Leveraging the Power of Component-based Development for Front-End Components: Insights from a Study of React Applications

Chen YANG*, Yan LIU✉†, Yiwei LIN§
School of Software Engineering, Tongji University
Shanghai, China
Email: *1610833@tongji.edu.cn, †yanliu.sse@tongji.edu.cn, §1532790@tongji.edu.cn

Jia Yu‡
SEEBURGER China Inc.
Shanghai, China
Email: ‡j.yu@seeburger.com

***Abstract*—Classic design patterns, architectural styles, and design principles have been introduced and enhanced in Web front-end development. Recently, component-based architecture, successfully introduced in React.js, has tended to replace MVC and other MV* patterns in front-end frameworks. However, we still know little about design strategies for leveraging the power of component-based development. We conducted a study to explore the use of components in React-based applications from two levels. Three private repositories were analyzed to get practical insights into the nature, limitations and potentials of CBD for front-end implementations. Our research started with an aerial view, where we examined the dependency, connectivity, and overall of components. Quite different architectural and programming styles were observed; these can be easily attributed to the lack of front-end component design paradigms. Meanwhile, all cases exhibit similar component connectivity and dependency patterns, which enlighten the study to categorize them further. Next, the study zoomed in on the architectural elements level, where we classified front-end components into four categories. Our observations suggest that design components on the architectural elements level may dramatically boost the power of component-based front-end development.**

***Keywords*—Component-Based Development, Web Front-End, React Framework, Case Study**

I. Introduction

Along with the development of Internet and cloud computing, business systems form a unique web-based development style. Due to the plethora of applications served by JavaScript and varieties of programming needs, JavaScript Frameworks have been developed to facilitate the work of Web programmers[1]. The use of React for building web applications has given rise to the popularization of Component-Based Development (CBD) for front-end development. React is mainly applied to developing applications with declarative views and composable components[2]. Owing to the differences between programming languages, scenarios, and implementation method, front-end CBD has a unique nature. However, few studies have been done in this area[6]. In this paper, we explored the implementations of CBD for front-end, and the discoveries stimulated us to categorize front-end components. We hope that our findings can lead to the potential improvements in CBD.

DOI reference number: 10.18293/SEKE2018-147.

This paper makes following contributions: (1) Creation of an experimental exploration process for front-end CBD from a code observation view. (2) Derivation of significant discoveries, such as *orphan components*, connectivity, and dependency patterns from the exploration process. (3) Tentative proposal of a four-category classification of front-end components, mapping of categories to the selected cases, and confirmation of the rationality of the classification.

The paper is organized as follows. In Section II, the scope will be introduced. Section III illustrates the exploration process. In Section IV and Section V, the different levels in the exploration process will be elaborated. In Section VI, the progress in CBD and front-end development will be traced. The conclusion and future work are in Section VII.

II. Research Scope

A. Case Study

CBD for front-ends is at the initial stage, and it's obviously beneficial to explore an old theory under a new situation by doing a case study. Just like follows, we started with selecting independent and complete repositories which can reflect real corporate situations. Then we proceeded to preprocess, where a Javascript analytical tool chain was integrated to extract information from the source code. Besides, same observation views and priorities were applied to the three selected repositories. Next, we continued our exploration to get insights from two levels. Discoveries and conclusions were reached throughout the whole process.

B. Study Scope

To carry out the exploration, we used repositories from private corporate software environments as cases. Repositories released as open source are normally libraries or projects created by individuals. Component libraries act as the basis of other applications, yet few relations can be extracted from these loose components. Elementary, exploratory, or instructional repositories are unable to reflect the stressful, complicated, and systematic corporative application scenarios.

Three private repositories with complete processes were selected, involving different groups; all utilize React to achieve the effect of CBD. These repositories all possess high separability and use AJAX to handle interactions with the back-end through standard JSON-based REST API. Therefore, these repositories are fully equivalent. Discoveries of our case study did not interfere with the programmers behavior since we commenced our research after the repositories had developed.

A brief description of the selected repositories is presented in Table I. React is the *View* layer in *MVC* patterns, so we simply focused on the visible components. Thus only the JS/JSX source code were extracted, while CSS and configuration were not taken into account. Besides, a React-based Web application relies on many different kinds of third-party libraries; we merely focused on libraries related to *View*, paying no attention to those related to language, framework, etc.

TABLE I
OVERVIEW OF CASE PROJECTS

Project agname	LOC	Start - End date	Duration (week)	Contri-butors	Third-party library/View	Usage Scenario
BD	6,929	2016.12.26-2017.05.22	21	6	8	life service provider
MY	6,523	2016.10.24-2017.03.08	19	6	14	financial management
FE	26,807	2016.08.07-2016.11.08	13	3	9	fund analysis

In order to guarantee the validity of results, we integrated a Javascript analytical tool chain. Firstly, due to the incompatibility between the ES6 in React-based projects and *Esprima*[1], the raw source code without irrelevant files was compiled into the ES5 format via *Babel*[2]. By *Estraverse*[3], we traversed, extracted, and persisted the preliminary data, including components, relations, properties, functions from the abstract syntax tree generated by *Esprima*. Lastly, we combined or disassembled this information and converted it into an understandable visualization.

III. EXPLORATION PROCESS

The theory of CBD has become mature after 50 years of back-end development[4]. However, little is known about the design strategies for harnessing CBD capabilities for front-end development. So, we created an exploration process in an attempt to figure out the current situation of front-end CBD. Fig.1 illustrates the basic workflow of our exploration process from three key phases. Progressive relationship exists between the phases, and they also influence each other conversely. In this paper, we summarize the major conclusions in the second phase due to space limitations. A brief description of the first phase are given to pave the way for the second. Although the elaboration of the *component timeline* isn't presented in this paper, evidence suggests that it deserves further investigation.

The process starts with global features at the aerial view. We extracted components and relations from source code and got observations from three different aspects. Above all, we did an overall observation. Afterwards, certain features (see Section IV) are selected to analyze the state or extent of all components connected in *connectivity* aspect. Finally, regarding the *dependency* aspect, there are many remarkable features associated with the degree of *dependency* between one component and others. Significant observations on the aerial view provided the basis for launching a thorough analysis.

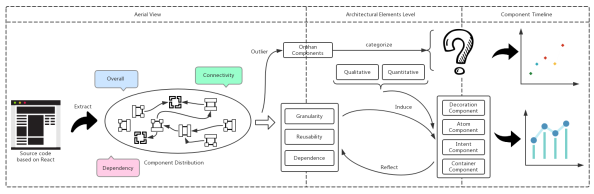


Fig. 1. Exploration Process

Meaningful discoveries could be derived from the aerial view, which we generalized into three aspects. Firstly *Granularity* refers to the encapsulation capability of components, specifically the number of attributes and functions. We also made discoveries about *Reusability*, like the reuse degree and component reuse pattern. Valuable relation features were obtained from the *Dependence* among components. Besides, abnormal discoveries like orphan components and groups were also made. Further analyzing the timeline of orphan components can help us investigate the accumulation process.

With the help of external features emerging from the *aerial view*, we proceeded with our exploration at the *architectural elements level*, where we focused on the nature of one component itself. We mapped components into four categories considering their typical usage scenarios, and examples were presented to facilitate the readers understanding. By attaching internal qualitative and quantitative features, each component was categorized uniquely according to its distinctive nature. Similarly, these categories also reflect external features.

IV. OBSERVATIONS FROM THE AERIAL VIEW

A. Feature Selection and Case Exhibition

To get a better idea of the aerial view, a brief summary of the key features derived from these aspects is shown in Table II, and important attributes that represent the key features quantitatively are listed too. Statistical features such as scale (Table II, 1.1) are shown directly. Graphs in Fig.2 are created to facilitate the understanding of the remaining features that cannot be captured in a numerical form. Moreover, attributes like Scale and Code Clone were calculated using SonarQube[4], and attributes like Maximum Path were gained by combining or disassembling the basic information extracted in Section II.B. The dependency graphs are drawn by Gephi[5]. Finally, Table III shows the observations and explanations of the cases.

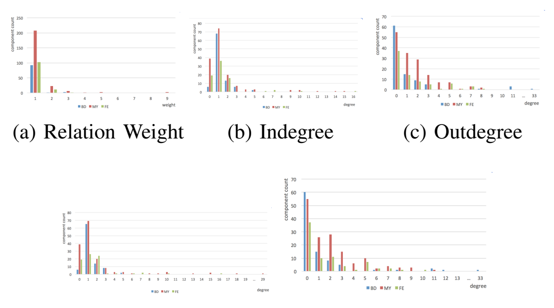
The dependency graphs describe the state of components and relations from a global perspective. As shown in Fig.3, the internal invocation structure of the project can be regarded as

[1]Esprima: http://esprima.org/
[2]Babel: http://babeljs.io/
[3]Estraverse: https://github.com/estools/estraverse
[4]SonarQube: https://www.sonarqube.org/
[5]Gephi: https://gephi.org/

graphs, where nodes represent components and edges represent relations between them. The degree of a node in graphs is the number of edges attached to the given node. The nodes in gray, red, blue, and green respectively represent the ordinary, leading indegree, leading outdegree, and leading degree nodes. The size and shade of a edge indicate the weight of it, determined by the number of relations between one component and others. In addition, these graphs are directed graphs, where the direction of an arrow denotes the callee.

TABLE II
ATTRIBUTES DERIVED FROM FEATURES

Aspect	Key Feature	Attribute	BD	MY	FE
1.Overall	1.1. Scale	Scale	6929	6523	26807
	1.2. Code Clone	Code Clone	7.6%	10.7%	24.1%
	1.3. Components and Relations Summary	Components	95	153	76
		Relations	122	241	114
2.Connectivity	2.1. Global Relations Pattern	Maximum Path	3	4	3
		Average Path Length	1.355	1.482	1.371
		Maximum Relation Weight	3	9	3
		Relation Weight Distribution	As shown in Fig. 2. (a)		
	2.2. Disconnected Components	Disconnected Set	4	7	2
		Disconnected Components	17/15.2%	23/13.1%	14/15.6%
3.Dependency	3.1. Relation Strength	Average Degree	2.568	3.15	3
		Maximum Indegree	5	15	16
		Maximum Outdegree	33	8	8
		Average Weighted Degree	1.337	1.98	1.671
		Maximum Weighted Indegree	5	29	16
		Maximum Weighted Outdegree	33	11	10
	3.2. Relations Distribution	Indegree Distribution	As shown in Fig. 2. (b)		
		Outdegree Distribution	As shown in Fig. 2. (c)		
		Weighted Indegree Distribution	As shown in Fig. 2. (d)		
		Weighted Outdegree Distribution	As shown in Fig. 2. (e)		


(a) Relation Weight (b) Indegree (c) Outdegree

(d) Weighted Indegree (e) Weighted Outdegree

Fig. 2. Attributes Distribution

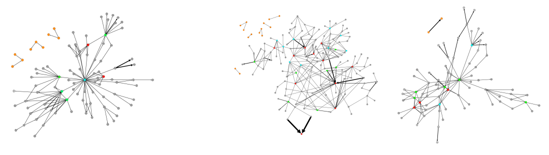

(a) BD (b) MY (c) FE

Fig. 3. Dependency Graphs

B. Discovery

In light of the observations, our preliminary discoveries regarding the three aspects are as follows: **Granularity:** (i) Components with fine granularity tend to possess a higher

TABLE III
OBSERVATIONS FROM THREE ASPECTS

Aspect	Observations	Explanations / Possibilities
Overall	An asymmetry exists between the number of components & relations and the scale of the respective project.	Different coding styles may account for this phenomenon, *FE* exhibits a lack of decomposition, while *MY* tended to be overencapsulated.
	A project with larger components (*FE*) has the highest code clone rate (24.1%).	Similar tiny functions are implemented in the form of code clones between components.
Connectivity	Complex multi-layer nesting rarely exists.	Nestings of components in React come at a cost.
	The relation weight of the dependency graphs are generally very small.	Cannot determine if a higher relation weight is beneficial, yet the callee component with larger relation weight (9), importing third-party components directly with nothing attached, can be considered as a kind of over-encapsulation.
	Orphan component (a component that is independent of the primary components set) and *group* exist with a fairly high ratio (14%).	Reasons such as code examples, strategy, and requirements changes led to the legacy of orphan groups
Dependency	Components with smaller degrees account for the majority; few differences exist between distributions with weight and those without.	Simple dependency is dominant.
	Different top-ranked in-degree components perform different functions; Over-encapsulation exists in components with an extremely high in-degree.	Components with a higher in-degree tend to either be (1) basic components applied in different pages, or (2) relatively complex components implemented a specific function.
	Different top-ranked out-degree components show different features.	Components with a higher out-degree tend to be (1) page-level components that invoke many other components; or (2) normal components that invoke over-encapsulation components.

in-degree; (ii) components with rough granularity tend to possess a higher out-degree; (iii) a potential correlation may exist between granularity and code clones. **Reusability:** (i) Reusability didn't achieve the optimal effect; (ii) the over-encapsulation phenomenon may affect the reusability of components. **Dependence:**(i) Simple dependency plays a major role; (ii) orphan components deserve more discussion.

V. INSIGHTS FROM THE ARCHITECTURAL ELEMENTS LEVEL

In the exploration of the *aerial view*, quite different architectural and programming styles were observed. These can be easily attributed to the lack of front-end component design paradigms, which motivated us to perform a thorough analysis.

For existing front-end development, efforts have been made regarding different types of business logics encapsulation. For example, React is regarded as a presentation (view) layer, and often used with Redux, which is separated into an action (controller) layer and a reducer (model) layer as well, to interact with interfaces[3]. The appearance of stateless components in React also indicates that components in front-end gradually perform their respective roles. So that a layered architecture for front-end is one possible way to improve development by providing more controllability. According to programming experiences and design specifications for building components, we tentatively propose a classification methodology.

A. A Four-Category Classification of Front-End Components

We propose a four-category classification of components for React-based applications in Table IV considering typical usage scenarios. As space is limited, a graphical example to help readers understand the component categories is available online[6]. It's strongly recommended for readers who want to understand the usage scenarios and code features of different categories. Reasonable use of the four-category classification may have a positive effect on good design.

[6]Example: https://github.com/Ada12/RCCE/blob/master/example.md

TABLE IV
CLASSIFICATION OF THE COMPONENTS

Component	Definitions
Decoration Component	A *Decoration Component* is a tiny component that is only responsible for decorative functionalities with extreme dependence upon parent components, and pays no attention to its own state and lifecycle.
Atom Component	An *Atom Component* is a basic component that tends to be an inseparable unit of functionality with complete lifecycle management. It can be a constituent of other more complicated components.
Intent Component	An *Intent Component* is a more complex component, which can be thought of as the glue between components. It implements a complete presentational business process, and can be composed of *Decoration Components* and *Atom Components*.
Container Component	A *Container Component* is a page-level component that manages states exposed by subcomponents unifiedly, and composed of the components mentioned above. It's responsible for communications between different subcomponents and interacted with external interfaces.

B. *Findings from the Case Study*

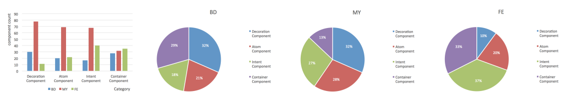
In order to observe the categories in real-world React-based projects, we continued the analysis of our cases. Although we mapped the categories according to usage scenarios and qualitative features, definite quantitative features should still be used as benchmarks. In addition, the preliminary discoveries summed up in Section IV were also taken into account. Table V lists the features we utilized to categorize the components. Table VI shows the mapping of components and features, in which the symbol $\surd$ represents one component possesses a given feature, and symbol $\uparrow$ or $\downarrow$ represents the component possesses a larger or smaller value of a given feature.

TABLE V
SUMMARY OF BASIC FEATURES

Observation Aspect	Quantitative	Qualitative	Discoveries
Key Element	1.1 state; 1.2 props; 1.3 invoke third-party library 1.4 invoke other component 1.5 life cycle management 1.6 interact with interfaces	2.1 usage scenarios 2.2 business coupling 2.3 scale	3.1 indegree 3.2 outdegree 3.3 reusability

TABLE VI
MAPPING OF COMPONENT AND FEATURES

Component	Features									
	1.1	1.2	1.3	1.4	1.5	1.6	2.3	3.1	3.2	3.3
Decoration		✓					↓	↑	↓	↑
Atom	✓	✓	✓		✓		↓	↑	↓	↑
Intent	✓	✓	✓	✓	✓		↑	↓	↑	↓
Container	✓	✓	✓	✓	✓	✓	↑	↓	↑	↓



(a) Counts (b) BD (c) MY (d) FE

Fig. 4. Component Distribution by Category

By the category criteria of Table V and VI, we categorized the three cases manually; the results are shown in Fig.4. We found that (1) project (*'FE'*) which possesses a lack of decomposition and a higher code clone rate, has less *Decoration* and *Atom Components*, due to the unreusable tiny functions; (2) project (*'BD'*) with high quality demands of user experience tend to have more *Decoration Components*; (3) more *Intent Components* exist in project (*'MY'*) that focuses a great deal on business process.

C. *Conclusion*

In conclusion: (1) Significant differences exist between the various component categories; (2) different categories can represent specific usage scenarios; (3) a slight variation in distribution exists for categories under different scenarios.

VI. RELATED WORK

It has been fifty years since software components was firstly proposed[4]. CBD developed rapidly and gradually formed its own philosophy. Alan[5] indicated that a component is a deliverable piece of functionality that can independently provide interface access to its services. Tassio[6] investigated 1231 studies dating from 1984 to 2012, and summarized the domains where CBSE has been applied; yet most of them were related to the server side, only one showed solicitude for providing better services for the front-end. These all indicate that little research has been done on CBD for front-ends.

Based on the keyword JavaScript, we explored papers in 15 conferences in the field of SE over past five years. 48 papers involving a variety of different areas, matched the condition. However, none of them pay close attention to the field of CBD for front-end development. In addition, some papers explored the related area. Cappiello[7] defined a quality model for building blocks of mashup applications. Magnusson[8] found that a certain framework and the way to implement its data flow pattern are the main reasons that impact maintainability.

VII. CONCLUSION AND FUTURE WORK

In this paper, we carried out an exploration tentatively, and the constant outpouring of discoveries prompt us to propose a four-category classification of front-end components. By the researches above, we concluded that (1) Hard-to-maintain phenomena such as lack of decomposition and over-encapsulation may be avoided by adopting a good design prior to CBD; (2) Components for front-end may have more capabilities of prefactoring, so components can be created purposefully according to the categories we proposed. In the future, we will continue our exploration on the *component timeline*, *orphan components*, and component code clones.

REFERENCES

[1] Andreas Gizas, F., Sotiris P. Christodoulou, S., Theodore S. Papatheodorou, S.: Comparative evaluation of javascript frameworks. In: WWW '12 Companion Proceedings of the 21st International Conference on World Wide Web, 513-514(2012).
[2] "React home," https://facebook.github.io/react/, accessed: Dec.15, 2017.
[3] "React&Redux MVC," https://hackernoon.com/thinking-in-redux-when-all-youve-known-is-mvc-c78a74d35133, accessed: Feb.5, 2018.
[4] Brown A W. Component-Based Development[J]. 2000.
[5] Alan W. Brown, Large-Scale, Component Based Development, Prentice Hall PTR, Upper Saddle River, NJ, 2000
[6] Vale T, Crnkovic I, Almeida E S D, et al. Twenty-eight years of component-based software engineering[J]. Journal of Systems & Software, 2016, 111:128-148.
[7] Cappiello C, Daniel F, Matera M. A quality model for mashup components[C]//International Conference on Web Engineering. Springer, Berlin, Heidelberg, 2009: 236-250.
[8] Magnusson E, Grenmyr D. An Investigation of Data Flow Patterns Impact on Maintainability When Implementing Additional Functionality[J].2016.