




The Impact of Context-Oriented Programming on Declarative UI Design in React

Koshi You¹^a, Hiroaki Fukuda¹^b and Paul Leger²^c

¹Shibaura Institute of Technology, Tokyo, Japan

²

^{le}

al20092@shibaura-it.ac.jp, hiroaki@shibaura-it.ac.jp, pleger@ucn.cl

Keywords: Context-Oriented Programming, React, React, Declarative UI, Mobile Application, Static Code Analysis.


Abstract: Context-Oriented Programming (COP) is a programming paradigm that modularizes context-dependent behavior. COP provides a language mechanism that enables behavior to switch depending on the current context, such as mobile device orientation or user behavior. COP is particularly promising in domains that frequently involve context-dependent behavior, such as mobile applications and IoT systems. Although various COP abstractions and mechanisms are being proposed, applications implemented with COP are mostly small-scale and designed to validate each proposed approach. These cases are often ideal scenarios for the proposed approach, raising concerns about the sufficiency of the effectiveness assessment. To address this, this paper mimics the functionality of a real-world application and implements it using an application framework. By applying COP to the refactoring, we measure the software metrics of the implementation. As a result, we quantitatively identify the impact of the framework abstraction on the code when COP is used.


1 INTRODUCTION


The importance of context-awareness is increasingly growing due to the expansion of mobile computing and the demand for interactive user interface (UI). In mobile applications, when the UI changes based on the device's orientation, the orientation can be considered as the context, and the process of displaying the UI can be seen as behavior dependent on the context. In addition, a certain number of processes depends on contexts. For example in a mobile application, it will change a method to get data weather the mobile device is online or not (*e.g.*, from cloud via network or a local storage).

Context-oriented Programming (COP) is a programming technique to modularize behavior dependent on contexts (Hirschfeld et al., 2008). COP provides a module called *layer* that encapsulates a set of context dependent operations. In addition, COP also provides a activation mechanism to switch layers depending on contexts called *activation*. Starting with ContextL (Costanza and Hirschfeld, 2005), the first COP language, various COP languages with different

features have been proposed so far to meet various development needs (Fukuda et al., 2022). Besides these languages have been applied to various applications to verify their effectiveness. For example, while ContextL (Costanza and Hirschfeld, 2005) specifies layer activation using dynamic scope, JCOP (Appeltauer et al., 2013) proposes a mechanism that, instead of explicitly specifying the scope, activates layers automatically based on predefined activation rules when context-dependent operations are executed. However, these applications are sometimes ideal and specific to the applied COP languages or their scales are not enough to verify how COP is useful in context-aware application development. In fact, CJEdit (Appeltauer et al., 2009) and RetroAdventure (Appeltauer et al., 2013), which are the largest applications in previous case studies with 3500 Lines of Code (LOC), were implemented using Java GUI frameworks Qt-jambi and Swing, but the frameworks' paradigms are outdated and their code scales are small compared to real products. Moreover, with the complexities and increasing development cycles, using frameworks becomes common (Gandodhar and Chaware, 2018). Thereby it is valuable to evaluate how COP might fit to the context-aware application development with multifunctional framework. In this paper we evaluate the effectiveness of COP in a practical

^a <https://orcid.org/0009-0006-2256-7130>

^b <https://orcid.org/0000-0003-1228-3186>

^c <https://orcid.org/0000-0003-0969-5139>

context-aware application development with a framework. We choose a practical mobile application called *Runkeeper* and re-implement it with a well-known UI framework called *ReactNative* (Native, 2024) using JavaScript. We implement two versions of the mobile application: one using COP and the other without COP. We apply software metrics such as Lines of Code (LOC) and Cyclomatic Complexity (McCabe, 1976) to compare the two versions. Finally, we elucidate the advantages (and disadvantages) of COP based on our experiences from different aspects.

This paper is structured as follows. Section 2 provides an overview of the mobile application with the COP language and ReactNative as a framework. Section 3 describes the layers and activation mechanisms in ContextJS, the COP language used for implementation. Section 4 discusses the software metrics applied in this paper. Section 5 presents the experimental results and provides an evaluation based on these results. Section 6 discusses related work. Finally, Section 7 closes the paper with the conclusion and avenues of future work.

2 StrideTracker

In this section, we introduce the features and context-specific behaviors of the fitness tracking application called *StrideTracker*. Then we briefly explain the ReactNative used for StrideTracker.

2.1 StrideTracker

StrideTracker is an application that mimics the functions of *RunKeeper* (Runkeeper, 2024) that is a commercially available mobile application. Users can use the GPS function to record activities such as running. By analyzing activity data, it can propose exercise plans suitable for the user. After observing the user's exercise, the related data is saved to the server, allowing the user to grasp activity statistics and edit exercise data from the history within the application. StrideTracker is implemented using JavaScript and ReactNative. It also uses *Firebase Realtime Database* (Firebase, 2024), a cloud-based NoSQL database, to load/store necessary data using the APIs provided by Firebase.

2.2 Context-Specific Behaviors

In this section, we show a couple of the context-specific behaviors that need to change behavior based on the execution context in StrideTracker. In our observation, the pieces of code that carry out the



Figure 1: Activity Measurement Mode.

context-specific behavior crosscut several modules such as UI related code, activity data measurement and business logics (e.g., storing data to the database).

Activity Measurement Mode

As shown in Figure 1, the "map mode" for outdoor activities and the "stopwatch mode" for indoor activities involve different UIs, with the map and clock being displayed respectively. The UI code needs to recognize the mode and change either the composition or properties (such as color or size) of the UI. In React Native, the UI is defined by composing small components, which means that multiple components need to be context-aware. In addition to UI-related code, there are also processes like calorie calculation methods and data formats saved to the database, that are also context dependent

Foreground and Background

The running mode of the application such as foreground and/or background can be considered as a context. Several behaviors might be changed based on the running mode, the frequency of data retrieval from the server, the available APIs for obtaining location information, voice guidance and notifications. Specifically, in the foreground, these functions are frequently used to provide users with real-time information and feedback. On the other hand, in the background, the usage frequency of these functions is reduced to save resources and minimize battery consumption.

Saving and Editing Activities

Figure 2 shows the UI for editing activity results in StrideTracker. While Figures 2 (a) and 2 (b) have the same screen layout, their behavior changes depending on whether the user is editing immediately after exercise or from the activity history. When editing from the activity history, pressing the save button ei-

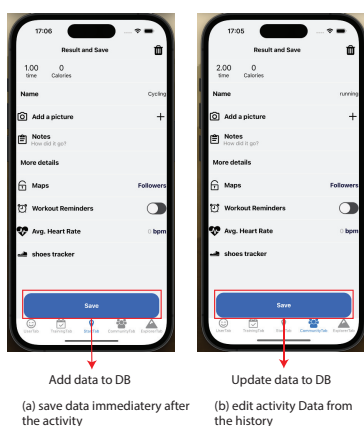


Figure 2: Activity Edit Screen.

ther adds new data to the database or updates the existing saved data.

2.3 ReactNative

ReactNative (Native, 2024) is a framework for developing mobile applications and is one of the most widely used tools for cross-platform application development, alongside Flutter (Flutter, 2024). ReactNative enables programmers to reuse the syntax and design philosophy of React (React, 2024b), which is extensively used in web front-end development. We explain the features of React/ReactNative based development and runtime behavior in the followings.

React

React is a framework for building interactive UI for web applications. Due to the requirements of rich UIs, it becomes common to use frameworks for GUI programming and these frameworks themselves have evolved. In recent years, React has been widely used in the web front-end development and its design philosophy, which includes the component-based architecture and declarative UI, has been widely accepted by programmers. This style has also been widely adopted by other frameworks such as Vue.js (Vue, 2024), Angular (Angular, 2024), and Flutter. Furthermore, React’s technology is applied in various fields, including mobile applications with ReactNative and desktop applications with Electron (Electron, 2024).

Component-Based Architecture

In React, a UI consists of a set of independent components and each component builds a tree structure. Each component has more than one *state*, which can be considered member variables in a class. Moreover a component that is a child of another component (*i.e.*, a parent component) can receive the par-

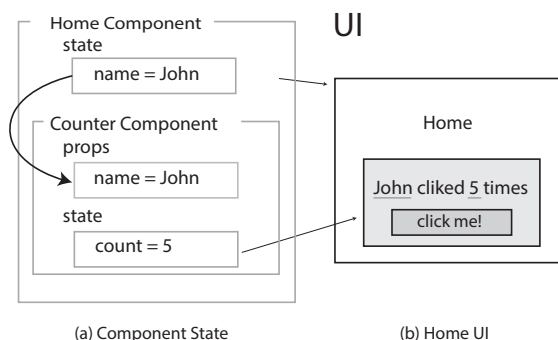


Figure 3: React Component Structure.

ent’s state as *props*. A change of a state becomes a trigger for refreshing/re-rendering GUI.

Figure 3(b) shows a UI that consists of the “click me” button and some texts in which the number (*i.e.*, 5) represents how many times is the button clicked. This UI consists of two components: *Home* component as a parent and *Counter* component as a child as shown in Figure 3(a). The parent state *name* the value of which is “John” is passed to the child as *props*, therefore the child can show the value in addition to its own state such as *count*.

As we have seen so far, in component-based architecture, a system consists of isolated components and share some values by giving their references.

Declarative UI

React uses a declarative UI programming style where pieces of code related to UI are written using UI specific language such as HTML, instead of imperative style where UI are built via APIs such as Swing in Java. Unlike imperative UI, declarative UI explicitly describes the outcome of the UI and its interactions rather than listing the steps to create the UI sequentially. React provides JSX (JavaScript XML) (React, 2024c), a syntax extension for JavaScript that allows programmers to define UI components using HTML-like syntax.

Snippet 1 shows pieces of code that represents the UI shown in Figure 3, and consists of JavaScript in addition to JSX. In React, a component is defined as a function and a component is mainly composed of two blocks. For example the *Counter* component in Snippet 1 has the state declaration (Line 2) and UI declaration using JSX (Lines 3–10). We can refer the state from JSX with curly brackets (Lines 5–6) such as *count*, and an event handler invoked when a certain event happens (*i.e.*, *onClick*) is implemented directly as a lambda function. The *Counter* component is used declaratively as a JSX element in the *Home* component (Line 17). Likewise, JSX allows programmers

to build interactive UI declaratively.

```

1 function Counter({name}) {
2   const [count, setCount] = useState(0);
3   return (
4     <div>
5       <p>{name} clicked {count} times</p>
6       <button onClick={() =>
7         setCount(count + 1)}>
8         Click me
9       </button>
10    </div>
11  );
12 }
13 function Home() {
14   const [name, setName] =
15     useState("John");
16   return(
17     <div>
18       <h1>HOME</h1>
19       <Counter name={name}/>
20     </div>
21  );
22 }

```

Snippet 1: Declarative UI in React.

Component Execution Flow

We can define a UI combining various granularities of components. We explain the process how these components are displayed using Figure 4. As we have seen in Snippet 1, a component in React is defined as a standard JavaScript function, however it should return a JSX element at the end of its execution. JSX is syntactic sugar of JavaScript function (*i.e.*, `react.createElement()`), therefore once a component is executed, the returned JSX element is added to the dedicated tree called *fiber tree*. This adding process is executed synchronously, however React does not refresh the display. Instead, React internally traverses the fiber tree later asynchronously to determine which components should be refreshed. Consequently programmers can declaratively declare UI without worrying about how to detect the change of state(s) and corresponding reactions (*e.g.*, refreshing display).

3 ContextJS AND ITS APPLICATION IN StrideTracker

This section demonstrates how context-dependent behaviors are modularized into layers in ContextJS, a JavaScript COP extension used in applications, and how these layers are activated and deactivated. Furthermore, we explain the characteristics of the *with*

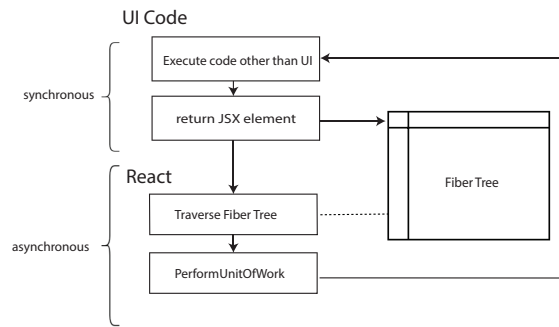


Figure 4: React Component Execution Flow.

block activation mechanism employed in ContextJS.

3.1 ContextJS

ContextJS adopts the class-in-layer approach as a module system for layers, where parts of class methods are defined within layers. The activation mechanism allows layers to be explicitly activated using the `with` syntax, enabling activation within the global scope, the dynamic extent of a block, or the scope per object of the code.

```

1 class MainScreen {
2   renderTitle() {
3     // Display the title
4   }
5   renderBody() {
6     // Display the body
7   }
8   render() {
9     this.renderTitle();
10    this.renderBody();
11  }
12 }
13 const landscapeLayer = new
14   Layer("landscapeLayer");
15 landscapeLayer.refineClass(MainScreen,
16   {
17     renderTitle: function() {
18       // Display the title for landscape
19     },
20     renderBody: function() {
21       // Display the body for landscape
22     }
23   });
24 withLayers([landscapeLayer],
25   function() {
26     MainScreen.render();
27   });

```

Snippet 2: Render the screen using ContextJS.

In Snippet 2, the screen layout changes based on device orientation. The basic behavior is defined in the `MainScreen` class in lines 1-12. In lines 13-21, methods dependent on the landscape context are defined in the `landscapeLayer`. In Line 22, we in-

voke `withLayers`(with block) to activate the landscapeLayer. Then the behavior defined in the landscapeLayer is executed within the block (scope) instead of that in MainScreen.

3.2 With Block Activation Mechanism

The *with block* is a common method for activating layers, enclosing the range to be activated within a block. The dynamic scope of the block defines the activation range. In Snippet 3, `withLayers` is used to create the *block*. Inside the block, `ex.print()` takes the context-dependent behavior and outputs "exampleLayer". Instead, `ex.print()` outside the block outputs the base behavior "baseLayer".

This method gives an easy activation control by explicitly declaring the block. However, it has the drawback of being difficult to express layer activation that spans across control flows. For example, in line 7 of Snippet 3, `ex.print()` within the block outputs "exampleLayer". However when `setTimeout()` is used in lines 8-10 to delay the execution by 1 second, `ex.print()` outputs "baseLayer" despite being within the block. This behavior occurs because `setTimeout()` schedules the callback function `ex.print()` as an asynchronous task, causing it to execute in a different control flow (outside the block).

```

1  withLayers([exampleLayer], () => {
2    ex.print(); // "exampleLayer"
3  });
4  ex.print(); // "baseLayer"
5
6  withLayers([exampleLayer], () => {
7    ex.print(); // "exampleLayer"
8    setTimeout(() => {
9      ex.print(); // "baseLayer"
10   }, 1000);
11 });

```

Snippet 3: Layer activation using with block.

4 COP QUALITY MEASURES

This section outlines the evaluation metrics used to assess the implementation of StrideTracker with COP. Unfortunately, there are no studies focused on measuring metrics for COP code, and since React does not follow an object-oriented paradigm, applying metrics that evaluate modularity in OOP is not possible. Therefore, we adopted Cyclomatic complexity (McCabe, 1976) and its improved version, Cognitive complexity (AnnCampbell, 2017), to measure software complexity, and defined coupling metrics for React's component-based architecture to evaluate modularity.

Component Coupling

Component coupling is a metric that indicates the strength of the coupling among components. In this metrics, if more than two components refer the same variable such as `state`, they are connected. Component coupling is calculated by equation 1 where C_i represents how many other components share the same variables (*i.e.*, states) for *i*th component.

$$\text{Component Coupling} = \frac{1}{n} \sum_{i=1}^n C_i \quad (1)$$

5 VALIDATION

This section evaluates COP using the StrideTracker. We first show the analytics results using code quality metrics detailed in Section 5.1, then consider applicabilities of COP for practical application development. We finally discuss the constraints of applying COP based on the StrideTracker implementation experiences.

5.1 Code Quality Metrics

We show the results of code quality metrics in Table 1 in which the metrics results are averages of all React components. The cyclomatic complexity decreased 0.21 while the cognitive complexity was still the same. Finally, the component coupling also decreased only 0.02. These results mean that applying COP does not improve software metrics drastically. We will discuss this point in Section 5.3.1.

Table 1: Comparison of Metrics with and without COP.

Metric Units	Without	With
LOC	11106	11263
Number of Components	94	88
Cyclomatic Complexity	1.47	1.26
Cognitive Complexity	1.81	1.81
Component Coupling	2.60	2.58

5.2 Applicability of COP

The applicability of COP means that how COP can be utilized for practical software development using frameworks. We measure the application rates from three view points: Context, Component and Requirement, for the StrideTracker implementation.

Context Rate

In React, states that need to be handled across multiple screens or throughout the entire application, such

as appearance settings and user information, are defined as *Context* and we call it as *CTX* to define contexts in COP. The *CTX* is one of the React Context API (React, 2024a), then React is observing the *CTX* and rebuilds UIs when needed, meaning that it is natural to use the *CTX* to (de)activate layers in COP (*i.e.*, ContextJS). As shown in Table 2, from the Context Rate view point, 10 *CTX*s are used to (de)activate layers even though we use 45 *CTX*(s) in the StrideTracker.

Component Rate

The Component Rate represents the proportion of components whose behavior has been refined using COP. In Table 2, we define 88 components totally and refine the behavior of 13 components using COP.

Requirement Rate

The Requirement Rate is the ratio that represents how many components were refined using COP compared to a set of components that we assumed to be refined before the implementation observing Runkeeper functionalities.

Table 2: Application Rates of COP.

View Point	Total	Applied	Rate (%)
Context Rate	45	10	22.22
Component Rate	88	13	14.77
Requirement Rate	21	10	47.62

5.3 Discussion

This section discusses to what extent COP is effective in practical software development. We first discuss it based on the results of Section 5.1, and 5.2, then mention limitations and constraints applying COP as qualitative evaluations from the experiences.

5.3.1 Code Quality Metrics

Regarding LOC, we observed the slight increase. This is because we need to write glue pieces of code to apply COP. As for cyclomatic complexity, we observed the slight decrease. The reason is obviously clear because applying COP allows programmers to define context-dependent behavior at one place (*i.e.*, layer), therefore they do not need to write conditional branches in each component. However its decrease ratio is much smaller than we expected. This is because, in StrideTracker, we observed other branches and loop that did not depend on a specific context. In addition, we sometimes used more than one state to activate a specific context, requiring additional conditional branches to specify the context. Finally, as we

will mention in Section 5.3.3, a semantic mismatch between COP and React is one of the reason. To sum up, simply applying COP basically cannot contribute to increase the software metrics.

5.3.2 Applicability of COP

As we have mentioned in Section 5.2, the Context Rate is approximately 20% while the Component Rate is approximately 15%. This means that even though an application (at least in the case of StrideTracker) consists of a certain number of components and variables that might become triggers to change the screens, the effect of COP is still limited. Moreover, regarding Requirement Rate, we expected the number of context in StrideTracker was 21, however in fact, we applied COP to only 10 contexts. This is because some of context-specific behavior such as authentication management and native API permission management are encapsulated by ReactNative as framework functions.

5.3.3 Limitations and Constraints from the Experience

In addition to the discussions we have mentioned so far, we found semantic mismatches between the activation mechanism of ContextJS and React(Native) through the implementation of StrideTracker. As we explained in Section 2, ContextJS provide *withLayers* that limits the duration of the layer in a block with which we do not need to deactivate the layer explicitly. Thereby we expect that all operations within the block behave as we refined in the layer. However it is not always true when we use COP and ReactNative at the same time. Snippet 4 illustrates this mismatch. In this Snippet, *Body* contains *View* by default (Line 17-19) and it is refined in the *landscapeLayer* using *refineFunction* (Line 25-26). Using these definitions, *MainScreen* activates *landscapeLayer* using *withLayers* (Line 4). Given this definition, we expect the behavior of the *Body* is that we refined in the *landscapeLayer*, however it is still the default behavior. This is because, as we have explained in Section 2.3, *Body* in Line 8 is added to the fiber tree synchronously, meaning that this adding is executed in *landscapeLayer*, however the *Body* is not executed yet. Instead, the *Body* is executed asynchronously by ReactNative considering states, meaning that it will be done outside of the block. For the expected behavior, we need to observe the state inside the *Body* and activate *landscapeLayer*, increasing conditional branches¹. Otherwise, we use only one big compo-

¹In fact, we apply this strategy for the implementation of StrideTracker.

ment, breaking the philosophy of React. This semantic mismatch should be solved in the future for the practical software development using COP with frameworks.

```

1 function MainScreen() {
2   const {isLandscape} =
3     useContext();
4   if(isLandscape) {
5     withLayers([landscapeLayer], ()
6       => {
7         return (
8           <View>
9             <Title/>
10            <Body/>
11            </View>
12          )
13        });
14  }
15  function Body() {
16    return (
17      <View>
18        // Portrait
19      </View>
20    )
21  }
22  const landscapeLayer = new
23    Layer("landscapeLayer");
24  landscapeLayer.refineFunction(Body,
25    function() {
26      return (
27        <View>
28          // Landscape
29        </View>
30      )
31    });

```

Snippet 4: COP Layer Activation in ReactNative.

6 RELATED WORK

In this section, we firstly describe code quality models for frontend frameworks including JavaScript and React. We then investigate how application frameworks except React address context-awareness problems. We finally explore alternative COP solutions that might solve the problems observed in this paper.

There are few code quality models that consider the characteristics of frontend frameworks. (Lin et al., 2017) proposed code quality metrics for react-based web applications. This model introduced 16 metrics based on the characteristics of JavaScript and React, establishing quantitative rules for the metrics. In experimental verification, two projects were used as benchmarks, and this model was applied to practical projects. Our work focuses on the evaluation of applying COP and react-based framework: ReactNa-

tive for a practical mobile application development using JavaScript, and gives quantitative evaluations and points out problems to be solved.

As explained in Section 3, React addresses the challenge of context awareness in the UI by providing APIs to manage state and `CTX`, which automatically trigger re-rendering when changes occur. Similarly, other application frameworks, such as Flutter in the mobile application domain, offer a class called `StatefulWidget` (Flutter, 2024), a blueprint for UI elements that dynamically changes in response to user actions. Like React, Flutter triggers re-rendering when the state changes. In the game engine Unity, it supports defining and executing processes triggered by events such as area entry or object collision in a game (Unity, 2024). These application frameworks share a common feature: they support the management of context and detection of changes to execute domain-critical processes (e.g., rendering or collision detection). However, they do not support the modularization of context-dependent processing arising from application-specific use cases in the way that COP does.

The discussion on applying COP to frameworks is exemplified by the JCop framework, used in the RetroAdventure case study presented in Section 2. JCop identifies issues related to integrating COP with frameworks and proposes language mechanisms to address these challenges. Specifically, in framework implementations, the framework code is typically separated from the user code, and users usually only define entry points in the control flow. As a result, layer activations are moved to these entry points in the user code, leading to redundant activation statements. To tackle this issue, JCop introduces a scoping strategy that declares layer activation conditions independently of the base control flow, similar to pointcuts in aspect-oriented programming. This allows layers to be activated based on predefined conditions during at the start of the execution of partial methods.

In our analysis, we observed similar issues concerning COP in application frameworks, particularly the challenge of activating layers at the appropriate timing. Furthermore, we provide quantitative evidence of how these issues impact code quality. Additionally, we found that the abstraction provided by frameworks, such as declarative UIs, affects control flow in ways that can pose challenges for COP. While an appropriate scoping strategy and activation mechanism compatible with framework abstractions are necessary, current JavaScript COP implementations have not yet addressed this need. One potential approach is to adopt JCop's strategy of activating layers at the start of partial method execution.

7 CONCLUSION AND FUTURE WORK

With the growth of context-awareness using mobile computing, Context-oriented Programming (COP) has become important for the development context-dependent applications. COP provides a module called *layer* to encapsulate a set of context dependent operation to enhance modularity of software systems. Various COP languages with different features have been proposed so far, and they have verified the effectiveness with some applications. However these applications are sometimes ideal and their scales are not enough.

In this paper, we evaluate the applicability of COP through the practical context-aware mobile application development that uses ContextJS as a COP extension of JavaScript and ReactNative as a UI framework. We also apply some software metrics such as cyclomatic complexity in order to evaluate the COP effectiveness from the quantity view points. As a consequence, at least in our implementation, applying COP does not improve software metrics drastically. We think there are two main reasons: 1) some of context dependent behaviors where COP can apply are handled by the framework, meaning that COP is not necessary, 2) semantic mismatches between the activation mechanism (*i.e.*, `with` block) and asynchronous executions in ReactNative, requiring additional conditional branches to achieve the expected behaviors. Even though this paper does not cover all cases that might be happened using COP and framework in practical software development, we have shown that activating layers with `with` block and component-based framework might have problems in a practical application development.

As future work, further validation across various types of mobile applications is necessary to clarify the correlation between applicability and code quality. Additionally, it will be important to measure or propose metrics beyond those presented in this paper to assess the impact of COP on code, with the goal of identifying metrics that effectively evaluate COP's influence on code quality.

REFERENCES

- Angular (2024). Angular. <https://angular.io/>.
- AnnCampbell, G. (2017). Cognitive complexity - a new way of measuring understandability. Technical Report. SounarSource SA, Switzerland.
- Appeltauer, M., Hirschfeld, R., and Lincke, J. (2013). Declarative layer composition with the jcop programming language. *Journal of Object Technology*, 12(2):4:1–37.
- Appeltauer, M., Hirschfeld, R., and Masuhara, H. (2009). Improving the development of context-dependent java applications with contextj. In *International Workshop on Context-Oriented Programming*, page 5, Genova, Italy.
- Costanza, P. and Hirschfeld, R. (2005). Language constructs for context-oriented programming: An overview of contextl. In *Proceedings of the 2005 Symposium on Dynamic Languages*, pages 1–10, San Diego, California.
- Electron (2024). Electron. <https://www.electronjs.org/>.
- Firebase (2024). Firebase. <https://firebase.google.com>.
- Flutter (2024). Stateful widget. <https://flutter.dev/docs/development/ui/widgets/stateful>.
- Fukuda, H., Leger, P., and Cardozo, N. (2022). Layer activation mechanism for asynchronous executions in javascript. In *Proceedings of the 14th ACM International Workshop on Context-Oriented Programming and Advanced Modularity*, pages 1–8. Association for Computing Machinery.
- Gandodhar, P. S. and Chaware, S. M. (2018). Context aware computing systems: A survey. In *2018 2nd International Conference on I-SMAC*, pages 605–608.
- Hirschfeld, R., Costanza, P., and Nierstrasz, O. (2008). Context-oriented programming. In *Journal of Object Technology*, volume 7, pages 125–151.
- Lin, Y., Li, M., Yang, C., and Yin, C. (2017). A code quality metrics model for react-based web applications. In *Intelligent Computing Methodologies*, Cham, Switzerland: Springer.
- MCCABE, T. J. (1976). A complexity measure. *IEEE Transactions on Software Engineering*, 2(4).
- Native, R. (2024). React native. <https://reactnative.dev>.
- React (2024a). Passing data deeply with context. <https://react.dev/learn/passing-data-deeply-with-context>.
- React (2024b). React. <https://react.dev>.
- React (2024c). Writing markup with jsx. <https://react.dev/learn/writing-markup-with-jsx>.
- Runkeeper (2024). Runkeeper. <https://runkeeper.com>.
- Unity (2024). Unity collider. <https://docs.unity3d.com/Manual/CollidersOverview.html>.
- Vue (2024). Vue. <https://vuejs.org/>.