UX-Painter: Fostering UX improvement in an Agile Setting

Juan Cruz Gardey[1,2], Julián Grigera[1,2,3], Gustavo Rossi[1,2], and Alejandra Garrido[1,2]

[1] LIFIA, Fac. Informática, Univ. Nac. de La Plata, La Plata, Argentina
{jcgardey,juliang,gustavo,garrido}@lifia.info.unlp.edu.ar
[2] CONICET, Argentina
[3] CIC-PBA, Pcia. Buenos Aires, Argentina

Abstract. It is generally difficult in agile teams, specially those geographically distributed, to keep up with the user experience (UX) issues that emerge on each product increment. UX designers need the help of developers to set up user testing environments and to code improvements to the user interface, while developers are too busy with functionality issues. This paper describes a tool called UX-Painter and shows through a case study, how it may help in the above setting to synchronize UX practices and allow for continuous UX improvement during an agile development. UX-Painter allows designers to set up A/B testing environments, exploring interface design alternatives without the need of programming skills, through predefined transformations called client-side web refactorings. Once a design alternative is selected to be implemented in the application's codebase, UX-Painter may also facilitate this step, exporting the applied refactorings to different frontend frameworks. Thus, we foster a method where UX backlog items can be systematically tackled and resolved in an agile setting.

Keywords: Agile Methods · User Experience · Web Engineering.

1 Introduction

User Experience (UX) is crucial for the success of web applications. Adopting a User-Centered Design (UCD) approach ensures that software products are analyzed, designed and evaluated pursuing a high usability and UX, by allocating a significant amount of resources to user research [12]. However, UCD practices, as many research studies have pointed out, do not integrate well with agile methods [4, 5, 12]. While agile methods pursue customer satisfaction, UCD focuses on the user needs [4, 12], but most importantly, UCD practices are too costly for agile teams, which usually cannot allot time for UX improvement during agile cycles. Recent methods, like Lean UX, aim at incorporating user research and Design Thinking practices into agile software development through a high degree of collaboration among UX designers and developers in a team [11]. Moreover, there are several artifacts being used to promote collaboration and communication among team members in the early phases of product design [8]. However,

there is still a gap in practices and artifacts at late stages, when UX issues on already deployed product increments appear and UX improvement should take place, i.e., when solutions to existing UX issues must be evaluated, compared, communicated to developers and implemented [9].

In a previous work, we have developed UX-Painter, a visual programming tool for UX designers to apply and compose small changes to the client-side of a web application to set up alternative designs without the need of any script programming knowledge [9]. The building blocks to create alternative designs in UX-Painter are Client-Side Web Refactorings (CSWRs). A CSWR is a predefined transformation on a webpage element or interaction, which is intended to solve a specific UX issue while preserving the underlying functionality. UX-Painter allows designers to quickly set up new versions of a production web application by combining CSWRs to create alternative versions. Thus, in the context of an agile setting, while developers are taking care of sprint backlog issues, UX designers may use UX-Painter to set up A/B testing environments to evaluate alternative fixes to the UX problems that appeared on the product increment of the preceding sprint [7].

The refactorings of UX-Painter are developed as scripts that perform alterations on a rendered web page. The set of applied refactorings are saved as application versions that can be accessed at any time and exported to be re-created in others browsers. In this way, at the time of testing a particular version, a designer only needs to share a file that tests subjects may load and run remotely in their browsers.

Finally, once a particular version of the web application is selected after testing, it must be added to the product backlog for its implementation in the next sprint. In our previous work, developers had to code each CSWR applied on the version in the codebase [9], which could again be costly and thus left unattended. With the goal of facilitating this process, in this paper we propose an extension to UX-Painter that may generate a preliminary implementation of the refactorings for the libraries and frameworks used nowadays to build web user interfaces. Developers then may adapt this source code to fit it in the target application. Although this implementation depends on the existing code, our case study shows that it can reduce the load on developers to remediate UX issues.

We next describe some related work; then we show UX-Painter in action and present an extension which allows exporting the applied CSWRs to ReactJS[4], one of the most popular front-end libraries. We finish with a conclusion and future work.

2 Related Work

By their nature, agile methods have always tried to involve users into the process. Consequently, incorporating UX aspects is an issue that both the academy and industry have studied for years [4]. The lack of support for including and

[4] https://reactjs.org

tracing UX requirements in a systematic and coordinated way has been noted by different authors [5, 2]. In a recent study on an agile team in UK, Zaina et al. [13] found many problems in the UX information flow, such as user perspective aspects not being captured with artefacts, but rather verbally.

To bridge the gap between development team and UX designers, prototyping has been widely used as an effective communication device between development teams, UX professionals, and end users. Modern applications like Figma or InVision allow creating high fidelity prototoypes that even allow basic interactions, being an effective way to get a feel of the final product. However, they do not support making quick changes to already deployed user interfaces (UIs), or creating prototypes on top of them.

This need for quickly altering running UIs has been tackled in different research works. The field of web augmentation proposes making client-side alterations for specific viewpoints or contexts, even in 3rd party applications [1]. For example, Ghiani et al. [10] use web augmentation to adapt the UI to different context of use (technology, users or environment). Other approaches propose empowering end-users, allowing them to create personalized contents such as WebMakeUp [6]. Our work can be considered a web augmentation proposal but with the focus on improving the UX rather than personalizing content or adapting an interface to different contexts.

The automatic generation of code to facilitate the UI development also has been under-analysis [3]. The difference is that they intend to derive the code of the complete UI from mockups, which is useful in early stages, but not for incremental adjustments required to fix UX problems at later stages.

3 UX-Painter in Action

UX-Painter is a web-extension to perform changes to the websites through the assisted application of Client-Side Web Refactorings. The alternatives designs generated can then be saved as application's versions for further evaluations such as A/B testing or inspection reviews [9].

In the context of agile approaches, UX-Painter is useful to mediate collaboration between developers and designers to improve the UX of a product increment that has been developed. During the current sprint, designers may want to explore design alternatives for some issues found, for instance, in the review meeting of the previous sprint [7]. Thus, designers can use UX-Painter to dispense developers from coding changes. Developers instead can focus on adding new functionality from the product backlog.

To show a concrete example of the tool usage, suppose that a team is working on an e-commerce web application[5] using ReactJS for the front-end. In the previous sprint, developers worked on the checkout process, building the UI shown in Fig. 1. The UX team decided to use UX-Painter to inspect some design changes.

[5] https://github.com/bradtraversy/proshop_mern

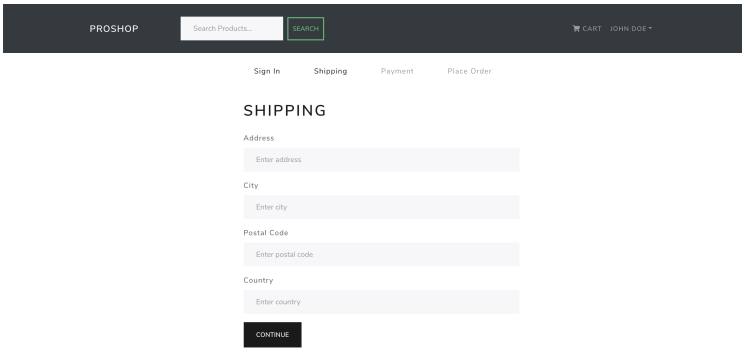

Fig. 1. Checkout form of an e-commerce web application

For example, the shipping form does not provide a client validation. Even when the user submits the form without filling in any field, the information is sent to the server for its validation. In order to minimize failed form submissions because of incomplete information, a prior validation can be added to check mandatory fields. The CSWR that includes this feature is *Add Late Form Validation*.

The changes are made on a specific version, so the first step to apply a refactoring is to create a new application's version (see Fig. 2). When the new version is edited, all the available CSWRs are listed. Once a refactoring is selected, the tool guides the user in the application process. The first step is to select the element to be refactored (the form) directly over the target page. After that, the tool may request additional parameters to complete the refactoring execution. *Add Late Form Validation* requires the fields to be checked when the form is submitted. Finally, a preview of the refactoring's result is shown where the user can confirm or cancel it, and choose the appropriate style. In this case the preview allows to observe that when the submit button is clicked, a red border is added to the empty fields that are mandatory.

UX-Painter gives the possibility to combine different refactorings to produce larger design changes. In the form previously refactored, supposing that the shipping is only available for a specific set of countries, an alternative for country's text field is to turn it into a select box through *Turn Input into Select* to choose the country from a predefined options list. The application process for all the refactorings is similar, the user must follow the same steps: select the target element, complete refactoring-specific parameters and confirm the preview.

Postal code field can also be changed to improve the user interaction. In particular, assuming that postal codes have only numbers and at most 5 digits, with *Format Input* a mask can be applied that helps to prevent the format errors that may arise. Moreover, through *Resize Input* the field can be narrowed to give the users a hint of the expected input length.

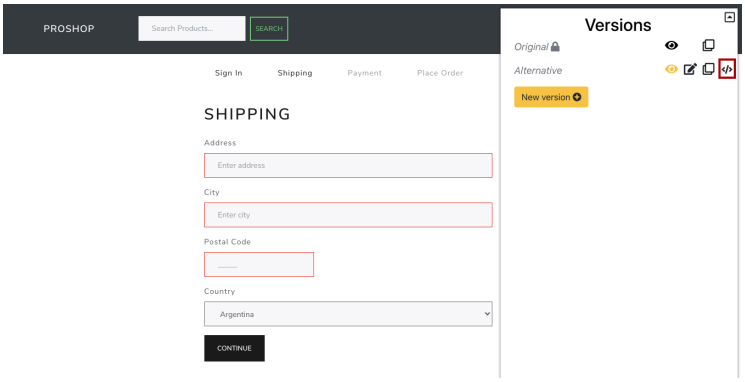

Fig. 2. Alternative version of the shipping form with some refactorings applied. The "eye" icons allow to show the target version.

The new version of the shipping form is shown in Fig. 2, besides UX-Painter's versions menu. The designer must save the version with all the applied CSWRs to persist the changes. The tool saves in the browser's local storage the information needed to recreate each CSWR in future page's visits. Anytime, the designer can choose which version to see to compare the differences.

The next step for the UX team is to assess if the new version generated really causes an improvement for final users. For instance, designers can run a user test with some subjects to analyze how they interact with the modified shipping form. In order to facilitate remote user testing, UX-Painter allows to export a version through a JSON file. Then, subjects can import the generated file in UX-Painter to load the shared version in their browser.

Finally, if the new version works better than the previous one, the UX team has to communicate the changes to developers to be implemented in the application's codebase during the following sprint. At this point, generating a preliminary version of the refactorings' source code can help developers to reduce the effort required for the implementation. In this way, they can focus on the following product increment without leaving aside the UX improvement of the product increment already developed. The next section describes the CSWR's code generation.

4 Implementing Refactorings

This section shows how the refactorings explored before are implemented in the application used in the case study. Using UX-Painter, it is possible to automatically generate a basic implementation of the refactorings that developers might adapt or refine to add it to the application's codebase.

In order to generate the code for a specific created version, the user must click on the "source-code" icon (highlighted with a red box in the top right of Fig. 2). Next, the tool shows a list of the generated ReactJS components.

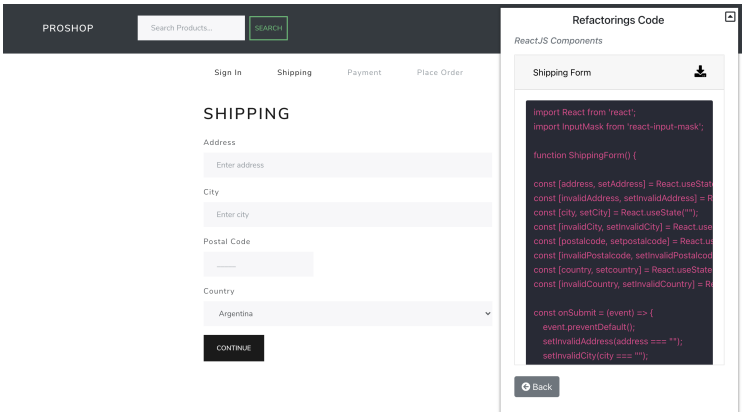

Fig. 3. ReactJS component generated for the alternative version. The user can observe a preview before downloading it.

Components are the building blocks of a ReactJS application; they are basically JavaScript functions that define reusable pieces of the user-interface. For the version described in the previous section, Fig. 3 shows that there is only one component that contains the implementation for the whole shipping form. Since there are CSWRs that apply very small changes, it may not be realistic to create a different component for each refactoring. Instead, the tool looks for high-level elements that were refactored like a form, and creates a component with all the refactorings included in them. Each component can be previewed and downloaded as a JavaScript file (see Fig. 3).

Once the generated code is downloaded, it should be manually merged with the application's codebase. This process depends on different factors of the codebase such as how the UI is divided into different components, the dependencies used, among others. Thus, the goal of UX-Painter is not to give a full implementation of the CSWRs, but to provide a potential solution that developers can use as a starting point to integrate these refactorings into the target application.

The case study used in this paper was selected from Github to avoid any bias of coding the case study ourselves, improving external validity. The results of using UX-Painter to generate the refactored code were quite promising, as code merging was straightforward. In this case, the structure of the generated component for the shipping form is similar to that of the codebase. Figure 4 shows a comparison of them. Some parts were not included to avoid making the code more extensive. Highlighted lines contain CSWRs' code. In general, these lines could be added directly to the codebase without any modification, except for the country's select box that is implemented through a component defined by Bootstrap[6], the library that provides a set of components used by the application. The result of the merging process is shown in Fig. 5.

[6] https://react-bootstrap.github.io

Further experiments are necessary, as these results are limited to a single case study, and the selected application is small-sized (so the code could fit in the available space). Although other cases may show that the refactored code offered by UX-Painter is not entirely suitable or difficult to merge, we believe that the tool is helpful for both designers and developers, for experimenting with ready-to-apply solutions to UX problems, for communicating this solutions to the whole team, and to provide at least good hints of how to code them.

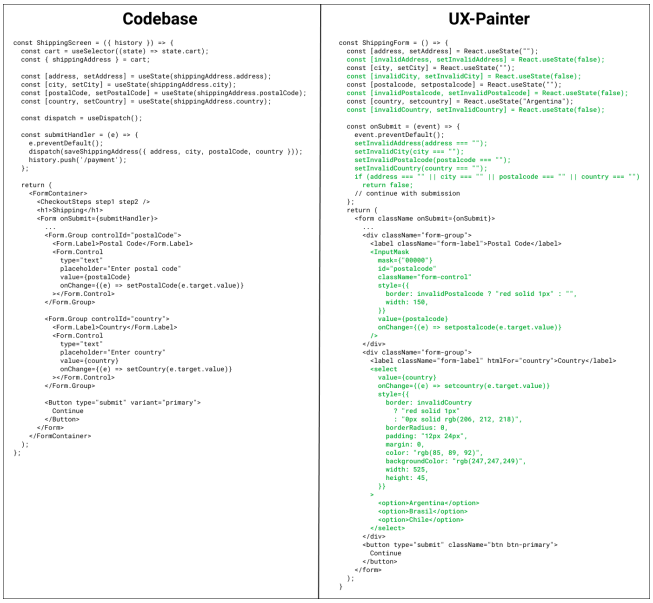

Fig. 4. Comparison of the application's codebase against the code generated by UX-Painter. Green lines correspond to the changes imposed by refactorings.

5 Conclusion

This paper described UX-Painter, a tool that allows synchronizing and communicating UX practices, fostering a method for the systematic improvement of UX issues in agile cycles. We show how the nature of UX-painter makes it particularly useful in an agile setting.

Future work includes the assessment of the tool in a real context of use. In particular, we plan to evaluate if the refactorings provided are suitable for the solutions that designers want to test during a sprint, and the effectiveness of the generated code to reduce developers effort required to implement the refactorings.

```
const ShippingScreen = ({ history }) => {
  const cart = useSelector((state) => state.cart);
  const { shippingAddress } = cart;

  const [address, setAddress] = useState(shippingAddress.address || '');
  const [invalidAddress, setInvalidAddress] = useState(false);
  const [city, setCity] = useState(shippingAddress.city || '');
  const [invalidCity, setInvalidCity] = useState(false);
  const [postalCode, setPostalCode] = useState(shippingAddress.postalCode || '');
  const [invalidPostalCode, setInvalidPostalCode] = useState(false);
  const [country, setCountry] = useState(shippingAddress.country || '');
  const [invalidCountry, setInvalidCountry] = useState(false);

  const dispatch = useDispatch();

  const submitHandler = (e) => {
    e.preventDefault();
    setInvalidAddress(address === '');
    setInvalidCity(city === '');
    setInvalidPostalCode(postalCode === '');
    setInvalidCountry(country === '');
    if (address === '' || city === '' || postalCode === '' || country === '')
      return false;
    dispatch(saveShippingAddress({ address, city, postalCode, country }));
    history.push('/payment');
  };

  return (
    <FormContainer>
      <CheckoutSteps step1 step2 />
      <h1>Shipping</h1>
      <Form onSubmit={submitHandler}>
        ...
        <Form.Group controlId="postalCode">
          <Form.Label>Postal Code</Form.Label>
          <Form.Control
            type="text"
            placeholder="Enter postal code"
            value={postalCode}
            onChange={(e) => setPostalCode(e.target.value)}
            style={{
              border: invalidPostalCode ? 'red solid 1px' : '',
              width: 200,
            }}
          ></Form.Control>
        </Form.Group>

        <Form.Group controlId="country">
          <Form.Label>Country</Form.Label>
          <Form.Control
            as="select"
            value={country}
            onChange={(e) => setCountry(e.target.value)}
            style={{ border: invalidCountry ? 'red solid 1px' : '' }}
          >
            <option value="">Select</option>
            <option>Argentina</option>
            <option>Brasil</option>
            <option>Chile</option>
          </Form.Control>
        </Form.Group>

        <Button type="submit" variant="primary">
          Continue
        </Button>
      </Form>
    </FormContainer>
  );
};
```

Fig. 5. New version of the application's codebase including the refactorings.

References

1. Aldalur, I., Winckler, M., Díaz, O., Palanque, P.: Web Augmentation as a Promising Technology for End User Development. In: New Perspectives in End-User Development, pp. 433–459. Springer International Publishing, Cham (2017)
2. Almughram, O., Alyahya, S.: Coordination support for integrating user centered design in distributed agile projects. In: 15th IEEE/ACIS Int Conf Software Eng. Research, Mgmt and Applications. pp. 229–238 (2017)
3. Bajammal, M., Mazinanian, D., Mesbah, A.: Generating reusable web components from mockups. ASE 2018 - Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering pp. 601–611 (2018)
4. Brhel, M., Meth, H., Maedche, A., Werder, K.: Exploring principles of user-centered agile software development: A literature review. Information and Software Technology **61**, 163–181 (2015)
5. Da Silva, T.S., Silveira, M.S., Maurer, F., Silveira, F.F.: The evolution of agile uxd. Information and Software Technology **102**, 1–5 (2018)
6. Díaz, O., Arellano, C., Aldalur, I., Medina, H., Firmenich, S.: Web Mashups with WebMakeup. In: Rapid Mashup Development Tools. pp. 82–97. Springer (2016)
7. Firmenich, S., Garrido, A., Grigera, J., Rivero, J.M., Rossi, G.: Usability improvement through A/B testing and refactoring. Software Qual J **27**(1), 203–240 (2019)
8. Garcia, A., da Silva, T.S., Silveira, M.S.: Artifact-Facilitated Communication in Agile User-Centered Design, vol. 2. Springer International Publishing (2019)
9. Gardey, J.C., Garrido, A., Firmenich, S., Grigera, J., Rossi, G.: UX-Painter: An Approach to Explore Interaction Fixes in the Browser. Proceedings of the ACM on Human-Computer Interaction **4**(EICS) (2020)
10. Ghiani, G., Manca, M., O, F.P.: Personalization of Context-Dependent Applications Through Trigger-Action Rules. In: ACM TOCHI. vol. 24 (2017)

11. Gothelf, J., Seiden, J.: Lean UX: designing great products with agile teams. O'Reilly Media, Inc. (2016)
12. Jurca, G., Hellmann, T.D., Maurer, F.: Integrating agile and user-centered design: A systematic mapping and review of evaluation and validation studies of agile-UX. Proceedings - 2014 Agile Conference pp. 24–32 (2014)
13. Zaina, L., Sharp, H., Barroca, L.: UX information in the daily work of an agile team: A distributed cognition analysis. Int. J. Human-Computer Studies **147** (2021)