Generating Reusable Web Components from Mockups

Mohammad Bajammal
University of British Columbia
Vancouver, BC, Canada
bajammal@ece.ubc.ca

Davood Mazinanian
University of British Columbia
Vancouver, BC, Canada
dmazinanian@ece.ubc.ca

Ali Mesbah
University of British Columbia
Vancouver, BC, Canada
amesbah@ece.ubc.ca

ABSTRACT

The transformation of a user interface mockup designed by a graphic designer to web components in the final app built by a web developer is often laborious, involving manual and time consuming steps. We propose an approach to automate this aspect of web development by generating reusable web components from a mockup. Our approach employs visual analysis of the mockup, and unsupervised learning of visual cues to create reusable web components (e.g., React components). We evaluated our approach, implemented in a tool called VizMod, on five real-world web mockups, and assessed the transformations and generated components through comparison with web development experts. The results show that VizMod achieves on average 94% precision and 75% recall in terms of agreement with the developers' assessment. Furthermore, the refactorings yielded 22% code reusability, on average.

CCS CONCEPTS

• **Software and its engineering** → *Software development techniques*; *Software prototyping*;

KEYWORDS

web UI, web components, web refactoring, machine learning, computer vision

ACM Reference Format:
Mohammad Bajammal, Davood Mazinanian, and Ali Mesbah. 2018. Generating Reusable Web Components from Mockups. In *Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18), September 3–7, 2018, Montpellier, France.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3238147.3238194

1 INTRODUCTION

The development of user interfaces (UIs) for web apps is often a manual and time consuming task. In a survey of more than 5,700 developers, 51% reported working on app UI design tasks on a daily basis [15], more so than other development tasks, which they tended to perform every few days. Another study also showed that an average of 48% of the code size of software is related to the user interface [39].

A common workflow for creating web user interfaces is *mockup based design* [40, 41]. In this approach, a graphic designer creates a rough illustration of the anticipated UI design, called the *mockup* or *wireframe*, usually through a graphic design software or a WYSIWYG editor. This mockup is then exported to HTML to be rendered in a browser. A web developer then examines the mockup and begins constructing web components for the app, which are nowadays implemented in one of the popular front-end frameworks such as Angular [13] or React [16].

The main building block of UI design, and a cornerstone of these front-end frameworks, is the concept of *reusable components* [14, 17], which are a set of APIs and coding practices allowing reuse and encapsulation of repeated patterns on the front-end. Reusable components help improve modularity and maintainability, make the code more testable, and effectively remove duplication, by offloading the task of creating repetitive patterns to the web browser at runtime. Recent surveys show that using front-end frameworks is extensively popular among web developers. In one survey more than 92% of around 28,000 surveyed web developers stated that they use a framework rather than constructing UIs using pure HTML [53]. As a result, creating reusable components is often an essential element of building an app's front-end.

This component creation process can often be time consuming and tedious [47] in practice; it requires several manual steps, including the examination of the mockup, checking potential elements that may or may not be suitable for conversion to components, constructing a template for components that unifies repeated segments, adding placeholders for variable content, and refactoring the code to replace instances with instantiated components [47].

To the best of our knowledge, there has been little to no automated support in creating these reusable web components from mockups. Existing techniques help to manage mockups themselves, but do not generate any components. For instance, one set of approaches [44, 51] takes a mockup as input and converts its layout into a responsive code (e.g., through CSS) such that it is flexible to maintain the layout on different display sizes. Others [35] propose a tool that overlays the mockup as a transparency layer while implementing the UI, and performs a snapping-like functionality that aligns against various parts of the mockup.

In this paper, we propose a technique, implemented in a tool, VizMod, to fill this gap by automatically generating reusable web components from mockups. Given a web mockup, our technique automatically identifies patterns on the UI, refactors the HTML code, and creates reusable components for popular front-end frameworks that are already familiar to developers such as React or Angular. At the core of our approach is an unsupervised machine learning process for the detection of reusable UI patterns; we use features composed of a hybrid of information obtained from the Document Object Model (DOM) as well as the visual analysis of the UI.

ASE '18, September 3–7, 2018, Montpellier, France
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5937-5/18/09…$15.00
https://doi.org/10.1145/3238147.3238194

We evaluate VizMod on five real-world web mockups by automatically identifying and transforming 120 component instances into 25 components. We also ask five experts to manually find patterns on the mockups and compare the output from our approach with the manually-identified patterns. Our approach is able to achieve 94% precision and 75% recall, on average, in correctly detecting reusable patterns in the UIs.

This paper makes the following main contributions:

- A novel approach for automatically generating web components (e.g., React, Angular) from mockups, which is the first to address this issue, to the best of our knowledge.
- A technique for visually analyzing the structure of a mockup and transforming it into components via unsupervised machine learning.
- An implementation of our approach, available in a tool called VizMod.
- A qualitative and quantitative evaluation of VizMod in terms of its accuracy and reusability of the generated components.

2 MOTIVATING EXAMPLE

Figure 1 illustrates a part of a sample UI mockup for a job hunting website. The mockup, often designed by a graphics designer in a team, provides a visual representation of what the UI of the web app is supposed to look like. The code corresponding to this mockup includes the HTML code, which defines the mockup's structure and content, and CSS code, which defines its style and presentation. This code is typically generated automatically using popular web UI editors (e.g., Muse, Dreamweaver, Visual Studio). The HTML and CSS code is interpreted by web browsers to render the UI.

Subsequently, a web developer oversees the creation of the final front-end code for the app. For the vast majority of developers, a major part of this process is the creation of reusable components [53]. Using components is key in improving modularity and maintainability and achieving the software engineering best practice of DRY (Don't Repeat Yourself). It is also an effective way to remove duplications in the app's code, which has been shown to be associated with increased error-proneness [19], maintenance effort [28], code instability [37], as well as higher hosting costs and rendering delays due to the transmission of redundant data. The utilization of reusable web components can help to address these issues.

For example, observe in Figure 1 that there are four groups of elements repeated in the mockup, denoted by Ⓐ, Ⓑ, Ⓒ and Ⓓ. Notice that the repeated elements are not exactly similar; there are differences in terms of, for example, the text and images appearing within the elements. Nevertheless, the structure of the repeated elements in each group and their overall visual appearances are unquestionably repeating. Reptitions in UI are unavoidable and neccessary. In fact, repetition is an important aspect of effective visual design [33], and is known as a *functional technique* to achieve appealing designs [11]. Research has shown that, when a *visual stimuli* is repeated, it is more likely to be accepted by people, a phenomenon called *repeated exposure* [23].

However, the process of creating components is rather time consuming, requiring manual effort [47]. When analyzing repetitions in a mockup to construct components in one of the modern UI frameworks, developers often face the following challenges:

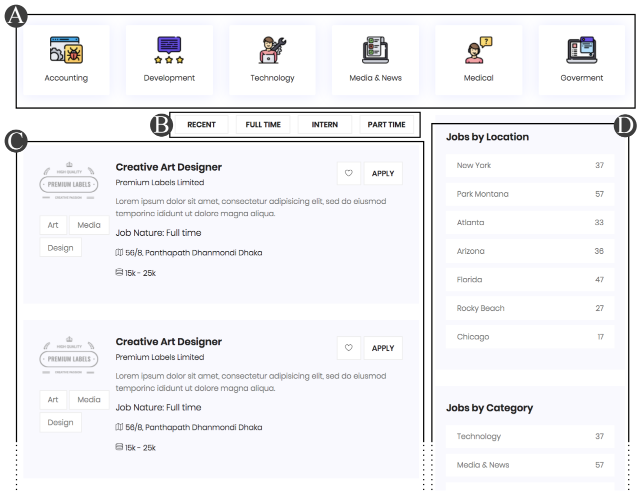


Figure 1: An example of a web UI mockup.

- They need to visually glance at the mockup and manually identify the patterns in the UI that can be potentially refactored to create a reusable component. For instance, for group Ⓐ, they need to find all patterns in the page that represents components similar to the elements inside group Ⓐ. The repetition might be spread across the web page, making the identification more challenging. The developer has to repeat this same process for other groups of components on the page, which quickly becomes a time consuming manual effort. Note that, this identification is not possible by only using existing code clone detection tools that support HTML code as input (e.g., NiCad [49]), due to several reasons:
  (1) These tools leave out the visual appearance of the elements and only work at the source code level, which is sub-optimal since there are several *inherent patterns* in HTML which do not necessarily represent a UI component. For example, HTML tables are declared using a `<table>` tag followed by a series of other tags, e.g., `<thead>`, `<tbody>`, `<colgroup>`, `<tr>`, and `<td>`, nested in a predefined hierarchy. The clone detector might mark all tables on the page for extraction, even if they do not visually constitute a reusable component in the UI. The same happens for several other elements, such as (un)ordered, description, and drop-down lists.
  (2) Clone detectors need to be configured properly in order to yield desirable clones. There is usually a large list of parameters and thresholds to tune, and finding an optimal configuration is a laborious task [57].
  (3) Clone detectors are not aware of the ultimate reason for detecting clones, e.g., there is no configuration that can force them to only identify refactorable clones.
- The developer also needs to *unify* the patterns to construct a reusable component in a UI framework. This process needs careful investigation of repeated HTML, to identify how elements can be unified into one representative component, and which elements can be *parameterized* when there are differences. For example, in group Ⓐ, a developer would examine each button

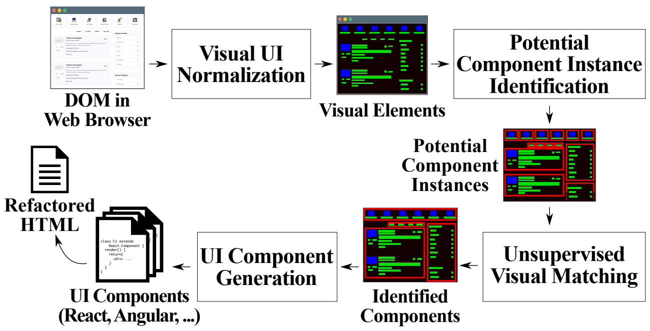

DOM in Web Browser
Visual UI Normalization
Visual Elements
Potential Component Instance Identification
Potential Component Instances
Unsupervised Visual Matching
Identified Components
UI Component Generation
UI Components (React, Angular, ...)
Refactored HTML

Figure 2: Overview of the proposed approach.

in the group, and determine which parts are repeated between the buttons, and which part is variable (e.g., the button icon and its label). The constructed component should resemble the exact hierarchy of the original repeated elements, or else the output of the resulting UI might differ from the original one.

- Moreover, to use the constructed component, the developer has to *instantiate* it in the places where the repeated elements originally appeared, with the appropriate parameters (e.g., original texts or images) to preserve the output of the mockup. For example, in group A, the developer needs to refactor the original code and replace every occurrence of a button with a call to the button component, passing along arguments for the button label and its image.

To the best of our knowledge, there has been no techniques available to address the aforementioned issues and support developers in the generation of components.

3 PROPOSED APPROACH

Figure 2 shows an overview of our proposed approach to automatically generate modularized reusable UI components from mockups. The approach begins by retrieving the DOM of the web app's mockup. Next, a visual abstraction is performed to generate a normalized and abstract representation of the web app's UI layout. This transforms the mockup into a set of visual elements (VEs) on which further analysis is conducted. The approach then performs a dynamic grouping of visual elements, to identify subtrees which correspond to potential instances of a UI component. This grouping is used in the next step, where an unsupervised machine learning technique applied on the potential UI component instances identifies UI components. Finally, the actual code for the UI components is generated by refactoring the original HTML code.

In the following subsections, we describe each step of the proposed approach and illustrate some of their major components and analysis procedures.

3.1 Definitions

Before we proceed to describe the details of the proposed approach, we begin by declaring a few important definitions that are used throughout the paper.

Definition 1 (**UI Component**). *A UI component $c_E = \langle n, N \rangle$ for a repeated group of UI element trees E in a web application is a tree structure rooted at $n \in N$, where $N = T \cup P$ is a set of abstract user interface elements. The component includes the* template T *and the* placeholders P. *The template of the UI component denotes the nodes which do not change wherever the component is used (i.e.,* instantiated*), while the placeholders allow parameterized nodes.*

In this paper, we use the terms *UI component* and *component* interchangeably.

Definition 2 (**Component Instance**). *A component instance $i = \langle c_E, f \rangle$ is a concrete and specific instantiation of a UI component c_E. Component instances share the template part with other instances of the same component, but differ in the placeholder parts. The function $f : P \rightarrow V$ assigns values $v \in V$ to the placeholders $p \in P$ of c_E.*

Definition 3 (**Potential Instance**). *A potential instance is a subtree of the DOM constructed for a web application's user interface, representing a concrete UI element tree that is* likely *to form a component instance, but may not be so.*

Potential instances are processed at multiple stages of the proposed approach until they are either discarded or associated with a component.

3.2 Visual UI Normalization

In the first step of the approach, we take as input the DOM of the mockup after it is loaded and rendered in a browser, and perform a *visual normalization* that transforms the DOM into a set of *visual elements*. The goal of this step is to normalize the visual presentation of a web user interface into a set of abstract elements that signify the salient features of the page from a visual perspective, which may represent potential component instances. The intuition behind this is that normalization and abstraction can be helpful to achieve our goal of detecting reusable patterns, since the exact and minute details are less relevant when identifying repeated regions of a web page. Furthermore, component instances are generally different from each other in some aspects, while they still have similar overall visual appearance. This normalization step enables obtaining a big picture to identify these potential similarities.

The visual normalization is achieved as follows. First, we extract from the DOM a set of nodes that represent visual content of the UI, and we refer to each of these as *visual elements*. We define two main types of visual elements: textual and graphical (image). The extraction of text content is achieved by traversing text nodes of the DOM. More specifically:

$$\Gamma_T := \{E(node) : node \in DOM_R \\ \wedge node.hasTextContent\} \quad (1)$$

where Γ_T is the set of all visual elements that represent text in the UI, DOM_R is the rendered DOM in the web browser, and $E(node)$ maps the node to the corresponding element. The predicate *hasTextContent* examines whether there is a text associated with the node, and covers two possibilities: non-empty nodes of type `#TEXT`, representing string literals in DOM_R, and nodes of `input` elements that have an associated text value (e.g., buttons or lists). Subsequently, we perform another extraction for image content. We define this as follows:

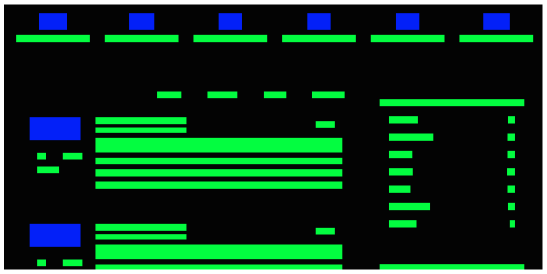


Figure 3: The result of the visual UI normalization stage (as applied to the motivating example of Figure 1). Best viewed on a color display.

$$\Gamma_I \coloneqq \{E(node) : node \in DOM_R \wedge node.hasImageContent\} \tag{2}$$

where Γ_I is the set of all visual elements that represent images. As in the previous case, the predicate *hasImageContent* examines if there is an image associated with the node. This again has two possibilities: nodes of `<img>` elements and non-img nodes with a non-null background image attribute.

Subsequently, we use the set of all visual elements to construct the normalized UI:

$$\text{UI}_N = V(\Gamma_I \cup \Gamma_T) \tag{3}$$

where UI_N is the resultant normalized UI and V is a visual projection operation that generates an image from the union of visual elements. This is achieved as follows. First, we begin by collecting the final *computed* properties of each element, when rendered in the web browser. These properties represent the final state of elements after the propagation of all changes and events. The properties we collect are the size, location, and z-orders of these elements. Next, we assign different colors to each class of visual elements. We assign green for elements in Γ_T, and blue for elements in Γ_I. While any arbitrary colors could have been chosen, we chose these two colors in order to facilitate faster visual analysis in subsequent steps, since these two are typically represented in separate color channels. Figure 3 illustrates an example of the output generated from this visual normalization step. As can be observed, the minute details of the page are abstracted away while the main and essential structure of the UI is accentuated.

3.3 Potential Instance Identification

The result of the previous step consists of only a set of visual elements. These visual elements on their own do not *necessarily* represent reusable repetitive UI patterns. The goal of this step is to transform the set of individual visual elements into a set of *potential* reusable UI component instances. These potential component instances will be further checked and analyzed in the subsequent steps in order to generate a final set of components.

Identifying potential component instances can be an intricate decision since there are multiple levels of hierarchy that can be considered. For example, consider group Ⓐ in Figure 1. Notice how the icons in that group would constitute repeated elements. The same is true for the text labels under the icons. Yet another repetition pattern is taking the icon and text as one component that is repeated multiple times. Accordingly, in order to identify potential component instances, we propose an approach that aims to maximize two complementary aspects, namely, the number of repetitions of a component, and the amount of repetitions encapsulated *within* each component instance. We refer to this combination of aspects as the *modularization potential*, where a high value of modularization potential indicates a highly reusable UI component. Our goal is therefore to utilize this modularization potential to optimize a set of potential instances, Ψ:

$$\Psi \coloneqq \underset{C}{\text{argmax}} \prod_{c_i \in C} \left|\{c_i(\gamma_T, \gamma_I) : \gamma_T, \gamma_I \subset \text{UI}_N\}\right| \tag{4}$$

where C is the set of all component instances, c_i is a potential instance, and the optimized function is the modularization potential. This optimization yields a global optimum set of potential instances between two extremes. At one end of the spectrum, each visual element represents a component of its own. This yields a sub-optimal component set that has low modularization potential because of a lack of repetitions. For this case, the modularization potential in eq. (4) yields a score of 1 since each component encapsulates only a single element. At the other end of the spectrum, one might theoretically consider the *entire* collection of visual elements to represent a single component that is repeated only once. This results in a score equal to N, the number of total visual elements, in eq. (4). Ψ, on the other hand, represents a global optimum between the aforementioned two extremes. Ψ captures a set of potential component instances that aims for *both* a large number of components, and for an instance that in itself has a large number of UI repetitions. The subsequent steps of the approach will therefore only use Ψ for further analyses and final generation of components.

We now describe the implementation for generating Ψ. Figure 4 shows an illustration of this process. First, we obtain DOM locators (e.g., XPath expressions) for each of the visual elements. Next, starting from these locators as leaf nodes, we iteratively build a tree from the bottom up (as shown in Figure 4), adding the DOM parent of every tree node with each iteration. At each iteration, we calculate the modularizaton potential of eq. (4), with every node's subtree representing a potential instance c_i. The potential instances are illustrated using the red outlines in Figure 4. Note how at the very first iteration, each potential instance is simply the visual element itself. In the next iteration, the potential instances grow larger to include more visual elements as shown by the larger red outlines at iteration 1. Finally, the iteration that yields the maximal modularization is reported as the Ψ set and passed to the subsequent stage.

3.4 Unsupervised Visual Matching

The output of the previous step is a set of *potential* component instances that maximizes the modularization potential out of many alternative sets of instances. However, these are only *potential* instances that may or may not actually belong to a component. In other words, there is still no information as to which subgroup of potential instances do indeed belong together and constitute a reusable component, versus other potential instances that are simply visual elements that do not represent repetitive reusable

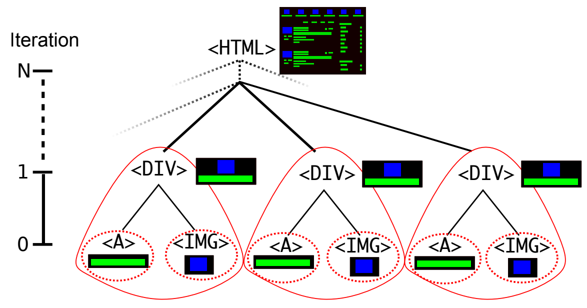


Figure 4: Illustration of the potential instance identification stage. Each iteration considers a different group of potential instances before selecting an optimum set.

components. In this stage, we process the set of *potential* component instances and reduce it into a final set of components.

In order to create the final components, we propose an approach that visually examines potential instances and combines them into components via unsupervised machine learning. The intuition behind adopting this approach is that if potential instances match with other potential instances, the "potential" qualifier can be dropped from these instances and they would be recognized as constituting a component together. In this approach, we use a clustering mechanism to create components in order to facilitate robust matching of potential instances.

We now describe the details of the process. First, we obtain the screenshot image of the visual elements per potential instance. This results in one image (containing all visual elements) for each potential instance. Next, for each potential instance image, we extract a feature vector. We compute the feature vector using a *vectorized pixel histogram*, which is a process that captures a summary of the overall content in the instance image. However, unlike typical approaches from the machine vision literature [26, 27] where a binning parameter (a parameter for categorizing pixels) is required, we generate the vectorized histogram without requiring this parameter. Instead, due to the nature of visual normalization that we have proposed, only two categories need to be considered: one for text visual elements, and another for image visual elements. Therefore, we finally end up with a feature vector for each potential instance. Subsequently, we compute the cosine distance between each pair $\mathbf{I}_i, \mathbf{I}_j$ of potential instances:

$$D_{i,j} = 1 - \frac{\mathbf{I}_i \cdot \mathbf{I}_j}{\|\mathbf{I}_i\|\|\mathbf{I}_j\|} \tag{5}$$

Next, we perform an unsupervised clustering process. The selection of an appropriate clustering is of paramount importance due to a couple of challenges. First, the clustering can be challenging due to the wide range of possibilities of arrangements and structures of component instances. In other words, there is potentially a large range of *inter-* and *intra-*component variations. This makes it difficult to use hierarchical clustering, for instance, due to its very high sensitivity to outliers and therefore would be a poor choice for handling large component variations, and also due to its high dependence on order of data, which can make it less effective for detecting instances far way from each other. Furthermore, performing a cut on the clustered hierarchies often requires specifying the number of clusters or some other parameter, which can be difficult and brittle to specify. Density-based algorithms (e.g., DBSCAN) would not be effective either, as they would have difficulty handling the *variable* densities present between potential clusters of instances. Accordingly, we opted for a technique that can be flexible enough to correctly identify such variations and be able to better recognize the final components. To do this, we select a method that performs variable-density clustering with a hierarchy of densities [5]. The hierarchy of variable-densities allows the method to automatically detect stable clusters in a parameter-free fashion. More importantly, the method is built to handle varying-densities, which becomes very important when handling the potentially large range of inter- and intra-component variations.

Once the components have been identified through unsupervised visual matching, we extract the corresponding locator in the DOM (e.g., XPaths) per instance. The final result is a superset of component instance locator sets. This superset is passed on to the next step in order to combine the component instances into final components.

3.5 UI Component Generation

We propose an algorithm that unifies the UI component instances identified in the previous steps into a component implemented using a web framework (e.g., React, Angular, HTML Web Components [38]). However, instead of directly generating the framework-specific code for components, we opt for constructing an *intermediate model* that effectively represents components at a higher level of abstraction. This allows building different *translation strategies* for generating the actual code for different frameworks from the same model, with the added benefit of remaining agnostic to the specific details of a particular framework. Our implementation supports the React [16] translation strategy, which is the preferred framework for a significant number of developers in practice [52, 53]. We first define the terms used in this step.

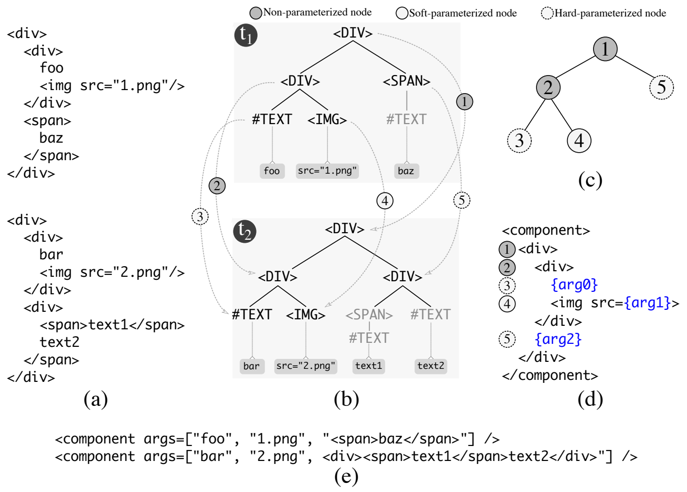
Definition 4 (**Mapping Nodes Set**). *Let $T = \{t_1 \ldots t_n\}$ be the list of DOM subtrees for n instances of a UI component identified by the previous phases of the approach. A set $D = \{d_1 \in t_1, d_2 \in t_2 \ldots d_n \in t_n\}$ of DOM nodes corresponding to T is a Mapping Nodes Set, when every pair (d_i, d_j) of DOM nodes belonging to D are* ***mapping****.*

Definition 5 (**Mapping**). *Two DOM nodes d_i and d_j are mapping (denoted as $d_i \leftrightsquigarrow d_j$) when:*

- *Both d_i and d_j are root nodes of their trees, or*
- *d_i and d_j are not root nodes, and*
  - *d_i.parent.tag = d_j.parent.tag, and*
  - *d_i.parent $\leftrightsquigarrow$ d_j.parent, and*
  - *d_i and d_j have the same child index (e.g., they are both the first child of their parents).*

Definition 6 (**Component Intermediate Model**). *The Component Intermediate Model is a rooted, ordered tree in which each node corresponds to a Mapping Nodes Set. The hierarchy of this tree follows the mapping DOM nodes' hierarchy.*

Example. Figure 5(a) depicts the HTML code snippets corresponding to two identified UI component instances. The corresponding



```
<div>
  <div>
    foo
    <img src="1.png"/>
  </div>
  <span>
    baz
  </span>
</div>
```

```
<div>
  <div>
    bar
    <img src="2.png"/>
  </div>
  <div>
    <span>text1</span>
    text2
  </span>
</div>
```

(a)

(b)

(c)

```
<component>
① <div>
②   <div>
③     {arg0}
④     <img src={arg1}>
    </div>
⑤   {arg2}
  </div>
</component>
```

(d)

```
<component args=["foo", "1.png", "<span>baz</span>"] />
<component args=["bar", "2.png", <div><span>text1</span>text2</div>"] />
```

(e)

Figure 5: (a) Initial HTML code fragments. (b) Corresponding DOM subtrees. (c) The constructed Component Intermediate Model. (d) The final generated UI component. (e) The *calls* to the generated UI component.

DOM subtrees, and the constructed Component Intermediate Model for these subtrees are respectively shown in Figure 5(b) and (c). The connected DOM nodes with dotted arrows form Mapping Nodes Sets. Notice that non-mapping DOM nodes do not form a node in the intermediate model. This model can be translated to a React-like component similar to what is shown in Figure 5(d). Finally, the generated component is instantiated two times in the refactored HTML code to replace the originally-repeated DOM nodes. The calls to the component can look like what is shown in Figure 5(e).

When generating the actual framework code, each model node results into a DOM node in the framework component (as depicted in Figure 5(d)), which essentially *unify* the nodes in the Mapping Nodes Set to remove duplication. There are three possibilities for framework component DOM nodes:

- When all pairs of DOM nodes in a Mapping Nodes Set have the same tag and identical attribute values, they can be unified in one DOM node of the same tag. For example, the two `<div>` nodes in Figure 5 corresponding to model node ❶ form a `<div>` node in the component.
- A pair of DOM nodes in a Mapping Nodes Set which have different tag names cannot be unified into one DOM node in the component (e.g., `<span>` and `<div>` corresponding to model node ❺ in Figure 5). Similar is two text nodes with different content (e.g., the `foo` and `bar` corresponding to the model node ❸ in Figure 5). In such cases, the DOM nodes (and the whole subtree rooted at them) should be *hard-parameterized* in the resulting component, i.e., a *placeholder* should be created. The original parameterized DOM nodes are later passed as *arguments* when instantiating the component to recreate the original DOM hierarchy.
- A pair of DOM nodes in a Mapping Nodes Set that have the same tag name but different values for one of their attributes might be unifiable into a DOM node via *soft parameterization*, where the differing attribute values are parameterized (e.g., the `<img>` tags corresponding to model node ❹ in Figure 5, with parameterized `src` attribute values). This can be done only if the used framework supports parameterizing attribute values. Otherwise, the parameterization should be done as if it was a hard parameterization.

The intermediate model construction and refactoring algorithm. The inputs of Algorithm 1 are the original mockup HTML, the list of component instance DOM subtrees, and the translation strategy. The output is the refactored HTML wherein duplication is removed using the UI components.

Algorithm 1 Component Intermediate Model Generation

Input: The original DOM of the mockup ($DOM_{original}$), UI component instances DOM subtrees ($subtrees$), UI component translation strategy ($strategy$)

Output: The new DOM after refactoring ($DOM_{refactored}$)

```
 1: $model \leftarrow$ ConstructEmptyIntermediateModel()
 2: $coveredNodes \leftarrow \varnothing$
 3: $templateTree \leftarrow$ getSmallestTree($subtrees$)
 4: $templateNodes \leftarrow$ BFS($templateTree$)
 5: for $templateNode \in templateNodes \setminus coveredNodes$ do
 6:     $coveredNodes \leftarrow coveredNodes \cup \{templateNode\}$
 7:     $mappedNodes \leftarrow$ getMappedNodesSet($templateNode, subtrees$)
 8:     $parameterization \leftarrow$ NULL
 9:     for $currentNode \in mappedNodes \setminus coveredNodes$ do
10:         $parameterization \leftarrow$ compare($templateNode, currentNode$)
11:         if $parameterization \neq$ NULL then
12:             break
13:         end if
14:     end for
15:     $parent \leftarrow model$.getModelNodeFor($templateNode.parent$)
16:     if $parameterization \neq$ NULL then
17:         if $parameterization =$ SOFT_PARAMETERIZATION
                $\wedge strategy$.supportsAttributeParameters() then
18:             $model$.addSoftParamNode($parent, mappedNedesSet$)
19:             $coveredNodes \leftarrow coveredNodes \cup mappedNodes$
20:         else
21:             $model$.addHardParamNode($parent, mappedNedesSet$)
22:             $coveredNodes \leftarrow coveredNodes \cup$
                    GetAllSubtreeNodes($mappedNodes$)
23:         end if
24:     else
25:         $model$.addNonParamNode($parent, mappedNedesSet$)
26:         $coveredNodes \leftarrow coveredNodes \cup mappedNodes$
27:     end if
28: end for
29: $DOM_{refactored} \leftarrow strategy$.refactor($DOM_{original}, model$)
```

Algorithm 1 starts by constructing an empty model (line 1), and an exclusion list (`coveredNodes` in line 2) that contains the original DOM nodes of the component instances which are already covered by the algorithm (e.g., a model node has been created for them), so that they are skipped in future iterations. To construct the intermediate model, the algorithm chooses the DOM subtree of one of the component instances (i.e., the *template subtree*) to follow its hierarchy. The template subtree is the one with the smallest number of DOM nodes, chosen in line 3. This is because the intermediate model cannot have more DOM nodes than the smallest subtree, as it resembles the *intersection* of the component instances' DOM subtrees. The algorithm loops over all the uncovered template subtree's DOM nodes, following the subtree's breadth-first traversal order (lines 5 to 28). Each template DOM node is compared to other DOM nodes of its Mapping Nodes Set (identified according to Definition 4 in line 7) using the `compare()` function (line 10), which returns the type of parameterization needed to unify two given DOM nodes, and `NULL` if no parameterization is needed. Note that, even if one node in a Mapping Nodes Set should be parameterized when compared to the template DOM node, the resulting model node will be either hard- or soft-parameterized, thus comparing other nodes of Mapping Nodes Set is not required (line 12).

The intuition behind comparing nodes in the breadth-first order is that, across the component instances' DOM subtrees, it is more likely that the the inner nodes (which define the structure of the final UI component) are similar, while the leaf nodes (texts, images) are more probable to differ. The inner nodes are thus compared before leaves, also facilitating the identification of Mapping Nodes Set based on Definition 4, as the nodes' child indices follow the BFS traversal order.

The algorithm then continues to add a model node for each Mapping Nodes Set (lines 15 to 27). First, the model node that has been created for the template DOM node's parent (in the previous runs of the loop) is retrieved from the model (line 15), to which the new model nodes will be added as children. This effectively allows the model to preserve the original hierarchy of the instances' DOM subtrees. If the model is empty, the new model node will form the model's root. The subsequent lines of the algorithm add the new model node based on the parameterization type. In each step, the DOM nodes in the Mapping Nodes Set for which a model node is created are added to the `coveredNodes` to be skipped in the next iterations. As mentioned, in case of a hard-parameterized model node, all the DOM nodes belonging to the subtrees rooted under the corresponding mapping DOM nodes should be marked to be skipped (e.g., node ❺ in Figure 5).

Finally, the actual refactoring is conducted using the constructed Component Intermediate Model (line 29). The details of the refactoring are built-in the translation strategy, which can be implemented virtually for any UI framework of interest.

3.6 Implementation

We implemented the proposed approach in a tool called VizMod [36] (short for **Vis**ual **Mod**ularizer). VizMod is implemented in Java and Python 3. We use the Selenium web driver to view the mockup and extract DOM trees and their relevant computed properties. For clustering, we use the implementation provided by Campello et. al. [5] and the `numpy` [56] library for mathematical and numerical functions.

4 EVALUATION

To evaluate VizMod, we conducted qualitative and quantitative studies aiming at answering the following research questions:

RQ1 Are the refactorings by VizMod's component generation correct?

RQ2 How effective is VizMod in identifying UI components compared to manual examination by web developers?

RQ3 How much code reusability can be achieved through the proposed refactorings?

In the following subsections, we discuss the details of the experiments that we designed to answer each research question, together with the results.

4.1 RQ1: Correctness of Component Generation Refactorings

4.1.1 Study Design. For the proposed componentization applied on HTML to be safe, the main criterion is that the original and the refactored HTMLs must result into the same DOM tree landed into the users' web browsers. Consequently, to devise a technique that can automatically assess the safety of the applied transformations, we relied on the equivalence of the DOM subtrees rendered in the web browser, before and after refactoring. If the DOM trees are the same, given that our refactorings do not change any CSS style rules, the resulting presentation semantics of the HTML files remain intact.

To automate this process, we serialized the final DOM trees rendered in the browser to the pretty-printed HTML code and compared them pre- and post-refactoring. This allows a fast comparison of the structure of the DOM trees. We normalized the DOM trees by removing text nodes which are empty or contain only white spaces. This is done because React interprets these nodes differently [18, 46] compared to the standard HTML specifications.

4.1.2 Results and Discussion. Using the aforementioned technique, we compared the DOM subtrees of the UI component instances before and after refactoring for the 120 UI component instances (i.e., 25 UI components) identified by VizMod. The tests has passed for all subjects, indicating that the refactorings introduced by component generation do preserve the DOM trees, and as a result, the transformations are safe to apply.

4.2 RQ2: UI Component Identification

4.2.1 Study Design. We asked independent expert web developers to participate in a qualitative study, with the goal of understanding what they would identify as being a component pattern in a web UI. With this study, we aim at evaluating the (dis)agreement between the proposed approach and expert developers in terms of identifying the UI components.

Subject Systems. The author(s) searched the Internet to find mockups suitable for this study (using keywords like "web mockups", "web templates", "front-end templates"). Our selection criteria for choosing mockups were:

- They should be non-trivial, both visually and code-wise (i.e., HTML and CSS). Note in Table 1 that the mockups are indeed complex, in terms of the number of DOM nodes and CSS code size.
- The number of subjects should be small and manageable enough so that we can ask participants to highlight potential components in *all* of them, without causing too much burden, mental fatigue, or boredom on them which can negatively distort the study.
- They should only represent the UI front-end, i.e., without back-end or front-end business logic or functionalities.

Based on these criteria we chose five mockups for our evaluation. We use the same mockups in all the evaluation experiments. Table 1 shows descriptive statistics for them.

Table 1: Subjects' descriptive statics

Subject#	Body size (KB)	#DOM nodes	CSS size (KB)[†]
1	18	754	323
2	20	915	279
3	33	1,990	350
4	24	1,226	330
5	49	1,065	254

[†] Some mockups use CSS frameworks (e.g., Bootstrap). This corresponds to the total CSS size, including the frameworks.

Participants. We emailed developers who have worked in local businesses or research labs and asked them to voluntarily participate in our study. We attached a zip package containing our subject systems together with a link to a post-study questionnaire aimed at collecting information about the participants' demographics (e.g., number of years of experience in software engineering in general and in web development in particular, and their self-assessment on web application development skills). We asked them to manually highlight repetitions on the UI of each subject (by drawing a rectangle on each repetition) and send the results back to us.

Accordingly, we emailed 10 developers and informed them of the study, and asked them to also pass it on to their contacts. We received responses from a total of five developers. Table 2 shows participants' demographic information. As it is shown, all the participants were quite experienced in web development, as measured by the years of software and web development experience and their own self assessment.

Comparison with Developers. For each subject, we compared the components highlighted by the experts with those components that VizMod automatically identified as UI components. In particular, when more than half of the experts highlighted a pattern on the mockup as repeated, we assume the majority is correct and consider it as a UI component that our technique should be able to identify. The performance of VizMod is then determined using the well-known *precision* and *recall* measures. A *true positive* for the approach is defined as a UI component that has been manually identified by more than half of the experts (in our case, three or more participants). A *false positive*, on the other hand, is a UI component that is reported by the approach, yet less than half of the experts have identified it. Finally, a *false negative* of the approach is a UI component that is reported by more than half of the experts, but the approach could not identify.

Table 2: Demographics of the Participants

Participant#	SW dev. (#Years)	Web dev. (#Years)	Web dev. self assessment (1–5, 5=Highly Expert)
1	10	5	4
2	8	2	4
3	3	3	4
4	11	3	3
5	9	8	5

4.2.2 *Results and Discussion.* Table 3 shows the results of comparing the UI patterns identified by our approach to the UI patterns identified by participating developers in our experiment. The table shows the values for true positives, false positives, false negatives, and finally precision and recall. The values for recall range between 74% and 100%. We examined the subjects at the lower end of the range to investigate further. Almost all the components that were missed by our approach had many elements that had animations or moving sub-elements (e.g., a carousel that changes every few seconds). Our technique was not designed with animations in mind. Capturing and analyzing animations can be challenging, due to difficulties in keeping track of changes over time and deciding which time instant to take as representative. This might be a possible venue for future work.

As for the precision, we further examined the nature of false positives in order to better understand the performance. Following this examination, we identified another variable while performing the comparison with participants: the *potentially missed opportunities*. We define missed opportunities as those patterns that were reported by as few as one developer (but not the majority), *as well as* our tool. The reason for introducing this variable is that, by manually examining the false positives, we noticed that there were a few potential opportunities that were missed by the majority of developers. We postulate a number of possible causes as to why such patterns were not reported by the majority of participants:

- Some repeating components were laid out far away from each other (e.g., at the very top and very bottom of the page). This often makes it difficult for human developers to remember patterns that are not immediately visible within the same view, especially if there are lots of patterns that they have to keep track of. The human brain has been shown to have a short-term memory capacity of only around 3 to 7 objects at a time [8]. This fact, coupled with patterns that are far away from each other and interlaced with multiple other patterns, can cause humans to miss some patterns. Our approach, however, is agnostic to where the pattern is located, and is able to recognize matching patterns from far ends of a web UI just as easily as patterns immediately next to each other.
- Some components included images or icons that were designed to be faint or barely visible due to artistic reasons. Such icons, especially when present close to very vibrant and large repeating components, are often skipped potentially due to the visual attention in the brain being directed at the larger clearer patterns. However, due to the visual normalization adopted in our approach, such artistic choice do not make any difference and the pattern is recognizable regardless of how visually pronounced it is.

For these reasons, we believe that the potentially missed opportunities are a better indicator of performance and we include their numbers in the table. Therefore, we finally indicate that the precision and recall performance of the tool are 93.6% and 75.2%, respectively, as shown in Table 3.

4.3 RQ3: Code Reusability

4.3.1 *Study Design.* We now proceed to determine how much code reusability can be attained with the components generated by the

Table 3: Comparison of automatically identified components to manually-identified ones by developers

Subject	#Identified Refactoring Opportunities		FN	TP	FP	Precision	Recall	Considering PMO†		
	#Components	#Component Instances						#PMO	Precision	Recall
1	5	29	0	19	9	67.9%	100.0%	8	96.4%	77.1%
2	5	27	0	15	2	88.2%	100.0%	2	100.0%	77.3%
3	5	26	6	17	5	77.3%	73.9%	0	77.3%	68.0%
4	6	21	3	9	12	42.9%	75.0%	12	100.0%	84.0%
5	4	17	3	13	4	76.5%	81.3%	3	94.1%	69.6%
Avg.	**5**	**22**				**70.5%**	**86.0%**	**5**	**93.6%**	**75.2%**

† PMO = Potentially Missed Opportunities.

approach. For each test subject, we compare the size of the HTML code of the mockups before and after refactoring as a measure of how much reusability has been achieved.

Let T_r be the set of DOM subtrees corresponding to the UI component instances which are going to be removed by a refactoring operation, r, from the original HTML. The refactoring r adds the necessary code which unifies T_r subtrees into a UI component u_r to the original HTML code. Moreover, r replaces T_r subtrees with a set C_r of calls to instantiate u_r. Accordingly, the size reduction SR_r for the refactoring r is computed as:

$$SR_r = \sum_{t \in T_r} sizeOf(t) - sizeOf(u_r) + \sum_{c \in C_r} sizeOf(c) \tag{6}$$

where t is a component instance, c is a component instantiation call, and $sizeOf(x)$ is the number of bytes corresponding to x when serialized to HTML. In an HTML mockup, there might be several sets of component instances (i.e., several UI components might be created). The overall size reduction SR, which is achieved by applying the set R of all refactoring opportunities found in a mockup, is calculated as $SR = \sum_{r \in R} SR_r$.

We calculate the size reduction in two different ways: 1) based on an implementation using a UI framework (which we have chosen to be React), and 2) based on the representation contained in the Component Intermediate Model. This is because each UI framework (e.g., React, Angular) has its own syntax and idiomatic mechanisms for creating UI components and instantiating them. As a result, the actual size reduction would be different depending on which, and how, a UI framework is used. All UI frameworks, however, follow the same basic principle: the set of DOM nodes that can be unified into single DOM nodes form a *template* for the UI component, while other nodes form the *parameters* (i.e., *placeholders*) in the UI component. These placeholders are filled with the *arguments* passed when calling the UI framework. As a result, calculating the size reduction based on the nodes and arguments identified when constructing the Component Intermediate Model allows a more accurate determination of how the algorithm *intrinsically* performs in terms of code reusability, regardless of the differences between the many possible UI frameworks that can be used.

Moreover, when using a third-party UI framework, it is usually required that the framework's JavaScript library code is imported at the client-side so that the web browser is able to render the UI, potentially increasing the overall size of the client-side code.

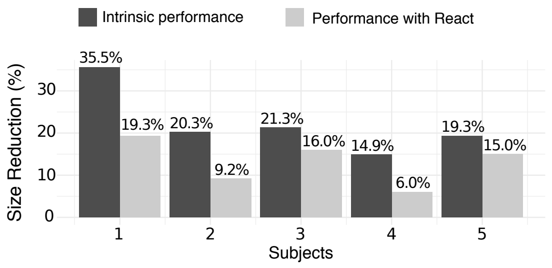


Figure 6: Code reusability performance achieved by the proposed component generation, as measured by final size reduction.

However, if the web application wants to enjoy the maintainability benefits of the UI framework, the JavaScript files should be imported anyway. As mentioned, this is an extensively-popular trend among the developers [52, 53]. Notwithstanding, if developers opt for using standard HTML Web Components [38] instead of third-party UI frameworks, there will be no burden in terms of the additional imported JavaScript files. As a result, when reporting the size measurements for React, we only consider the code generated by our approach for implementing UI components, not React's own core JavaScript code.

4.3.2 Results and Discussion. Figure 6 illustrates the results of applying the proposed refactorings on the test subjects. Observe that, using React implementation, refactoring UI components results in reducing the size of the HTML code by 6%–19.34%, with an average of 11.56%. The *intrinsic* performance of the algorithm itself, however, is higher: 14.90%–35.54%, with an average of 18.96%. This difference highlights that React components require quite considerable amount of added code to the original DOM information of the UI component instances. For example, a UI component shown in Figure 5(d) needs to be wrapped into a function named `render` implemented in a JavaScript class that extends the internal React class, `React.Component`. We also need to add additional code to pass arguments to the UI components for each UI component instance. As mentioned, using another UI framework can yield different saving ratios. It is for these reasons that reporting the intrinsic performance is important.

It is worth mentioning that this saving is not meant to replace existing techniques that, for instance, minify HTML by removing white space, or via any other reduction approach. Rather, whatever savings obtained from the components can complement them by adding even more saved bytes on top of what they would normally save.

5 DISCUSSION

Context within web development. The approach we present in this paper refactors repetitions in the UI and merges them into a template, which is finally converted into a component in one of the common front-end frameworks (e.g., React, Angular). This automates one of the initial steps in making a full-fledged web app UI, which is often time consuming and is done manually. The use of this approach, of course, would not mean that the web app is ready to launch to clients and the development process is finished. The developer would use these components and continue the development, by, e.g., adding business logic, handling events, connecting to databases or other sources. Furthermore, the approach we present in this paper is for modularizing the UI view itself, and is therefore orthogonal to the remaining components of the architecture pattern of the app (e.g., Model-view-controller (MVC), Model-view-presenter (MVP)) and to any backend server functionality.

Threats to validity. We chose test subjects (i.e., mockups) randomly from the Internet with the mentioned criteria in Section 4.2.1, to avoid any selection bias. Plus, the evaluation participants are expert web developers with different years of web development experience, mitigating the threats to the internal validity of the study. The mockups are diverse and complex enough to be representative of real-world app front-ends, mitigating the external validity of the study by making the results generalizable. To make the study replicable, we have made VizMod's source code, evaluation subjects, and the anonymized participants' responses available online [36].

6 RELATED WORK

Visual analysis. There exist a few techniques that analyze web applications from a visual perspective. Choudhary et al. [6] propose an approach that detects cross-browser compatibility by examining visual differences between the same app running in multiple browsers. Burg et al. [3] present a tool that helps developers understand the behavior of front-end apps. It allows developers to specify which element they are interested in, then tracks that element for any visual changes and the corresponding code changes. Bajammal et al. [1] propose an approach to analyze and test web canvas element through visual inference of the state of the canvas and its objects, and allowing canvas elements to be testable using common DOM testing approaches. In contrast to our work, none of these studies aims to automatically identify and extract web components. Stocco et al. [22, 54] explore visual techniques for web testing applications, including visual-based test repair and techniques for migrating DOM-based tests to visual tests.

Clone detection. There is a large body of work on clone detection in conventional source code [45, 48, 50]. Some techniques also exist targeting web artifacts, such as for identifying duplicated content [2] or script function clones [4, 21], and quantifying the structural similarity across pages [10]. A number of existing publications [12, 24, 25, 29] propose template identification for Java code by defining a number of heuristics to compute code similarity. Rajapakse and Jarzabek [42] use CCFinder [20] to identify duplication in web applications. Synytskyy et al. [55] use an *island grammer* to identify cloned HTML forms and tables. Cordy et al. [7] propose a language-independent technique to identify exact/near-miss clones (initially in HTML) using island grammars, pretty-printing and textual differencing. Inspired by that work, NiCad clone detector is proposed [49].

Transformation and refactoring. Various techniques are proposed to convert static pages to dynamic ones [2, 55], to generalize dynamic web pages [9, 43], or to find similar functionalities across web pages [10]. Other techniques [34] use clustering to group similar static web pages together to extract single-page templates. Pattern mining techniques are used [30–32] for identifying and refactoring duplicated CSS code in web apps. In contrast to our work, none of these studies aims at automatically identifying and extracting web components from mockups.

7 CONCLUSIONS

The development of a web app front-end involves multiple stakeholders, chief among them the graphics designer and web developer. A UI mockup designed by the graphics designer has to be analyzed and processed by a web developer in order create the app's front-end code, a task that is laborious and involves manual time consuming steps. In this paper, we proposed an approach to automate this aspect of web development by generating reusable web components from a mockup. We implemented our approach in a tool called VizMod, and evaluated on real-world web mockups and assessed its generated components through comparison with expert developers. It achieves an average of 94% precision and 75% recall in terms of agreement with the developers' assessment, performs the refactorings in a correct manner, and the components achieve a 22% reusability, on average.

REFERENCES

[1] Mohammad Bajammal and Ali Mesbah. 2018. Web Canvas Testing through Visual Inference. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*. IEEE Computer Society, 12 pages.

[2] Cornelia Boldyreff and Richard Kewish. 2001. Reverse engineering to achieve maintainable WWW sites. In *Proceedings of the 8th Working Conference on Reverse Engineering (WCRE)*. 249–257.

[3] Brian Burg, Andrew J Ko, and Michael D Ernst. 2015. Explaining visual changes in web interfaces. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*. ACM, 259–268.

[4] Fabio Calefato, Filippo Lanubile, and Teresa Mallardo. 2004. Function clone detection in web applications: a semiautomated approach. *Journal of Web Engineering* 3, 1 (2004), 3–21.

[5] Ricardo JGB Campello, Davoud Moulavi, and Jörg Sander. 2013. Density-based clustering based on hierarchical density estimates. In *Pacific-Asia conference on knowledge discovery and data mining*. Springer, 160–172.

[6] Shauvik Roy Choudhary, Mukul R Prasad, and Alessandro Orso. 2012. Crosscheck: Combining crawling and differencing to better detect cross-browser incompatibilities in web applications. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*. IEEE, 171–180.

[7] James R Cordy and Thomas R. Dean. 2004. Practical language-independent detection of near-miss clones. In *Proceedings of the 14th Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*. 1–12.

[8] Nelson Cowan. 2001. The magical number 4 in short-term memory: A reconsideration of mental storage capacity. *Behavioral and Brain Sciences* 24, 1 (2001), 87–114.

[9] A. De Lucia, R. Francese, G. Scanniello, and G. Tortora. 2004. Reengineering web applications based on cloned pattern analysis. In *Proceedings of 12th IEEE International Workshop on Program Comprehension*. IEEE, 132–141.

[10] A. De Lucia, Rita Francese, G. Scanniello, and G. Tortora. 2005. Understanding cloned patterns in web applications. In *Proceedings of the 13th International Workshop on Program Comprehension (ICPC)*. IEEE, 333–336.

[11] Donis A Dondis. 1974. *A primer of visual literacy*. MIT Press.

[12] Stefan Fischer, Lukas Linsbauer, Roberto Erick Lopez-Herrejon, and Alexander Egyed. 2014. Enhancing clone-and-own with systematic reuse for developing software variants. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*. IEEE, 391–400.

[13] Google Inc. 2016. Angular. https://angular.io/ Accessed: 15 February 2018.

[14] Google Inc. 2017. Angular Core Documentation: Components. https://angular.io/api/core/Component Accessed: 4 April 2018.

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

[16] Jordan Walke, Facebook, Instagram and community. 2013. React - A JavaScript library for building user interfaces. https://reactjs.org/ Accessed: 15 February 2018.

[17] Jordan Walke, Facebook, Instagram, and community. 2013. React documentation: React.Component. https://reactjs.org/docs/react-component.html Accessed: 4 April 2018.

[18] Jordan Walke, Facebook, Instagram and community. 2014. JSX Whitespace. https://reactjs.org/blog/2014/02/20/react-v0.9.html#jsx-whitespace Accessed: 16 April 2018.

[19] Elmar Juergens, Florian Deissenboeck, Benjamin Hummel, and Stefan Wagner. 2009. Do code clones matter?. In *Proceedings of the 31st International Conference on Software Engineering (ICSE)*. 485–495.

[20] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 2002. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. on Software Engineering* 28, 7 (2002), 654–670.

[21] Filippo Lanubile and Teresa Mallardo. 2003. Finding function clones in web applications. In *Proceedings of the 7th European Conference on Software Maintenance and Reengineering (CSMR)*. 379–386.

[22] Maurizio Leotta, Andrea Stocco, Filippo Ricca, and Paolo Tonella. 2018. PESTO: Automated migration of DOM-based Web tests towards the visual approach. *Software Testing, Verification And Reliability* 28, 4 (2018), e:1665.

[23] William Lidwell, Kritina Holden, and Jill Butler. 2010. *Universal principles of design, revised and updated*. Rockport Pub.

[24] Yun Lin, Guozhu Meng, Yinxing Xue, Zhenchang Xing, Jun Sun, Xin Peng, Yang Liu, Wenyun Zhao, and Jinsong Dong. 2017. Mining implicit design templates for actionable code reuse. In *Automated Software Engineering (ASE), 2017 32nd IEEE/ACM International Conference on*. IEEE, 394–404.

[25] Yun Lin, Xin Peng, Zhenchang Xing, Diwen Zheng, and Wenyun Zhao. 2015. Clone-based and interactive recommendation for modifying pasted code. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 520–531.

[26] Guang-Hai Liu, Lei Zhang, Ying-Kun Hou, Zuo-Yong Li, and Jing-Yu Yang. 2010. Image retrieval based on multi-texton histogram. *Pattern Recognition* 43, 7 (2010), 2380–2389.

[27] Nuno Vieira Lopes, Pedro A Mogadouro do Couto, Humberto Bustince, and Pedro Melo-Pinto. 2010. Automatic histogram threshold using fuzzy measures. *IEEE Transactions on Image Processing* 19, 1 (2010), 199–204.

[28] Angela Lozano and Michel Wermelinger. 2008. Assessing the effect of clones on changeability. In *Proceedings of the 24th IEEE International Conference on Software Maintenance (ICSM)*. 227–236.

[29] Jabier Martinez, Tewfik Ziadi, Tegawende F Bissyande, Jacques Klein, and Yves Le Traon. 2015. Automating the extraction of model-based software product lines from model variants (T). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 396–406.

[30] Davood Mazinanian and Nikolaos Tsantalis. 2016. Migrating Cascading Style Sheets to Preprocessors by Introducing Mixins. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE) (ASE 2016)*. 672–683.

[31] Davood Mazinanian and Nikolaos Tsantalis. 2017. CSSDev: Refactoring duplication in Cascading Style Sheets. In *Proceedings of the 39th International Conference on Software Engineering (ICSE) Companion (ICSE 2017)*. 4.

[32] Davood Mazinanian, Nikolaos Tsantalis, and Ali Mesbah. 2014. Discovering Refactoring Opportunities in Cascading Style Sheets. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. 496–506.

[33] Philip B. Meggs. 1992. *Type and Image: The Language of Graphic Design*. Van Nostrand Reinhold. 206 pages.

[34] Ali Mesbah and Arie van Deursen. 2007. Migrating Multi-page Web Applications to Single-page Ajax Interfaces. In *Proceedings of the Conference on Software Maintenance and Reengineering (CSMR)*. IEEE Computer Society, 181–190.

[35] Bogdan Mihalcea. 2014. User interface construction with mockup images. US Patent 8,650,503.

[36] Mohammad Bajammal, Davood Mazinanian, and Ali Mesbah. 2018. VizMod tool repository. https://github.com/msbajammal/vizmod

[37] Manishankar Mondal, Chanchal K. Roy, and Kevin A. Schneider. 2012. An empirical study on clone stability. *Applied Computing Review* 12, 3 (Sept. 2012), 20–36.

[38] Mozilla Developer Network. 2017. Web Components. https://developer.mozilla.org/en-US/docs/Web/Web_Components Accessed: 4 April 2018.

[39] Brad A Myers and Mary Beth Rosson. 1992. Survey on user interface programming. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM, 195–202.

[40] Mark W. Newman and James A. Landay. 2000. Sitemaps, Storyboards, and Specifications: A Sketch of Web Site Design Practice. In *Proceedings of the 3rd Conference on Designing Interactive Systems: Processes, Practices, Methods, and Techniques (DIS)*. 263–274.

[41] Fatih Kursat Ozenc, Miso Kim, John Zimmerman, Stephen Oney, and Brad Myers. 2010. How to Support Designers in Getting Hold of the Immaterial Material of Software. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI)*. 2513–2522.

[42] Damith C. Rajapakse and Stan Jarzabek. 2005. An Investigation of Cloning in Web Applications. In *Proceedings of the 5th International Conference of Web Engineering (ICWE)*. 252–262.

[43] Damith C. Rajapakse and Stan Jarzabek. 2007. Using Server Pages to Unify Clones in Web Applications: A Trade-Off Analysis. In *Proceedings of the 29th International Conference on Software Engineering (ICSE)*. 116–126.

[44] Óscar Sánchez Ramón, Jesús Sánchez Cuadrado, Jesús García Molina, and Jean Vanderdonckt. 2016. A layout inference algorithm for Graphical User Interfaces. *Information and Software Technology* 70 (2016), 155–175.

[45] Dhavleesh Rattan, Rajesh Bhatia, and Maninder Singh. 2013. Software clone detection: A systematic review. *Information and Software Technology* 55, 7 (2013), 1165 – 1199.

[46] React Issues on GitHub. 2017. White-space between inline elements #1643. https://github.com/facebook/react/issues/1643 Accessed: 16 April 2018.

[47] ReactJS. 2018. Thinking in React. https://reactjs.org/docs/thinking-in-react.html Accessed: 15 February 2018.

[48] Chanchal Kumar Roy and James R Cordy. 2007. *A survey on software clone detection research*. Technical Report. Queen's School of Computing.

[49] Chanchal K. Roy and James R. Cordy. 2008. NiCad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *Proceedings of the 16th IEEE International Conference on Program Comprehension (ICPC)*. 172–181.

[50] Chanchal K. Roy, James R. Cordy, and Rainer Koschke. 2009. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming* 74, 7 (2009), 470 – 495.

[51] Nishant Sinha and Rezwana Karim. 2013. Compiling Mockups to Flexible UIs. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. 312–322.

[52] StackOverflow. 2017. Developer Survey Results. https://insights.stackoverflow.com/survey/2017 Accessed: 4 April 2018.

[53] stateofjs.com. 2017. Worldwide usage of JavaScript front-end libraries. https://stateofjs.com/2017/front-end/results/ Accessed: 10 April 2018.

[54] Andrea Stocco, Rahulkrishna Yandrapally, and Ali Mesbah. 2018. Visual Web Test Repair. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*. ACM.

[55] Nikita Synytskyy, James R. Cordy, and Thomas R. Dean. 2003. Resolution of static clones in dynamic Web pages. In *Proceedings of the 5th IEEE International Workshop on Web Site Evolution (WSE)*. 49–56.

[56] Stéfan van der Walt, S Chris Colbert, and Gael Varoquaux. 2011. The NumPy array: a structure for efficient numerical computation. *Computing in Science & Engineering* 13, 2 (2011), 22–30.

[57] Tiantian Wang, Mark Harman, Yue Jia, and Jens Krinke. 2013. Searching for Better Configurations: A Rigorous Approach to Clone Evaluation. In *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. 455–465.