Automaticly Generating Web Page From A Mockup

Ruozi Huang
School of Data and Computer Science, Sun Yat-san University
Guangzhou China
huangrz@mail2.sysu.edu.cn

Yonghao Long
School of Data and Computer Science, National Engineering Research Center of Digital Life,
Sun Yat-san University
Guangzhou China
longyh3@mail2.sysu.edu.cn

Xiangping Chen*
Institute of Advanced Technology, Sun Yat-san University, Guangzhou China
Research Institute of Sun Yat-sen University in Shenzhen, China
chenxp8@mail.sysu.edu.cn

***Abstract*—UI is an important part of software product. Considering the complexity of web UI, generating the web page from a mockup proposes requirements for rich experience of developer. Extracting visible elements and their relationship, selecting proper tags, generating source code are time-consuming and error-prone task. In this paper, we propose a method to automate the transforming of the mockup to the web page. Our approach starts from the mockup designed by the art designers, and extracts the elements based on the color features of the edges. Then a bottom-up tag generating method based on the Random Forest is proposed to select the tags for elements. Finally the web page is generated by the definition of the elements. The generating tags can achieve an average accuracy of more than 84%, which can meet the basic requirements of the developers.**

Keywords—web generating, machine learning, web design

I. Introduction

As an important part of the software products, the user interface (UI) builds a bridge between the end-user and the functionality. Well-designed UI have good usability and aesthetics, which will attract the users. However, getting a satisfying UI is not easy. The coding includes understanding complex widgets, trying different prototypes to achieve good user experience, and a number of layout strategies. These requirements limit the speed of UI development.

Generally, the UI of web page is relatively complex. Firstly, the number of element is larger. The number of DOM nodes in most web pages is between 50 and 200 [1], while the number of elements in UI of most app is between 10 and 30 based on the statistics of 61,089 UI pages[18]. Secondly, there are numerous tags to be used to define UI elements. For some tags with similar usage context, choosing tag becomes difficult for non-expert developers.

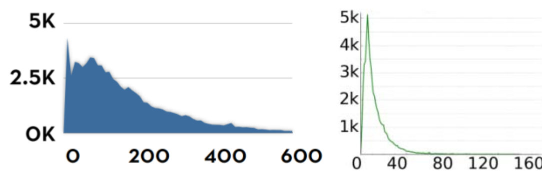

Figure 1. Number of nodes in a web page and UI page.

Many works [8-13] had been proposed to simplify or automatic the UI development process. Modern development environments provide some tools to support the UI development. However, choosing the proper widgets is confusing or non-expert developers. Example based approaches [8, 9] are proposed considering large amount of UI examples in the internet. Examples can tell the developers how to arrange the layout and use the proper widgets. These approaches show that knowledge in the examples is useful to be reused in generating color scheme, device-adaptive UI, and the layouts. However, no approach is proposed for guiding tag selection in web page implementation.

Typical process of developing a draft web page includes the following steps: 1) a wireframe is designed to show the basic functions and structures of the page; and 2) the art designers design a mockup based on the wireframe, the mockup can be regarded as a screenshot of the web page with well designed layout and color theme; 3) the developers extract the elements from the mockup and select tags for the elements; finally 4) developers generate the hierarchical structure of the elements and write HTML and CSS codes to generate a prototype of the web page.

We notice that the process of transforming the mockup to the prototype of web page has the potential to be automated. The edges of rectangles recognized from mockup can be used to extract elements and their relationship. In addition, uniform and standard usage of most tags, which can be learnt from the existing webs. It will save time for developers if they can get a prototype quickly and focus more on the page refinement.

In this paper, we propose a method to generate a draft web page from the mockup. An algorithm is proposed to extract the elements from the mockup based on the wireframe of the mockup. According to the nested relations of the elements, we can get a hierarchical tree of the elements' structures. Since most tags have uniform using in existing web pages, we propose a bottom-up tag generating algorithm to choose tags for each element. This algorithm is based on the Random Forest method. Finally, a prototype of web page is generated by the definition of these elements.

We choose 50 web pages from different websites to verify our method. In the experiment, our algorithm selects the tags

DOI reference number: 10.18293/SEKE2016-231

for 9627 elements with the accuracy of 84.4%. The generated web page can meet basic requirements and provide as a prototype.

The remainder of the paper is organized as follows: Section 2 discusses previous work which is related to our work; Section 3 introduces how to generate the web's source code from a mockup; Section 4 presents the experiments for evaluating the accuracy of the generating tag to the actual tag designed by developers. Finally, Section 5 concludes our work.

II. RELATED WORK

Generating a satisfying Graphical User Interface is very important but difficult in the software development. Most interfaces are still hand-coded, even though there have been a great number of tools to simplify or even automatic the process of GUI generation. Brad et.al [2] had summarized the state of GUI design tools in 2000.

A. *User Interface Generation*

A number of modern development environments have proposed tools to support the GUI development such as the Dreamweaver, Eclipse, and Visual Studio etc. Developers can drag and drop widgets into containers and set the various properties of the widgets, the basic interface code will generated once the UI is designed. These tools are very convenient for generating a prototype of the UI but are not ideal in that a) the code that is generated may not be in a style or form the programmer desires; b) once the code is modified it becomes difficult to use the support to update or change the user interface; c) the generated code often use absolute positions and it is complex to generate easily resizable interfaces and d) it's hard to choose a proper widget for the beginners especially when they are dealing with numerous kinds of widgets.

Another way of getting UI software is searching the existing software from the internet. Developers can get the UI by inputting some keywords in some websites such as Github (www.github.com), Krugle(www.krugle.org), and Open Hub(www.openhub.net) etc. These websites provide rich UI sources but getting a corresponding UI which is close to the developer's requirement is not easy. A suitable searching result needs a set of accurate keywords and a long time searching from the resulting candidates. Some works focused on the improvement of searching engine [3, 4] and simplifying the process of searching [5, 6], which let the UI searching more convenient and convincible. Developers can use the existing UI for inspiration or studying, they can also reuse the UI but generating an original UI is still a tough work.

B. *GUI Redesign*

The user interface design often involves the rapid iterative design, exploration and comparison of different interface implementations [7]. Lots of change will occur during the development of UI like the change of color theme, widget sizes and positions. It takes a lot of energy in changing the user interfaces that some works try to automate the refactoring or redesigning process. Kumar et.al [8, 9] proposed an example-based webpage retargeting method, which enables developers choose an existing web as the example and change the source page to be the example-liked one. Some works focused on the automatic color redesigning of the web for different requirements like the energy saving [10, 11], adaptive to color-deficiency users [12], and color theme modification [13].

With the development of different display devices, some usability problems occur when the traditional web is displaying on these devices. A number of works were proposed to make the traditional web adaptive to the devices' sizes. The key problem includes segmenting the existing UI into some semantic parts [14], resize and rearrange the widgets [15], and refactoring the existing source code [16]. A good summary of the adaptive model-driven UI development is provided by [17].

III. GENERATING THE SOURCE CODE FROM MOCKUP

In this section, we describe the method of extracting the elements from a mockup based on the color feature and the shape of the elements. A bottom-up tag generating algorithm is proposed to generate tags for the elements extracted from the mockup. Based on the information on the elements and their tags, we generate the prototype of the web page.

A. *Extracting the elements from mockup*

The mockup can be regarded as the screenshot of the web page. Noticing that the elements are always rectangle, developers can get the position and size of the element by detecting the edge of the element and extract the sizes and positions. The extracting of the elements in the mockup is finding the separator line in the mockup and extracting the divided blocks. A line is the separator line if it meets the following conditions: 1) the line contains only one kind of color and 2) more than one neighbor line contains different colors. The neighbor line is defined by the following formula:

$$\begin{cases} l_0 : \{ y = y_0, x \in [x_0, x_1] \} \\ neighbor(l_0) : \{ y = y_0 \pm 1, x \in [x_0, x_1] \} \end{cases}$$

We then proposed the separator generation algorithm which finds the separator line in the mockup: for the current image, we find all the horizontal and vertical separator line by the above definition. These lines separate the image into a set of sub-images. Then we do the same procedure for each sub-image. The algorithm ends when no separator line can be found in the image.

Algorithm: Segmenting(G_i ,lines)

Input: An image G_0 of visual design.

Output: An image G_0'with separation lines.

The boolean function isPure(l_m) returns true if the line l_m contains only one kind of color.

Start

if(i=0)

then add the boards of G_i in lines

else{

for(m=0 **to** $G_i.width$)

```
    get vertical line lm (x=m, y∈[0,Gi.width])
    if( isPure(lm) and (!isPure(lm-1)||!isPure(lm+1)))
     then v_l.add(lm);
         sp_Line.add(v_l);
  for(n=0 to Gi.height)
    get horizontal line ln (y=n, x∈[0,Gi.height])
    if( isPure(ln) and (!isPure(ln-1)||!isPure(ln+1)))
      then h_l.add(ln);
           sp_Line.add(h_l);
  G0'= drawLine(G0,sp_Line);
  subG = getSubGraph(Gi, v_l, h_l);
  if(subG.size<=1){
    return;
  }
  else{
     for(Gi+k∈subG)
        segmenting(Gi+k);
   }
End
```

B. Generating tags for the elements

Choosing appropriate tags for elements is very important in front-end development. Well-designed web pages obey the rules of W3C, which will be friendly with the search engine and easy to be understood by the developers. After observing great amounts of web pages we found that elements with different semantics have differences in the performance of vision and structure, thus the appropriate tags can be gotten by considering the elements' visual and structural properties.

Most tags are used in particular situations; we use the Random Forest method to learn the using of tags. An element's tag is defined both by the predicting result and some heuristic rules proposed by us. The following of this section describes the features we get from the elements for training and a bottom-up tag generating method with some heuristic rules.

1) Extracting the elements' features.

The using of the tag is concerned with the content of the element and its function. Which means an element's tag can be speculated by analyzing the element's visual and structure properties. Thus we extract the elements' visual and structure features, and use the Random Forest method to learn the features of tags' using to predict the elements' tags.

The features we extracted from an element include two kinds. One is the visual properties like the element's sizes, positions, main colors and so on. The other is the structural properties like the element's level number in the hierarchical tree, its siblings' and children's number, and its neighbors and so on. Detailed description is proposed in Table 1.

2) Generating the appropriate tags for elements

The web contains numerous kinds of tags, some tags like the <DIV> and <SPAN> don't have the exact semantic meaning. Their using is not regular that their visual properties are not particular. But they often appear in the inner nodes and can be inferred by their children. This phenomenon let us do the tag's generating from the leaf nodes first.

In order to know the frequency tags used in the web, we firstly extract 50 web pages tags and count the tags in them, the results are showed in Fig.1. The statistics of the frequency tags shows that the leaf nodes contains many simpler tags than the inner nodes such as <a>, <img>. These tags are often used in a uniform way and they have more outstanding features such as the <img> often contains complex colors, the <a> and <span> often have a big aspect ratio cause they are always a single line of texts, and so on.

As for the tags of inner elements, the using of tags is tending to represent the structure of web, such as the widely used of <div>. Some of these tags can hardly find some clearly visual features: their position and size are diverse. But some structure features are obvious: e.g. a <ul> always contains some <li> tags, a <p> will contain some <a> or <span>. But the structure features can be extracted only if the tags of children are determined. Thus we give tags for the leaf nodes at first. This is based on two concerns: 1) leaf nodes have simpler tags and their features are more significant than the inner nodes as Fig. 2(a) shows below, and 2) the determination of the leaf nodes' tags can give a hint to the chosen of their parents' tags, thus the whole elements' tags can be determined recursively.

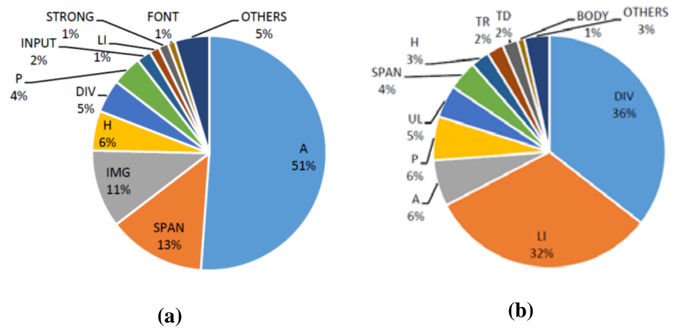

Figure 2. The Top 10th frequency tags used in (a) leaf nodes (A 51%, SPAN 13%, IMG 11%, H 6% , etc) and (b) inner nodes (DIV 36%, UL 32%, A 6%, P 6%, etc).

We use the random forest method to generate tags for the leaf nodes. Since the using of tags of inner nodes is more complex than the leaf nodes, the same method performs badly in this situation. But the leaf nodes' tags can give a hint for finding appropriate tags for the inner nodes. A bottom-up tag generating method is proposed for finding tags for the inner nodes: a node's tags can be determined by its own properties and its children's properties.

TABLE I. DIMENSIONS FOR THE ELEMENTS' TAG GENERATING

ID	Name	Description
1	h	The element's height
2	w	The element's width
3	x	The element's left-top corner's x-value
4	y	The element's left-top corner's y-value
5	r_w	The result of element's width relative to its parent
6	r_h	The result of element's height relative to its parent
7	r_x	The result of element's x-value relative to its parent
8	r_y	The result of element's y-value relative to its parent
9	aspectRatio	width/height
10	Area	width*height
11	Level	Tree level number in the hierarchical tree
12	siblingNum	Number of siblings
13	siblingNum_x	Number of siblings with same x-value
14	siblingNum_y	Number of siblings with same y-value
15	siblingNum_w	Number of siblings with same w-value
16	siblingNum_h	Number of siblings with same h-value
17	colorNum	Number of element's main colors
18	childrenNum	Number of children
19	childrenNum_x	Number of children with same x
20	childrenNum_y	Number of children with same y
21	childrenNum_w	Number of children with same w
22	childrenNum_h	Number of children with same h
23	subtree_height	Height of the element's subtree
24	coverage	Coverage ratio of the element
25	children_type	A string of element's children's types
26	DIV number	Number of children with tag <DIV>
27	P number	Number of children with tag <P>
28	LI number	Number of children with tag <LI>
29	UL number	Number of children with tag <UL>
30	H number	Number of children with tag <H>
31	FORM number	Number of children with tag <FORM>
32	IMG number	Number of children with tag <IMG>
33	INPUT number	Number of children with tag <INPUT>

3) Proofreading

The generating tag from the previous section may contain some mistakes; we propose some heuristic methods to proofread the result.

Our proofreading includes finding the missing elements which belong to a list or table; generating the headers and footers according to the elements' positions; and changing the element's tag according to its children.

The elements which can form a list or table have some obvious features. They are arranged in a line and their widths or heights are close, the neighbors are close. The table can be regarded as a set of adjacent lists with same number of elements. Based on the above features, we propose a heuristic algorithm to find the missing elements which belong to list or table.

Algorithm: findList(e)

Input: An element e of the web page.

Output: A tag of the element *e*.

Start

$E_x = \{e_i \in e.\text{sibling} \mid e_i.x = e.x\}$;

$E_y = \{e_j \in \text{e.sibling} \mid e_j.y = e.y\}$;

Table = $\varnothing$;

if(E_x.*length*>0 and E_y.*length*>0){

// This means they may be in a table

for all $e_i \in E_x$

for all $e_j \in E_y$

if there exists an element e_{ij} s.t.:

$e_{ij}.y=e_i.y;\ e_{ij}.w=e_i.w; e_{ij}.x=e_j.x; e_{ij}.h=e_j.h$;

then

Table.add(e_{ij});

Table.add(e_i) if it doesn't contain it;

Table.add(e_j) if it doesn't contain it;

for all $e_t \in$ Table{

e_t.tag=<TD>

e_t.parent.tag=<TABLE>

}

e.tag = <TD>;

}

else{

if(E_x.length>0){

e.tag = <LI>;

e_i.tag=<LI> ($e_i \in E_x$)

if(*e*.sibling-E_x=$\varnothing$) **then** *e*.parent = <UL>;

else generate a new parent element <UL> of *e* and E_x;

}**else if**(E_y.length>0){

e.tag = <LI>;

e_j.tag=<LI>($e_j \in$ Ey)

if(*e*.sibling-E_y=$\varnothing$) **then** *e*.parent = <UL>;

else generate a new parent element <UL> of *e* and E_y;

}**else**{

e.tag= <DIV>;

}

}

End

C. Layout generation

From the generating elements above, we can write the HTML source code using a stack to achieve the goal. And a

CSS selector is generated to define the element's visual properties. Fig 3. shows the HTML and CSS code of element *e*.

```
HTML:
<e.tag id="e.tag_e.id">
    ...
</e.tag>

CSS:
#e.tag_e.id{
    width: e.width;
    height: e.height;
    offsetLeft: e.x-e.parent.x;
    offsetTop: e.y-e.parent.y;
    backgournd-color: e.getMainColor;
}
```

Figure 3. The HTML and CSS code of element *e*.

IV. EXPERIMENT

A. Implementation of the experiment

The key problem in the web generating from the mockup is guarantee the accuracy of elements' tags. We select 50 web pages from different websites to verify our result. We choose 50 universities' web pages which are following the standards of W3C to do the experiment. Since the elements extracted by the DOM tree are different from those we extracted from the mockup, we do some preprocessing to remove the elements which are fully covered by their children or are not appeared on the web page.

For each time, we use 40 web pages as the training set to predict the left 10 pages. For the generating result, the *prediction* of tag T is the total number of those tags whose tags are predicted to be T. The *ground truth* is the total number of elements whose tags are T. Then we define the *overlap* as the intersection of *prediction* and *ground truth*. And the *precision*, *recall*, and *F1 value* is defined by the following formulas.

$$\begin{cases} precision = \dfrac{overlap}{prediction} \\ recall = \dfrac{overlap}{groundTruth} \\ F1 = \dfrac{2 * precision * recall}{precision + recall} \end{cases}$$

In each time, we record the *precision*, *recall*, and *F1 value* of the generated tags. Then we manually check the result since some tags can be replaced by other tags, the corrected is marked as the accuracy. We will repeat this process until 50 web pages are checked.

B. The analysing of the accuracy of the generating tags

Table 2. records the average accuracy of the generating tags from 50 pages. Since the tag <SPAN> in the leaf nodes are often mixed with the tag <A>, we devide the two tags into different kinds when they are inner nodes and mix them into one kind if they are leaves. The tag set 'H' means the tag of <H1> to <H6>.

From the result we can see the accuracy of the tag <SPAN> in inner nodes is very low. This is because the number of this tag in 50 pages is very small, the use of this tag can be replaced by the tag <A> in most cases.

TABLE II. THE ACCURACY OF THE GENERTAING TAGS

Tag Set	*Precision*	*Recall*	*F1 Value*	*Accuracy*
DIV	0.609	0.822	0.7	0.651
IMG	0.836	0.782	0.808	0.844
INPUT	0.738	0.674	0.705	0.762
P	0.683	0.499	0.577	0.807
SPAN (inner nodes)	0.5	0.074	0.128	0.964
H	0.619	0.609	0.614	0.723
LI/TH/TD	0.861	0.659	0.747	0.958
UL/OL/TABLE	0.662	0.634	0.647	0.946
FORM	0.419	0.818	0.554	0.814
A	0.864	0.697	0.771	0.917
SPAN (leaves)	0.911	0.952	0.931	0.940
Weighted Average	0.782	0.782	0.775	0.844

C. Result discussion

We choose some pages as the examples to analyze the generating tags. From table 2 we can see the tag <P> has a very low accuracy before we do the manually correction. This is because most of these elements are grouped to the class DIV, and in this case, this classification is accepted as Fig. 3(a) shows. The low accuracy of <FORM> has the same reason that some parent tag of <INPUT> is <DIV> but we change them to be the <FORM> which is also acceptable.

Some of our result has improved the structure as Fig. 3 (b) shows, in this case, the original tag of the element is <A> but it's a title obviously, thus our method gives it a tag <H>. Same case can be found in Fig. 3(c) that the original tag is <LI> but it's a single tag with a line of text, the predicting tag is <A>.

Also our results have some mistakes, as Fig. 3 (d) and (e) shows. In Fig. 3(d), the element should be an image with tag <IMG>, but our method gives it a tag of <P>. It's because our method count the number of main color by counting the clusters of colors in the image, which returns two colors and the method thought it's a paragraph of texts. Adding the analysis of fonts with the help of OCR may reduce these mistakes. The lacking of the analysis of fonts causes the similar errors in predicting the titles. Our method doesn't take the properties of font into consideration that it wrongly gives the elements a tag <A> instead of <H>.

(a) the predicting tag is <DIV> and the original tag is <P>

(b) The predicting tag is <H> and the original tag is <A>

(c) The predicting tag is <A> but the original tag is <LI>

(d) The predicting tag is <P> but the original tag is <IMG>

(e) The predicting tag is <A> but the original tag is <H>

Figure 4. Some inconsistencies between the generared tags and the element's actual tags (a) both of the tags are accpetable, (b),(c), our results are accpetable but the actual tags are wrong; (d),(e) our results are wrong.

The high accuracy of leaves with tag <A> and <SPAN> has two reasons; the first is out method can find a line of text is effectively, and the second is most leaves are texts in our selecting web pages.

The results show that in most cases, our method can generate convincible tags for elements. But the analysis of fonts could be added to reduce mistakes in the generating of some tags of texts and logos. And the analyzing of color features should be refined.

Conclusion And Future Work

In this paper, we propose a method which can generate a web page from the mockup. The bottom-up tag generating method is proved to be corrective by the experiment. Future work could extend to make the generating layout to be the responsive layout to meets the requirements of numerous kinds of devices. The current generating web page is static and some basic interactive features could be added in it. Furthermore, some strategies will be used to improve the accuracy of the generating tags.

Acknowledgment

This research is supported by the NSFC Guangdong Joint Fund (No. U1201252), the Science and Technology Planning Project of Guangdong Province (No. 2014B010110003), and the Research Project of Educational Commission of Guangdong Province (No. 2013CXZDB001).

References

[1] Kumar Ranjitha, Arvind Satyanarayan, Cesar Torres, Maxine Lim, Salman Ahmad, Scott R. Klemmer, and Jerry O. Talton. "Webzeitgeist: Design mining the web." In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, pp. 3083-3092. ACM, 2013.

[2] Myers B, Hudson S E, Pausch R. Past, present, and future of user interface software tools[J]. ACM Transactions on Computer-Human Interaction (TOCHI), 2000, 7(1): 3-28.

[3] Sushil Bajracharya, Trung Ngo, Erik Linstead, Yimeng Dou, Paul Rigor, Pierre Baldi, and Cristina Lopes, "Sourcerer: a search engine for open source code supporting structurebased search." Proceedings ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications 2006, pp. 682-682.

[4] Wing-Kwan Chan, Hong Cheng, and David Lo, "Searching connected API subgraph via text phrases." pp. 1-11 in Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, (2012).

[5] Reiss, Steven P. "Seeking the user interface." Proceedings of the 29th ACM/IEEE international conference on Automated software engineering. ACM, 2014, pp. 103-114.

[6] Steven P. Reiss, "Semantics-based code search," International Conference on Software Engineering 2009, pp. 243-253 (May 2009).

[7] Bjorn Hartmann, Leith Abdulla, Manas Mittal, and Scott R. Klemmer, "Authoring sensor based interactions through direct manipulation and pattern matching," Proceedings of CHI 2007: ACM Conference on Human Factors in Computing Systems, pp. 145-154 (2007).

[8] Lee, B., Srivastava, S., Kumar, R., Brafman, R., Klemmer, S. R. Designing with interactive example galleries. Proc. CHI (2010), ACM.

[9] Kumar, R., Talton, J. O., Ahmad, S., & Klemmer, S. R. (2011, May). Bricolage: example-based retargeting for web design. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (pp. 2197-2206). ACM.

[10] Dong, Mian, and Lin Zhong. "Chameleon: a color-adaptive web browser for mobile OLED displays." Mobile Computing, IEEE Transactions on 11.5 (2012): 724-738.

[11] Li, Ding, Angelica Huyen Tran, and William GJ Halfond. "Making web applications more energy efficient for OLED smartphones." Proceedings of the 36th International Conference on Software Engineering. ACM, 2014.

[12] Flatla, D. R., Reinecke, K., Gutwin, C., & Gajos, K. Z. (2013, April). SPRWeb: Preserving subjective responses to website colour schemes through automatic recolouring. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (pp. 2069-2078). ACM.

[13] Chen, X., Long, Y., & Luo, X. (2015). Automatic Color Modification for Web Page Based on Partitional Color Transfer. In Software Reuse for Dynamic Systems in the Cloud and Beyond (pp. 204-220). Springer International Publishing.

[14] Cai, D., Yu, S., Wen, J. R., & Ma, W. Y. (2003). VIPS: a visionbased page segmentation algorithm (p. 28). Microsoft technical report, MSR-TR-2003-79.

[15] Rossi G, Urbieta M, Ginzburg J, et al. Refactoring to rich internet applications. A model-driven approach[C]//Web Engineering, 2008. ICWE'08. Eighth International Conference on. IEEE, 2008: 1-12.

[16] Sánchez Ramón, Ó., Sánchez Cuadrado, J., & García Molina, J. (2010, September). Model-driven reverse engineering of legacy graphical user interfaces. In Proceedings of the IEEE/ACM international conference on Automated software engineering (pp. 147-150). ACM.

[17] Akiki, Pierre A., Arosha K. Bandara, and Yijun Yu. "Adaptive model-driven user interface development systems." ACM Computing Surveys 47.1 (2015).

[18] http://research.defool.me/appprofiler/