環境及び要件の変化に対応するWebアプリケーションフレームワークの研究

A Study on Web Application Frameworks

to Meet Changing Environments and Requirements

松塚　貴英

Taka Matsutsuka

概要

近年、グローバル化をはじめとしてますます激しくなるビジネス環境の変化により、企業を取り巻く不確実性が増している。そのような環境において、変化に対応していくことは企業にとっての重要課題である。特に、Web アプリケーションは顧客や従業員などの利用者によって直接利用されるため、特にこのような変化の影響を受けるソフトウェアである。具体的には、近年のスマートフォンをはじめとしたデバイスの普及により、クライアントの種類も動作環境もきわめて多様になっており、動作中の環境変化も激しくなっている。これに対し、従来の Web アプリケーションではサーバーが事前に設定した対応しかできないため、様々なクライアントの環境変化に対応しきれないという問題がある。これらの変化に対応するには、ソフトウェアコードに手を入れることなく、実行中に機動的に対応できる仕組みが必要である。一方で、実行中の対応には限界があり、大きな変化に対してはソフトウェアを修正して対応することになるため、ソフトウェアコードが高い保守性を持つ必要がある。従来 Web アプリケーションはサーバとクライアントでフレームワークが異なっており、開発者は両方のフレームワークを習得しなければならないだけでなく、クライアントとサーバの連携をフレームワークの組み合わせごとに定義しなければならない。するとクライアントの変化に対応するための連携機能も個別実装になり保守性が大きく悪化してしまうという問題がある。

これらの問題に対し、本研究においては、従来サーバーおよびクライアントにて別々に実装されているフレームワークの構造をモデル化して統合し、同じ概念で双方のフレームワークを実装できるようにする。これによりクライアントとサーバの連携方法をはじめとして一貫性が確保され、保守性と柔軟性を高めることができる。また、クライアントからの環境情報に合わせ、実行中にサーバーが対応を変化させられる仕組みを導入する。このとき、環境変化が当初予期していた範囲を超えていた場合でも、サーバーがその変化に対してより抽象度の高い解釈をして対応することができることが特徴である。また、大きな対応が必要な場合はソフトウェアコードの修正を行う必要があるため、実行中に対応を変化できる仕組みをフレームワークに組み入れることで保守性を高める。

本論文は、次の 6 章で構成される。

第 1 章では、背景を述べ、Web アプリケーションが抱える問題と、それに対する課題を議論したうえで、本研究の目的を述べる。ソフトウェアが環境や要件の変化に対応するには、ソフトウェアを一度止め、ソースコードを変更する静的対応と、実行中に動作の設定や構造を変えることで動作を変更する動的対応の 2 種類に大別される。Web アプリケーションにおいては、静的対応としてフレームワーク技術が利用されるが、クライアントとサーバーという 2 種類の問題領域が存在するため、それぞれ個別にフレームワークが開発されている。すると開発時には開発者がそれぞれに対して熟達しなければならず、学習コストや保守コストが高くなることが問題である。そのため、これらのフレームワークを統合することを 1 つ目の課題とする。また、動的対応においては自己適応技術が知られているが、開発者がフレームワークとは全く別の技術として取り扱わなければいけないという問題がある。このことから、Web のフレームワークに動的対応を追加することを 2 つ目の課題とする。本研究の目的は、これら 2 つの課題を解決するために、フレームワークモデルの提案をすることである。

第 2 章では、関連研究について検討し、本研究の位置づけを述べる。まず、静的対応においては、要求レベル、仕様レベル、コードレベルで様々な研究がある中で、Web アプリケーションにおいては静的対応と動的対応を統合するという観点から、主にコードレベルに着目することについて述べる。次に、動的対応の技術として自己適応技術について検討し、自己適応技術は管理対象サブシステム、管理サブシステムから構成されること、変更のメカニズムがパラメーター変更と構造変更に大別されることを述べる。本研究では、静的対応と動的対応の両方に対応するために、Web アプリケーションフレームワークへの自己適応技術の統合を行うこととした。

第 3 章では、フレームワークモデルの提案を行う。Web アプリケーションはフレームワークとの親和性が高いため、静的対応・動的対応を一つのフレームワークとして統合できることが望ましいが、現実にはサーバーとクライアントでは利用されるプログラミング言語や動くプラットフォームが異なり、完全に同一な実装のフレームワークをサーバーおよびクライアントの双方で動作させることはできない。このことから、クライアントおよびサーバーにおけるフレームワークを一段抽象的な概念でモデル化し、同じモデルからサーバーとクライアント双方のフレームワークを実装できるようにするアプローチを取る。

まずサーバーフレームワークとクライアントフレームワークを統合するために、双方のメカニズムの共通点から処理の流れを 7 段階のステップに整理し、それぞれのステップについて詳述する。各ステップはデータ入力、データ抽象化、データチェック、ロジック選択、ロジック実行、画面作成、画面表示から構成される。ロジック選択ステップでは、呼出判別表という定義体を利用してクライアントのリクエストをロジック呼び出しにマッピングしている。処理を行う際の入出力データとその処理の相互依存関係を解消し、画面表現と入出力オブジェクトの組み合わせを部品としてアプリケーション内で再利用ができるようにする。

次に、自己適応技術の統合を行う。自己適応技術は実行中の環境の変化に対応するために、管理サブシステムと管理対象サブシステムから構成される。Web アプリケーションにおいてはクライアントからの要求に対してどのようなロジックを実行するかが、もっとも変化が多く本質的な部分となるため、ロジックの選択を動的に変えることができれば、様々な環境の変化に対応することができるようになる。これは、管理対象サブシステムとして呼出判別表を用いればよい。呼出判別表に条件項目を追加することで、入力情報に対してきめ細かく対応できるようにする。

また、入出力オブジェクトにクラス階層を導入する。これは、変化する状況下においてあらかじめ想定した処理では対応できない場合に、段階的に縮退することを可能にするためである。入出力オブジェクトは、スーパークラスにより一般的な内容が設定され、サブクラスにより個別の内容が設定される。クライアントからの情報が個別の情報であればあるほど、特定の状況に合わせたロジックを呼び出すように設定することができる。しかし、クライアントからの情報がサーバー側で想定していた値の範囲に収まらなかったり、値が欠損していたりした場合は、その特定の状況に合わせたロジックを選択することができない。代わりに、そのような場合はスーパークラスに対応したロジックを呼び出す、すなわちサーバーがより抽象度の高い対応をすることで、クライアントの状況に対して次善の策を取ることができる。このスーパークラスへの縮退は再帰的に行われるため、最終的にどのクラスでの対応も不可能であった時も、最終的に全てにマッチするエントリーを呼出判別表に用意することで例外対応をすることができる。

本モデルに対する考察として、以下について議論する。保守性に対しては、コード量、関心の分離、再利用性という観点に関して、本モデルは効果があることを示す。クラス階層の利用の一例として、アプリ

ケーションが更新された時に、多様なクライアントが更新に即座に追従できないことがあるが、このような場合も入出力をスーパークラスに縮退させることでサーバーが対応することができる。本フレームワークモデルで導入している呼出判別表は、プログラミング言語のメソッド呼び出しに相当するが、条件指定とオーバーロードという二点において拡張されていることを示す。また、動的対応のレベルを自己適応技術に照らし合わせて議論する。本モデルにおいては、Foreseen, Foreseeable という 2 種類の変化について対応することができる。さらに、本モデルの Web アプリケーション以外への適用可能性を検討する。フレームワークのコア部分はリアクティブな動作メカニズムであることから、Web アプリケーション以外のユーザインタフェースを実装するフレームワークにも適応可能であるほか、マルチエージェントシステムへの応用に可能性がある。

第 4 章では、モデルによるフレームワークの実装例として、サーバー型フレームワーク、クライアント型フレームワーク、適応型フレームワークの 3 種について実装とその評価を行う。

サーバー型フレームワークにおいては、Servlet/JSP 環境を利用した実装フレームワークについて述べる。本フレームワークモデルに従って実装することで、画面の部品化、再利用ができるようになった。また、ロジックの実装量が大幅に削減されるとともに関心の分離を達成し、開発効率及び保守性を上げることができた。

クライアント型フレームワークにおいては、HTML/JavaScript 環境における実装フレームワークを示す。クライアント型においては画面動作をきめ細かくコントロールする必要があるため、フレームワークモデルの画面作成ステップにおいて、画面部品をウィジェットと機能付加部品に分割し、様々な機能を持つ画面部品を容易に拡張できる枠組みを作る。これにより、画面部品を組み合わせで増やすことができるようになったほか、計測においてロジックの実装量が大幅に削減される効果を確認した。

適応型フレームワークにおいては、試作したフレームワークにおいて、人材交流イベントを支援するサービスを事例として効果を確認する。呼出判別表を動的に更新する仕組みを導入することで、人材のマッチングを行うロジックをクライアントで行うのかサーバーで行うのかを動的に切り替えることができるようになり、参加者人数に応じて最適な応答時間で実行するロジックを選択することが可能となった。また、呼出判別表のクラス階層の仕組みを導入することで、クライアントからの要求人数に対するマッチングロジックの予想所要時間があまりに長い場合に、簡易的なマッチングロジックへ縮退する機構を実現した。

第 5 章では、フレームワーク移行について述べる。いったんあるフレームワークを採用してアプリケーションを開発すると、他のフレームワークに移行するのは困難である。この点は、他のフレームワークから本モデルをベースとしたフレームワークに移行する際も同様である。この問題を解決するため、モデル駆動型開発を用いたアプリケーションのフレームワーク間の移行技術を開発・実証する。モデル駆動開発は、Web アプリケーションの仕様をプラットフォームに独立な形で形式的に記述することで、様々な実装フレームワークに展開することができるようになるという考え方である。ここでは、まず Web アプリケーションの仕様記述方法である WebWare 意味モデルを開発する。次に Struts フレームワークに準拠した検証用アプリケーションを用意し、研究用に作成したリバースエンジニアリングツールで検証用アプリケーションに対応する WebWare 意味モデルを導出する。さらに、WebWare 意味モデルから本章のサーバー型フレームワークに準拠したアプリケーションを生成することで、フレームワーク間の移行ができることを検証する。

第 6 章では、本研究のまとめを行い、今後の課題について述べる。

目次

第 6 章　結論

第1章　序論

本章では、背景として Web アプリケーションの概要について述べたのち、本研究の目的について述べる。次に、現在の課題について議論し、本研究でのアプローチについて示す。

1.1 背景

近年、グローバル化をはじめとしてますますますます激しくなるビジネス環境の変化により、企業を取り巻く不確実性が増している。そのような環境において、変化に対応していくことは企業にとっての重要課題である(McGrath, 2013)。企業で使われるソフトウェアのうち、特に Web アプリケーションは、近年スマートフォンをはじめとした技術の頻繁な変化により、クライアントの種類も動作環境も多様になっている。また、顧客や従業員などの利用者によって直接利用されるため、特にこのような変化の影響を受けるソフトウェアである(Figure 1.1)。この変化に対応するために様々な対応策が取られる。対応策を大別すると、変化に対して静的に対応する方法(静的対応)と動的に対応する方法(動的対応)がある。静的対応と動的対応は、Table 1.1 にまとめるように、その契機や対応方法、留意点等に違いがある。

静的対応とは、要件の変化に対応するもので、顧客要求の変化や技術の変化などを契機とし、ソフトウェアのソースコードを変更することにより動作を変更することである。ここで要求とは顧客が作って欲しいものの条件(利用者視点)で、要件とはシステムが実現すべき条件(システム視点)である。静的対応のためには一般的にはソフトウェアを一度止め、ソースコードを変更する必要がある。その結果、コンパイル、テストといった開発工程ののちに動くソフトウェアとして配備される。開発工程を経るため、広範な変化に対応が可能な反面、対応に一定の時間がかかる特徴がある。また、開発効率や保守性など開発に関して留意する必要がある。

一方、動的対応とは、環境の変化に対応するもので、デバイスの実行状態変化を契機とし、ソフトウェアを停止することもソースコードを変更することもなく、実行中にソフトウェアの設定や構造を変えることで動作を変更することである。動的対応は、ソフトウェアの周辺環境の変化に対し即応性の高い対応が可能となることと、近年 IoT をはじめとした技術により環境情報の取得・活用が可能となったこととを合わせ、着目されている。動的対応においては、デバイスの実行状態の変化やソフトウェア実行環境の変化などを契機に、ソフトウェアの設定や構造、すなわちパラメーターやモジュール間のつながり方などを変更することで対応が行われる。動作中に人手を介さずに変更を行うため、即応的に対応ができることが特徴である一方で、対応範囲が限られるという特徴がある。その対応範囲や実際に対応にかかる時間などが留意点となる。

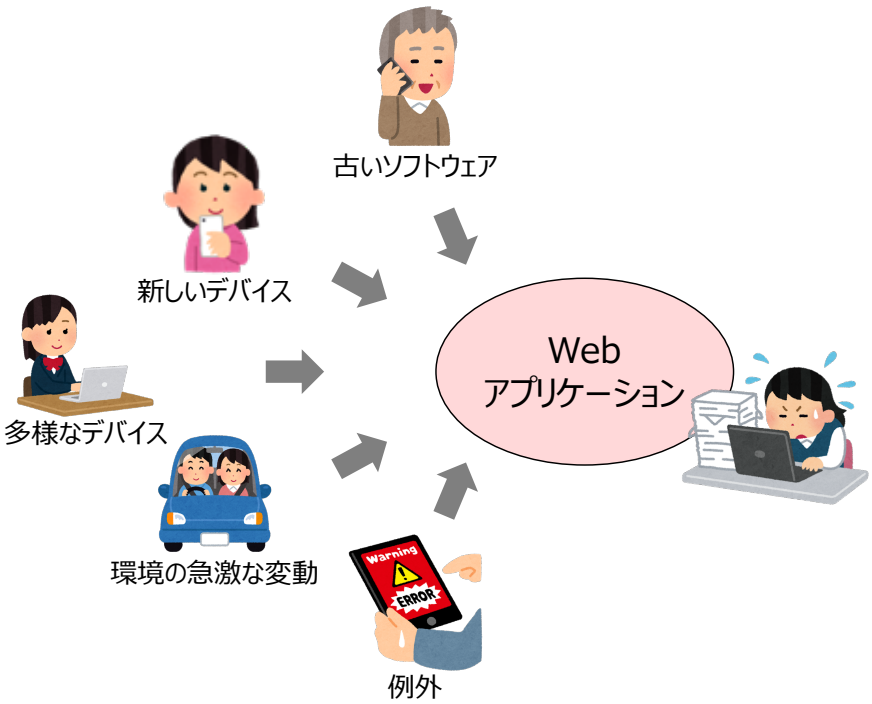

Figure 1.1 様々な変化を受ける Web アプリケーション

Table 1.1 変化の種類と対応

対応の種類	静的に対応(静的対応)	動的に対応(動的対応)
主に対応する変化	要件の変化	環境の変化
トリガーの例	顧客要求の変化 技術の変化	デバイスの実行状態変化や変更 実行環境の異常
対応方法	ソフトウェアを止め、 ソフトウェア(コード)を変更する	実行中に設定や構造を変える ことで動作を変更する
留意点	開発効率、保守性など	対応範囲、対応速度など
技術	フレームワーク	自己適応

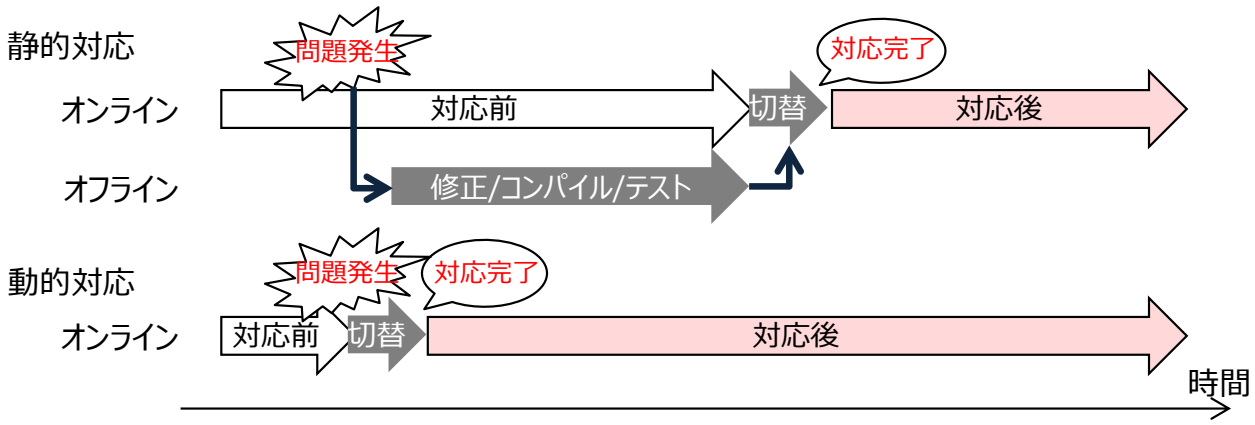

Figure 1.2 静的対応と動的対応

1.2 問題と課題

1.2.1 問題

前節で議論したとおり、企業はビジネス環境の変化に素早く対応したい。Web アプリケーションが様々な変化を受け、これに対応しなければならない理由は、サーバーとは独立に存在する多様なクライアントの変化を事前に予測できないことに起因する。これに対し、従来の Web アプリケーションではサーバーが事前に設定した対応しかできないため、これらクライアントの変化に対応しにくいという問題がある。この問題に対するアプローチが静的対応と動的対応の 2 種類であることを述べた。この 2 種類の違いの本質は、変化が発生してからそれに対応するまでの間に人間の思考の時間が含まれているかどうかにある。この概要を Figure 1.2 に示す。

動的対応を使うと実行中に自動で環境変化に対応できるため、素早く対応する、という要求に応えることができる。しかし、動作中に自動的にソフトウェアに対して変更を行うため、対応中に人間の思考を入れることはできない。そのため、動的対応は範囲に限界があり、予期しない事象への対応や、大きな変更を伴う対応はできない。静的対応はソースコードを変更するため、予期しない事象や大きな変更に対しても対応が可能である。これは、ソースコードを変更するという対応内容に人間の思考の時間が含まれているためであるが、問題が発生してから対応が完了するまでに必要な時間は動的対応よりも長くなる。また、ソースコードを変更する際にはコンパイルやテストといった付随する作業も必要であり、この点も対応時間を長くする一因となる。以上より、静的対応と動的対応は広範な事象に対応するか、素早く事象に対応するか、という点で相互補完の関係にある。

Web アプリケーションの静的対応においては、フレームワーク技術が利用されるのが普通である。フレームワークは実装技術であるため、一般には特定の問題領域に対して実装される。Web アプリケーションにはクライアントとサーバーという 2 種類の問題領域が存在するため、それぞれフレームワークは別々の種類となる。実際の Web アプリケーションの構築形式は主にサーバーで処理するもの、主にクライアントで処理するものなど様々である。サーバーとクライアントでフレームワークが別々であると、開発者がそれぞれに対して熟達し、開発後も両方の対応をしなければならないため、学習コストや保守コストが高くなることが問題である。一例として、サーバーおよびクライアント間での処理の移行を挙げる。近年、スマートフォンの高性能化などの技術の変化により、クライアントにおいて高度な処理が可能に

なっている。これは、従来サーバーで処理していた内容をクライアントでも処理できるようになることを意味する。しかし、サーバーとクライアントのフレームワークが異なると、サーバーからクライアントへロジックを容易に移行することができず、別のフレームワーク上における作り直しが発生するという問題が発生する。

動的対応においては、自己適応技術が知られている。これは運用中の変化に合わせてソフトウェア自身が自己調整することを目的とした技術(Salehie & Tahvildari, 2009)で、フレームワークとは独立して進化している。そのため、この点においても開発者は完全に異なる 2 つの技術を学ぶことになり、学習コストが高くなってしまう。また、自己適応技術はフレームワークなどの開発技術に統合されていないために、クライアントの状況変化をどう捉えて対応するかという点について考慮されていないこと、および開発者が Web アプリケーションフレームワークとは全く別の技術として取り扱わなければいけないという問題がある。

1.2.2 課題

これらの問題に対し、次のように解決すべき課題を設定する。

課題 1

Web アプリケーションのフレームワークがサーバーやクライアントで異なる動作や実装になっている問題については、サーバーおよびクライアントのフレームワークを統合することを課題とする。

課題 2

Web アプリケーションがクライアントの変化に動的に対応できない問題については、Web のフレームワークに動的対応の機能を追加することを課題とする。

1.3 本研究の目的

前節で議論した問題および課題から、本研究では、静的対応および動的対応双方を扱うことができる Web アプリケーションを開発するためのフレームワークモデルを提案する。そのアイデアおよび対象範囲について Figure 1.3 に示す。

静的対応において、フレームワークは特定の問題領域ごとに実装されるため、サーバーとクライアントのフレームワークの実装は異なるのが実態である。また、実際にサーバーとクライアントでは使われる技術やプログラミング言語が異なるため、完全に同じ実装のフレームワークをサーバーとクライアントの両方で展開することはできない。そのため、クライアント、サーバーにおけるフレームワークを一段抽象的な概念でモデル化し、同じ概念で実装できるようにすることにより、サーバーとクライアントのフレームワークを統合する。これをフレームワークモデルと呼び、サーバーおよびクライアントにおいては、このフレームワークモデルに基づいたフレームワークを実装・展開する。これにより、Web アプリケーションのサーバーおよびクライアントの実装において、それぞれのフレームワークの利用方法を学習するためのコストが低減されることが期待できる。また、サーバーフレームワークを使って作成したアプリケーションを変換することによりクライアントフレームワークで使う、といったことも可能となる。

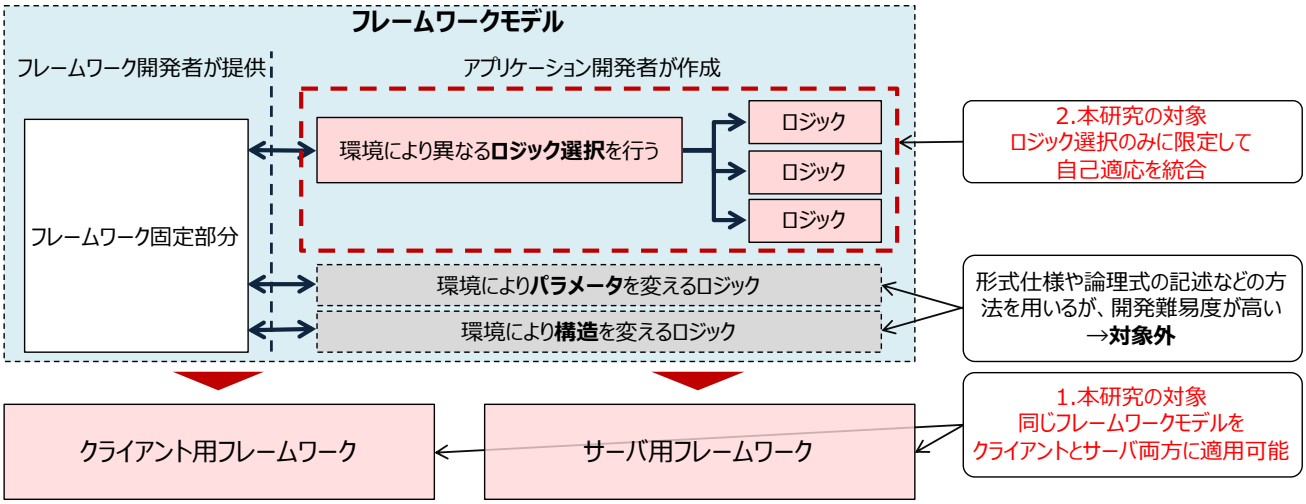

Figure 1.3 フレームワークモデルおよび研究対象範囲

動的対応については、自己適応技術を採用する。自己適応技術をフレームワークに追加統合することで、一定の変化に対しては自己適応技術を使って動的に対応を行いつつ、大きな変化が必要なときはフレームワークを活用することにより静的な変更を効率的に行えるようにする。自己適応技術の機構としては一般にパラメーターを変える、構造を変えるといったものがある(Cheng et al., 2016)。これらを検討し、Web フレームワークに必要な適応の仕組みに絞ることで仕組みを簡素化し、アプリケーション開発者に理解しやすい構造とする。具体的には、Web アプリケーションはクライアントでの利用者要求をサービス(ロジック)につなぐのが本質であるため、そのロジック選択を対象にすることで、適応範囲を限定できると考えた。

以上を考慮したフレームワークモデルの提案により静的対応と動的対応を融合し、Web アプリケーションが要件の変化と環境の変化の双方に対応できるようにする。これにより、様々なクライアントの変化に対応できる Web アプリケーションをフレームワークに則って容易に実現できるという効果が期待できる。また、事例においてモデルの実装を示すことで、モデルが実際のアプリケーションに適応可能であることを示す。

1.4 本論文の構成

本論文の議論は次のように進める。第 2 章において、技術背景と関連研究について検討し、本研究の位置付けについて明確にする。次に、第 3 章でモデルの提案を行う。第 4 章では対応範囲において、提案モデルを使い Web アプリケーションフレームワークが実装できることを確認する。第 5 章では、他フレームワークから本フレームワークへの移行について述べる。最後に第 6 章で結論を示す。

第2章　関連研究

本章では、本研究の技術的な背景及び関連する研究について検討するとともに、本研究の位置付けを述べる。

2.1 静的対応の実現に必要な性質および関連研究

2.1.1 静的対応の種類

静的対応は様々な領域があるが、ここでは一般的な開発ライフサイクルの考え方(Forsberg & Mooz, 1991)に従い、要求、仕様、コードの各レベルにおいて、変化に対応するための技術について議論する。

要求レベルにおいては、利用者の要求はそのままではソフトウェアとして動作させることができないため、要求を分析して仕様を作る必要がある。要求分析については要求工学という分野において研究が行われている。要求工学は、ソフトウェアシステムに関する実世界のゴール、機能、制約を扱い(Zave & Jackson, 1997)、利用者の要求を分析要素として具体化し、ソフトウェアの設計、実装、テストの仕様を作成するプロセスである(Ramamoothy & Tsai, 1996)。要求工学の中で変化に対応するために、要求変更管理の研究が行われている(Jayatilleke & Lai, 2018)。

仕様レベルにおいては、開発対象ソフトウェアの仕様を記述し、詳細化を行う。仕様は、ソフトウェアコードを開発するために、そのソフトウェアがどう動くべきかを記述する文書である。仕様は UML (Rumbaugh, Jacobson, & Booch, 2004)などの人間にとっての理解を向上させるための言語で記述されることもあれば、より正確な仕様を記述するために Z (ISO, 2002)や VDM++ (Durr & van Katwijk, 1992)などの形式的(formal)な仕様記述言語が使われる場合もある。通常、論理学的あるいは代数学的に形式記述された仕様であってもソフトウェアコードと１対１に対応するものではなく、仕様記述されたものが自動的に動作するソフトウェアになるわけではない。これは、仕様記述言語が仕様の抜けや漏れがない、デッドロックがないといった、システムの性質を確認・保証するために使われるためである。仕様記述言語を別途記述するアプローチの他に、プログラムを入力としてシステムの性質を確認する方法もある。この領域の研究としては、Java PathFinder (Havelund & Pressburger, 2000)やシンボリック実行(片山ほか, 2013)が知られる。また仕様記述には Model Driven Development (MDD)という手法がある(Beydeda & Book, 2005)。これは、ソフトウェアの本質、すなわち実装されるべき機能をソフトウェアが配備されるクラウドなどのプラットフォームから分離し、UML などの仕様記述言語を使ってプラットフォームやフレームワークに非依存な形式で記述する考え方である。ここで、プラットフォーム非依存な仕様モデルのことを Platform Independent Model (PIM)といい、プラットフォーム固有の仕様モデルを Platform Specific Model (PSM)という。MDD の長所は、開発において PIM から PSM、またはソフトウェア実装を自動的ないし半自動的に生成できる点にある(Marın, Pereira, Giachetti, Hermosilla, & Serral, 2013)。また、静的に変化に対応するため、仕様の変化をモデルの修正で対応するアプローチが研究されている(Brosch et al., 2012)。プラットフォーム非依存な形で仕様を表現できる MDD の特徴は、プラットフォームやフレームワークが変化する

可能性がある場合は利点が大きい。この点は第 5 章において、フレームワークを移行する事例において議論する。

要求と仕様には次の関係がある。開発時に想定した実行環境をD、要求Rを充足するソフトウェア仕様 S は以下を満たす(Zave & Jackson, 1997)。

$$S, D \vDash R$$

本研究では、Web アプリケーションにおけるクライアントの環境変化、つまり上に示した式のDの変化を扱う。ここで環境が変わっても利用者の要求は変わらないため、仕様Sを変化させる、すなわち動作を変化させることにより要求Rを満足させるようにしたい。このことから、要求レベルの変化について対象から外し、仕様および仕様を実装したコードを対象として検討する。

コードレベルにおいては、実際に仕様に従ったソフトウェアを作成する。本研究では、静的対応と動的対応を統合するという観点から、主にコードレベルに着目する。これは、仕様を具現化しているのはコードであること、動的対応を行うためにはコードの変更が必要なことから、静的対応に動的対応を統合するためにはコードレベルの技術が必要なためである。コードレベルではソフトウェアを新規に作成・追加したり、既存のソフトウェアを修正したりする必要がある。そのため、開発効率や保守性を高くすることが求められる。特に、動的対応に関しては一度開発した後のコード変更が必要なため、静的対応における保守性が重要となる。この点は 2.1.3 節で議論する。コードレベルにおける開発効率・保守性に関する研究には、プロダクトラインおよびフレームワークというアプローチがある。いずれも共通で使われるコードを整備し、必要となる部分のみ実装することでアプリケーションを完成させることができる手法である。

プロダクトラインは、あるアプリケーション群の開発のために共通部分となる再利用可能な部品群(コア資産)を整備し、アプリケーションごとに必要な箇所のみを個別開発資産として開発し、組み合わせて完成品のアプリケーションにする手法である。複数のアプリケーションをシリーズとして開発する必要がある場合はプロダクトラインの考え方を活用できる。プロダクトラインにおいては、ユビキタス・コンピューティング、サービス・ロボティクスなどの要求やリソースの制約が大きく変化する領域においては、動作時に設定やモジュール間の関係が変化するダイナミックプロダクトラインのアプローチが研究されている(Hinchey, Park, & Schmid, 2012)。たとえば、サービスロボットにおいて、動作時に環境に適応させる仕組みの研究がある(Murwantara, 2020)。

フレームワークは、Web アプリケーションなどの特定のアプリケーション領域において、ソフトウェアの中で共通化できる部分と、要件により変わる部分を分離し、両者を組み合わせてソフトウェアが完成するようにしたプログラムの雛形である。共通化部分を coldspot、要件により変わる部分を hotspot という(Pree, 1994)。それぞれ coldspot はライブラリの形式であらかじめ提供され、hotspot を要件に従って開発することでアプリケーションを完成させることができる。そのとき、アプリケーションの開発者として開発する必要があるのは hotspot のみとなり、全体の開発量を削減することができる。フレームワークは通常アプリケーション個別に開発するコードが受動的であり、フレームワーク側でソフトウェア自体の動作の流れを規定する。そのため、個別に開発されるコードはフレームワークから呼び出されることになる。本研究で対象とする Web アプリケーションにおいては、クライアントにおけるキー入力、ボタン押下などのイベント発生を契機として動作が開始する。イベントの処理は Web ブラウザや HTTP サーバーが担い、そこから具体的な処理を行うために個別のコードの実行を行うため、フレームワークの構造との親和性が高い。

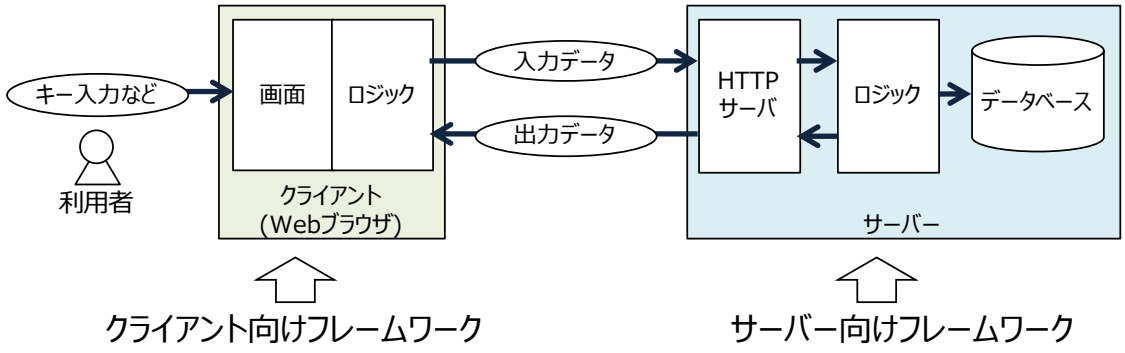

Figure 2.1 Web アプリケーション

2.1.2 Web アプリケーションフレームワーク

フレームワークはアプリケーション領域ごとに実装が必要となる。これは、フレームワークがコードレベルの技術であり、アプリケーション領域ごとに必要な機能が大きく異なるためである。たとえば、ロボティクス向けのフレームワーク(Meier, 2015)においては、飛行機など時間制約を持つロボットのための制御の仕組みを持ち、マイクロコントローラに展開されることから、クロスプラットフォームでの開発を支援する仕組みを備えている。また、医療センサーのデータ処理を対象としたフレームワーク(Rajput, 2016)では、体温計やパルスセンサーなどの医療センサーを対象に、デバイスの検知からアプリケーションまでをカバーするために、デバイス層、ゲートウェイ、サービス、ネットワークの 4 つの層からなるフレームワークのアーキテクチャを提案している。一般にフレームワークが異なると、仮にアプリケーション領域が同じでも、coldspot、hotspot とも異なる実装となり、互換性はない。これは、動作環境や解決する問題の対象、プログラミング言語の違いなどでフレームワークの実装が変わってくるためである。

本論文で対象とする Web アプリケーションは、一般に Figure 2.1 のようにクライアント(Web ブラウザ)およびサーバーより構成される。そのため、Web アプリケーションは、大きくクライアント向けのフレームワークとサーバー向けのフレームワークに分かれる。クライアントのフレームワークとしては、React.js (Facebook Inc, 2021)や Angular (Angular Team, 2021)がある。サーバー向けのフレームワークとしては、利用できる言語の種類が多いため、言語ごとに様々なフレームワークが提案・利用されている。一例として、Python 言語で書かれた Django (Django Software Foundation, 2021)、Ruby 言語で書かれた Ruby on Rails (Hansson, 2021)、JavaScript 言語で書かれた Node.js, (OpenJS Foundation, 2021)、Scala 言語で書かれた Play (Lightbend, 2021)などがある。これらの違いは、クライアント向けのフレームワークが動的に変化するリッチな画面表現やインタラクションを提供することを主眼としている(Serrano & Aroztegi, 2007)のに対し、サーバー向けのフレームワークはビジネスロジックや画面遷移などを主な機能としている点である。これらのフレームワークは言語や実装がそれぞれ異なるため、互いに互換性はない。そのため、開発者が Web アプリケーションを作る場合は、少なくともサーバー用のフレームワークとクライアント用のフレームワークをそれぞれ学習する必要がある。

2.1.3 保守性

1.1 節で議論したように、静的対応とは、顧客要求の変化や技術の変化などを契機とし、ソフトウェアのソースコードを変更することにより動作を変更することである。この作業は反復して行われるため、

ソフトウェア自体の保守性を高く保つ必要がある。ここでは、保守性に関する指標として、コード量、関心の分離、再利用性について検討する。

コード量は初期開発時、修正時双方に影響する保守性の重要な数値指標である。ソースコードから取得できる最も単純な数値であることから、多くの保守性診断の研究で利用されている(Riaz, Mendes, & Tempero, 2009; Ganpati, Kalia, & Singh, 2012)。一般にはコード量、特にアプリケーション開発者が開発する部分については少ない方が望ましい。

関心の分離(separation of concerns)とは、ソフトウェアにはアルゴリズム、データ処理、同期、実行場所制御、エラー処理などの実現すべき様々な関心があり、各関心を分離して実装することにより疎結合とし、保守性を向上させる考え方である(Hürsch & Lopes, 1995)。特に、Web アプリケーションにおいては、プレゼンテーションとドメインの分離、すなわちユーザインタフェースのコードを他のコードから分離することが重要である(Fowler, 2001)。また、業務アプリケーションは規模が大きくなりがちであり、多数の開発者が関わった開発が必要であるが、その際に関心の分離があることによって、開発者の役割分担を明確にすることができる。

再利用性(reusability)は、ソフトウェア品質基準 SQuaRE による定義では「一つ以上のシステムに，又は他の資産作りに，資産を使用することができる度合い」である(ISO, 2011; JIS, 2013)。より具体的には、アプリケーション内での再利用とアプリケーションを越えた再利用がある。フレームワークにおいては、もともと coldspot 部分が複数のアプリケーション開発のために使えるように作られているため、アプリケーション間での一定の再利用性は確保されていると言える。加えて、hotspot 部分についても、アプリケーション内で複数箇所に利用できると望ましい。これは、Web アプリケーションにおいては、例えば画面の部品を hotspot として開発したときに、この部品をアプリケーションの複数の画面から使えるようにする、といったことが相当する。

2.2 動的対応の実現に必要な性質および関連研究

2.2.1 動的対応

動的対応、すなわちソフトウェアが動作中に環境の変化に合わせることは、次のように表現できる。2.1.1 節で示した Zave & Jackson の式を再掲する。

$$S,\ D \vDash R$$

ここで、実行環境が変化し、$D \rightarrow D_1$となったときには、更新された実行環境の元で R を充足する新たな S_1 を求める必要がある。

$$S_1,\ D_1 \vDash R$$

このような修正をオフラインで人手により行う場合は、適応保守(adaptive maintenance)と呼ばれ、ソフトウェア自身がオンラインで行う場合は、自己適応(self-adaptation)と呼ばれる(鄭ほか, 2014)。自己適応技術においては、ソフトウェア自身が動作中に発生する要求や環境の変化に対応していく必要があるが、そのためにはS, D, Rの各要素はソフトウェアで解釈できなければいけない。

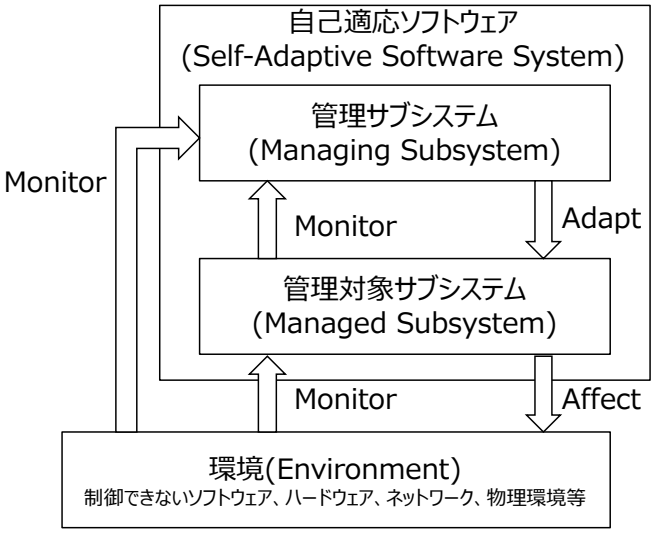

Figure 2.2 自己適応モデル(Weyns et al., 2013)

2.2.2 自己適応技術

構成

自己適応を行うソフトウェアの構成は、アプリケーションロジックそのものに自己適応の仕組みを内包する内部アプローチ(internal approach)と、アプリケーションロジックの外側に自己適応の仕組みを設ける外部アプローチ(external approach)に大別することができる。内部アプローチは、適応プロセスのためのロジックがアプリケーションロジックと混在するため、保守性が低下することが多い(Salehie, 2007)。そのため、本研究では外部アプローチを採用する。

自己適応ソフトウェアは管理対象サブシステム(managed subsystem)、管理サブシステム(managing subsystem)から構成される(Weyns et al., 2013)。また、システム自体が環境(environment)と相互作用している。この様子を Figure 2.2 に示す。

環境は、自己適応ソフトウェアが相互作用する実行の前提となるもので、$S, D \vDash R$の式におけるDを表す。Web アプリケーションにおいては、Web サーバーソフトウェアなどのミドルウェアやそれらが置かれている物理環境など、様々な要素が含まれる。ある対象物が環境と自己適応ソフトウェアのどちらに含まれるかは、その対象物が適応の対象になっているかどうかで区別する。本研究の場合、Web アプリケーションにおいて、表示される画面や利用者が入力するデータなどは自己適応ソフトウェアに含まれる。一方、Web アプリケーションはクライアントデバイスとインタラクションするが、デバイスを管理するわけではないので、クライアントデバイスが搭載するセンサ(GPS やカメラなど)は環境の一部とみなされる。

管理対象サブシステムは、アプリケーションロジックで構成されている。Web アプリケーションにおいては、利用者にサービスを提供するためのロジックが該当する。言い換えると、$S, D \vDash R$の式におけるSを表すのが管理対象サブシステムである。管理対象サブシステムは、環境とインタラクションしており、環境状態を把握(monitor)して、仕様に沿って環境に影響を与える(affect)。

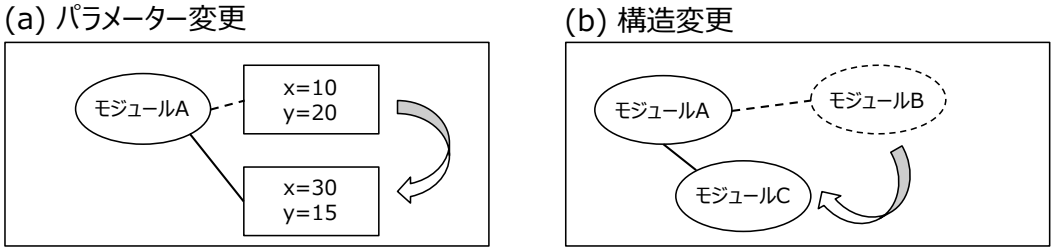

Figure 2.3 パラメーター変更と構造変更

管理サブシステムは、管理対象サブシステムを適応するためのメカニズムを持つ。具体的には、Web アプリケーションにおいては、環境の変化に応じてサービス提供のロジックを修正することが相当する。そのために、環境および管理対象サブシステムの状態を把握(monitor)して、状況にあった状態にするために管理対象サブシステムの仕様をどう変更すればよいかを導出し、その通りに管理対象サブシステムを適応(adapt)させる。

変更のメカニズム

自己適応の仕様変更のメカニズムに着目すると、パラメーター変更および構造変更に大別される(Cheng et al., 2009; Grua, Malavolta, & Lago, 2019)。この様子を Figure 2.3 に示す。パラメーター変更は、適応を行うソフトウェアモジュールのパラメーターを変更することにより動作を変更する適応メカニズムである。Figure 2.3(a)においては、あるモジュール A のパラメーターが(x=10, y=20)であったものを、環境の変化に応じて修正し、(x=30, y=15)としている。たとえば、GUI のボタンや画像などの様々な要素を、高齢者を想定した利用者が使いやすいように調整するために適応メカニズムが使われている研究(Raheel, 2016)では、GUI コンポーネントの位置や大きさをパラメーターとして適応的に変更する。また、ビデオストリームをネットワーク送信するためのエンコーダの研究(Maggio, Papadopoulos, Filieri, & Hoffmann, 2017)では、ネットワーク要件や品質要件に適合したビデオフレームを生成するために、エンコード品質やフィルタのパラメーターを適応的に変更する。

構造変更は、モジュールどうしのつながり方を変更することにより動作を変更する適応メカニズムである。Figure 2.3(b)においては、モジュール A にモジュール B が接続されていたものを、モジュール C に接続し直している。例として、地表の測量などを行う無人航空機(Unmanned Aerial Vehicle: UAV)の適応的動作の研究(Braberman, D'Ippolito, Kramer, Sykes, & Uchitel, 2015)では、GPS など一部の機器に故障が発生した場合に、ミッションを完遂するために代替のセンサーデータを使って同等の機能が提供できるようにモジュールの再構成を行う。

本研究では、Web アプリケーションにおける動的な実行環境の変化を対象にするが、その際に、利用者に最適なサービスを提供するために、サービスそのものの種類や内容を変更する必要がある。すると、サービスの特定のパラメーター変更では十分な変化を表現できない。そのため、メカニズムとしては構造変更を選択する。自己適応技術を Web アプリケーションに応用した研究には、負荷に応じたサーバーの増減を行うもの(Garlan, Cheng, Huang, Schmerl, & Steenkiste, 2004)や、画面に表示されたコンポーネントを最適化する研究(Raheel, 2016)など様々なものが提案されている。これらの研究における自己適応技術は Web アプリケーションフレームワークと統合した形で提案されていないため、アプリケーション開発者が実際に Web アプリケーションを開発する際は、Web アプリケーションフレームワークと自己適応技術の組み合わせを行う必要がある。

2.3 本研究の位置付け

これまで議論した関連研究から本研究を位置づけると、要点は静的対応と動的対応の 2 点となる。まず、Web アプリケーションの静的対応として様々なフレームワークが提案されているが、これらのサーバーフレームワークとクライアントフレームワークは構造が異なるため、それぞれ個別に実装されている。これにより、開発者は少なくともサーバー用およびクライアント用の 2 つのフレームワークを別個に学習する必要がある。また、スマートフォンの高性能化などの技術の変化により、クライアントにおいて高度な処理が可能になっているが、サーバーとクライアントのフレームワークが異なると、サーバー・クライアント間でロジックを容易に移行することができず、作り直しになってしまうという問題が発生する。これは 1.2.2 節で議論した本研究が解決しようとする課題 1 に相当する。次に動的対応については、開発者が Web アプリケーションフレームワークと自己適応という別々に進化する異なる複数の技術を使い、アプリケーションの実装ごとに組み合わせを行わなければいけないという問題がある。特に、スマートフォンなどデバイスの普及により激しくなっているクライアント環境の変化に即応していくには、Web アプリケーションフレームワークとして動的対応ができるようになっていることが必要である。これは 1.2.2 節で議論した本研究が解決しようとする課題 2 に相当する。

本研究で調査した範囲では、これらの問題に効果的に対処する試みは行われていないため、サーバーおよびクライアントのフレームワークの統合および Web アプリケーションへの自己適応技術の統合を行い、静的対応と動的対応双方を行うことができる Web アプリケーションフレームワークを構築することで両課題を解決する。動的対応を採用することにより無停止でクライアントの変化に対応しつつ、動的には対応しきれない大きな変化に対しては高い保守性でコードの変更を支援することにより効率的に静的対応を行う。両者がシームレスに統合されていることにより、開発者が一つのフレームワークで様々な変化に対応できるようになる。

第3章　モデルの提案

本章では、静的変化および動的変化双方に対応する Web アプリケーションを開発するためのフレームワークモデルを提案する。具体的には、1.2.2 節課題 1 の対応として、静的対応を実現するサーバーフレームワークとクライアントフレームワークの統合を行う。次に、それに加え、1.2.2 節課題 2 の対応として、動的対応を実現する自己適応技術を統合する。

3.1 静的対応の統合

前章で議論したように、Web アプリケーションを構築するためにはフレームワークを利用するのが効果的である。一方、サーバーフレームワーク(サーバー型)とクライアントフレームワーク(クライアント型)は動作および構造が異なる。ソフトウェアの構造が変わると 1 つのフレームワークで対応することはできないため、異なるフレームワークが提案されている。本節ではこの問題に対処する。

3.1.1 フレームワークの共通化

複数のフレームワークを使うと、学習コストや保守コストが上がってしまうため、サーバーとクライアントでフレームワークを共通化したい。ただ、現実にはサーバーとクライアントでは利用されるプログラミング言語や動くプラットフォームが異なるため、完全に同一な実装のフレームワークを双方で動作させることは困難である。そのため、本章では実装レベルでのフレームワークを議論するのではなく、抽象レベルでモデル化を行なって互換性の確保を行うこととし、実装できるかどうかについて第 4 章で議論する。

2.1.2 節で議論したように、既存で提供されているクライアント向けとサーバー向けの各フレームワークは、特に互換性に関しては考慮されていない。これはフレームワークが実装上の問題を解決するためであることが大きいと考えられる。しかし、各フレームワークを抽象化して考えると次のような共通点が認められる。

- 利用者からの要求(イベントや通信など)によって処理が開始する
- 処理では画面に入力された内容を入力データとする
- 処理を行なった結果は画面に反映される

これらを抽象的なレベルで共通化しモデル化することで、モデルレベルでフレームワークを統合できると考えた。

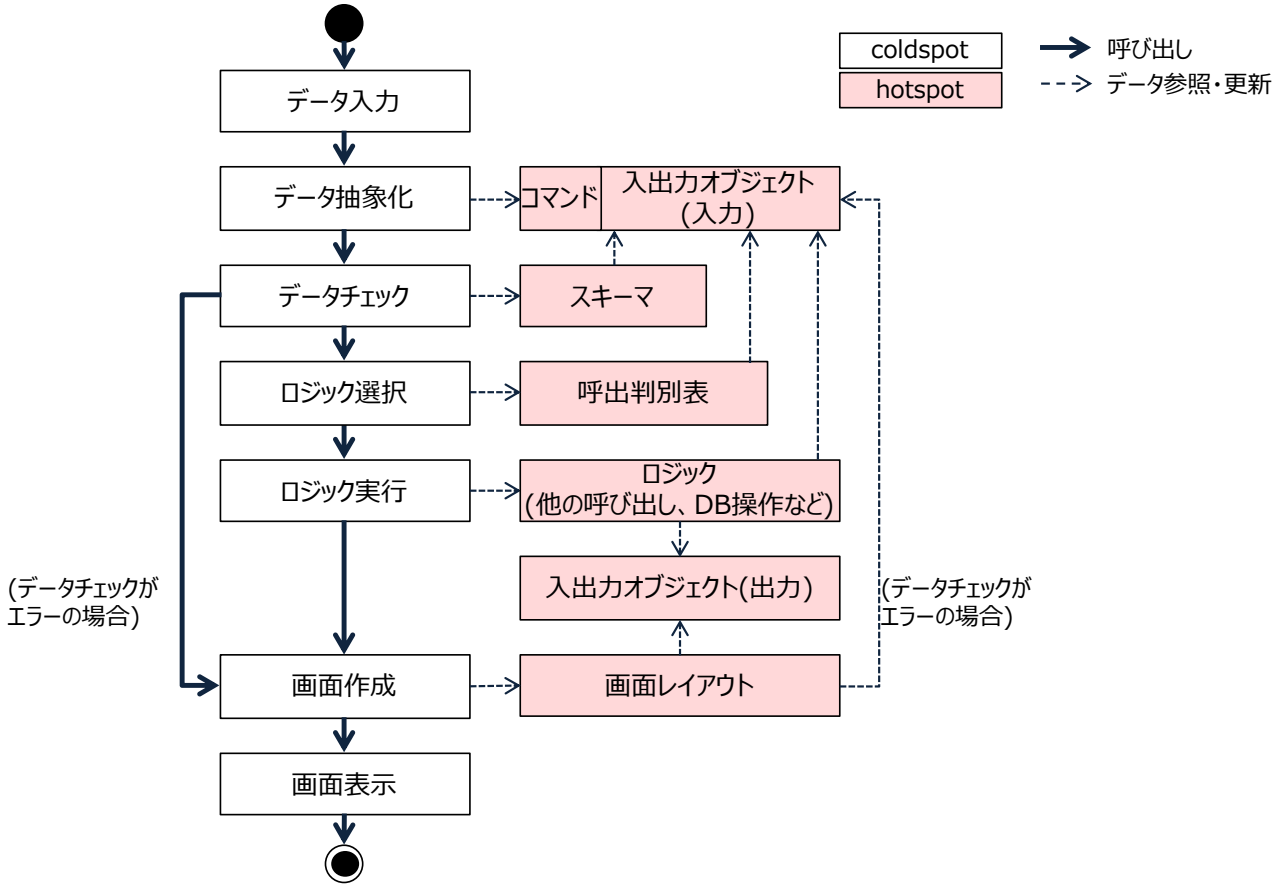

Figure 3.1 静的対応モデル

3.1.2 モデルの設計

前節の共通点を考慮しつつ、サーバー、クライアント双方の既存アプリケーションの実装を検討し、処理の流れを以下のステップで共通化した(Figure 3.1)。

ステップ1. データ入力
ステップ2. データ抽象化
ステップ3. データチェック
ステップ4. ロジック選択
ステップ5. ロジック実行
ステップ6. 画面作成
ステップ7. 画面表示

以降で各ステップについて説明する。なお、Figure 3.1 では、UML のアクティビティ図(UML, 2017)を参考に、以下の要素を使ってモデル表記を行なった。

- 初期ノード、最終ノード
  処理の開始、終了を示す。
- アクションノード
  動作モジュールを示す。モジュールはプログラムまたは定義体で記述される。hotspot と coldspot

に分かれる。

- 矢印
  ノード間の遷移を表す。具体的には制御の移動が行われる。
- データ参照・更新関係(破線矢印)
  モジュールからモジュールの参照または更新を行う。本矢印はデータフローを表すものではなく、参照または更新を行う方向を表す。

上記のプログラムと定義体の違いについて述べる。プログラムは、プログラミング言語を使い、逐次的に実行される命令列を記述したものである。定義体とは、XML や JSON、独自の表記法などを使い、情報を格納したもので、宣言的に記述されるものである。どちらが使われるかは実装によるため、ここでは議論しない。以下のステップでは、ほとんどの場合において逐次的記述を行う必要があるのはステップ 5 で扱うロジックのみとなる。他のステップについては特定の記述内容が決まっており、表などの定義形式で記述可能である。

ステップ1. データ入力

画面のデータのうち、利用者入力など処理対象になるものをフレームワークに入力する。この動作は coldspot のみで行われる。サーバーの場合は HTTP リクエストの入力、クライアントの場合は画面情報を表現する DOM (Document Object Model)からの情報の取得が相当する。

ステップ2. データ抽象化

ここでは、データ入力ステップにおいて入力されたデータを入出力オブジェクトにマッピングする。アプリケーション固有の hotspot として入出力オブジェクト及びコマンドを使用する。なお、入力された情報が格納された入出力オブジェクトを特に入力オブジェクト、処理結果として出力すべき情報を格納した入出力オブジェクトを特に出力オブジェクトという。入力オブジェクトと出力オブジェクトは実装が異なるわけではなく、同じインスタンスに対して場面によって役割が異なることを示している。入出力オブジェクトはデータを保持するためのプログラミング言語上のオブジェクトで、複数のキーと値を持つペアからなるデータ要素から構成される。これはサーバーであれば例えば POJO (Plain Old Java Object)やハッシュテーブル、クライアントであれば JSON (JavaScript Object Notation)などで表現することができる。疑似コードで記述した入出力オブジェクトの一例は以下となる。

```
user = {
    name: "John Smith",
    pwd: 1234,
    command: "logon"
}
```

ここでは、user という名前のオブジェクトが、name および pwd という名前の要素を持つ。それぞれ値は文字列の”John Smith”および数値の 1234 である。また、入出力オブジェクトの中に command という名前の要素でコマンドが定義されており、値は”logon”である。コマンドはオプションで、無指定の場合もある。入出力オブジェクトのどの要素が画面上のどこにマッピングされるかについてはここでは定義せず、後述の画面レイアウトで記述する。

ステップ3. データチェック

このステップでは、hotspot としてデータの型や値域を定義したスキーマを使い、入力データをチェックする。スキーマの用途は、データ入力ステップにおいてブラウザから入力されたデータが、アプリケーションで規定する内容に合っているかをチェックすることである。規定に合っていなければこの後のロジック選択およびロジック実行のステップを実行せずに画面作成ステップに移動して、エラーメッセージを画面に表示するという処理を行う。入力データにエラーがあった場合にどのようなエラーメッセージを表示するかなどの対処方法は、画面レイアウトに記述される。

以下にスキーマの例を示す。ここでは、name が文字列であり、pwd が 1000 から 9999 の間の整数であることが定義されている。スキーマ情報の要素は入出力オブジェクトの要素に対応する。

```
userSchema = {
    name: { type: "string" },
    pwd: { type: "integer", minValue: 1000, maxValue: 9999 }
}
```

本ステップではデータの型や値域などデータ単体で確認可能な条件を想定している。スキーマ表現の実装によってはデータベースを利用して複雑なチェックを行うことも可能であるが、宣言的な記述方法で複雑な処理を記述すると可読性が下がり保守性が悪化することがあるため、注意が必要である。

ステップ4. ロジック選択

呼出判別表を hotspot として使い、入出力オブジェクトとコマンドからどのロジックを実行するかを決定する。詳細については次節で説明する。

ステップ5. ロジック実行

ロジック実行ステップでは、プログラムとして記述されるロジックの本体を呼び出し、利用者の操作に対応した動作を行う。たとえば、サーバーであればデータベースの操作を行う、クライアントであればサーバーに問い合わせを行う、などの方法で処理結果となるデータを準備する。一般に、ロジックは遷移前の画面からの情報を使って必要な処理を行い、遷移後の画面の情報を作成する。ここで画面の情報は入出力オブジェクトを使う。遷移前後の画面が異なる場合は、2 つの入出力オブジェクトを扱うことになる (Figure 3.2)。

ステップ6. 画面作成

処理結果に基づき、画面を作成する。このとき、hotspot として画面レイアウトを使う。画面レイアウトは画面項目と入出力オブジェクトの対応、およびデータチェックが失敗した時の表示動作についての情報を持つ。

画面項目と入出力オブジェクトの対応では、画面の特定の項目が入出力オブジェクトのどの要素に対応するかを定義する。具体的には、HTML に記述される input などの入力項目それぞれについて、入出力オブジェクトの要素への対応関係が定義される。

Figure 3.2 ロジックと入出力オブジェクト

Table 3.1 呼出判別表の例

行	入出力オブジェクト*	コマンド	ロジック名
1	User	logon	login
2	User	pwd	chpass
3	Product	order	orderProduct
4	...	...	...

*厳密には「入出力オブジェクトのクラス名」だが、便宜的に略記する。以降も同様。

データチェックが失敗した時の動作では、ステップ 3 のデータチェックにおけるチェックの結果が渡される。その結果に対して画面上どのように対応するか、例えば赤文字でエラーメッセージを表示する、などの定義を行う。この時は異なる画面への遷移を行わないため、データ抽象化ステップで使われた入出力オブジェクトが画面レイアウトに渡される。

ステップ7. 画面表示

作成された画面を表示する。これは coldspot のみで行われる。サーバーの場合であればブラウザに対する HTML の送信が該当し、クライアントの場合は画面レイアウトの変更の DOM への反映と表示が該当する。

3.1.3 呼出判別表

モデルにおいて、ロジック呼び出しのために使われるのが呼出判別表である。呼出判別表の例について Table 3.1 に示す。なおここで行番号は説明用に便宜的に設けているものである。

呼出判別表は、入出力オブジェクトのクラス名および指定されたコマンドから、ロジックを決めるものである。クラス名とコマンドの両方を使う理由は、ある入出力オブジェクト、すなわちある画面から、複数の操作が可能な場合があるためである。たとえば、user オブジェクト(User クラスのインスタンス)がユーザ名及びパスワードから構成されている、すなわち利用者情報としてユーザ名とパスワードが入力可能になっている画面において可能な操作は、ログインかもしれないし、パスワード変更かもしれない。これを Table 3.1 で説明する。ログインもパスワード変更もどちらも User クラスの入力オブジェクトが入力される。ここで 1 行目では、コマンドが logon だったときに、ログイン操作なので login という名前のロジックを実行し、2 行目でコマンドが pwd だったときにパスワード変更なので chpass という名前のロジックを実行する設定とすることで、両者を分けている。コマンドを省略することも可能である。その場合は、コマンドが何も書かれていない判別表の要素にマッチする。実務上はこれを入出力オブジェクトに対するデフォルトの操作として扱う。

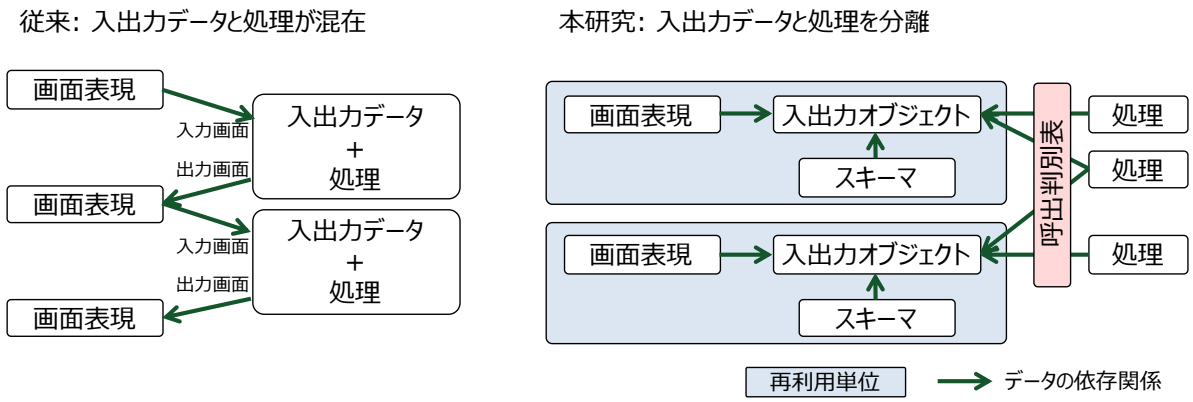

Figure 3.3 データと処理の依存関係

3.1.4 本モデルの特徴

本モデルは、呼出判別表を導入することにより、利用者からのリクエストに対応した処理を行う際の入出力データとその処理(ロジック)を分離したことに特徴がある。この様子を Figure 3.3 に示す。業務アプリケーションは多数の画面で構成されるため、画面の再利用ができることが重要である。従来、画面表現とロジックは相互に依存関係があるため、画面遷移の流れと同じ依存関係の連鎖が発生し、画面の再利用が困難であった。具体的には、ある画面からその画面の情報を処理するためのロジックへの依存があり、ロジックから次の画面への依存がある、という形になる(Figure 3.3 左側)。

本研究では、画面表現は入出力データ、すなわち入出力オブジェクトと組み合わせることで、処理とは分離できると考えた。ここで呼出判別表を導入することにより、画面や入出力オブジェクトから直接処理を依存させるのではなく、呼出判別表が入出力オブジェクトと処理を対応づけることとした。この方法により、画面から処理、処理から画面となる依存の連鎖を切ることができ、「画面表現と入出力オブジェクト」のセットで再利用性を確保できるようになった(Figure 3.3 右側)。

3.1 節で提案したモデルには、クライアント、サーバーという動作場所に関する概念は登場しない。そのため、本モデルはどちらのフレームワークとしても実装することが可能である。例えば入出力オブジェクトはクライアントでの実体(例: JSON)でもサーバーでの実体(例: Java オブジェクト)でも表現することができる。

3.2 動的対応の統合

Web アプリケーションフレームワークにおいて動的対応を行うためには、自己適応技術をどのようにフレームワークに適合させるかを考える必要がある。自己適応技術は幅広く、あらゆる技術を入れるのは現実的ではないし、学習コストの面から見ても効果的ではない。特に、1.2.2 節の課題 2 で設定したクライアントの状況変化に効果的に対応できるものでなければならない。そのため、本節では、自己適応技術のうち Web アプリケーションにおいて必要な範囲を考察し、前節で議論したフレームワークに統合するアプローチを取る。

3.2.1 自己適応の範囲

静的対応において検討したように、フレームワークモデルにおいてはロジックの部分以外は特定の記述内容が決まっており、定義体として宣言的に記述可能である。唯一、ロジックについては内容のバリエーションが極めて多く、プログラム記述が必要となる。つまり、Web アプリケーションにおいてはクライアントからの要求に対してどのようなロジックを実行するかが、もっとも変化が多く本質的な部分となる。そのため、ロジックを動的に変えることができれば、様々な環境の変化に対応することができるようになる。現実的には、ロジックを表現するプログラムそのものを動的に生成することは技術的に困難なため、状況ごとのロジックを準備しておき、その選択を動的に行うことで環境の変化に対応する。

3.1 節で議論したように、ロジックの選択は呼出判別表を使って行うため、この表を動的に変更される仕様と解釈すれば、自己適応モデルとしての設計が行える。具体的には、Weyns et al. (2013)のモデルにおいて、Managed Subsystem として呼出判別表を用い、Managing Subsystem が呼出判別表を更新するようにすればよい。その対応関係を Figure 3.4 に図示する。

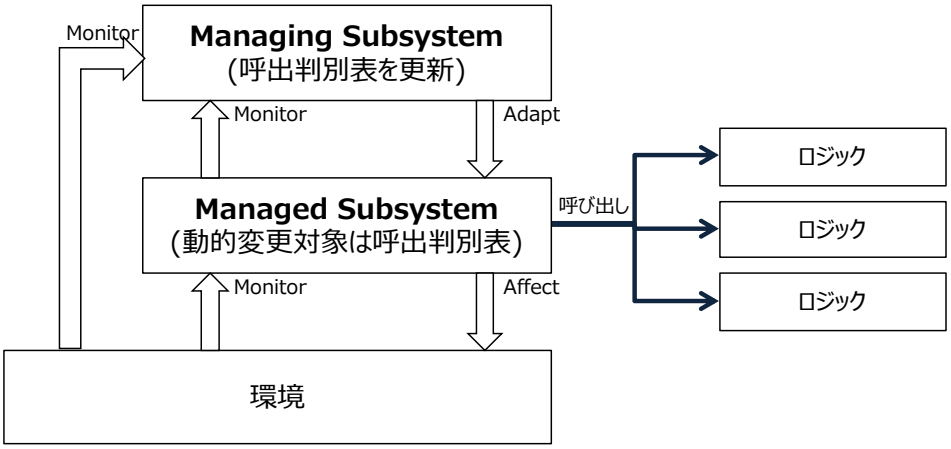

Figure 3.4 Weyns et al. (2013)と本研究の対応

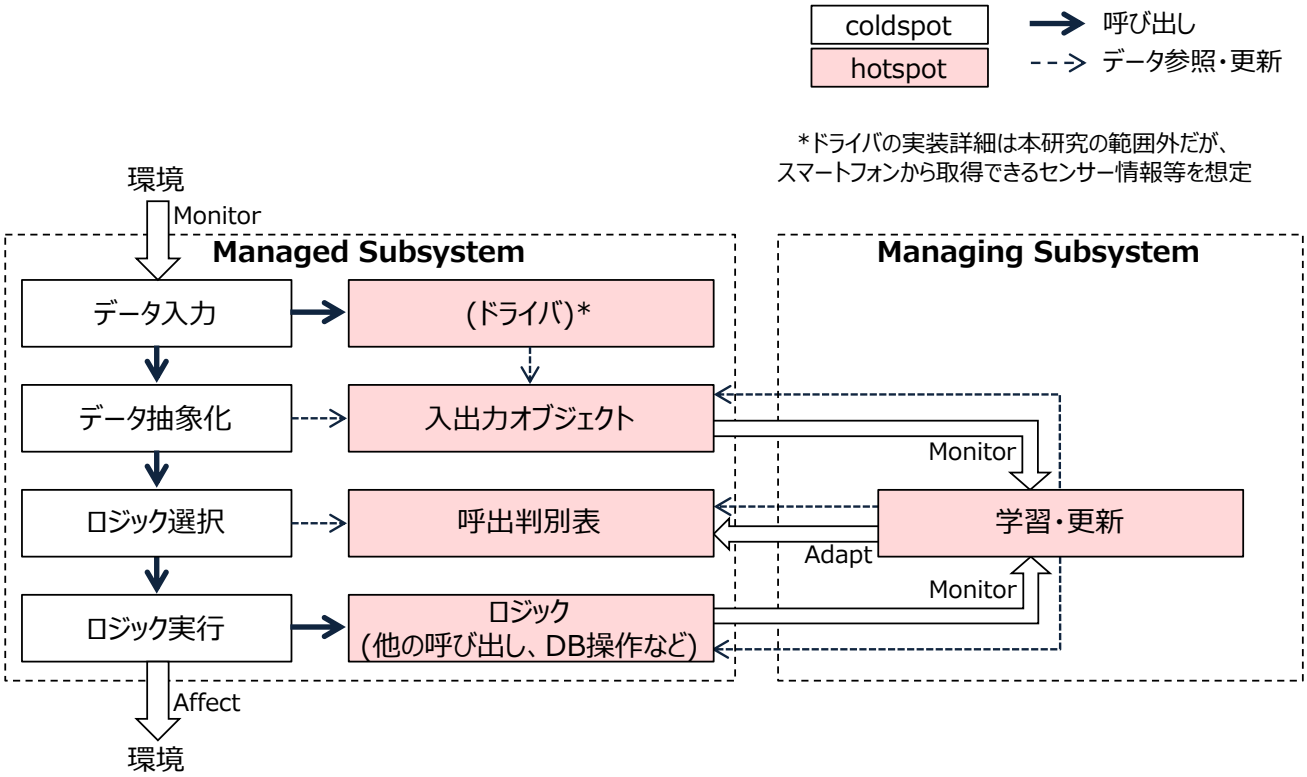

Figure 3.5 動的対応の追加

3.2.2 モデルの拡張

動的対応を可能にするために、Figure 3.4 の対応関係を参考にしながら 3.1.2 節で導入したフレームワークモデルの変更部分を Figure 3.5 に示す。ここでは、呼出判別表を Managed Subsystem の中で変更される仕様とみなし、それに対する Managing Subsystem として、学習・更新モジュールを追加している。学習・更新モジュールは入出力オブジェクトの情報とロジックからの情報を入力として自己適応に必要な学習を行い、結果を呼出判別表の更新として出力する。

静的対応においては、利用者から入力される情報を入出力オブジェクトにマッピングし、そのオブジェクトのクラス名を元に呼出判別表からロジックを選択していた。ここでは、より状況にあったロジックを選択できるようにするため、以下の 3 点においてモデルを拡張する。

1. 入力情報において、利用者からの入力だけでなく、環境からの入力も扱う
2. 入出力オブジェクトのクラス階層を扱う
3. ロジックの選択に入力値の条件を使う

環境からの入力

入出力オブジェクトには利用者からの入力だけでなく、環境からの情報も入力される。これは例えば利用者の位置であったり、入力が発生した時点のサーバーの負荷状況であったりする。そのため、データ入力ステップにおいて扱われるデータの情報源は複数になることがある。ただ、入出力オブジェクトにデータが設定されるタイミングはデータ抽象化ステップのままであり、その点において静的対応モデルからの変化はない。

環境からの情報を取得するためのドライバについては本研究の範囲外であるが、特にクライアントからの情報についてはスマートフォンの発達により様々な環境情報が取得可能である(Google, 2021; Apple, 2021)。これらについては Web ブラウザにおいても JavaScript を使って取得可能であり、利用者からの入力情報と同様に本フレームワークの入出力オブジェクトで扱うことができる。

入出力オブジェクトのクラス階層

呼出判別表において、入出力オブジェクトのクラス階層(スーパークラス、サブクラスの関係)を考慮したロジック選択を行うように拡張する。その理由は、変化する状況下においてある入出力オブジェクトに対応した処理ができない場合に、段階的に入出力オブジェクトのスーパークラスに対応するロジックを代替処理として選択する、一種の縮退を可能にするためである。

Figure 3.6 に示すショッピングモールの例で説明する。ショッピングモールにおいて、利用者がいる場所に応じた店舗アプリを使えるとする。たとえば利用者が店舗 A にいれば、その位置を検出することにより店舗 A のサービス serviceA を提供することができる。また、店舗 B にいれば店舗 B のサービス serviceB を提供することができる。これは利用者が GPS などで取得した自身の位置情報の利用を許可することにより可能となる。もし、利用者がいる場所がわからない、またはどの店舗にもいない場合は店舗のサービスを提供する代わりに対応店舗を表示したり、ショッピングセンター全体の地図を表示したりする show_avail を提供した方がよい。さらに、利用者がそもそも利用者登録をしていない場合は、利用者登録を促してサービスを利用して欲しいので、reg_user というロジックを実行したい。Figure 3.6 ではこれらの利用パターンを入れ子の構造で表現している。より内側の枠内が利用者にとってより特定の状況で、利用者から提供される情報も充実している。逆に外側の枠内は、利用者からの提供可能な情報が少ない。

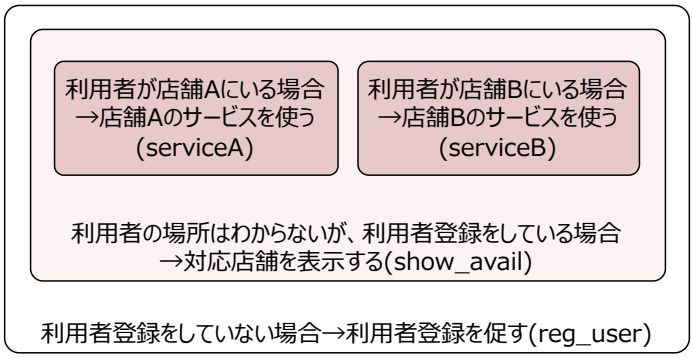

Figure 3.6 ショッピングモールにおけるサービス提供

これを入出力オブジェクトのクラスの構造に対応づけると、より特定の状況をサブクラスとして表現することができる。これを入出力オブジェクトのクラスとして、疑似コードで記述すると以下のようになる。

```
class C1 {
    id: String
}
class C2 extends C1 {
    name: String
}
class C3 extends C2 {
    loc: Position
}
```

C1 は最も一般的な場合、すなわち利用者登録をしていない場合の入出力オブジェクトのクラスで、id のみを持つ。C2 は C1 を継承して作られるクラスで、追加のメンバーとして利用者名を表す name を持つ。これは、利用者登録をしているということを想定している。C3 は C2 を継承して作られるクラスで、追加のメンバーとして位置情報 Position を表す loc を持つ。ここで Position は利用者のデバイスから GPS などを利用して得られるものとする。

ロジックを呼び出す時には、入出力オブジェクトの情報を元に、サブクラスから順にマッチングを行なって対応するロジックを見つける。まず位置情報を提供するクラス C3 の入出力オブジェクトに対しては店舗のサービスを提供するロジック serviceA や serviceB が呼び出される。しかし、位置情報がない、または位置情報があってもどの店舗ともマッチしないのであれば、C3 の親クラスである C2 が選択される。このクラスは利用者登録をしている入出力オブジェクトを表すので、対応店舗を表示する show_avail を呼び出す。さらに、C2 に期待される情報、つまり利用者情報がない場合には C2 のさらに親である C1 が選択され、利用者登録を促す reg_user を呼び出す。

入力値の条件

C3 が入力された時に適切なロジックを選ぶためには、実際にはクラスだけでなく、値の範囲も必要で

ある。利用者から位置情報が提供された時に、その場所が店舗 A であれば店舗 A のロジック serviceA を呼び出し、店舗 B であれば店舗 B のロジック serviceB を呼び出す。

以上により、入出力情報に応じたロジックを選択する際に、情報が得られない、または条件が合わなければ、条件を緩和して親クラスに対応させることにより、その状況に合わせたロジックを対応づけることができる。これをクラス階層にしたがって再帰的に繰り返すことにより、段階的に縮退させたロジックを実行することができるようになる。

3.2.3 呼出判別表

前節の内容を呼出判別表の形式にしたものが Table 3.2 である。これは、Table 3.1 で導入した判別表に対し、条件の項目が追加されている。なおここで行番号は説明用に便宜的に設けているものである。

1 行目および 2 行目は、どちらも C3 というクラスの入出力オブジェクトに対し、call というコマンドが指定された時のものである。Table 3.1 と異なり、それぞれの行に条件が指定されている。1 行目は、利用者の現在位置locが店舗 A の位置pos_Aであれば、serviceA を実行することを定義している。同様に 2 行目は、利用者の現在位置locが店舗 B の位置pos_Bであれば、serviceB を実行することを定義している。つまり、同じ入力オブジェクト、同じコマンドが入力されても、その時の入力オブジェクトの要素が示すデータの内容によって選択が変化する。3 行目は C2 クラスの入出力オブジェクトが入力された時のロジックを示している。このとき、条件として name が null でない時のみ、show_avail が実行される。同様に、入力が C3 クラスの入出力オブジェクトだったときに、1 行目および 2 行目で示された条件に合わない、すなわち利用者の現在位置が店舗 A、店舗 B のいずれでもない(そして name が null ではない)場合にも show_avail が選択される。さらに、入出力オブジェクトが C1 クラスだったとき、または 1 行目から 3 行目までの条件に合わなかったときに、4 行目の reg_user が選択される。ここでアスタリスクはワイルドカード(任意の入力にマッチする)を意味する。

最終的に入出力オブジェクトに対応するクラスのエントリーがない、またはいずれの条件にも合わない場合は、6 行目のエントリーが対応する。この例では、最終的な縮退ロジック fallback を実行する、となっている。ここではシステムレベルの例外処理を行ったり、管理者に通知したりするなどの対応を取ることができる。

Table 3.2 拡張した呼出判別表

行	入出力オブジェクト	コマンド	条件	ロジック名
1	C3	call	$loc \in pos_A$	serviceA
2	C3	call	$loc \in pos_B$	serviceB
3	C2	call	$name \neq null$	show_avail
4	C1	*	*	reg_user
5	...	...	...	...
6	*	*	*	fallback

Table 3.3 更新された呼出判別表

行	入出力オブジェクト	コマンド	条件	ロジック名
1	C3	call	$loc \in pos_A$	serviceA
2	C3	call	$loc \in pos_C$	serviceB
3	C2	call	$name \neq null$	show_avail
4	C1	*	*	reg_user
5	...	...	...	...
6	*	*	*	fallback

3.2.4 動的更新

呼出判別表は動作時に変更可能である。これは動作中の仕様変更に相当し、Figure 3.5 に示したように、更新のためのアルゴリズムは Managing Subsystem の学習・更新モジュールに実装する。学習・更新に使うデータは、入出力オブジェクトからのデータとロジックからのデータである。処理結果は呼出判別表に反映する。

Figure 3.6 に示すショッピングモールの例で説明する。店舗 B が固定の店舗ではなく、ストール(移動店舗)だとする。このとき、serviceB において店舗 B の移動先が把握できるため、学習・更新モジュールは serviceB から移動先の情報を取得する。これをもとに更新された呼出判別表を Table 3.3 に挙げ、更新部分を網掛けで示す。ここでは、店舗 B の位置がpos_Bからpos_Cに変更されている。この後の利用者からのリクエストは、その位置条件が変わっており、pos_Cにいるときに serviceB が呼び出されるようになる。

3.3 モデルの設計

以上のことから、静的対応および動的対応双方を含めたモデルを Figure 3.7 に示す。各ステップにおける hotspot とその関係は、以下の通りである。

- データ抽象化ステップにおいて、データ入力ステップで得られたデータから入出力オブジェクトを作成する。
- データチェックステップにおいて、スキーマで指定された内容に従いが入出力オブジェクトの内容をチェックする。チェックが失敗した場合は画面作成ステップに移行し、画面レイアウトが入力オブジェクトを使って処理結果となる画面を作成する。
- チェックが成功している場合はロジック選択ステップにおいて呼出判別表を参照し、呼び出すべきロジックを選択する。
- ロジック実行ステップでは、選択されたロジックを呼び出す。ロジックは入力オブジェクトのデータを参照しながら他のサービスの呼び出しやデータベースの操作などを行い、処理結果を出力オブジェクトに格納する。
- 画面作成ステップでは、画面レイアウトが入出力オブジェクトを使って処理結果となる画面を作成する。

学習・更新モジュールは 3.2.4 節で説明した動作を担い、入力オブジェクトとロジックからの情報を参照し、呼出判別表を更新する。この処理は各ステップとは非同期に実行できる。

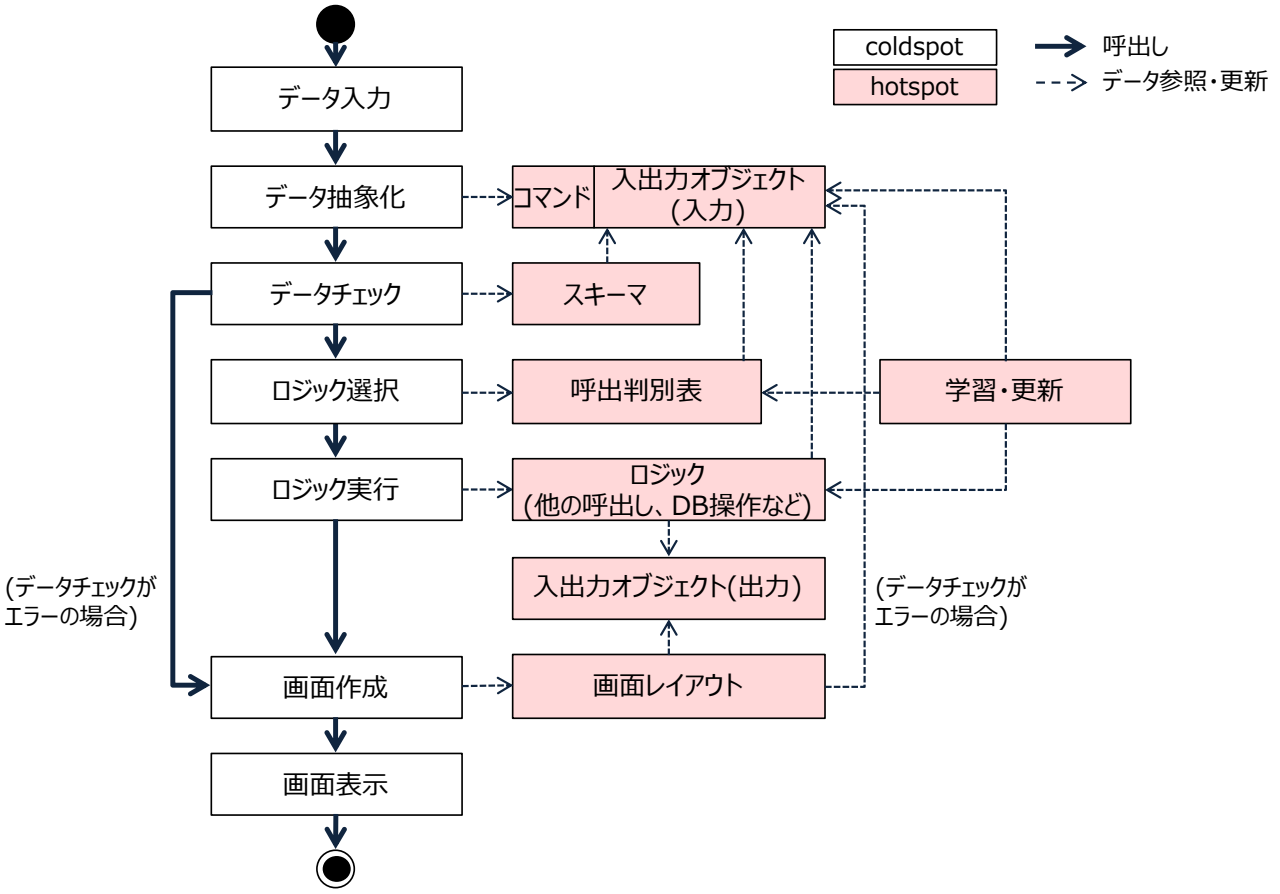

Figure 3.7 静的対応・動的対応を統合したモデル

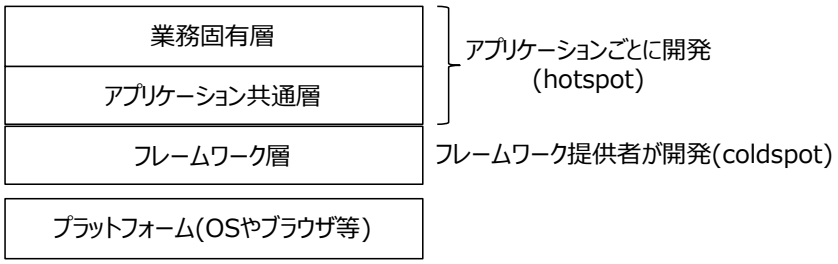

Figure 3.8 フレームワークを利用したアプリケーションの開発分担

3.4 考察

本節では、本研究で提案したモデルが持つ意味について考察する。

3.4.1 保守性

2.1.3 節において、保守性についての指標、すなわちコード量、関心の分離、再利用性について検討した。ここでは、モデルの導入でこれらの指標の達成がどのように期待できるかについて議論する。なお、実際にどのように達成したかについては第 4 章で示す。

コード量

まず、コード量については、フレームワークの導入で実装が hotspot のみになるため、低減される。さらに本フレームワークモデルでは、入出力オブジェクト、スキーマ、呼出判別表、画面レイアウトについては宣言的に定義できるため、GUI ツールを使っての作成や、Microsoft Excel などの外部ツールでの開発が可能となる。4.1 節および 4.2 節において、実際のフレームワークにおけるコード量の低減効果について検証する。

関心の分離

関心の分離については、2.1.3 節でユーザインタフェースのコードの分離が重要であることを示した。本フレームワークモデルでは、ユーザインタフェースに関連するコードは入出力オブジェクトと画面レイアウトである。この両者にはユーザインタフェース以外のコード、すなわちアプリケーションロジックやデータベース操作などに関するコードは入らないため、プレゼンテーションとドメインの分離は達成可能である。また、それぞれの hotspot 要素の間の依存関係が定義されているため、変更がある場合の影響範囲についても容易に見積もりが可能となる。

関心の分離は誰が何を実装するかに大きく影響する。実際のアプリケーション開発では、全体を 3 つの層に分け、必要に応じて別の担当者が実装する。この様子を Figure 3.8 に示す。アプリケーション実行のベースとなる OS やクラウド、ブラウザなどのプラットフォームの上に、まずフレームワーク層があり、これはフレームワーク提供者が開発する。フレームワークは様々なアプリケーションで利用できるように作るために高い技術が要求されるが、製品や OSS (Open Source Software)などの形式で提供されるため、個別のアプリケーション開発者が実装する必要はない。次にアプリケーション共通層で、あるアプリケーションを開発する際に共通で使える画面や部品などを実装する。最後に業務固有層で、各業務の画面やアプリケーションロジックの開発を行う。アプリケーション共通層と業務固有層は同じ開発者が実装することも、異なる開発者が実装することもある。大規模なアプリケーションの場合、両者で開発者チームを分けることがある。その場合、業務固有層は対象となる業務に詳しい開発担当者が実装し、アプリケーション共通層は比較的技術スキルの高い共通チームや基盤チームと言われる開発者が実装を担当する。

各層において開発する hotspot をモデルとの対応関係で示すと以下の通りになる。本フレームワークモデルでは、各 hotspot が関心に相当する。

- 業務固有層: 入出力オブジェクト、スキーマ、呼出判別表、ロジック、画面レイアウト
- アプリケーション共通層: 呼出判別表、学習・更新

この中で、呼出判別表は業務ごとに決められる業務固有層に相当する部分と、アプリケーション全体に影響するアプリケーション共通層に相当する部分が存在する。入出力オブジェクトにクラス階層があることで、サブクラスのエントリーについては業務固有層、スーパークラスのエントリーについてはアプ

リケーション共通層という分類で記述することで、役割分担が可能である。これを実務上は、呼出判別表を複数のファイルに分けて、各業務用の判別表と共通の判別表として記述し、最後にマージするか、全てのファイルを実行時に読み込むようにすれば良い。

以上のことから、開発・保守の場面においては、関心ごとに担当者に割り当てができる。担当者ごとに独立に hotspot をメンテナンスすることができるため、保守性を向上することができる。

再利用性

再利用性については、3.1.4 節で議論したように、入出力オブジェクトと画面レイアウト、オプションとしてスキーマを組み合わせた画面単位での再利用が可能である。業務アプリケーションでは多数の画面を複数の場面で扱うため、画面を部品化して組み合わせ・再利用できることによる効果は大きい。たとえば、従業員を示す Employee オブジェクトとその画面レイアウト、申請期間を表す ApplyDate オブジェクトとその画面レイアウトをそれぞれセットで部品化したとする。入出力オブジェクトはそれぞれ疑似コードで次のように表せる。

```
class Employee {
    employeeId: Integer
    employeeName: String
    divisionCode: Integer
    division: String
}
class ApplyDate {
    startDate: Date
    endDate: Date
    validStartDate: Date
    validEndDate: Date
}
```

この部品は業務アプリケーションの各画面に表示される従業員情報および申請期間情報として再利用ができる。これを 4.1 節で取り上げるアプリケーション例から Figure 3.9 で示す。ここでは国内出張旅費申請と年次休暇申請という 2 種類の業務画面を例とする。破線枠の部分が Employee オブジェクトを表示する箇所、点線枠の部分が ApplyDate オブジェクトを表示・入力する箇所として、複数の画面で再利用されている様子を示している。このように、業務アプリケーションにおいて似たような項目を複数の画面で入力・表示する必要性は高く、画面を部品として再利用できると効果が高い。

なお、この手法で作られる部品を 2.1.1 節で議論したプロダクトラインのコア資産として捉え、複数のアプリケーション間で再利用を行うことも論理的には可能である。しかし、現実のソフトウェア開発では画面の個別性が極めて高いため、あるアプリケーション用に開発した画面部品を他のアプリケーションに流用することは困難である。このことから、本研究では部品の再利用に関してプロダクトラインのアプローチは採用していない。

Figure 3.9 画面部品の再利用

3.4.2 フレームワーク間の移植

抽象レベルでフレームワークを定義することにより、技術変化により異なる言語やプラットフォームの上でアプリケーションを動作させなければいけないという要求が発生した場合にも、当該の言語やプラットフォーム上で同じモデルに基づくフレームワークを実装することにより、hotspot については自動変換または半自動変換でアプリケーションを移植できるようになることが期待できる。

ここではその例として、サーバーで実装した hotspot をクライアントに移植することを検討する。スマートフォンなどのデバイスの高機能化・高性能化により、従来サーバーで実装していた Web アプリケーションの機能をクライアント上で実行させることができるようになっている。これまで、サーバーフレームワークとクライアントフレームワークは実装が異なり、このような移植は手作業に依っていた。本研究で提案するモデルを利用したフレームワークの場合、適切な変換ソフトウェアを開発することで hotspot をフレームワーク間で変換することが可能となる。実際に変換を行うソフトウェアを実装しようとしたときに各 hotspot の要素に対して留意する点を挙げる。

- 入出力オブジェクト
  これは単なるデータ定義のため、容易に変換が可能である。スキーマにも関係するが、データ型、例えば数値精度などがプログラミング言語によって異なることがある点には注意が必要である。
- スキーマ
  動作する言語がフレームワークによって異なるため、言語間の型の互換性に留意する。また、チェックの結果エラーが出た時の動作に違いが出る可能性がある。クライアントフレームワークでは入力の瞬間にエラーメッセージを表示できるのに対し、サーバーフレームワークでは HTTP 通信によって結果がクライアントに返されたときに画面が更新される。そのため、実装によってはエラーメッセージの表示タイミングの違いが出る場合がある。
- 判別表
  動作言語によって条件やロジック名の表現方法に差異があるため、その点に留意して変換を行う。
- 画面レイアウト
  Web アプリケーションにおいては、画面レイアウトは HTML であることが共通しているために多くの部分で変換は不要となる。入出力オブジェクトとのマッピング方法にバリエーションが出る可能性がある為、その点を留意して変換を行う必要がある。具体的には、サーバーフレームワークが HTML の標準機能(HTML タグなど)により入力データの指定をしている一方、クライアントフレームワークは独自のデータ指定方法を使うことがある。
- ロジック
  ロジックは記述範囲が広く、言語の違いがあるため変換の難易度は他の hotspot に比べて高い。しかし、Google による J2CL (Gokdogan, 2021)などのトランスパイラー(言語変換器)が開発されており、ある程度の手作業を行うことで半自動変換が可能となる。また、他サービスを呼び出す際の通信方法の違いなどはフレームワーク側で吸収が可能である。

プログラムや定義体の変換が可能な場合でも、実際にはセキュリティに関する制約と、性能による制約を受ける場合がある点は注意が必要である。例えば、ロジックの中からデータベース呼び出しを行っている場合、サーバーで動作させるプログラムであればそれほど問題はない。しかし、クライアントで動作させる場合、JavaScript プログラムが利用者から見えてしまうという点から、SQL インジェクションなどの攻撃対策を取る必要がある。また、クライアントが直接サーバーにあるサービスを呼び出す場合は、ブ

ラウザに Same Origin Policy (MDN, 2021a)があるため、呼び出し先に限界がある。性能による制約としては、移植前の hotspot がサーバーにおいて他のサーバーコンポーネントと密接にインタラクションするロジックだったとする。このような場合はクライアントに移行することによって通信が多く発生することにより性能が劣化する可能性がある。これらのことから、ロジックについては自動変換が可能としても、人による確認は必要である。

3.4.3 クラス階層の意味

3.2.2 節において、入出力オブジェクトにクラス階層を導入し、条件選択と合わせて柔軟なロジック選択を可能とした。この仕組みにより、以下の 2 点が可能となっている。

個別化対応

一つは、個別化対応である。個別化はパーソナライゼーションとも言われ、利用者のニーズに基づいて企業が利用者とつながることを可能にする(Zealley, Wollan, & Bellin, 2018)。そのためには、利用者の特定のニーズに合わせてサービスを適応させるメカニズムが存在する必要がある(Jørstad, van Thanh, & Dustdar, 2005)。

ニーズは入力される利用者や環境のデータとして表現されるため、呼出判別表では、得られたデータの値の条件に基づいて呼び出すロジックを決定することで、ニーズに合わせた選択を可能としている。ここで、利用者の特定のニーズが判断できるデータが常に取得できることが望ましいが、必ずしもすべての情報を常に全ての利用者から得られるとは限らない。例えば、利用者がスマートフォンを使っている場合に、そのプライバシー設定によっては位置情報を取得することができないかもしれない。このような場合に、入出力オブジェクトがクラス構造を持つことで、クラスの親子関係を使うことにより表すデータの多寡を表現することができる。何らかの理由でサブクラスが必要とするデータを全て用意できない場合に、代わりにスーパークラスを使ったエントリーを呼出判別表に設けることで、利用可能なデータのみで処理可能なロジックを呼び出す。

以上により、特定の入力や環境の条件に合わせた、より個別性の高いサービスを提供することが可能となる。

更新対応

クラス階層の導入で可能になったもう一つの点は、更新への対応である。アプリケーションは継続的に更新され続けるが、Web アプリケーションのようにネットワークを利用したソフトウェアの場合は、この継続的な更新によって問題が発生することがある。具体的には、ノード間で導入しているソフトウェアのバージョンが合わなくなることである。

ここで Table 3.2 で示す判別表で動作しているソフトウェアを更新する例を挙げる。これまでクラス C3 では位置情報を提供していたが、これに決済情報も提供するようなクラス C4 にする。C4 は次のように定義できる。

```
class C4 extends C2 {
    loc: Position
    payment: Creditcard
}
```

Table 3.4 サービスの更新

行	入出力オブジェクト	コマンド	条件	ロジック名
1	C4	call	$loc \in pos(x_A, y_A)$	serviceAnew
2	C4	call	$loc \in pos(x_B, y_B)$	serviceBnew
3	C2	call	$name \neq null$	show_avail
4	C1	*	*	reg_user
5	...	...	...	...
6	*	*	*	fallback

Table 3.5 メソッド呼び出し説明用の呼出判別表

行	入出力オブジェクト	コマンド	条件	ロジック名
1	C2	call	*	serviceA
2	C3	call	$c \geq 0$	serviceB
3	C1	call	*	serviceC

これに合わせ、サービスも入出力オブジェクト C4 に対応するロジック serviceAnew および serviceBnew とする。すると、判別表は Table 3.4 の網掛け部分のように更新される。これで通常の場合は問題なく新しいサービスが動作する。しかし、クライアントとサーバーはネットワーク的に離れているため、全てのネットワークノードにおける更新が同時に完了するわけではない。そのため、サーバーにおける更新が完了しても、クライアントの更新が完了していない場合が発生する。この場合に、クライアントからは古いバージョン C3 のデータが送出されてしまう。C3 に対応する項目は呼出判別表に存在しない。しかし、C3 の親クラスは C2 であることが分かっているため、C3 に対応する表のエントリーがなくても、その親クラスである C2 のエントリーを選択することができる。つまり入力オブジェクト C3 に対しては C2 とみなして呼出判別表を引き、対応するロジック(show_avail)を実行することが可能である。

これは、環境や要求の変化に対して静的な対応によりソフトウェアの変更を行なったとしても、それが実際にアプリケーションのシステム全体に波及するまでには時間がかかる場合があることを示しており、それを動的な仕組みで吸収していることになる。

3.4.4 メソッド呼び出しのネットワーク拡張

本フレームワークモデルでは、呼出判別表を使うことにより、入力されるデータをロジック名にマップしている。これは、プログラミング言語としてはメソッド呼び出しに相当する。例として、次のクラス定義および Table 3.5 で表す呼出判別表があるとする。

```
class C1
class C2 extends C1
class C3 extends C1 {
    c: Integer
}
class C4 extends C1
```

ここで呼出判別表の 1 行目が選択されるのは、引数のクラスが C2 の場合である。2 行目が選択されるのは、引数のクラスが C3 で、かつ C3 のメンバ変数 c が非負の場合である。引数のクラスが C3 でかつメンバ変数 c が負の場合か、引数のクラスが C1 の場合は、3 行目が選択される。このことをプログラミング言語で考えると、

```
call(C2)
call(C3)
call(C1)
```

という 3 つのメソッドをオーバーロードしていることに相当する。ただしプログラミング言語のオーバーロードにおいては、引数条件は存在しないため、この部分は本研究による拡張である。ここでメソッドは呼出判別表で指定されるロジックであり、本フレームワークがクライアント・サーバー環境で実行できることから、ネットワーク上別のノードにあるメソッドを呼び出すことができる。

従来、ネットワーク環境でソフトウェア同士がインタラクションを行うために CORBA や WSDL といった仕組みが提案されてきた。これらは異種言語間でのメソッドコールを実現しているが、本研究の仕組みでは以下の 2 点において拡張されている。

一つは、値の条件が存在することである。本フレームワークモデルにおける呼び出しはネットワークを介することがあるため、条件が合わない場合のコストが高いことから、あらかじめ条件句で値域を指定することにより、条件に合わない場合の呼び出しを排除している。

もう一つは、言語非依存でオーバーロード、すなわち同一のコマンド名称で異なるロジックへの呼び出しが実現されていることである。ネットワークを介してメソッドコールを実現する手法は様々なものが提案されているが、異種環境における呼び出しを実現するために、実装言語による制約を受ける。具体的には、異種環境で使う言語は多岐に渡り、ある言語でサポートされている機能が別の言語ではサポートされていないことがある。オーバーロードもその一つで、C++や Java などオブジェクト指向言語の多くでオーバーロードがサポートされる一方、C や COBOL などではサポートされない。このことから、オーバーロードは明示的に禁止されていたり、問題が起きたりすることが報告されている(Chumbley, Durand, Pilz, & Rutt, 2010; 尾島, 1999; Beugnard & Sadou, 2007)。本フレームワークモデルでは、フレームワークエンジン側でオーバーロードを処理してしまうため、言語による制約が発生しないことが特徴である。

これを別の観点で見ると、本来ソースコード、つまり静的対応で行う呼び出しの変更を動的対応で行っていることになる。この考え方を推し進めると、Web アプリケーションのロジックの呼び出しという点においては、動的対応のみでさまざまな変化に対応していけるようになる。なお、動的対応による呼び出し変更を可能とするためにはフレームワークモデルを導入しインタフェースを呼び出し側と呼び出される側の双方で合わせる必要があるので、静的対応が不要ということにはならない。

3.4.5 動的対応の適応レベル

本研究で可能となる適応のレベルについて述べる。自己適応技術においては、その適応がどのような変化に対応できるかが議論されている(Grua et al., 2019; Laprie, 2008)。これは、大きく分けて Foreseen, Foreseeable, Unforeseen の 3 種類がある。

Foreseen は設計時に予期されており、システム動作時に起こり得る変化について対応できるかどうかである。本研究の方法では、環境や利用者の入力データが十分でない場合や、入力の内容があらかじめ条件づけられた範囲を超えている場合に、入出力オブジェクトの親クラスのエントリーを選択することにより対応ができる。

Foreseeable は設計時には予期されていないが、システム動作時に対処し得る変化について対応できるかどうかである。本研究では、これは呼出判別表において対応できる内容が存在しない場合に相当する。そのような場合に備えて、呼出判別表にはワイルドカードがあり、設計時に予期していない条件に対してシステムレベルでの例外処理などを割り当てることができる。

Unforeseen は初めて起こるまで予想できない不測の変化のことである。このような不測の変化は、設計時と実行時の両方において，本質的に不確実性のレベルが高いために対応は非常に困難である。これまでこのような状況に対応した研究は存在せず、本研究でも対応していないが、将来的には挑戦的な研究領域である(Grua et al., 2019)。

3.4.6 Web アプリケーション以外への適用可能性

フレームワークは実装技術であるため、一般には特定の問題領域に対して実装される。Web アプリケーションにおいても、サーバーとクライアントでは問題領域が異なるため、フレームワークの実装としては異なるものになるが、今回はこれを一段抽象化することで、フレームワークのモデルとしてサーバーおよびクライアントを共通化した。これをさらに抽象化することで、Web アプリケーション以外にも利用できる可能性がある。ここでは本フレームワークモデルの特徴や制約について検討する。

まず、フレームワークであるため、実行の際にフレームワーク側に制御があり、ステップごとに hotspot に制御が渡されるという特徴がある。これは、Web アプリケーションがクライアントからのイベントやリクエストによって 1 つの実行を開始するためで、実行開始時の制御(スレッド)は必然的にフレームワーク(coldspot)側が持ち、各処理はそこから呼び出される形になることに起因している。そのため、hotspot は呼び出されたときにそのスレッドを永遠に使い続けることはできず、必要な処理を行なったら終了して制御を coldspot に返さなければいけない。特に、多くの GUI においてはシングルスレッドモデルを採用しているため、ある処理がスレッドを使い続けると画面全体がデッドロックに陥ってしまう。このことは、hotspot、特に本フレームワークモデルではロジック部を実装する際の制約となる。また、実装レベルのフレームワークでは、各 hotspot を呼び出せる、または参照できるようにするために、インタフェースも明確に定まる。これらのことは、hotspot 実装者にとっては自由度が減ることを意味するが、実装内容が明確になるというのはフレームワーク本来のメリットでもある。

次に、本フレームワークが持つ特徴的な機能は、入出力オブジェクトに基づいてロジック呼び出しの制御を行うことである。これに加え、入出力オブジェクトのクラス階層を持つことでロジックの選択を入力に応じて動的に選択機能を縮退させることと、実行中の呼出判別表の更新を可能にすることにより環境変化に応じて呼び出し先を動的に変更させることを実現している。

これらの特徴をもとにフレームワークのコア部分を抽出すると、Figure 3.10 となる。このコア部分が表

すのは、イベントからデータを取得しそれをもとにロジック選択を行うという、リアクティブに動作するメカニズムである。また、そのロジック選択を状況の変化において動的に更新する仕組みが備わっている。

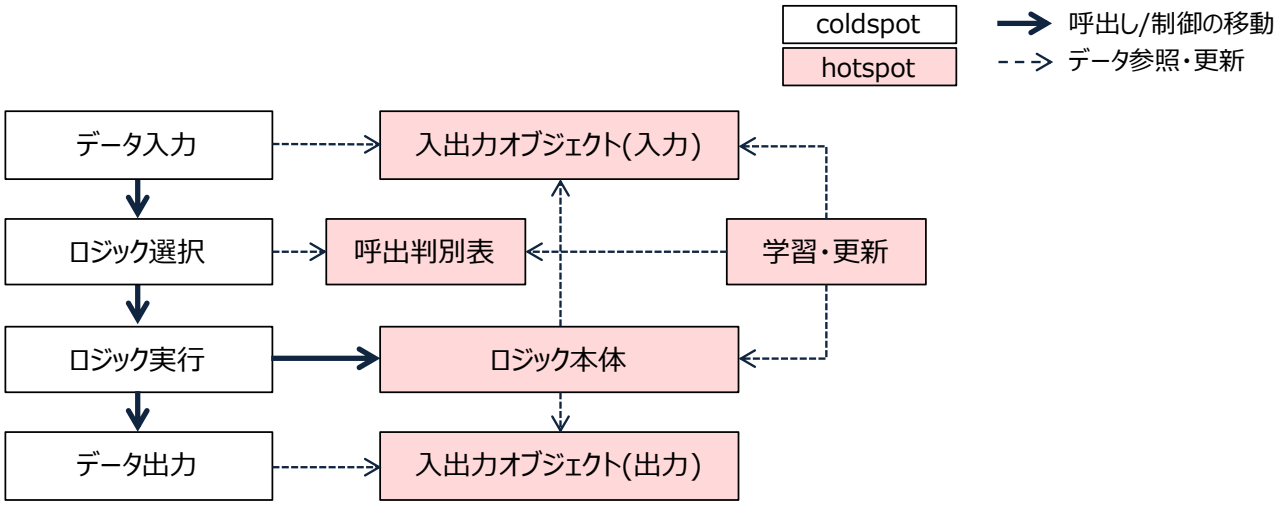

Figure 3.10 フレームワークモデルのコア

このような機能が有効に機能する可能性のあるソフトウェアの例として、Web アプリケーション以外のユーザインタフェースを実装するフレームワークがある。本研究に関連して、筆者の研究においてクライアント・サーバー型のアプリケーションフレームワークにおいて類似するアーキテクチャを採用している(Matsutsuka, 2000)が、これがその一例である。また IoT アプリケーションにおける、デバイスイベントに対する動作の選択などは、デバイスからの情報に基づいて動作を変更する必要があるため適合すると考えられる。

もう一つは、マルチエージェントシステムにおけるエージェントの実装が挙げられる。エージェントとは目的を達成するために自律的に動作を行うソフトウェアコンポーネントで、マルチエージェントシステムは複数のエージェントが相互にインタラクションすることにより、解析的に解くことが難しい複雑な問題を解くシステムである。たとえば、交通システムでは、絶えず変化する周囲の状況に合わせ、車や歩行者、信号機などの各エージェントがどのように動けば最適かのシミュレーションを行う。エージェントの動作はルールで記述されるが、これは呼出判別表の動作に相当する。呼出判別表から選択されるロジックをエージェント内外から選ぶことで、自身で処理できるものはエージェント内で解決し、できないものは外部のエージェントに依頼するという動作が記述できる。また、状況が変わることにより学習・更新モジュールで呼出判別表が更新されることで、環境変化に対して適応的な動作を行うことができる。さらに、クラス階層があることで、互いの情報に一部非互換があっても縮退動作できるため、動作中に一部のモジュールを入れ替えるなど、より高度なシステムを構築することができる。なお、同様の仕組みはシミュレーションだけでなく、AGV (Automatic Guided Vehicle)など実システムでも活用できる可能性がある(Weyns & Georgeff, 2009)。

3.5 まとめ

本章では、Web アプリケーションを対象に静的対応と動的対応を統合したモデルを定義した。本モデルは静的対応としてサーバーおよびクライアントのフレームワークを共通化しており、開発者がサーバー、

クライアントの別なく統一的に理解をすることができる。また、動的対応として入出力オブジェクトのクラス階層を利用した呼び出し先変更と、学習・更新モジュールを利用した呼出判別表の動的更新をサポートしており、変化する状況に対応した動作をさせることが可能となる。

本モデルはフレームワークモデルであるため、ここから各プラットフォームに対応させたフレームワークを実装することで実際に動作させることができる。第 4 章では、本モデルが実装レベルで効果があることを示すために、実装レベルのフレームワークを事例として挙げ、性質について調査する。

第4章　モデルによるフレームワークの実装例

本章では、3 つの事例について、該当する静的対応、動的対応の性質を満たすかどうかを調べる。また、フレームワークの移行について実装したものを説明する。

4.1 サーバー型フレームワーク

サーバー型フレームワークは、サーバーサイドフレームワークとも呼ばれ、Web サーバーで動作し、Web ブラウザとのインタラクションを行う仕組みである。サーバー型フレームワークの一つの動作は Web ブラウザからリクエストを受け取ることによって開始し、ビジネスロジックなどの動作を行い、Web ブラウザに画面を送ることによって終了する。ここでは、Servlet と JSP を使った環境について実証を行なった。

4.1.1 課題

Web サーバーにおいて、フレームワークが解決すべき課題について挙げる。

課題 4.1-1: Web アプリケーションにおいては多数の画面を扱うが、表示・入力の内容は画面間で共通部分も多い。たとえば、ユーザ情報を扱う画面はログインでも利用者情報の修正でも扱う。そのため、画面を再利用できる仕組みが必要となる。

課題 4.1-2: 業務利用する Web アプリケーションは画面数も多く、規模が大きくなるため、チーム開発がしやすい構造にしておく必要がある。そのためには、関心の分離を行っておきたい。

4.1.2 設計・実装

本研究のフレームワークモデルをサーバー型フレームワークに当てはめると、Figure 4.1 のようになる。本フレームワーク実装はプラットフォームとして Servlet/JSP (Oracle, 2009; Oracle, 2017)の環境を利用して構築した。

フレームワークの動作はブラウザからのリクエストによって開始するが、その際クライアントからは入力が HTTP のリクエストパラメーターとして送られてくる。JSP の仕様に従い入出力オブジェクトを Java の POJO (Plain Old Java Object)に実装し、key と value の組で表現される HTTP のリクエストパラメーターをマッピングする。HTTP リクエストパラメーターの特定のキー(verb)で示される値をコマンドとして使う。呼出判別表はファイルで定義し、入出力オブジェクトのクラス名およびコマンド文字列から、呼び出し先のクラス名、メソッド名を記述する方法とした。ロジックについてはハンドラと呼ぶ入出力オブジェクトとは別のオブジェクトに実装される。

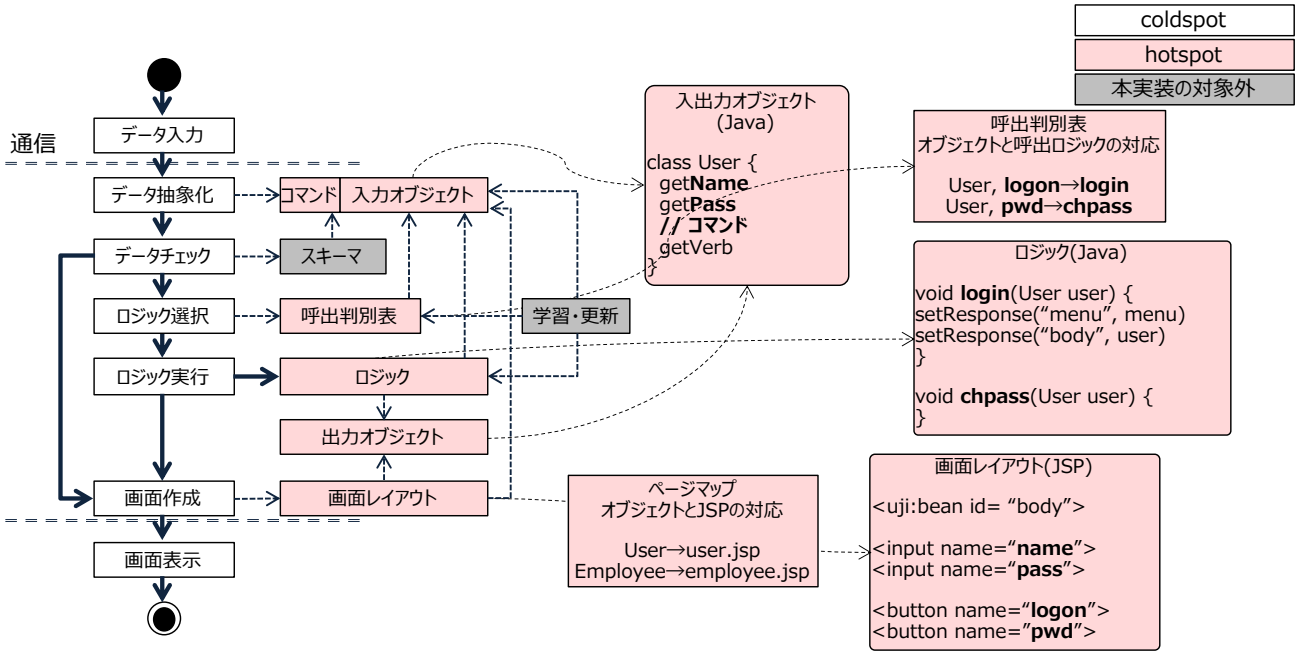

Figure 4.1 サーバー型フレームワーク

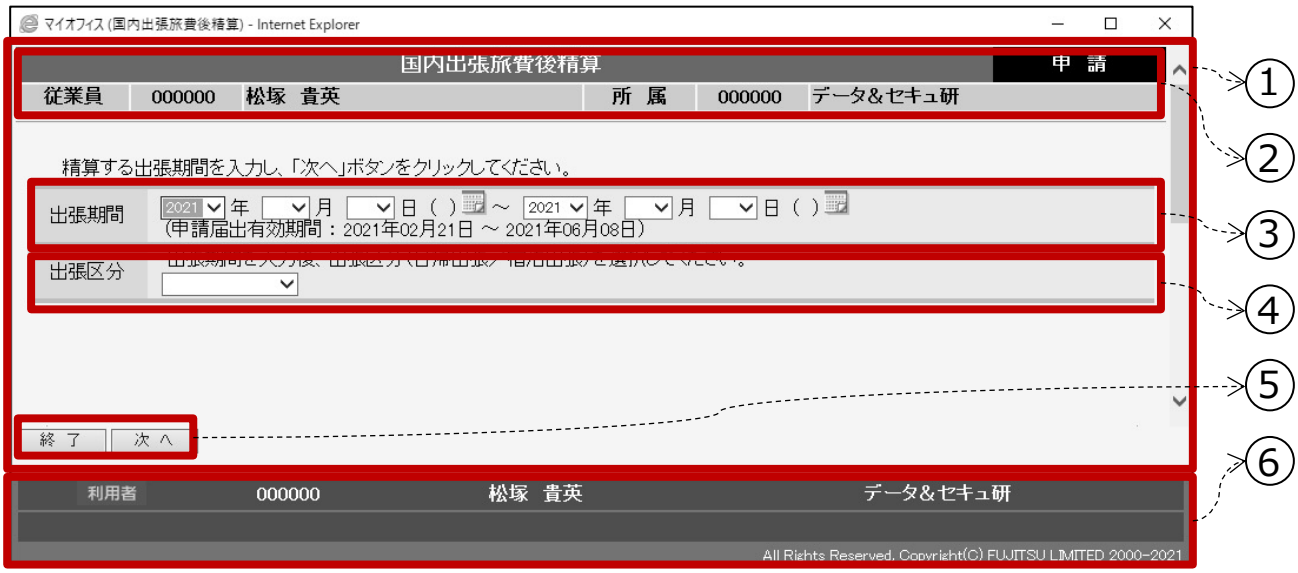

Figure 4.2 画面レイアウトの例

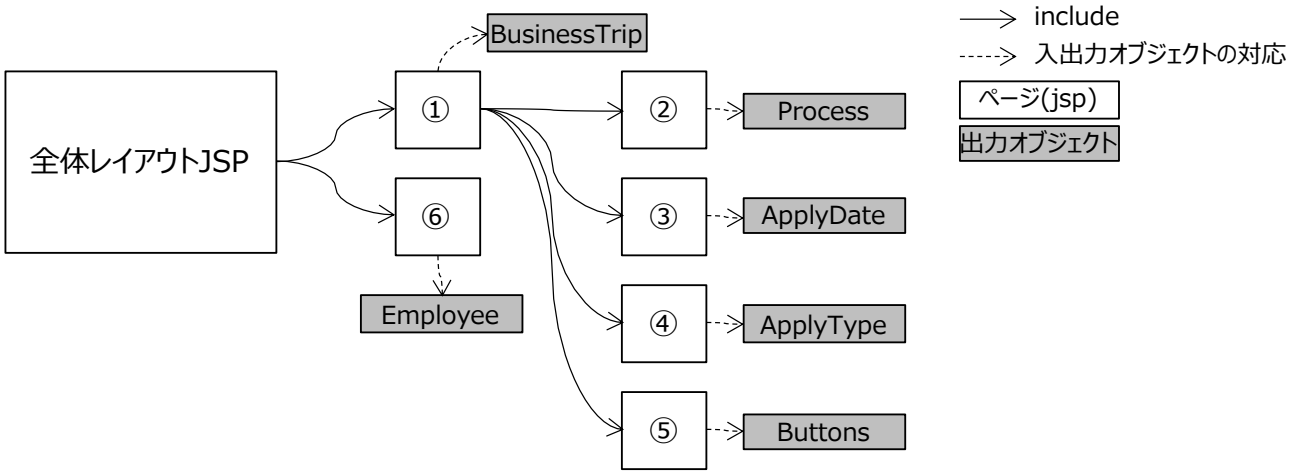

Figure 4.3 画面の include

課題 4.1-1 に挙げた画面の再利用については、画面を複数の部品の組み合わせで表現できるようにした。このためには、画面作成ステップにおいて部品の組み合わせができる必要がある。そのため、ロジックでは複数の出力オブジェクトに対する値を設定し、それらの出力オブジェクトごとに JSP で記述される画面レイアウトを選択できる構造とした。出力オブジェクトごとに画面レイアウトファイルを決定するために、ページマップという対応定義体を用いてファイルを決定するようにした。

実際の画面レイアウトの例を Figure 4.2、この画面を表示するためのページと出力オブジェクトの関係を Figure 4.3 に示す。ここでは、全体の画面レイアウトが国内出張旅費後精算業務を表示するページ①とログイン情報を表すページ⑥に分かれており、それぞれが BusinessTrip、Employee という出力オブジェクトの情報を表示する。さらに①は、業務名を表示するページ②、出張期間を表示入力するページ③、出張区分を表示するページ④、ボタンを表示するページ⑤に分かれており、それぞれのページが、Process、ApplyDate、ApplyType、Buttons という出力オブジェクトの情報を表示する。

これらのページと出力オブジェクトを組み合わせたレイアウトの仕組みにより、画面の部分を部品として再利用しつつ、アプリケーション全体として一貫性を持った画面構造を取ることができる。

4.1.3 考察

本フレームワーク実装を利用し、サンプルにおける計測を行った(Table 4.1)。計測条件は以下の通りである。

- 対象アプリケーションは総務系 1 業務。画面周りの実装とし、データベース入出力は含まない
- 開発者は業務系エンジニア
- フレームワークなし版は Servlet を使用
- 作成したソースコードのコメントを除いた行数を計測

ここでは、記述量についてはロジック部分が 84%減少した。代わりに定義体が増加しているが、これらはツールを使うことによって生成することができる。なお、ここでの定義体は呼出判別表、ページマップ、JSP、入出力オブジェクトのクラス定義である。

Table 4.1 サンプルにおける実装量の計測(サーバー)

フレームワーク	なし(行)	あり(行)	削減率
ロジック	1,697	264	84%
定義体	406	1,707	−
全体	2,103	1,971	13%

Table 4.2 サーバー型フレームワークにおける各 hotspot の実装

	画面表現	マッピング	ロジック
業務固有層	業務ごとの画面レイアウト	呼出判別表 ページマップ	業務ごとの処理
アプリケーション共通層	全体画面レイアウト 共通画面部品	呼出判別表 ページマップ	データベースの管理など

課題 4.1-2 で挙げた関心の分離については、本フレームワークを使う場合の hotspot は大きく画面表現に実装されるものとロジックに実装されるもの、そして両者をマッピングするものに分かれる。また、アプリケーション共通層と業務固有層でも分かれる(Table 4.2)。画面表現に関係する関心としては、アプリケーション共通層において、全体画面レイアウトとアプリケーション共通で使われる画面部品が実装され、業務固有層では業務ごとの画面レイアウトが実装される。フレームワークに関係する関心としては、3.4.1 で述べた通りファイルを分けて呼出判別表とページマップが実装される。

ロジックに関係する関心としては、アプリケーション共通層でデータベース管理や共通ライブラリなどを実装する一方で、業務固有層では業務ごとの処理を記述する。以上の記述はモジュールとして別に記述するようになっており、関心の分離を達成している。これらの分離は、大規模アプリケーションにおいては担当者を分けることができるようになるため、重要である。

再利用性については、ロジックをデータから分離することで、依存の連鎖をなくすことができた。これにより前節で述べたように、画面を JSP と入出力オブジェクトの組として定義し、アプリケーションの異なる複数の部分に使うことができるようになった。

4.1.4 まとめ

ビジネスアプリケーションを Web アプリケーションとして開発する際には、大量のデータと画面を部品化するなど効率的に実装するための枠組みが必要となる。本実証では、モデルに従ってフレームワークを構成したことで、画面を表す JSP とロジックを実装するハンドラを疎に連携させることができるようになった。その結果、画面の部品化が可能となった。

本フレームワークを Servlet/JSP 環境上に実装し、アプリケーション実装による評価を行なった。Servlet による業務アプリケーションからの書き換えにおいては、ロジックがより整理され簡潔になるとともに、総記述量では 10%程度、ロジックとしてメンテナンスする部分は 80%以上の削減が可能であった。性能については、計測を行った結果フレームワーク導入によるオーバーヘッドはほとんどないことが分かった。これらの結果を踏まえ、本フレームワークは富士通株式会社において製品化されるとともに人事勤

労・総務のサービスを行うマイオフィスシステムに採用され、最大 10 万人に利用されている。

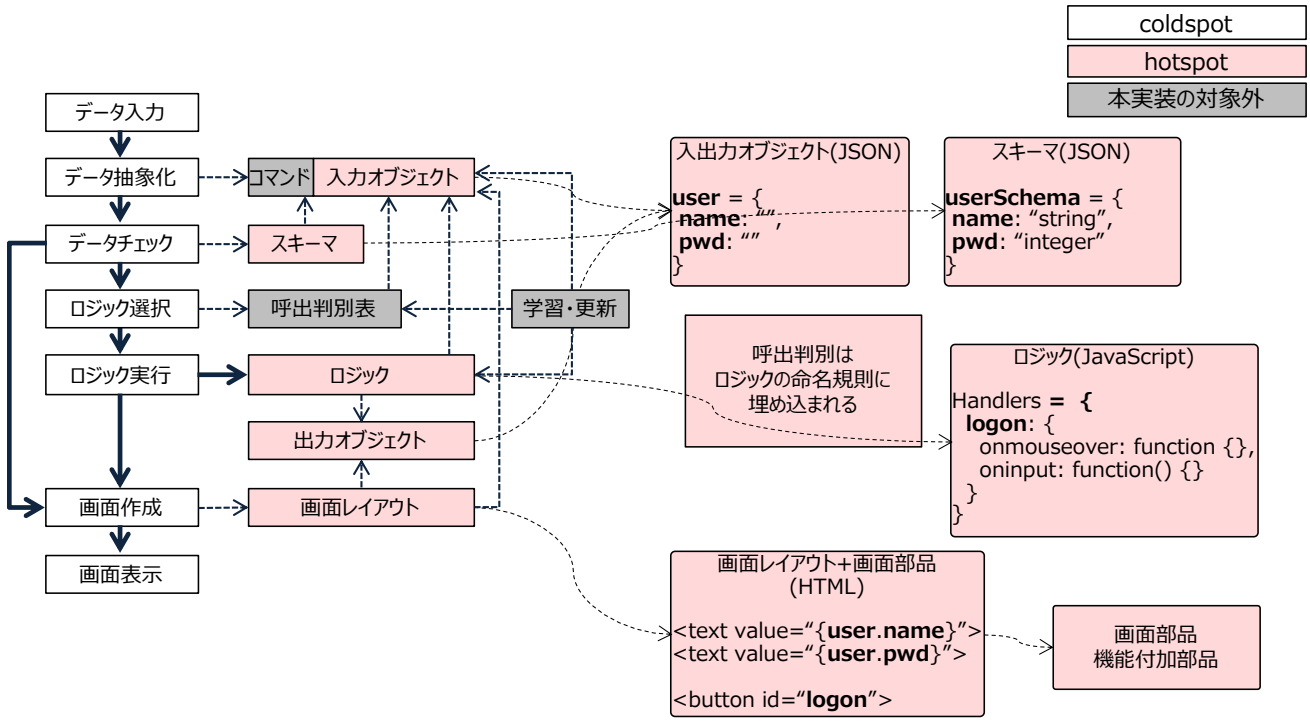

Figure 4.4 クライアント型フレームワーク

4.2 クライアント型フレームワーク

クライアント型フレームワークはクライアントサイドフレームワークとも呼ばれ、主にブラウザにおける画面の表現力とインタラクションを改善する。ここでは、HTML と JavaScript を使った Ajax (MDN, 2021b)と言われる環境における実証を行なった。

4.2.1 課題

クライアントにおいて、フレームワークが解決すべき課題を挙げる。

課題 4.2-1: クライアント型のフレームワークにおいては、ブラウザの持つ機能をフルに活かしたアプリケーションを開発するために、画面動作の高機能化や、そのきめ細かなコントロールが必要となる。画面構築のベースとなる HTML の機能は基本機能の提供にとどまるため、JavaScript を組み合わせて実装される。この画面の種類が多くなることから、部品化や効率的に作ることができる仕組みが必要となる。

課題 4.2-2: JavaScript が使えることにより、アプリケーションロジックの一部、特にユーザインタラクションに関係する部分をクライアントで実装することで、反応性の高いアプリケーションとしたい。このとき、画面構築に使う JavaScript とは役割が異なるので、関心の分離ができるような構造にしたい。

課題 4.2-3: 利用者が使うブラウザを開発者が選ぶことはできないため、アプリケーションはクロスブラウザで動作することが求められる。ブラウザ間の非互換性についてはフレームワークが吸収することが望ましい。

4.2.2 設計・実装

本研究のフレームワークモデルをクライアント型フレームワークに当てはめたものを Figure 4.4 に示す。ここで入出力オブジェクトは、実行言語が JavaScript であることから、JSON (JavaScript Object Notation)を採用した。画面部品と入出力オブジェクトは、画面部品から入出力オブジェクトへのパス(user お部ジェクトの name 要素を user.name などと表す)を指定することで関連づける。また、スキーマオブジェクトへのパスも指摘できるようにすることにより、データチェックも可能とした。

動作上サーバー型と異なるのは、入出力オブジェクトへのデータ反映に通信を必要としないことである。そのため、ソースとなる画面部品の状態が変更されたら、リアルタイムで入力オブジェクトにデータが反映される仕組みとした。一方で、出力オブジェクトから画面部品への反映は少し異なり、ロジックから明示的に反映する指示があった時に画面にデータが反映される仕組みとした。この理由は 2 つある。一つは、JSON の値を変更することをトリガとして画面にデータを反映させる仕組みが言語仕様として存在しないこと、もう一つは、処理の途中での一時的な値の変更を画面に反映させないためである。ロジック実行中は様々な処理を行うが、その途中で JSON の値を変更することもあり得る。これにより常に画面の更新をしてしまうと、計算途中の値が画面表示されてしまう。そのため、ロジックからの出力オブジェクト更新の画面反映は明示的に行うこととした。

ロジックは、本実証では画面部品の ID とイベント名から選択され、これが関数名として直接ロジック内に記述される。具体的には、たとえば logon という部品(ボタンを想定)の click および mouseover というイベントが発生した時に実行される関数の名前は、以下のように記述される。

```
Handlers = {
  logon: {
    // logon の click イベントが発生
    click: function() { ... }
  }
  // logon の mouseover イベントが発生、ショートカット記述法
  logon_mouseover: function() { ... }
}
```

画面部品からロジックを直接指定している理由は、クライアント型では画面部品のインスタンスを明示的に記述することができるためである。サーバー型では、画面を表す JSP において直接ロジックを指定してしまうと、再利用されるべき画面からロジックへの依存が生まれ、Figure 3.3 で示した依存の連鎖が発生してしまう。このため呼出判別表を使って両者を分離した。一方、クライアント型では画面部品をクラスとして定義するが、実際に画面を表す HTML には個別に HTML タグとして記述され、それぞれが異なるインスタンスになる。そのため、ロジックを直接指定しても依存関係の連鎖は発生しない。

コマンドにおいても、サーバー型と異なり、ブラウザで発生したイベントはコマンドに抽象化する必要がないため、イベント名をそのまま使うこととしている。これらの対応はアプリケーション開発者の実装の簡素化を目的として行なっている。技術的にはフレームワークモデルに即してコマンドおよび判別表を用意することも容易に可能である。

なお、課題 4.2-3 に示したクロスブラウザ対応については、ブラウザ非依存クラスが実体化される時に依存クラスが非依存クラスを上書きする仕組みをフレームワーク内に開発した。これにより、実行時の

ブラウザ判断のオーバーヘッドなしでブラウザ固有の挙動を実現できるようになった。

4.2.3 画面部品

課題 4.2-1 で挙げたように、クライアントでは、画面動作のきめ細かいコントロールのため、HTML と JavaScript を組み合わせて画面部品を作り、それを再利用単位としたい。これらの部品には様々な機能があるため、バリエーションが増大する。そのため、画面部品をウィジェットと機能付加部品に分割した。ウィジェットはボタンやテキストフィールドなどの通常の HTML でも用いられるコントロールを含む、画面に表示される部品である。

機能付加部品はウィジェットに機能を追加する部品で、例えば入力を整数のみに制限する機能、入力の自動補完をする機能、フォーカス移動を管理する機能などを実装する。機能付加部品がウィジェットと異なるのは、それ単体で画面に表示される部品ではないことで、必ず 1 個以上のウィジェットに関連付けられる。機能付加部品の機構は、機能を付加する対象となる画面部品の種類を固定していないため、同じ機能を複数種の部品に付加することができる。例えば、「入力を数値に制限する」という部品が欲しい場合、従来方法ではテキストフィールド、テキストエリア、リッチテキスト入力部品など、ベースとなる部品ごとにクラスの拡張が必要となるが、本方式ではその必要がなく、「入力を数値に制限する機能付加部品」をそれぞれの画面部品に付加させればよい、この機能付加部品は「キー入力」のイベントを検知して、そのイベントで入力されたキーが数値ではなかった場合、イベントを無効化する(入力がなかったことにする)処理を行う。機能付加部品も画面部品の一種であるため、入出力オブジェクトとのデータのやり取りが可能である。たとえば、フォーカスの順番を状況によって変更したい場合は、ロジックで状況を判断した上で、入出力オブジェクトにフォーカス順番を設定し、フォーカス移動を管理する部品がその入出力オブジェクトの情報を利用すれば良い。

4.2.4 考察

次の計測条件でサンプルにおける計測を行なった。

- 対象アプリケーションは Google Maps と JavaServer Faces (Oracle, n.d.)に付属している Cardemo サンプルをマッシュアップして、動的な価格変更や自動補完機能などを盛り込んだもの。
- 開発者は本研究を行なったグループのメンバー
- フレームワークなし版、あり版ともサーバーは同一
- 作成したソースコードのコメントを除いた行数を計測

記述量については、ロジック部分が 62%減少した(Table 4.3)。サーバー型フレームワーク同様、ロジック以外の部分については、ツールを用いて生成することが可能である。

課題 4.2-2 で挙げた関心の分離については、それぞれ Table 4.4 のように画面表現およびロジックに対し、アプリケーション共通層と業務固有層の軸がある。画面表現に関係する関心としては、業務固有層でそれぞれの画面に関する実装が提供される。画面実装に使う画面部品はテーブルやツリーなど基本的なものはフレームワークで提供されており、これらで不十分な場合のみアプリケーション共通層においてウィジェットおよび機能付加部品を実装する。ロジックに関係する関心としては、アプリケーション共通層で共通ライブラリなどを実装する一方で、業務固有層において業務ごとの処理を提供する。

再利用性については、画面部品単位で再利用が可能である。特に、ウィジェットと機能付加部品の組み合わせで部品を増やす仕組みにより、似たような機能を持つ部品を増やす際に業務固有層のみで容易に作ることができるようになった。

Table 4.3 サンプルにおける実装量の計測(クライアント)

フレームワーク	なし(行)	あり(行)	削減率
ロジック	733	276	62%
メッセージリソース	446	446	0%
入出力オブジェクト	0	55	N/A
HTML	260	185	29%
全体	1,439	962	33%

Table 4.4 クライアント型フレームワークにおける各 hotspot の実装

	画面表現	ロジック
業務固有層	業務ごとの画面	業務ごとの処理
アプリケーション共通層	ウィジェット 機能付加部品	共通ライブラリなど

4.2.5 まとめ

クライアント環境では、HTML と JavaScript を使って効率的にアプリケーションを構築する仕組みが必要である。本実証では、フレームワークモデルに従って入出力オブジェクトとロジックを分離した。また、部品をウィジェットと機能付加部品に分割し、様々な機能を持つ画面部品を容易に拡張できる枠組みを開発した。

本フレームワーク上でのアプリケーションでの評価を行なった。総記述量で 33%、JavaScript によるロジックとしてメンテナンスが必要な部分は 60%以上の削減を確認した。これらの結果を踏まえ、本フレームワークは富士通株式会社において製品化、販売されるという成果を挙げた。

4.3 適応型フレームワーク

動的対応を実証するため、人材交流イベントを支援するサービスを事例として効果を確認した。

4.3.1 課題

端末性能の向上によって、端末側で様々な処理が可能となっている。このことは、従来のアプリケーションにおいてサーバーで行われていた処理を、クライアント側でも実行できることを意味する(Hales, 2012)。これにより、サーバーの負荷をオフロードできたり、利用者の入力に対するレスポンスを高めたりすることができる一方で、パフォーマンスの観点から見ると、必ずしもすべてクライアントで処理を行えばいいというわけではない(Walker & Chapra, 2014)。一方でサーバーにおいても、システムが実行されている環境は、現在接続しているクライアントデバイスの数や、クラウドにおける他のサービス実行による負荷変動など、多数の要素で刻一刻と変化する。この状況の中で「これから実行する処理」をサー

バーおよびクライアントのどちらで実行するのが性能として最適かを大きなオーバーヘッドを追加することなく判断しなければならない。さらに、状況によっては、急激な負荷変動や環境変化により、ある機能をサーバー、クライアントのいずれで実行しようとも意味のある時間で処理が終わらないということがありえる。このような例外的な状況に対しては、安全サイドに倒した縮退機能を用意しておき、切り替える仕組みも必要である。

ここで事例としたアプリケーション Buddyup!は、人材交流イベントにおいて互いに目的とする人材を見つけるために使われるアプリケーションである(富士通, 2020)。Buddyup!に参加者が自己紹介文を登録すると、文が形態素解析され,「スキル」や「興味」に関する単語(以降タグと記載する)に分解される。参加者に自分と共通のタグを持つ他の参加者を推薦することで、同じ興味を持つ人を発見しやすくなる。これにより、イベントで質問を繰り返して相手の情報を引き出す必要がなくなり、人材交流を円滑に進めることができる。人材交流イベントでは、それまで互いに見ず知らずの人が会い、その場で入力された自己紹介をもとに人同士のマッチングが行われる。自己紹介はイベント中にも変更されるため、参加者推薦のためのタグのマッチング処理はリアルタイムで行い、結果は即座に提供する必要がある。マッチング処理に要する時間は参加者のタグ数に依存するため、Buddyup!では、計算量の異なる 2 種類のマッチングアルゴリズムを実装している。

- nx1 マッチング
  ユーザが持つタグ 1 個ごとに、同じタグを持つ参加者 n 人(以降マッチング人数 n と記載する)を抽出するアルゴリズムである。ユーザが複数のタグを持つ場合、抽出されたマッチング人数 n が多い順にタグがソートされて推薦される。計算量は$O(n)$となる。
- nxm マッチング
  ユーザが持つタグのうち m 個に対して、同じタグを共通に持つ参加者 n 人(マッチング人数 n)を抽出するアルゴリズムである。計算量削減のため、あらかじめ nx1 マッチングで共通のタグを持つ参加者を抽出したうえで、複数のタグを共通で持つ参加者を探索する。共通のタグ数 m が多いグループから順に表示し、タグ数 m が同じ場合はグループ内の人数が多いグループから順に推薦する。計算量は$O(n^4)$となる。

複数の共通タグを持つ参加者を提示する nxm マッチングの方がより精度の高いマッチングを行うため望ましいが、計算量が大きいため、期待性能、すなわち処理時間の見積もりによって適切に処理を切り替える必要がある。

4.3.2 設計

本アプリケーションの要件をフレームワークモデルに当てはめたものを Figure 4.5 に示す。入出力オブジェクトはマッチング人数およびタグ情報を入力する。コマンドは常に同じのため省略しており、指定していない。また、本事例においては、スキーマは使用しなかった。

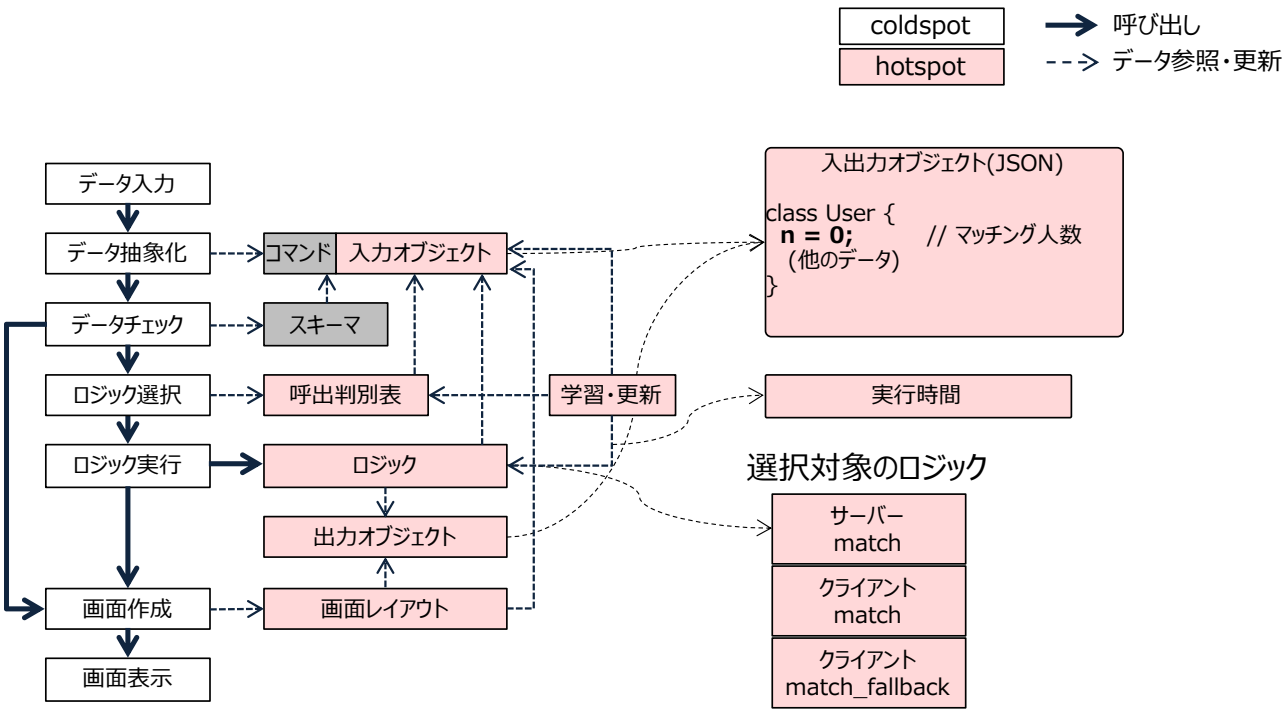

Figure 4.5 適応型フレームワーク

Table 4.5 Buddyup!の呼出判別表

行	入出力オブジェクト	条件(n はマッチング人数)	ロジック名
1	User	0 ≦ n < 9	match_client
2	User	10 ≦ n < 20	match_server
3	User	20 ≦ n < 30	null
4	...	...	...
5	Object	*	match_fallback

Table 4.6 関数n_{fid}の性能特性値マップ

量子空間番号	値(n はマッチング人数)	移動平均値	実行時間
1	0 ≦ n < 9	$\mu(n_{fid}, 1, k)$	$x(n_{fid}, 1, k)$
2	10 ≦ n < 20	$\mu(n_{fid}, 2, k)$	$x(n_{fid}, 2, k)$
3	20 ≦ n < 30	$\mu(n_{fid}, 3, k)$	$x(n_{fid}, 3, k)$
...			

実際の呼出判別表を Table 4.5 に示す。呼出判別表で選択対象となるロジックはクライアントの nxm マッチング、サーバーの nxm マッチング、nx1 マッチングの 3 種類である。これらは Table 4.5 においてそれぞれ match_client、match_server、match_fallback のロジック名で示される。nxm マッチングはサーバー、クライアントに関わらず n の数が大きくなると指数関数的に実行時間が増大するため、参加者数が多くなった時に実行時間が課題になってしまうことがある。このような場合に nx1 マッチングを縮退機能として用いる。ここでは、条件としてマッチング人数を使っており、選択されるロジックがマッチング人数で変わることが示されている。

4.3.3 呼出判別表の動的更新

それぞれの関数の期待性能は、サーバーやクライアントの負荷変動などで変化するため、どのロジックを呼び出すのが最適かは刻一刻と変化する。この動的な環境変化に対応するため、呼出判別表の動的更新アルゴリズムを次のように定義し、学習・更新モジュールに実装した。

各ロジックの性能履歴を把握するため、関数に識別番号n_{fid}を定義し、n_{fid}ごとに性能特性値マップを持つ(Table 4.6)。性能特性値マップはマッチング人数を量子空間に分け、量子空間ごとに実行時間の移動平均値$\mu(n_{fid}, j, k)$、最新の実行時間$x(n_{fid}, j, k)$を持つ。ここで、jは量子空間に付けた番号、kはサイクル数(整数 $0, 1, 2, 3, ...$)である。移動平均値は性能特性値の代表値として利用する。最新性能特性値は移動平均値の更新を行うために利用する。各ロジックの性能特性値マップをもとに、量子空間ごとに実行時間の移動平均値が最小となる関数$n_{fid,opt}$

$$n_{fid,opt} = arg\min_{n_{fid}} \mu(n_{fid}, j, 0)$$

を導出し、これを呼出判別表のロジック名とすることで呼出判別表を更新する。

ただし、ある量子空間において、各関数の実行時間がすべて、あらかじめ定めた閾値$\theta_1(k)$よりも大きくなってしまう場合がある。このときはその量子空間におけるロジックは null とする。これは、縮退関数以外のいかなるロジックでもユーザの要求する最低限の性能特性値が得られないと判定することを意味する。この場合、当該量子空間に対するロジックは選択できないことになり、判別表においては処理対象 Object に対する match_fallback が縮退ロジックとして選択される。

このあと、次の式を用いて移動平均値を更新する。ここで、wは過去の観測値をどこまで考慮して逐次の平均値や分散値を導出するかを調整する重みである。

$$\mu(n_{fid}, j, k) = \frac{1}{w+1}(w\mu(n_{fid}, j, k-1) + x(n_{fid}, j, k))$$

ここにあげた動作は、ロジックが実行された時に、その実行時間を使って性能特性値マップおよび呼出判別表を更新する。ただし、当該ロジックが長時間選択されないことによって実行時間が実性能を反映しなくなってしまうことを防ぐため、利用者からのリクエストとは無関係に、非同期に実行を行なって性能特性値マップおよび呼出判別表の更新を行う。

今回の性能特性値マップの実装では、マッチング人数を 10 ごとの量子空間に分けてロジックごとの性能値を管理した。そのため、呼出判別表でも条件部はマッチング人数を 10 ずつの固定の範囲とし、各範囲におけるロジック名を変更するよう動的更新を行なった。

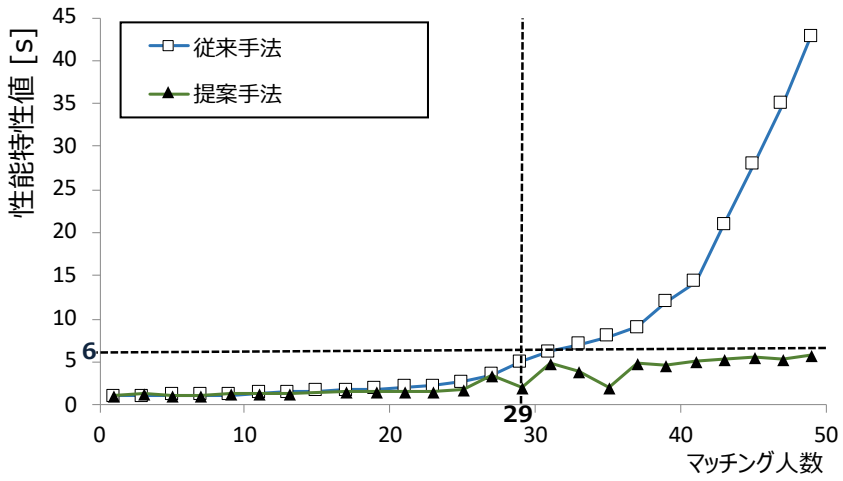

Figure 4.6 マッチング人数による応答時間の変化

4.3.4 考察

従来手法と、本モデルに基づき学習更新ロジックを追加した場合の応答時間の変化を Figure 4.6 に示す。比較対象としている従来手法は、本アプリケーションで当初実装されていた、適応的にロジック選択を更新することなく、固定的にクライアントにおいてマッチング処理を実行していたものである。これに対し、前述のようにサーバーおよびクライアントの双方でマッチング処理を用意し、呼出判別表を動的更新したものを提案手法とした。

本実証では、マッチング人数 29 人未満は既存手法と同様にクライアント端末で演算が実行される。29 人以上から既存手法よりも高速に応答している。これは、学習・更新モジュールによって呼出判別表が更新されたことにより、サーバーのマッチング関数にロジックが切り替わり応答時間が改善したためである。また、本実証ではマッチング人数が 50 人に至っても 6 秒以内を維持しており、イベントが大規模化してもユーザーメリットを損なうことなく、 nxm マッチング機能を提供可能である。以上のことから、本提案手法により応答時間を改善することができた。

本事例で扱った対象である Buddyup!は、イベント型人材交流を支援する Web サービスである。イベント型人材交流では、事前に相互に知らない人同士が限られた時間で互いにコミュニケーションを繰り返し、相手の情報を引き出して目的とする人材を見つけ出す必要がある。したがって、参加者の要求に応じて、即座にマッチングの出力結果を提供する必要があり、そのマッチングの出力結果とその応答時間は参加者の体験価値に大きな影響を与える。出力の応答時間はクライアント端末のタスク使用状況、通信の混雑状況、サーバーの負荷状況、参加者数の変化により大きく変化する不確実性があり、これを解決するためにロジック選択に対して適応的な自動更新を行った。

4.3.5 まとめ

本実証ではイベント型人材交流サービスを事例とし、本フレームワークモデルにおける呼出判別表の動的更新という特徴を使い、性能の変化によるロジックの自動変更を実現した。これにより、マッチング人数やサーバー、クライアントの負荷の変動などがある環境において、呼出判別表に基づき最適な応答時間で実行するロジックを選択することを可能とした。

第5章　フレームワークの移行

フレームワークは実装技術であるため、いったんあるフレームワークを採用してアプリケーションを開発すると、アプリケーション内の様々なコンポーネントがフレームワークに依存した形になる。このとき、他のフレームワークに移行するのは困難である。この点は、他のフレームワークから本モデルをベースとしたフレームワークに移行する際も同様であり、フレームワーク採用の障壁になり得る。この問題を解決するため、フレームワーク間の移行を可能とする技術を開発した。

5.1 フレームワークの仕様とアプリケーションの仕様

2.1.1 節において、開発ライフサイクルには要求、仕様、コードのレベルがあることを述べた。このうち、本研究で提案しているフレームワークモデルはフレームワークの仕様を示しており、これをコードレベルで実装することで動作させることができる。

一方、アプリケーションの仕様を記述し、実装することでアプリケーションを動作させることができる。モデル駆動型開発(Model Driven Development, MDD)という考え方においては、ソフトウェアの提供機能をプラットフォームに非依存な形で記述することで、ソフトウェアの実装を半自動ないし全自動で生成するアプローチである。4.1 節に述べたフレームワークにおいても、MDD に基づいて Web アプリケーションを生成できるツールを提案している(Matsutsuka, 2005)。アプリケーション仕様とフレームワーク仕様の関係を Figure 5.1 に示す。

この MDD をさらに発展させると、仕様を介して Web アプリケーションを異なるフレームワーク間で移行することができる。そのためには、Web アプリケーションの仕様をフレームワークに非依存な形で表現する必要がある。ここでは、その仕様を表現するためのモデルを開発した。これを WebWare 意味モデルといい、本実証ではフレームワーク間において WebWare 意味モデルを介して Web アプリケーションを移行する方法を示す。

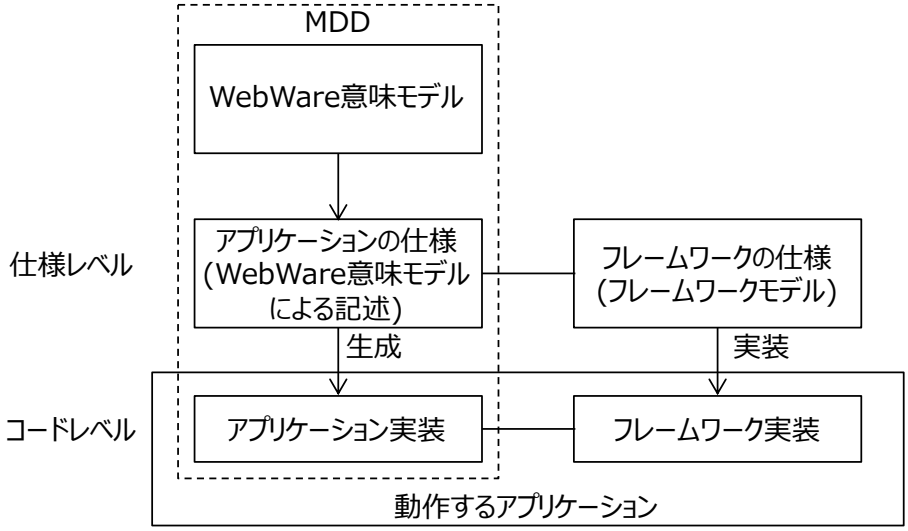

Figure 5.1 アプリケーションとフレームワーク

5.2 WebWare 意味モデル

ここでは、Web アプリケーションの仕様を記述するためのモデルである WebWare 意味モデルについて説明する。意味モデルを特定の実装技術に依存させないことで、様々な実装フレームワークに対応させることができる。WebWare 意味モデルの設計はコンポーネント、ポート、コネクタ(Shaw & Garlan, 1996)による方法を用いる。なお、これは UML (OMG, 2017)のアクティビティ図のサブセットにもなっている。

Page と Action

本意味モデルでは、Figure 5.2 に示すように、Page と Action という、それぞれ 0 個以上の入力ポートと 0 個以上の出力ポートを持つ 2 つのコンポーネントを定義する。Page は論理的なページを表す。Page はブラウザにおいて一つのフレームに表示されるものを一つとする。Action は論理的なサーバーのロジックである。WebWare 意味モデルでは、サーバーロジックを入出力で定義しており、具体的なコンポーネント(Servlet など)を規定しない。これは，フレームワークにより、サーバーコンポーネントと捉えるべき単位が異なるためである。

ポートは 4 種類ある。Page の入力ポート(accept)は、HTML における URL など、Page を表示するために必要な指定情報を表す。Page の出力ポート(target)は、Action または Page に対するリクエストを示す。Action の入力ポート(accept)は Action を起動するために必要な指定情報を表す。Action の出力ポート(response)は Action からのレスポンスを示す。それぞれのポートは、フレームワークごとに異なる属性のセットを持つ。これは、フレームワークごとに Page の表示や Action の起動などに必要な情報が異なるためである。たとえば、Struts (Apache, 2018)の場合，Action の Accept は Struts の場合は URI で表されるが、本研究におけるフレームワークモデルの場合は入出力データのクラスとコマンドで示される。

遷移

Page と Action の定義を使い、遷移を構築する手法を示す。遷移は Page または Action の出力ポートと、Page または Action の入力ポートをコネクタによって接続したものである。接続は出力ポートと入力ポートをネゴシエーションさせることによって導出する。ネゴシエーションの結果、以下の 4 種類のうちの 1

つの遷移が得られる。

1. Page から Page への直接の遷移
2. Page から Action へのリクエスト
3. Action から表示する Page の選択
4. Action から Action の呼び出し

ポートの属性はフレームワークによって異なるため、ネゴシエーションの詳細はフレームワークによって異なる。一例として、ページ間直接遷移および Struts フレームワークによるネゴシエーションを Figure 5.3 に示す。

本意味モデルでは、ウィンドウを複数開いたりするなど、画面部分ごとに複数の動作が並行に発生する状態をサポートする。Figure 5.4 に示す例では、親画面が子画面を開き、両者が平行に存在する様子を示す。ここでは簡単化のためポートの記述を省略している。

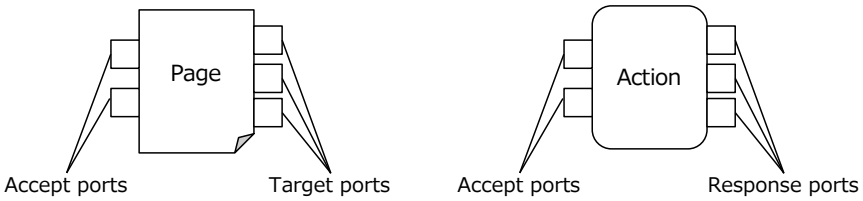

Figure 5.2 Page と Action

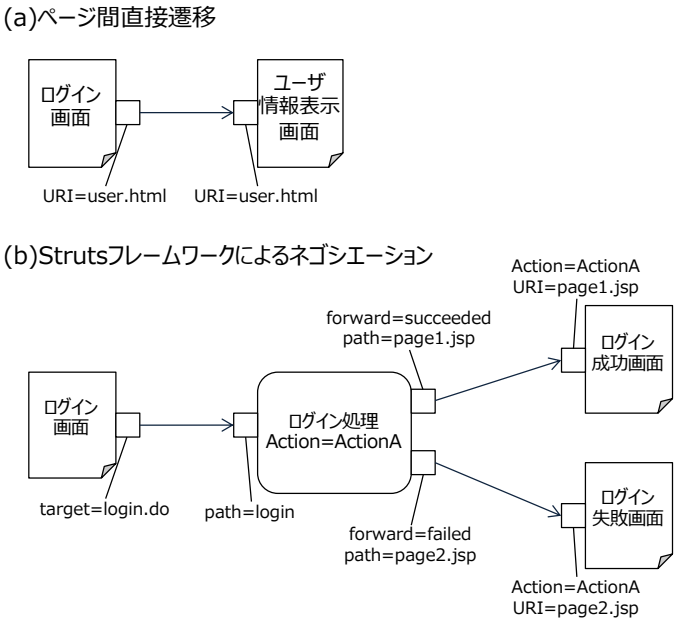

Figure 5.3 ネゴシエーションの例

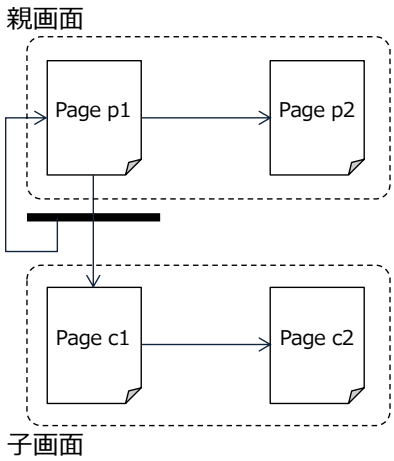

Figure 5.4 並行状態

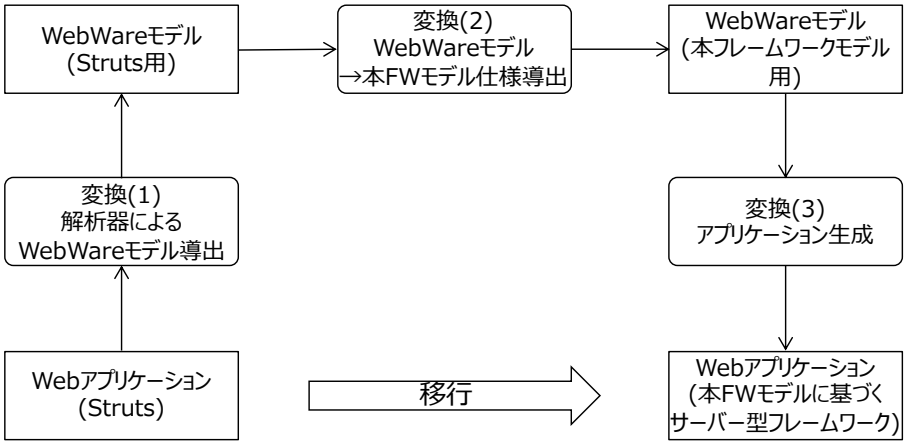

Figure 5.5 フレームワークの移行

5.3 フレームワーク間の移行

前節の意味モデルを使ってフレームワーク間の移行を行った。具体的には、サンプルとして構築した既存の Web アプリケーションからリバースエンジニアリングで WebWare 意味モデルを導出し、そのモデルから再びアプリケーションの生成を行う。この実証のため、3 個のページから構成されるサンプルアプリケーションを、Struts フレームワークを利用して作成した。

移行の流れを Figure 5.5 に示す。Struts アプリケーションからの WebWare モデルの導出はリバースエンジニアリングによる解析器を作成した(福安ほか, 2004; 桑原ほか, 2008)。ここで導出されたモデルはネゴシエーションが Struts 用になっているため、これを本フレームワークモデルで動作するネゴシエーションに変換する。さらに、本研究におけるフレームワークモデルに準拠したアプリケーションを生成し、移行後のアプリケーションの動作が移行前と同等であることを確認した。なお、本移行で対象とした移行後のフレームワークは 4.1 節で示したサーバー型フレームワークである。

5.4 まとめ

本実証では、Web アプリケーションの意味を記述するためのモデルである WebWare 意味モデルを構築した。この意味モデルの特徴は、特定の実装技術に依存しないことと、フレームワーク間移行を行うことができることである。これに基づき、実際に既存のアプリケーションから意味モデルを導出し、モデル変換および自動生成を行うことにより、本論文で提案するサーバー型フレームワークに移行できることを示した。

第6章　結論

本章では、研究の成果をまとめ、今後の課題について述べる。

6.1 まとめ

本研究では、静的変化と動的変化を統合する Web アプリケーションのためのフレームワークモデルを定義し、有用性について考察した。

第 1 章では、背景を述べ、Web アプリケーションが抱える問題と、それに対する課題を議論したうえで、本研究の目的を述べた。ソフトウェアが環境や要件の変化に対応するための手段は、静的対応と動的対応という 2 種類に大別される。ここで、課題を 2 つにまとめた。

1. 静的対応において、クライアントとサーバーという 2 種類の問題領域が存在するため、それぞれ個別にフレームワークが開発されている。すると開発時には開発者がそれぞれに対して熟達しなければならず、学習コストや保守コストが高くなることが問題であるため、サーバーとクライアントで動作が異なる Web フレームワークを統合する。
2. 動的対応において、クライアントの変化に動的に対応するためのメカニズムとして自己適応技術が知られているが、開発者がフレームワークとは全く別の技術として取り扱わなければいけないという問題がある。このことから、Web フレームワークに動的対応を追加する。

第 2 章では、関連研究について検討し、本研究の位置づけを述べた。静的対応においては、Web アプリケーションにおいてクライアントの環境変化を動作の変更で吸収することから、仕様及びコードについて検討対象とした。動的対応においては自己適応技術について検討し、管理対象サブシステム、管理サブシステムから構成されること、変更のメカニズムとして構造変更を採用することを述べた。本研究では、静的対応と動的対応の両方に対応するために、Web アプリケーションへの自己適応技術の統合を行うこととした。

第 3 章では、フレームワークモデルの提案を行った。静的対応において、サーバーおよびクライアントで全く同一実装の Web フレームワークを動作させるのは非現実的なことから、フレームワークを一段抽象的な概念でモデル化し、同じ概念で実装できるようにしたものをフレームワークモデルと定義した。サーバー型、クライアント型のフレームワークを 7 段階のステップに整理し、コマンド、入出力オブジェクト、呼出判別表で抽象化することによりフレームワークモデルを作成した。動的対応を実現するため、入出力オブジェクトにクラス階層を導入したことと、呼出判別表の学習・更新を可能としたことにより、クライアントの変化に追従し続けることができるようにした。変化が想定以上だった場合に対象となるロジックを入出力オブジェクトのスーパークラスに対応するロジックに縮退させることにより、広い範囲の変化に対応することができるようにした。次に、本モデルに対する以下の考察を行なった。

1. コード量、関心の分離、再利用性という観点に関して効果があることを示した。
2. 抽象レベルでフレームワークを定義することにより、技術変化により異なる言語やプラットフォームの上でアプリケーションを動作させなければいけないという要求が発生した場合にも、

自動変換または半自動変換でアプリケーションを移植することが可能になることを示した。

3. クラス階層の利用の一例として、アプリケーションが更新された時に、多様なクライアントが更新に即座に追従できないことがあるが、このような場合も入出力をスーパークラスに縮退させることでサーバーが対応することができることを示した。
4. 本フレームワークモデルで導入している呼出判別表は、プログラミング言語のメソッド呼び出しに相当するが、ネットワークを隔てたノード間での呼び出しに対応するため、条件指定とオーバーロードという二点において拡張されていることを示した。
5. 動的対応のレベルを自己適応技術に照らし合わせて議論した。本モデルにおいては、Foreseen, Foreseeable という 2 種類の変化について対応することができる。
6. 本モデルの Web アプリケーション以外への適用可能性を検討した。フレームワークのコア部分はリアクティブな動作メカニズムであることから、Web アプリケーション以外のユーザインタフェースを実装するフレームワークにも適応可能であるほか、マルチエージェントシステムへの応用に可能性がある。

第 4 章では、フレームワークモデルの実証としてサーバー型フレームワーク、クライアント型フレームワーク、適用型フレームワークの 3 つの事例におけるフレームワークの実装を行い、考察した。

- サーバー型フレームワークにおいては、Servlet/JSP 環境を利用した実装フレームワークを本フレームワークモデルに従って実装することで、画面の部品化、再利用ができるようになった。また、ロジックの実装量が大幅に削減されるとともに関心の分離を達成し、開発効率及び保守性を上げることができた。
- クライアント型フレームワークにおいては、HTML/JavaScript 環境における実装フレームワー示した。画面部品を組み合わせで増やすことができるようになったほか、計測においてロジックの実装量が大幅に削減される効果を確認した。
- 適応型フレームワークにおいては、試作したフレームワークにおいて、人材交流イベントを支援するサービスを事例として効果を確認した。呼出判別表を動的に更新する仕組みを導入することで、人材のマッチングを行うロジックをクライアントで行うのかサーバーで行うのかを動的に切り替えることができるようになり、参加者人数に応じて最適な応答時間で実行するロジックを選択することが可能となった。また、呼出判別表のクラス階層により、クライアントからの要求人数に対するマッチングロジックの予想所要時間があまりに長い場合に、簡易的なマッチングロジックへ縮退する機構を実現した。

第 5 章では、モデル駆動型開発を用いたアプリケーションのフレームワーク間の移行技術を開発し実証した。移行は Web アプリケーションの仕様を介して行うため、仕様を表現するための WebWare 意味モデルを開発した。実証は既存のアプリケーションから WebWare 意味モデルを導出し、本研究のフレームワークモデルに準拠したアプリケーションを自動生成する手法による。移行前後でアプリケーションの動作が同等になることを確認した。

本フレームワークモデルを利用することの利点として、以下が挙げられる。

1. サーバーとクライアントで同じフレームワークモデルを利用していることにより、学習コストが低減できること。また、保守性を高めることができること。
2. 動的対応をフレームワークに統合していることにより、環境の変化に対応した Web アプリケーションを容易に実現できること。特に、事前に予期しない変化がクライアントに発生したときも、スーパークラスで対応することでより抽象度の高いレベルでの対応が可能となること。

6.2 今後の課題

今後の課題として、以下の 2 点が挙げられる。

一点目は、要求が変更されたときに、動的に対応することである。今回動的に対応したのは環境の変化であり、これは

$$S_1,\ D_1 \vDash R$$

であるときに、D_1 に対応する S_1 を求めることに相当する。要件の変化は

$$S_2,\ D \vDash R_2$$

のときに、R_2 に対応する S_2 を導出することである。本研究では動的対応と静的対応を統合しているため、要件の変化については静的対応で解決することは可能である。また、一般に要件の変化は予測できないものが多いため、全てを動的に対応することは不可能である。しかし、Lamsweerde (2009)をはじめとして要件の変化の範囲を形式的に定義する研究もあり、形式的に記述できる範囲において要件変化を扱うことができれば即応的に対応することが可能となる。

二点目は、ロジックの仕様を形式的に記述して、要求や環境の変化に対応させることである。これは、

$$S,\ \mathrm{D} \vDash R$$

の各要素を形式記述して、D および R の変化があった場合の S の対応を導出する方法である。これが可能になると、D や R にどのように S が対応するのか、その対応関係を正確に導出することが可能になる。一般にソフトウェアの形式記述は記述に相応の工数がかかること、外部入出力があるシステムではそのモデル化が必要なことなどの理由から、Web アプリケーションで効果的な形式検証を行うのは難しい。しかし、セキュリティなどクリティカルな部分に限って対応するなど、アプローチによっては効果的に導入できる可能性がある。

謝辞

本論文は、私のこれまでの研究開発の集大成としてまとめることができました。このような形で研究をまとめる機会を頂けたことは、社会人研究者として無上の幸せです。

本研究を遂行するにあたり、多くの方々から、多大なご指導、ご支援をいただきました。

秋田県立大学大学院 総合システム科学技術研究科 堂坂浩二教授、草刈良至准教授、静岡大学大学院総合科学技術研究科 西垣正勝教授に深く感謝します。堂坂教授には研究のみならず、普段論文執筆に慣れていない私に対し論文の構成や論理展開から表現に至るまで丁寧に見ていただき、おかげで本論文をまとめることができました。西垣教授、草刈准教授には、研究のディスカッションにおいて数々の適切な御助言を賜り、見落としていた数々の観点を得ることができました。心から感謝いたします。

普段から研究指導を頂いた秋田県立大学 総合システム科学技術研究科 飯田一朗教授に深く感謝いたします。理解のなかなか進まない私に辛抱強く指導いただきました。厳しくも暖かいご助言があったおかげで、物事を複数の角度から深く洞察し、本質を見極めることを学びました。

最後になりましたが、さまざまなご議論をいただいた飯田研究室の皆様、ご協力をいただいた職場の皆様、日頃から心の支えとなり励ましてくれた家族の皆様に心から感謝いたします。

参考文献

1. Angular Team. (2021). Angular - The modern web developer's platform. Retrieved June, 2021. Angular: https://angular.io
2. Apache. (2018). Welcome to the Apache Struts project. Retrieved June, 2021. Apache Struts: https://struts.apache.org/
3. Apple, (2021). SensorKit. Retrieved June, 2021. Apple Developer Documentation: https://developer.apple.com/documentation/sensorkit
4. Beugnard, A., & Sadou, S. (2007). Method overloading and overriding cause distribution transparency and encapsulation flaws. *Journal of Object Technology*, *6*(2), 31-45.
5. Beydeda, S., Book, M, & Gruhn, V. (Eds.) (2005). *Model-driven software development.* Springer, Berlin, Heidelberg.
6. Braberman, V., D'Ippolito, N., Kramer, J., Sykes, D., & Uchitel, S. (2015, August). Morph: A reference architecture for conafiguration and behaviour self-adaptation. In *International workshop on control theory for software engineering*, 9-16.
7. Brosch, P., Kappel, G., Langer, P., Seidl, M., Wieland, K., & Wimmer, M. (2012, June). An introduction to model versioning. In *International school on formal methods for the design of computer, communication and software systems*, 336-398. Springer, Berlin, Heidelberg.
8. Cheng, B., de Lemos, R., Giese, H., Inverardi, P., Magee, J., Andersson, J., Becker, B., Bencomo, N., Brun, Y., Cukic, B., Di Marzo Serugendo, G., Dustdar, S., Finkelstein, A., Gacek, C., Geihs, K., Grassi, V., Karsai, G., Kienle, H., Kramer, J., Litoiu, M., Malek, S., Mirandola, R., M¨uller, H., Park, S., Shaw, M., Tichy, M., Tivoli, M., Weyns, D., & J. Whittle. (2009). Software engineering for self-adaptive systems: A research roadmap. *Lecture notes in computer science*, 5525, 1-20, Springer, Berlin, Heidelberg.
9. Chumbley, R., Durand, J., Pilz, G., & Rutt, T. (2010). Basic profile version 2.0 final material. Retrieved July, 2021. Web Services Interoperability Organization. WS-I: http://ws-i.org/profiles/BasicProfile-2.0-2010-11-09.html
10. Django Software Foundation. (2021). Django: The Web framework for perfectionists with deadlines. Retrieved June, 2021. Django: https://www.djangoproject.com/
11. Durr, E., & van Katwijk, J. (1992). VDM++, a formal specification language for object-oriented designs. In *CompEuro Computer Systems and Software Engineering*, 214-219. IEEE.
12. Facebook Inc. (2021). React - A JavaScript library for building user interfaces. Retrieved June, 2021. reactjs.org: https://reactjs.org
13. Forsberg, K., & Mooz, H. (1991, October). The relationship of system engineering to the project cycle. In *INCOSE international symposium*, 1(1), 57-65.
14. Fowler M. (2001). Separating user interface code. *IEEE Software*, 18(2), 96-97.
15. Ganpati, A., Kalia, A., & Singh, H. (2012). A comparative study of maintainability index of open source software. *International Journal of Emerging Technology and Advanced Engineering*, *2*(10), 228-230.
16. Garlan, D., Cheng, S. W., Huang, A. C., Schmerl, B., & Steenkiste, P. (2004). Rainbow: Architecture-based self-adaptation with reusable infrastructure. *IEEE Computer*, *37*(10), 46-54.
17. Gokdogan, G., (2021). J2CL. Retrieved June, 2021. GitHub: https://github.com/google/j2cl/blob/master/README.md

18. Google, (2021). Sensors overview | Android developers. Retrieved June, 2021. Andoid for Developers: https://developer.android.com/guide/topics/sensors/sensors_overview
19. Grua, E. M., Malavolta, I., & Lago, P. (2019, May). Self-adaptation in mobile apps: a systematic literature study. In *2019 IEEE/ACM 14th international symposium on software engineering for adaptive and self-managing systems (SEAMS)*, 51-62. IEEE.
20. Hales, W. (2012). *HTML5 and JavaScript Web Apps: Bridging the gap between the web and the mobile web*, O'Reilly Media, Inc.
21. Hansson, D. H. (2021). Ruby on Rails | A web-application framework that includes everything needed to create database-backed web applications according to the Model-View-Controller (MVC) pattern. Retrieved June, 2021. Ruby on Rails: https://rubyonrails.org/
22. Havelund, K., & Pressburger, T. (2000). Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer*, *2*(4), 366-381.
23. Hinchey, M., Park, S., & Schmid, K. (2012). Building dynamic software product lines. *IEEE Computer*, *45*(10), 22-26.
24. Hürsch, W., & Lopes, C.V. (1995). Separation of concerns. *Technical report NU-CCS-95-03*, Northeastern university, Boston, Massachusetts.
25. International Organization for Standardization. (2002). *Information technology — Z formal specification notation — Syntax, type system and semantics* (ISO/IEC 13568:2002).
26. International Organization for Standardization. (2011). *Systems and software engineering — Systems and software quality requirements and evaluation (SQuaRE) — System and software quality models* (ISO/IEC 25010:2011).
27. Japan Industrial Standard (2013). システム及びソフトウェア製品の品質要求及び評価(SQuaRE) ― システム及びソフトウェア品質モデル (JIS X 25010:2013).
28. Jayatilleke, S., & Lai, R. (2018). A systematic review of requirements change management. *Information and Software Technology*, *93*, 163-185.
29. Jørstad, I., van Thanh, D., & Dustdar, S. (2005, August). The personalization of mobile services. In *IEEE international conference on wireless and mobile computing, networking and communications (WiMob'2005)*, 4, 59-65. IEEE.
30. Lamsweerde, A. V. (2009). *Requirements engineering: from system goals to UML models to software specifications*. Wiley, Chichester.
31. Laprie, J. C. (2008, June). From dependability to resilience. In *38th IEEE/IFIP international conference on dependable systems and networks*, G8-G9.
32. Lightbend. (2021). Play Framework - Build modern & scalable web apps with Java and Scala. Retrieved June, 2021. Play: https://www.playframework.com/
33. Maggio, M., Papadopoulos, A. V., Filieri, A., & Hoffmann, H. (2017, May). Self-adaptive video encoder: Comparison of multiple adaptation strategies made simple. In *2017 IEEE/ACM 12th international symposium on software engineering for adaptive and self-managing systems (SEAMS)*, 123-128. IEEE.
34. Marın, B., Pereira, J., Giachetti, G., Hermosilla, F., & Serral, E. (2013). A general framework for the development of MDD projects. In *1st international conference on model-driven engineering and software development (MODELSWARD)*, 257-260.
35. Matsutsuka, T., Nagahashi, K., Nomura, Y., Hara, H. (2000). An architecture to develop presentation logic for enterprise business applications. In *Evolve 2000 conference*.

36. Matsutsuka, T. (2005). Model-driven development approach to web applications. In *IASTED international conference on software engineering*.
37. McGrath, R. G. (2013). *The end of competitive advantage*, Hardvard business review press.
38. MDN. (2021a). Same-origin policy - Web security, Retrieved July, 2021. MDN Web Docs: https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy
39. MDN. (2021b). Ajax - Developer guides, Retrieved July, 2021. MDN Web Docs: https://developer.mozilla.org/en-US/docs/Web/Guide/AJAX
40. Meier, L., Honegger, D., & Pollefeys, M. (2015, May). PX4: A node-based multithreaded open source robotics framework for deeply embedded platforms. In *2015 IEEE international conference on robotics and automation (ICRA)*, 6235-6240. IEEE.
41. Murwantara, I. M. (2020, September). An initial framework of dynamic software product line engineering for adaptive service robot. In *2020 international conference on computer science and its application in agriculture (ICOSICA)*, 1-6. IEEE.
42. OMG. (2017). Unified modeling language specification. Retrieved June, 2021. Object Management Group: https://www.omg.org/spec/UML/About-UML/
43. OpenJS Foundation. (2021). Node.js. Retrieved June, 2021. Node.js: https://nodejs.org/en/
44. Oracle. (2009). JavaServer pages technology. Retrieved June, 2021. Oracle: https://www.oracle.com/java/technologies/jspt.html
45. Oracle. (2017). JSR 369: Java Servlet 4.0 specification. Retrieved June, 2021. Java Community Process: https://jcp.org/en/jsr/detail?id=369
46. Oracle. (n.d.). JavaServer Faces technology. Retrieved July, 2021. Oracle: https://www.oracle.com/java/technologies/javaserverfaces.html
47. Pree, W. (1994, July). Meta patterns—A means for capturing the essentials of reusable object-oriented design. In *European conference on object-oriented programming*, 150-162. Springer, Berlin, Heidelberg.
48. Raheel, S. (2016, November). Improving the user experience using an intelligent adaptive user interface in mobile applications. In *2016 IEEE international multidisciplinary conference on engineering technology (IMCET)*, 64-68. IEEE.
49. Rajput, D. S., & Gour, R. (2016). An IoT framework for healthcare monitoring systems. *International Journal of Computer Science and Information Security*, *14*(5), 451.
50. Ramamoorthy, C. V., & Tsai, W. T. (1996). Advances in software engineering. *IEEE Computer*, *29*(10), 47-58.
51. Riaz, M., Mendes, E., & Tempero, E. (2009, October). A systematic review of software maintainability prediction and metrics. In *2009 3rd international symposium on empirical software engineering and measurement*, 367-377. IEEE.
52. Rumbaugh, J., Jacobson, I., & Booch, G. (2004). *The unified modeling language reference manual, 2nd edition*, Pearson higher education.
53. Salehie, M., & Tahvildari, L. (2007, May). A quality-driven approach to enable decision-making in self-adaptive software. In *29th international conference on software engineering (ICSE'07 Companion)*, 103-104. IEEE.
54. Salehie, M., & Tahvildari, L. (2009). Self-adaptive software: Landscape and research challenges. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, *4*(2), 1-42.
55. Serrano, N., & Aroztegi, J. P. (2007). Ajax frameworks in interactive web apps. *IEEE Software*, *24*(5), 12-14.

56. Shaw, M., & Garlan, D. (1996). *Software architecture: perspectives on an emerging discipline*. Prentice Hall.
57. Walker, J. D., & Chapra, S. C. (2014). A client-side web application for interactive environmental simulation modelling. *Environmental Modelling & Software*, 55, 49-60.
58. Weyns, D., & Georgeff, M. (2009). Self-adaptation using multiagent systems. *IEEE Software*, *27*(1), 86-91.
59. Weyns, D., Schmerl, B., Grassi, V., Malek, S., Mirandola, R., Prehofer, C., Wuttke, J., Andersson, J., Giese, H., & Göschka, K. M. (2013). On patterns for decentralized control in self-adaptive systems. In *Software engineering for self-adaptive systems*, 76-107. Springer, Berlin, Heidelberg.
60. Zave, P., & Jackson, M. (1997). Four dark corners of requirements engineering. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, *6*(1), 1-30.
61. Zealley, J., Wollan, R., & Bellin, J. (2018). Marketers need to stop focusing on loyalty and start thinking about relevance. *Harvard business review*: https://hbr.org/2018/03/marketers-need-to-stop-focusing-on-loyalty-and-start-thinking-about-relevance
62. 尾島良司. (1999, Nov). Java と CORBA によるシステム開発., *Unisys Technology Review*, 63, 52-59.
63. 片山 朝子, 上原 忠弘, 藤原 翔一朗, 宗像 一樹, 徳本 晋, & 前田 芳晴. (2013). 業務システムを対象としたシンボリック実行による検証試行. *ソフトウェアエンジニアリングシンポジウム*, 1-6.
64. 桑原 寛明, 金子 伸幸, 渥美 紀寿, 山本 晋一郎, & 阿草 清滋. (2008). 学と産の連携による基盤ソフトウェアの先進的開発:7.高信頼 WebWare 生成技術：WebWare のテスト・解析・作成支援. *情報処理*, 49(11), 1271-1276.
65. 鄭 顕志, 清水 遼, 高橋 竜一, & 石川 冬樹. (2014). 自己適応ソフトウェアのための自己適応性設計に関する研究動向, *コンピュータソフトウェア*, 31(1), 1_49-1_59.
66. 富士通. (2020, Mar). イベント連動型マッチングサービス「Buddyup!」を活用した川崎市域での経済活性化の取り組みを川崎市産業振興財団と開始. Retrieved June, 2021. 富士通研究所プレスリリース: https://www.fujitsu.com/jp/group/labs/about/resources/article/202002-buddyup.html
67. 福安 直樹, 初見 太輔, 満田 成紀, 吉田 敦, & 鯵坂 恒夫. (2004, Nov). JavaScript を含む Web アプリケーションの解析手法. *日本ソフトウェア科学会ソフトウェア工学の基礎研究会*, 229-232.

業績

学術誌論文

1. 松塚 貴英, 阿草 清滋, 山本 晋一郎. (2005). ラウンドトリップエンジニアリングを目指したWebアプリケーションのための意味モデル, *情報処理学会論文誌*, 46(5), 1145-1154.
2. 小高 敏裕, 松塚 貴英, 野村 佳秀, 村上 雅彦, 山本 里枝子. (2007). SIPアプリケーションフレームワークの開発と適用, *情報処理学会論文誌*, 48(8), 2674-2683.
3. 松塚貴英. (2008). 業務アプリケーションに適用する Ajax フレームワーク, *情報処理学会論文誌*, 49(7), 2360-2371.

国際会議発表

1. Matsutsuka, T., Nagahashi, K., Nomura, Y., & Hara, H. (2000). An architecture to develop presentation logic for enterprise business applications. In *Evolve 2000 conference*.
2. Ookubo, T, Matsutsuka, T., Nomura, Y., Hara, H., & Uehara, S. (2000). Practical experiences of designing a distributed collaborative system. In *Fourth international enterprise distributed object computing conference (EDOC)*, IEEE.
3. Matsutsuka, T. (2005, Feb). Model-driven development approach to web applications. In *IASTED international conference on software engineering*.
4. Yamamoto, K., & Matsutsuka, T. (2016, January). Efficient call path detection for Android-OS size of huge source code, In *Sixth international conference on computer science, engineering and applications (CCSEA 2016)*, Dubai, UAE.
5. Matsutsuka, T., & Iida, I. (2021, Mar). A matching service to adapt changing situations. In *19th IADIS international conference e-Society*.

国内学会発表

1. 松塚 貴英, 野村 佳秀. (2000). JSP を用いた Web アプリケーション構築のためのフレームワーク UJI, *情報処理学会研究報告(2000-SE-129)*, 9-16.
2. 野村 佳秀, 松塚 貴英. (2000). Webアプリケーションフレームワーク UJI の GUI 構成部品のコンポーネント化技術, *情報処理学会研究報告(2000-SE-129)*, 17-24.
3. 松塚 貴英. (2001). ビジネスアプリケーションにおけるフレームワークの適用に関する考察, *ウィンターワークショップ・イン・金沢論文集*, 情報処理学会.
4. 松塚 貴英, 高山 龍二. (2007). 業務アプリケーションに利用するための Ajax フレームワーク, *ソフトウェアエンジニアリングシンポジウム 2007 論文集*.
5. 松塚 貴英, 飯田 一朗. (2019). 動的適応部品による自己適応ソフトウェアの検討. *情報処理学会研究報告コンシューマ・デバイス&システム(CDS)* ,2019-CDS-25(5),1-8 (2019-05-23) , 2188-8604.
6. 安曽 徳康, 小川 雅俊, 松塚 貴英, 鄭 顕志. (2020). モデル予測制御と離散制御器合成による外部環境の動的特性を考慮した適応制御手法, *マルチメディア, 分散, 協調とモバイル(DICOMO2020)シンポジウム*, 623-624.

その他発表

1. Fujikawa, Y., & Matsutsuka, T. (2004, Jun). New web application development tool and its MDA-based support methodology, *Fujitsu Science & Technical Journal*, 40(1), 94-101.
2. Hu B., Carvalho, N., Laera, L., Lee V., & Matsutsuka, T. (2012). Applying semantic technologies to public sector: A case study in fraud detection. In *Joint international semantic technology conference (JIST)*, 319-325.
3. Naseer, A., Laera, L., & Matsutsuka, T. (2013). Enterprise BigGraph. In *Hawaii international conference on system sciences (HICSS)*, 1005-1014.
4. Hu, B., Carvalho, N., & Matsutsuka, T. (2013). Towards big linked data: A large-scale, distributed semantic data storage. *International Journal of Data Warehousing and Mining (IJDWM)*, 9(4), 19-43, IGI Global.
5. Igata, N., Nishino, F., Kume, T., & Matsutsuka, T. (2014, January). Information integration and utilization technology using linked data, *Fujitsu Science & Technology Journal*, 50(1), 3-8.
6. Hu, B., Naseer, N., & Matsutsuka, T. (2014, January). Anomaly detection technology using BigGraph, *Fujitsu Science & Technology Journal*, 50(1), 9-15.
7. Matsutsuka, T., Takahashi, M., Noma, Y., Washio, S., & Ikeda, E. (2020, May). Social Conditions Regarding Data Utilization and Fujitsu's Initiatives. *Fujitsu Technical Review*, 2.