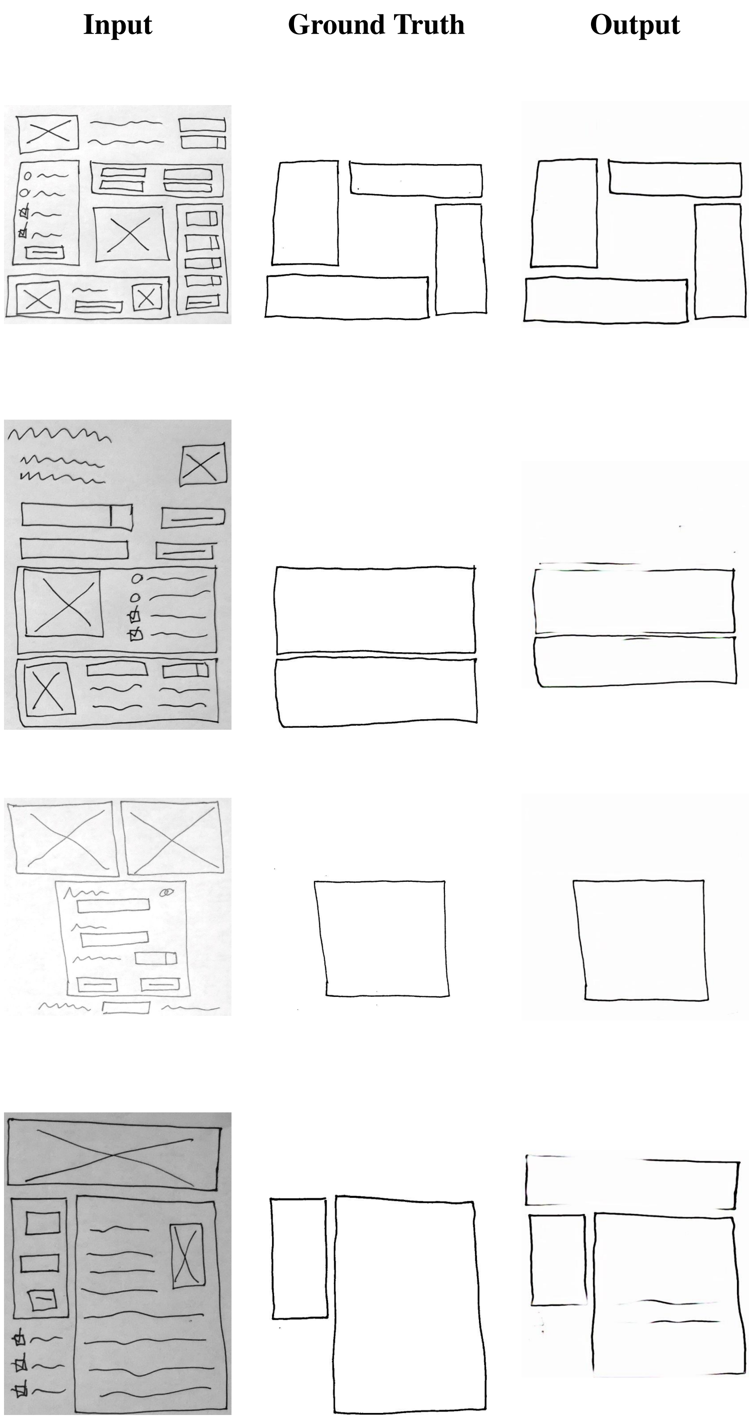


Figure 5.6: Container segmentation, produced by Pix2pix.

5.3.3 Hierarchical Reconstruction

From the Element Detection Phase (Section 5.3.2), a list of elements is received as input for this phase. This list does not have any hierarchical complexity, only the raw elements, and containers that were identified. This step was divided into multiple modules, they interact with each other similarly to the pipeline design pattern, where the output from one module is fed as an input to the next. Therefore, each module is executed in sequence, and their execution order may impact the final result.

The first module has the most basic role, by having as input a list of elements and containers, match the elements to check in which container they belong. This aggregates all the elements that belong to a container and set them as its children. The check is made linearly, by evaluating the overlap between the areas of the element and container respectively. The overlap is measured as a percentage, if the element totally fits the container, or has some space outside. In case a certain threshold is met, then it is added as a child. Lastly, all the remaining elements and containers, are added to a new container, representing the root of the document its bounding box is adjusted to match the minimum and maximum of their children.

With a simple hierarchy in place, the next module executed aims to join the elements, that this work classified as annotations, with their respective containers. While iterating over the containers, if a child is considered an annotation a new field is added to the container to store a list of annotations. In the current implementation, this field is called components. The intention to use annotations was to provide interaction during the drawing process, with them it is possible to control the generated prototype by adding a tag to the current container. Currently, they are only used to modify the style of a target container, but its use can be adapted to modify the behavior instead.

When using checkboxes or radio buttons, they are commonly placed near text descriptions. This module aims to aggregate these elements with nearby text by using simple overlapping techniques. For each container, the checkboxes and radio buttons are iterated and tested against the lines of detected text. The overlapping test uses the bounding box of the checkboxes/radio buttons and stretches them horizontally, the evaluation is them made, in case it overlaps with a textbox, a check is made to make sure no more elements are between them. Finally, in case it goes well, a horizontal container is created to aggregate those two elements.

It is usual for the websites to present multiple elements of the same type near each other, with this in mind the next module is responsible to identify lists of elements of the same kind. This module is re-used two times, in order to identify vertical and then horizontal lists. Assuming that the module is identifying vertical lists, by iterating through all the elements inside a container, each element is tested with elements of its kind. Similarly to the module above, the selected element is expanded vertically, and a check is made to verify if any other element overlaps with it. In case of an overlap, the algorithm then evaluates if any other object is between these elements. This is done by creating a bounding box which encapsulates both elements. In case no other element collides, it is added to a list containing the matching elements. Finally, a vertical container is created to

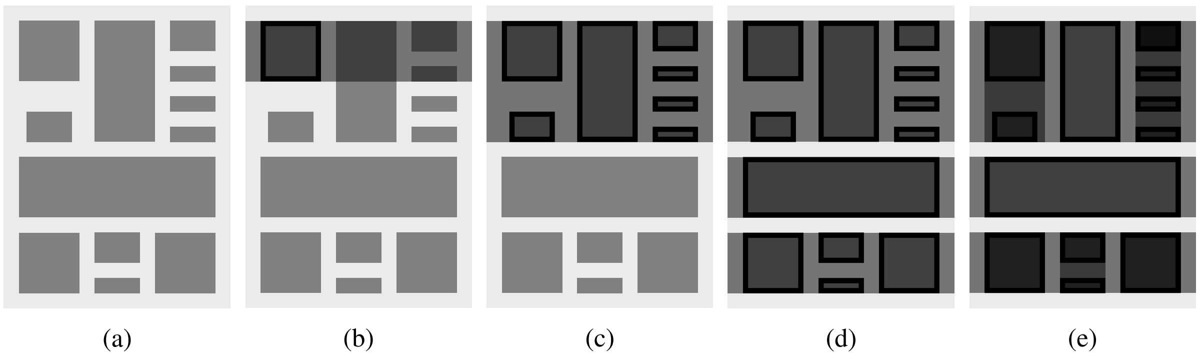


Figure 5.7: Hierarchy Generation Algorithm.

(a) Original hierarchy. (b) First element selected and Testing Box expanded horizontally. (c) Testing Box expanded to Accommodate all elements. (d) Process repeated for the remaining elements. (e) Process applied to the generated sub-containers vertically.

store these elements, with a bounding box size which encapsulates the new children. This module is then executed targeting horizontal lists.

The last module is responsible for the aggregation of the elements into lists, to generate a more concise and aligned hierarchy. This module proposes a Hierarchy Reconstruction Algorithm. The main principle followed by this algorithm is that: given a container, its contents can be divided into horizontal or vertical layouts. Starting at the container that represents the root, the algorithm begins to separate vertically or horizontally the elements that compose the container. When evaluating a given orientation, the container's children are sorted accordingly. For illustration purposes, let's assume the expansion is vertical, as depicted by Figure 5.7. With all the elements sorted vertically, the first is selected and a *Testing Box* is created. Since the expansion is vertical, the created box will define its bounds by using the first element to set the vertical dimensions and the container for the horizontal dimensions. In other words, this box is expanded horizontally in order to occupy all the container's width and with the vertical size and position of the first element (Figure 5.7b). This box is then used to detect which elements are nearby. To put in another way, it is used to detect which elements are horizontally aligned with the first element. After the creation of the *Testing Box*, the remaining children are then tested, if they overlap with the box, then the box's bounds are updated in order o encapsulate the newly inserted object. The algorithm repeats until (a) an element do not overlap (Figure 5.7c), (b) all elements were tested. The elements that were collected during this process are then added to a list, in case only one element was detected, it is added to the final list as it is, but if multiple elements were collected, then a new container is created encapsulating this list as its children. The algorithm then repeats until all elements are added to their own list (Figure 5.7d). This algorithm outputs a new list of elements and can happen in two different cases. (a) The list has only one element, this fact informs that the elements collide with each other in the given expansion, making it impossible to separate them, so the algorithm must be executed again with an opposite direction. In case both orientations return only one element, then this container's orientation is marked as a grid and the execution continues normally. (b) The list has more than one element, in this case, the algorithm was able to separate the elements and they are added as the new children of the target container, the container orientation is them properly assigned. The algorithm then proceeds to be applied to each child container but using the opposite direction (Figure 5.7e). This phase returns a hierarchy composed of containers that define three types of orientations: horizontal, vertical, or grid.

5.3.4 Code Generation

From the previous phase, a hierarchy is given as input for this phase. This hierarchy is then sent to the available generators, to allow for the code generation for multiple platforms. The current implementation only contains a simple HTML generator. This generator uses an HTML CSS and Javascript template, and then fills the template HTML with the hierarchy received. The generation starts at the root container and explores the hierarchy using a depth-first search approach. For each item inside the list, the respective HTML code is created. In case of a container that was labeled with annotations, the CSS tags associated with them are applied to the container, thus affecting the final style. The containers that are carry a horizontal or vertical orientation in the final HTML are represented as a flex layout, with their childen written in sequence. In case of a grid orientation, the container is marked to use CSS-Grid. In this mode, each of the children is snapped into place. To achieve this, the default implementation divides each container into 8 cells horizontally and calculates the vertical cells. Each element then is snapped into place by using the integer result from the division between the cell dimension and the bounding box. In the horizontal direction when the starting position is snapped it will have a value between [1;8]. The CSS attributes *grid-column* and *grid-row* are then initialized with these values for each element inside a grid. The generated code is then stored and the result is displayed in a browser window.

5.4 Conclusions

In this chapter, the underlying decisions, and details regarding the implementation of two distinct systems were introduced. A synthetic generator of hand-drawn datasets was proposed. It was also introduced all the techniques used in order to give an hand-drawn feel to the final result. This implementation leads to the ability for generating images with their corresponding annotations and segmentation masks. The implementation of a pipeline was also discussed. Where it was divided into four main phases, including the image acquisition step, element detection, hierarchical reconstruction, and code generation. The proposal of a hierarchy reconstruction algorithm and a container detection technique resulted from this implementation, along with the proposal of annotations that could be used in order to control the prototype generation.

Chapter 6

Validation

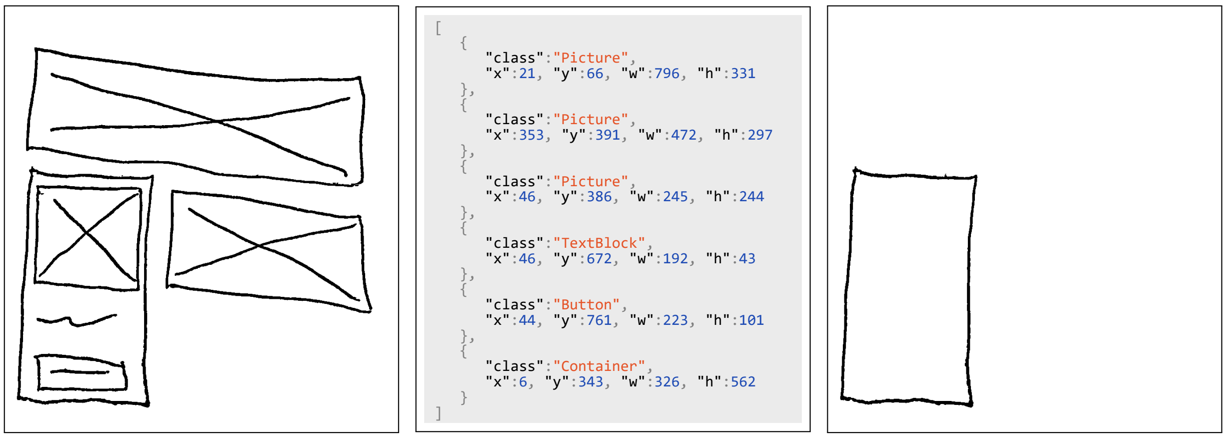
The act of reducing the prototype development cycle has its own challenges, and depending on the solution broad range o challenges emerge that need to be answered. Until now this document introduced the main context of prototype development, the problem's main motivations and challenges. The necessary background was then introduced, including contents related to computer vision techniques and Deep learning. Then, the state of the art solutions related to the generation of prototypes from an image medium were presented and categorized by approach. This document then overviewed over the current state of the art implementations and its challenges, concluding with a suggested solution and the main contributions this document aims to achieve. The implementation steps regarding a synthetic dataset generation algorithm targeting mockups as well as the suggestion of an alternative methodology to generate prototypes from hand-drawn mockups were also described.

Section 6.1 clarifies over the number of datasets, their conception and how they are organized. Section 6.2 introduces the evaluation process used to evaluate the synthetic generator and the methodologies developed regarding element segmentation and detection. Section 6.3 introduces and provides a judgment over the obtained results. Section 6.4 describes the validation threats that may affect the presented results. Section 6.5 reflects over the results achieved and the evaluation methodology.

6.1 Datasets Used

With the development of synthetic mockup generator, it is important to evaluate how reliable this generator can be, what *bias* can eventually be transferred toward the developed networks, and how it can impact the final results.

In order to measure this impact, two datasets were used. The structure of each dataset remains the same, they are organized in three main collections. Given the models used for the development of this project, it is mandatory for the dataset to contain (a) the original mockup image, (b) its



```
[
    {
        "class":"Picture",
        "x":21, "y":66, "w":796, "h":331
    },
    {
        "class":"Picture",
        "x":353, "y":391, "w":472, "h":297
    },
    {
        "class":"Picture",
        "x":46, "y":386, "w":245, "h":244
    },
    {
        "class":"TextBlock",
        "x":46, "y":672, "w":192, "h":43
    },
    {
        "class":"Button",
        "x":44, "y":761, "w":223, "h":101
    },
    {
        "class":"Container",
        "x":6, "y":343, "w":326, "h":562
    }
]
```

(a) Mockup (b) Annotation (c) Containers

Figure 6.1: Example from hand-drawn dataset.

annotation file, in the JSON format and with all the elements properly labeled, and finally (c) a segmented image which must only represent the containers. Figure 6.1 contains an example for a given mockup from the hand-drawn dataset.The first dataset consists entirely of mockups generated by MockGen, the total number of examples are 12000, split into 8400 and 3600 for training and validation sets respectively. The second dataset is composed of hand-drawn mockups, this dataset represents a case where the number of data available is in lower quantity. With this in mind, this dataset contains a total of 107 examples, which were split into 74 training and 33 validation examples.

The process of creation of the hand-drawn dataset was done by sketching multiple examples in blank papers, keeping in mind the visual syntax used for each element. Then, after moving these drawings to a digital format, each image, that was composed of multiple mockups, was divided using an image editing software. The same software was also used to create the containers-only version for each image, to achieve this, each atomic element was masked manually, only leaving the containers intact. The final versions were then annotated using Visual Object Tagging Tool (VoTT) [1], an open source annotation tool commonly used to annotate datasets for object detector models. The files generated by VoTT were then modified to a simplified JSON format. Due to the fact that MockGen generates high contrast images without any sort of noise, the hand-drawn dataset went through one final processing step. This step was processed using the same technique used in the image acquisition phase (Section 5.3.1). The adaptive threshold algorithm was applied to the intensity levels of the image, and, by using a structuring element in shape of an ellipse, two morphological operations were applied in sequence, namely, an opening and an erosion. Figure 6.2 shows an example of these operations applied to an original image.

[1]https://github.com/microsoft/VoTT

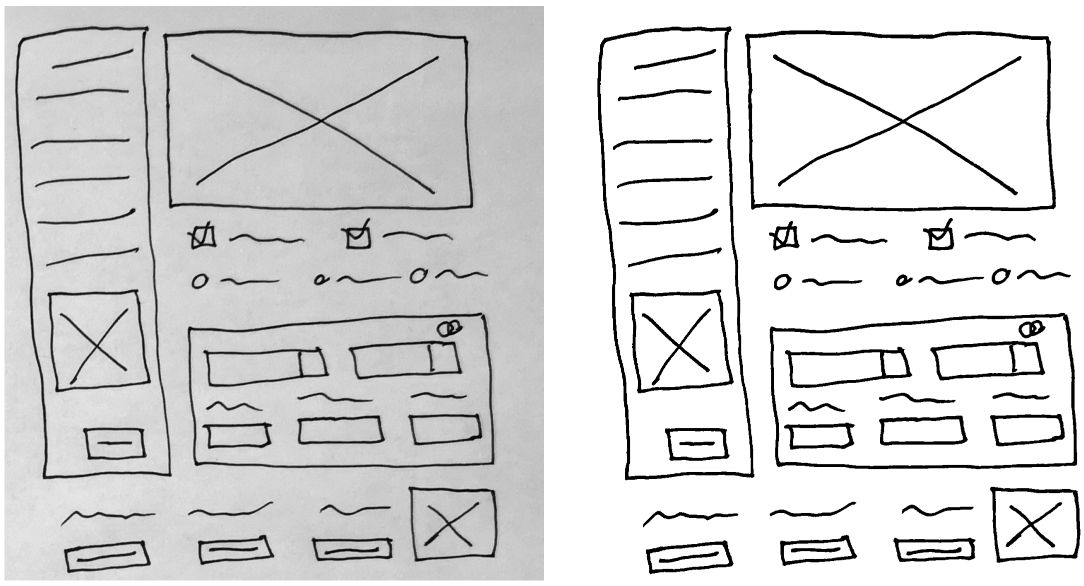

(a) Original (b) Treated

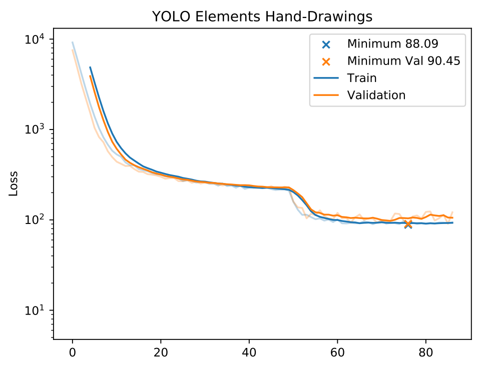
Figure 6.2: Hand-drawn image processed using adaptive threshold and morphological operations.

6.2 Validation Process

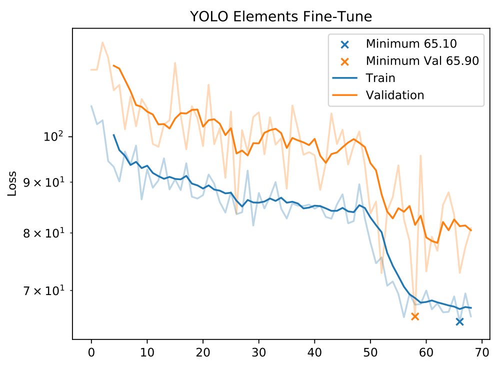
The validation process was responsible to evaluate two distinct tools that are tied in such a way that the results from one can be used to measure the impact of the other. One of those tools is the synthetic generator of mockups. The validation process must be able to tell if the use of such a generator can be made useful in environments where it is used to train neural networks capable of identifying the elements form a mockup. The pipeline developed, to generate prototypes, uses such networks and thus the performance evaluation from this tool will reflect the impact of the mockup generator (MockGen).

In light of the aforementioned relationship, each model contains three distinct versions, each one representing a possible scenario. (1) The model was trained only using the real hand-drawn dataset, this model represents a case where only a low quantity of data is available. (2) The model was trained using the synthetic generated dataset, it represents a synthetic only approach, and it is used to compare with the first model in order to assess the possibility of using such a model. (3) The model that was trained using the synthetic dataset is fine-tuned, using the hand-drawn dataset. As mentioned before two main modules were used during the detection phase, element and container detection modules.

Regarding the element detection module, it is responsible for the extraction of atomic elements such as (a) buttons, (b) text blocks, (c) figures. This extraction was performed using YOLO, an object detection model with an emphasis in speed while having an overall respected performance. Three different versions of this model were trained, as mentioned before. The models used the original YOLO weights as the starting point, The versions that employed the use of the hand-drawn dataset were trained in two steps, (1) the first used 50 epochs while the first layers remained



(a) Hand-Drawn Data



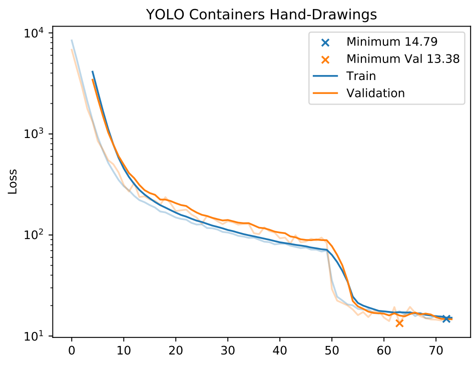
(b) Fine-Tune

Figure 6.3: Element extraction model's training loss.

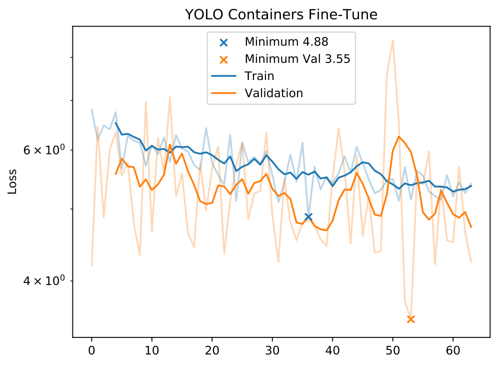
frozen and using a batch size of 16 and (2) 50 epochs of fine-tuning with the model unfrozen using a batch size of 2. The version of the model that used the MockGen dataset adopted a similar strategy but during multiple iterations. Initially, the model was trained with the same batch-sizes as before, but with a smaller epoch number, of 25 for each step. This process was repeated over 100 epochs. Due to the large time the model took to process all this information, a cloud computing service was used to speed up this process. This service enabled to increase the number of epochs to 50 for each step and for the first step, which has the layers frozen, the batch-size was increased to 128. For the second step where the layers are unfrozen, the batch size was set to 16. The total amount of epochs to train the model were about 400. The loss values for the element detection model can be consulted Figure 6.3.

The container detection model used two models executed sequentially. The two models were Pix2Pix and YOLO. The three versions of the Pix2Pix model were trained during 100 epochs with a batch-size of 2, due to memory constraints. The result of this model is then processed by YOLO, similarly, this model has three different versions. Each one was trained using the same two-step approach, (1) 50 epochs with the first layers frozen and a batch size of 16 and (2) 50 epochs with the model unfrozen and a batch size of 2. This module was evaluated as a whole, thus reflecting the interactions and final results from these models. The loss values for the container detection models can be consulted Figure 6.4. In order to validate the developed pipeline and the impact of the synthetic datasets, the detection phase was prioritized. Therefore it was evaluated against the testing set consisting of 33 hand-drawn mockups (Figure 6.5). The metrics used to target the evaluation of the performance of object detectors are the mean average precision (mAP) and the log-average miss rate [RHGS17, RDGF16, GDDM14, SSSJ19, RASC14, WSPD11, ZZXW18]. The detection results including the true and false positives were also calculated.

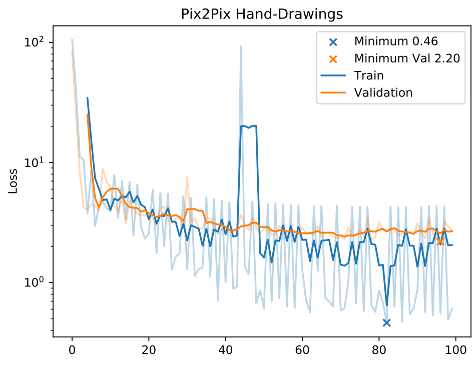
The mAP is calculated in the following way, for each class it is plotted the precision and recall curve, then the area is calculated by simplifying the peaks of the curve. That is because of the zigzag pattern that may emerge from these charts, the curve is smoothed by using the maximum precision value to the right of the current value, which results in a step-like a chart. The mean of



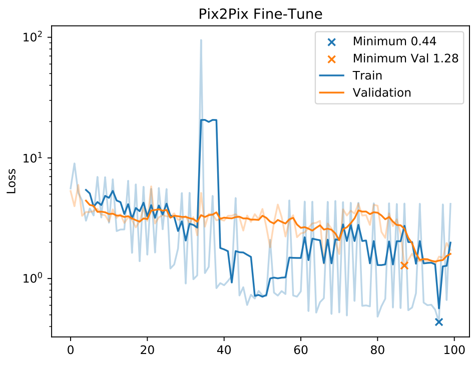
(a) YOLO Hand-Drawn Data



(b) YOLO Fine-Tune



(c) Pix2Pix Hand-Drawn Data



(d) Pix2Pix Fine-Tune

Figure 6.4: Container extraction model's training loss.

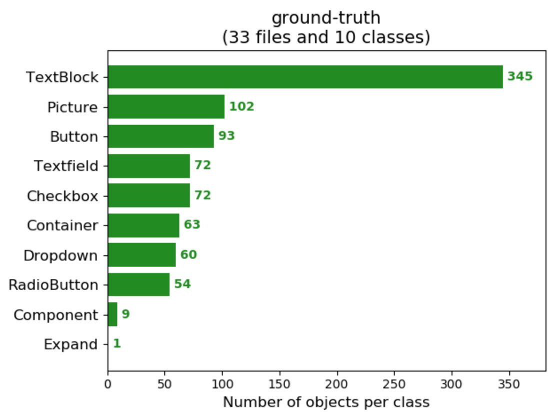


Figure 6.5: Element Distribution Ground Truth.

the areas representing each class is then calculated to obtain the final mAP value.

The log-average miss rate is defined by the averaging of the miss rates (Equation 6.1). The miss rates are calculated at nine false positives per image (FPPI) rates, these rates are obtained by varying the threshold on the detection's confidence, and are evenly spaced in log-space $[10^{-2}, 10^{0}]$ [WSPD11].

$$MR = \frac{FN}{TP+FN} \tag{6.1}$$

Execution times of the system were also evaluated to evaluate the impact of each phase. Therefore identifying possible constraints and solutions to contribute to the enhancement of the overall system.

6.3 Results

This section exhibits and reflects about results obtained from the evaluation of the system. The results are related to the element and container detection performances as well as the overall performance. This section also aims to assess the impact of the synthetic dataset in the performances of the models used.

6.3.1 Element Detection Performance

The version that used MockGen as the training data had a bias towards the buttons and text blocks. After the validation of the model against the real hand-drawn dataset, even though the mAP was similar (60% and 59%), a bias towards buttons and texts is evident, this can be observed in Table 6.1b This fact is also supported by the log-average miss rate, which also reflects this bias due to its large values, but it is more evident by the number of false positives. The latter is higher than when compared with the results of the detection from the model that used hand-drawn data. This bias may be attributed to the size distribution present in the synthetic dataset of elements

Table 6.1: Result comparison of Element Detection phase.

(a) Network trained using hand-drawn dataset

Model	Ground-truth	False Positives	True Positives	mAP (59.7%)	log-average miss rate
TextBlock	345	24	308	86	0.48
Picture	102	6	88	86	0.20
Button	93	10	80	83	0.34
Textfield	72	21	57	69	0.57
Checkbox	72	8	36	43	0.78
Dropdown	60	13	42	65	0.49
RadioButton	54	6	37	66	0.50
Component	9	6	4	86	0.20
Expand	1	0	0	0	0.00

(b) Network trained using only MockGen

Model	Ground-truth	False Positives	True Positives	mAP (60.0%)	log-average miss rate
TextBlock	345	42	231	61	0.78
Picture	102	2	84	82	0.19
Button	93	76	88	61	0.83
Textfield	72	0	15	21	0.79
Checkbox	72	12	58	68	0.63
Dropdown	60	0	28	47	0.53
RadioButton	54	0	42	78	0.22
Component	9	0	2	22	0.78
Expand	1	1	1	100	0.00

(c) Network trained with fine-tune

Model	Ground-truth	False Positives	True Positives	mAP (95.3%)	log-average miss rate
TextBlock	345	3	334	97	0.06
Picture	102	0	99	97	0.03
Button	93	2	92	98	0.05
Textfield	72	1	70	97	0.03
Checkbox	72	3	64	88	0.20
Dropdown	60	2	58	97	0.04
RadioButton	54	0	53	98	0.02
Component	9	1	8	86	0.20
Expand	1	0	1	100	0.00

Table 6.2: Result comparison of Container Detection phase.

Model	**False Positives**	**True Positives**	**mAP (%)**	**log-average miss rate**
Hand-Drawn	**9**	**56**	**87.78**	**0.19**
Synthetic	245	30	20.88	0.88
Fine-Tune	13	56	87.14	0.22

similar to the buttons. By checking the log-average miss rate and the number of false positives one can conclude that multiple text fields were miss-classified as buttons.

The fine-tune version, that used hand-drawn data tho train the Model version that used the MockGen dataset, had a high-performance impact. Table 6.1c represents the model obtained from Table 6.1b fine-tuned using the hand-drawn Dataset. The results show high mAP of 95% and lower log-average miss rates. Consequently, the number of false positives is lower when compared with the other two versions of the model. It also contributed to the total number of true positives, meaning that even more elements are detected.

6.3.2 Container Detection Performance

The use of MockGen did not generalized well for hand-drawn data. The model that was trained using the dataset produced by MockGen had a negative impact when evaluated against the hand-drawn test dataset, achieving only an mAP of 20% (Table 6.2). Therefore, affecting the results produced after being fine-tuned. The main reason behind this might coincide with the fact that MockGen containers are restricted and constrained to a more rectangular shape. The mockups generated by MockGen has also all the cells filled with shapes wherein the hand-drawn dataset some containers have empty areas around them.

The segmentation technique had reasonable values for element detection. The technique used by mixing Pix2Pix and YOLO seems to have a good detection mAP of 87% when the models used only the hand-drawn data. Keeping in mind that they were trained independently of each other, the results were higher than expected. The alternative technique used during the development of this project was the use of a U-Net instead, but due to the thin lines that formed the containers, that network could not generalize well and the results were not feasible (no container was segmented). While not perfect and restricted to the dataset size, the use of Pix2Pix architecture improved greatly the final results.

6.3.3 Overall Performance

The best models' versions were used for the evaluation of the overall pipeline performance. These models are, (1) element detection using the fine-tune version, (2) container detection using the version trained with the hand-drawn dataset. The results can be consulted Table 6.3.

Overall High Element Detection. Scoring an mAP of 94% as detection performance. The log-average miss rate is still a little high for the containers, checkbox and component elements.

Table 6.3: Overall detection results.

Model	**Ground-truth**	**False Positives**	**True Positives**	**mAP (94.6%)**	**log-average miss rate**
TextBlock	345	3	334	97	0.06
Picture	102	0	99	97	0.03
Button	93	2	92	98	0.05
Textfield	72	1	70	97	0.03
Checkbox	72	3	64	88	0.20
Container	63	9	56	88	0.19
Dropdown	60	2	58	97	0.04
RadioButton	54	0	53	98	0.02
Component	9	1	8	86	0.20
Expand	1	0	1	100	0.00

Table 6.4: Execution times for 107 mockups.

Phase		**Time**
Detection		0.4901 s
	Elements	0.1449 s
	Containers	0.3452 s
Processing		0.0036 s
Generation		0.0064 s
Total		**0.5001 s**

The component element had a lower distribution when compared to others, so a slower performance on this side is expected. Even though the values around 20% may be tolerable for other applications, when detecting mockups, it is useful to extract these elements with accuracy. Therefore, when a frequent item that is present in lists, such as checkboxes, is missing, that can make the final prototype look a little bit off.

Container detection is the most inaccurate phase. While having a high mAP value, as mentioned before, the number of false positives impacts the final results. Since the generated prototypes are easily influenced by the containers. During the pre-processing step, the empty containers are removed, contributing to some improvements over the final results, but it is not capable of solving all the miss classifications. The issues could possibly be attenuated using more examples and further training.

The detection phase had the higher execution time. As expected the time it took to identify the elements was the highest, with the Pix2Pix model consuming most of the time. Even though the average execution time was around 0.5 seconds (Table 6.4), it is possible for the system to be executed in near real time, depending on the hardware available. Therefore it enables to take multiple captures to test different perspectives to adjust the final result.

The generation is affected by the image's orientation. While an algorithm was used, to identify the mockup and adjust its position, when it fails, or in cases where it is not used, the output can be different. This is mainly due to the nature of the algorithms applied. When acquiring a new image, if the mockup is not properly aligned, the detection phase does not account for the rotation. Therefore, this phase always outputs bounding boxes aligned to the image and not to the mockup itself.

6.4 Validation Threats

There are some validation threats present, due to the evaluation environment used. The tests were done using a small dataset, hand-drawn and developed by one person, which means the drawing style and handedness remained the same. Therefore the results are constrained to the nature of the dataset. The complexity of the mockups can also be questionable, while there was an attempt to produce examples with multiple elements, their distribution varies among them. The system always applies a pre-processing step in order to convert the image in into a high contrast version, that way the drawings used blank paper, and the results may vary depending on the texture of the surface used. The generated examples had multiple types of stroke sizes, in contrast, for the hand-drawn dataset the instrument used remained the same for every mockup.

6.5 Conclusions

This section introduced the validation process used to evaluate the synthetic dataset generator and the proposed segmentation and detection technique. In this section, an overview of the datasets used was made, including their conception and the main information necessary for the evaluation of both systems. The evaluation process along with the respective results were then introduced. A synthetic dataset generator showed to be useful on the atomic element detection task. Using some modifications to the generator in order to augment the data, it is expected for these results to increase. The use of the generator for the segmentation of containers proved to not be worth, mainly due to the regularity of the containers produced versus the containers produced by a human. With this information, future implementations should take this regularity into account in order to improve the generation steps. The use of two sequentially executed networks for the segmentation of container yield inaccurate results. More so, when the images acquired had irregular perspectives than the ones used. It is also important to keep in mind that this performance reflects the use of a low quantity of information.

Chapter 7

Conclusions and Future Work

One way of developing prototypes is by making use of the prototype development cycle, where a mockup is used by designers and clients to convey ideas, it is then forwarded to a development team which is responsible to implement the graphical interface, resulting in a prototype which can then be evaluated by the client. The repetition of this cycle can be resource and time-consuming. Providing immediate feedback at the conception of the mockup hopefully would decrease the number of times the cycle repeated. This thesis started by explaining the general background concepts from fields such as computer vision, deep learning and object detection. The current state of the art was presented, which introduced the heuristic, end-to-end, object detection, data-driven, and structural information based methodologies. Finally, due to the current limitations, a solution was proposed, along with the challenges, main contributions and evaluation methodology associated with them. Then the implementation details were presented, including the conception of a synthetic dataset generator and a pipeline responsible to extract information from mockups into their respective prototypes. Moreover, an evaluation process was also presented, along with the judgment of the results.

Section 7.1 introduces the main difficulties faced during the development of this work. Section 7.2 clearly outlines the main contributions that originated from this thesis. Section 7.3 answers and reviews the proposed research questions. Section 7.4 reflects over the current work and provides a glimpse of improvements that could be applied in future works.

Section 7.5 includes the main conclusions that originated from this work.

7.1 Main Difficulties

While researching and preparing this thesis, although some goals were defined problems naturally emerged and the original view was slightly moved towards the final implementation, presented in this document. While multiple proposals were made to transform mockups into digital prototypes, a majority of them focus on digital mockups, where the unpredictable human nature is regularized by the digital media. Therefore a more consistent and regularized mockup is produced. This fact contributes to efficiently use of well-established methodologies, that employ a more traditional approach while segmenting the mockup. This leaves the translation of hand-drawn mockups to be parsed using main approaches, (a) the use of object detectors, or (b) the use of end-to-end solutions. The selection of one approach affects the dataset and environment needed, hence an important step to be committed to.

The creation of a pipeline was then opted, resulting in the use of object detector that imply a simpler dataset, when compared to the end-to-end solutions. The detection of containers in hand-drawn mockups did not seem to be prioritized in the literature, while some solutions were provided [KPW+18], they did not seem feasible to be implemented. Notwithstanding the human capability to draw concise mockups, there can be some small miss alignments or gaps in the final mockup. The extraction of the containers could become then a challenge, and while the use of two independent networks contributed to a usable solution, merging both networks might contribute to better results. The generation of the final hierarchy, as shown by the literature [MBCC+18], can be ambiguous, meaning that a given mockup can have multiple hierarchical representations and the best might differ depending on the final use by the developer. While there are solutions that use real-world data, this work ultimately proposed layout organization algorithms.

7.2 Main Contributions

The synthetic mockup generation (MockGen) was one of the main contributions that resulted from this work. This document describes the steps for the implementation of such an algorithm as well as the impact such a tool can have when training a deep neural network mainly responsible for object detection. Using this algorithm was possible to define the types of elements to be drawn, obtain their respective annotation files as well as segmented examples where an element could be extracted, as an example, the containers that were extracted. The of purely synthetic mockups improved the performance in detection tasks, namely when this data was used as a base and later fine-tuned using real-world information. Therefore contributing not only to a lower number of false positives and a greater number of objects detected. The main conclusion to be taken from this implementation is that it can be useful when hand-drawing examples are in low quantities.

An approach for container detection was also proposed. Due to the irregular nature of hand-drawn mockups, the extraction of containers proven to be a challenge. This detection was possible by executing two distinct models in sequence, (1) for the segmentation technique and (2) posterior detection. Notwithstanding the lower performance when compared to the element detection techniques, the results produced from this implementation were usable. Some improvements are still needed to use this method in a production environment, mainly the creation of a model that results from the combination of both models, thus facilitating the training phase, as well as a higher dataset could improve the final performance.

An extensible pipeline was also proposed. The process to convert an image to a final prototype was divided into four main steps. Therefore the (a) image acquisition, (b) element detection, (c) hierarchy reconstruction, and (d) code generation contributed to a pluggable environment, where the steps run in sequence and the output of one are used as input for the next. This relationship produces a controllable environment, where the intermediate steps can be more easily debugged. When comparing to end-to-end methodologies, the detection, and intermediate code generation is at the demand of the network, thus the quality of the generated prototype is reflected by the training data. With the proposal of the pipeline, new methods techniques can be implemented without having to modify the other phases.

The process of reconstructing a hierarchy was also proposed, namely an algorithm that attempts to organize a set of elements into horizontal or vertical layouts. With this algorithm, it is possible to define which orientation is preferred as the starting point. The layout of a UI can be ambiguous and thus, multiple representations can emerge, depending on the likes or priorities of a developer. This phase was the least explored throughout the development phase, but the usage of real-world data as explored by [MBCC+18] could also be applied in an attempt to create even more useful layouts.

The use of annotations was also proposed, even though there was a small room to experiment with them. Defining custom annotations and enabling the developer or designer to associate a style of behavior of the code generation algorithm can impact the generated prototype. Moreover, more than controlling the style and layout of the generated prototype, it would be possible to define annotations which aggregate elements, thus contributing to the conception of reusable code.

From this work, two papers were produced, one targeting the generator of synthetic datasets and other proposing the pipeline techniques used to translate hand-drawn mockups into a UI representation. To be submitted to a future conference.

7.3 Summary of Specific Challenges

After experiencing the process of implementation of this project, the answers to the hypotheses and specific challenges presented in Chapter 4 became more clear. Therefore in this section, the answer to these questions will be properly addressed taking into account the implementation strategy that was proposed.

H1. *The use of synthetic datasets improves the generation of code prototypes from hand-drawn mockups.*

This document explored the use of synthetic datasets to an object detection model, namely YOLO. From the results included in Subsection 6.3.1 it was concluded that the influence of a synthetic mockup generator is apparent. The results presented reflect the context where the hand-drawn data exists in low quantities. In this context, the detection accuracy had similar results when the network was trained using the synthetic or the hand-drawn data. The further finetune of a model trained using the synthetic information with the use of the hand-drawn dataset improved the mAP value, which in itself contributes to a lower false positive detection rate. The total number of detected objects also increased, this is reflected by the lower log average miss rates for both classes as well as the increase in the true positives that were detected.

The synthetic generator did not perform so well when segmenting and extracting the containers (Subsection 6.3.2), possibly because of these containers were more regular than the hand-drawn ones. Another fault could be the methodology used to train both networks. Both models were not merged and thus an intermediate result must represent perfectly the segmented container in order for the next model extract its position.

In conclusion, it is expected that the usage of a synthetic generator can contribute positively to train a base of object detection models. The further enhancement of the generated mockups is something to take into account, to avoid the need for converting the real world wireframes into high contrast versions. In the end, it is expected that the more elements that are detected the richer and closer to reality the final prototype is.

H2. *The ability to specify and/or infer containers during the drawing process of mockups, improves the layout of the generated prototypes.*

Since the evaluation of the inferred hierarchy can be ambiguous an evaluation as not performed at that level. From the examples presented in Appendix B is possible to verify the impact of the container detection on the inferred layout. From a control point of view, it offers more control over the generated hierarchies. Furthermore, the use of annotations could even improve the definition of the element's disposition. The methodology used to extract the containers did not perform from what was expected (Subsection 6.3.2). Such underperformance could come from the methodology itself or the hand-drawn dataset, which was limited by the camera perspective and number of examples. Nonetheless, providing the ability to manually define containers, during the drawing process, enables the capability for the designer to impose restrictions and less reliability on a layout inferred algorithm.

1. **"How the generation of code can be made and what should it take into account?"** From the approach and experiments that were done during the development of this project (Subsection 5.3.4), the generation of code can be made using a parser that makes use of a stack to store the current container that is currently in use to generate the final code. Using this approach there are multiple types of information that one must take into account in order to process the final UI. The most fundamental is the element's positioning and dimension. Using this information it is possible to compare the sizes of the element inside a given container and thus distribute their size. For instance, usually the elements are slightly miss aligned, the approximation of the elements to a virtual grid can indeed minimize this representation and thus produce more concise results. While some elements don't need to have their height specified by the drawing's representation, figures have a high negative impact when this information is ignored. In conclusion, if the intermediate representation is correctly defined, the code generation just has to iterate over it, only taking the structural and style information into account.

2. **"What information should be included in the intermediate representation?"** The intermediate representation is what connects the element detection results to the code generation. The information that needs to be included may be different from the target results and thus can be customized. The most fundamental types of information are (a) the type of elements and (b) their structural information. It is evident why the type is required, to properly associate later the element with its code representation, but their structural information, while fundamental can be of two kinds. (1) Only relative positioning and hierarchy. This only uses the hierarchy layout without taking into account the dimensions or absolute position of the items. (2) The absolute coordinates of the elements, where it makes possible to generate a prototype even more close to reality. The current implementation also enabled the use of annotations to modify the style of a given container. So it is expected that this information is also tied to the container in some way (Subsection 5.3.3). While the current implementation did not take into account the type of styles or the detection of text, this information can easily be incorporated to specify the color used for the drawing and the text associated.

3. **"How is the hierarchical structure reconstructed?"** The hierarchical reconstruction is nontrivial due to its ambiguous nature, where a given layout can be represented in different ways. The proposed implementation (Subsection 5.3.3) aimed to cover some simple hierarchical patterns, such as the rule where radio buttons may be associated with their text representation, and repeated elements may be included in the same container. But the most significant challenge was the organization of the overall architecture. The algorithm proposed aims to recursively find horizontal or vertical separations in a given mockup. While it can cover some basic cases there are still impossible to solve layouts that cannot be divided into lists and a compromise then must be made. The current implementation instead of compromise this layouts it places them inside one container and classifies it as a grid. Therefore it is then the responsibility of the code generation step to deal with grid-like structures. In summary, the reconstruction of a hierarchical is ambiguous and when using algorithms to tackle this problem, they may not be able to generate the most optimum hierarchy in the eyes of a developer. The use of a distanced based model that aims to find matches in already existing UI hierarchies, may produce better results.

4. **"What techniques are used to extract element related information from a given image?"** The techniques used are mainly based on object detection algorithms, while a small experiment was made to extract using segmentation techniques, due to the irregular nature of hand-drawn mockups, the segmentation was not reliable in cases where (a) lines were not connected or (b) elements were intersecting each other. The use of object detectors facilitated this task of extracting the atomic elements, enabling the possibility to extract elements with such anomalies. For the extraction of containers, the same segmentation techniques were performed, but then again, the irregularities made this task unreliable. Ultimately deep neural network models were used for the segmentation and detection of such containers

(Subsection 5.3.2). Above all, from the experiments made, the use of deep neural networks seems to be a good candidate to extract hand-drawn, data as seen in Section 6.3.

5. **"How hierarchies are extracted from a given Image?"** With the use of end-to-end methodologies this extraction is at the responsibility of the network, this can give a low control over the final result. Another way of extracting information is by performing segmentation techniques. While this approach works great in digital mockups, when the same technique is applied to hand-drawing mockups, the final results can be inconsistent. Therefore, the designer must pay more attention during the drawing process. The final approach and the one that was proposed in this document (Subsection 5.3.3) was to reconstruct the hierarchy, using the relative item positioning provided by object detection techniques. All things considered, the technique used highly depends on the situation and data on hands, and the measurement of the trade-offs is at the hands of the developer.

6. **"What elements can be detected by current methodologies?"** During the development of this project, the object detection technique used had a great overall mAP value, given the data available. One point to take into account is the element's representation. They must be distinct enough so there are no uncertainties, both for the designers and networks during the classification phase. In Subsection 5.3.2, it was proposed a method to extract containers using a deep learning model to segment and isolate them for further detection steps . While the results produced seems reasonable (Subsection 6.3.2), the proposed technique did not perform as well as the atomic element phase. The thin lines used to represent the containers made it more challenging to generate the final mask. Above all, while the use of deep neural networks enabled the extraction and classification of atomic elements, extracting and isolating the containers seems to be more challenging. The use of a network capable of generating regions of interest proposals may yield better results.

7.4 Future Work

Naturally, some ideas emerged during the development of this work. While most of them are already implemented some could improve the final results and usability of this tool. The use of an extensible pipeline enables to implement and experiment with already tested solutions, such as the use of OCR. Thus a semantic analysis would improve the hierarchy pre-processing step better defining the type of element and thus enabling a more refined code generation. The use of other object detectors that trade-off speed for accuracy could also be easily incorporated. By using the same environment, the measurement and comparison of multiple techniques could be measured more precisely for these kinds of works. Another useful addition would be a user interface that controlled the whole pipeline. To that end, a user could quickly iterate and experiment with multiple models and algorithms using a node based editor. With a user interface, the intermediate steps could be better visualized and the user could support the detection steps by manually adding missing elements. These misses could then be stored for latter training.

The synthetic mockup generator that was developed generates high contrast images and the addition of procedural noise or even a set of textures could help to define new drawing surfaces. The use of distortion filters could also be applied to distort even more the final result as well as skew the whole image. The definition of elements is currently tied to classes, decoupling this data from the code is something that would enable the definition of more elements, and rules to be applied. For instance, by using an electronic stylus, a designer could draw multiple versions of a certain element in a vector based program. These vectors could then be added to the synthetic mockup generator. With vector representations, the points that compose the element can be easily skewed and scaled without affecting the final drawn lines. The addition of semantic relationships could also improve the generation of mockups, creating examples closer to the real world.

The use of annotations was lightly explored, but using them in a system similar to this could have a high impact. Using annotations to define, not only the style but the alignment and behavior of elements would improve the richness of the mockup design language. Therefore, providing more ways for designers to express themselves, resulting in more control over the generated UI prototype.

A validation targeting the inferred layouts and the use of annotations is also something to keep in mind and to be performed. A controlled experiment could be performed to evaluate both parameters. One group composed of web developers and another with knowledge of the design language. Both challenged to implement a set of web interfaces including some modifications steps.

7.5 Conclusions

Hand-drawing mockups are not as a well-controlled environment as the digital representations, this imposes multiple challenges when dealing with the irregular nature of humans. Moreover, the introduction of a mockup drawing language is often needed in order to avoid ambiguities over the classification task. When developing such systems using deep neural network methodologies, it is often needed large quantities of data and the proposed synthetic dataset generator seems to fill this gap at the atomic detection level. While the implementation was mainly based on the use of high contrast mockups. The employment of more techniques such as noise, distortion and the addition of texture, could contribute for better results, mimicking the natural environment, therefore increasing the robustness of the generator.

While a user interface oriented to the interaction over the different levels of fidelity [SSSJ19] emerged during the development of this thesis, this work slightly drifted into a more developer-oriented approach, with an extensible pipeline with clearly defined phases, inputs, and outputs, that could improve the research methods. When compared to end-to-end methodologies, the intermediate steps are obfuscated and the quality of the dataset is responsible for the end result. Using a pipeline the addition of new functionality is achievable, creating the possibility for creating a user interface for users, to aid the detection phase [SSSJ19], or experiment with new models and incorporate state of the art techniques. The proposed technique of segmentation and detection of containers proved to underperform, this could be caused by the use of two different models that executed sequentially and by the low quantity of data that was available. When the camera orientation was not similar to the one used on the dataset, the results were unstable and multiple adjustments to the camera needed to be made in order to correct the result.

From the experiments made, even though the use of annotations were not fully explored, they seem to increase a designer's options when it comes to the manipulation of the drawn elements as well as the use definition of different styles. In the current implementation, the detection of elements and annotations is performed by the same network. A dedicated network aiming for the detection of annotations could be preferred, thus enabling the overlapping of annotations and elements. There are some ambiguities that could arise, and some representations might need to be adjusted, but using the developed synthetic dataset generator, this data can be easily generated, separating the annotation into the two types.

References

[ARC$^+$19] Ademar Aguiar, André Restivo, Filipe Figueiredo Correia, Hugo Sereno Ferreira, and João Pedro Dias. Live software development — tightening the feedback loops. In *Proceedings of the 5th Programming Experience (PX) Workshop*, apr 2019.

[Bel18] Tony Beltramelli. Pix2Code: Generating Code from a Graphical User Interface Screenshot. In *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '18, pages 3:1—-3:6, New York, NY, USA, 2018. ACM.

[BFdH$^+$13] Sebastian Burckhardt, Manuel Fahndrich, Peli de Halleux, Sean McDirmid, Michal Moskal, Nikolai Tillmann, and Jun Kato. It's alive! continuous feedback in UI programming. *ACM SIGPLAN Notices*, 48(6):95, 2013.

[BM12] Saket Bhardwaj and Ajay Mittal. A Survey on Various Edge Detector Techniques. *Procedia Technology*, 4:220–226, 2012.

[BMM18] Mohammad Bajammal, Davood Mazinanian, and Ali Mesbah. Generating reusable web components from mockups. *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering - ASE 2018*, pages 601–611, 2018.

[Can86] John Canny. A computational approach to edge detection. *IEEE TRANSACTIONS ON PATTERN ANALYSIS AND MACHINE INTELLIGENCE*, PAMI-8(6):679–698, 1986.

[CG13] Ruchika Chandel and Gaurav Gupta. Image Filtering Algorithms and Techniques : A Review. 2013.

[CMS13] Ricardo J. G. B. Campello, Davoud Moulavi, and Joerg Sander. Density-Based Clustering Based on Hierarchical Density Estimates. pages 160–172. Springer, Berlin, Heidelberg, 2013.

[CSM$^+$18] Chunyang Chen, Ting Su, Guozhu Meng, Zhenchang Xing, and Yang Liu. 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*, ICSE '18, pages 665–676, New York, NY, USA, 2018. ACM.

[DD08] Tore Dybå and Torgeir Dingsøyr. Empirical studies of agile software development: A systematic review. *Information and Software Technology*, 50(9-10):833–859, aug 2008.

[ERSLT17] Kevin Ellis, Daniel Ritchie, Armando Solar-Lezama, and Joshua B Tenenbaum. Learning to Infer Graphics Programs from Hand-Drawn Images. *CoRR*, abs/1707.0, 2017.

[GDDM14] Ross B Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation. *2014 IEEE Conference on Computer Vision and Pattern Recognition*, pages 580–587, 2014.

[Gir15] Ross B Girshick. Fast R-CNN. *CoRR*, abs/1504.0, 2015.

[GMH13] Alex Graves, Abdel-rahman Mohamed, and Geoffrey E Hinton. Speech recognition with deep recurrent neural networks. *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 6645–6649, 2013.

[GS09] Alex Graves and Jürgen Schmidhuber. Offline Handwriting Recognition with Multidimensional Recurrent Neural Networks. In D Koller, D Schuurmans, Y Bengio, and L Bottou, editors, *Advances in Neural Information Processing Systems 21*, pages 545–552. Curran Associates, Inc., 2009.

[HABK18] Saad Hassan, Manan Arya, Ujjwal Bhardwaj, and Silica Kole. Extraction and Classification of User Interface Components from an Image. *International Journal of Pure and Applied Mathematics*, 118(24):1–16, 2018.

[HLC16] R. Huang, Y. Long, and Xiangping Chen. Automaticly generating web page from a mockup. *Proceedings of the International Conference on Software Engineering and Knowledge Engineering, SEKE*, 2016-Janua, 2016.

[HU97] Sepp Hochreiter and Jj Urgen Schmidhuber. LONG SHORT-TERM MEMORY. *Neural Computation*, 9(8):1735–1780, 1997.

[IZZE16] Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A. Efros. Image-to-Image Translation with Conditional Adversarial Networks. nov 2016.

[KFF15] Andrej Karpathy and Li Fei-Fei. Deep visual-semantic alignments for generating image descriptions. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 3128–3137. IEEE, jun 2015.

[Kim14] Yoon Kim. Convolutional Neural Networks for Sentence Classification. aug 2014.

[KPJ$^+$18] Seoyeon Kim, Jisu Park, Jinman Jung, Seongbae Eun, Young-Sun Yun, Sunsup So, Bongjae Kim, Hong Min, and Junyoung Heo. Identifying UI widgets of mobile applications from sketch images. *Journal of Engineering and Applied Sciences*, 13(6):1561–1566, 2018.

[KPW$^+$18] Bada Kim, Sangmin Park, Taeyeon Won, Junyoung Heo, and Bongjae Kim. Deep-learning based web UI automatic programming. In *Proceedings of the 2018 Conference on Research in Adaptive and Convergent Systems - RACS '18*, pages 64–65, New York, New York, USA, 2018. ACM Press.

[Lan95] James a. Landay. Interactive sketching for user interface design. *Conference companion on Human factors in computing systems - CHI '95*, pages 63–64, 1995.

[LB15] Zachary Chase Lipton and John Berkowitz. A Critical Review of Recurrent Neural Networks for Sequence Learning. *CoRR*, abs/1506.0, 2015.

[LBH98] Yann LeCun, Léon Bottou, and Patrick Haffner. Gradient-Based Learning Applied to Document Recognition, 1998.

[LBH15] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, may 2015.

[LCS$^+$18] Thomas F. Liu, Mark Craft, Jason Situ, Ersin Yumer, Radomir Mech, and Ranjitha Kumar. Learning Design Semantics for Mobile Apps. *The 31st Annual ACM Symposium on User Interface Software and Technology - UIST '18*, pages 569–579, 2018.

[LGH$^+$16] Wang Ling, Edward Grefenstette, Karl Moritz Hermann, Tomáš Kočiský, Andrew Senior, Fumin Wang, and Phil Blunsom. Latent Predictor Networks for Code Generation. mar 2016.

[MBCC$^+$18] Kevin Patrick Moran, Carlos Bernal-Cardenas, Michael Curcio, Richard Bonett, and Denys Poshyvanyk. Machine Learning-Based Prototyping of Graphical User Interfaces for Mobile Apps. *IEEE Transactions on Software Engineering*, 5589(May):1–26, 2018.

[NC15] 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)*, pages 248–259, nov 2015.

[PYE$^+$18] Jisu Park, Young-Sun Yun, Seongbae Eun, Sin Cha, Sun-Sup So, and Jinman Jung. Deep neural networks based user interface detection for mobile applications using symbol marker. In *Proceedings of the 2018 Conference on Research in Adaptive and Convergent Systems - RACS '18*, pages 66–67, New York, New York, USA, 2018. ACM Press.

[RASC14] Ali Sharif Razavian, Hossein Azizpour, Josephine Sullivan, and Stefan Carlsson. CNN Features Off-the-Shelf: An Astounding Baseline for Recognition. In *2014 IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 512–519. IEEE, jun 2014.

[RDGF16] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You Only Look Once: Unified, Real-Time Object Detection. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 779–788. IEEE, jun 2016.

[RHGS17] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 39(6):1137–1149, jun 2017.

[SS14] Ritu Sharma and Rajesh Sharma. Image Segmentation Using Morphological Operation for Automatic Region Growing. *International Journal of Computer Science and Information Technologies*, 4(6):5686–5692, 2014.

[SSSJ19] Sarah Suleri, Vinoth Pandian Sermuga Pandian, Svetlana Shishkovets, and Matthias Jarke. Eve. In *Extended Abstracts of the 2019 CHI Conference on Human Factors in Computing Systems - CHI EA '19*, pages 1–6, New York, New York, USA, 2019. ACM Press.

[Sze11] Richard Szeliski. Computer Vision - Algorithms and Applications. In *Texts in Computer Science*, 2011.

[VA13] R Verma and J Ali. A comparative study of various types of image noise and efficient noise removal techniques. *International Journal of Advanced Research in Computer Science and Software Engineering*, 3:617–622, 2013.

[WS15] Paweł Weichbroth and Marcin Sikorski. User Interface Prototyping. Techniques, Methods and Tools. *Studia Ekonomiczne. Zeszyty Naukowe Uniwersytetu Ekonomicznego w Katowicach*, 234:184–198, 2015.

[WSPD11] Christian Wojek, Bernt Schiele, Pietro Perona, and Piotr Doll. Pedestrian Detection : An Evaluation of the State of the Art, 2011.

[YJE$^+$19] Young-Sun Yun, Jinman Jung, Seongbae Eun, Sun-Sup So, and Junyoung Heo. Detection of GUI Elements on Sketch Images Using Object Detector Based on Deep Neural Networks. In Seong Oun Hwang, Syh Yuan Tan, and Franklin Bien, editors, *Proceedings of the Sixth International Conference on Green and Human Information Technology*, pages 86–90, Singapore, 2019. Springer Singapore.

[YJW$^+$16] Quanzeng You, Hailin Jin, Zhaowen Wang, Chen Fang, and Jiebo Luo. Image Captioning with Semantic Attention. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4651–4659. IEEE, jun 2016.

[Zha15] Zhifei Zhang. Image Noise : Detection , Measurement , and Removal Techniques, 2015.

[ZXY18] Zhihao Zhu, Zhan Xue, and Zejian Yuan. Automatic Graphics Program Generation using Attention-Based Hierarchical Decoder. *CoRR*, abs/1810.1, 2018.

[ZZXW18] Zhong-Qiu Zhao, Peng Zheng, Shou-tao Xu, and Xindong Wu. Object Detection with Deep Learning: A Review. *CoRR*, abs/1807.0, 2018.

Appendix A

Mockup Dataset Generator Examples

In this chapter, examples generated using the mockup generator tool are presented, with an example of the generated annotation file.

A.1 Dataset generation overview

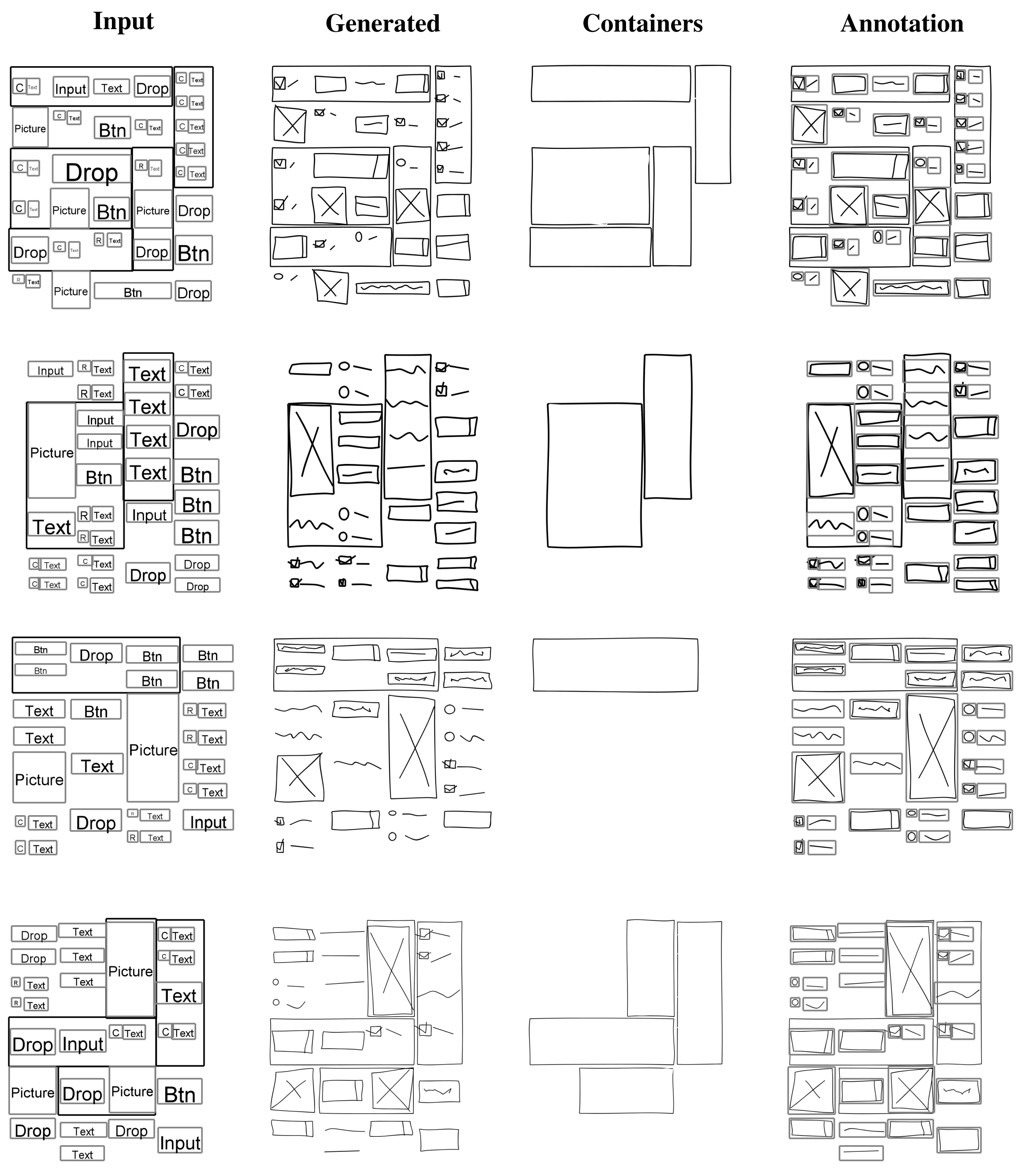


Figure A.1: Example mockups generated using the synthetic dataset generator (MockGen).

A.2 Annotation file example

```
[
    {
        "class":"TextBlock",
        "x":41,"y":61,"w":295,"h":60
    },
    {
        "class":"Textfield",
        "x":358,"y":51,"w":139,"h":80
    },
    {
        "class":"Dropdown",
        "x":510,"y":56,"w":143,"h":70
    },
    {
        "class":"Picture",
        "x":37,"y":174,"w":465,"h":461
    },
    {
        "class":"Picture",
        "x":505,"y":173,"w":143,"h":141
    },
    {
        "class":"Component",
        "x":514,"y":337,"w":118,"h":129
    },
    {
        "class":"TextBlock",
        "x":514,"y":497,"w":134,"h":51
    },
    {
        "class":"Button",
        "x":43,"y":675,"w":296,"h":79
    },
    {
        "class":"Component",
        "x":358,"y":649,"w":126,"h":106
    },
    {
        "class":"Component",
        "x":514,"y":649,"w":103,"h":116
    }
]
```

Mockup Dataset Generator Examples

Appendix B

Code Generation Examples

In this chapter, examples of inputs and the generated websites by using the developed tool are presented.

B.1 Examples of generation given an input

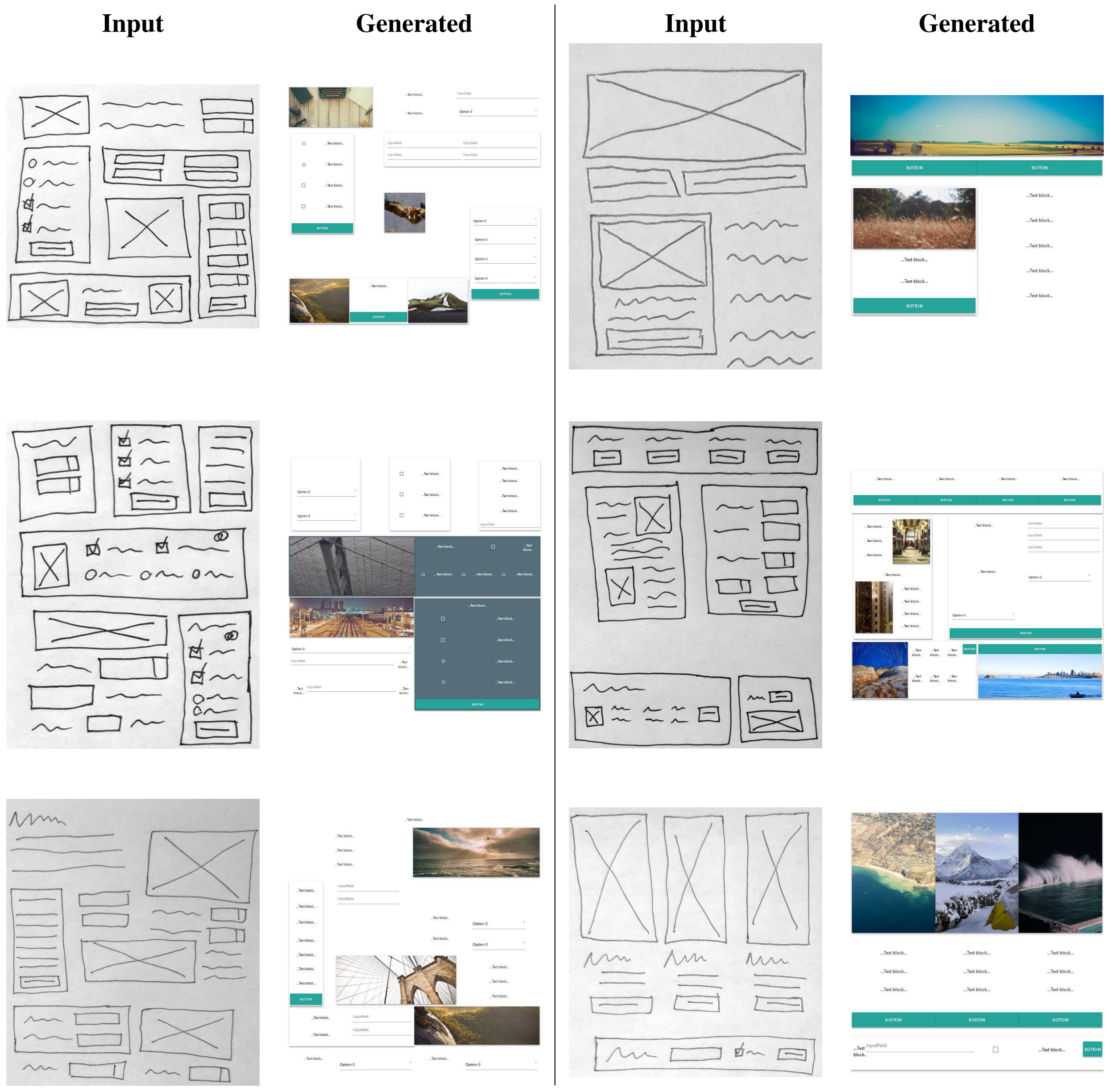


Figure B.1: Results produced from the developed solution.

B.2 Particular example for one Hand-Drawn file

The following section contains a particular example where a mockup is used as an input and the intermediate code, generated HTML and render is provided.

B.2.1 Original image

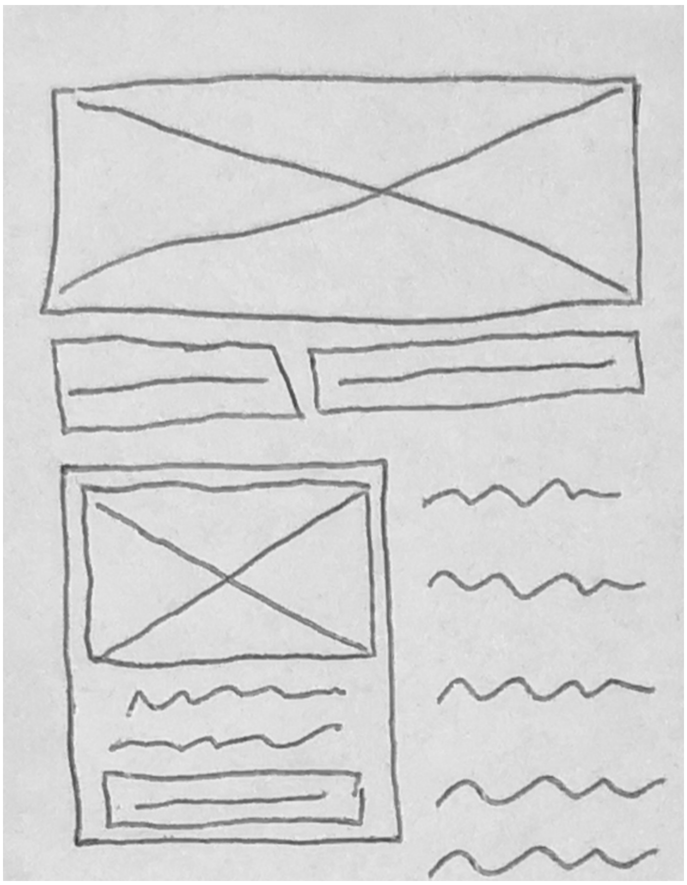


Figure B.2: Hand-drawn mockup example.

B.2.2 Intermediate code representation

```
[
  {
     "class":"Container",
     "x":43,"y":57,"w":677,"h":885,
     "childs":[
        {
           "class":"Picture",
           "class_id":6,
           "x":43,"y":57,"w":651,"h":291,
           "score":0.9964596
        },
        {
           "class":"Container",
            "type":"Button",
            "orientation":"h",
            "x":43,"y":338,"w":663,"h":129,
            "childs":[
                {
                    "class":"Button",
                    "class_id":1,
                    "x":43,"y":348,"w":296,"h":119,
                    "score":0.9040264
                },
                {
                    "class":"Button",
                    "class_id":1,
                    "x":319,"y":338,"w":387,"h":108,
                    "score":0.9823332
                }
            ]
        },
        {
            "class":"Container",
            "type":"_h_",
            "orientation":"h",
            "x":74,"y":478,"w":645,"h":463,
            "childs":[
                {
                    "class":"Container",
                    "class_id":0,
                    "x":74,"y":478,"w":344,"h":433,
                    "score":0,
                    "user": true,
                    "childs":[
                        {
                            "class":"Picture",
                            "class_id":6,
                            "x":75,"y":498,"w":333,"h":218,
                            "score":0.9814187
                        },
                        {
                            "class":"Container",
                            "type":"TextBlock",
                     "orientation":"v",
                     "x":112,"y":725,"w":264,"h":81,
                     "childs":[
                        {
                           "class":"TextBlock",
                           "class_id":5,
                           "x":114,"y":725,"w":262,"h":39,
                           "score":0.5837518
                        },
                        {
                           "class":"TextBlock",
                           "class_id":5,
                           "x":112,"y":774,"w":262,"h":32,
                           "score":0.80906326
                        }
                     ]
                  },
                  {
                     "class":"Button",
                     "class_id":1,
                     "x":106,"y":818,"w":281,"h":80,
                     "score":0.8628884
                  }
               ],
               "components":[],
               "orientation":"v"
            },
            {
               "class":"Container",
               "type":"TextBlock",
               "orientation":"v",
               "x":445,"y":501,"w":275,"h":441,
               "childs":[
                  {
                     "class":"TextBlock",
                     "class_id":5,
                     "x":448,"y":501,"w":231,"h":51,
                     "score":0.93740183
                  },
                  {
                       "class":"TextBlock",
                       "class_id":5,
                       "x":445,"y":598,"w":258,"h":52,
                       "score":0.96128076
                    },
                    {
                       "class":"TextBlock",
                       "class_id":5,
                       "x":446,"y":898,"w":267,"h":44,
                       "score":0.85145843
                    },
                    {
                       "class":"TextBlock",
                       "class_id":5,
                       "x":450,"y":713,"w":263,"h":51,
                       "score":0.9639419
                    },
                    {
                       "class":"TextBlock",
                       "class_id":5,
                       "x":462,"y":818,"w":258,"h":53,
                       "score":0.925761
                    }
                 ]
              }
           ]
        }
     ],
     "components":[],
     "orientation":"v"
  }
]
```

B.2.3 Generated HTML

```
<!doctype html>
<html lang="en">

<head>
```

```html
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">
    <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/materialize/1.0.0/css/materialize.min.css">
    <link rel="stylesheet" href="./style.css">
    <script src="https://cdnjs.cloudflare.com/ajax/libs/materialize/1.0.0/js/materialize.min.js"></script>
    <script src="./main.js"></script>
    <title>Generated Page</title>
</head>

<body>
    <div class="container">
        <div class="container   column  " style="
      grid-column: 2/12 ;
      grid-row: 2/16 ;">
            <div class="img z-depth-2" style="background-image:url(https://picsum.photos/769/295?random=1); min-height:147.5px;"> </div>
            <div class="container   row  " style="">
                <Button class="btn" style=""> Button</Button>
                <Button class="btn" style=""> Button</Button>
            </div>
            <div class="container   row  " style="">
                <div class="container  card z-depth-2 column  " style="">
                    <div class="img z-depth-2" style="background-image:url(https://picsum.photos/393/221?random=2); min-height:110.5px;"> </div>
                    <div class="container   column  " style="">
                        <p class="" style=""> ...Text block...</p>
                        <p class="" style=""> ...Text block...</p>
                    </div>
                    <Button class="btn" style=""> Button</Button>
                </div>
                <div class="container   column  " style="">
                    <p class="" style=""> ...Text block...</p>
                    <p class="" style=""> ...Text block...</p>
                    <p class="" style=""> ...Text block...</p>
                    <p class="" style=""> ...Text block...</p>
                    <p class="" style=""> ...Text block...</p>
                </div>
            </div>
        </div>
    </div>
</body>

</html>
```

B.2.4 HTML Render

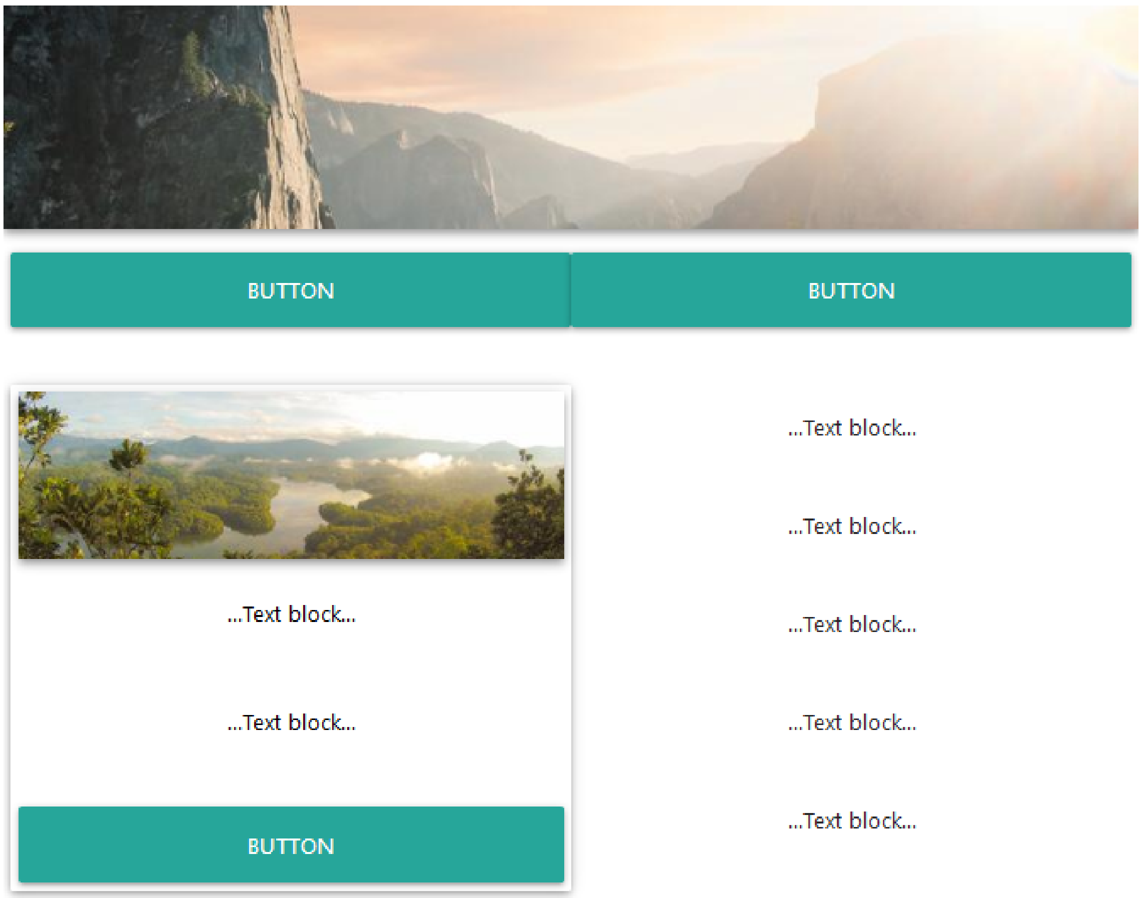


Figure B.3: Example of a generated website.