情報処理学会論文誌 Vol.61 No.10 1605–1616 (Oct. 2020)

Jact：JavaScriptフレームワーク理解支援のためのプレイグラウンド型ツール

中島 望[1,a) 松本 真佑[1,b) 楠本 真二[1,c)

受付日 2019年10月18日, 採録日 2020年7月7日

概要：JavaScript フレームワークとは，Web のフロントエンド開発を効率化するための枠組みである．大規模な Web アプリケーションを開発する際にはフレームワークの使用が一般的であり，開発者は使用するフレームワークを適切に選択する必要がある．しかし，適切なフレームワークの選択は開発者にとって容易ではない．フレームワークは記法等の実装面および処理時間等の性能面でそれぞれ特徴がある．またフレームワークは日々新しく誕生しており，改良されたものが継続的にリリースされている．そのため，フレームワークの単純な比較情報がすぐに古くなってしまい，正しい特徴理解の妨げになる可能性がある．そこで本研究では，フレームワークの特徴理解を支援するのためのプレイグラウンド型ツール Jact を提案する．Jact は各フレームワークの記法と処理時間を，Web 開発における典型的なタスクをベースに比較する機能を提供している．プレイグラウンドという性質により，Jact はブラウザ上でのタスクの実行，およびソースコードの編集と共有が可能となる．さらに，利用者による比較のためのタスクとソースコードのサーバへの投稿も可能にしている．比較用のタスクとソースコードを利用者自身が登録することで，Jact は各フレームワークの実装面および性能面の情報を継続的に提供できる．またフレームワーク理解における Jact の効果を評価するために，被験者を用いた評価実験を行った．

キーワード：JavaScript フレームワーク，Web アプリケーション，プレイグラウンド

Jact: A Playground Tool Supporting Comprehension of JavaScript Frameworks

Nozomi Nakajima[1,a) Shinsuke Matsumoto[1,b) Shinji Kusumoto[1,c)

Received: October 18, 2019, Accepted: July 7, 2020

Abstract: JavaScript framework (in short, JSF) is a skeleton for improving the efficiency of Web front-end development. JSFs are generally adopted to develop large-scale web applications. Selecting the most appropriate JSF from various JSFs is critical but difficult to achieve. Different JSFs have different characteristics in terms of coding and performance. Also, a novel JSF has been often developed, and existing JSFs have been updated frequently. Therefore, comparison information of JSFs can quickly become obsolete. Outdated information may hinder understanding of JSF characteristics. In this paper, we present a playground tool named Jact to support comprehension of individual JSF characteristics. Jact enables users to compare grammar and runtime performances of each JSF based on typical tasks in web development. By the concept of playground, users can freely execute, edit, and share source code in their web browsers. Furthermore, users can submit tasks and source code to the server. By registering tasks and source code written by users themselves, Jact can continuously provide information relating to JSF. In order to evaluate the effectiveness of Jact, we conducted a subject experiment with 13 participants.

Keywords: JavaScript framework, web application, playground

© 2020 Information Processing Society of Japan

1. はじめに

Web 周辺技術の進化および爆発的な需要の増加にともない，Web アプリケーションは複雑化の一途をたどっている [9]．この複雑化を回避し，開発効率や保守性を向上するために，JavaScript フレームワーク（以下，JSF）を用いた Web 開発が広く行われている [6], [14]．JSF の具体例としては Vue.js や Angular が知られている．

各 JSF は MVC（Model-View-Controller）や MVVM（Model-View-ViewModel）といった様々なアーキテクチャパターンを採用しており [13]，開発者は JSF のパターンに基づいてアプリケーションを構造化する [7]．またアプリケーションの開発時に必要とされる機能があらかじめ関数として用意されており，それらを利用することでソースコードを簡略化できる [7], [14]．開発者は JSF を適用することで，高品質なソースコードを効率的に開発できる．

JSF の採用はアプリケーションの開発面では有益な反面，パフォーマンスの低下や消費電力の増大といった実行面での負の効果が避けられない [11]．これは JSF 利用により様々な機能が付与されることによる副作用といえる．そして適用する JSF の選択においては，プログラミング言語やパラダイムの選択と同様に正解は存在しない．開発するアプリケーションの特性や，開発者達の好み，スキル等の条件に合うものを選ぶ必要がある [14]．また機能を提供するライブラリは特定の機能を別ライブラリと容易に差し替えられるが，全体の構造を規定する JSF は開発途中での変更が容易ではない．よって高品質なソースコードと高機能なアプリケーションを実現するためには，JSF の実装面および性能面の特徴を把握し，開発するアプリケーションに応じた適切な JSF の選択が重要である [7]．

一方で適切な JSF の選択は容易ではない．JSF の選択を難しくする要因として，以下の点があげられる．1 つ目は，JSF の選択肢の豊富さである．同じアーキテクチャパターンを採用している JSF が複数存在しているため，採用したいアーキテクチャパターンから 1 つの JSF を選び出すことは難しい．たとえば MVVM というアーキテクチャパターンを採用している JSF には Vue.js や Knockout.js 等があげられる．同じアーキテクチャパターンを採用する JSF でもそれぞれ記法は異なるため，比較時には各 JSF の記法の違いを把握する必要がある．

2 つ目は，実行環境の多様さである．Web アプリケーションの特性上，Google Chrome や Firefox といったブラウザ，スマートフォンや PC といったデバイスの組合せにより，多数の実行環境が想定される．JSF は実行環境によらず一様に動作させるための Polyfill [*1]を含んで実装されているため [14]，JSF の性能は実行環境によって異なるという指摘も存在する [6]．比較時には各 JSF の実行環境による性能面の違いを把握する必要がある．

3 つ目は，情報入手の難しさである．Web 技術は成長が早く [4], [8]，JSF だけでなく実行環境であるブラウザも頻繁にアップデートが行われる [2]．たとえば Google を中心に開発されている JSF の Angular は，2018 年 5 月にバージョン 6.0.0 がリリースされた後，5 カ月後の 10 月にバージョン 7.0.0 がリリースされている [1]．リリース時には JSF の記法の変更やパフォーマンスの改善が行われることが多く，利用したい条件での情報を得ることは難しい．

本稿では開発者に対する JSF の理解と選択の支援を目的として，JSF を実装面および性能面で比較するプレイグラウンド型ツール Jact を提案する．Jact は 1 つ目の課題に対応するためにソースコードの比較機能とプレイグラウンドを，2 つ目の課題に対応するためにオンデマンドな処理時間計測機能を持つ．またプレイグラウンドによって JSF についての情報を共有可能にし，3 つ目の課題に対応する．なお，Jact は現在公開サーバにデプロイ済み[*2]であり，実際に利用することが可能である．評価実験として Jact による JSF の記法および処理時間の理解における有用性を評価する被験者実験を行った．実験の結果，JSF の理解において Jact の利用が有効であることが確認できた．

2. 準備

2.1 プレイグラウンド

ブラウザ上でソースコードの編集，実行および共有が可能なサービスをプレイグラウンドと呼ぶ．代表的な JavaScript のプレイグラウンドとしては，JSFiddle [*3]があげられる．利用者は実行環境を準備する必要がないため，ソースコードの編集や動作確認をブラウザ上で手軽に行うことができる．また URL を経由してソースコードとその実行環境を共有することも可能である．

2.2 JavaScript フレームワーク

JavaScript フレームワーク（JSF）とは，Web のフロントエンド開発を効率化するために使用されるソースコードの枠組みである．一般にソースコードに対して部分的に機能を付与するために利用されるライブラリとは異なり，フレームワークはソースコード全体に MVC 等のアーキテクチャを付与する目的で利用されている [21]．JSF はアーキテクチャパターンだけでなく，記法や利用できる関数に

[1] 大阪大学大学院情報科学研究科
Graduate School of Information Science and Technology, Osaka University, Suita, Osaka, 565–0871, Japan
[a] n-nakajm@ist.osaka-u.ac.jp
[b] shinsuke@ist.osaka-u.ac.jp
[c] kusumoto@ist.osaka-u.ac.jp

[*1] ブラウザ間の仕様の違いに対応するために埋め込まれるソースコードを指す．
[*2] Jact のデモページ：http://13.231.18.92/
[*3] https://jsfiddle.net/

© 2020 Information Processing Society of Japan

それぞれ特徴がある．JSF の利用によってソースコードを構造化したり，あらかじめ定義されている様々な関数を利用したりすることができるため，コード品質の向上や開発の効率化が期待できる [14]．JSF の利用例として，同一の機能を実現する JavaScript のソースコードと Vue.js を用いたソースコードを**図 1** に示す．図 1 は Web ページ内の文字列 “World” を “JS” に書き換えるソースコードである．Vue.js は MVVM というアーキテクチャパターンを採用している．MVVM は画面上の表示（以下，View）とアプリケーションのデータ（以下，Model）を紐付ける．通常 JavaScript では View もしくは Model の一方が変更された際，もう一方に変更を反映させる処理が必要になる．MVVM は View と Model の紐付けによって，変更の反映処理が不要になる．これによって，開発者はインタラクティブなインタフェースの作成を容易に行うことができる．

```
<div>
  <p>Hello
    <span id="msg">
      World</span>
  </p>
</div>
<script>
  let str = 'JS';
  document
    .getElementById('msg')
    .innerText = str;
</script>
```

(a) JavaScript（JSF なし）

```
<div id="app">
  <p>Hello {{ msg }}</p>
</div>
<script>
  let app = new Vue({
    el: '#app',
    data: {
      msg: 'World'
    }
  })
  app.msg = 'JS';
</script>
```

(b) Vue.js

図 1 同機能の実装に対する JSF の記法の違い

Fig. 1 Difference of implementation with and without JSF.

2.3 フレームワーク選択における課題

JSF の利用は開発者にとって様々なメリットがある一方で，適切な JSF の選択は容易ではない [6], [7], [14]．JSF の選択における課題点として，以下の点があげられる．

2.3.1 P_1：選択肢の豊富さ

JSF によって採用しているアーキテクチャパターンは様々であり，例として MVC や MVVM があげられる．同じアーキテクチャパターンを採用している JSF は複数存在するが，ソースコードの記法は JSF によって異なる．例として MVVM をアーキテクチャパターンに採用している Vue.js と Knockout.js をあげる．図 1 に示したとおり，Vue.js では HTML タグで変数を囲んでいる．一方 Knockout.js では，HTML タグの属性に紐付けるデータを明示する必要があり，Vue.js とは記法が異なる．このように，開発者はアーキテクチャパターンを選択すれば JSF を決定できるのではなく，それぞれの JSF の記法を把握して比較する必要がある．

2.3.2 P_2：実行環境の多様さ

アプリケーションの処理性能は開発者，利用者どちらにも重要である [18]．特に Web アプリケーションは実行環境が使用するブラウザ，デバイスによって構成されるため，様々な環境で利用されることを考慮する必要がある．Gizas らの研究 [6] 等で示されるとおり，JSF の性能は実行環境によって大きく異なる．

2.3.3 P_3：情報入手の難しさ

JSF そのもの，そして実行環境であるブラウザも頻繁にアップデートが行われる．JSF の 1 つである Angular はわずか 5 カ月で新たにメジャーバージョンがリリースされ，新機能の追加やパフォーマンスの改善が行われている [1]．また Baysal らの研究 [2] によると，Google Chrome は 2 カ月半ごと，Firefox は 10 カ月ごとにメジャーバージョンがリリースされている．他のソフトウェア開発技術と比較しても Web 技術の進化は早く [4], [8]，実装方法や性能の比較情報の寿命は短いといえる．実際に開発者が JSF の比較を行う際には比較情報が古くなっている場合が多く，自分が利用したい条件での情報を得ることが難しい．そして JSF に関する古い情報を利用することが，間違った JSF の性質理解につながる可能性もある．

3. 提案手法：Jact

3.1 概要

本稿では前章で述べた課題の解決を目的とした**タスクベース**の JSF 理解支援ツール Jact を提案する．Jact は**ソースコードの比較**，**オンデマンドな処理時間計測**，**プレイグラウンド**という 3 つの性質を持つ．また利用者は Jact から**タスクおよびソースコードを投稿**できる．投稿機能の実現のために，Jact は**ソースコードのテスト機能**を持つ．これらの性質と課題 P_1〜P_3 の対応を**表 1** に示す．表の列は Jact の性質を，行は課題 P_1〜P_3 を示す．Jact の利用によって JSF の実装面および性能面の特徴が比較可能になり，開発者の JSF の理解や選択に対する支援が期待できる．

3.2 特徴

3.2.1 タスクベース

Jact では JSF の実装面，性能面の比較をタスクという単位で実施する．ここでのタスクは Web アプリケーションを実装する際に用いられる汎用的な機能を意味しており，DOM 操作や Ajax 通信等が該当する [6]．

タスクベースの JSF 比較は，JSF の特徴を理解するという大きな目標を小さく分割し，段階的な JSF の理解を促す狙いがある．短く簡潔なソースコードの方が，複雑なソースコードよりもプログラム理解においては有効であることを示す研究も存在する [19]．実際に，プログラミング言語のドキュメントや教本には，単純な機能を実現するための数行のソースコードが掲載されていることが多い．このよ

© 2020 Information Processing Society of Japan

表 1 Jact の各性質と課題 P_1〜P_3 の対応

Table 1 Features of Jact corresponding to the three problems.

性質	P_1	P_2	P_3
ソースコード比較	◯		
処理時間計測		◯	
プレイグラウンド	◯		◯
利用者による投稿			◯
ソースコードのテスト			△[*4]

うに，JSF の全体像を把握するために最初から巨大で複雑なアプリケーションを提示されるよりも，アプリケーションを分割し，機能単位でソースコードを提示される方が理解の助けになると考えられる．また，簡単なソースコード例をもとに記法を理解する手法は開発者にも用いられており [14]，実際の開発現場においても有効な手段だといえる．

タスクの粒度については，単体テストの対象となるような単一の意味を持った処理を想定している．たとえば図 1 に示すような，特定の ID を持つ DOM 要素を選択し，その要素の内部テキストを書き換える，というような処理が該当する．この例では，DOM 選択と DOM 書き換えの 2 つの操作が必要であり，手続き的に行うか（図 1 左），宣言的に行うか（図 1 右）という 2 つの記法の差が明確に表れている．フロントエンド開発はこのような小さな機能の組合せで実現される．よって，ある単一の機能がどのように実現されているかを複数 JSF 間で比較することにより，各種 JSF の違いの理解支援が実現できると考えられる．

3.2.2 ソースコードの比較

あるタスクを実装した各 JSF のソースコードを比較することで，課題 P_1：選択肢の豊富さに対応する．ソースコードを比較することで，JSF の実装面の特徴である記法を比較できる．

3.2.3 オンデマンドな処理時間計測

また Jact は課題 P_2：実行環境の多様さに対応するために，各 JSF の性能面の比較を実現する処理時間計測機能を持つ．選択したタスクに登録されている各 JSF のソースコードを実行して処理時間をオンデマンドに計測し，グラフで表示する．様々な環境で本機能を用いることで，JSF の性能面の特徴である処理時間を比較できる．

3.2.4 プレイグラウンド

ソースコードの比較機能とともに，課題 P_1：選択肢の豊富さに対応する性質である．JSF の理解においては，実際にソースコードを動作させることが有効である [14]．ツールをプレイグラウンド化することでブラウザ上でのソースコードの編集や動作確認が可能になり，JSF の理解を深めることができる．

Jact のプレイグラウンド化は，課題 P_3：情報入手の難しさにも有効である．既存のプレイグラウンドではソースコードの共有が可能だが，類似した機能の実装をまとめて表示する機能等は提供されていない．Jact では各タスクごとにソースコードを共有し，同じ条件下での JSF の比較を可能にすることで，リアルタイムな JSF の情報を提供可能にする．

*4 ソースコードのテストは利用者の投稿を実現するための機能であり，直接的に P_3 に対応しているわけではない．

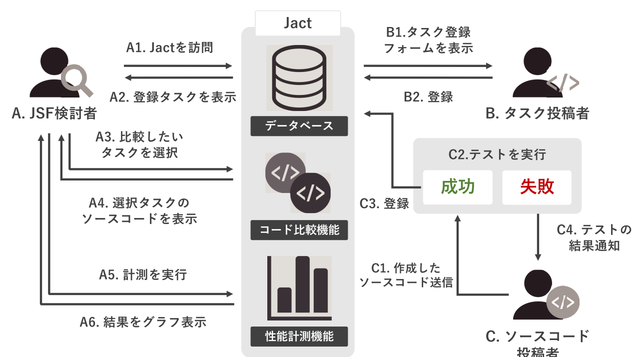

図 2 Jact の利用の流れ

Fig. 2 Architecture of Jact and its usage flow.

3.2.5 利用者による投稿

Jact のプレイグラウンド化を実現するための性質である．事前に定義されたタスクやソースコードのみでは，新しい情報への対応や十分な情報の提供は難しい．そこで，登録されていないが開発時に必要となるタスクや，同じ JSF を用いた別のソースコード，そして別の JSF を用いたソースコードの投稿を可能にした．利用者自身が情報を追加できる性質により，新しい情報や不足している情報を補うことができる．

3.2.6 ソースコードのテスト

ソースコードの投稿を実現するために，ソースコードのテスト機能が追加されている．投稿されたソースコードが，タスクを実現するための条件を満たしているかテストを行う．この機能により，タスクを実現するための条件をすべて満たしたソースコードのみの登録を実現する．

3.3 利用の流れ

Jact の利用の流れを図 2 に示す．Jact は 3 種類の利用者を想定しており，利用者によって利用の流れは異なる．

JSF 検討者：Jact を利用することで自分が使用する JSF を検討したり，JSF の特徴を把握する利用者である．JSF 検討者は図 2 において A1〜A6 の矢印で示されるように，Jact に示されたタスクから自分の利用したいタスクを選択し，そのタスクを実現するソースコードを比較できる．また処理時間計測を実行することで，自分の環境下での各 JSF の処理時間を計測できる．

タスク投稿者：Jact に用意されていないタスクを追加で投稿する利用者である．タスク投稿者は図 2 において B1，B2 の矢印で示されるように，タスクの情報を Jact に投稿する．

ソースコード投稿者：Jact で用意されているタスクに対

© 2020 Information Processing Society of Japan

して，自作のソースコードを投稿する利用者である．ソースコード投稿者は図 2 において C1〜C4 の矢印で示されるように，自作のソースコードを Jact に送信する．ソースコードに対してテストが行われ，成功すればソースコードとして登録される．またテストの結果がどちらであっても，投稿者に通知される．

4. 実装

4.1 概要

Jact は JavaScript によって実装された Web アプリケーションである．ユーザインタフェースの実現のために，Vue.js を使用している．またエディタのシンタックスハイライトのために CodeMirror を使用している．Jact は REST API を経由してサーバからデータベースに保存されたタスク，ソースコードおよびテストの情報を取得している．API サーバには Node.js を，REST API の作成には Node.js のフレームワークである Express を使用している．タスクおよびソースコードの情報を保持するデータベースとして MongoDB を使用している．開発期間は約 6 カ月，コード行数は約 1,200 行である．

4.2 ソースコード比較

図 3 に Jact のソースコード比較画面を示す．JSF 検討者は図 3 に示される画面でソースコードの比較を行う．画面は左側のバーと右側のエリアに分割されている．左側のバーでは**タスクおよびソースコードの選択**，**タスク投稿画面への遷移**が可能である．右側の 5 分割されたエリアでは，それぞれ**選択したソースコードの表示**，**選択したソースコードのコピーや投稿用ソースコードの編集**，**ソースコードのプレビュー**，**テスト情報の確認・実行**，**処理時間計測情報の確認・実行**が可能である．

Jact に登録される典型的なタスクの例を**表 2** にあげる．表 2 にあげられる DOM 操作や Ajax 通信，コールバックは，Web フロントエンド開発における基本的な操作である．タスクを基本的な操作に設定することで，開発者は自分が実現したい機能に近いタスクを比較できる．

表 2 Jact で使用されるタスクの例

Table 2 Examples of typical tasks used in Jact.

タスクカテゴリ	タスク名
DOM 操作	テキストの変更
	複数テキストの色の一括変更
	クラスの追加
	クラスの削除
Ajax 通信	JSON の取得
	JSON の送信
コールバック	取得データによるページの書き換え
	入力のバリデーション

4.3 処理時間計測

処理時間計測時には，JSF 検討者が選択したタスクに対して各ソースコードがイベント発火から処理を完了するまでに要する時間を計測する．各ソースコードはそれぞれ 100 回実行され，100 回分の処理時間を合計して出力する．処理時間計測は，iframe と REST API を用いて各ソースコードごとに以下の手順で行われる．

(1) iframe を 100 個生成
同じ iframe を使用して実験を行う場合，キャッシュが適用されて正しく計測できない可能性がある．これを回避するために 100 回それぞれ別の iframe を利用して計測を行った．

(2) iframe のソースを API から取得
iframe の src 属性に URL を指定することで，当該フレーム内に Web ページを埋め込むことができる．Jact では REST API を使用することでソースコードが取得できるように設計しているため，その URL を iframe の src 属性に指定している．

(3) 監視対象の DOM 要素を指定
イベントの発火による DOM 要素の変更を検知するために，MutationObserver [*5] を用いた．MutationObserver を利用するための準備として，変更を監視すべき DOM 要素を指定する．

(4) 計測を開始
計測開始時の時間を記録し，イベントを発火する．

(5) DOM 要素の変更を検知
イベント発火による DOM 要素の変更が検知された時間を記録する．計測開始時間との差分を取り，処理時間を記録する．

(6) 100 回の処理時間を合算
(3)〜(5) を 100 回直列実行し，処理時間を合算する．

以上の手順をタスクに登録されたすべてのソースコードに対して直列実行する．すべてのソースコードの計測が終了したら，グラフで表示する．計測結果の表示画面を**図 4** に示す．計測結果の表示画面では**計測したタスク名**，**各ソースコードの処理時間のグラフ**，**実行環境**が表示される．

4.4 タスクの投稿

タスクの投稿時にはタスクの情報として**タスク名**，**投稿者名**，**サンプルのソースコードとその名称**の 4 つの基本項目が必要になる．それらの基本的な情報に加え，ソースコードのテストを行うための以下の情報も要求される．

Pre：ページ上の要素のうち，変更のターゲットとなる DOM 要素の初期状態を表す．ソースコードの振舞いを確認する際の事前条件に該当する．

*5 JavaScript に用意されている API の 1 つで，監視対象とした DOM 要素が変更されたときに指定したコールバック関数を実行することができる．

© 2020 Information Processing Society of Japan

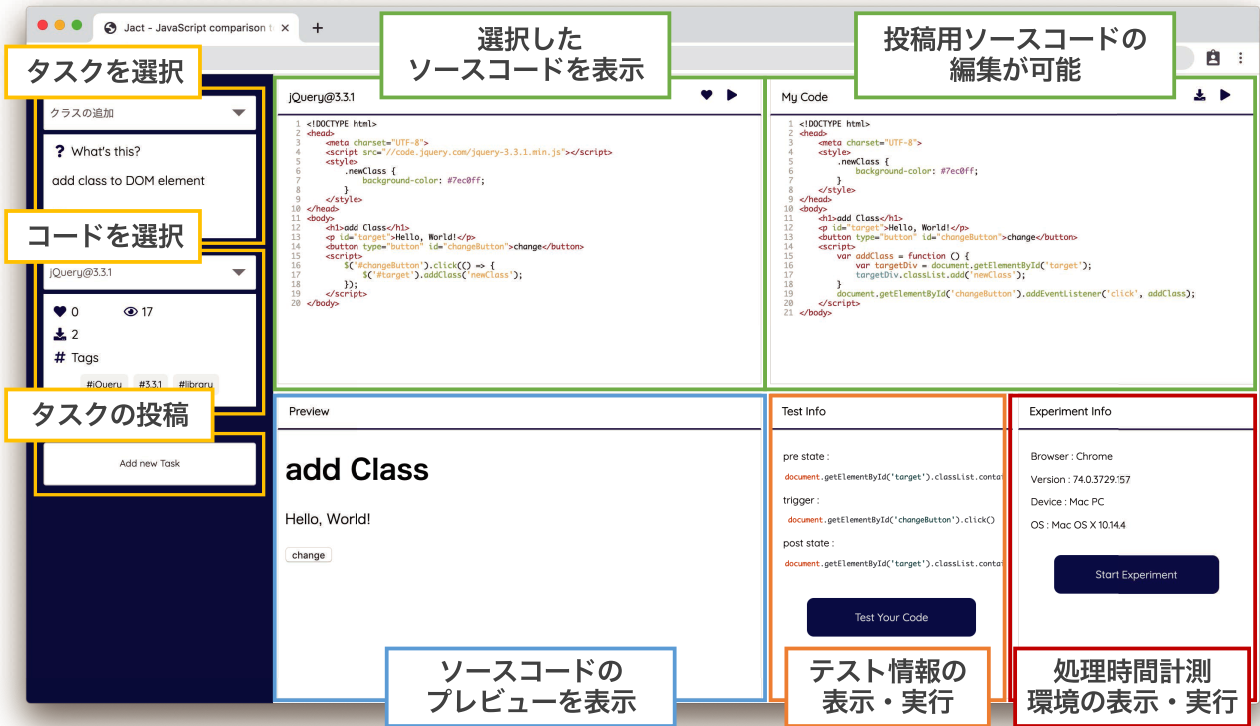

図 3　ソースコード比較画面
Fig. 3　Screenshot of Jact.

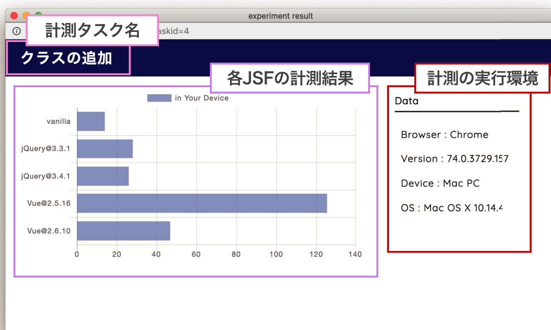

図 4　処理時間計測結果の表示画面
Fig. 4　Screenshot of the result of runtime performance measurement.

Task Registration Form
task name : change text
your name : nozomi
description : change text of specific element
code name : Vue@2.6.10
your code :
```
1 let app = new Vue(
2   {el: '#app', data: {msg: 'World'}});
3 app.msg='JS';
```
pre state : document.getElementById('target').innerText === 'World'
trigger : document.getElementById('button').click()
post state : document.getElementById('target').innerText === 'JS'
Run　Test　Submit

図 5　タスクの登録画面
Fig. 5　Screenshot of the task registration form.

Trigger：ボタンのクリック等，ソースコードにおける JavaScript イベントの発火方法を指す．

Post：イベント発火後のターゲットの期待値を指す．ソースコードの振舞いを確認する際の事後条件に該当する．

タスク投稿者は 4 つの基本項目と 3 つのテスト項目を送信することで，タスクを投稿できる．

具体的なタスクの登録画面を図 **5** に，タスク定義の JSON データ構造を図 **6** に示す．この例は図 1 の DOM 書き換えを題材としている．すべての定義項目の要素はテキスト形式で指定できる．サンプルソースコードやテスト項目等の処理にかかわる項目に関しても，JavaScript のソースコードとしてテキストで指定する．よって，Jact を通じて実行されるタスクの処理に限らず，Jact 自体が提供するタスクのテスト処理自体も JavaScript で実行される．これはプレイグラウンドの考えに基づいた実装であり，すべての振舞いがブラウザ上で実現されるため，高い可搬性を持つシステムの提供が可能となる．また，プレイグラウンドという性質から，cookie や localStorage，ページ遷移，サーバ通信といったブラウザ上で実現可能な機能もタスク内で利用することができる．

© 2020 Information Processing Society of Japan

```
{
  task-name: "change text",
  author: "nozomi",
  description: "change text of a specific element",
  example-source: {
    title: "Vue@2.6.10",
    source:
      "let app = new Vue(\n
         {el: '#app', data: {msg: 'World'}});\n
       app.msg = 'JS';",
  },
  test: {
    trigger: "document.getElementById('button').click()",
    pre-state:
      "document.getElementById('target')
        .innerText === 'World'",
    post-state:
      "document.getElementById('target')
        .innerText === 'JS'"
  }
}
```

図 6　タスク定義のデータ構造

Fig. 6　Data structure of task definition.

4.5　ソースコードの投稿

ソースコードの投稿時には，後に記述するテストが実行される．テストに成功した際にはモーダルウィンドウを表示し，**登録するソースコード名**，**ソースコードの特徴を示すタグ**，**投稿者名**を要求する．登録ボタンを押すとソースコードと入力された情報を登録し，ソースコード一覧に反映する．

4.6　ソースコードのテスト

ソースコードの投稿機能を実現するためにソースコードのテスト機能が実装されている．ソースコードのテスト時には，各タスクに登録された **Pre**，**Trigger**，**Post** を使用する．テストには処理時間計測時と同様に iframe を使用する．テストの手順は以下のとおりである．

(1) iframe の更新
テストを実行するために，iframe のソースコードをテスト対象のソースコードに変更しておく．

(2) iframe に対して Pre を実行
Pre として登録されたソースコードを実行し，ソースコードの事前条件を確認する．

(3) iframe に対して Trigger を実行
ソースコード内のイベントを発火させるために，Trigger として登録されたソースコードを iframe に対して実行する．

(4) iframe に対して Post を実行
Post として登録されたソースコードを実行し，ソースコードの事後条件を確認する．

(2) と (4) の実行に対して true が得られた場合，テストは成功となる．一方でも false となった場合や，イベントが正しく発火されなかった場合にはテストは失敗となる．

5.　評価実験

5.1　概要

本実験の目的は，JSF の記法と処理時間の理解において，Jact がどの程度有用であるかを確認することである．このために，被験者に実際に Jact を利用してもらい，各 JSF の記法の違い，およびそれらの処理時間の違いについてどの程度正しく理解できるかを確認する．また Jact のユーザビリティを確認するために，Jact を使用した印象やコメントを収集する．JSF の特徴把握のためのタスクには，著者らが作成した 4 つのタスクを利用する．理解対象の JSF としては jQuery と Vue.js の 2 つを採用する．jQuery の採用理由は利用率の高さである．jQuery は世界中の全サイトのうち 74%で利用されており，非常に利用率の高い JSF [*6] といえる．また Vue.js は開発者からの関心度が最も高い JSF として採用している．Vue.js の GitHub リポジトリは現在，JSF のなかで最も多くのスターを獲得している [3]．本実験では 2 つの JSF の他に，JSF を利用しない JavaScript のみの実装（以降，vanilla）を JSF 比較の基準として用いる．

5.2　被験者

被験者は大阪大学大学院情報科学研究科に所属する修士の学生 9 名，同研究科所属の教員 1 名，大阪大学基礎工学部の学部生 3 名の計 13 名を対象とした．被験者の JavaScript，jQuery および Vue.js の使用経験を**図 7** に示す．被験者のうち 5 名は JavaScript を使用した経験がない．また半数の被験者は jQuery および Vue.js を使用した経験がない．本実験では言語の使用経験がない被験者でもソースコードを理解できるよう，タスクを基本文法レベルに設定し，ソースコード中にはコメントを挿入した．

5.3　登録タスク

本実験用に以下の 4 つのタスクを Jact に登録する．

要素の選択：HTML 中の DOM 要素を id 名，および class 名で指定するタスク．

単一要素の書き換え：HTML 中の DOM 要素を id 名で指定し，単一の DOM 要素を書き換えるタスク．

複数要素の書き換え：HTML 中の DOM 要素を class 名で指定し，すべての DOM 要素を書き換えるタスク．

イベントの割り当て：ボタン要素を指定し，クリックイベントおよびマウスオーバイベントの割当てを行うタス

[*6] jQuery は厳密には JSF ではなくライブラリであるが，本実験では JSF の 1 つとして採用する．なお，JavaScript においてはライブラリとフレームワークの境界が曖昧だという指摘も存在する [21]．

© 2020 Information Processing Society of Japan

情報処理学会論文誌 Vol.61 No.10 1605–1616 (Oct. 2020)

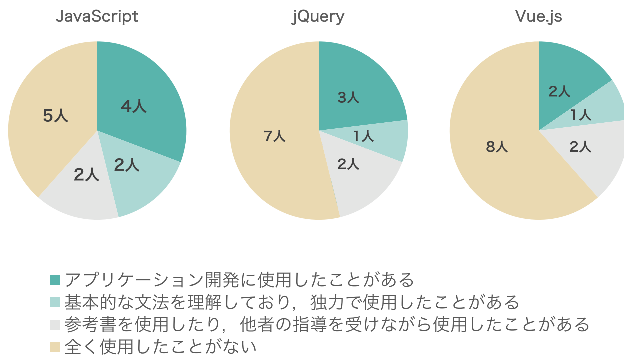

図 7 被験者の JavaScript 使用経験
Fig. 7 Usage experience of participants.

ク．このタスクは上記 3 つのタスクの内容を含む．

5.4 実験課題

実験課題として各 JSF（jQuery と Vue.js）に対して，ソースコードの記法の理解，および処理時間の理解の 2 つを行う．

5.4.1 記法の理解実験

5.3 節に述べた 4 つのタスクについて，Jact のソースコード比較機能とプレイグラウンドを利用する．vanilla と比較して jQuery，Vue.js の記法にどのような特徴が見られたかを各タスクごとに記述式で回答する．

5.4.2 処理時間の理解実験

Jact が提供する性能比較機能により，被験者が各 JSF の性能差を正しく理解できるかを確かめる．実験では vanilla を基準として，jQuery，Vue.js の処理時間にどのような特徴が見られるか，Jact の処理時間計測機能を用いて調査し回答する．5.3 節に述べたイベントの割当てタスクの処理時間を 2 種類以上のブラウザで計測し，以下の観点でどのような特徴が見られたかを記述式で回答する．

- vanilla と比較した各 JSF の特徴
- 各 JSF のバージョンごとの特徴
- 記法/API ごとの特徴
- ブラウザごとの特徴

各観点には，著者らが事前に確認しておいた性能差特徴の解（期待値）が設けられている．たとえば，vanilla との比較においては，どの JSF でも vanilla と比較して 10%～200%の速度低下が見られる，という解釈が解となる．また，JSF バージョンの比較においては，Vue.js の ver.2.5 と ver.2.6 を比較すると，ver2.6 で 200%の速度改善が確認できる．これは Vue.js の ver.2.6 で適用された，レンダリング処理のパフォーマンス改善[*7]によるものである．これら解と被験者らの回答を比較し，その一致率を性能理解の正答率とする．また，これら 4 つの観点に注目した処理時間計測結果をふまえ，JSF の処理時間の特徴についてにどのような感想を抱いたかを記述式で回答する．

5.4.3 ユーザビリティ評価

上記 2 つの実験課題の実施後，被験者に以下のアンケートを答えてもらい，Jact の各機能および全体の有用性を 5 段階評価で確認する．

- ソースコードの比較機能により，JSF の記法の違いを理解できたか
- プレイグラウンド機能は JSF の記法の理解に有用か
- 処理時間計測機能により，JSF の処理時間の違いを理解できたか
- 処理時間計測のオンデマンド性は JSF の処理時間の理解に有用か
- Jact は JSF の理解に有用か

また，その他 Jact に対する意見を自由記述で確認する．

5.5 実験結果

5.5.1 記法の理解

記法の理解実験で被験者から得られた回答を抜粋する．各タスクへの回答のうち，jQuery については以下のような回答を得られた．

> 対象要素の書き換えは vanilla では代入だが，jQuery では関数の呼び出しによる書き換えとなっている．

> jQuery では$を使うことで指定した ID(クラス) をもつ DOM 要素を取得できる．$の引数が “.” から始まる時はクラス，“#” から始まる時は ID になる．

> vanilla は “eventListener” を追加している．引数の 1 つ目がどんなアクションかで，2 つ目が 1 つ目のアクションが行われたときの処理 (関数) である．jQuery ではこのアクションごとにメソッドが定義されているため，引数として処理内容を与えることでアクションを追加できる．

これらの回答から，jQuery の特徴である$関数による DOM 要素の指定方法を理解できていることが分かる．また DOM 要素の操作やイベントの割当てを関数で行うという vanilla との違いに言及する回答が多く見られた．

次に Vue.js の記法の特徴について得られた回答を，以下に抜粋する．

> Vue.js ではインスタンス生成時にメソッドを定義することができる．定義や処理を別々に宣言することで，見やすくなると思った．

> Vue.js では，HTML において同じデータを参照する部分に同じプレースホルダーを書けば，対応するプレースホルダーのデータに再代入するだけでその部分が同時に書き換えられる．

[*7] https://medium.com/the-vue-point/vue-2-6-released-66aa6c8e785e

© 2020 Information Processing Society of Japan

> Vue.js では対象要素を事前に「変数」としてバインディングを行う．コンポーネント内の data でその値に代入を行うことで書き換えを行う．

Vue.js については，インスタンスの HTML との紐付けが必要であるという Vue.js の持つ特徴への言及が見られた．また，インスタンスにおいて変数やメソッドを宣言できることに着目した被験者も複数いたことが分かる．

5.5.2　処理時間の理解

処理時間の理解実験における，各比較の観点での正答率の結果は以下のとおりとなった．vanilla と比較した各 JSF の特徴：100%，各 JSF のバージョンごとの特徴：92%，記法/API ごとの特徴：85%，ブラウザごとの特徴：100%．どの観点においても 80%以上の正答率が得られており，被験者が正しく性能差を理解しているといえる．いくつか誤答も見受けられたが，いずれの誤答も 10%程度の性能差を，差ありととらえるか否かという解釈の違いによるものであった．さらに，本実験で得られた自然言語での感想を以下に抜粋する．

> 全く同じ機能を実装しているのにも関わらず，これだけの差があるということは知らなかった．また，Web アプリを開発する際にこれらのことを考えるのは面倒だと思った．その点で，Jact はその性能差を目で見て気軽に確認できるので良いツールだと思った．

> 数倍程度も差があるとは想像していなかった．処理速度を重視するようなプロジェクトでは，使用する JSF の選択も重要になると感じた．

> 簡潔に書けるもの程，実行処理にかかる時間が長くなる印象を受けた．

処理時間計測によって JSF の選択に対しての性能の認識が変わったという感想を得られており，被験者は各 JSF の処理時間に大きな差があることを理解できたといえる．その処理時間の差は記法の簡潔さとのトレードオフであることに気づいた被験者もいた．

5.5.3　ユーザビリティ評価

ユーザビリティ評価のアンケートによって得られた Jact に対する意見を図 **8** に示す．この結果から，本実験において与えたタスクにおいて Jact を利用することで，JSF の記法および処理時間の特徴を理解できたことが分かる．また，記法および処理時間の理解において Jact のソースコード比較，処理時間計測，プレイグラウンドは有用であることが分かる．Jact が JSF の理解に有用であると感じた被験者は 13 名中 12 名であった．

Jact に対する意見として収集した自由記述の回答を抜粋する．好意的な意見としては，以下のような意見があった．

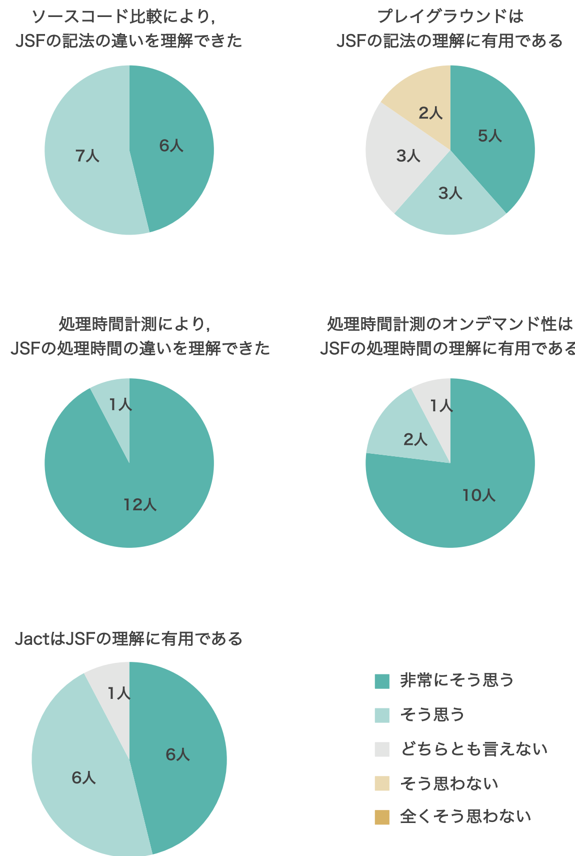

図 **8**　アンケート結果
Fig. 8　Results of usability questionnaire.

> 記法の差と性能差がこんなに手軽に知れるのは非常に良い．実際に使用したい．

> 登録されているコードと自分のコードを比較しながら作業できるのが非常に便利そうだと感じた．処理時間計測は性能を知るのにすごく便利な機能だと感じた．

一方で，Jact の改善点に関する意見も見受けられた．

> 差分がないところは色を薄くする等，JSF 間のソースコードの差分を強調表示してほしい．

> AltJS（TypeScript 等のコンパイルが必要な JavaScript の代替言語）にも対応してほしい．

5.6　考察

被験者実験の結果より，Jact の使用は JSF の理解に有用であると結論づけることができる．記法と処理時間の差を知る手段としての手軽さへの言及が複数見られた．また，実際の開発時に利用したいとの声も得られたことから JSF の利用を検討する開発者の利用も期待できると考えられる．

実際の開発現場では，本実験での設定とは異なり，最終的に導入する JSF を検討して選択する必要がある．その際，記法の分かりやすさや導入しやすさや処理時間等，優

© 2020 Information Processing Society of Japan

先すべき特徴は場合によって異なるうえに，表 2 に示したような，DOM 操作や Ajax 通信といった本実験のタスクよりも複雑な実装が要求される．そのような状況下では JSF の性質が開発するソフトウェアに大きく影響するうえ，比較したい条件での記法と性能の情報が必要になる．よって Jact のソースコード比較機能や処理時間計測機能の手軽さが，さらに JSF の理解に効果を発揮できると考えられる．

6. 妥当性の脅威

本研究で実施した評価実験の被験者のうち，半分以上が JSF を利用した経験がなかった．Jact は JavaScript や JSF の利用経験を持つ開発者からの利用を想定している．開発者による Jact の利用効果および評価は，評価実験の結果とは異なる可能性がある．

また，評価実験による Jact の評価はすべて被験者からの回答に基づくものであり，その内容は被験者の主観に強く依存している．そのため，設問内容や回答の方法等によって実験結果が変わる可能性がある．理解に要する時間の計測や，回答内容の正誤に基づく評価といった，定量的かつ客観的な制御実験による Jact の評価は今後の課題である．

7. 関連研究

本研究で採用した典型的なタスクに基づく比較という手法は，プログラミング言語間やアプリケーション間の比較でも広く用いられている [5], [10], [12]．これらの言語比較においては，同じタスクの様々な言語での実装を公開している Rosetta Code [*8]が利用されている．このようなタスクに基づいた比較は，対象の特徴を俯瞰的に列挙して比較するトップダウンな方法とは異なり，具体的なコードの視点から比較するボトムアップな方法ととらえられる．本研究で提案する Jact はボトムアップな JSF の比較支援技術であり，トップダウンな比較情報の補完として活用できる効果的な方法であるといえる．

本研究と同様，JSF の理解や選択の支援を目的とした研究もいくつか行われている．Pano らの研究では，JSF の選択における意思決定要因について調査している [14]．この調査によって，JSF のドキュメントには一般的なタスクを実装するためのコード例を含むべきであることや，開発者達が実際に小さなタスクを実現するコード例を作成し，JSF の理解に役立てていることが明らかになっている．本研究では，そのような開発者達が利用している実用的な手法を JSF の比較に採用している．また JSF の比較は，性能の観点でも行われている [6], [20]．Gizas らの研究 [6] では，JSF のソフトウェアメトリクスや実行時パフォーマンスを計測することで，JSF を比較している．Zochniak らの研究 [20] では，後述する TodoMVC を用いて，同じアプリケーションの実装に対する JSF の性能差を比較している．このほかにも，セキュリティ面での JSF の比較研究 [15] や特定の JSF の性質についての調査研究 [16], [17] が実施されているが，継続的に JSF の情報を収集し，提供することを目的とした研究は我々の知る限り存在しない．

*8 http://www.rosettacode.org

表 3 Jact と類似ツールの比較

Table 3 Features of Jact corresponding to the three problems.

性質	Jact	TodoMVC	js-fw-bench
ソースコード比較	○	○	○
処理時間計測	○		○
プレイグラウンド	○	○	
利用者による投稿	○	△[*11]	△[*11]
ソースコードのテスト	○		

JSF の比較によって JSF の理解支援を試みるツールも複数存在している，既存の JSF 比較ツールとしては，TodoMVC [*9]や js-framework-benchmark [*10]があげられる．TodoMVC は，様々な JSF を用いて実装した Todo 管理アプリケーションを提供している．ホームページではアプリケーションを実際に利用可能で，ソースコードは GitHub リポジトリにて公開されている．js-framework-benchmark は，JSF の性能を測定できるベンチマークである．GitHub リポジトリで公開されているソースコードをクローンし，ローカル環境で実行することで，実行環境下での各 JSF の性能情報を入手できる．これら JSF 理解支援を目的とするツールと提案手法の比較を**表 3** に示す．Jact はこれらのツールにはないプレイグラウンドを性質に加えており，ブラウザ上で動作させられる．そのため，環境構築をせずにソースコードの編集や動作確認ができるほか，処理時間の計測も様々なブラウザで容易に試すことができる．そして利用者からの投稿によって JSF の情報を随時取り入れ，JSF の比較に使用できる．また比較の基準を汎用的なタスクとし，小さな単位での比較とすることで，ボトムアップな JSF の性質理解が可能になっているほか，さらに複雑な機能を実装する際の拡張を容易にしている．

8. おわりに

本研究では JSF の選択における課題を解決するために，JSF の記法および処理時間を比較し，開発者の JSF 理解を支援するプレイグラウンド型ツール Jact を提案した．また評価実験として Jact を用いた JSF の理解支援に関する被験者実験を行った．実験では，JSF の記法および処理時間の特徴把握のタスクに対して Jact が有効であることを示した．また複数の被験者から Jact に好意的な意見を得ることができた．

*9 http://todomvc.com/

*10 https://github.com/krausest/js-framework-benchmark

*11 GitHub へのコミット push により投稿自体は可能だが専用の UI は設けられていない．

© 2020 Information Processing Society of Japan

今後の課題として，利用者による投稿機能の評価があげられる．本稿で実施した JSF 理解の実験は JavaScript の利用経験がなくても取り組める内容であったが，投稿機能の利用には JavaScript や JSF の利用経験が必要となる．今回の実験で得られた回答をもとに被験者を決定し，投稿機能に対しても評価を行うことで，投稿機能の有用性に関する評価や改善点についての意見を得ることができる．

謝辞 本研究の一部は，日本学術振興会科学研究費補助金基盤研究（B）（課題番号：18H03222）の助成を得て行われた．

参考文献

[1] Angular versioning and releases, available from ⟨https://angular.io/guide/releases⟩ (accessed 2019-07-22).

[2] Baysal, O., Davis, I. and Godfrey, M.W.: A Tale of Two Browsers, *Proc. 8th Working Conference on Mining Software Repositories*, pp.238–241 (2011).

[3] Collection: Front-end JavaScript frameworks, available from ⟨https://github.com/collections/front-end-javascript-frameworks⟩ (accessed at 2019/7/22).

[4] Deshpande, Y. and Hansen, S.: Web engineering: Creating a discipline among disciplines, *IEEE MultiMedia*, Vol.8, No.2, pp.82–87 (2001).

[5] Georgiou, S., Kechagia, M., Louridas, P. and Spinellis, D.: What Are Your Programming Language's Energy-delay Implications?, *Proc. 15th International Conference on Mining Software Repositories*, pp.303–313 (2018).

[6] Gizas, A., Christodoulou, S. and Papatheodorou, T.: Comparative Evaluation of JavaScript Frameworks, *Proc. International Conference on World Wide Web*, pp.513–514 (2012).

[7] Graziotin, D. and Abrahamsson, P.: Making Sense Out of a Jungle of JavaScript Frameworks, *Proc. Product-Focused Software Process Improvement*, pp.334–337 (2013).

[8] Lei, K., Ma, Y. and Tan, Z.: Performance Comparison and Evaluation of Web Development Technologies in PHP, Python, and Node.js, *IEEE 17th International Conference on Computational Science and Engineering*, pp.661–668 (2014).

[9] Murugesan, S., Deshpande, Y., Hansen, S. and Ginige, A.: *Web Engineering: A New Discipline for Development of Web-Based Systems*, pp.3–13, Springer Berlin Heidelberg (2001).

[10] Nanz, S. and Furia, C.A.: A Comparative Study of Programming Languages in Rosetta Code, *Proc. 37th International Conference on Software Engineering*, pp.778–788 (2015).

[11] Obbink, N.G., Malavolta, I., Scoccia, G.L. and Lago, P.: An Extensible Approach for Taming the Challenges of JavaScript Dead Code Elimination, *IEEE 25th International Conference on Software Analysis, Evolution and Reengineering*, pp.291–401 (2018).

[12] Oliveira, W., Oliveira, R. and Castor, F.: A Study on the Energy Consumption of Android App Development Approaches, *Proc. 14th International Conference on Mining Software Repositories*, pp.42–52 (2017).

[13] Osmani, A.: Learning JavaScript Design Patterns – A JavaScript and jQuery Developer's Guide, O'Reilly Media (2012).

[14] Pano, A., Graziotin, D. and Abrahamsson, P.: Factors and actors leading to the adoption of a JavaScript framework, *Journal on Empirical Software Engineering*, Vol.23, No.6, pp.3503–3534 (2018).

[15] Peguero, K., Zhang, N. and Cheng, X.: An Empirical Study of the Framework Impact on the Security of JavaScript Web Applications, *Companion Proc. Web Conference 2018*, pp.753–758 (2018).

[16] Ramos, M., Valente, M.T. and Terra, R.: AngularJS Performance: A Survey Study, *IEEE Software*, Vol.35, No.2, pp.72–79 (2018).

[17] Ramos, M., Valente, M.T., Terra, R. and Santos, G.: AngularJS in the Wild: A Survey with 460 Developers, *Proc. 7th International Workshop on Evaluation and Usability of Programming Languages and Tools*, pp.9–16 (2016).

[18] Ratanaworabhan, P., Livshits, B. and Zorn, B.G.: JSMeter: Comparing the Behavior of JavaScript Benchmarks with Real Web Applications, *Proc. 2010 USENIX Conference on Web Application Development*, p.3 (2010).

[19] Sillito, J., Maurer, F., Nasehi, S.M. and Burns, C.: What Makes a Good Code Example?: A Study of Programming Q&A in StackOverflow, *Proc. 2012 IEEE International Conference on Software Maintenance (ICSM)*, pp.25–34 (2012).

[20] Zochniak, A. and Walkowiak, T.: Performance Comparison of Observer Design Pattern Implementations in JavaScript, *Reliability and Statistics in Transportation and Communication*, Springer International Publishing, pp.466–475 (2018).

[21] 掌田津耶乃：JavaScript フレームワーク入門，秀和システム (2016).

中島 望

2018 年大阪大学基礎工学部情報科学科中退．2020 年同大学大学院情報科学研究科コンピュータサイエンス専攻博士前期課程修了．在学時プログラム理解やソースコード分析に関する研究に従事．

松本 真佑 （正会員）

2010 年奈良先端科学技術大学院大学博士後期課程修了．同年神戸大学大学院システム情報学研究科特命助教．2016 年大阪大学大学院情報科学研究科助教．博士（工学）．エンピリカルソフトウェア工学の研究に従事．

© 2020 Information Processing Society of Japan

楠本 真二 （正会員）

1988 年大阪大学基礎工学部卒業．1991 年同大学大学院博士課程中退．同年同大学基礎工学部助手．1996 年同講師．1999 年同助教授．2002 年同大学大学院情報科学研究科助教授．2005 年同教授．博士（工学）．ソフトウェアの生産性や品質の定量的評価に関する研究に従事．電子情報通信学会，IEEE，JFPUG，PM 学会，ソフトウェア技術者協会各会員．

© 2020 Information Processing Society of Japan