# 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)]　柗本 真佑[1,b)]　楠本 真二[1,c)]

**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．フレームワークはアーキテクチャパターンや記法といった実装面及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，実装や性能の単純な比較情報はすぐに古くなってしまい，比較したい条件での情報を収集することは難しい．そこで本研究では，フレームワークを用いたソースコードの比較，及び処理時間の計測が可能なプレイグラウンド型のフレームワーク理解支援ツール Jact を提案する．Jact は同機能を実現する各フレームワークのソースコードの比較機能，更にその処理時間をオンデマンドに計測する処理時間計測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 はブラウザ上でのソースコードの編集及び共有を可能にする．これらの機能により，Jact は各フレームワークの実装面及び性能面の情報を継続的に提供できる．また，Jact を評価するために実際のツールと被験者を用いた評価実験を行った．実験の結果，Jact のフレームワーク理解に対する効果が示された．

**キーワード**：JavaScript フレームワーク，フロントエンド開発，プレイグラウンド

## 1. はじめに

Web 周辺技術の進化及び爆発的な需要の増加に伴い，Web アプリケーションは複雑化の一途を辿っている [10]．アプリケーションのソースコードの複雑化を改善し，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 [4][10]．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，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 [5]．またアプリケーションの開発時に必要とされる機能が予め関数として用意されており，それを利用することでソースコードを簡略化できる [5][10]．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 は開発面での品質向上を目的としているため，パフォーマンスとのトレードオフが存在することも知られている [7]．そして適用する JSF の選択，すなわちアーキテクチャパターンの選択においては，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 [10]．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 の実装面及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 [5]．

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，以下の点が挙げられる．一つ目は，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一つの JSF を選び出すことは難しい．例えば 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 等が挙げられる．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．

[1] 大阪大学 大学院情報科学研究科
[a)] n-nakajm@ist.osaka-u.ac.jp
[b)] shinsuke@ist.osaka-u.ac.jp
[c)] kusumoto@ist.osaka-u.ac.jp

二つ目は，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み合わせにより，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 [10]，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 [4]．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．

三つ目は，情報入手の難しさである．Web 技術は成長が早く，JSF のみならず実行環境であるブラウザもアップデートが頻繁にリリースされる．例えば 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 ヶ月後の 10 月にバージョン 7.0.0 がリリースされている [1]．バージョン更新時には 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 の選択の支援を目的として，JSF を実装面及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 は一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，二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，三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 の記法及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，実行及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，JSFiddle や CodePen が挙げられる．利用者は実行環境を準備する必要がないため，ソースコードの動作をブラウザ上で手軽に確かめることができる．またプレイグラウンドを利用してソースコードを共有することで，ソースコードと共に実行環境も共有できる．

### 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目的で利用されているライブラリとは異なり，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 [14]．開発者は選択した JSF の記法に従ってアプリケーションを作成する．JSF はアーキテクチャパターンや記法，利用できる関数に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 [10]．

例として Vue.js と Knockout.js のソースコードを図 1 に示す．図 1 は Web ページに文字列"Hello!"を出力するソースコードである．Vue.js と Knockout.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ーフェースの作成を容易に行うことができる．

図 1: "Hello!"を表示するソースコード

(a) Vue.js

```
<div id="app">
  <p> {{ msg }} </p>
</div>
<script>
  var app = new Vue({
    el: '#app',
    data: {
      msg: 'Hello!'
    }
  })
</script>
```

(b) Knockout.js

```
<p data-bind="text:m"></p>
<script>
  ko.applyBindings({
    m: 'Hello!'
  });
</script>
```

### 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 [4][5][10]．JSF の選択における課題点として，以下の点が挙げられる．

#### 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，MVVM が挙げられる．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．例として同一のアーキテクチャパターンを採用している Vue.js と Knockout.js を挙げる．図 1 に示した通り，Vue.js は HTML 中に変数を埋め込むが，Knockout.js では HTML 中に変数を埋め込まずに HTML タグ内に紐付けるデータを明示する．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．

*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/

#### 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 [11]．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 [4] 等で示される通り，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リリースされる．例えば Java のフレームワークである Spring Framework はメジャーバージョンのリリースに約 4 年を要しているが [12]，JSF の一つである Angular はわずか 5 ヶ月で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]．他のソフトウェア開発技術と比較しても Web 技術の発達速度は速く，実装方法や性能の比較情報の寿命は短いと言える．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．

## 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．またアクタは Jact から**タスク及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 の実装面及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頻繁に用いられる機能を意味しており，DOM 操作や Ajax 通信等が該当する [4]．汎用的なタスクを登録しておくことで，Jact の利用者は実装したい機能に拡張できる．

表 1: Jact の各性質と課題 $P_1$〜$P_3$ の対応

| 性質 | $P_1$ | $P_2$ | $P_3$ |
|---|---|---|---|
| コード比較 | ○ | | |
| 処理時間計測 | | ○ | |
| プレイグラウンド | ○ | | ○ |
| アクタの投稿 | | | ○ |
| ソースコードのテスト | | | △[*3] |

[*3] ソースコードのテストはアクタ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．

#### 3.2.2 ソースコードの比較

あるタスクを実現する各 JSF を用いた様々なソースコードの比較を実現することで，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．想定される実行環境で本ツールを用いることで，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共に，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 [10]．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．

また 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．JSFiddle や Codepen 等のプレイグラウンドとは異なり，タスクごとにソースコードを共有することで JSF の情報を共有でき，より多くの情報を用いた JSF の比較が可能になる．

#### 3.2.5 アクタによる投稿

Jact のプレイグラウンド化を実現するための性質である．タスク及びソースコードを管理者が用意するだけでは，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．

#### 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全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 は図 2 に示す三種類のアクタを想定しており，アクタ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 の特徴を把握するアクタである．

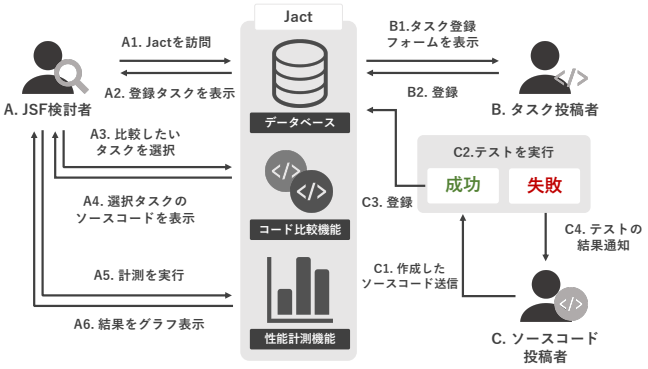


図 2: Jact の利用の流れ

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 の性能を計測できる．

**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アクタである．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して，自作のソースコードを投稿するアクタである．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 に投稿する．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．

## 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 を一部使用している．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，ソースコード及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．タスク及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 ヶ月，コード行数は約 1,200 行である．

### 4.2 コード比較

図 3 に Jact の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**タスク及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．

現在登録されているタスク及びソースコードを表 2 に挙げる．表 2 に挙げられる DOM 操作や Ajax 通信は，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．

### 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毎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[*4]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

表 2: 用意されているタスク

| タスクカテゴリ | タスク名 |
|---|---|
| DOM 操作 | テキストの変更 |
| | テキストの色の変更 |
| | 複数テキストの一括変更 |
| | 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 |

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

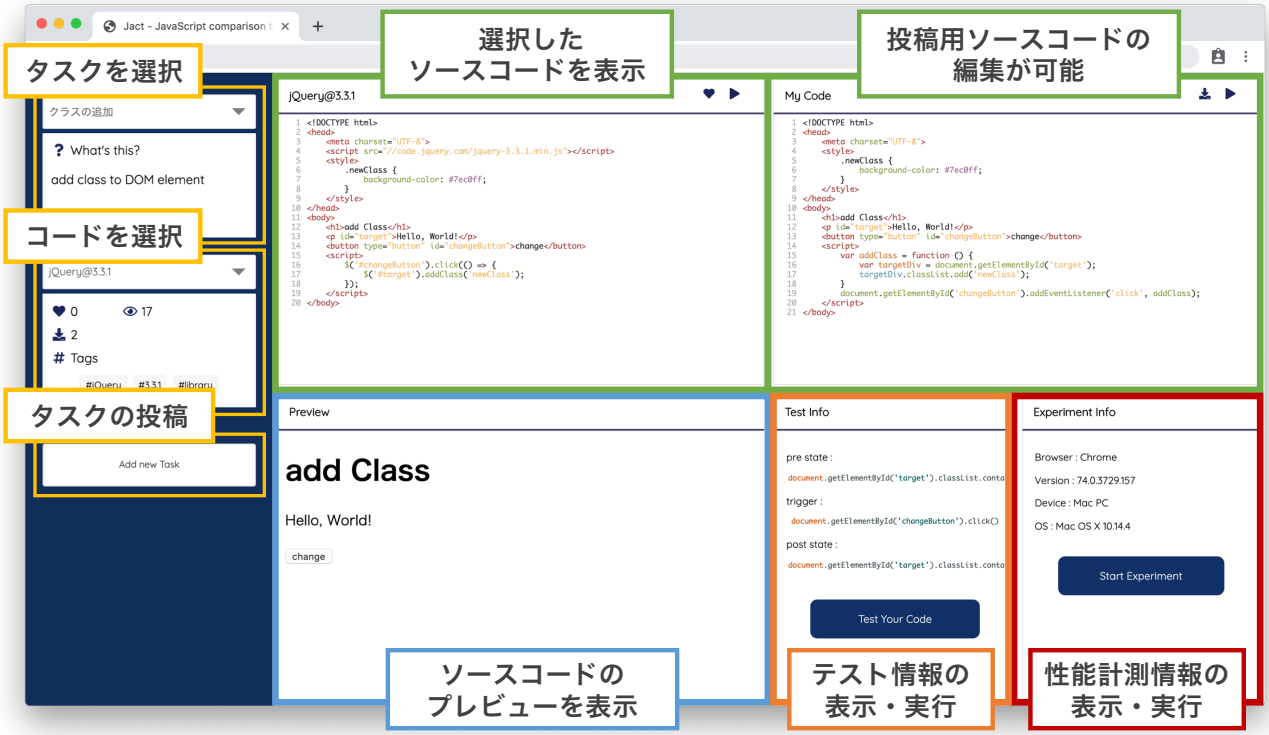


図 3: コード比較画面

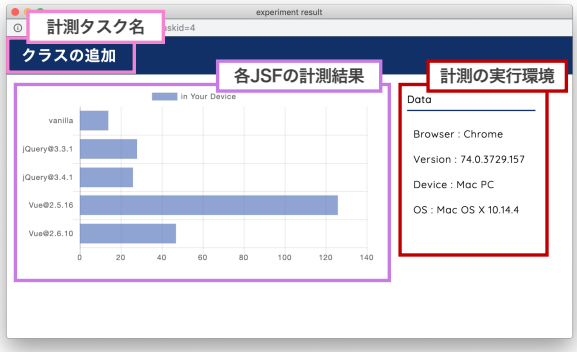


図 4: 処理時間計測結果の表示画面

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全てのソースコードに対して直列実行する．全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．ソースコードの振る舞いを確認する際の事前条件に該当する．

**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．ソースコードの振る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．

### 4.5 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通りである．

(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，及びそれらの性能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 [*5]と言える．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 の中で最も多くのスターを獲得している [2]．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 及び Vue.js の使用経験を図 5 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 及び 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．

[*5] 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 の一つとして採用する．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 [14].

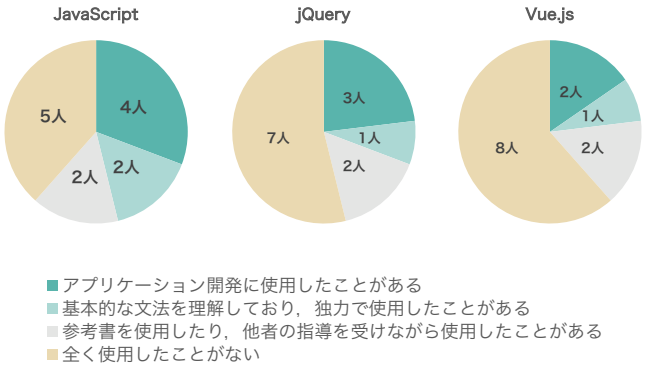


図 5: 被験者の JavaScript 使用経験

**要素の選択**：HTML 中の DOM 要素を id 名，及び 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，全ての 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，クリックイベント及びマウスオーバーイベントの割り当てを行うタスク．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，及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

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り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

また，これら 4 つの観点に注目した処理時間計測結果を踏まえ，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 の各機能及び全体の有用性を 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
- オンデマンド計測機能は JSF の性能の理解に有用か
- 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，

> 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．

> 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．

> 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わかる．また DOM 要素の操作やイベントの割り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．

> 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わかる．

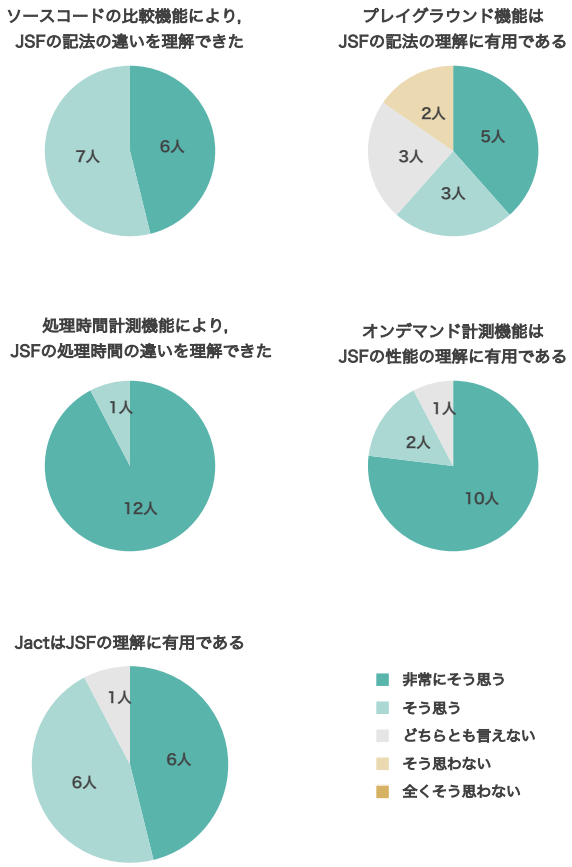


図 6: アンケート結果

#### 5.5.2 処理時間の理解

処理時間計測機能を利用し，5.4.2 項で述べた 4 つの観点で結果を考察してもらった後，どのような感想を得たかという質問について得られた回答を以下に抜粋する．

> 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言える．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 6 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 の記法及び処理時間の特徴を理解できたことがわかる．また，記法及び処理時間の理解において Jact のソースコード比較，プレイグラウンド，処理時間計測は有用であることがわかる．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．

> 記法の差と性能差がこんなに手軽に知れるのは最高だと思った．使用したいので，他の JSF にも対応してほしい．

> テンプレート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，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．

本実験で被験者に提示した 4 つのタスクは JavaScript の基本文法を元に作成しており，JavaScript の未経験者でも理解できる内容であった．実際の開発現場では表 2 に示したような，DOM 操作や Ajax 通信が用いられる．そのような状況では記法及び処理時間の違いは実装するソフトウェアに大きく影響する上，記法の素早い理解が要求されるため，Jact のコード比較機能や処理時間計測機能の手軽さは，更に 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

本研究では，評価実験の対象に，JavaScript の利用経験がない被験者を多く含んだ．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 の利用効果及び評価は，評価実験の結果とは異なる可能性がある．

また，評価実験で利用した 4 つのタスクは JavaScript の基本文法に基づくものであり，初心者でも理解できる内容に設定した．実際の開発では，表 2 に挙げられるような更に複雑なタスクが利用されているため，タスクの選択が実験結果の妥当性に影響を与えている可能性がある．

本研究で実施した Jact の評価は，全て定性的なものである．現在，Jact と同じ観点で JSF の理解を支援するツールが存在しないため，既存ツールと比較し被験者の理解度を定量的に評価することは困難である．そのため，本研究では全ての評価が定性的であり，その評価は全て被験者の主観に基づいている．

## 7. 関連研究

プログラミング言語の比較において，典型的なタスクに基づいた比較は効果的かつ実用的な手法である．タスクに基づいたプログラミング言語の比較研究は多く存在する [3][8][9]．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[*6]が利用されている．

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 [10]．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 の性能面での比較研究としては Gizas らの研究 [4] が挙げられる．Gizas らの研究では，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し，JSF を比較している．他にも JSF の選択や比較についての研究は実施されているが，継続的に JSF の情報を取得し，提供するための研究は我々の知る限り存在しない．

既存の JSF 比較ツールとしては，TodoMVC[13] や js-framework-benchmark[6] が挙げられる．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ることで，環境構築なしにソースコードの編集や動作確認を可能にしている．また汎用的なタスクをベースに JSF を比較することで，開発者が実現したい機能へ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 の記法及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 の記法及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，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．

今後の課題として，アクタによる投稿機能の評価が挙げられる．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

[*6] http://www.rosettacode.org

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: `https://angular.io/guide/releases` (accessed at 2019/7/22).
[2] Collection: Front-end JavaScript frameworks: `https://github.com/collections/front-end-javascript-frameworks` (accessed at 2019/7/22).
[3] Georgiou, S., Kechagia, M., Louridas, P. and Spinellis, D.: What Are Your Programming Language's Energy-delay Implications?, *Proceedings of the 15th International Conference on Mining Software Repositories*, pp. 303–313 (2018).
[4] Gizas, A., Christodoulou, S. and Papatheodorou, T.: Comparative Evaluation of JavaScript Frameworks, *Proceedings of the International Conference on World Wide Web*, pp. 513–514 (2012).
[5] Graziotin, D. and Abrahamsson, P.: Making Sense Out of a Jungle of JavaScript Frameworks, *Proceedings of Product-Focused Software Process Improvement*, pp. 334–337 (2013).
[6] js-framework-benchmark: `https://github.com/krausest/js-framework-benchmark`.
[7] Ladan, Z.: Comparing performance between plain JavaScript and popular JavaScript frameworks, Bachelor's thesis, Linnaeus University, Department of Computer Science (2015).
[8] Nanz, S. and Furia, C. A.: A Comparative Study of Programming Languages in Rosetta Code, *Proceedings of the 37th International Conference on Software Engineering*, pp. 778–788 (2015).
[9] Oliveira, W., Oliveira, R. and Castor, F.: A Study on the Energy Consumption of Android App Development Approaches, *Proceedings of the 14th International Conference on Mining Software Repositories*, pp. 42–52 (2017).
[10] 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).
[11] Ratanaworabhan, P., Livshits, B. and Zorn, B. G.: JSMeter: Comparing the Behavior of JavaScript Benchmarks with Real Web Applications, *Proceedings of the 2010 USENIX Conference on Web Application Development*, pp. 3–3 (2010).
[12] Releases · spring-projects/spring-framework: `https://github.com/spring-projects/spring-framework/releases` (accessed at 2019/7/22).
[13] TodoMVC: `http://todomvc.com/`.
[14] 掌田津耶乃: JavaScript フレームワーク入門，秀和システム (2016).