情報処理学会論文誌 Vol.60 No.10 1896–1914 (Oct. 2019)

Web APIの習得容易性と相互運用性，およびその定量評価方法の提案と適用評価

山本 里枝子[1,a)] 大橋 恭子[1] 福寄 雅洋[1] 木村 功作[1] 関口 敦二[1] 上原 忠弘[1] 青山 幹雄[2]

受付日 2018年12月27日, 採録日 2019年7月3日

概要：クラウドの普及にともない，REST に準拠した Web API が企業の情報システムに広がり，Web API の利用や提供のためのソフトウェア開発が急速に増加している．そのため，Web API の品質がそれを利用したアプリケーション開発の生産性と品質に大きな影響を及ぼすことが明らかになってきた．従来のシステム内 API と異なり，Web API はリモートで実行され，ユーザと独立に変更される．これらの特徴は Web API のソフトウェア工学の新たな問題を提起しており，特に数が増えているエンタープライズ Web API を利用するユーザのリスクとなっている．本稿では，システム API と異なる Web API の品質面の特徴をとらえる試みとして 2 つの品質特性を定義した．Web API を利用するアプリケーション開発者のパースペクティブから，ユーザビリティの品質副特性である習得容易性と互換性の品質副特性である相互運用性が我々の課題に対応すると特定し，品質モデルを定義した．この品質モデルに基づいて，尺度と定量的評価方法も提案する．本稿では提案する品質モデルを，Uber，WordPress，OpenStack，メディア処理を含む実際の Web API に適用した．提案したモデルを検証するため，Web API の習得容易性と相互運用性について実証的実験を行った．提案した品質の統計値と実験結果を比較し，提案した品質モデルと尺度の有効性を検証した．

キーワード：Web API，REST，品質モデル，品質特性，API ドキュメント，ユーザビリティ，習得容易性，相互運用性

Learnability and Interoperability of Web APIs and Those Quantitative Evaluation Method and the Practical Applications

Rieko Yamamoto[1,a)] Kyoko Ohashi[1] Masahiro Fukuyori[1] Kosaku Kimura[1] Atsuji Sekiguchi[1] Tadahiro Uehara[1] Mikio Aoyama[2]

Received: December 27, 2018, Accepted: July 3, 2019

Abstract: With the spread of cloud services, Representational State Transfer (REST)-based Web APIs are spreading to enterprise information systems and software development for the use and provision of Web APIs is rapidly increasing. So it has become clear that the quality of the Web APIs significantly influences the application quality and development productivity. Web APIs differ from conventional APIs in that they execute remotely on different servers and may change independently of their users. These unique characteristics introduce new problems in the software engineering of Web APIs, and impose risks to the users, especially those using enterprise Web APIs, whose numbers are increasing. In this paper, as an attempt to capture the quality features of the Web APIs different from the system APIs, we embody two quality characteristics. From the perspective of application developers using Web APIs, we identified that learnability as the subcharacteristic of usability, and interoperability as the subcharacteristic of compatibility, which correspond to our problems. And we defined the quality model for those. Based on this quality model, we also propose a set of measures and a quantitative evaluation method. In this study, we applied the proposed quality model and evaluation method to four types of actual Web APIs, including Uber, WordPress, OpenStack, and Media Processing. To validate the proposed model, we also conducted an empirical study of the usability and compatibility of the Web APIs. Our comparison of the proposed quality statistics with those from the empirical study validates the effectiveness of the proposed quality model and its associated measures of the learnability and interoperability of Web APIs.

Keywords: Web API, REST, quality model, quality characteristics, API document, usability, learnability, interoperability

1. はじめに

近年，Web API が企業情報システム開発のコア資産となっている [1], [2]．Web API はインターネットを介した Web サービスを提供する手段であり，REST（Representational State Transfer）アーキテクチャスタイルに準拠する [3]．Web API のグローバルなディレクトリ Programmable Web [4] には 2018 年 12 月時点で 20,000 を超える Web API が登録され，今後も急速に増加すると見込まれている．さらに，Web API を基礎とするソフトウェアとビジネスのエコシステムが国内外で成長している [1], [5]．また，企業がエンタープライズ Web API として戦略的に Web API をパートナやコミュニティに公開し，新しいサービスやアプリケーションの開発を促し，Web API ビジネスを展開する動きがある [2]．このように Web API は今後の企業情報システム開発のプラットフォームとして戦略的な研究開発が必要である．

しかし，企業情報システム開発に Web API を利用するには多くの課題がある [6], [7]．Web API は REST [3] を基礎としてそのインタフェース定義は実装言語と独立なリソースのテキスト表現（Representation）となるため，非形式的で，かつ，標準化もされていない．実際，多くのインタフェース定義の記述形式は個別で形式性と厳密性に欠けている．

企業情報システム開発に Web API を利用するためには品質保証は不可欠である．しかし，Web API はそのユーザとは独立にリモートで開発，保守され実行されることから，従来のシステム内の API（以下，システム API と呼ぶ）のようなユーザの実行環境上でインタフェースチェックができない等の問題があり，新たな研究課題を提起している．さらに，オープンなサービスとして公開され，アジャイル開発，DevOps 等の開発，提供形態の拡大にともない，Web API が短期間に，あるいは，継続的に変更される場合も発生している [8], [9]．

このような背景のもとに Web API に関するソフトウェア工学の研究が必要とされているが，従来のシステム API と比較し，その研究はきわめて少なく萌芽的な段階にある [13]．

本稿では，企業情報システム開発を対象に，Web API の品質問題に焦点を当てる．そのため，Web API が持つ従来のシステム API と異なる性質に着目し，品質モデルの定義とその具体的な品質特性を明らかにする．さらに，その品質特性を評価するための尺度と測定方法を提案する．

[1] 株式会社富士通研究所
Fujitsu Laboratories Ltd., Kawasaki, Kanagawa 211–8588, Japan
[2] 南山大学
Nanzan University, Nagoya, Aichi 466–8673, Japan
[a)] r.yamamoto@jp.fujitsu.com

本稿の構成は以下のとおりである．2 章で背景と研究課題，3 章で関連研究を述べ 4 章でアプローチを説明する．5 章と 6 章で Web API 品質モデルと品質評価方法を提案し，7 章で実際の Web API に適用した結果を述べた後，適用結果に基づき，8 章で品質モデルの妥当性，有効性を考察する．

2. 背景と研究課題

2.1 Web API の基本的な概念

Web API の定義は必ずしも統一されていないが，本稿では “HTTP プロトコルを利用してネットワーク越しに Web サーバをアクセスする API（Application Programming Interface）” とする [10]．Web API に関する基本的な概念は以下である．

リソース（Resource）：URI（Uniform Resource Identifier）でアクセス可能な Web 上のすべてのプログラム，データ等をリソースと呼ぶ．URI を指定してアクセスした結果は表現（Representation）と呼ばれ，その形式には HTML，JSON，XML が用いられる [11]．

REST：REST は HTTP をベースとしたアーキテクチャスタイルであり，いくつかの設計原則が知られている [3]．そのなかでインタフェース定義にかかわる主要な設計原則は，1) すべてのリソースは URI で表される識別子により参照できる，2) そのリソースにアクセスするよく定義された操作のセット，“GET”，“POST”，”PUT”，”DELETE” を提供する，等である．

2.2 Web API のインタフェース定義例と実装例

Web API のインタフェース定義の具体的な事例を**図 1** に示す．事例は，ある企業の顧客として登録されたユーザが，ボタンを押下するだけで商品を発注できるサービスを想定し，発注，発注取消，等の Web API を公開している，とする．

図 1 は，ボタンを押下して注文する際に用いる Web API のインタフェース定義である．このインタフェース定義には，HTTP メソッドが “POST” であること，リソースにアクセスするには “/v1/orders” で特定される URI をとることが記述されている．さらに，リクエストメッセージには 1 個のボディパラメータが必要で，そのボディパラメータはユーザが押下するボタンの ID が記述されている．一方，レスポンスメッセージには “buttonID” に関連付けられたユーザの情報，注文された商品の情報，商品の配送にかかわる情報が含まれること，が記述されている．

このインタフェース定義には，リクエストとレスポンスのサンプルが Example として記述されている．このサンプルには，リクエストやレスポンスの項目値のサンプルやレスポンスが JSON 形式であることが記述されている．

この Web API を利用する開発者は，インタフェース定

POST /orders

Order Request

Order Requestエンドポイントでは，予めAromaボタンに登録されている商品の注文を行います．

Resource

POST /v1/orders

Authentification

OAuth 2.0 bearer token with the request scope.

Body Parameters

Name	type	optional	Description
buttonID	string		注文に使うAromaボタンのボタンID

Response

Name	type	Description
buttonID	string	注文に使われたAromaButtonのボタンID
userID	string	注文者のユーザID
userName	string	注文者の氏名
orderID	string	注文伝票ID
itemID	string	注文商品ID
itemName	string	注文商品名
amount	int	注文数量
orderDate	string	注文日
deliveryAddress	string	配送先の住所
deliveryDate	string	配送予定日
deliveryTime	string	配送予定時間

Error Response

Name	type	Description
errors	string	エラーのリスト
errors.statusCode	int	レスポンスのステータスコード
errors.code	string	レスポンスコード
errors.title	string	レスポンスのタイトル

Error Response一覧

Status	Code	Title
400	insufficient_inputs	The input is not sufficient.
400	invalid_value	The input's value is invalid.
503	service_unavailable	Aroma Develper Site is not availabled.

Example

Example1

Request

```
curl -X POST \
  -H 'Authorization: Bearer <token>' \
  -H 'Content-Type: application/json' \
  -d '{ \
    "buttonID": "b123" \
  }' \
  https://api.aroma.com/aromaAPI/v1/orders
```

Response

Status Code: 200 OK

```
{
  "buttonID": "b123",
  "userID": "ABCDE",
  "userName": "富士　通男",
  "orderID": "order98765",
  "itemID": "productA",
  "itemName": "紙おむつ(M) 50枚入り",
  "amout": "2",
  "orderDate": "2018/12/01",
  "deliveryAddress": "神奈川県川崎市中原区上小田中4-1-1",
  "deliveryDate": "2018/12/05",
  "deliveryTime": "時間指定無し"
}
```

図 1　Web API のインタフェース定義例
Fig. 1　Example of Web API interface definition.

義を読み取り，サンプルを参考にして Web API の仕様に合致した HTTP リクエストメッセージを送信し，レスポンスを処理するためのプログラムを実装する．

図 1 の Web API のサーバ側の実装の一部を**図 2** に示す．Java を実装言語としたプログラム例である．図 2 に実装されたサーバ側のプログラムにアクセスするため，クライアント側のプログラムでは図 1 から以下のような HTTP の POST メソッドを表現する文字列を作成し，サーバへリクエストとして送信する．

POST /v1/orders HTTP/1.1
...(略)...
{“buttonID”: “b123”}

このリクエストからは，レスポンスコードと図 1 に示した JSON 形式の文字列がレスポンスとして返る．Java の Java Remote Method Invocation であれば，メソッドを呼び出すのも Java で記述できて，型定義によるチェックが可能である．しかし，REST ではこのようにリソースが文字列として表現される等，型定義も明示的でない．

2.3　研究課題

Web API を企業情報システム開発で利用するために，Web API の品質を定量的に測定する必要がある．その際，Web API は従来のシステム API と異なる以下のような特徴を持つため，その違いを考慮した議論が重要である．

(1)　実装言語と独立したインタフェース定義

従来のシステム API は，その実装言語でインタフェースを定義することから，インタフェース定義を生成するツールも提供されている．しかし Web API のインタフェース定義は，実装言語とは独立に REST の原則 [3] に従うリソースの表現として文字列で定義される．そのためツール化されておらず，インタフェース定義の作成は人手に依存している．さらに，そのリソース表現は，同一リソースであっても，JSON や XML 等複数の形式をとりえて，自己記述メッセージ（Self-descriptive Message）として交換される．このため，型チェック等が適用できない．このようなインタフェース定義の品質モデルは未確立である．

(2)　ユーザと独立した進化

Web API はインターネットを介した Web サービスを提供する手段である．従来のシステム API がそのユーザの計算機資源の中に取り込んだライブラリ等をアクセスするのに対し，Web API はユーザの計算機資源とは異なる環境で開発，修正され，遠隔にサービスとして提供される．そのため，ユーザと独立して，機能の変更，追加が可能である．一方，利用できていた Web API が，ユーザが知らない間に変更され，その Web API を使ったアプリケーションが動作しなくなる，という問題にもつながる [9]．この

```
public class DeliveryItemsApiServiceImpl extends DeliveryItemsApiService {
    @Override
    public Response deliveryItemsPost(CreateDeliveryItemParam createDeliveryItemParam, SecurityContext securityContext) {
        try {
            checkParams(createDeliveryItemParam);
            DeliveryItemCreatedResponse responseData = registerDeliveryItem(createDeliveryItemParam);
            return Response.ok().entity(new ApiResponseMessage(ApiResponseMessage.OK, responseData.toString())).build();

        } catch(Exception e) {
            return Response.notAcceptable().entity(new ErrorResponse(e)).build();
        }
    }
```

図 2 Web API の実装例
Fig. 2 Example of Web API implementation.

ような問題に対して，Web API のプロバイダはガイドラインを策定しているが，その内容はプロバイダごとに異なり，統一や標準化されていないのが現状である [12].

このような背景のもとに，本稿では，システム API と異なる上記の特徴を持つ Web API を利用したアプリケーション開発において，アプリケーションの品質確保のために，アプリケーション開発の初期に上記の特徴を測定可能とすることを目的とする．開発の初期は，実際に Web API を使い始める前の作業が必要であり，Biehl [11] が “Test the API” と称したように，Web API に関する情報を収集してその適否を判断する．そして実際に使うまでの時間や仕様変更時の対応工数等を判断して，開発工数の見積りへつなげる．

そのため，以下の研究設問を設定する．

RQ1：システム API と異なる上記 (1)，(2) の特徴をとらえるための Web API の品質特性と測定方法は何か？

Web API を利用したアプリケーションの品質を確保するために，従来のシステム API と異なる Web API の特徴をとらえた，Web API の品質モデルを定義する．本稿では，Web API を用いたアプリケーション開発の初期を前提とする．Web API のインタフェース定義が実装言語と独立なリソースのテキスト表現であること，ユーザとは独立した開発サイクルが可能であること，等の品質にかかわる新たな課題を解決するための品質モデル，品質特性，尺度とその測定方法を明らかにする．

RQ2：上記 (1)，(2) の特徴をとらえるために提案する測定方法は実際の Web API に有効か？

実際に利用されている Web API を対象に，提案した品質モデルとその尺度，測定方法を適用し，妥当性，有用性を実証する．

3. 関連研究

3.1 Web API の急速な普及とその課題

Web API の急速な普及にともない，その価値の認識 [2], [5] とともにその技術課題も明らかになってきた [7]．しかし，従来のシステム API と比較し Web API に関する研究はきわめて少なく，萌芽的段階にあるといえる．さらに，従来主としてスマートフォンアプリケーションや Web アプリケーション向けであった Web API に対してエンタープライズ Web API と呼ばれる企業情報システム開発や B2B 向けの Web API が提供されるようになり，Web API の品質や Web API を用いた企業情報システム開発が重要な課題となっている [13].

3.2 API の品質とアプリケーション開発への影響分析

Web API の品質はその仕様記述ドキュメントの品質としてとらえられている [14]．Web API の品質は，それに関与するステークホルダのパースペクティブによる [15].

一般に，システム API，Web API を含む API 品質はそのユーザであるアプリケーション開発者に最も影響を及ぼすことから開発者のパースペクティブから評価すべきであると考えられている [6]．API 品質がそれを用いたアプリケーション開発の生産性に影響していることが実態調査と実証実験により明らかになっている [16], [17]．その中で API ユーザビリティの重要性が認識され，その研究が活発に行われてきた．ソフトウェア製品のユーザビリティの概念をシステム API へ拡張した API ユーザビリティの概念が提唱された [18]．さらに，この概念を Web API に適用した Web API ユーザビリティの概念が最近提唱されている [19]．Web API ユーザビリティがアプリケーション開発の生産性に及ぼす影響は開発者へのアンケート調査 [19] や実証実験 [16] によって明らかになっている．これらの結果から，Web API を用いた開発における開発者経験として DX（Developer eXperience）の重要性が認識されるようになっている [6].

しかし，これらの研究においては，ユーザビリティを含むいくつかの品質特性が示されているにとどまり，Web API の品質特性に関する包括的な定義は未確立である．

3.3 API 仕様記述における例示の効果

システム API の利用において，その仕様記述に例示を含むことの有効性が示されている [19]．この成果を Web API に適用し，Web API 仕様記述に例示を含む効果を開発比較実験により実証している [16]．また，アンケート調査に

よって適切な例示の必要性も示されている [20]．これらの結果から Web API 仕様記述のスタイルガイドラインの検討も提案されている [21]．しかし，従来の研究では例示の有無にとどまり，その構造や内容等に関する議論は見られない．

3.4 Web API の進化の問題

システムそのものの進化に関してはこれまで多くの研究がある [21]．この一領域としてインタフェースの進化，すなわちシステム API の進化に関する一連の研究がある [22]．これに対して Web API がシステム API とは異なる本質的な進化の特性として “実行時の独立した進化” がある．すなわち Web API のコンシューマが Web API を利用中にその Web API のプロバイダが独立に変更，削除する可能性があることである．これは Web サービス共通の特性として知られているが，サービス中断等の深刻な問題を引き起こすリスクとなる [23]．

そのため Web API コンシューマに影響のある API 変更を行う場合はエンドポイントを別に用意する等の方法で既存の Web API 利用ユーザに影響を与えないようにする，という設計プラクティスがある [10]．しかし，Web API プロバイダにとって複数のバージョンを並行して保守していくことは大きな負担であるため，最終的にはユーザが余儀なくバージョン移行を迫られるケースが少なくない [7]．さらに，Web API のプロバイダがそのプラクティスをまとめたガイドライン，たとえば，文献 [24], [25] 等においても，バージョンの扱いは統一されていない．バージョンごとにエンドポイント URI を分ける方法や HTTP ヘッダで指定する方法が勧告されており，アプリケーション開発者の負担となっている [12], [26]．

このような Web API の変更への対処を考えるうえで，過去に発生した変更内容を分析することは重要である．Li らは 226 件の Web API の変更を 16 のパターンに分類し，その中に従来のシステム API にない新たな変更パターンを 6 つ指摘した [27]．この研究を発展させ，Wang らは StackOverflow 上での質疑から，21 の変更パターンを特定し，その中で，7 つが新たなパターンとしている [9]．また，約 82%の変更が改版の途中で行われていることを指摘し，変更が改版によらず常時行われていることも指摘している．近年のアジャイル開発や DevOps の普及により，この傾向はいっそう顕著になる可能性がある．

しかし，Web API 進化のこれまでの研究は変化のパターン化にとどまり，Web API の進化がその品質特性としてとらえられるまでには至っていない．

3.5 ソフトウェア品質モデル

ソフトウェア品質の体系である品質モデルはソフトウェア工学の基礎的研究課題として長年多くの成果がある [28], [29], [30]．これらの成果と測定に関する国際規格 [31] を統合して，ISO/IEC 25000 シリーズが広く認知され，利用されている [32], [33], [34]．しかし，これらの成果は，単一システムの品質を対象としている．これに対して，Web サービス，Web API 等のネットワークで連携するシステム，あるいは，それらが構成するエコシステムでは，前述した新たな問題が提起され，その品質保証にも新たな課題があることが指摘されている [35]．しかし，このような Web API システムに対する品質モデルに関する研究は少なく，その品質モデルの確立が求められている．

4. アプローチ

本稿では，ソフトウェア製品の品質モデルを基礎として，Web API の特性とそれによって提起されている品質に関する新たな課題を解決するように品質モデルを拡張するアプローチをとる．そのための着眼点とそれに基づく具体的なアプローチを以下に示す．

4.1 Web API 品質に関与するステークホルダとそのパースペクティブの設定

一般に，品質要求は ISO/IEC/IEEE 29148 [36] で規定するようにステークホルダごとに異なる．また，品質モデル国際規格 ISO/IEC 25010（以下，ISO 25010 と略記）では，ステークホルダとして 1 次，2 次，間接の 3 つのユーザを設定し，各ユーザのパースペクティブから品質特性を議論している．しかし，このユーザモデルは単一システムが前提となっており，Web API では適切とはいえない．これに対して，ネットワーク化にともないステークホルダの多様化や様々な事例が指摘されている [25]．しかし，上述の Web API の特性をとらえたステークホルダの定義とそのパースペクティブからの品質特性の議論にまでは至っていない．そのため，Web API において一般に知られている次の 3 つのステークホルダ [11], [19] を定義し，そのパースペクティブを起点として品質をとらえる．

(1) API プロバイダ：Web API を実装し提供するバックエンドサービスの提供者
(2) API コンシューマ：Web API を用いたアプリケーション/プロダクトの開発者
(3) アプリケーションユーザ：Web API を利用したアプリケーションのユーザ

本稿では，Web API を使用するアプリケーション開発における API コンシューマのパースペクティブからの品質に焦点を当てる．**図 3** に，3 つのステークホルダの関係を示す．

ISO25010 はソフトウェア製品の提供者とそのユーザ間での品質を規定する．しかし，API プロバイダが提供する Web API はアプリケーションユーザが直接利用するわけではなく，一般にはそれ単独のユーザインタフェースも持

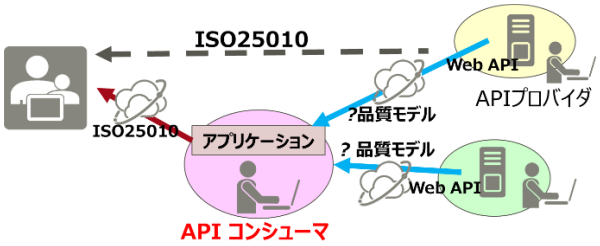


図 3 Web API 品質に関与するステークホルダ
Fig. 3 Stakeholders involving with quality.

たない．API コンシューマが開発するアプリケーションが Web API を利用し，そのアプリケーションがアプリケーションユーザに提供される．API コンシューマは開発者であり，開発者経験（Developer eXperience）の観点からも，ISO25010 を単純に適用するには議論が必要である．

4.2 ソフトウェア製品の品質モデルの拡張

ソフトウェア製品の品質モデルの国際規格 ISO 25000 シリーズは広く受け入れられ，実践においても普及している．本稿では，この品質モデルを基礎とし，Web API の提起する課題に対する品質特性を特定し拡張するアプローチをとる．これによって，従来の品質モデルとの整合をとり，実践における導入や活用を容易にする．

5. Web API の特徴をとらえる品質特性

5.1 API コンシューマの開発の観点からの分析

従来，個別に提案されてきた Web API のユーザビリティ等の品質特性 [15], [18], [19] の概念が提示されているが，ソフトウェア製品の品質モデルの国際規格 ISO 25010 [32] の分類と照らした議論は不十分である．

品質を評価する対象に関して，一般に，Web API を含む API は，その “インタフェース定義” と “実装” に分けられる．前述の図 1 が “インタフェース定義” を例示し，図 2 が “実装” を例示する．さらに，品質特性分類に ISO 25030 [34] で規定されている製品品質の外部品質と利用品質の概念がある．たとえば，利用品質の 1 つである効率性（Efficiency）は，API コンシューマにとっては Web API を利用した開発の効率性となり，アプリケーションユーザにとってはそれを利用して，ユーザがタスクを遂行する効率性を意味することから，異なる定義となる．このことは，Web API の品質特性として効率性の再定義が必要であることを意味している．

さらに，本稿では，Web API を利用したアプリケーション開発の初期工程で，実際に Web API をアクセスする前に判断すべき品質特性に着目する．実際に Web API をアクセスするには一般に ID 取得等の手続きや使用料支払い等の費用が必要となる．このような作業を行う前に，その必要性を判断し，また，利用上のリスクに応じて対応作業を予測できることが望まれている．これによって，その後のアプリケーション開発作業の見積りの適正化や工数確保等の準備ができる．そのため，Web API の “インタフェース定義” からその品質を評価する対象とし，“API コンシューマから見た外部品質” の視点で議論する．

5.2 非機能要求モデルに基づく品質特性の構造モデル化

API コンシューマのパースペクティブから Web API の品質特性を構造化し，ISO 25010 の品質特性に対する拡張性を議論するために，非機能要求の構造モデルである Chung らの非機能要求フレームワーク [37] を導入する．

非機能要求フレームワークに基づき，ソフトゴールと課題（非機能要求フレームワークではクレームと呼ぶ）と品質特性との関係を定義した結果を**図 4** に示す．表記法は非機能要求フレームワークに基づいているが，後述するように，一部拡張を行った．

本稿では “API コンシューマが開発に使いやすい” をソフトゴールとして，前述したシステム API と異なる Web API の 2 つの特徴 “実装言語と独立なインタフェース定義” と “ユーザと独立した進化” を Chung らの課題ととらえる．これらを計測するための品質特性を ISO25010 のユーザビリティ，互換性，保守性をベースに定義した．図の下部に API コンシューマが開発初期に Web API の品質を判断するためのソフトゴールを定義し，先に記述した Web API の特徴が提起する課題を関連づけている．Chung らの表記を用いて，ソフトゴールが副特性に positive に作用することを + で，negative に作用することを − で表現し，その度合いを + と − の数で表した．図の上部は ISO25010 の特性の一部と副特性の階層を記述した．図中，拡張した表記法として，API コンシューマが開発初期の課題に貢献する副特性は白色，関与しない副特性は黒色で示した．現在の ISO25010 を修正した特性は縦縞で示した．また，品質特性の修正部分を《修正》としても明示した．

インタフェース定義を対象にして，ソフトゴール “API コンシューマが開発に使いやすい” を前述の課題に対応するソフトゴールに分解し，“習得と利用が容易である”，“変更に対する対処が少ない” を定義した．

前者は品質特性 “ユーザビリティ（Usability）” の副特性の “習得容易性（Learnability）” が強く関与する．ユーザビリティの他の副特性に関しては，Web API がアプリケーションプログラムからアクセスされるため，アプリケーションユーザとのユーザインタフェースを主に想定した運用操作性，ユーザインタフェース快美性，アクセサビリティは関与しないと判断できる．適切度認識性とユーザエラー防止性は，“インタフェース定義” だけでなく “実装” の評価も必要となるので，関与の度合いが習得容易性よりも低いと判断した．

後者は，品質特性 “互換性（Compatibility）” と “保守性（Maintainability）” が関与する．“互換性” の副特性 “相互

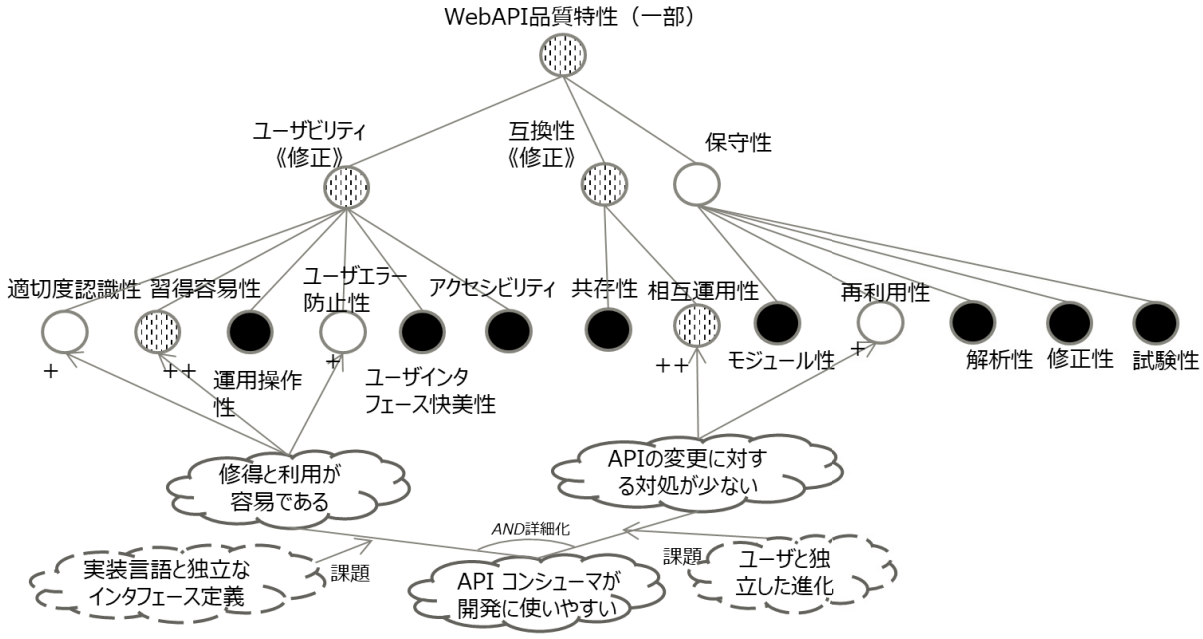


図 4 API コンシューマが開発初期に課題とする品質特性

Fig. 4 Quality characteristics concerned by API consumers in the early development phase.

運用性” は “2 つ以上のシステム，製品または構成要素が情報を交換し，すでに交換された情報を使用することができる度合い” [38] を表す．この定義から “API 変更への対処が少ない” に強く関与すると判断した．保守性は，副特性の再利用性も関連するが，“インタフェース定義” だけでなく “実装” の評価も必要となるので，関与の度合いが相互運用性よりも低いと判断した．保守性の他の副特性は “実装” を対象とするため関与しないと判断した．

相互運用性と関連する概念として，保守性から発展した “進化可能性（Evolvability）” が提案されており，システムが要求，環境，実現技術の変化に対応できる性質として定義されている [39]．これは，Web API が短期間に，あるいは，継続的に変更される場合も発生している [8], [9] ことから，Web API を利用するとき変更を肯定的かつ積極的に受け止めて対応すべきとの意味づけがされている．これに対して，API コンシューマのパースペクティブからは “変更への対処が少ない” ことが求められていることから，それに対して positive に強く関与する品質特性を明確にするため，相互運用性を採用する．

5.3 開発初期の課題のための Web API 品質特性の定義

図 4 と上述の議論から，Web API と従来のシステム API の特徴の差をとらえた品質特性として，Web API を用いたアプリケーション開発の初期に，API コンシューマにとって重要となる品質特性として以下に定義する 2 つを導出した．

(1) 習得容易性（Learnability）：習得容易性とは，ある特定のユーザがある特定の学習目標を達成するために Web API を使用することによる有効性，効率性，リスクと満足度の程度である．

ISO 25010 で定義されているソフトウェア製品に対する品質特性の習得性の拡張であり，ユーザビリティの一特性である．3.2 節で述べたように，システム API での API ユーザビリティの重要性は実証されており，また，Web API でも API ユーザビリティの重要性が認識されつつあるため，ユーザビリティを対象とする．システム API がその実装言語でインタフェースを定義するので一意の理解となるのに対し，Web API のインタフェース定義は実装言語と独立に REST の原則に従うリソースの表現として定義し，同一リソースでも複数の表現をとりうる．この性質の違いに着目して，ユーザビリティの中でもその起点となる習得容易性を本稿の対象とする．

(2) 相互運用性（Interoperability）：相互運用性とは，Web API のインタフェースがある時間にわたって維持される度合いであり，維持される度合いが大きい場合は相互運用性が高く，小さい場合は低くなる．

これは，Web API コンシューマが変更に対応するための開発期間への制約となる．この定義は，ライブラリの安定性 [40] を Web API 向けに解釈しなおし ISO25010 の定義の特性に対応付けしたうえで修正したものである．この修正は，Web API がシステム API と異なり，API コンシューマと独立にリモートで開発されて，実行時の “ユーザと独立した進化” をする特徴をとらえたものである．また，3.4 節で述べたように，Web API が変更されるとそれを利用するアプリケーションも変更せざるをえない状況が出てくる．これは，企業情報システムに大きなリスクとなること

から，開発の初期段階で Web API の変更を予測する必要がある．このために，本稿では相互運用性を対象とする．

6. Web API の品質評価モデルと品質評価方法

6.1 メタモデル駆動型 Web API 品質評価モデル

関連研究で述べたように，Web API を用いる開発の広まりとともに，本稿を含む様々な品質特性の議論が期待される．そのため，Web API の品質モデルの全体像として，概念，スコープ，語彙を定義するためのメタモデルとして，Web API 品質特性メタモデルを定義する．次に，各 Web API 品質特性の品質モデルをメタモデルのインスタンスとして定義する．メタモデルは Web API 品質モデルの基礎となり，モデルの一貫性を保証するために用いる．

6.2 Web API 品質評価メタモデル

ソフトウェア開発の視点からとらえた品質特性とその測定の共通認識の基礎として，ソフトウェアの汎用的な品質評価のメタモデルをまず定義する．ISO 15939 測定情報モデル [31]（プロジェクトで必要な情報を測定可能なプロセスやプロダクトと関係づけるモデルの定義）を参照し，品質特性とその評価に必要な要素のモデルを**図 5** に示す．

品質特性は階層的に構成し，その最下層の品質特性を測定するための尺度を関連づけている．評価対象システムは必要とする品質特性を選択し，測定結果と目標値を比較して品質を評価する．

6.3 習得容易性の評価

6.3.1 習得容易性の評価モデル

本稿では，習得容易性に注目して前節に示したメタモデルの品質特性を具体化した．具体化に際しては，開発初期の動作環境が整備される前であっても利用可能な情報，たとえば API ドキュメントを用いた尺度を定義することを目指した．

Web API の習得容易性の品質モデルを**図 6** に示す．この品質モデルでは，習得容易性を以下の 3 つの品質特性に詳細化した．各品質特性の概要と品質モデルの中で取り上げた理由を以下に示す．

(1) Web API サンプル充実性：Web API がその仕様だけでなくサンプルでも示されている程度．API コンシューマが Web API の選択や習得し始めようとした際にまず参照するのがサンプルだと想定したためである．

(2) Web API 標準適合性：Web API が設計原則の標準である REST [3] に則っていること．標準設計原則に則った Web API であれば，その仕様の理解が容易で，開発にも利用しやすいと考える．

(3) Web API サポート充実性：API コンシューマに対するサポートが充実している程度．Web API に限らず何かを習得しようとする際には，疑問点や意図したとおりにならないことが発生する．そのような場合に効率的に解決できることはアプリケーション開発に重要であるため，本品質特性を取り上げた．ここで，サポートとは API コンシューマ向けの情報提供サイトや FAQ 等である．

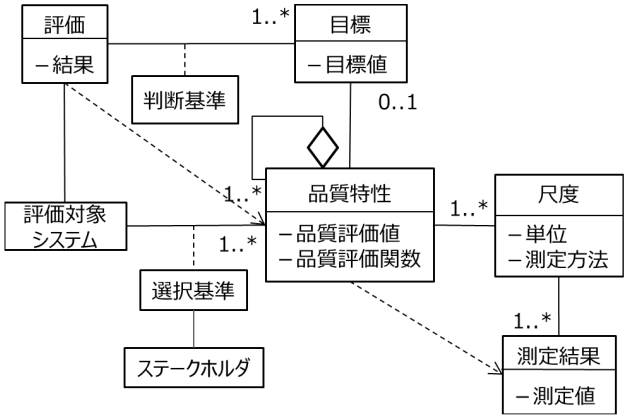


図 5 Web API 品質評価メタモデル
Fig. 5 Quality evaluation metamodel of Web APIs.

表 1 習得容易性の尺度
Table 1 Learnability metrics.

		粒度	
		全体	構成要素
対象	Req.	(1) リクエストサンプル記述網羅率 (ReqSampleCoverage)	(2)リクエストパラメータサンプル記述網羅率 (ReqParamCoverage)
	Res.	－	(3)レスポンスプロパティサンプル記述網羅率 (ResPropCoverage)

これら 3 つの品質特性のうち，以降では “(1) Web API サンプル充実性” について議論する．この品質特性を取り上げた理由は，品質特性の達成度合いの機械的な判断に最も適していると判断したからである．

6.3.2 習得容易性の評価尺度と評価方法

3.3 節で述べたように API のサンプルが習得容易性に寄与することが Uddin ら [14] や Sohan ら [16] により実証されている．本稿においてもサンプルの重要性を認識し，前節の品質特性 “Web API サンプル充実性” に注目して 3 つの尺度を定義する（**表 1**）．尺度は次の 2 つの観点から整理した．

a) 対象：サンプルが API のリクエストとレスポンスのいずれを対象としているか
b) 粒度：サンプルが，API のリクエストのサンプルであるか，あるいはリクエストやレスポンスを構成するパラメータやプロパティのサンプルであるか

以降，リクエストを Req., レスポンスを Res. と略す．

表中の “リクエストサンプル” とは，Web API の動作確認をするために最低限必要な HTTP メソッド名，エンド

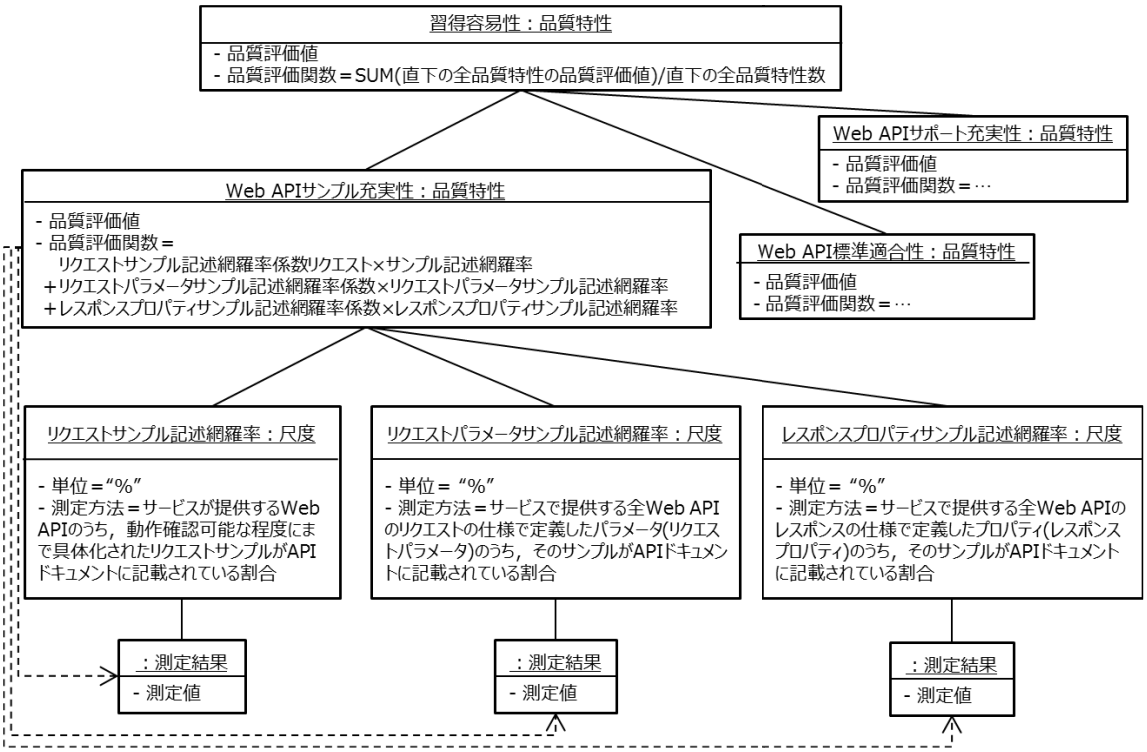


図 6 Web API 品質評価モデルの一部（習得容易性）
Fig. 6 Learnability quality model of Web APIs.

ポイントのパス，全必須パラメータと任意個のオプショナルなパラメータのサンプルの組合せである．

これら習得容易性の 3 つの尺度の定義を示す．

(1) リクエストサンプル記述網羅率（ReqSampleCoverage）：API コンシューマによる動作確認が可能な程度に具体化したリクエストのサンプルが提供されている程度を表す．定義を式 (1) に示す．値域は 0 から 1．すべての Web API に対してリクエストのサンプルが存在すると 1，まったく存在しないと 0 となる．

NumOfAPIs：サービスが提供する Web API の総数

NumOfAPIsWithSample：サービスで提供している全 Web API のうち，そのままで動作確認可能なリクエストサンプルを提供する API の数

$$\text{ReqSampleCoverage} = \frac{NumOfAPIsWithSample}{NumOfAPIs} \tag{1}$$

(2) リクエストパラメータサンプル記述網羅率（ReqParamCoverage）：リクエストに用いるパラメータの仕様に対して，その使い方を理解するためのサンプルが提供されている程度を表す．定義を式 (2) に示す．値域は 0 から 1．全パラメータにサンプルが存在すると 1，まったく存在しないと 0 となる．

NumOfParams：サービスで提供する全 Web API のリクエストの仕様で定義したパラメータの総数

NumOfParmsWithSample：サービスで提供する全 Web API の全リクエストパラメータの中で，そのサンプルを API ドキュメントに示したパラメータの数

$$\text{ReqParamCoverage} = \frac{NumOfParmsWithSample}{NumOfParams} \tag{2}$$

Name	Type	Description
buttonID	string	Aroma ボタンの識別子

(a) パラメータの仕様の例

```
curl -X POST \
  –H 'Authentification: Bearer <TOKEN>' \
  -H “Content-Type: application/json” \
  -d ‘{ \
  “buttonID“ : “b123 ” } \
  https://api.aroma.com/aromaAPI/v1/orders
```

(b) リクエストサンプルの例

図 7 Web API の定義例 1
Fig. 7 Example 1 of Web API definition.

リクエストのパラメータの仕様とそのサンプルの例を**図 7** と**図 8** に示す．また，図中の 2 個の Web API の例を用いてリクエストサンプル記述網羅率とリクエストパラメータサンプル記述網羅率の計算方法を示す．

まず，リクエストサンプル記述網羅率の計算例を示す．Web API は 2 個なので，NumOfAPIs は 2 である．各図の (a) に示した Web API の仕様に対して (b) に示したサンプルがあるので，開発されるアプリケーションごとに異なる部分，すなわち図中の〈TOKEN〉部分を除いて，そのまま使用して動作確認が可能なサンプルは 2 個である．したがっ

Name	Type	Description
buttonID	string	注文に用いる Aroma ボタン ID
userID	string	注文者のユーザ ID
itemID	string	注文する商品の商品 ID
amount	int	注文数量

(a) パラメータの仕様の例

```
curl -X POST \
  –H 'Authentification: Bearer <TOKEN>' \
  -H “Content-Type: application/json” \
  -d ‘{ \
  “buttonID“ : “b123 ”, \
  “userID”: “userA” } \
  https://api.aroma.com/aromaAPI//v1/deliveryItems
```

(b) リクエストサンプルの例

図 8 Web API の定義例 2
Fig. 8 Example 2 of Web API definition.

て NumOfAPIsWithSample は 2 である．NumOfAPIs と NumOfAPIsWithSample から，ReqSampleCoverage の値として 1 が得られる．

次に，リクエストパラメータサンプル記述網羅率の計算例を示す．図 7 (a) には 1 つのパラメータ buttonID，図 8 (a) には 4 つのパラメータ buttonID，userID，itemID，amount の仕様が記述されている．2 つの図で合計 5 個のパラメータの仕様が記述されているので，NumOfParams は 5 である．次に，図 7 (b) にはパラメータ “buttonID” のサンプルが含まれているが，図 8 (b) にはパラメータ “itemID” と “amount” の 2 つはサンプルに含まれていない．サンプルを持つパラメータは合計 3 個なので NumOfParmsWithSample は 3 である．NumOfParams と NumOfParmsWithSample から ReqParamCoverage の値として 3/5 が得られる．

これら 2 つの尺度の値は，リクエストやパラメータに 2 個以上のサンプルがあっても同値となる．これに対して，サンプル数を尺度の値に反映させることも検討した．しかし，次の 2 つの理由から，本稿では尺度の定義にはサンプル数とサンプルが含まれるドメインは反映していない．

1 つ目の理由は，API コンシューマにとって，サンプル数が 0 個か 1 個かの差は大きいと考えたが，2 個以上ではその差が有意になるとは限らないからである．2 つ目の理由は，習得容易性の達成度合いにサンプル数を考慮するのであれば，それが属するサンプルのドメインも考慮すべきであるが，現状ではドメインの判別が困難であると考えたからである．2 つ目の理由を詳細に説明する．

サンプルのドメインとは，テストケース設計技術の 1 つである同値分割によって得られるドメインに該当するものである [41]．Web API のリクエストやレスポンスのサンプルのドメインは，たとえば成功とエラーの 2 つが考えられる．ドメインテストと同様，同じドメインに属する Web API のサンプル数が増えてもそれらの情報量が増えるとは限らないため，API コンシューマの習得容易性の向上は期待できないと考えた．さらに，現状の Web API の仕様記述の規格 Open API Specification（OAI）[42] では，サンプルに名前を付けることはできるが，サンプルのドメインの判別はサンプルの名称や内容等から人間が判断せざるをえない．そのため，ドメイン判別の客観性を担保できないと考えた．

なお，次項で示す尺度レスポンスプロパティサンプル記述網羅率の定義においても同様に考えた．

(3) レスポンスプロパティサンプル記述網羅率（ResPropCoverage）：レスポンスを構成するプロパティの仕様を理解するためのサンプルが提供されている程度を表す．定義を式 (3) に示す．値域は 0 から 1．全プロパティにサンプルが存在すると 1，まったく存在しないと 0 となる．

NumOfProps：サービスが提供する Web API のレスポンスで，仕様として定義したプロパティの総数

NumOfPropsWithSample：サービスで提供する全 Web API の全レスポンスプロパティの中で，そのサンプルを API ドキュメントで示したレスポンスプロパティの数

$$\text{ResPropCoverage} = \frac{NumOfPropsWithSample}{NumOfProps} \tag{3}$$

ResPropCoverage の計算に必要なデータは，ReqParamCoverage と同様にして API ドキュメントから取得する．

6.4 相互運用性の評価

6.4.1 相互運用性の評価モデル

本稿では，相互運用性に注目して 6.2 節で示したメタモデルの品質特性を具体化した．3.4 節で述べたように Web API は変更が改版によらず常時行われているため，変更が発生した場合の追従に想定外の工数が発生する可能性がある．そのため，Web API を利用する前に保守の期間や工数を予測できることが重要である．従来のシステム API については Raemaekers らにより “ソフトウェアライブラリのユーザが対応を要するかもしれない，時間の経過にともない積み重なっていくインタフェースや実装の変更の度合い” として “安定性” が定義されている [40]．変更の度合いが小さければ安定性が高く，変更の度合いが大きければ安定性が低くなる．これは 5.3 節 (2) で定義した相互運用性の定義 “インタフェースがある時間にわたって維持される度合い” と同義である．

システム API の安定性の計測では，削除メソッド数，既存メソッド内の変更割合，新旧メソッドの変更割合比，新メソッドの割合の 4 つの指標が提案されている．それらの指標値の算出には実装上の API の変更を抽出する必要があり，2 章で述べたように Web API のインタフェース定義は実装言語とは独立に REST の原則に従うリソースの

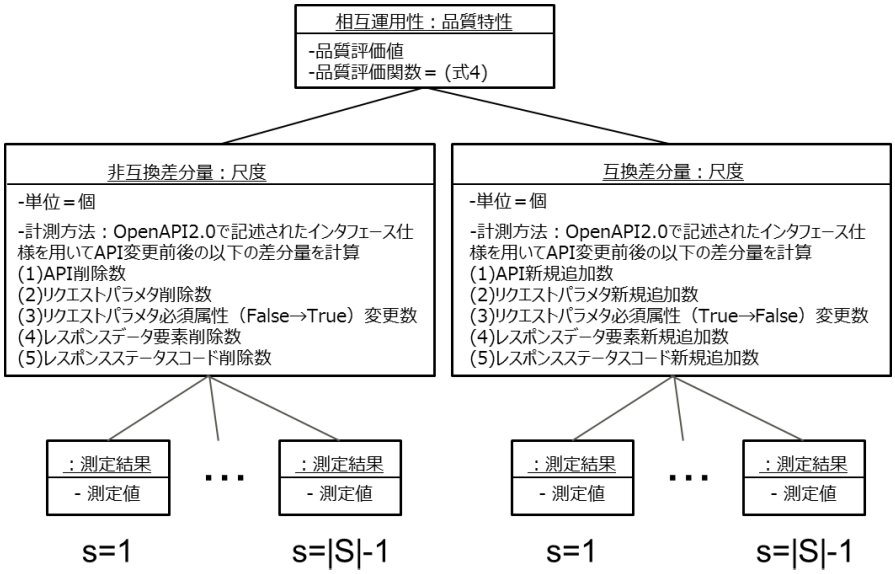


図 9 Web API 品質評価モデルの一部（相互運用性）
Fig. 9 Interoperability quality model of Web APIs.

表現として定義されるため，従来のシステム API に対する測定方法はそのままでは適用できない．そこで Li ら [27]，Wang ら [9] によって提案された Web API の変更のパターン分類と同様に，Web API の構成要素をもとに Web API の変更を評価する方法として，Web API の変更数を互換性有無の観点で区別して相互運用性の算出に利用する方法を提案する．特に，互換性のない変更は相互運用性の低下が大きく，互換性のある変更は低下が小さくなるように定義する．互換性のある変更は API コンシューマに影響を与えないと考えることもできるが，インタフェース上は変更がない場合においても振舞いが異なるために API コンシューマの対応が必要な場合があるため，本定義では相互運用性の低下をもたらすものとした．

また，変更履歴が利用できない場合，相互運用性を算出することはできない．そのような場合は相互運用性を N/A（Not Available）とし，将来発生するかもしれない Web API 変更が判断できない（Web API 変更に対する対処が発生する度合いが判断できない），というリスクがあることを明示することで Web API コンシューマに注意を促すこととする．

以上で説明した相互運用性の品質モデルを**図 9** に示す．

6.4.2 相互運用性の評価尺度と評価方法

Web API の変更量を算出するために，変更前後の 2 つの Web API の差分を抽出する必要がある．Web API は従来のシステム API とは異なり，ネットワーク上に存在する機能をリソースとして表現した文字列によって呼び出す．このため，従来の構文解析による抽出方法の適用は困難であり，Web API のインタフェース定義の差分抽出方法をとることが多い．本稿においてもインタフェース定義を利用して API の差分を抽出する方法を採用する．以降では相互運用性の品質モデルで採用した非互換差分量と互換差分量の算出方法について説明する．

(1) 非互換差分量

Web API に変更が発生した場合に API コンシューマが変更を必要とするような非互換な変更の要素数を非互換差分量と定義し，以下の測定値の総和とする．

a) API 削除数
b) リクエストパラメータ削除数
c) リクエストパラメータ必須属性（False→True）変更数
d) レスポンスデータ要素削除数
e) レスポンスステータスコード削除数

(2) 互換差分量

Web API に変更が発生した場合に API コンシューマが変更を必要としないような互換性のある変更の要素数を互換差分量と定義し，以下の測定値の総和とする．

a) API 新規追加数
b) リクエストパラメータ新規追加数
c) リクエストパラメータ必須属性（True→False）変更数
d) レスポンスデータ要素新規追加数
e) レスポンスステータスコード新規追加数

ソフトウェアで変更が発生した箇所は，最初は不安定な状態だが変更を重ねると安定する傾向をとる性質がある．このような時間的変化の傾向が指標値に利用されることがある [43]．従来のシステム API の安定性についても “時間の経過に応じて変更の影響は小さくなる” と仮定し，変更発生からの時間が重みとして利用されている [40]．この仮定は Web API においても成立すると考え，相互運用性の品質特性の算出に Web API に変更が発生してからの時間を重み $hw(s)$（s はソフトウェアの特定のスナップショットを示す番号であり，1 から始まり過去にさかのぼるにつ

表 2 評価式 (4) の変数定義
Table 2 Variable in Web API stability equation (4).

記号	説明
s	ソフトウェアの特定のスナップショットを示す番号，1 から始まり過去にさかのぼるにつれて増加する
S	すべてのスナップショットの集合
k_i	非互換差分量係数
k_c	互換差分量係数
$I(s)$	非互換差分量(s+1 から s の間に発生した差分量)
$C(s)$	互換差分量(s+1 から s の間に発生した差分量)
$hw(s)$	時間係数 $\frac{1}{2^s}$

れて増加する）として導入する．なお重み $hw(s)$ には “時間経過とともに影響が小さくなる” という仮定から，現在からの時間に反比例する $1/t$ 関数とすることも可能である．しかし，直近の過去の影響が大きくなり，経過時間によって影響がより小さくなるように，Raemaekers ら [40] の計算式を参考に時間の指数関数の逆数に比例する $\frac{1}{2^s}$ を利用する．また，6.4.1 項で述べたように，API コンシューマへの影響の大きさは (2) 互換差分量よりも (1) 非互換差分量が大きくなる．このため，相互運用性の品質評価関数において，(1) と (2) の重みを変えるために非互換差分量係数 k_i，互換差分量係数 k_c を導入する．

以上より，相互運用性の品質特性の評価関数を式 (4) で定義する．値域は 0 から 1 であり，1 に近ければ相互運用性が高く，0 に近ければ相互運用性が低いことを示す．この評価値により保守工数の推定が可能となる．評価関数で用いる変数を**表 2** に示す．

$$\text{相互運用性の品質評価関数} = \frac{1}{1+\sum_{s=1}^{|S|-1}(k_i I(s)+k_c C(s))hw(s)} \quad (4)$$

7. 品質モデルと評価方法の適用評価

7.1 習得容易性の適用評価

習得容易性を**表 3** に示す 4 つのサービスの Web API に対して測定した．測定したサービスの概要や特徴，尺度の測定方法，および測定結果を示す．

各サービスに対し，以下の尺度を用いて測定した．

(1) 尺度 “リクエストサンプル記述網羅率”

Web 上に公開されている API ドキュメントを参照しながら人手で測定した．OpenAPI Specification（OAS）2.0 形式 [42] のファイルを入力とし，API ドキュメントを生成するツール [44] が公開されている．このツールを利用して API ドキュメントを公開する場合，ツール機能を用いてリクエストサンプルを生成できる．このようなツールで API ドキュメントを公開し，リクエストサンプルが生成される場合は，“サンプルを有する” と判断する．メディア処理 API の API ドキュメントは，Swagger UI で公開しているため，本稿で提案した方法を用いて測定を行った．

表 3 測定対象のサービス
Table 3 Web services for evaluation.

サービス名(版)	概要や特徴
Uber (v1.2)	配車サービス向けのサービス
WordPress(V2)	Web ページやブログの作成を支援するサービス
OpenStack (Version Pike)	オープンソースで開発されているクラウドの IaaS 環境を構築するためのソフトウェアで，多くのコンポーネントから構成されている．本稿では，Ceilometer, Cinder, Designate, Glance, Heat, Keystone, Neutron, Nova, Swift のコンポーネントを利用した．外部に API を公開しているだけでなく，API 開発者も API を利用していることが特徴である
メディア処理 API(v1)	社内向けに公開されている画像や音声等の各種メディアを処理する Web API 群である．API ドキュメントは，Swagger UI で公開されている [44]．本稿では，本 API 群の API プロバイダに対し，API コンシューマの立場で良いドキュメントとは何かについて指導を行っていた

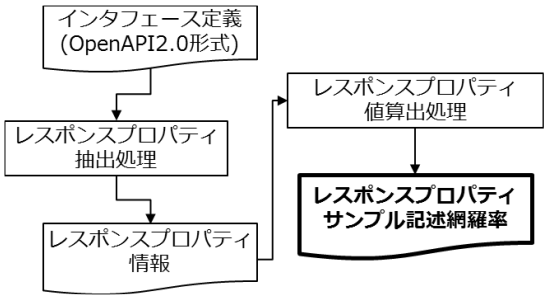


図 10 レスポンスプロパティサンプル記述網羅率の測定手順
Fig. 10 Measurement process of learnability quality.

(2) 尺度 “リクエストパラメータサンプル記述網羅率”

Web 上に公開されている API ドキュメントを参照しながら人手で測定した．

(3) 尺度 “レスポンスプロパティサンプル記述網羅率”

OAS2.0 形式のファイルに記載されたレスポンスプロパティの仕様とそのサンプルを取得して測定した．レスポンスは構造が複雑であることが多く人手では測定困難なためツールを作成した．ツールでの測定手順を**図 10** に示す．なお，API プロバイダから OAS2.0 形式のファイルが公開されていない場合は，ファイルを作成したうえで測定した．

Web API サンプル充実性の品質評価関数には各尺度の重要度を示す係数が含まれる．本稿では，一対比較法を行い各尺度の重要度を決定し，その値を品質評価関数の係数として用いることにした [45]．一対比較法の適用にあたっては，まず 5 段階の一対比較値を設定した．そのうえで，Web API を用いたアプリケーション開発経験のある 6 名が一対比較値を設定して尺度の重要度を計算した．

測定にあたり，前述のサービスが提供する Web API の一部を対象とした．測定対象の Web API 数を**表 4** に示す．

各尺度の係数，尺度の測定結果，および品質特性の品質評価値を**表 5** に示す．

表 4 測定対象の Web API 数
Table 4 Number of Web APIs measured.

サービス名	尺度(1)(2)の Web API 測定数	尺度(3)の Web API 測定数
Uber	10	4
WordPress	10	10
OpenStack	11	993
メディア処理 API	10	34

表 5 測定結果
Table 5 Measurement results of *"SufficiencyOfWebAPISample"*.

	尺度			品質特性
	(1)	(2)	(3)	Web API サンプル充実性
係数 / サービス名	0.77	0.16	0.06	—
Uber	1.00	0.82	0.91	0.96
WordPress	0.90	0.02	0.00	0.70
OpenStack	0.00	0.37	0.95	0.12
メディア処理 API	1.00	0.87	0.25	0.92

7.2 相互運用性の適用評価

相互運用性を算出する場合，6.4.1 項で述べたように以下の条件が必要である．

(1) Web API の時系列変化が確認できるデータが取得可能

Web API の仕様の変更履歴が公開されている場合と公開されていない場合がある．Web API の仕様の変更履歴が公開されている場合は，API Changes や Changelog という形で公開されている [46], [47]．一方，変更履歴が公開されていない場合においても，API の時系列変化を管理する API Management サービスが提供されている [6], [48], [49]．

本稿では Web API のインタフェース定義を利用して API の差分を抽出する方法を採用するため，さらに以下の条件を設定し，測定対象として OpenStack を採用した．

(2) Web API のインタフェース定義ファイルが構造化されたフォーマットで管理されており機械的に処理可能

OpenStack では Web API のインタフェース定義は各サービスのソースリポジトリに格納されている．またフォーマットは Python の reStructuredText を用いており，Web API の仕様を OpenStack 独自のルールに従って記載している [50]．測定対象期間は本フォーマットでのインタフェース定義の管理が始まった 2016 年夏ごろ（サービスによって若干異なる）とし，OpenStack の中心的なサービス群から Nova (Compute Service), Cinder (Block Storage), Glance (Image Service), Trove (Database Service), Ironic (Bare Metal Provisioning Service), Sahara (Big Data Processing Framework Provisioning), Manila (Shared Filesystems) を選択し，相互運用性を測定した．

図 11 に測定手順を示す．まず，サービスのリポジトリから各月のスナップショットとなるソースコードを取り出す．次に Web API のインタフェース定義である reStructuredText（.inc）ファイルを利用し，Web API の仕様記述として普及している OAS2.0 の形式に変換する．OAS2.0 に変換する理由は，Web API 差分を抽出するツール等の周辺ツールが揃っており，将来様々な Web API が OAS2.0 形式のインタフェースを公開した際に本方法が適用できるようにするためである．ただし，サービスによっては reStructuredText（.inc）ファイルが不完全なものが存在しており，OAS2.0 形式へ変換できないものも見られた．今回の評価では，1 サービスあたり 20〜60 の reStructuredText（.inc）ファイルを数カ月にまたがって処理する必要があり，1 つ 1 つ完全なファイルを手作業で作成することが困難であったことから，そのような不完全なファイルは調査対象外として除外した．除外したファイルは全体の 7.2%であり，適用評価全体への影響はない．

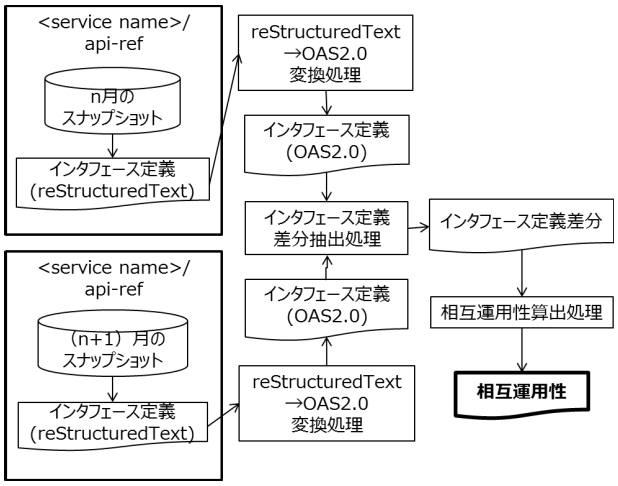


図 11 相互運用性の測定手順
Fig. 11 Measurement process of interoperability quality.

表 6 相互運用性の測定結果
Table 6 Measurement results of interoperability.

サービス名	重み付き非互換差分量 *1	重み付き互換差分量 *1	相互運用性
Nova	1.18	0.13	0.43
Cinder	11.35	2.76	0.07
Glance	0.00	0.05	0.95
Trove	0.00	0.00	1.00
Ironic	0.44	0.53	0.51
Sahara	0.01	0.00	0.99
Manila	0.73	0.30	0.49

そして，相互運用性の尺度である，非互換差分量，互換差分量を算出するために各月に発生する Web API 差分を抽出する．差分抽出には OSS である swagger-diff [51] を参考に，本品質特性計算用にツールを開発し，適用した．

表 6 に最新月における重み付き非互換差分量*1，重み付き互換差分量*1，相互運用性を示す．6.4.1 項で述べたように非互換の変更は API コンシューマに大きな影響を与える

*1 式 (4) に示す，時間と API コンシューマへの影響の重みを考慮に入れた差分量

ため相互運用性への係数に反映する必要がある．そこで，一対比較法 [45] で用いられる 9 段階尺度（等しく重要な場合は 1 点，とても重要な場合は 9 点をつける手法）を参考に，互換差分量を 1 ととらえた場合に非互換差分量を 9（とても重要）ととらえ，互換差分量係数 : 非互換差分量係数 = 1 : 9 となるように互換差分量係数 $k_c = 1/10$，非互換差分量係数 $k_i = 9/10$ とした．測定結果より Glance，Trove，Sahara は重み付き非互換差分量，互換差分量ともに小さく相互運用性が高いことが分かった．逆に Cinder は重み付き非互換差分量が多く相互運用性が低いことが分かった．これらと比較すると，Nova，Ironic，Manila は相互運用性が中程度であることが分かる．

8. 考察

8.1 RQ1：システム API と異なる 2.3 節の (1)，(2) で述べた特徴をとらえるための Web API の品質特性と測定方法は何か？

Web API を “HTTP プロトコルを利用してネットワーク越しに Web サーバをアクセスする API（Application Programming Interface）” と定義したうえで，システム API と異なる Web API の特徴として “実装言語と独立したインタフェース定義” と “ユーザと独立した進化” を指摘した．Web API は Java 等のプログラミング言語を用いて Web サーバ上で実装される．それをアクセスするためのインタフェース定義は REST に代表されるリソース表現であるため，システム API のような実装言語による型チェック等の厳格なチェックができない．また，リモートでアクセスするため，アクセス元のアプリケーションプログラムが動いていても，独立して修正や削除が行われる．アプリケーション開発においてこれらの特徴は開発工数を増やす原因となる．そのため，アプリケーション開発にかかわるステークホルダのうち，API コンシューマによるアプリケーション開発のしやすさに着目した．開発初期で実際に Web API を使い始める前にリスクを検知する目的で，非機能要求フレームワークに基づき，“API コンシューマが開発しやすい” をソフトゴールととらえてゴール分析を実施した．実際に Web API をアクセスする前なので，API ドキュメントを対象に測定可能であることを重視して，ISO 25010 の品質特性をベースに，ユーザビリティに関して習得容易性を特定し，また，互換性の副特性である相互運用性の修正を提案した．

習得容易性は，API ユーザビリティの一特性である．Web API の “インタフェース定義” が REST の原則に従うリソース表現として XML や JSON 等で定義され，ソフトウェアの仕様記述と類似の性質を有することを品質モデルに反映させた．

相互運用性は，システム API と異なり実行時に API ユーザとは独立に進化する Web API の特徴に着目して，Web API の構成要素の変更量に基づいて品質モデルを定義した．

なお，今後も継続する品質特性の議論のために品質モデルのメタモデルを最初に定義した．この考え方は，GQM [52] による品質特性から尺度を導き出すアプローチの一実現手段ともいえる．本稿のメタモデルを，ビジネスアプリケーションの UX 評価手法の開発にも適用し，メタモデルに従って品質特性とその尺度，測定方法を定義した [53]．実プロジェクトで，定義した尺度と測定方法が実測可能であることを検証した [53]．これにより，メタモデルとして妥当であることも確認した．

習得容易性に関して 3 つの尺度を定義し，その中で達成度合いを機械的な評価に適していると考えられる Web API サンプル充実性の尺度を定義し，実測可能であることを示した．相互運用性では，互換性に着目した尺度と測定方法を定義し，実測可能であることを確認した．

表 7 習得容易性の評価結果の比較
Table 7 Comparison of quality evaluation results.

サービス名	習得容易性(Web API サンプル充実性)の評価結果	実証実験での評価結果
Uber	0.96 <1>	0.9 <1>
WordPress	0.70 <3>	0.3 <3>
OpenStack	0.12 <4>	0.2 <4>
メディア処理 API	0.92 <2>	0.8 <2>

8.2 RQ2：2.3 節の (1)，(2) で述べた特徴をとらえる提案方法は実際の Web API に有効か？

8.2.1 習得容易性

品質評価関数の有効性を確認するため，API コンシューマ 7 名を被験者としてサービスの習得容易性を評価する実証実験を実施した．すべての被験者は，評価対象とした 4 サービスの Web API を用いたアプリケーション開発の経験はない．しかし，全被験者は一般的な API ドキュメントに記述される内容やその意味の知識を有している．また一部の被験者は API ドキュメントを参照しながら Web API を用いたアプリケーション開発の経験がある．

被験者は各 API ドキュメントを最大 20 分間参照したうえで，開発の容易性を 3 段階で評価した．3 段階で評価した理由は，段階数が多いと被験者が段階を選ぶ際に迷いが生じ，その結果選ぶ段階に誤差が生じると考えたからである．表 3 に示したサービスに対し，習得が容易と見なしたものを 1，困難を 0，中間を 0.5 でスコアリングを行い，回答結果の平均値を算出した．それらの結果を**表 7** に示す．表中の<番号>は品質評価値の順位を示す．本稿では “習得容易性” のうち “Web API サンプル充実性” に着目した．以後の議論では “Web API サンプル充実性” の品質評価値を “習得容易性” の品質評価値と見なす．

7.1 節で算出した習得容易性の品質評価値と実証実験での評価値の順位が一致した．したがって，尺度，測定方法，

品質評価関数は妥当であると判断できる．また，習得容易性の尺度の値を求める際に API ドキュメントのみを用いたので，開発初期に適用可能となる．

各サービスの品質評価値について考察する．

Uber は全リクエストにサンプルがあり，必須のパラメータやプロパティ以外は 1 個以上のサンプルが記載されていることが高評価につながった．

OpenStack の品質評価値が低いのは，リクエストのパラメータが階層構造を持っている Web API だけにしかサンプルが記載されていないからある．OpenStack の API コンシューマは OpenStack の API プロバイダでもあり，現状は問題視されていないと推察している．しかし，一般の API コンシューマにとっては習得が困難であると判断した．

WordPress はリクエストのサンプルはあるものの，オプショナルなパラメータにまったくサンプルを提示しない．さらにレスポンスのプロパティはサンプルだけでなく定義もまったく記載しないため品質評価値が下がる．

メディア処理 API は，API ドキュメントとしての記述項目の充実を図るための指導を受けていた．その指導によりリクエスト側の記述が非常に充実し，高評価につながった．一方，レスポンス側のサンプルが不十分であることが明確になったので，API プロバイダに対して改善に向けた提言を行うことができた．

この結果を Uddin らの実証結果 [14] と比較する．Uddin らは API ドキュメントの問題を 10 個あげ，それらを発生頻度，影響の大きさで分類している．その中で深刻さが高い問題として不完全さ（Incompleteness），曖昧さ（Ambiguity），説明不足（Unexplained examples），不正確さ（Incorrectness）の 4 つをあげている．これらの中で，曖昧さと説明不足の問題は，ドキュメント中のサンプルによって軽減できることが実証されている．これら 2 つの問題を，本稿で示した習得容易性によって定量的に評価できるようになる．なお，不完全さは他の Web API とのインタラクションの説明不足の問題，不正確さは API ドキュメントと実際の動作に差異があることによる問題である．不完全さは複数の API のドキュメントの横断的な分析が必要であり，不正確さの評価は Web API の実際の動作を確認する必要があるため，本稿で示した習得容易性では定量化できない．

8.2.2 相互運用性

Web API 向けの相互運用性の評価データは存在しない．そこで 6.4.1 項で述べたとおり “Web API を利用する前に保守の期間や工数を予測できるか” との観点で算出した相互運用性について考察する．

図 12 は相互運用性を算出した 7 サービスについて，各月における相互運用性をプロットし，時系列変化を示している．7.2 節では，相互運用性を高い，低い，中程度の 3 つに分類できた．時系列変化も，相互運用性が高くかつ変動量が少なく推移しているもの，低くかつ変動量が少なく推移しているもの，相互運用性が変動しているもの，の 3 つが確認できる．そこで，それぞれから代表的な 1 サービスをサンプルとし，実際に発生している Web API の変更量を確認する．

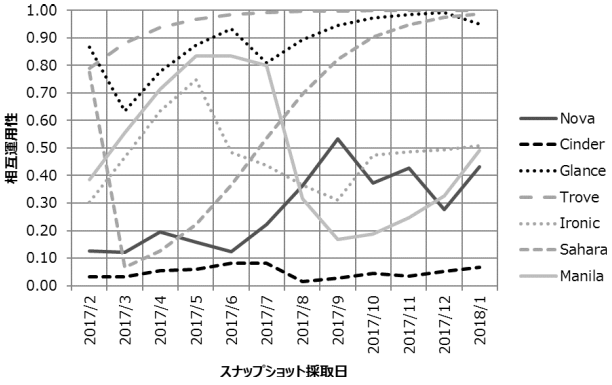


図 12 OpenStack の相互運用性の時系列変化
Fig. 12 Monthly transition of interoperability of OpenStack components.

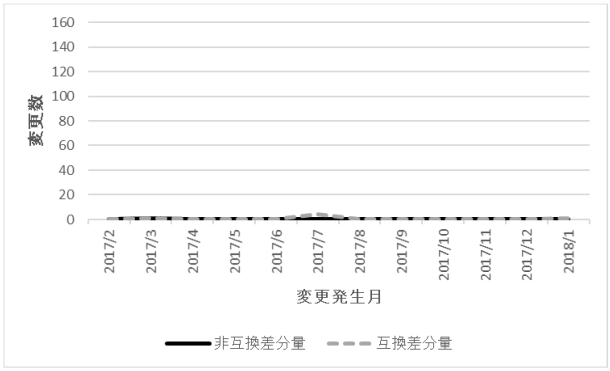


図 13 Glance の変更量の時系列変化
Fig. 13 Monthly transition of Web API changes in Glance.

(a) 相互運用性が高くかつ変動量が少なく推移しているもの：Glance

図 13 は Glance の非互換差分量と互換差分量の推移である．非互換差分は 2017 年 3 月に 1 件発生しているのみである．また互換差分は 2017 年 3 月に 1 件，7 月に 4 件，2018 年 1 月に 1 件と数カ月に 1 度程度でしか発生していない．そのため Web API コンシューマは，非互換変更によるインタフェース変更への対応はほとんど必要なく，また，互換変更による品質影響の確認もほとんど必要ない状態であるといえる．そのため，相互運用性の高さが Web API コンシューマの保守工数の低さを十分に表しているといえる．

(b) 相互運用性が低くかつ変動量が少なく推移しているもの：Cinder

図 14 は Cinder の非互換差分量と互換差分量の推移である．非互換差分量は互換差分量よりも少ないものの毎月

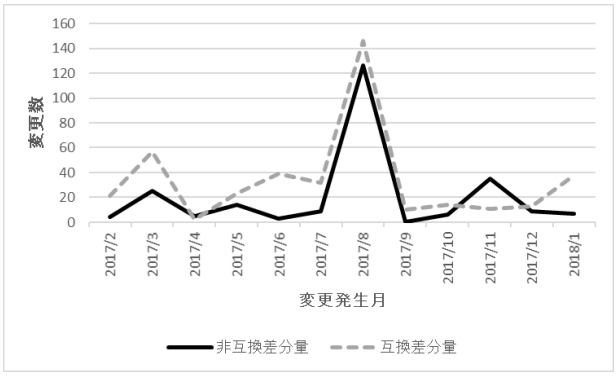


図 14 Cinder の変更量の時系列変化
Fig. 14 Monthly transition of Web API changes in Cinder.

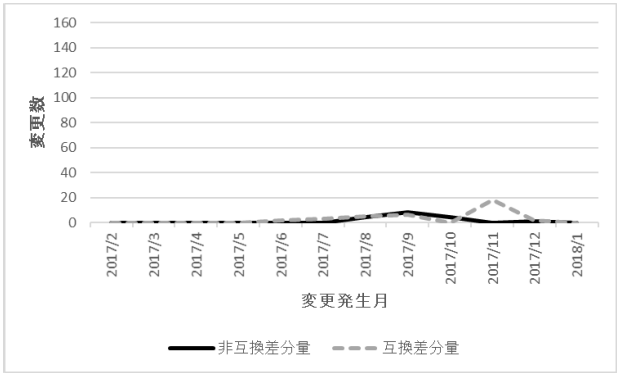


図 15 Manila の変更量の時系列変化
Fig. 15 Monthly transition of Web API changes in Manila.

10 件前後は発生しており，Web API コンシューマがつねにインタフェース変更へ注意する必要があることが読み取れる．また，互換差分量もつねに 20 件前後は発生している状態であるため，インタフェース変更に現れない内部品質にも注意が必要な状態であると推測できる．

(c) 相互運用性が変動しているもの：Manila

図 15 は Manila の非互換差分量と互換差分量の推移である．非互換変更に着目すると，2017 年 2 月から 7 月までは 0 件，また 11 月から 2018 年 1 月まででは 1 件しか発生していない．しかし，2017 年 8 月～10 月は合計で 16 件も非互換変更が発生している．互換変更については非互換変更と完全に重なっているわけではないが，非互換変更と同様に変更が発生している期間と発生していない期間がある．

このように時期によって変更量が大きく異なる場合は，その該当時期においては Web API コンシューマの対応工数が多くなることが推測できる．**図 16** は図 12 から Manila の相互運用性のみを抽出したものである．前述のとおり対応工数が多くなることが推測できる 2017 年 8 月～10 月には相互運用性が大きく低下しており，相互運用性が対応工数の増減を表していることが読み取れる．

上記の議論から，品質特性として定義した相互運用性により API コンシューマの保守に必要な対応工数の度合いを示すことができているといえる．この結果から相互運用性の品質モデルは，Web API の変更の評価尺度として妥当であり，開発初期での API コンシューマにとって有用な情報を提供できるといえる．

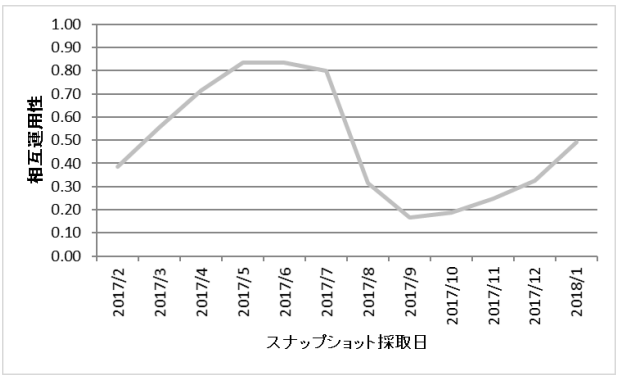


図 16 Manila の相互運用性の時系列変化
Fig. 16 Monthly transition of interoperability of Manila.

また，今回の測定において (c) で見られた “変更が発生する時期” について調査したところ，OpenStack のメジャーリリースの時期と重なっていることが分かった．たとえば，2017 年 8 月はメジャーリリース Pike の初期リリースが行われた時期となる．メジャーリリースでは大規模な機能追加が行われることが多い．そのような機能追加時はインタフェースの互換性を保つことが難しいため非互換変更が発生し，API コンシューマの対応が必要となることが多いと推測できる．この事例より，相互運用性の時系列変化を確認することで相互運用性の変動の傾向をとらえることができ，そのような兆候が見られる場合は事前に対応工数の上積みを行う等の対策をとることが可能である．

9. 今後の課題

本稿の測定では尺度の計算で利用する重みについて，一対比較法 [45] を用いて業務経験に照らし合わせて，重み設定時の条件を満たす値を選択した．今後様々なドメインでの測定で同様な考え方で調整して，適切な値を見分ける必要がある．また，本稿で定義した習得容易性と相互運用性は，アプリケーション開発の初期を想定して API ドキュメントを解析対象とした．Web API のインタフェース定義と実際の動作に齟齬がある実情があり，テストや動作確認をともなう “実装” に関する品質特性の議論が今後必要である．

10. まとめ

クラウドの普及にともない様々なサービスを Web API として提供し，アプリケーション開発に活用するようになっている．本稿では企業システム開発における Web API の利用を想定して，従来のソフトウェアシステムの製品品質モデルをベースとして，Web API の特性による課題をとらえた品質モデルを API コンシューマのパースペクティ

ブから提案した．

Web APIを利用したアプリケーション開発にかかわるステークホルダを整理したうえで，APIコンシューマの観点から見た外部品質に着目し，重要な品質特性として，ユーザビリティの習得容易性と互換性の相互運用性の概念を導出し，品質モデル，尺度，測定方法を提案した．提案した品質特性の測定方法を実際のWeb APIに適用して，測定可能であること，および，人による評価や保守に必要な対応工数の度合いを示すことを示した．このことから，提案した品質モデルはアプリケーション開発の初期段階でWeb APIの品質評価に有効であることを確認した．

このように，本研究は，Web APIの利用において新たに重要性が認識されている品質特性のモデル化とその定量的評価方法の確立，ならびに，従来のソフトウェア製品品質モデルをWeb APIを利用した企業情報システムの品質モデルへ拡張する1つのアプローチを提示した点で意義があると考えられる．

産業界ではすでにWeb APIを用いたアプリケーション開発が増加している．今後，Web APIを利用したアプリケーション開発の諸問題を解決するための研究を拡充していく．

参考文献

[1] Basole, R.C.: Accelerating Digital Transformation: Visual Insights from the API Ecosystem, *IEEE IT Professional*, Vol.18, No.6, pp.20–25 (Nov.–Dec. 2016).

[2] Iyer, B. and Subramaniam, M.: The Strategic Value of APIs, *Harvard Business Review* (Jan. 2015), available from ⟨https://hbr.org/2015/01/the-strategic-value-of-apis⟩.

[3] Fielding, R.T., Taylor, R.N., Erenkrantz, J.R., Gorlick, M.M., Whitehead, J. and Khare, R.: Reflections on the REST Architectural Style and "Principled Design of the Modern Web Architecture", *Proc. ESEC/FSE 2017*, pp.4–14, ACM (Sep. 2017).

[4] ProgrammableWeb, available from ⟨https://www.programmableweb.com/⟩.

[5] 岡部一詩：API経済圏，日経コンピュータ，2016/5/26号，pp.18–31 (2016).

[6] De, B.: *API Management: An Architect's Guide to Developing and Managing APIs for Your Organization*, APRESS (2017).

[7] Espinha, T., Zaidman, A. and Grosset, H.: Web API Growing Pains: Loosely Coupled Yet Strongly Tied, *J. Systems and Software*, Vol.100, pp.27–43 (Feb. 2015).

[8] Romano, D. and Pinzger, M.: Analyzing the Evolution of Web Services Using Fine-Grained Changes, *Proc. ICWS 2012*, pp.392–399, IEEE (June 2012).

[9] Wang, S., Keivanloo, I. and ZouS, Y.: How Do Developers React to Restful API Evolution?, *Proc. ICSOC 2014*, LNCS Vol.8831, Springer, pp.245–259 (Nov. 2014).

[10] 水野貴明：*Web API: The Good Parts*, O'Reilly (2014).

[11] Biehl, M.: *RESTful API Design*, API-University Press (2016).

[12] Murphy, L., Alliyu, T., Macvean, A., Kery, M.B. and Myers, B.A.: Preliminary Analysis of REST API Style Guidelines, *Proc. PLATEAU 2017/SPLASH 2017*, pp.1–9, ACM (Oct. 2017), available from ⟨https://2017.splashcon.org/track/plateau-2017⟩.

[13] Wittern, E., Ying, A., Zheng, Y., Laredo, J.A., Dolby, J., Young, C.C. and Slominski, A.A.: Opportunities in Software Engineering Research for Web API Consumption, *Proc. WAPI 2017/ICSE 2017*, pp.7–10, IEEE (May 2017).

[14] Uddin, G. and Robillard, M.P.: How API Documentation Fails, *IEEE Software*, Vol.32, No.4, pp.68–75 (July–Aug. 2015).

[15] Menascé, D.A.: QoS Issues in Web Services, *IEEE Internet*, Vol.6, No.6, pp.72–75 (Nov.–Dec. 2002).

[16] Sohan, S.M., Maurer, F., Anslow, C. and Robillard, M.P.: A Study of the Effectiveness of Usage Examples in REST API Documentation, *Proc. IEEE VL/HCC 2017*, pp.53–61 (Nov. 2017).

[17] Stylos, J., Faulring, A., Yang, Z. and Myers, B.A.: Improving API Documentation Using API Usage Information, *Proc. VL/HCC 2009*, pp.119–126, IEEE (Sep. 2009).

[18] McLellan, S.G., Roesler, A.W., Tempest, J.T., SpinuzziLellan, C.I., et al.: Building More Usable APIs, *IEEE Software*, Vol.15, No.3, pp.78–86 (May/June 1998).

[19] Myers, B.A. and Stylos, J.: Improving API Usability, *CACM*, Vol.59, No.6, pp.62–69 (June 2016).

[20] Robillard, M.P.: What Makes APIs Hard to Learn? Answers from Developers, *IEEE Software*, Vol.26, No.6, pp.29–34 (Nov./Dec. 2009).

[21] Bennet, K.H. and Rajlich, V.: Software Maintenance and Evolution: A Roadmap, The Future of Software Engineering, *ICSE 2000*, pp.75–87, ACM (May 2000).

[22] Henkel, J. and Diwan, A.: CtachUp! Capturing and Replaying Refactorings to Support API Evolution, *Proc. ICSE 2005*, pp.274–283, IEEE (May 2005).

[23] Fokaefs, M., Mikhaiel, R., Tsantalis, N., Stroulia, E. and Lau, A.: An Empirical Study on Web Service Evolution, *Proc. ICWS 2011*, pp.49–56, IEEE (July 2011).

[24] Google, API 設計ガイド，Feb. 2017, available from ⟨https://cloud.google.com/apis/design/⟩.

[25] Microsoft, API Design, Jan. 2018, available from ⟨https://docs.microsoft.com/en-us/azure/architecture/best-practices/api-design⟩.

[26] Murphy, L., Kery, M., B., Alliyu, T., Macvean, A. and Myers, B.A.: API Designers in the Field: Design Practices and Challenges for Creating Usable APIs, *Proc. VL/HCC 2018*, pp.249–258, IEEE (Oct. 2018).

[27] Li, J., Xiong, Y., Liu, X. and Zhang, L.: How Does Web Service API Evolution Affect Clients?, *Proc. ICWS 2013*, pp.300–307, IEEE (June–July 2013).

[28] Kitchenham, B. and Pfleeger, S.K.: Software Quality: The Elusive Target, *IEEE Software*, Vol.13, No.1, pp.12–21 (Jan. 2006).

[29] Ortega, M., Pérez, M. and Rojas, T.: Construction of a Systemic Quality Model for Evaluating a Software Product, *Software Quality J.*, Vol.11, No.3, pp.219–242 (July 2003).

[30] Côté, M.-A., Suryn, W. and Georgiadou, E.: In Search for a Widely Applicable and Accepted Software Quality Model for Software Quality Engineering, *Software Quality J.*, Vol.15, No.4, pp.401–416 (Dec. 2007).

[31] ISO/IEC 15939:2017, Systems and Software Engineering–Measurement Process (2017).

[32] ISO/IEC 25010:2011, Systems and software engineering–

Systems and Software Quality Requirements and Evaluation (SQuaRE)–System and Software Quality Models (2011).
[33] ISO/IEC 25020:2007, Software Engineering–Software Product Quality Requirements and Evaluation (SQuaRE) – Measurement Reference Model and Guide (2007).
[34] ISO/IEC 25030:2007, Software Engineering–Software Product Quality Requirements and Evaluation (SQuaRE)–Quality Requirements (2007).
[35] Jensen, S. and van Capelleveen, G.: Quality Review and Approval Methods for Extension in Software Ecosystems, *Software Ecosystems*, Jensen, S., Brinkkemper, S. and Cusumano, M.A. (Eds.), Edward Elgar, pp.187–211 (2013).
[36] ISO/IEC/IEEE 29148:2011, Software and Systems Engineering–Life Cycle Process - Requirements Engineering (2011).
[37] Chung, L., Nixon, B.A., Yu, E. and Mylopoulos, J.: The NFR Framework in Action, *Non-Functional Requirements in Software Engineering*, Chung, L., Nixon, B.A., Yu, E. and Mylopoulos, J. (Eds.), pp.15–45, Springer (1999).
[38] （独）情報処理推進機構：つながる世界のソフトウェア品質ガイド (2015). available from ⟨https://www.ipa.go.jp/sec/publish/20150529.html⟩.
[39] Ciraci, S. and van den Broek, P.: Evolvability as a Quality Attribute of Software Architectures, *Proc. Int'l ERCIM Workshop on Software Evolution 2006*, pp.29–31 (Apr. 2006).
[40] Raemaekers, S., Deursen, A.V. and Visser, J.: Measuring Software Library Stability through Historical Version Analysis, *Proc. ICSM 2012*, pp.378–387, IEEE (Sep. 2012).
[41] Perry, W.E.: *Effective Methods for Software Testing, 3rd Ed.*, John Wiley & Sons (2006).
[42] OpenAPI Specification (version 2.0), available from ⟨https://github.com/architecture/OAI/OpenAPI- Specification/blob/master/versions/2.0.md#schema⟩.
[43] Lewis, C. and Ou, R.: Bug Prediction at Google, available from ⟨http://google-engtools.blogspot.jp/2011/12/bug-prediction-at-google.html⟩.
[44] Swagger UI, available from ⟨https://swagger.io/swagger-ui/⟩.
[45] Mu, E. and Pereyra-Rojas, M.: *Practical Decision Making using Super Decisions v3*, Springer (2017).
[46] GitHub Changes, available from ⟨https://developer.github.com/changes/⟩.
[47] Stripe API changelog, available from ⟨https://stripe.com/docs/updates#api-changelog⟩.
[48] Rapid API, available from ⟨https://rapidapi.com/⟩.
[49] APIbank, available from ⟨https://www.apibank.jp/ApiBank/main⟩.
[50] OpenStack API Documentation, available from ⟨https://docs.openstack.org/doc-contrib-guide/api-guides.html⟩.
[51] Swagger-diff, available from ⟨https://github.com/civisanalytics/swagger-diff⟩.
[52] Basili, V.R. and Rombach, H.D.: The TAME Project: Towards Improvement-Oriented Software Environments, *IEEE Tran. Softw. Eng.*, Vol.14, No.6, pp.758–773 (June 1988).
[53] Ohashi, K., Katayama, A., Kurihara, H., Yamamoto, R., Doerr, J. and Magin, D.P.: Focusing Requirements Elicitation by Using a UX Measurement Method, *Proc. RE 2018*, pp.348–357, IEEE (Aug. 2018).

山本 里枝子 （正会員）

1983 年早稲田大学理工学部電子通信学科卒業．同年（株）富士通研究所入社．ソフトウェア開発技法，ソフトウェア開発環境，要求工学等の研究開発に従事．1999 年情報処理学会山下記念研究賞受賞．本会理事，監事，各種委員を歴任．博士（ソフトウェア工学）．IEEE 会員．日本学術会議会員．

大橋 恭子 （正会員）

1986 年津田塾大学学芸学部数学科卒業．同年（株）富士通研究所入社．ソフトウェア開発技法，ソフトウェア開発環境，要求工学，UX，ソフトウェア品質等の研究開発に従事．

福寄 雅洋 （正会員）

1998 年東京工業大学大学院総合理工学研究科知能システム科学専攻修士課程修了．修士（工学）．同年富士通（株）入社．入社後はサーバの性能チューニング，携帯電話のソフトウェア開発に従事．2015 年より（株）富士通研究所にて，ソフトウェア保守・プログラム解析の研究開発に従事．

木村 功作 （正会員）

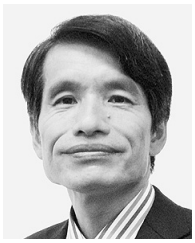
2005 年九州大学工学部電気情報工学科卒業．2010 年同大学大学院システム情報科学府博士後期課程修了．博士（工学）．同年（株）富士通研究所入社．ソフトウェア開発効率化・自動化に関する研究開発に従事．本会 SES2012 優秀論文賞受賞．電子情報通信学会，IEEE 各会員．

関口 敦二

1998 年早稲田大学大学院理工学研究科物理学及応用物理学専攻博士前期課程修了．修士（理学）．同年（株）富士通研究所入社．クラウドコンピューティング，Web API，マイクロサービス等のソフトウェアアーキテクチャ技術の研究開発に従事．電子情報通信学会会員．

上原 忠弘 （正会員）

1995 年東京工業大学大学院総合理工学研究科知能システム科学専攻修士課程修了．修士（工学）．同年（株）富士通研究所入社．ソフトウェア開発技法，ソフトウェアテスト，ソフトウェアアーキテクチャ技術の研究開発に従事．



青山 幹雄 （正会員）

1980 年岡山大学大学院工学研究科修士課程修了．同年富士通株式会社入社．大規模ソフトウェア開発とプロジェクト管理，ソフトウェア工学の実践に従事．1986～1988 年米国イリノイ大学客員研究員．1995 年 4 月～2001 年 3 月新潟工科大学情報電子工学科教授．2001 年南山大学数理情報学部情報通信学科教授．2009 年より情報理工学部ソフトウェア工学科教授．博士（工学）．クラウドコンピューティング，自動車組込みソフトウェア，機械学習ソフトウェア等を対象として，要求工学，ソフトウェアアーキテクチャ技術の研究と教育，人材育成に取り組む．著書「要求工学知識体系」(2011 年刊：共著）ほか多数．IEEE Software，IEEE Transactions on Services Computing 等の編集委員，本会理事を歴任．1993 年情報処理学会研究賞受賞．ソフトウェア科学会，自動車技術会，IEEE，ACM，SAE 各会員．