博士論文

インターネットサービスにおける投資効率を最大化するソフトウェア開発プロセスの研究

～継続的デリバリサイクルの短期化による効果とメカニズム～

大島　將義

主指導教員　　内平　直志

北陸先端科学技術大学院大学
先端科学技術専攻
（知識科学）

令和７年３月

Research on software development processes that maximize investment efficiency in internet services. -Effects and mechanisms of shortening the continuous delivery cycle.-

2240004 Masayoshi Oshima

In today's VUCA environment, companies must respond quickly and continuously improve. Therefore, it is crucial to quantitatively assess how shorter software release cycles help manage investment uncertainty and maximize profits.

In this study, we conducted a simulation based on a real case at Recruit Co., Ltd. to analyze how shorter Continuous Delivery cycles contribute to mechanisms for addressing investment uncertainty and increasing profit from the perspective of Continuous Software Engineering (CSE). Specifically, we accounted for the success rates of various improvement measures and examined how changes in these rates affect the expected value. We also clarified how early identification of successful measures, as well as the reliable detection and removal of unsuccessful ones, can influence business outcomes. These processes are enabled by shorter delivery cycles and by releasing improvement measures in the smallest measurable units, even when success rates are low.

The simulation results show that shorter delivery cycles are an effective means of increasing investment efficiency in highly uncertain environments. In particular, even in cases where success rates are low, the introduction of shorter cycles can improve overall profitability by actively capturing the success of improvement measures and quickly eliminating failures. This study provides rational criteria for internet service companies to adopt a rapid improvement process and offers empirical support for the economic benefits of shorter delivery cycles.

Previous studies have suggested the importance of CSE and the existence of benefits from shorter delivery cycles. They have often been vaguely expressed as “faster is better in a competitive environment” or “Just in Time.” The novelty of this research is that it quantitatively demonstrates these benefits, thereby clarifying the study's purpose and business significance—for example, by highlighting the challenges of introducing Continuous Delivery and proposing potential solutions. The academic contribution lies in demonstrating the importance of CSE research and enabling quantitative evaluation of CSE in future studies.

The results provide useful information not only for practitioners in software development but also for managers and investors. In particular, it is expected to play a vital role in corporate decision-making by concretely illustrating the business benefits gained through the adoption of shorter delivery cycles.

Nevertheless, this study has certain limitations. Our simulation model assumes a single developer and uniformly sized improvement measures, neglecting interactions among them.

Real projects often involve multiple developers, and overhead persists even with automation. Future simulations that address these complexities could provide a more accurate view of real-world software development processes and better validate the economic benefits of shorter delivery cycles.

Keyword: Internet Service, Software Engineering, Project Management, Program Management, Continuous Software Engineering, Continuous Delivery, Software Development Process

Abstract

本研究は、インターネットサービスにおけるソフトウェア開発プロセスの一環として、継続的デリバリ（Continuous Delivery, CD）サイクルの短期化が投資効率に与える影響を探るものである。現代のビジネス環境は VUCA の時代であり、企業は迅速な対応と継続的な改善を求められている。ここで、ソフトウェアの短いリリースサイクルがどのようにして投資の不確実性に対応し、利益を最大化するかを定量的に評価することは重要である。

本研究では、株式会社リクルートの実例を基に、Continuous Software Engineering（CSE）の観点から、デリバリサイクルの短期化が投資不確実性の対応メカニズムおよび獲得利益の増加メカニズムにどのように寄与するかについて、シミュレーションを通じて分析した。具体的には、改善施策の成功率を考慮し、成功確率の変動に伴う期待値の変化を評価した。また、成功率が低い場合でも、短期サイクルによって得られる成功施策の拾い上げと失敗施策の早期破棄が、ビジネスアウトカムに与える影響を明らかにした。

シミュレーション結果は、デリバリサイクルの短期化が不確実性の高い環境において、投資の効率性を高める有効な手段であることを示している。特に、成功率が低い場合においても、短期サイクルの導入は、改善施策の成功を積極的に取り込み、失敗を迅速に排除することで、全体的な利益を向上させることができる。本研究は、インターネットサービス企業が迅速な改善プロセスを採用するための合理的な判断基準を提供し、CD の短期化による経済的利益を実証的に裏付けるものである。

従来の研究では、CSE の重要性や、CD 短期化のメリットが存在することは前提とされてきた。競争環境において早い方が良い、Just In Time といった漠然としたメリットとして表現されてきた。本研究では、それを定量的に示す点が新規性であり、それは例えば Continuous Delivery 導入先行の困難や解消方法の提案といった研究に対して、その目的やビジネス上の重要性を説明可能にする。この様に CSE 研究の重要性を示し、今後の後続研究において、定量的に評価可能とすることに、学術的な貢献がある。

この成果は、ソフトウェア開発の現場の実務者だけでなく、経営層や投資家にとっても有益な情報を提供する。特に、短期的なリリースサイクルの導入によって得られるビジネス上の利点を具体的に示すことで、企業の意思決定プロセスにおいて重要な役割を果たすことが期待される。

ただし、この研究の限界は、シンプルなシミュレーションモデルを採用している。例えば、一人の開発者と一律のサイズの改善策を想定していたり、開発者間の相互作用を無視していたりする点である。今後、現実に近いより複数なシミュレーションに拡張し検討することで、現実のソフトウェア開発プロセスをより正確に捉え、デリバリサイクルの短縮によるメリットについての研究を深化させる余地がある。

目次

図目次

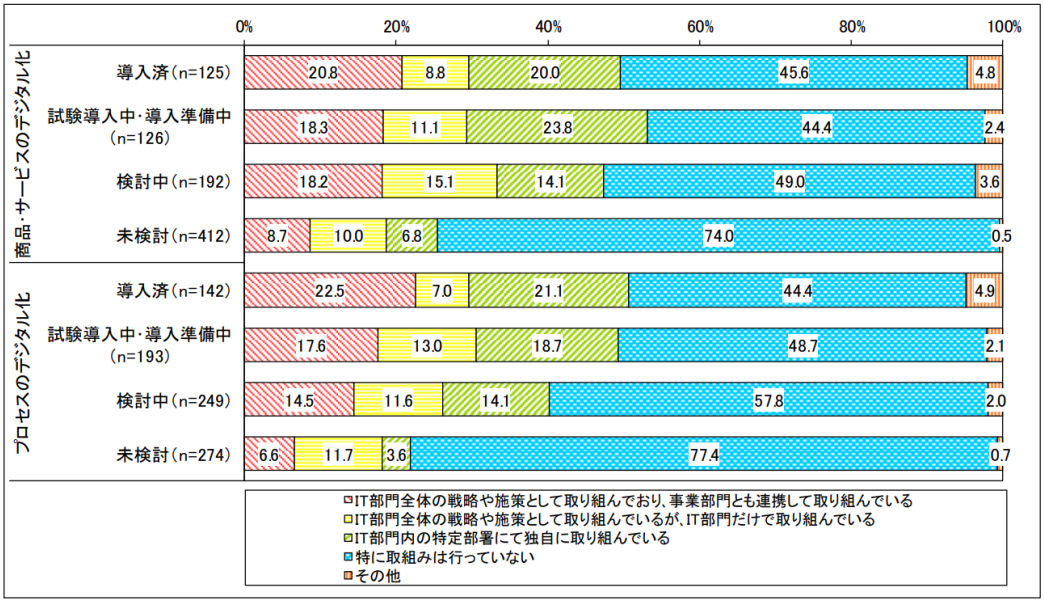
表目次

第1章 序論

1.1 研究の背景

現代は、変動（Volatility）、不確実（Uncertainty）、複雑（Complexity）、曖昧（Ambiguity）の頭文字からＶＵＣＡの時代と称される。その様な状況下で、各企業は競争力の維持・強化をおこない、事業存続・事業成長させていかなければならない。言い換えれば、何をすべきか、何が正解なのか、が分からない中で、それでもなお着実に利益を上げることが求められている。これは、ゴールから逆算的に計画を立て、実行していくという方法では立ち行かなくなっていることを意味する。もう一つの現代の特徴は、急速に進むデジタル技術の発展と、そのデジタル技術への経済依存度の高まりである。米国におけるデジタル経済による実質産出額は、2005 年に 1.6 兆ドルあったところ、2018 年には 3.0 兆ドルまで増加している(内閣府 2021)。

すなわち、現代は長期的な計画が難しい中で、ビジネスのデジタル化を実現（既存ビジネスのデジタル移行・新規デジタルビジネスの立ち上げ）していくことが求められていると言えるだろう。実際に、経済産業省の DX レポート 2 (経済産業省デジタルトランスフォーメーションの加速に向けた研究会 2020）においては、デジタル社会においては、顧客ニーズの変化に対して迅速に適応し続けることが企業の目指すべき方向性と示されており、日本国内においては、インターネットサービス開発を行う一部のユーザ企業でアジャイル開発が行われているとされている。DX レポート２．1 (経済産業省デジタル産業の創出に向けた研究会 2021）においても、顧客ニーズの変化に迅速に適応し続けるための成功パターンとしてアジャイル・DevOps の活用が重要とされている。また、企業 IT 動向調査報告書(日本情報システム・ユーザー協会 2020)では、デジタル化の進んでいる企業の 5 割はアジャイル開発に取り組んでおり、デジタル化に未着手の企業ではアジャイル開発への取り組みが 3 割以下である調査報告がされており、デジタル化とアジャイル開発には、一定の相関関係があると示唆されている（図 1)。



企業 IT 動向調査報告書２０２０(日本情報システム・ユーザー協会 2020)より引用

図 1 デジタル化への取組み状況別　アジャイル開発の取組み状況

従来、ソフトウェア開発において主流であったのはウォーターフォールと呼ばれる計画・実行・管理を中心とするマネジメント方法であった。しかし、計画を遂行するためには優れた方法であったが、不確実性の高い状況下で、計画変更が多発する場合においては相応しくないという課題があった(Cusumano 2004)。その様な課題感から、製造業における新製品開発のプロセスマネジメント研究(Takeuchi and Nonaka 1986) をヒントに実際のソフトウェア開発現場で構築されたプロセスを、抽象化してスクラムと命名した(Schwaber et al. 2003)。このスクラムと同時期に XP(extreme programming)など様々なプロセスが考案されたが、それらが２００１年に「アジャイルソフトウェア開発宣言(agilemanifesto.org 2001a)」として総称された。このように、アジャイル開発とは、計画可能であることを前提としたウォーターフォールに対する課題に対して生まれたものであり、まさに VUCA 時代にマッチするソフトウェア開発プロセスであると言える。

しかしながら、アジャイル開発自体は多様な方法論が存在しており、明確な定義が存在しない。唯一存在する共通的な定義は、「アジャイル開発宣言」のみであり、極めて抽象度の高いマインドセットと１２の原則が存在するのみである。一般的なワードとしては広く普及しているのだが、そこに共通の定義はない。特にアジャイル開発が不確実

性に対応しているという事自体は、アジャイル開発についての研究や書籍(Darrell et al. 2016; Boehm et al. 2004)において言及されることはあっても、その理由やメカニズムについては言及されていない。

この様に実務的にはアジャイル開発が、従来のソフトウェア工学の問題点を克服しようとしているのと並走する形で、学術的には Continuous Software Engineering（CSE）という概念が近年ホットなテーマとなってきている。これは、Facebook 社（現 Meta 社）のソフトウェア開発の事例研究において、「完成しないソフトウェア開発」が報告されたことに端を発する(Feitelson, Frachtenberg, and Beck 2013)。従来のソフトウェア工学は、ソフトウェアを完成させるための知識体系として発展してきたが、インターネットサービス業界では、ソフトウェアは完成せず永遠に改善を続けているため、従来のソフトウェア工学を拡張する必要があることを示唆していた。ただし、この CSE 研究は、純粋な工学的なアプローチだけではなく、経営学や情報システム学や組織学等の学際的な知識の統合が必要となると提言されている(Fitzgerald and Stol 2017)

1.2 研究の目的とリサーチ・クエスチョン

この様な背景の中で、本研究は CSE の 1 テーマである Continuous Delivery というテーマを使っている。これは、継続的な改善を短い期間で繰り返すプロセスを指している。本研究の目的は、この Continuous Delivery(CD)の 1 サイクルを短期化するという、ビジネスプロセスの変革がもたらすメリットについて、それを定量的に表現し、その発生メカニズムを明らかにすることである。結果として、単に「早いことは良い」という漠然とした目的ではなく、ビジネスアウトカムとして定量的表現することを可能とし、あらゆる企業にデリバリサイクルを短期化させる合理的判断軸を提供する。

よって、本研究のリサーチ・クエスチョン(RQ)は、

RQ: デリバリサイクル短期化がビジネス上の有効性（継続的な利益）を発揮する条件と、その有効性はどの程度か？

と設定した。この RQ を明らかにするため、以下 3 つの研究を実施している。

研究 1: デリバリサイクル短期化は、どの様なメカニズムで投資不確実性に有効性があり、その有効性はどの程度かをシミュレーションで明らかにする。

研究 2: デリバリサイクル短期化は、どの様なメカニズムで獲得利益を増大化させ、その増大量はどの程度かをシミュレーションで明らかにする。

研究 3: 研究 1,2 を統合した結果として、その有効性はどの程度かをシミュレーションで明らかにする。

インターネットサービスに対して、長期間で複数の改善をおこなう場合に、それらの改善をリリースする括り方（デリバリサイクル）を変更させることで、ビジネスアウトカムに差分が発生するのかを明らかにすることが RQ への回答である。ここで差分が発生する要因は、複数改善の成功率による影響と、純粋な括り方（デリバリサイクル）による影響となる。これを、それぞれ研究 1、研究 2 にて明らかにし、研究 3 で統合する。

1.3 研究の方法

本研究の方法は、シミュレーションを採用している。本研究の特色はデリバリサイクルという開発プロセスと、ビジネスアウトカムの関係について、その影響を定量的に表すことを目的としている。そのため、開発プロセスの複数パターン毎に、結果としてビジネスアウトカムの違いを比較することが必要となる。つまり、開発プロセスとビジネスアウトカムという 2 変数を扱う。事例研究では複数の開発プロセスパターンを探し出すことは出来るが、それが異なるインターネットサービスであれば、ビジネスアウトカムを比較することが出来ない。逆に、同じインターネットサービスだとしたら、同じ改善を異なるプロセスで複数回行われることはありえない。実証研究的に、全く同じインターネットサービスを複数用意して、同じ改善を複数のプロセスで行うことは、莫大なコストを要するため現実的ではない。また、その複数の同じインターネットサービスを同時に公開してビジネスアウトカムを測定することも非現実的である。そこで、第 3 章で現実のプロセスを事例研究し、それを模倣したシミュレーションを設計した。第 4 章・

第 5 章・第 6 章では、様々な条件でシミュレーションを行うというアプローチを採用した。

1.4 用語の定義

インターネットサービス

インターネットなどの電子的な手段を通じて提供される各種サービスの総称(Porter 2001; Rust and Kannan 2003)。広義には、電子メールなども含むが、本論文では WEB サイト・スマートフォンアプリの形で提供されるサービスを指す。

Software Engineering (SE)

ソフトウェア工学を指している。ソフトウェア工学とは、「ソフトウェアの作成と利用に関連した概念を科学的に抽出 体系化し、正しいソフトウェアを計画的かつ効率的に作成・利用するための理論と、実践的技術」(長尾 et al. 1990)という定義である。なお、本論文では、後述の CSE との関係性を分かりやすくするため、敢えて英語表記を使用している。

Continuous Software Engineering (CSE)

従来のソフトウェア開発が完成を目的としてきたことに対して、完成することなく継続的に改善と最適化を行うソフトウェア開発についての知識体系。Continuous Integration、Continuous Delivery、Continuous Deployment など、様々な分野を内包する、最も大きな概念(Klotins and Gorschek 2022)。

Continuous Delivery

CSE のテーマの一つ。ソフトウェアを短期間で反復的にリリースする能力を高める一連のプラクティスと原則であり、これにより、ソフトウェアを高い信頼性で継続的にリリースすることが可能となる(Humble and Farley 2010)。

省略すると同じ CD となることから、後述の Continuous Deployment と混同されて

いることが多い。この両者の使い分けは、リリース作業を手動で行うか、自動で行うかとされている。本論文も、それを踏襲してプロセス全般のことを Continuous Delivery（CD）と呼び、自動リリースについての言及をする場合は、Continuous Deployment という表現を採用している。

Continuous Deployment

全てのコード変更がテストを通過した後、自動的に本番環境にデプロイされ、ユーザに即座に提供されるプロセスのこと(Savor et al. 2016)。前述の通り、Continuous Delivery と混同されることが多い。Continuous Delivery の項を参照のこと。

デリバリサイクル

本論文独自の定義。繰り返し行われる Continuous Delivery の 1 回を指す。アジャイル開発のスクラムでいえばスプリントの概念と近く、要求（要件）定義、設計、実装、テストという一連のエンジニアリング工程のこと。

デリバリサイクルの短期化

本論文独自の定義。1 回のデリバリサイクル期間を短期にすること。例えば、半年かかるデリバリサイクルであれば２５週間であるが、これを短期化すると 1 週間となる。実質的には、リリース頻度のことを意味しているともいえる。半年に 1 度リリースをするならデリバリサイクル２５週であり、毎週リリースを繰り返すならデリバリサイクル１週である。

ビジネスアウトカム

本論文独自の定義。結果・成果を指す一般的な用語であるアウトカムに対して、ビジネス活動の結果として得られる成果という意味の造語。ビジネスごとに、重要なことが変わるため、売上、利益、顧客満足度、マーケットシェアなど様々なモノを指す。

改善

本論文では、インターネットサービスの改善を狙った施策が、結果的にビジネスアウトカムにたいしてポジティブな影響を生んだ場合を改善として扱う。

改悪

本論文では、インターネットサービスの改善を狙った施策が、結果的にビジネスアウトカムにたいしてネガティブな影響を生んだ場合を改悪として扱う。

成功率

成功する確率のこと。本論文では、インターネットサービスの改善を狙った施策が、結果的に改善・改悪・変化なしとなる。改善÷全施策数＝成功率という割合にて表現される。結果として、変化なしは成功にはカウントされない。

投資収益性

ソフトウェア開発に投下したリソースに対して、得られるビジネスアウトカムの割合を指す。1名の開発人件費を投下して年間１００万円のビジネスアウトカムを得るのと、２名の開発人件費を投下して年間１０００万円のビジネスアウトカムを得るのでは、後者の方が高い投資収益性となる。

1.5 本研究の構成

第１章　序論

本研究の背景、目的、リサーチ・クエスチョン、研究方法について説明する。これらの要素を通じて、本研究の全体像を明確にする。

第２章　先行研究レビュー

ソフトウェア開発に関する知識体系の系譜と課題を中心に先行研究をレビューする。これにより、本研究の学術的位置づけを明確にする。

第３章　インターネットサービスにおける改善プロセス

事例研究を通じて、インターネットサービスの具体的なプロセス、開発内容、その特性について明らかにする。これらの具体的情報を、以降の研究において利用するシミュレーションモデルの前提数値として扱う。

第４章　デリバリサイクル短期化による投資不確実性の対応メカニズム

研究１に該当する。第３章で、明らかになった改善の成功率を基準にシミュレーションモデルを構築し、デリバリサイクルを短期化することによって獲得するとされている不確実性への対応力が、どの様に発生するのか？そして、その影響の程度について明らかにする。

第５章　デリバリサイクル短期化による獲得利益の増加メカニズム

研究２に該当する。長期時間軸かつ、多数の開発施策を開発した場合の定量的なビジネスアウトカムへの影響を計算するシミュレーションモデルを構築する。このシミュレーションモデルをベースに第５章のシミュレーションを行う一助となる。また、成功率１００％の状況で、単純なリリースタイミングだけで、どの程度のビジネスアウトカムへの影響があるのかを定量的に評価する。また、その影響の発生メカニズムを明らかにする。

第 6 章　デリバリサイクル短期化による統合的なシミュレーション

研究 3 に該当する。第 4 章、第 5 章で明らかにしたメカニズムと、構築したシミュレーション方法を用いて、統合的なシミュレーションを行う。結果として、それらの統合されたビジネスアウトカムへの影響を定量的に評価する。さらに、デリバリサイクル短期化による損失・障壁についても検討を行う。

第 7 章　考察

本研究の結果を先行研究と比較・議論することを通じて、デリバリサイクルの短期化が、ソフトウェア開発投資の効率化に寄与するメカニズムを明らかにし、知識科学の視点、プログラムマネジメントの視点など、多角的な視点から本研究結果を考察する。

第 8 章　結論

リサーチ・クエスチョンへの回答と、本研究の理論的な意義と、実務的な意義をまとめる。さらに、今後の研究の方向性についても示唆する。

第2章 先行研究レビュー

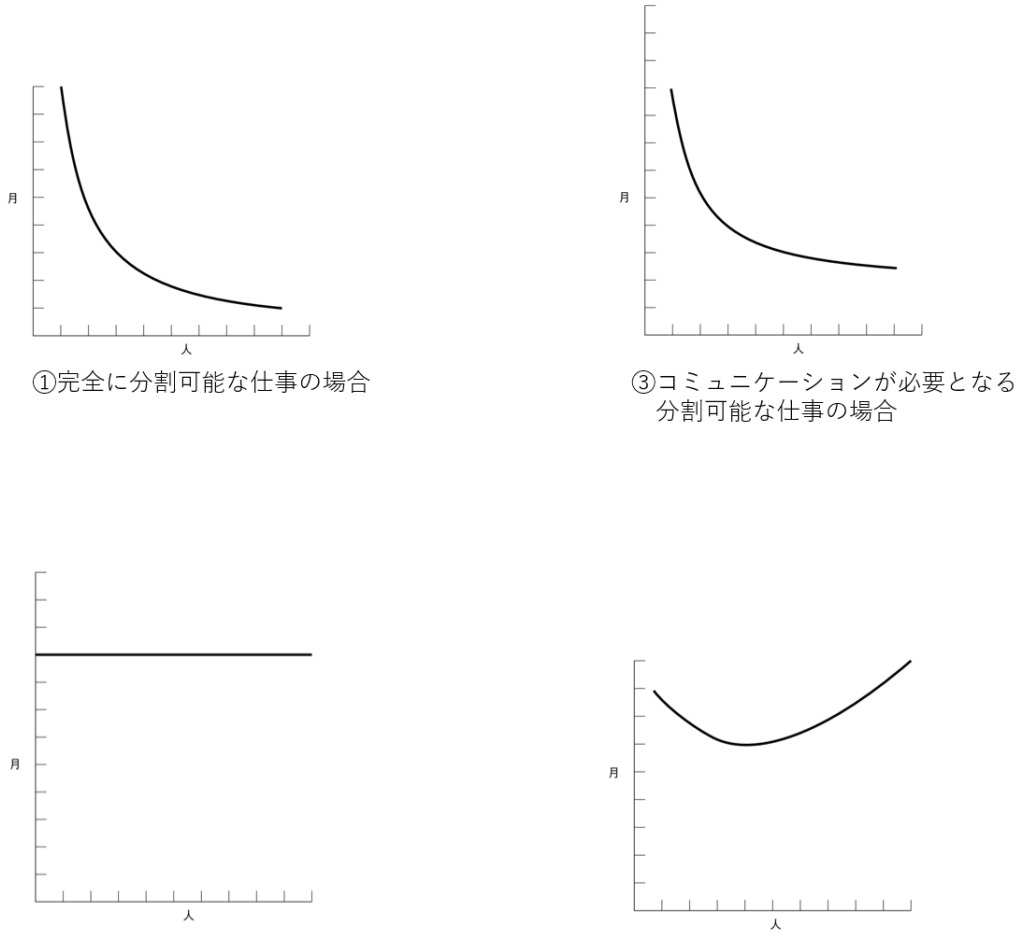
本章では、ウォーターフォール型開発を中心にソフトウェア工学の歴史的背景と基本的概念について概観する。次に、ウォーターフォール型開発の課題と、それを解決するために現場ノウハウ先行で台頭したアジャイル型開発についての背景と知識について整理する。その後、このような背景の中で、新しいソフトウェア工学として、近年注目されている CSE および、その中核となる CD について議論を行う。一方で、本研究はソフトウェア開発のおかれている環境が、近年ビジネス的に高い不確実性の影響を受けて発展していると考え、このような不確実性の高い環境下での投資理論としてリアル・オプション理論に関する研究を紹介する。これらを通じて、本研究の位置づけを明確にする。

2.1 Software Engineering(SE)に関する先行研究

ソフトウェア工学とは、「ソフトウェアの作成と利用に関連した概念を科学的に抽出体系化し、正しいソフトウェアを計画的かつ効率的に作成・利用するための理論と、実践的技術」(長尾 et al. 1990)である。この様に、ソフトウェア工学の範囲は非常に広範である。

開発プロセスとは、ソフトウェア工学の 1 分野であり、ユーザのニーズをソフトウェアに変換する過程といえるが、そのソフトウェアの特性によって、難易度が変動するという性質がある。これはソフトウェア開発の黎明期から発生した問題で、実際に 1964 年に発表された初の汎用コンピュータと言われる IBM の System/360 のオペレーティングシステムである OS/360 の開発プロジェクトにおいても巨大化し、組織の混乱も発生し、リリースが大幅に遅れることになったと言われている。この際のプロジェクトマネージャーであった F.P.Brooks は自著である the Mythical Man-Month(人月の神話)の中でブルックスの法則を記している(Brooks et al. 2002)。ブルックスの法則とは、作業総量は人月で表されるが、それ故に人数と時間には互換性が有ると錯覚に陥るという事である。実際には、作業の分割性、コミュニケーションの必要性、作業の相互関連性などで、人数と時間の関係は変わってしまい、必ずしも１０００人月の作業は１０００人で行えば 1 か月で終わる訳ではない。つまり、人月という概念は、物事を分かりやす

く表現する一方で、誤解を発生させる（図 2)。



The mythical man-month (Brooks et al. 2002)より、画像番号など筆者改変

図 2 ブルックスの法則

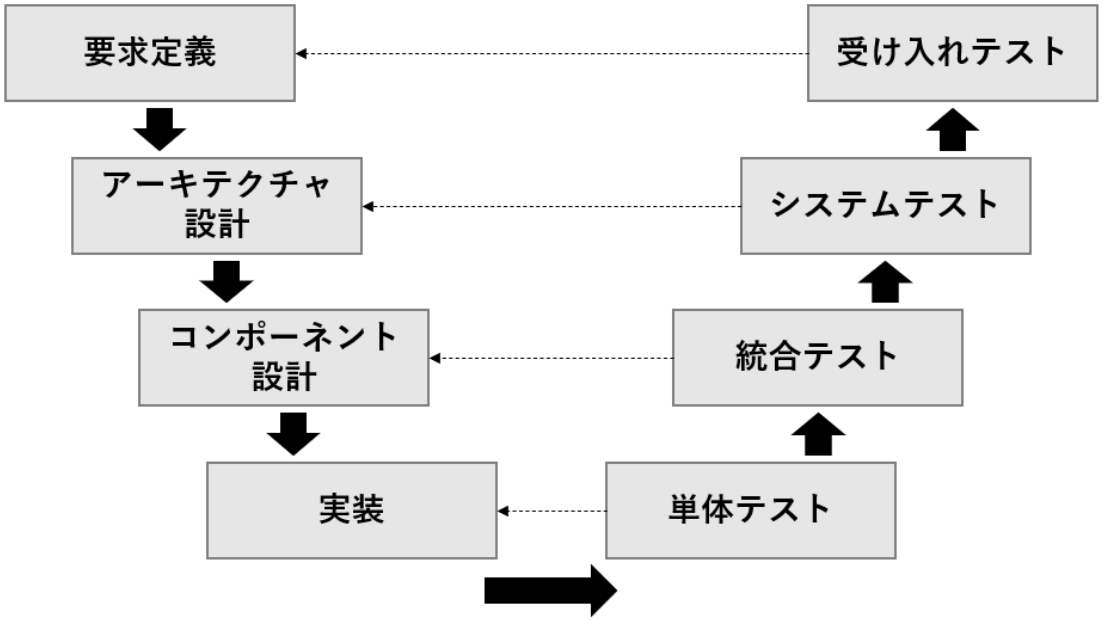
このように、現在においても何ら変わることのない難しさは、ソフトウェア開発の黎明期から存在していた。そのような中で、1968 年の 10 月にドイツの Garmisch で Software Engineering という名前の会議が開催された。ここでの議題は、情報システムの信頼性担保や、大規模ソフトウェア開発プロジェクトで、納期を守りながら仕様を満たすことが難しいということへの対処、ソフトウェア技術者の教育であったとされている(玉井 2008)。

その後、世界中のソフトウェア開発はウォーターフォールモデルと呼ばれるソフトウェア開発プロセスが主流となる。これは、W.W.Royce が考案した System Requirement、Software Requirement、Analytics、Program Design、Coding、Testing、Operations という 7 つのステップからなるソフトウェア開発プロセスモデル(Royce 1970)が源流である。そこでは、それまでの職人芸のようなソフトウェア開発ではなく、製品製造のようなトップダウンアプローチが求められているが、それだけではうまくいかず、①プログラム設計を最初に行う、②ドキュメントの最新性と完成品であることの保証、③2 度行う[1]、④テストを計画・管理・モニタする、⑤顧客の巻き込みが必要である　と主張している。ウォーターフォールという命名自体は、1976 年に TRW 社の Bell と Thayer が初めて使ったのが起源であり、プロジェクトマネジメント(Project Management Body of Knowledge：PMBOK)と共に広く普及した。ただし、この普及過程において、Royce の主張した③について無視され、ウォーターフォールとは、それぞれの作業ステップを一回だけ通して行うことが特徴であるという説明がなされるようになった(小椋 2013)。

2.2 アジャイル開発に関する先行研究

ウォーターフォールの問題は、①変化への対応②開発上のリスクへの対応の 2 点が存在する。①は、開発途中に競合他社が新製品を出した、OS がバージョンアップしたといった自社では計画しきれない問題への対応が不可欠であるときに、こういった変化への対応が明示的に定義されていないことであり、②は V 字モデル（図 3）を前提としているため、開発初期に定義した内容ほど後期にならなければ確認できない。結果として、「要求定義」における定義の誤りや誤解は、「受入れテスト」にて発覚するため、大きな手戻りが往々にして発生するということである(岸 and 野田 2016)。

[1] 初めて開発するコンピュータプログラムであれば、最終的な運用バージョンは試作を行い 2 番目のバージョンとなるように準備するとしている。



ソフトウェア工学(岸 and 野田 2016)を筆者模写

図 3 V字モデル

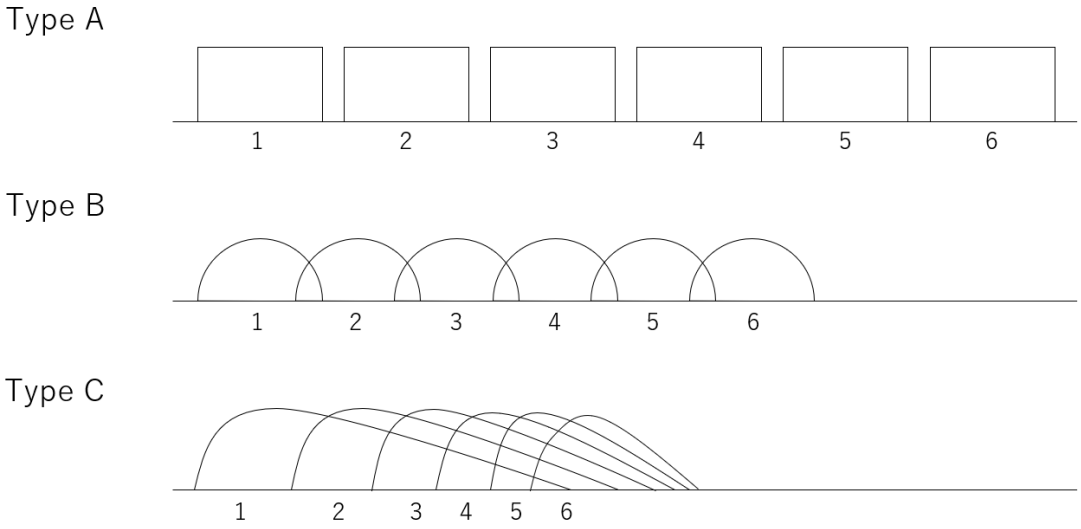
また、複数の機能を一斉にリリースを行うプロセスは、オープンソースソフトウェアの運営においては不都合が多く、多くのオープンソースソフトウェアでは早期・頻繁にリリースするプロセスが慣行として確立されている(Michlmayr, Fitzgerald, and Stol 2015)。

長らく、ソフトウェア開発におけるデファクトであったウォーターフォールプロセスは、ソフトウェア開発を成功に導くことに貢献したが、一斉に開発し、一斉にリリースするというプロセス特性に関して、多くの問題が存在する。一方で、ウォーターフォールモデルの課題を解決する手法として活用が拡大しているのがアジャイル開発である(Kurapati, Manyam, and Petersen 2012)。

しかし、アジャイル開発といっても、「ウォーターフォールではない手法」と指しているのではないかと思えるほどに、多様な方法論が存在する。それらは、様々な知識をソフトウェア開発に持ち込むことで確立されてきた。一例として、有名なアジャイル開発のプロセスがスクラムである。これは、製造業における新製品開発プロセスの研究(Takeuchi and Nonaka 1986)において TypeA、TypeB、TypeC と呼ばれるフェーズ間

の関係性の類型が整理され、TypeC の特徴をスクラム（図 4）と命名したことを参考にしたソフトウェア開発手法である。

TypeA とは工程間が明確であり、文書でバトンを渡すようにリレーするのに対して、TypeB は工程間の隣接するフェーズの境界のみオーバーラップし、TypeC は隣接だけでなく複数の工程に跨ったオーバーラップするとしている。この TypeC を、ラグビーの様にボールを前後に回しながらチーム一丸となってボールを運ぶ姿に例えてスクラムと命名した。この TypeC の特徴は、Built-in instability（不安定さ）、Self-organizing project teams（自己組織化されたプロジェクトチーム）、Overlapping development phases（重複させる開発フェーズ）、Multi learning（マルチ学習）、Subtle control（柔らかなマネジメント）、Organizational transfer of learning（学習を組織で共有する）としている。これは、新規製品開発というスピードと柔軟性が求められる場においてTypeC が有効であるという研究である。

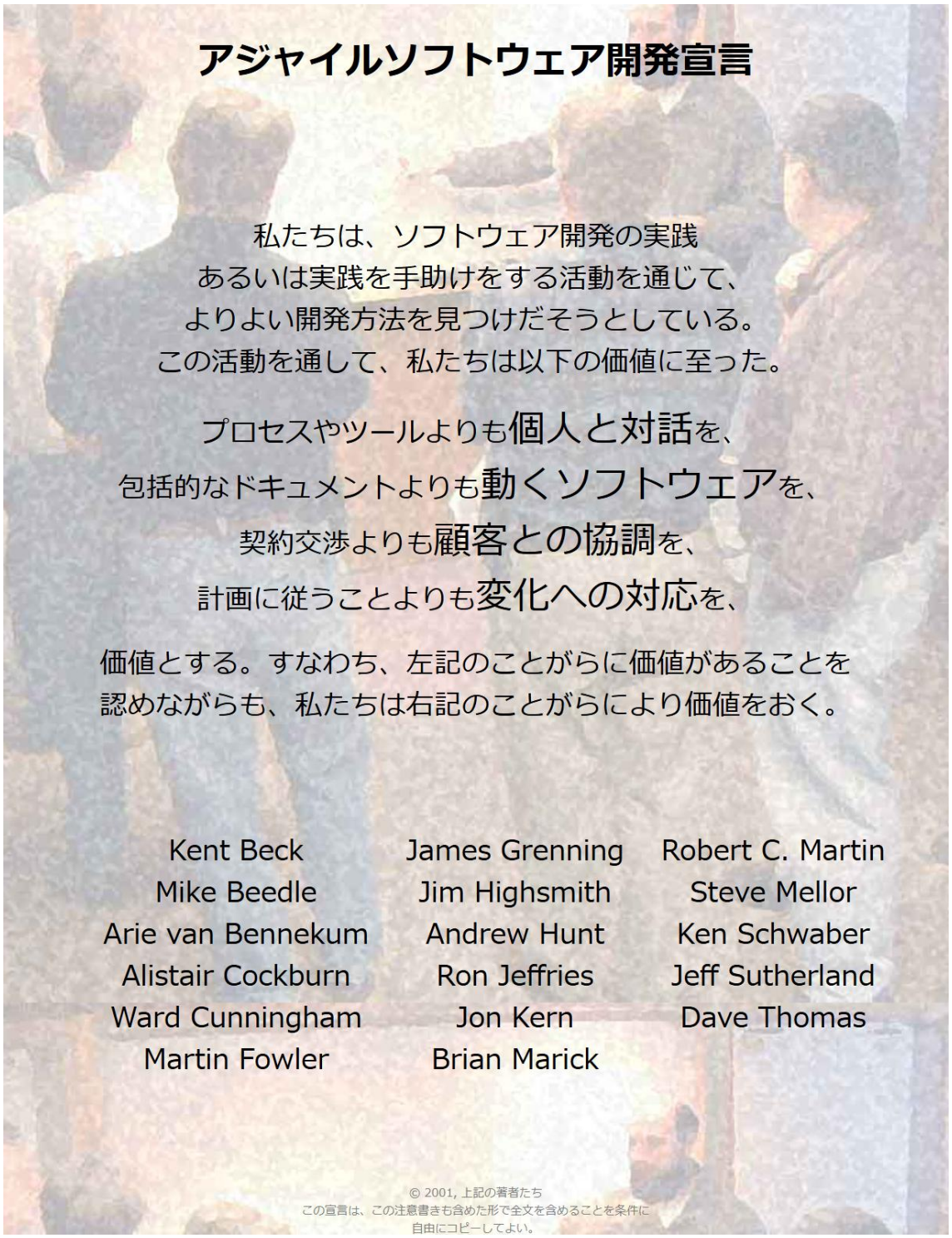


The new new product development game (Takeuchi and Nonaka 1986)を筆者模写

図 4 スクラム開発

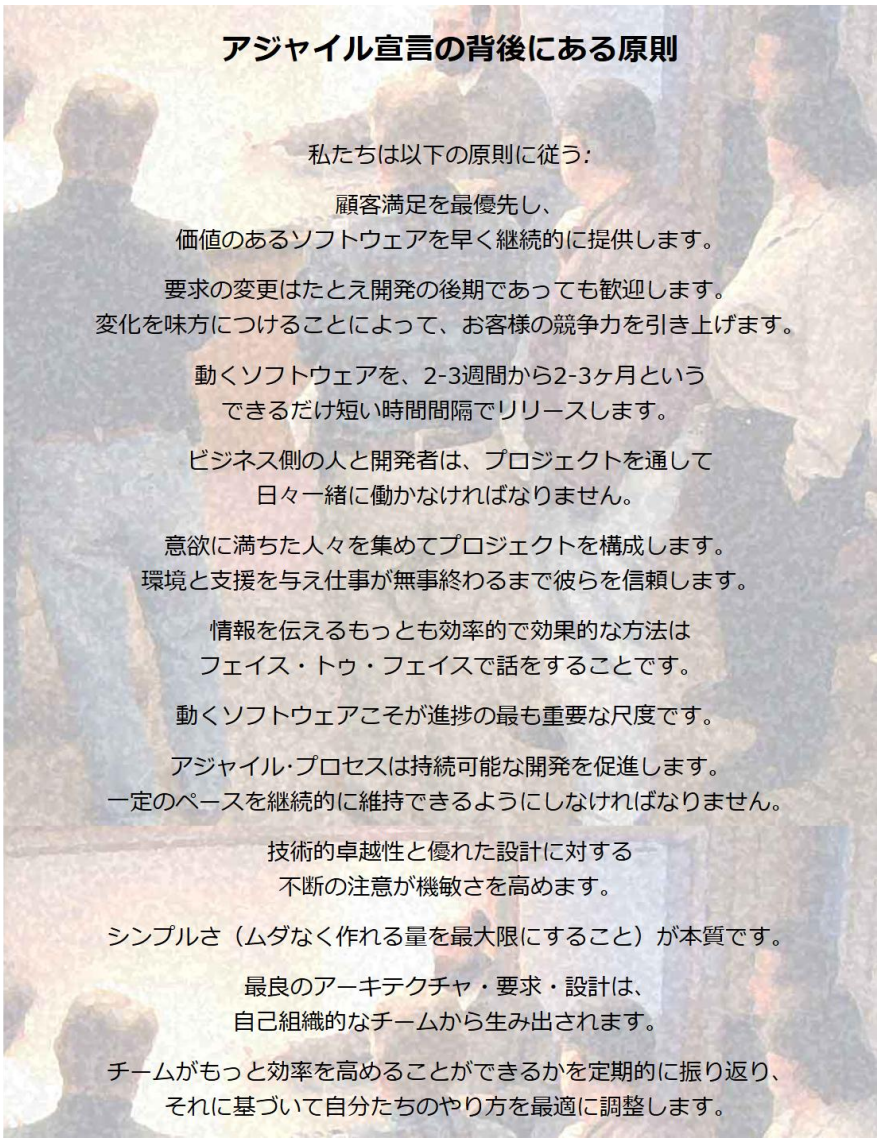
これを、ソフトウェア開発に適応させる形で生まれたのが「スクラム開発プロセス」である。これは、先にソフトウェア開発の現場でプロセスが出来上がり、それを抽象化する形でソフトウェア開発プロセスとしてモデル化された(Schwaber et al. 2003)。同時期に XP(extreme programming)など様々なプロセスが考案されたが、それらが２０

０１年に「アジャイルソフトウェア開発宣言」として括られ「アジャイル開発」と総称されるに至った。このような経緯からアジャイル開発自体ではプロセス定義はなされていない。スクラムや XP といったプロセスの要点を纏めた分類の名称である。公式に掲げられている定義は、「アジャイルソフトウェア開発宣言（図 5)」と「アジャイル宣言の背後にある原則（図 6)」である。しかし、きわめて抽象度が高いマインドセットと、その実現のための原則が規定されているに過ぎない。



アジャイルソフトウェア開発宣言(agilemanifesto.org 2001a)より転載

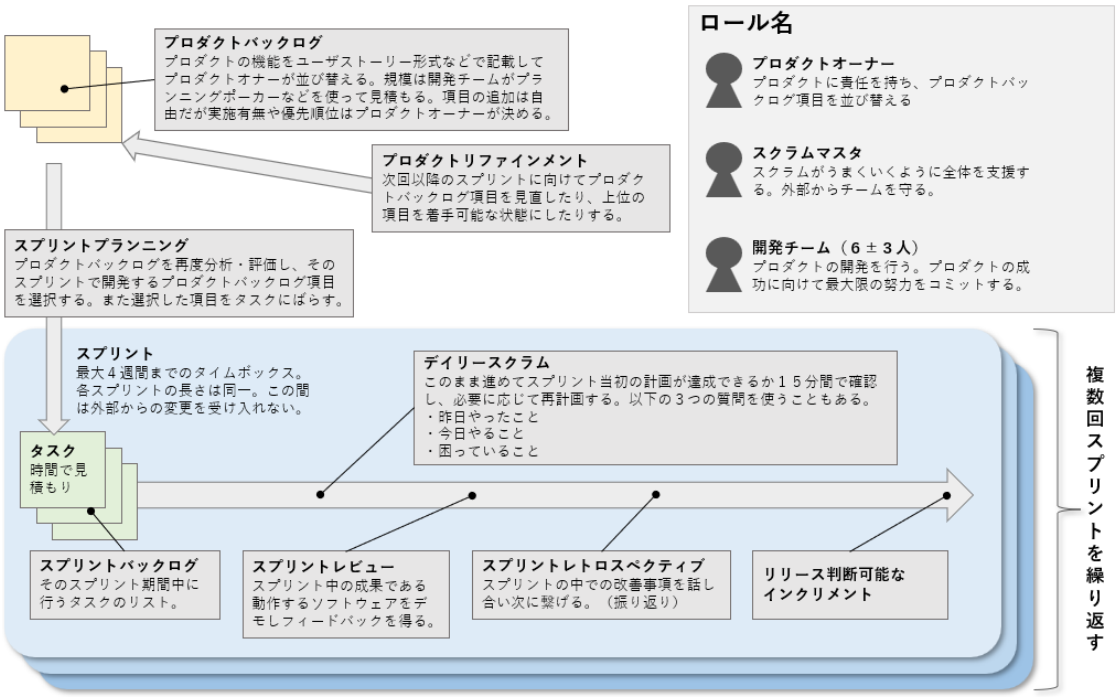
図 5 アジャイル開発宣言



アジャイル宣言の背後にある原則(agilemanifesto.org 2001b)より転載

図 6 アジャイル宣言の背後にある原則

一方で、具体的なプロセスについては、スクラムや XP といった別々のプロセスが規定されているが、きわめて具体的なロールやプロセスが提示されている。



アジャイル開発の進め方(情報処理推進機構 2020)　を参考に筆者作成

図 7 アジャイル開発のプロセス（スクラムの例）

この具体的な提示自体は、それぞれのプロセスが産まれたソフトウェア開発現場にとって最適化されたパッケージの状態であり、あまねく全てのソフトウェアにパッケージ状態で適用可能であるとは限らない。実際に、ビジネスや活動、文化などに合わせて、カスタマイズすることが必要とされている。しかし、なぜ？何を？カスタマイズするのか？といった難しさを抱えている。つまり、アジャイル開発の知識は、抽象度の高いコンセプトやマインドセット（図 5、図 6)、そして具体性高くパッケージ化された知識（図 7）の集合体を総称している状態（図 8）であると言える。

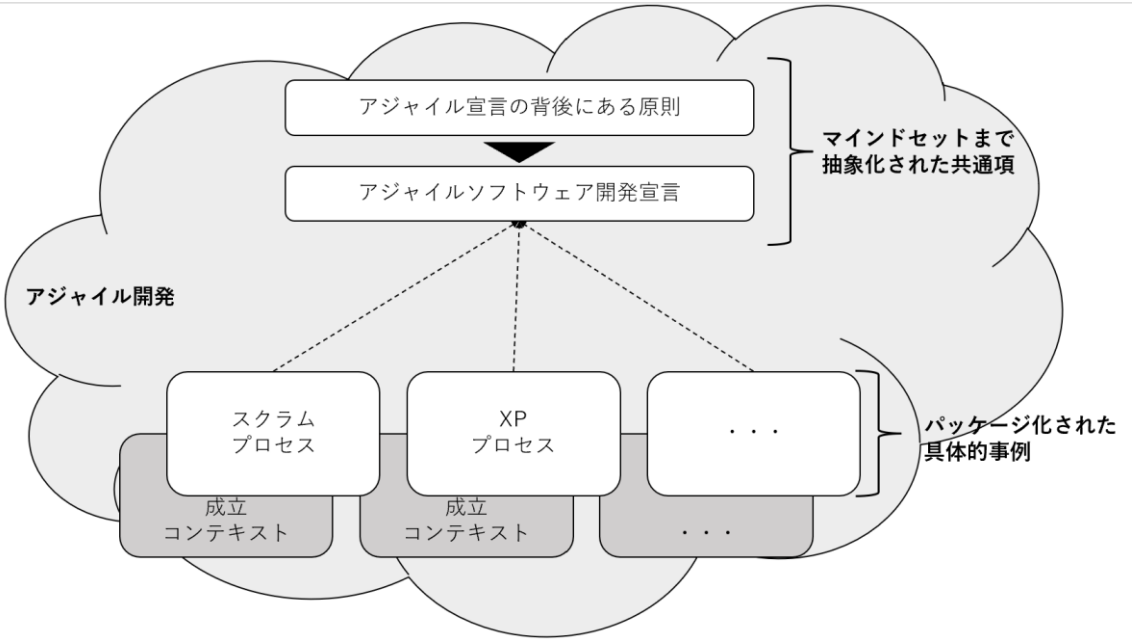


図 8 アジャイル開発の知識体系

結果的として、具体的手法を利用しようとすると、成立して具体的過ぎるため、コンテキストが合わずに適用困難なことが多い。そのため、それぞれの具体的手法では現場毎にカスタマイズした適用が推奨されているが、共通化された宣言に立ち返ってもマインドセットの提示のみで、何をカスタマイズしたらよいか分からないという状況が発生しがちである。

2.3 プログラムマネジメントに関する先行研究

プロジェクトマネジメントの知識体系は整備が進んでいる。特に広く活用されているのは、PMBOK（Project Management Body of Knowledge）ガイドであり、これは 1987 年にホワイトペーパーが米国 PM（Project Management）学会により策定され、1996 年に初版が発行された。

この PMBOK（図 9）は、PMI 倫理・職務規定と 5 つのプロセス・１０の知識エリアから構成された個別に利用可能なマトリックス構造の知識として整備されている。このように比較すると、アジャイル開発の知識体系の構造状態は、発展途上にあるということが分かる。

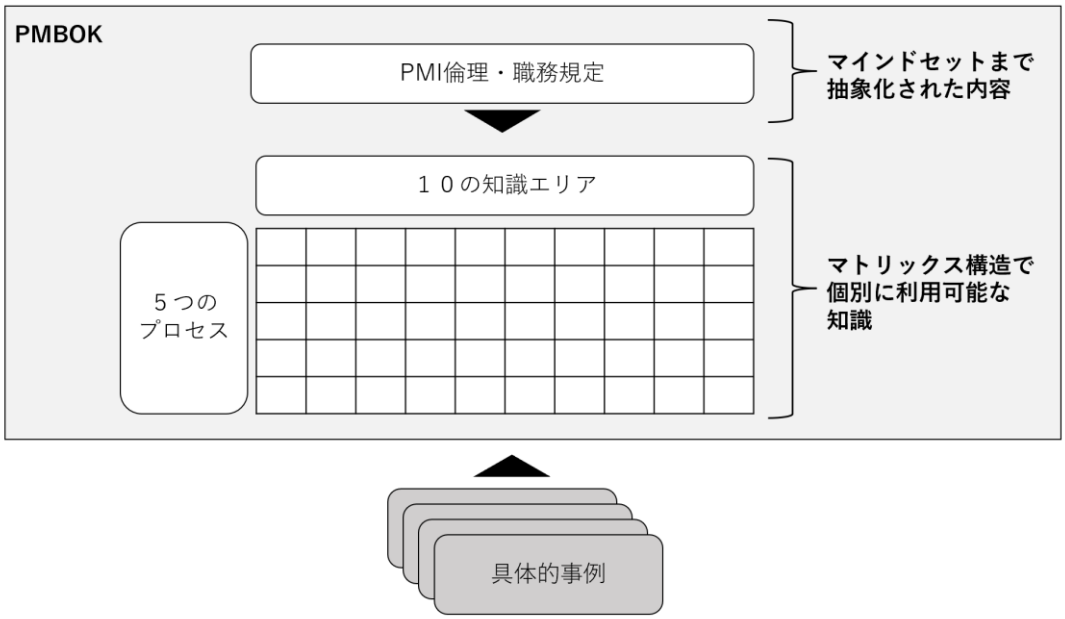


図 9 PMBOK の体系

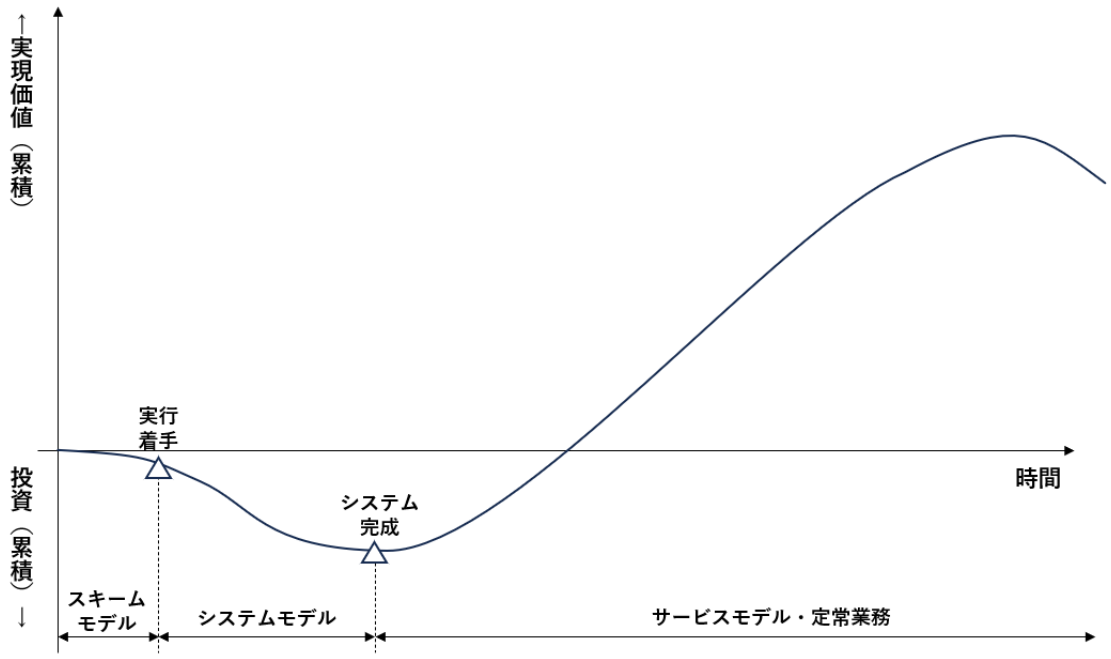
一方で、この整備された PMBOK においても、最新の第 7 版では大幅な改定が行われており、知識体系そのものの見直しが図られている。従来は、プロセスベースであった知識体系を、原理原則ベースの体系に見直している。これは、予測型ではなく適応型のプロジェクトにも活用可能にするためであると説明(プロジェクトマネジメント協会 2023)されている。これも、アジャイル開発同様に予測し計画することが困難なプロジェクトが増加していることが背景にあり、アジャイル開発を前提とするプロジェクトへの適用をより強めた結果[2]といえる。

プロジェクトマネジメントの知識体系には、複数のプロジェクトを組み合わせる概念として、プログラムマネジメントという概念が存在する。このプログラムマネジメントにフォーカスしているのが P2M 標準ガイドブックである。多数のテーマが重なる複合問題に全体的視点で対処するため、プロジェクトを有機的に組み合わせていくことがプログラムマネジメントであるとしている。また、プログラムマネジメントの目的は、下位のプロジェクトの自律性と、全体との有機的結合を成立させ、有用で安定した成果を効率的かつ公正に実現することにあるとしている。P2M のプログラムマネジメントで

[2] PMBOK においてアジャイル開発に言及したのは第 6 版が最初であり、第 7 版では強化したという位置づけである。

は、下位のプロジェクトをスキームモデル・システムモデル・サービスモデルという 3S モデルに分類されており、図 10 プログラムマネジメントにおける 3S モデルの様に価値を実現していくと整理されている。(日本プロジェクトマネジメント協会 2014)。

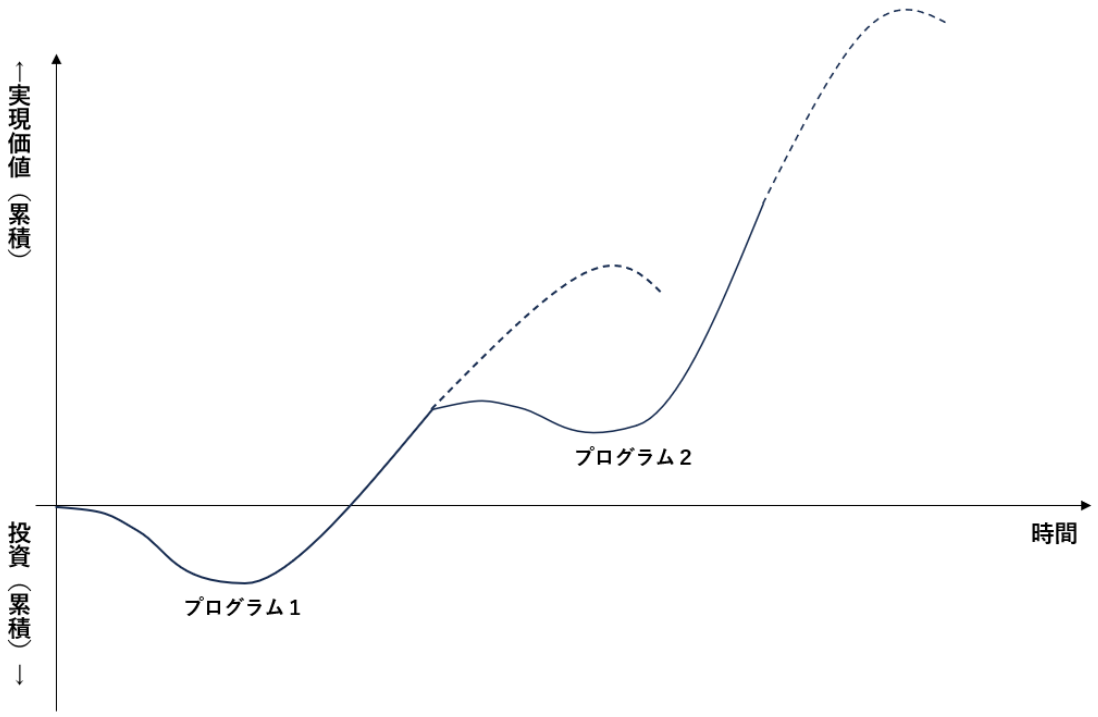
- スキームモデル：プログラムミッションの具体化・詳細化
- システムモデル：ミッションを実現するためのシステム（仕組み）づくり
- サービスモデル：作られたシステムを用いて、生産・販売・サービスなどを行い、最終的に価値を実現



P2M 標準ガイドブック(日本プロジェクトマネジメント協会 2014)を参考に筆者作成

図 10 プログラムマネジメントにおける 3S モデル

この様に、プログラムを通じて構想し、投資を経て、価値を高めるプロセスがモデル化されている。さらに、図 11 プログラムによる継続的な価値実現の様に、継続的に複数のプログラムを実行していく検討も行われている。この構造は、本研究のテーマである継続的な改善と類似な構造をとっているが、短期間のサイクリックな改善ではなく、多くのプロジェクトを包含した大掛かりなプログラムを継続的に行うことが意識されているため、スケールが全く異なる。



P2M 標準ガイドブック(日本プロジェクトマネジメント協会 2014)を参考に筆者作成

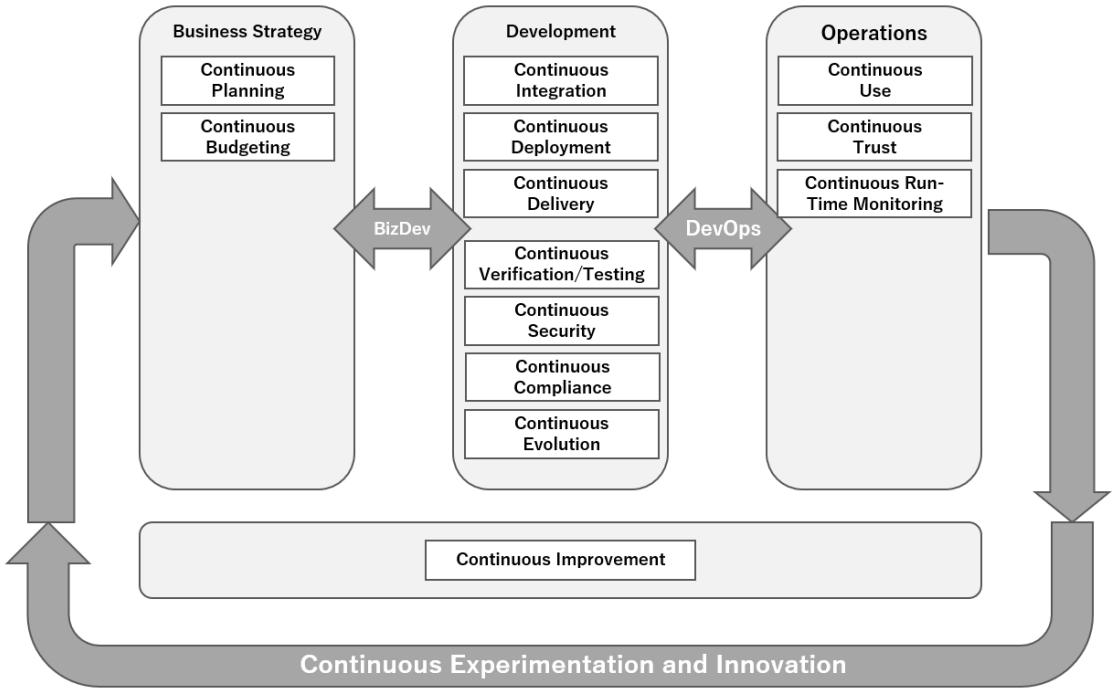
図 11 プログラムによる継続的な価値実現

また、スキームモデルでは、プログラムミッションの具体化・詳細化を行うと定義されている。つまり、プログラムのゴール状態を定義することが起点となる。この点も、1.1 の背景で触れたように、インターネットサービスにおけるソフトウェアは完成しないという特性を前提にすると、プログラムのゴールがない状態となる。このように、スケールサイズの違い、ゴールについての定義の違いなどがあり、連続的に価値を高めていくという大枠の発想は類似であるが、完全に一致するわけではない。

2.4 Continuous Software Engineering(CSE)に関する先行研究

この様に継続的に価値を積み上げ続けること自体は、いくつかの知識体系が存在する。ただし、アジャイル開発は現場ノウハウの集合知識という経緯から定義が曖昧であり、プログラムマネジメントは、知識体系は整備されているがスケールやゴール定義に違いがある。近年では、学術研究では類似分野を Continuous Software Engineering というテーマとして研究が展開されはじめている。Meta(旧社名 Facebook)社のソフトウェ

ア開発の事例研究を通じて、従来のソフトウェア工学(SE)がソフトウェアの完成を目的としているのに対して、インターネットサービスというソフトウェアは完成しない（”never be complete”）ことを前提としていると報告された(Feitelson, Frachtenberg, and Beck 2013)。このことは、従来の SE の目的そのものが置き換わることを示唆していた。一方で、アジャイル開発という概念が、単にソフトウェアの継続的なインテグレーションに焦点を当てており、本来的にはビジネス戦略・ソフトウェア開発・さらには財務・人事などを含んだ、より広範囲で全体的な視点で捉える必要があること。それが、エンタープライズアジャイル、DevOps、リーン全般と共通もしくは、統合された知識体系になり得ると指摘されている。さらに、それら全てを包含する形で Continuous Software Engineering（CSE）という視点が重要となり、その拡張のためには図 12 の様に経営学、情報システム学、ソフトウェア工学などの様々な分野に跨った知識統合が必要になると提言された(Fitzgerald and Stol 2017)。

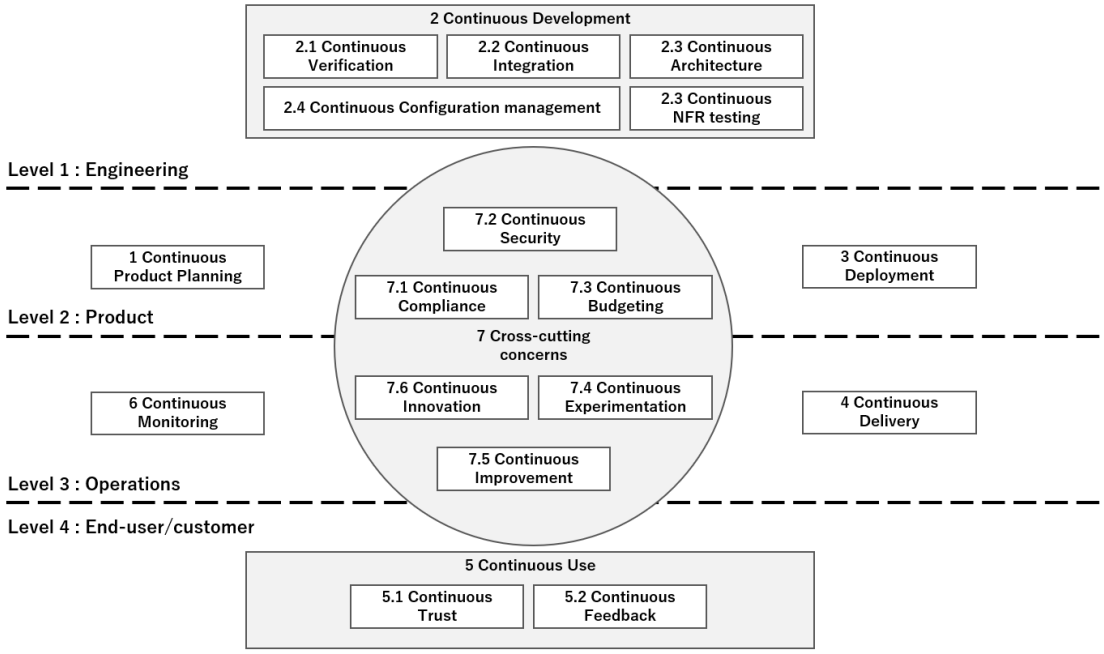


Continuous Software Engineering: A Roadmap and Agenda.(Fitzgerald and Stol 2017)より筆者模写

図 12 ビジネス、開発、オペレーション、イノベーションの統合的視点

CSE によって得られる、柔軟性・効率性・市場投入までの時間といった利点は多くの企業にとって有益であり、注目を集めている。しかし、その導入における根本的な問題

として、Should you do it?という課題が挙げられている。利点があることは分かっても、何らかのトレードオフの発生は予想され、大きな変革や、大きな投資を伴う可能性は高い。そのため、経営をはじめとした、様々な部門の協力は不可欠となる。そもそも、図 13 が示すように継続的なソフトウェア開発には、多様なプロセスとケイパビリティが求められる。そのため、企業組織全体を見直す必要が発生するかもしれない。これらの協力を引き出し、変革を進めるには、共通理解・共通目的とすることが可能となるクリアな利益の提示が不可欠であると報告されている(Klotins and Gorschek 2022; Klotins et al. 2022)。



Continuous Software Engineering in the Wild.(Klotins and Gorschek 2022)より筆者模写

図 13 CSE 全体像における利益と投資

また、CSE とデジタルトランスフォーメーションとの関係についても研究が行われている。デジタルトランスフォーメーションにおいても、新しいデジタル技術を採用しビジネス実装することを、継続的に行い続けることが重要であり、そのための基礎要件としてCSEが位置づくことになるとされている(Klotins and Peretz-Andersson 2022)。

2.5 Continuous Delivery / Experimentation に関する先行研究

CSE のテーマに、Continuous Delivery(CD)、Continuous Experimentation(CE)が存在する。もともと、Facebook 社のソフトウェア開発の事例調査から「完成しないソフトウェア開発」と報告された中で、２つの特徴が説明されていた。

１つ目は、リリースサイクルの短さである。Linux の新バージョンが２，３か月間隔でリリースされるのに対して、毎日リリースされている点が特徴とされていた。つまり、短いサイクルにすることを重視しているという点が特徴的とされていた。

２つ目は、AB テストを含むプロセスである。これは事前にユーザから要求を聞き出してソフトウェアに反映させるのではなく、利用者を A 群、B 群に分けて、様々な見た目や動作が異なるバリエーションを、それぞれの群に出し分けてデータから、どちらが望まれていたかを明らかにする手法である。実験的アプローチと説明されている。演繹法的に正解に近しいものを定め、それを作るのではなく。いろいろなバリエーションを用意して帰納法的に結果論で正解を見つけるという点が特徴的とされている(Feitelson, Frachtenberg, and Beck 2013)。

CSE という全体的な視点では、前者は CD に位置づけられ、後者は CE として位置付けられている(Klotins and Gorschek 2022)。2024 年現在、これらの手法は Facebook に特徴的なことではなくなっており、多くのインターネットサービス企業において取り組まれている。

これらに関する研究も活発に行われている。CD を行っていく障壁がアーキテクチャにあること(Schermann, Cito, Leitner, Zdun, et al. 2016)、それを容易にするためにはマイクロサービスアーキテクチャの採用が役立つという報告(Chen 2018)がされている。また、組織面でも障壁が存在することが報告(Leite et al. 2021)されている。一方で、CD の導入には、開発者・テスター・運用者を初めとする全ての人へのインセンティブ調整を行うために DevOps が必要であるとしている(Humble and Molesky 2011)。他にもソフトウェア品質が CD とトレードオフの関係にあることも報告されている(Schermann, Cito, Leitner, and Gall 2016)。この様に、CD 導入についての困難や解決策についての研究が多く展開されている。困難な企業が多く存在する一方で、既に導入

済みの企業も多数あり、それらの企業は CD のサイクル期間を、月次、週次、日次・・と短期化させ続けている(Feitelson, Frachtenberg, and Beck 2013)。つまり、デリバリサイクルの短期化にはメリットがあるためである。しかし、このメリットについての先行研究は進んでいない。短期化の利点を、ジャストインタイムの方が良いという説明(Rico, Sayani, and Sone 2009)であったり、競争環境において競争相手よりも早い方が良いという説明(Schermann, Cito, Leitner, Zdun, et al. 2016)といったように、漠然としたメリットへの言及に留まっている。

また、CE の視点では、実験的なリリースのスケジューリング方法に遺伝的アルゴリズムを活用するといった手法提案(Schermann and Leitner 2018)が行われている。CE 経験者へのサーベイにより、ソフトウェア自体のサービス品質の向上や、開発速度の向上などが見られたこと。一方で、失敗を受け入れ学びとする文化への変革が不可欠であること、インフラ整備などの技術課題、スキル習得などの人的課題が存在することが、報告されている(Auer et al. 2021)。

2.6 リアル・オプション理論とソフトウェア開発プロセスに関する先行研究

リアル・オプションの歴史は金融工学のオプション理論から始まり、事業評価・計画法としてのリアル・オプション、そして経営理論としてのリアル・オプションという流れで発展してきた(入山 2019)。この事業評価・計画法としてのリアル・オプション理論において、事業価値にプラスとして算入する 3 つの要素は、投資の意思決定を撤回できるか、投資の意思決定を先延ばしできるか、投資を行う事業の不確実性が大きいかという 3 点によって価値が算出される(刈屋 and 山本 2001)。

表 1 投資の意思決定を決める 3 つの要素の価値

(刈屋 and 山本 2001)リアル・オプション　新しい企業価値評価の技術　より引用

投資の意思決定を撤回できるか	できる できない	事業価値にプラス 事業価値に影響しない
投資の意思決定を先延ばしできるか否か	できる	事業価値にプラス

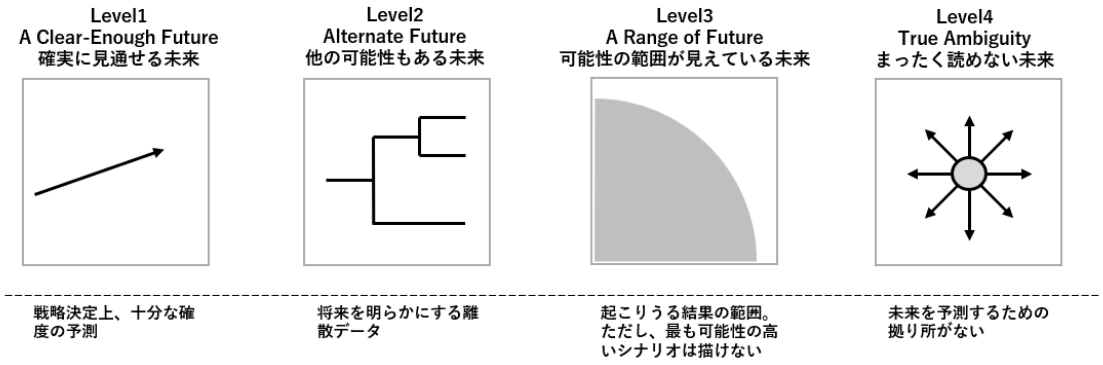
	できない	事業価値に影響しない
投資を行う事業の不確実性が大きいか否か	大きい 小さい	事業価値にプラス 事業価値に影響しない

また、具体的なオプションの型として代表的な 9 つを上げる。延期オプション（Option to Postpone）、拡大オプション（Expand Option）、縮小オプション（Option to Contract）、撤退オプション（Abandon Option）、段階オプション（Time to Build Option）、転用オプション（Option to Transfer）、事業の一時中断・再開オプション（Shutdown & Restart Option）、キャンセルオプション（Cancellation Option）、市場参入オプション（Market Entry Option）である(刈屋 and 山本 2001)。この様に多様なオプションの型が既に明らかになっており、実際に事業価値評価の際に利用されている。

これらのリアル・オプションの事業計画・評価法では、この不確実性を活かす発想をする。結果として得られるメリットは以下の 4 点となる。①ダウンサイドの幅を抑える、②アップサイドのチャンスを逃さない、③不確実性が高いほど、オプション価値が増大する、④学習効果　である(入山 2019)。

また、リアル・オプション理論が有効な不確実性についての研究も存在している。不確実性の分類として、内生的と外生的に分類をした場合にリアル・オプション理論が効くのは外生的であり、内生的には必ずしも有用ではないとされている(Cuypers and Xavier 2010)。なお、外生的とは、企業の自らの努力では低下させられない外部要因に起因する不確実性である。例えば災害などは典型的であるが、他にも政情や、テクノロジー進化や、人口動態や、競合他社の動向や、マーケットの流行なども含まれる。ネットサービスで言えば、GAFA を始めとするインターネットプラットフォームの変化やデバイスなども外生的な不確実性であると捉えることが可能である。別の不確実性の分類として Level1：A Clear-Enough Future（確実に見通せる未来）、Level2：Alternate Future（他の可能性もある未来）、Level3：A Range of Future（可能性の範囲が見えている未来）、Level4：True Ambiguity（まったく読めない未来）という分類（図 14）に

おいてリアル・オプション理論の有効性は Level2 に限る(Courtney, Kirkland, and Viguerie 2009)とされている。



不確実性自体の戦略思考(Courtney, Kirkland, and Viguerie 2009)　より筆者一部抜粋

図 14 不確実性の 4 種類

しかし、Level 2 に限られるのは「事業評価のリアル・オプション」であり、厳密な定量評価を求めない「経営理論のリアル・オプション」であれば、Level3 でも応用可能であるという意見もある(入山 2019)。

この様に、経営理論のリアル・オプションとは不確実性の高い状況下における行動のメカニズムや、とるべき戦略の姿を明らかにする経営理論となっている。先の不確実性の Level1～4 をソフトウェア開発に当てはめると、Level 1 はウォーターフォールモデルであり、Level2～4 はアジャイル開発がマッチするとも言える。そのため、アジャイル開発においてもリアル・オプション理論の適用が出来ないかを検討している研究(Racheva and Daneva 2010)が存在する。要求定義の工程でアジャイル開発における優先順位の付ける際にリアル・オプション理論の活用を提案する研究である。その研究では、実際のアジャイル開発チームに対するインタビューを通じた調査を行い、品質とスケジュールのトレードオフ、再利用の可能性、利用可能なリソース、同時進行のプロジェクトなどの判断の際に、定量的な比較ではないが暗黙的な思考としてリアル・オプション理論と同様の思考を行っていることを明らかにしている。

2.7 先行研究の課題と本研究の位置づけ

本章では、SE、アジャイル開発、CSE というソフトウェア工学研究の系譜についてレビューを行った。そこでは、ソフトウェアを完成させるための知識体系として、SE が構築された。一方で、インターネットサービスの事例研究を通じて、そこに完成しないソフトウェア開発が存在すること、そのために SE を CSE と拡張する必要性が提唱された。CSE では、SE 以上に組織学・経営学・情報システム学といった、様々な学術領域にまたがる学際的な研究の展開が必要であると指摘されている。CSE の中には、多様な研究テーマが存在している。本研究は、そのテーマの 1 つである Continuous Delivery（CD）というカテゴリに属する研究である。これは、継続的に機能を改善し続ける（デリバリ）ことを扱うテーマであり、CD 導入済みの企業群は、この改善するサイクルを年・月・週・日・・と短期化させている。この様な CD の導入に伴う、様々な困難について研究が多数展開されている。しかし、そもそも改善サイクルを短くするメリットについて言及はあるが、数理モデルで構造を明らかにする研究は管見の限りない。

本研究は、この改善サイクルを短くすることについてのメリットが、どの程度であるかを定量的に表現し、どの様なメカニズムからメリットが発生するのかを扱う。そもそもＣＤを導入するモチベーションを明確にすることで、先行研究の意義を更に深める関係にある。

第3章 インターネットサービスにおける改善プロセスの事例調査

3.1 事例調査の目的

本事例調査の目的は、インターネットサービスにおいて Continuous Delivery（CD）、Continuous Experimentation（CE）が実際に行われている事例から、本研究において様々なシミュレーションを行うための基礎数値を獲得することが目的である。具体的には、年間何件程度の Delivery と Experimentation が行われているのか？その勝率は何%なのか？といったことである。

3.2 事例調査：株式会社リクルートにおける継続的なサービス改善

3.2.1 インターネットサービスにおける改善のビジネス的意義

インターネットサービスにおいて、Conversion Rate(CVR)と呼ばれるサービスへの外部訪問者が購入[3]に至る割合は、非常に重視される。これは、様々なレバーによって変動するが、そのうちの 1 つが「サービスの使い勝手」である。このサービスの使い勝手とは、UI によって影響を受ける。使いやすければ CVR が向上し、使いにくければ CVR が低下する。これは、サービスの特性に大きく依存するが、一般的には数%といった小さい数字になることが多い。これは、訪問してから複数の画面を遷移する過程で離脱ユーザが発生し、最終的に購入に至るためである。そして、CVR が小さい数字であるため、CVR1%を CVR1.1%に 0.1pt 改善することによって大きなビジネスインパクトを産むことになる（図 15）。一方で、0.1pt 改悪するだけで大きなビジネスインパクトの棄損を産むといったことが起こりうる。インターネットサービスにおける UI 改善とは、この CVR を Key Performance Indicator(KPI)とする投資として行われることが多い。

[3] 登録であったり、予約であったりと、サービスの内容によって必ずしも購入ではない。しかし、ここでは分かりやすく購入と記載。

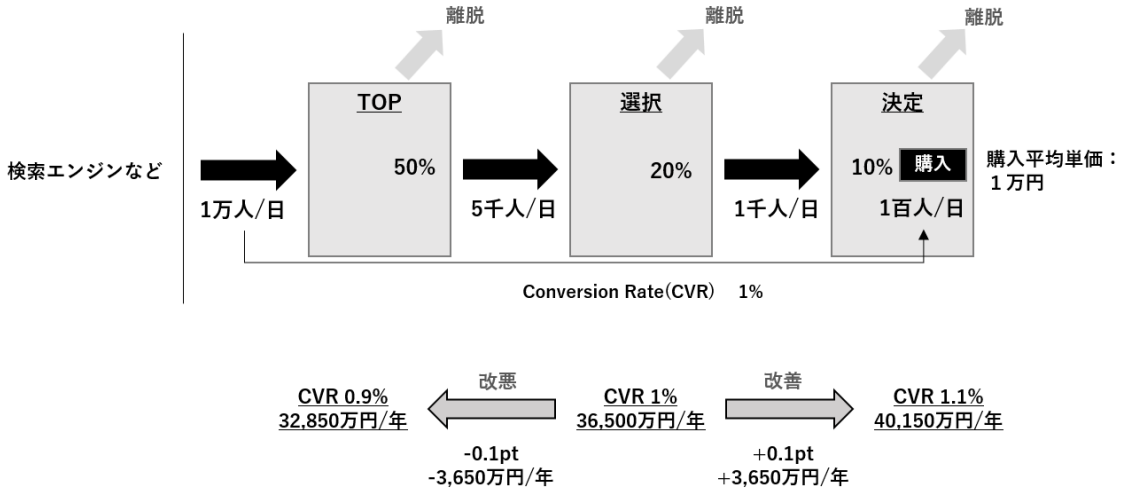


図 15 インターネットサービス改善のビジネス的意味

3.2.2 改善プロセス

インターネットサービスにおける UI 改善とは、一見変化点が分からないような改善で、どちらの方が利用者にとって使い勝手が良いのか事前に判断することも難しいような改善が多く行われる（図 16）。

例１：ボタンの並び順が異なる

例２：選択肢項目の表記が異なる

図 16 改善の事例

なお、改善後に利用者の行動を分析して、どちらの性能[4]が良いかを測定する。その結果から、ユーザ意識を読み取る。例えば、図 16 はアルバイト求人サービスの事例であるが、事例 1 の結果は右ケースの CVR が良かった。そこからアルバイトを探す時は職種ではなく場所（特に駅）から探す利用者が多く、一般的にスマートフォンの利用者はボタンを左から選択しやすいといった仮説を読み取り知識資産化していた。事例 2 の結果も右ケースの CVR が良かった。そこから、選択肢において選択前の状態では、選択の結果を想起する例示よりも、直接的な操作指示の方が選択肢であることを把握しやすいといった仮説を読み取っていた。これは、あくまで当該サービスに集まる利用者の性質であり、性質の異なるサービスであれば真逆の結果になることも有る。例えば、アルバイト求人ではなく中途社員求人であれば、駅よりも職種の方を先に決定するといったことが起こりうる。

4 インターネットサービスにおける性能とは、コンバージョンレートと呼ばれる指標が一般的である。EC サイトであればサイト訪問者の購入率であり、事例調査を行った求人サイトであれば、訪問者の応募率である。

この様な改善は、以下のプロセスで行われている（図 17）

① 期間と目標 KPI 数値が与えられる
- 活動の目的とゴールとして、ビジネスアウトカムが与えられる。具体的には、WEB サイトの CVR（Conversion Rate）を+X%上昇させるといった内容である。それ以降は、与えられた期間内で、②～⑦のプロセスをゴール達成まで回していく。

② ユーザ行動分析
- サイト内の利用者の行動を分析し、ビジネスアウトカムに寄与度が高いと思われる改善点を探す。具体例としては、検索結果が 0 件となった利用者は、離脱しやすいといったことを、サイトの挙動と行動の組み合わせから導く。

③ 改善内容の決定
- ②の結果から、課題仮説を立て、その解決策を検討し改善内容とする。この際に、利用者アンケートやインタビューなども併用して解決策を検討することもある。このタイミングで可能な限り開発施策の細分化を検討する。

④ テスト用設計開発
- あくまでテスト用と割り切り、本番稼働よりも低い品質かつ、メンテナンス性なども低い状態で、低コストで作成する。

⑤ AB テスト実施
- 現行の画面機能（オリジナル）に対して、１０%程度のトランザクションをテスト用モジュールに割り当て、期間（曜日や時間帯によって結果が変わってしまう）差異といったモジュール差異以外のノイズを、可能な限り除去した形で、オリジナルとの比較を行う。統計的な検定により、差異が有意となる利用数に到達する期間をテスト期間とする。もちろん、期間を延ばせば伸ばすほど利用数が増加して、有意差はつきやすくなるが 2 週間を上限として、その期間での利用数で有意差がつかない場合は、有意差なしとする。

⑥ 破棄
- ⑤の結果が、有意差ありでオリジナルに対して性能負け[5]している場合、もしくは有意差がつかなかった場合は、実施した施策を破棄する。

⑦ 本格実装
- ⑤の結果が、有意差ありでオリジナルに対して性能勝ちしている場合、実

[5] 3.2.1 の様に、CVR などの指標でオリジナルよりも指標が悪化した場合が性能負けとなる。逆に、指標が改善すれば、性能勝ちとなる。

施した施策を、本番稼働に耐えられる品質に高めたり、メンテナンス性を高めたりするための開発工程を経て、オリジナルモジュールとして組み込む。

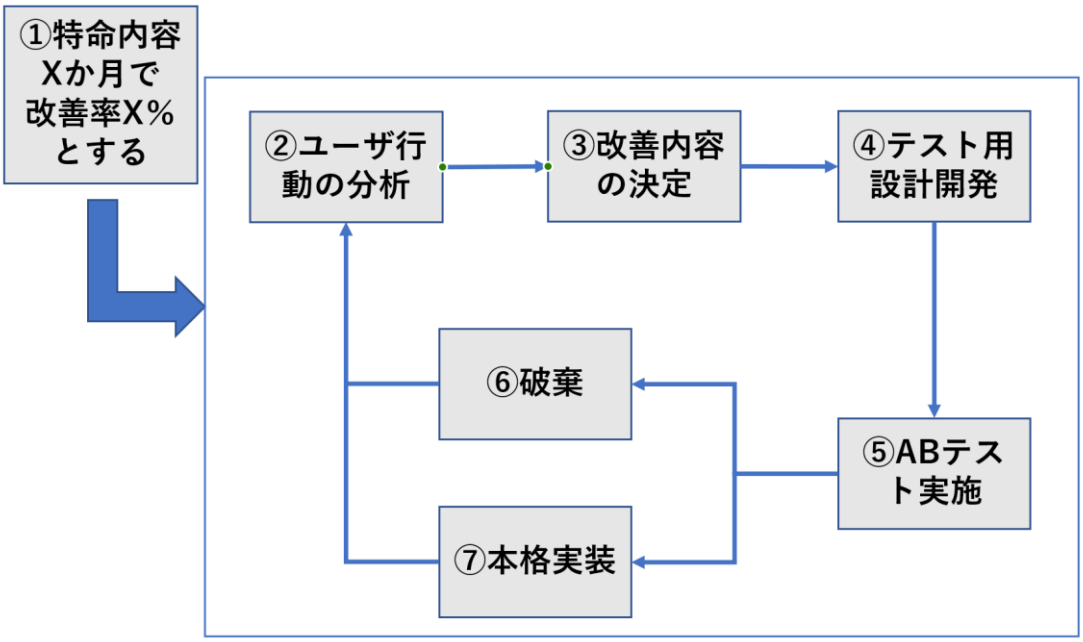


図 17 改善プロセス

3.2.3 成功と失敗の定義

なお、改善プロセス⑤において成功と失敗の評価は下記のように行われている。⑤において、AB テストの A:改善施策あり、B:改善施策なしで、同期間実際のサービス上で稼働させる、その結果を下記の評価で判定している。

成功：A と B の定量評価に有意差があり A＞B となる場合

失敗：A と B の定量評価に有意差なし or 有意差があり A＜B となる場合

有意差が認められない場合は、ビジネスアウトカムを生んでいないため失敗と分類している。またサービスに対して複雑性を積み上げるだけなので失敗として破棄する目的も含まれている。

3.2.4 調査結果

2 つのネットサービスにおける UI 改善の成功率について 2018 年 4 月～2021 年 7 月までの 3 年半の投資実績について調査を行った。サービス A では、月間平均で１７件の改善（調査全数：５４４件）を行っており、勝率の平均は３１％であった（図 18)。また、サービス B では、月間平均 9 件（調査全数：１７６件）の改善を行っており、勝率の平均は４７％であった（図 19)。

2 サービスの調査結果ではあるが、CD、CE が年間１００～２００回行われおり、CE の成功率は、３０～５０％程度であった。

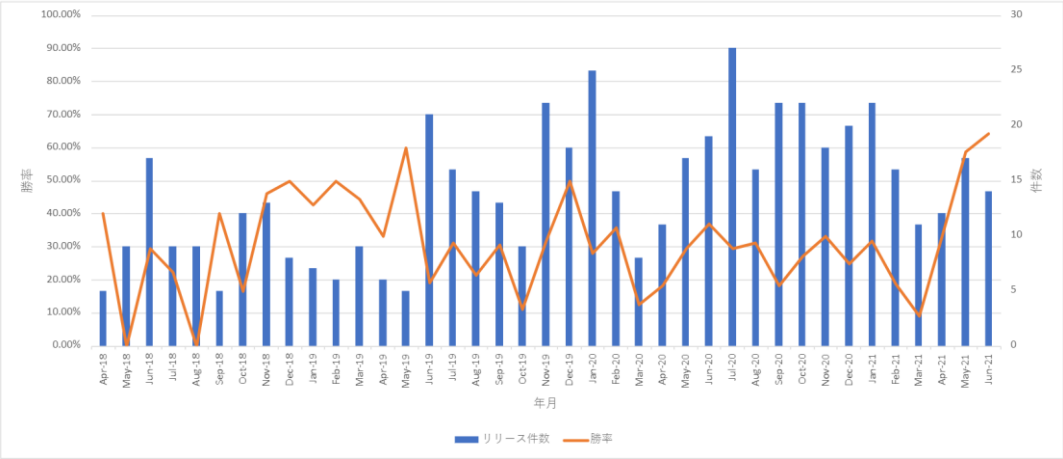


図 18 サービス A のリリース件数と勝率

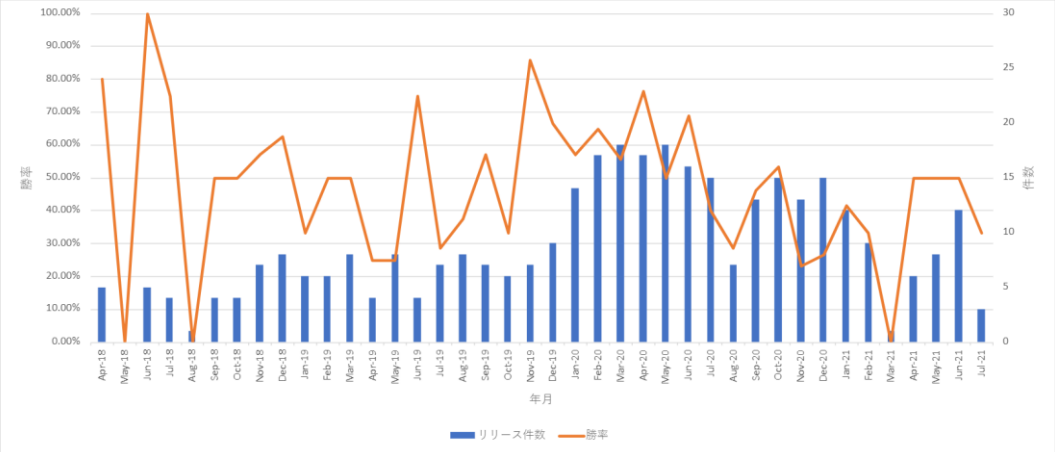


図 19 サービス B のリリース件数と勝率

3.3 考察

前提として、改善投資の活動であるため、当然改善されると見越した施策を実施している。その中には利用者からの要望に応えるケースや、利用者の操作状態から類推して改善すると思われる施策を検討している。また、4 年間の実績であり、担当者の変更なども多数発生していた。それにも関わらず、改善の成功率は４０％前後で推移しており、投資としては負ける可能性の方が高い状態であった。

この成否は、不特定多数の利用者による使いやすさという主観要素の強い要因によって成功と失敗が規定される。また、主観以前の問題として、使いやすさとは、スマートフォンの画面サイズや、利き手、使いなれている他ネットサービスに似ているといった、利用者一人一人の状況によっても左右されてしまう。そのため、全員が使いやすい、全員が使いにくいという状態になるのではなく、利用者の投票に近い形で、改善・改悪が決定する性質である。そのため、何を行えば改善になるのか？何を行うと改悪になるのかということを事前に知ることは困難であり、結果論に過ぎない。これが、投資としては不確実性が高い要因である。そのため、４０％前後の成功率は必然であり、常に改悪の発生を前提としており、成功時には本実装として永続化させるが、失敗時には切り戻し除去するということを行っている。

第4章 デリバリサイクル短期化による投資不確実性の対応メカニズム

4.1 メカニズム解明の目的と方法

本章では、デリバリサイクルを短期化させることと、不確実性の高い投資環境への対応についての関係を扱う。第 3 章の事例調査にもあったように、そもそも継続的に行っている改善投資自体の勝率が５０％以下である。一方で、インターネットサービス企業は、デリバリサイクルを短期化させることに努力している(Feitelson, Frachtenberg, and Beck 2013)。そのため、この低勝率な投資環境において、CD 期間を短期化させることの有効性、その程度について明らかにすることが目的である。そのための方法として、期待値計算による検討を行う。

4.2 計算モデルの設定

問題をシンプルにするために、下記の条件設定を行う。この際の業務上のオペレーションは、第 3 章にて記載の改善プロセスを踏襲する。

＜前提条件＞

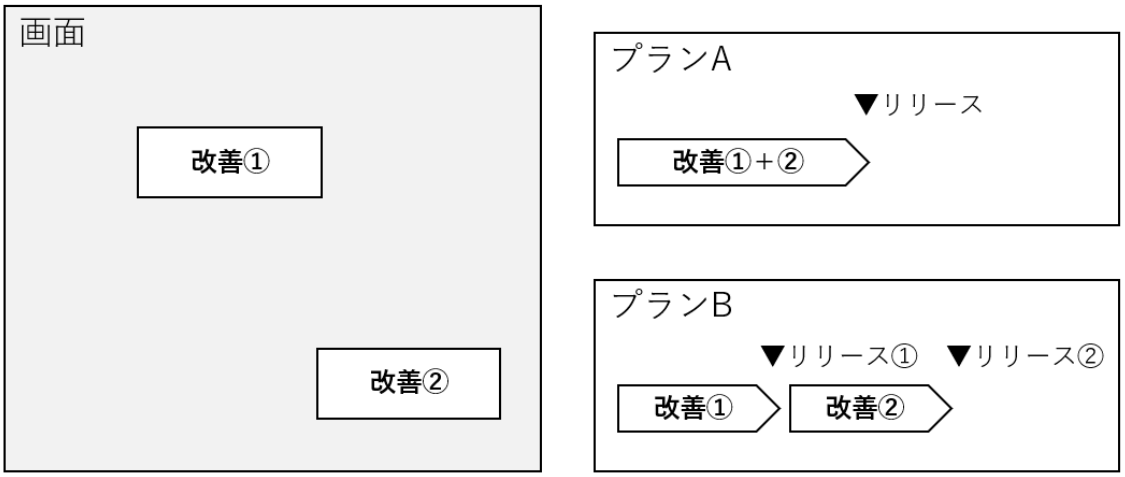
- １つの画面に２つ（改善①、改善②）の改善施策が存在する。
- どちらも開発着手が可能な状況にある。
- いずれの施策も、成功率は同程度である。
- いずれの施策も、改善時＋１０であり改悪時はー１０の影響を、ビジネス KPI に対して与える。
- 結果は改善か改悪のいずれかとなり、どちらになるかは成功率に依存する。
- 結果については、AB テストにて評価を行い、改善時は残し、改悪（効果なし）時は破棄する。

＜選択可能なプラン＞

- プラン A：改善施策①と②を１つのリリース単位に含め同時に実施する。

・　プラン B：改善施策①と②を別々とし 2 回のリリースにて実施する。

なお、プラン A は改善施策①②の結果が合算された状態で計測され、プラン B は①と②の結果が分離して計測される。同内容を（図 20）に図式化する。



＜前提条件＞
- １つの画面に２つ（改善①、改善②）の改善施策が存在する。
- どちらも開発着手が可能な状況にある。
- いずれの施策も、成功率は同程度である。
- いずれの施策も、改善時＋１０であり改悪時はー１０の影響を、ビジネスKPIに対して与える。
- 結果は改善か改悪のいずれかとなり、どちらになるかは成功率に依存する。
- 結果については、ABテストにて評価を行い、改善時は残し、改悪（効果なし）時は破棄する。

＜選択可能なプラン＞
- プランA：改善施策①と②を１つのリリース単位に含め同時に実施する。
- プランB：改善施策①と②を別々とし２回のリリースにて実施する。

なお、プランAは改善施策①②の結果が合算された状態で計測され、プランBは①と②の結果が分離して計測される。

図 20 計算モデルの設定

上記の条件設定の下で、ビジネス KPI に対する改善・改悪・成功率を元に期待値を計算する。ただし、通常の期待値計算と異なる点は、改悪となる計測結果が得られた場合は破棄が行われる点である。

4.3 計算結果

4.3.1 成功確率４０％と設定した場合の計算結果

第３章にて事例調査の結果、サービスＡの成功率は３１％であり、サービスＢの成功率は４７％であったため、想定モデルの改善施策の成功率を４０％と設定した。2件の施策が４０％の確率で成功となるため、両方成功は１６％(４０％×４０％)であり、片方成功は２４％（４０％×６０％、６０％×４０％)、両方失敗（６０％×６０％）は３６％となる。また、両方成功の場合は＋２０（＋１０＋１０）の利益、片方成功の場合は０（＋１０－１０、－１０＋１０）の利益、両方成功の場合は０（－１０－１０でー２０であるが破棄されるため０）の利益となる。結果として、プランＡの期待値は＋３．２となり、プラン Ｂ の期待値は＋８．０となった。以降も同様の計算を行うため、具体的な計算方法も図 21 に示す。プラン Ａ の場合は破棄の発生が１か所であるが、プランＢの場合は破棄の発生が４か所となる。結果として、期待値でいうと２．５倍の差が付く。あくまで期待値の差であるが、第３章にて調査した様な年間１００～２００回となる施策リリースを行う中では期待値に収斂し、実績数値としても大きな乖離が発生すると予想される。成功率４０％という状況であるからこそ、数少ないプラスを着実に拾う結果は非常に大きいと言える。

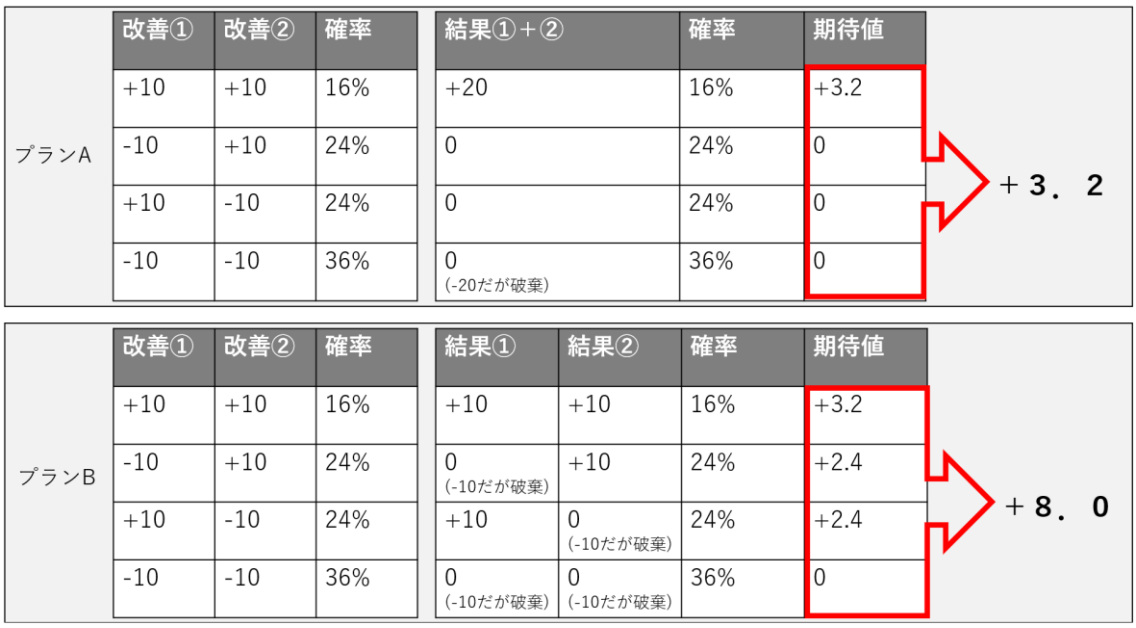


図 21 成功率４０％と設定した場合の計算結果

4.3.2 成功確率を変動させる計算

次に 4.3.1 で４０％とした成功率を０％から 100％まで、変動させる計算を行い表 2 成功率変動計算の結果（実数）の結果を得た。

表 2 成功率変動計算の結果（実数）

成功率	プランA 期待値	プランB 期待値	期待値 実数差
0%	0	0	0
10%	0.2	2	1.8
20%	0.8	4	3.2
30%	1.8	6	4.2
40%	3.2	8	4.8
50%	5	10	5
60%	7.2	12	4.8
70%	9.8	14	4.2
80%	12.8	16	3.2
90%	16.2	18	1.8
100%	20	20	0

これを、Y 軸に期待値として、X 軸を成功率とするグラフで表現すると図 22 となる。

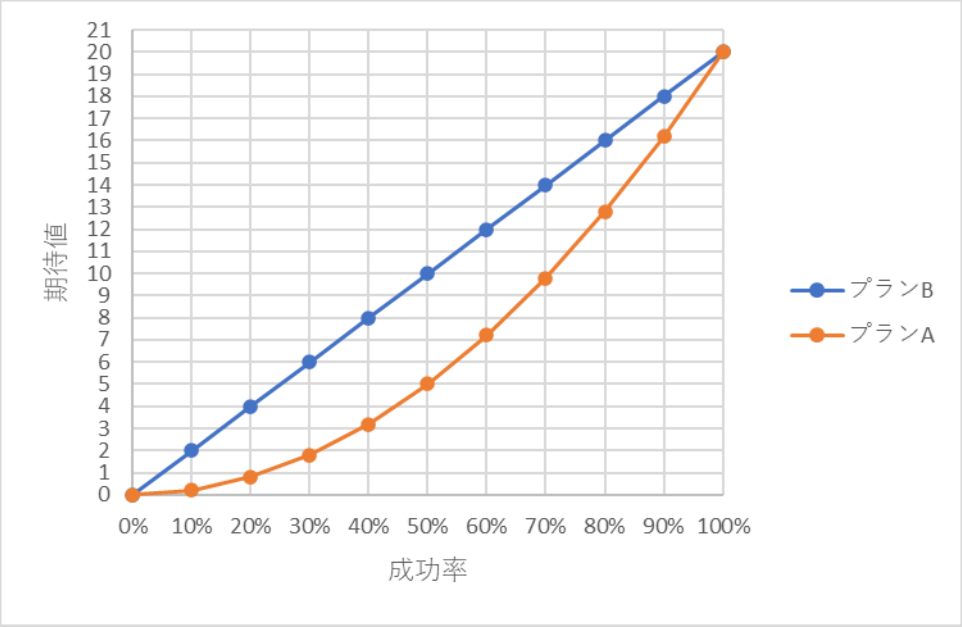


図 22 成功確率変動の計算結果（グラフ）

これらの結果より、下記のことが読み取れる。

- 成功率０％は必ず失敗するので差分が発生しない。
- 同様に成功率１００％も必ず成功するので差分が発生しない。
- 期待値の実数差でいうと、５０％の差分が最大となる。
- 期待値の比率でいうと、成功確率が低いほど最大化する。
- ただし、成功率０％であれば、確実に失敗するため期待値も０となる。
- プランＢは線形となり、プランＡは非線形となる。

興味深いのは、プランＢが線形となり、プランＡが非線形となる点である。つまり、プランＢの細分化が期待値を上げているのではなく、プランＡが期待値を下げているという解釈が正しいことを示唆する。これは、細分化することで全ての改善を拾い全ての改悪を破棄するため線形を取り、プラン A では改善と改悪の相殺が発生するため、改善の拾い漏れと改悪の破棄漏れが発生する為に期待値を下げる事になっている。これは、一見プラン A はリリース回数が少ない為、効率的であるという直感から乖離する結果と言える。

次に、このリリース回数が少ないことによるオーバーヘッドの影響について分析を行うため、プランＡとＢの期待値を比率で表 3 成功率を変動させた計算結果（比率）にて表す。

表 3 成功率を変動させた計算結果（比率）

成功率	プランA 期待値	プランB 期待値	プランA とB比率
0%	0	0	-
10%	0.2	2	10.0
20%	0.8	4	5.0
30%	1.8	6	3.3
40%	3.2	8	2.5
50%	5	10	2.0
60%	7.2	12	1.7
70%	9.8	14	1.4
80%	12.8	16	1.3
90%	16.2	18	1.1
100%	20	20	1.0

この結果から下記のことが読み取れる。

- 成功率 0 %はどちらも期待値 0 となるため比率にならない。
- 成功率 100%は期待値が同値になるため比率は 100%となる。
- 成功率が低ければ低いほど指数関数的に上昇する。
- 低成功率である 10%であれば、1000%の差となる。
- 高成功率である 90%であっても、111%の差となる。

プラン A・B のいずれであっても、実際の開発にかかるコストは変わらない為、分割によるオーバーヘッドはリリース回数増加による作業工数の増加となる。仮に高成功率である 90%という状況で有ったとしても、リリース作業の増加オーバーヘッドが全体の 10%以下であれば、分割する方が合理的であるといえる。なお、第 3 章で行った調査の事例では、リリース作業を自動化されており、作業量オーバーヘッドが発生しない状態であった。

4.3.3 改善・改悪の幅を変動させる計算結果

次に改善：＋１０、改悪：－１０という設定を変える計算を行った。まずは、どちらも１００倍して改善：＋１０００、改悪：－１０００の計算結果を 4.3.2 と同様のグラフで表現する（図 23)。

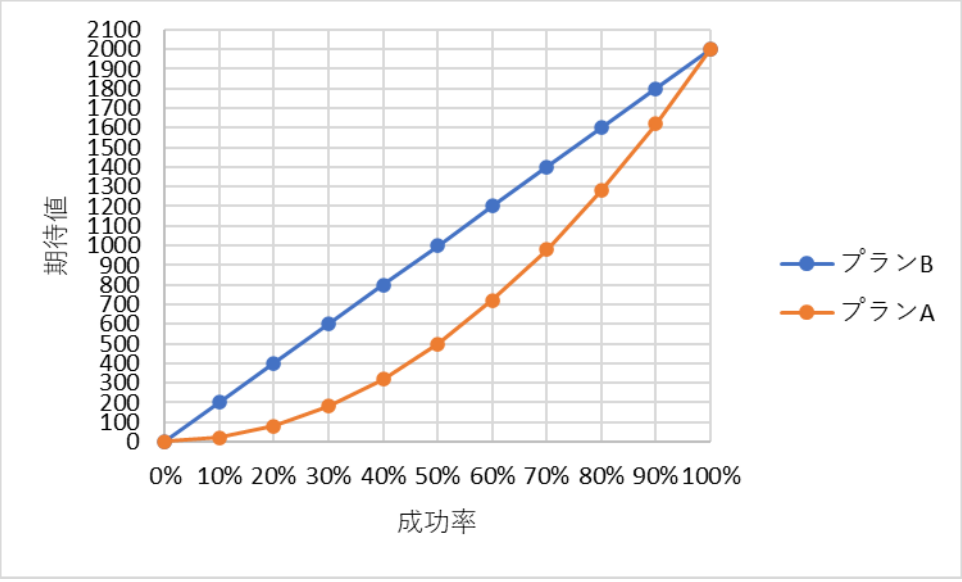


図 23 改善：+1000・改悪：-1000 の計算結果

実数差分は図 22 と変わるが、５０％が差分の頂点となる構造は変わらず、比率に関しても同じ結果となる。

次に、改善と改悪が非対称な数値なる場合の計算を行った。まずは、改善：＋１０００、改悪：－１０という、改善時は大きく改善され、改悪時の影響が小さい計算となる。投資においては、負けた場合の影響が小さいので有利な状況で有る計算結果（図 24）となる。

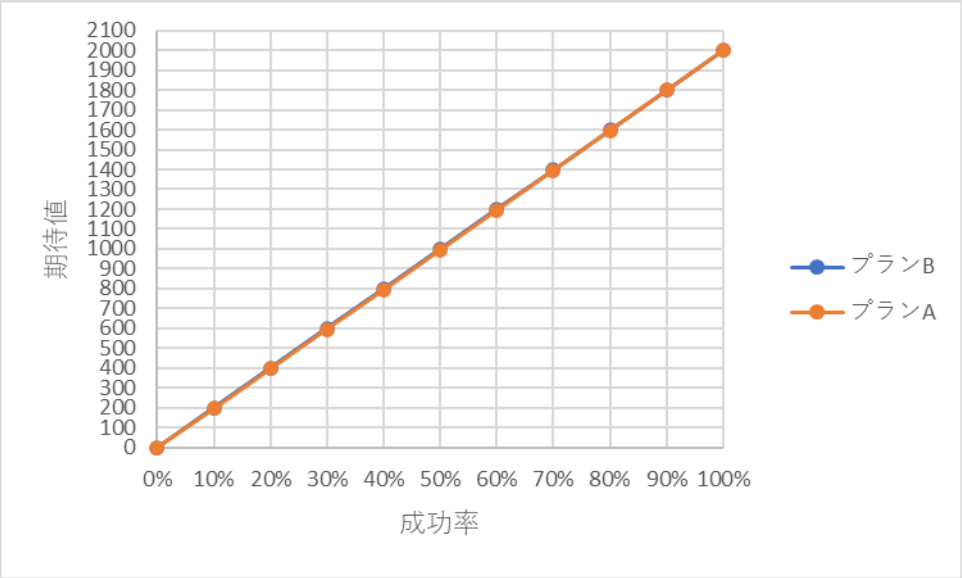


図 24 改善：+1000・改悪：-10 の計算結果

グラフのフォルムが大きく変わり、プラン A と B の差分が小さくなる（グラフの見た目からは小さすぎて分からないが差分は少量発生する）。これは、これまでの計算で発生していた改善と改悪の相殺が、改悪が非対称に小さいため相殺にならず、改悪の破棄漏れによる差分が付かない為である。

次に、改善：＋１０、改悪：－１０００という、改善時は小さい改善で、改悪時の影響が大きい設定である。投資においては、負けた場合の影響が大きいので非常に不利な状況の計算結果（図 25）となる。

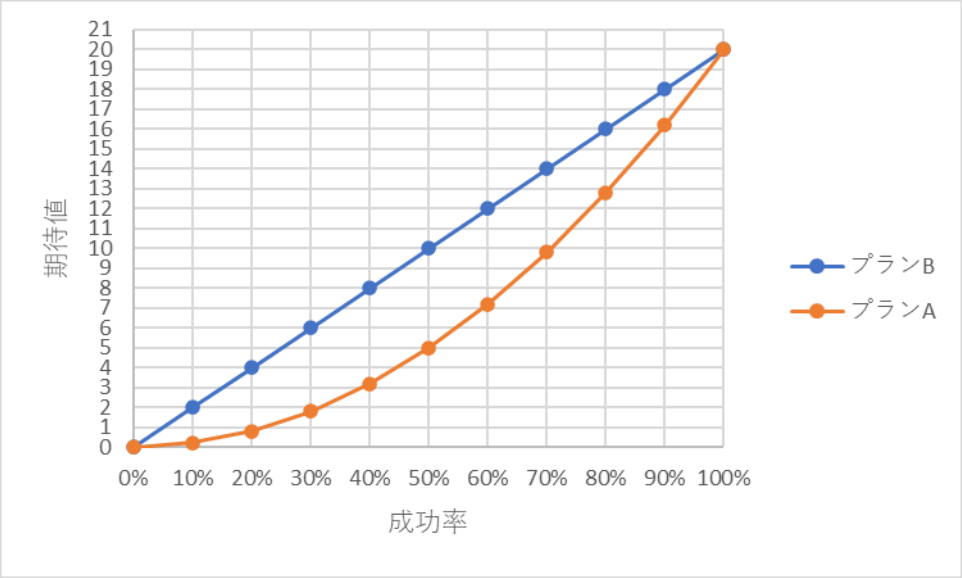


図 25 改善：+10・改悪：-1000 の計算結果

図 22 の改善：＋１０、改悪：－１０と同じグラフとなる。これは、改悪時には、どの様な大きさになろうとも破棄されてしまうので、意味が無いためである。なお、図 23 と比較すると、プラン A・B の期待値の比率が同じため、グラフのフォルムは同様であるが、実数差ではオーダーが大きく変わる点には注意が必要である。

ここまでの結果を期待値の実数差と比率の（表 4 改善・改悪幅計算結果まとめ）にまとめる。

表 4 改善・改悪幅計算結果まとめ

	改善+10 改悪-10				改善+10 改悪-1000				改善+1000 改悪-10				改善+1000 改悪-1000			
成功率	プランA	プランB	比率	実数差	プランA	プランB	比率	実数差	プランA	プランB	比率	実数差	プランA	プランB	比率	実数差
0%	0	0	-	0	0	0	-	0	0	0	-	0	0	0	-	0
10%	0.2	2	10.0	1.8	0.2	2	10.0	1.8	198.2	200	1.0	1.8	20	200	10.0	180
20%	0.8	4	5.0	3.2	0.8	4	5.0	3.2	396.8	400	1.0	3.2	80	400	5.0	320
30%	1.8	6	3.3	4.2	1.8	6	3.3	4.2	595.8	600	1.0	4.2	180	600	3.3	420
40%	3.2	8	2.5	4.8	3.2	8	2.5	4.8	795.2	800	1.0	4.8	320	800	2.5	480
50%	5	10	2.0	5	5	10	2.0	5	995	1000	1.0	5	500	1000	2.0	500
60%	7.2	12	1.7	4.8	7.2	12	1.7	4.8	1195.2	1200	1.0	4.8	720	1200	1.7	480
70%	9.8	14	1.4	4.2	9.8	14	1.4	4.2	1395.8	1400	1.0	4.2	980	1400	1.4	420
80%	12.8	16	1.3	3.2	12.8	16	1.3	3.2	1596.8	1600	1.0	3.2	1280	1600	1.3	320
90%	16.2	18	1.1	1.8	16.2	18	1.1	1.8	1798.2	1800	1.0	1.8	1620	1800	1.1	180
100%	20	20	1.0	0	20	20	1.0	0	2000	2000	1.0	0	2000	2000	1.0	0

この結果から下記のことが読み取れる。

- プラン B（分割）の期待値は、改善幅のみで決まる（改悪は漏れなく破棄されるため当然の結果である）。
- プラン A（同時）の期待値は、プラン B から改悪幅分だけ期待値を下げる（破棄から漏れる改悪の影響）形で決まる。ただし、下限は改善幅（ここを超えると、プラン A であっても結果がマイナスとなり破棄される）までとなる。

4.3.4 成功率と改悪幅を変動させる計算

ここまでの結果から、改善幅を+1000 に固定し、X 軸に成功率（0%-100%）、Y 軸に改悪幅（0-2000）として、Z 軸にプラン A・B の期待値差をプロットする計算（図 26）を行った。

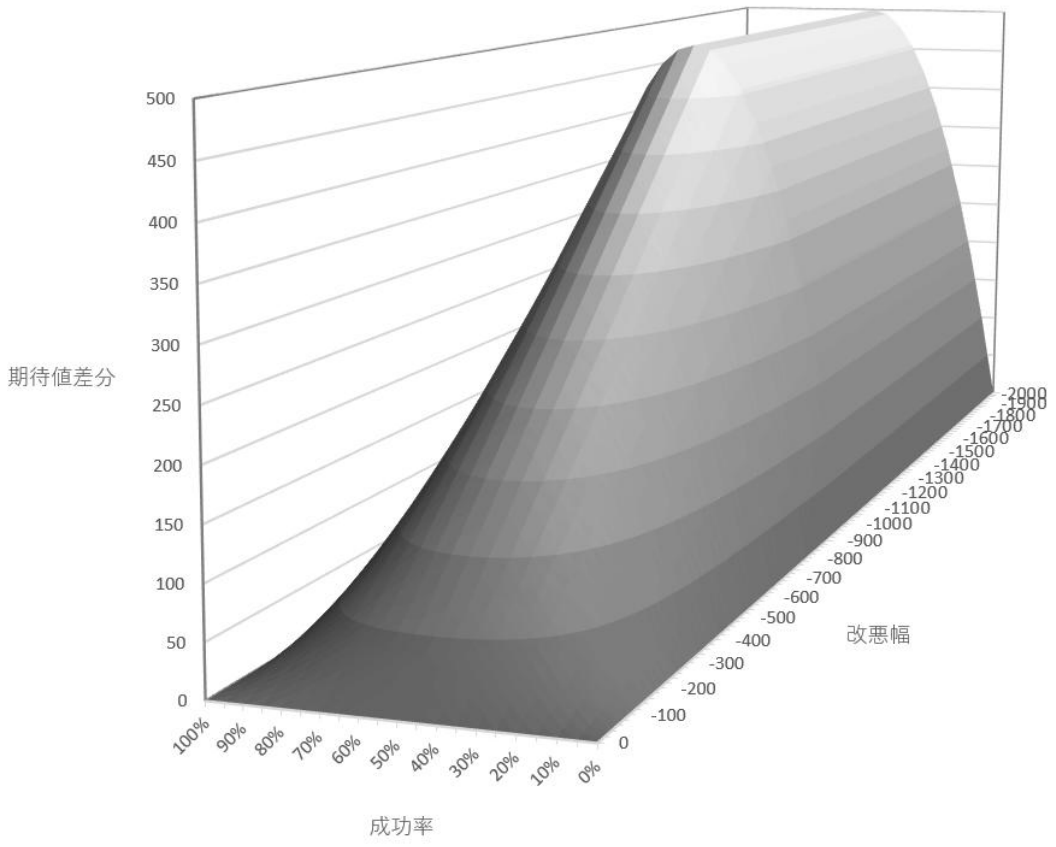


図 26 成功率と改悪幅の変動計算

この結果から、下記のことが確認できた。

- この計算結果における期待値差分とは、プラン B に対してプラン A の期待値が悪化する差分を指す。
- Y 軸（改悪幅）の断面で見ると、成功率 0%と 100%の期待値差分 0 となり、成功率５０%が頂点となる放物線を描く形になる。（改悪幅 0 は、その限りではない）
- X 軸（成功率）の断面で見ると改悪幅が大きくなるにつれて、期待値差分が大きくなる。ただし、固定した改善幅の 1000 と相殺される-1000 以降は、同じ形（すべて破棄される）となる。

X 軸の断面が複雑な形になるので、更に補足説明を行うため成功率 50%での断面を図 27 に表す。ここでは改善幅を 1000 と固定しているため、改悪幅<-1000 の間（グラフの前半）プラン A（同時）は改善と改悪が相殺しあった結果が改善となる。そのため、この期待値の低下は破棄されずに混入している改悪幅によって発生するということが分かる。しかし、改悪幅>-1000 の間（グラフの後半）プラン A（同時）は改善と改悪が相殺しあった結果が改悪に転じ破棄される。そのため、この期待値の低下は破棄されてしまった改善（この設定では+1000 と設定）が破棄されてしまったことにより発生するということが分かる。期待値低下の発生要因が、「破棄漏れした改悪施策」から「破棄に巻き込まれた改善施策」に変わるのは非常に興味深い。

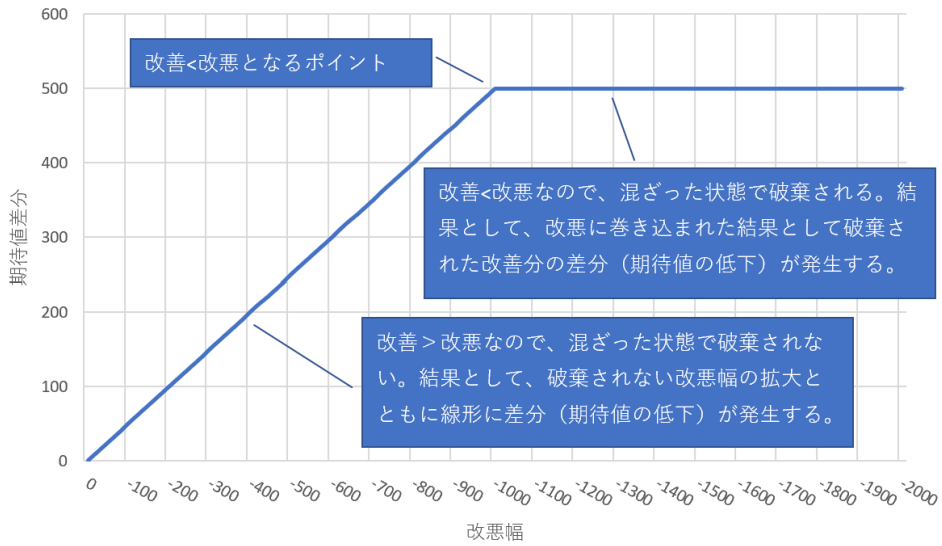


図 27 成功率 50%固定の改悪幅と期待値差分計算

4.4 メカニズムと定量的評価

今回の計算で分かったことは。開発施策の投資において０％＜成功率＜１００％であり、実施の結果として改悪（マイナスになる）が発生しうる場合において、計測可能な最小単位でリリースすることで、全ての改悪を測定・除去可能にするという点で投資効率が優位となるということだ。この優位性は、改悪の除去漏れが無くなることによって、ビジネスアウトカムに対して影響を及ぼす。正しくは、分離可能な開発施策を統合リリースすると、改悪の除去漏れが発生するためにビジネスアウトカムに対してマイナスの影響を生まれるというメカニズムによって差分が発生する。

このメカニズムは、リアル・オプション理論における撤退オプション（Option to Withdraw）で説明可能である。撤退オプションとは、将来の実現値が不確実な投資に対して、失敗が判明した場合に撤退を容易にしておくことで、損失を抑えることが可能となり、全体の収益期待値を向上させるという考え方である(刈屋 and 山本 2001)。

なお、定量的には、ビジネスアウトカムの期待値の差分が１０倍程度変化するため、その影響は大きい。また、成功率５０％で、この差分を最大となることが、今回の計算結果から明らかとなった。

4.5 数式によるモデル化

ここでは、数式によるモデル化を試みる。本章では、一般的な期待値計算を活用した様々な検討を行ってきたが、測定結果がマイナスになると除却し結果がゼロになる点が特殊である。数式化の方法は、改善・改悪のパターンに対して、確率と利益を設定し、それらを合計するという方法をとった。いわゆる期待値計算の手法である。

W:改善
L:改悪
K:改善確率
N:改善・改悪のビジネス数値

	プランA（同時）				プランB（分割）			
パターン	WW	WL	LW	LL	WW	WL	LW	LL
確率	K^2	$K(1-K)$	$K(1-K)$	$(1-K)^2$	K^2	$K(1-K)$	$K(1-K)$	$(1-K)^2$
利益	$2N$	0	0	0	$2N$	N	N	0
パターン期待値	$2NK^2$	0	0	0	$2NK^2$	$NK-NK^2$	$NK-NK^2$	0
プラン期待値	$2NK^2$				$2NK$			

図 28　2 案件の数式化

それぞれのプランで、改善・改悪の発生パターンに対して、発生確率と利益から期待値を計算する。プラン A では除却が発生しないため、改善・改悪が同時に発生するパターン（WL、LW）での利益が 0 となり、プラン B では改悪が除却されるため利益が N となる点が異なる。

結果として、プラン A（同時）の場合は$2NK^2$となり、プラン B（分割）の場合は$2NK$となった。これは、4.3.1 にあるとおり、プラン A は二次関数的な非線形となり、プラン B が線形のグラフとなることを示している。また、Kが確率であるため、それが 2 乗をとるプラン A は必ずプラン B よりも小さくなる（０％と１００％を除く）ことが分かる。特に確率Kが低いとプラン A では確率Kが 2 乗で効くため、期待値が大幅に低下することが分かる。図 29 にて、本章 4.3.1 で作成したグラフに数式を加筆した。

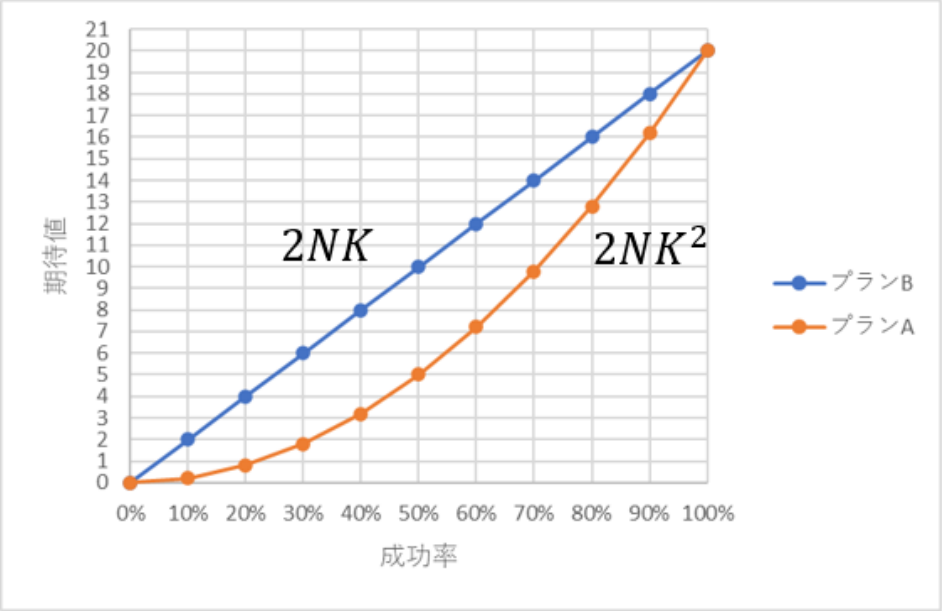


図 29 本章 4.3.1 のグラフに数式を加筆

4.6 本章のまとめ

本章では、デリバリサイクルを短期化させることと、不確実性の高い投資環境への対応についての関係を扱った。第 3 章の事例調査にもあったように、そもそも継続的に行っている改善投資自体の勝率が５０％以下である。一方で、インターネットサービス企業は、デリバリサイクルを短期化させることに努力している(Feitelson, Frachtenberg, and Beck 2013)。そのため、この低勝率な投資環境において、CD 期間を短期化させることの有効性、その程度について明らかにすることが目的であった。

そのため 4.2 で設定した計算モデルに対して、ビジネスアウトカムの期待値計算による検討を行った。この期待値計算の結果は、4.3 にあるとおり差分が１０倍程度変化する大きな影響を及ぼす優位性があることが明らかになった。また、成功率５０％で、この差分を最大となることが明らかとなった。

これは、開発施策の投資で改悪（マイナスになる）が発生しうる場合において、その成果を計測可能な最小単位でリリースすることで、全ての改悪を測定・除去可能にするという点で投資効率が優位となることが明らかになった。また、4.5 にて数理モデルに

よる検討を行ったところ、本来分割可能な施策を統合することで、成功・失敗の確率が2乗されることにより、期待値が低下することが示された。

このメカニズムは、リアル・オプション理論における撤退オプション（Option to Withdraw）で説明可能である。撤退オプションとは、将来の実現値が不確実な投資に対して、失敗が判明した場合に撤退を容易にしておくことで、損失を抑えることが可能となり、全体の収益期待値を向上させるという考え方である(刈屋 and 山本 2001)。

第5章 デリバリサイクル短期化による獲得利益の増加メカニズム

5.1 メカニズム解明の目的と方法

第 4 章では、AB テストと組み合わせたデリバリサイクルの短期化は、撤退オプションが作用し、週当たりに創出されるビジネスアウトカムの期待値が向上するという結果を得た。これは、完成することなく永遠に改善し続けるインターネットサービスにおいては、その一回の改善を切り出した視点である。現実には、これが永遠に続く。

そこで、本章では改善が複数回行われる状態についての検討を行う。この複数回行われる状態について、デリバリサイクルを変化させた場合の獲得利益変化についてシミュレーションする。実際には、第 4 章で扱ったように成功率を加味する必要があるが、それについては本章の結果を踏まえ第 6 章にて論じるため、本章では１００%成功するという前提にたって、デリバリサイクルの短期化、すなわち改善施策の小分け粒度によって、長期の時間軸で改善施策が生み出す利益が、どの様なメカニズムで変化するのかについて検討する。

5.2 シミュレーションモデルの設定

問題をシンプルにするために、下記のようなシミュレーションモデルを設定する。

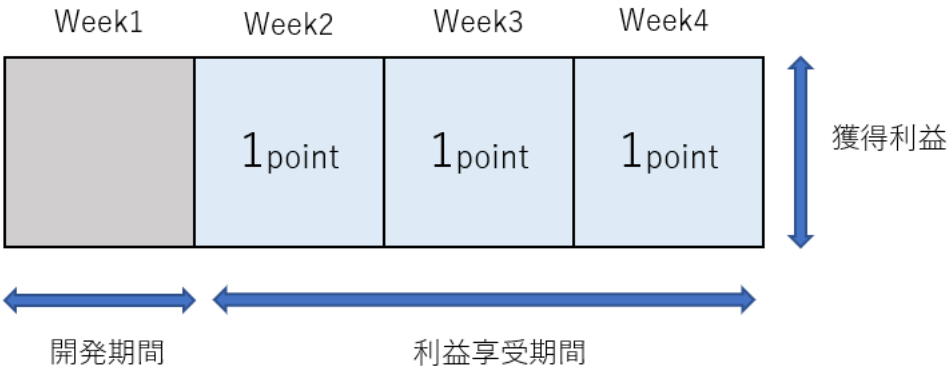


図 30 シミュレーションモデルの前提

シミュレーションモデルは以下の様に設定している。それを図 30 に示す。

- １名の開発者が１週間で開発可能な改善施策が無限に存在する。
- この改善施策は、リリースすると、毎週 1 Point の利益を生み出す。

この設定で、第 4 章におけるプラン A を表現すると図 31 となる。Item 1 と 2 を 2 週間かけて開発しリリースし、３週間目から毎週 2 Point の利益を生む。4 週目までに全部で 4 Point の利益を生み出す。

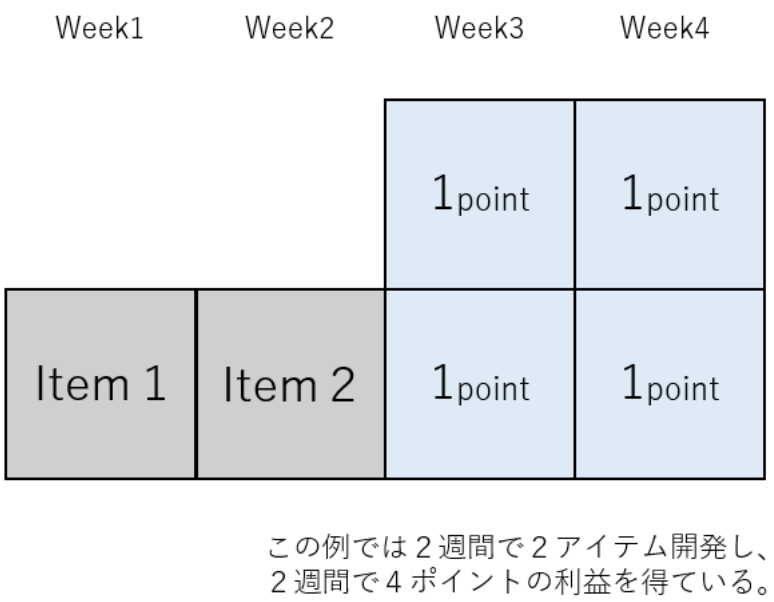


図 31 ２つの改善施策を統合して扱う

次に、第 4 章におけるプラン B を表現すると図 32 となる。Item 1 を 1 週目に開発してリリースする。２週目以降は毎週 1 Point ずつ利益を生み出す。Item2 は２週目に開発してリリースする。３週目以降は毎週 1 Point ずつ利益を生み出す。４週目までに全部で 5 Point の利益を生み出す。

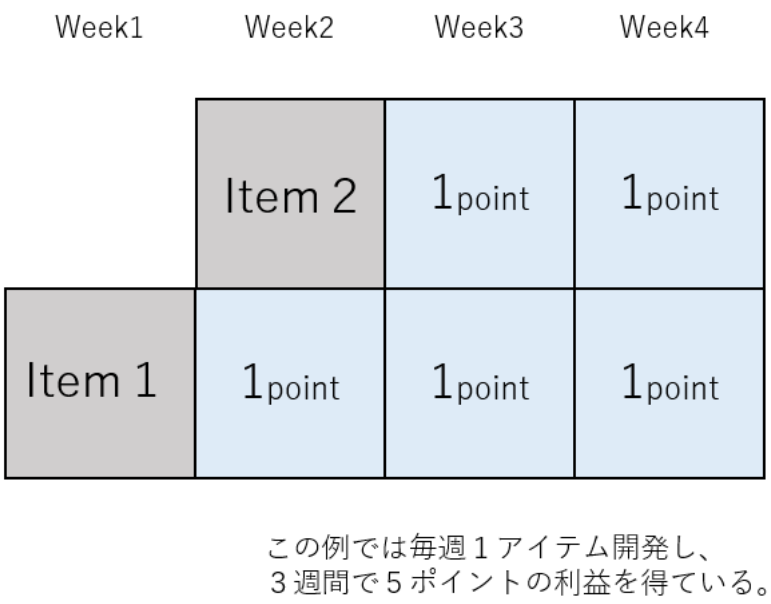


図 32 ２つの改善施策を分割してリリースする

この様に表現することとした。

次に、分割粒度を変更させるシミュレーションを１２週間で行うこととし、横軸に週数、縦軸に週当たり獲得利益をとったモデルを図 33 として表現した。毎週１つの改善施策を順にリリースした場合、１２週間で獲得する利益は６６Point となる。

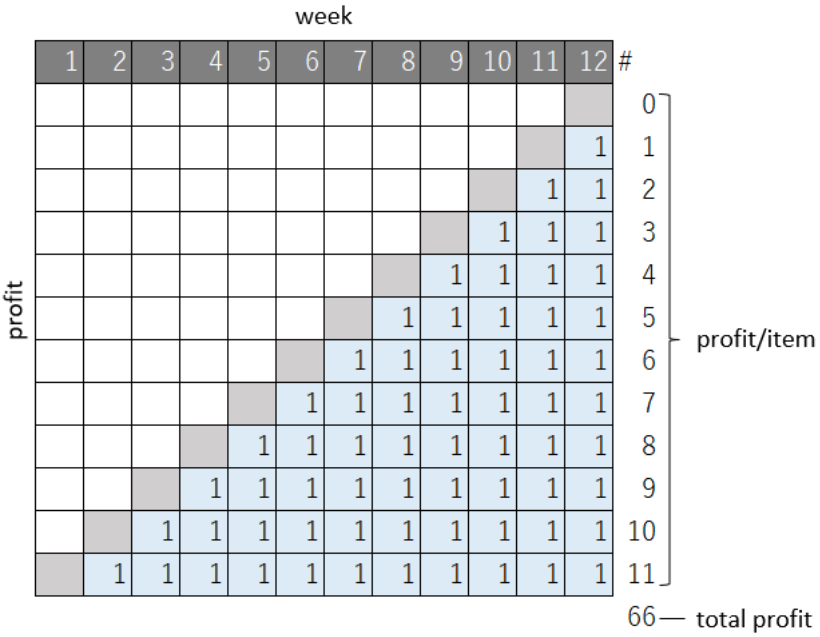


図 33 12 週間シミュレーション例

5.3 シミュレーション結果

デリバリサイクルを 1 週間（Case 1）～6 週間（Case 6）まで変動させた結果を図 34 に表す。

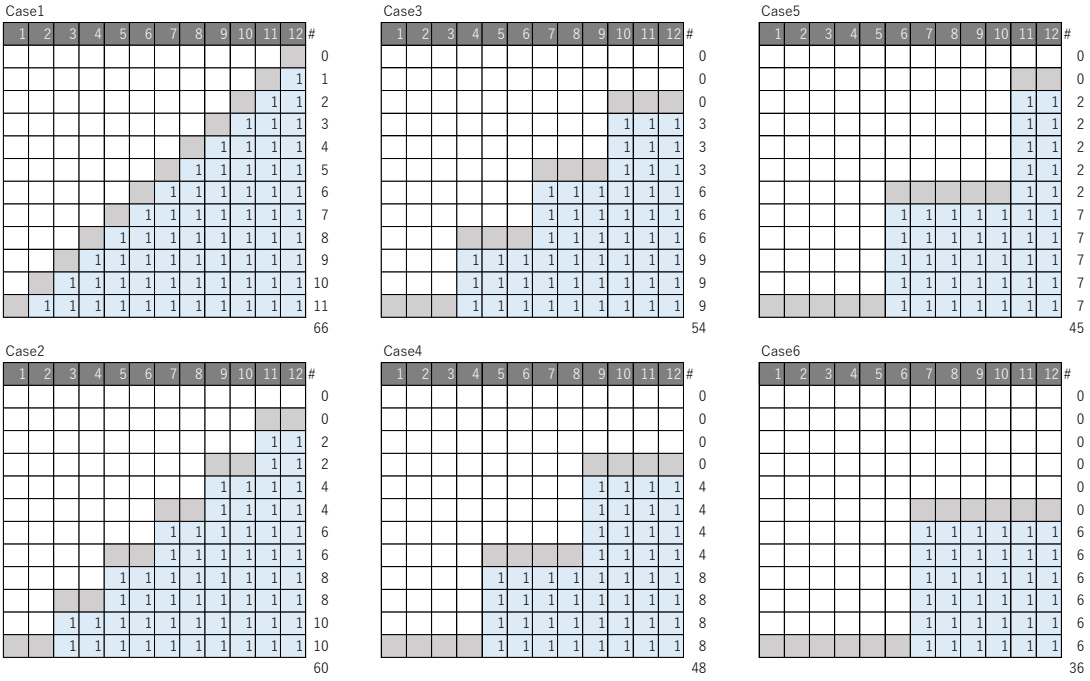


図 34 デリバリサイクル１～６週に変化させた結果

Case1～6 の利益の合計と Case1 に対する比率を（表 5 利益率評価）に示す。利益の欄は、図 34 の Case1～6 の結果を説明するものであり、Case1 が最も獲得利益が大きく、Case6 が最も獲得利益が小さい。したがって、Case1 を 100%とし、Case2～6 の利益を「利益率」として%で示した。

表 5 利益率評価

	Profit	Profitability
Case1	66	100%
Case2	60	91%
Case3	54	82%
Case4	48	73%
Case5	45	68%
Case6	36	55%

この結果から以下のことがわかった。

- 開発作業の総量はどの Case も同じである（グレーのセルの総量は 12)。
- 利益が最も大きいのは Case1 である。
- 最も利益が少ないのは Case6 である。
- その差は Case1 より Case6 の方が 45%小さい。
- デリバリサイクルの期間が短いほど、より多くの利益が得られる。
- デリバリサイクルの期間が長ければ長いほど、得られる利益は少なくなる。

5.4 メカニズムと定量的評価

ここでは、Case 1 と Case 6 で４５%の差が発生した要因について検討するが、このシミュレーションモデルの図示では非常に分かりやすく差分の発生個所を図 35 に赤で示す。

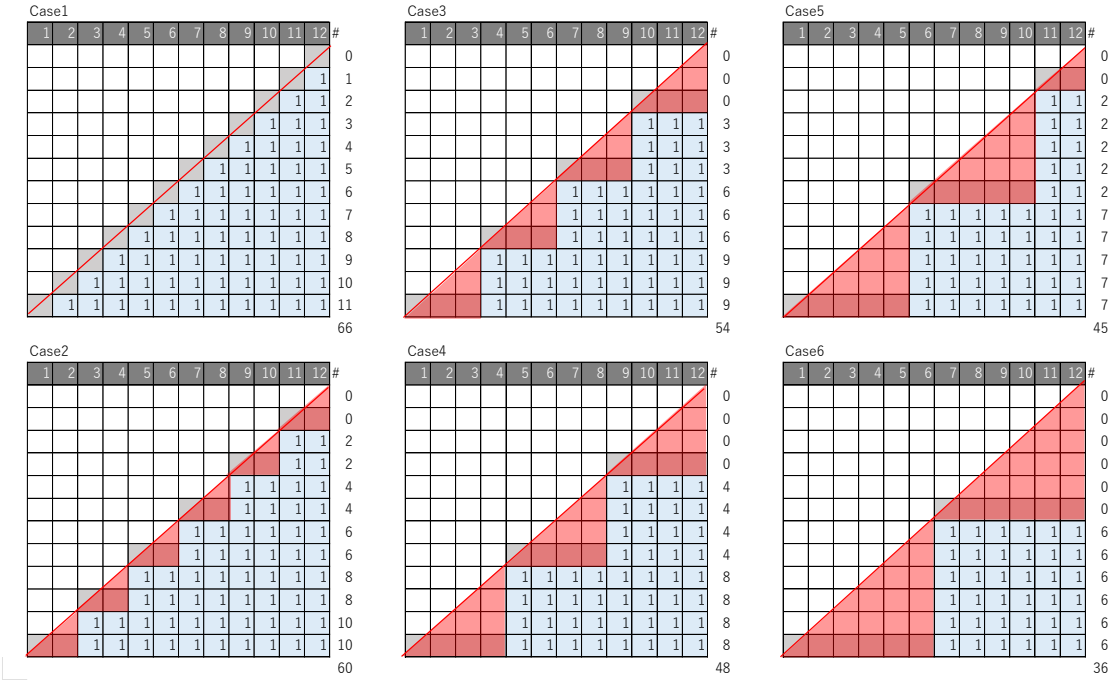


図 35 利益差分の発生個所

メカニズム自体は非常にシンプルで、Case 1 の様にデリバリサイクルを短くし、細切

れに改善施策をリリースすることで獲得できていた利益発生セルが、Case 6 のようにサイクルを長くし統合リリースすることで獲得できなくなっていき、赤いセルが拡大していく。言い換えれば、これはデリバリサイクルを短くすると利益獲得の機会損失の発生が最小化される、もしくは、デリバリサイクルを長くすると利益獲得の機会損失が拡大していくと言える。そして、その差分は本シミュレーションでは４５％というサイズになるということが分かった。つまり、6 週間サイクルでリリースをすると、1 週間サイクルでリリースしていくよりも、獲得利益が１２週間で半分程度になってしまうという事である。

なお、図 35 のとおり、どの Case であっても最後のリリース分は使用されない。そのため、シミュレーション設定した期間が１３週となる場合は、表 5 の生産性は変動することになる。しかし、デリバリサイクルが長くなればなるほど、機会損失が拡大することは変わりがない。

5.5 数式によるモデル化

ここでは、数式によるモデル化を試みる。5.4 のメカニズムを、そのまま数式化したものが図 36　階段状の価値積み上げ数式である。

W：計算対象の期間
C：リリースサイクル
N：創出利益

$$N\left(\frac{W^2}{2}-\frac{WC}{2}\right)$$

図 36　階段状の価値積み上げ数式

リリースサイクルCが大きい（頻度が低い）程、式全体における減少効果である$\frac{WC}{2}$が大きくなるため創出利益は減少する。また、W^2に比例する形で利益が増加するため、期間が長ければ長い程、創出利益のポテンシャルが高まる。一方、式全体における減少効果である$\frac{WC}{2}$はWと比例して増加するため、期間が長ければリリースサイクルCの影響が大きくなるとも言える。

現実世界では、永遠に続く改善はWが∞となることを意味している。いわば、サービスの終了や撤退がWとなる。つまり、結果論でしかない。そのため、リリースサイクルCを小さく保ち続けることで、$\frac{wc}{2}$という機会損失を最小化し続けることが重要であり、他にレバーがないともいえる。

5.6 本章のまとめ

第 4 章では、AB テストと組み合わせたデリバリサイクルの短期化は、撤退オプションが作用し、週当たりに創出されるビジネスアウトカムの期待値が向上するという結果を得た。これは、完成することなく永遠に改善し続けるインターネットサービスにおいては、その一回の改善を切り出した視点である。現実には、これが永遠に続くため、本章では、改善が複数回行われる状態についての検討を行った。

5.2 で設定した、改善が複数回行われる状態で、デリバリサイクルを 1 週～6 週まで変化させるシミュレーションを行った。結果として、1 週に対して 6 週は 4 5 %の悪化が発生することが分かった。これは、統合するために本来リリース可能だった時期が後ろ倒れることによる利益獲得の機会損失が発生した結果であることが明らかになった。

これを数理モデルによって検討した。結果、サービス継続期間の長さが利益創出のポテンシャルを高めることが分かった。これは、単純にサービスを継続しているということは何かしらのビジネスアウトカムが発生しているのは前提であり、その期間が長ければ獲得利益が増大化するということであり、自明な結果である。一方で、その獲得利益を毀損するのは、デリバリサイクルの長さであった。リリースサイクルの長さに比例する形で機会損失が拡大することが明らかになった。

第6章 デリバリサイクル短期化による統合的なシミュレーション

6.1 メカニズム解明の目的と方法

第 3 章では、インターネットサービスの改善というのは継続的に行われ続けており、そのサイクルを短くする努力が行われていること、一方で改善の勝率というのは高くなく成功率５０％を切るような不確実性の高い環境であることを明らかにした。第 4 章では、改善施策を小分けにしてデリバリサイクルを短くすることで、低い成功率であっても利益期待値を向上させられることを明らかにした。第 5 章では、同じくデリバリサイクルを短くすることで獲得利益の機会損失を低下させられることを明らかにした。

本章では、第 4 章・第 5 章を組み合わせたシミュレーションを行う。それぞれの章ではメカニズムを明らかにするために、敢えてシンプルなシミュレーション設定としたが、現実には同時に発生しており、デリバリサイクルを短くする効能というのは、両者の合算によって表現されるべきである。一方で、第 5 章の階段型の様なシミュレーションモデルに、第 4 章のような成功失敗を設定すると、図 37 の様に、どの改善アイテムを成功もしくは失敗としたかによって恣意的なシミュレーションとなってしまう。

そこで、本章では改善アイテムの成功・失敗がランダムに決定されるシミュレーションプログラムを作成し、その試行回数を増やすことで、両者のメカニズムが働いた際のビジネスアウトカムへの影響値を定量的に明らかにしていく。

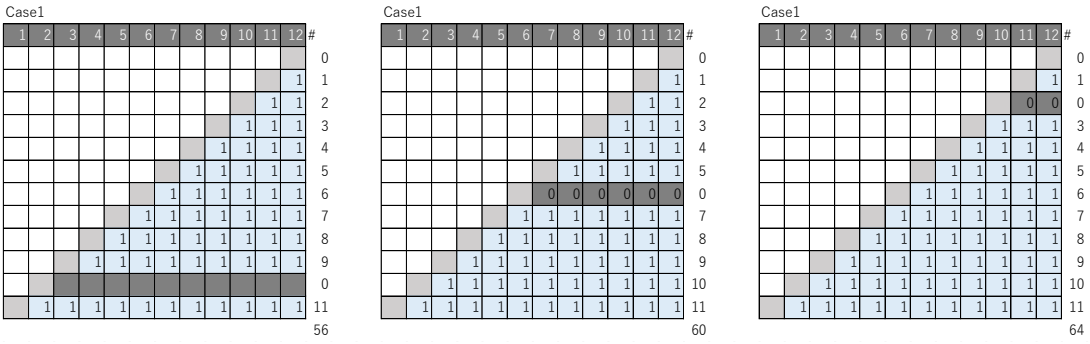


図 37 失敗の反映方法

6.2 シミュレーションモデルの設定

シミュレーションモデルの基本的な考え方は、第 5 章を踏襲している。右肩上がりの階段のような形態をとる。また、デリバリサイクルの設定についての表現も同様に踏襲している。そこに加えて、第 4 章で扱った成功率の概念を加える。

まず、改悪であることが、明らかになった時点で除却する。図 38 に例を示す。ここでは、デリバリサイクルが最も短いケースを表している、赤いセルのタイミングで改悪であることが判明し除却される。

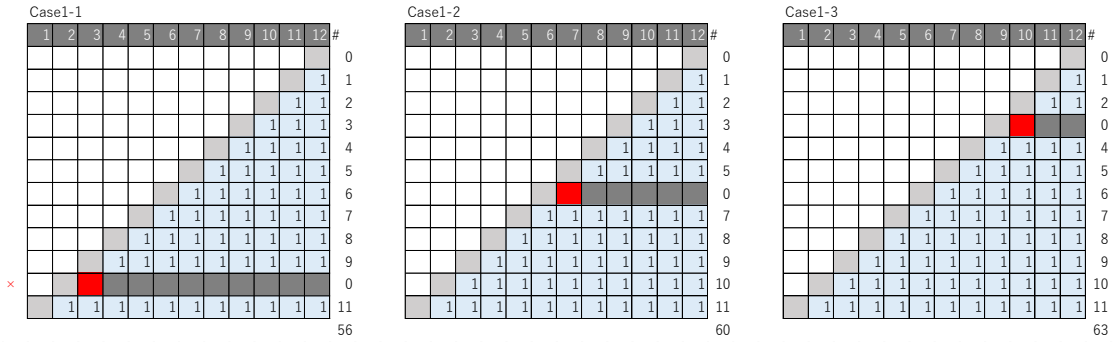


図 38 単純な改悪の例

次に、デリバリサイクルを長くした例を図 39 として示す。ここでは、デリバリサイクルを 3 週間とし、3 つの改善施策が同時リリースされる例となる。Case2-1 では、3 つの同時にリリースした改善施策のうち 1 つが改悪であり、2 つが改善となった場合を示す。本来的には、改悪が含まれているが、3 つ同時にリリースしているため、計測が 1 + 1 − 1 = 1 となっており改善に見えるため除却は行われない。次に、Case2-2 では、同じく 3 つの同時にリリースした改善施策のうち 2 つが改悪であり、1 つが改善となった場合を示す。本来的には、改善が含まれているが、3 つ同時にリリースしているため、計測が − 1 + 1 − 1 = − 1 となり改悪に見えるため全てが除却される。また、同時に測定される結果が 0 となる場合は、結果が 0 なので、除却することで結果は変わらないが、3.2.2 の事例に倣い除却としている。

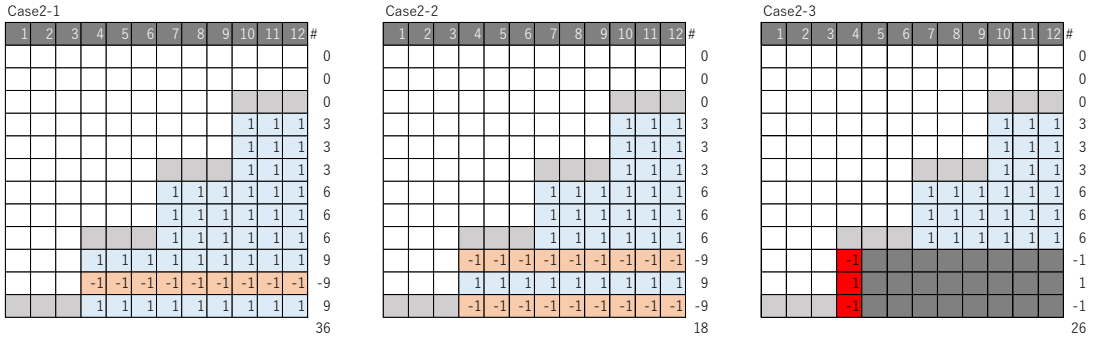


図 39 デリバリサイクルが長い時の改悪の例

ただし、6.2 でも触れたように、どの改善施策が改悪となるかによって大きく結果が変わってしまう。その為、成功率を設定するとランダムに改善と改悪が設定され、それを１０００回試行し平均値を採用する事とした。図 40 に例を示す。ランダムに３件の改悪が発生した例であるが、どこで発生したかによって全体の最終的な獲得利益が変わってしまう。

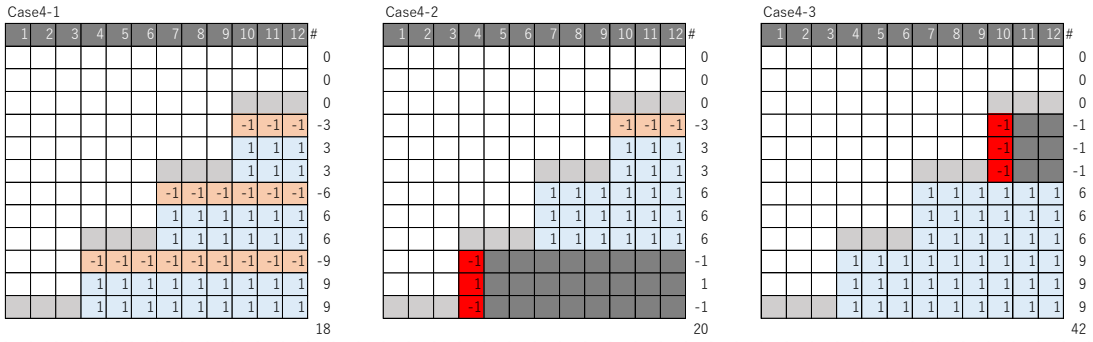


図 40 同成功率でランダムに改悪が発生する例

この様に、単純なロジックではあるが、どのリリースセットに発生するのか？それによって、生き残り続ける改悪の影響や、巻き込まれて除却される改善の影響、そもそも利益発生期間の長さによって、結果が変わる複雑なシミュレーションとなる。そのため、年間を想定した５０週間で、これまでの設定どおり１週間で１施策開発できるとして、５０施策を開発し、最大で５０施策をリリースする。そして、設定成功率に沿ってランダムに改悪が発生するといった計算を１０００回試行するシミュレーションプログラムを開発し検証を行った。

6.3 シミュレーション結果と評価

6.3.1 デリバリサイクル１～２５の結果

デリバリサイクルを１、５，１０，１５，２０，２５週に設定したシミュレーション結果を、横軸：勝率・縦軸：獲得利益というグラフで表現した（図 41)。シミュレーション自体は５０週で行っているため、デリバリサイクルとして設定できる最大値は５０週であるが、デリバリサイクルが２５週を超えるとリリース回数が１回となるため、ここでは２５週を最大とした。

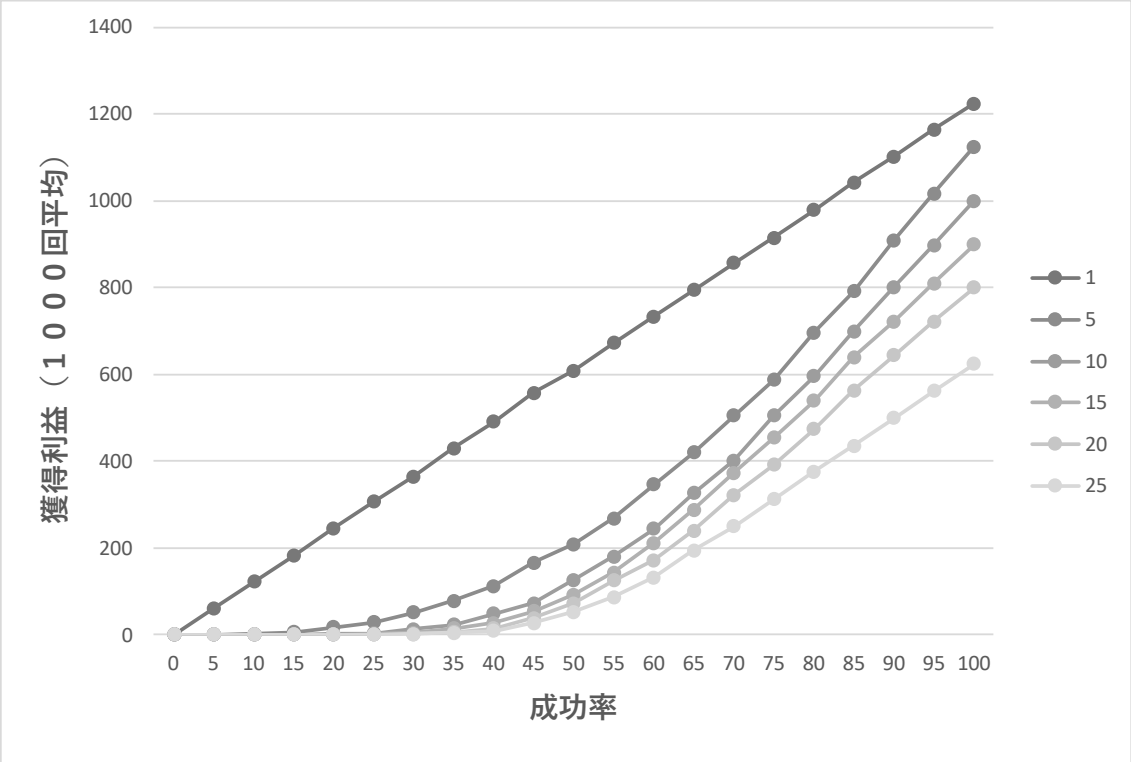


図 41 デリバリサイクルと勝率を変動させた結果

これ自体は、4.3.2 における図 22 と類似の形状をとっている。これらを図 42 にて比較した。

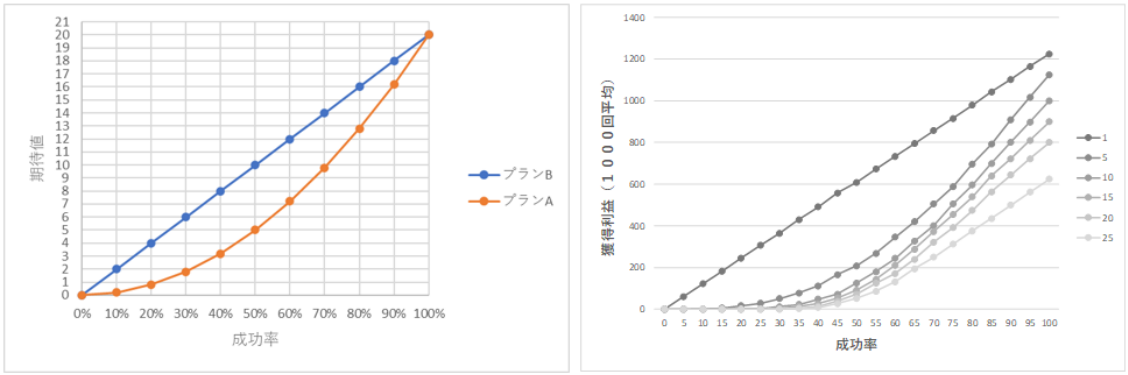


図 42 グラフ形状の比較

類似点は、1 次関数のような直線型の形状と、2 次関数のような曲線型の形状が現れることである。一方で、相違点は成功率１００％となったときに両者が一致するか？差分があるか？という点である。そもそも、本章におけるシミュレーションは第 4 章と第 5 章を組み合わせたものである。従って、類似点は第 4 章のメカニズムによって発生するものであり、相違点は第 5 章のメカニズムによって発生していると言える。実際に成功率１００％の状態で差が出るのは、第 5 章で論じている利益獲得の機会損失以外には発生しない。この 2 つのメカニズムが合成された結果となっている。

次に具体的な数値の違いについて表 6 にて確認する。

表 6 シミュレーション数値

		CDサイクル					
		1	5	10	15	20	25
成功率	0	0	0	0	0	0	0
	5	61.216	0.355	0	0	0	0
	10	123.117	2.405	0.04	0	0	0
	15	182.308	6.92	0.16	0	0	0
	20	245.559	16.94	1.48	0.52	0.06	0
	25	307.573	29.49	4.02	2.075	0.32	0.1
	30	364.828	51.125	12.64	5.45	1.8	1.525
	35	430.621	78.68	23.04	15.435	6.66	4.725
	40	491.299	112.725	49.26	27.025	14.98	9.65
	45	558.354	167.14	72.86	54.895	38.24	26.95
	50	609.727	209.615	127.06	92.845	72.34	52.825
	55	672.589	268.835	180.12	143.825	125	86.85
	60	734.496	346.04	245.1	211.585	171.76	133.075
	65	794.894	420.37	326.64	288.62	240.54	194.8
	70	857.823	505.265	402.54	373.645	322.14	250.025
	75	914.996	587.67	505.72	454.39	392.32	312.4
	80	979.062	696.66	597.06	539.385	474.04	377.075
	85	1043.148	792.795	699.7	639.965	564.34	436.1
	90	1102.991	909.735	801.8	722.39	644.08	499.95
	95	1164.744	1018.26	898.5	810.84	722.5	562.25
	100	1225	1125	1000	900	800	625

まず、１００％成功の場合は、デリバリサイクル１と２５で約２倍の差がある。これは、不確実性への対応力が無関係であるため、第５章の機会損失低下メカニズムによって発生した差である。次に、第３章の事例同等の成功率４０％でみると、デリバリサイクル１と２５で５０倍ほど獲得利益に差が発生する。この様に、成功率が下がるにつれて、この比率が拡大してくことが分かる。次に、少し視点を変えて、デリバリサイクル１で成功率５％の際に獲得した利益は、デリバリサイクル２５では、成功率５０－５５％付近と同等になることが分かる。つまり、デリバリサイクルを短期化させると、より低成功率な投資環境でも投資が継続できるようになることが分かる。この視点で、Min である０から MAX である１２２５までを、２５％区切りで色分けした（表 7 シミュレーション数値によって色分け）。同獲得利益帯が、デリバリサイクルを短期化させると低成功率にシフトしていくことが分かる。また、グラフ表現でも一次関数的な直線形状をとった様に、デリバリサイクル１で顕著に獲得利益額が大きくなることが分かる。

表 7 シミュレーション数値によって色分け

		CDサイクル					
		1	5	10	15	20	25
成功率	0	0	0	0	0	0	0
	5	61.216	0.355	0	0	0	0
	10	123.117	2.405	0.04	0	0	0
	15	182.308	6.92	0.16	0	0	0
	20	245.559	16.94	1.48	0.52	0.06	0
	25	307.573	29.49	4.02	2.075	0.32	0.1
	30	364.828	51.125	12.64	5.45	1.8	1.525
	35	430.621	78.68	23.04	15.435	6.66	4.725
	40	491.299	112.725	49.26	27.025	14.98	9.65
	45	558.354	167.14	72.86	54.895	38.24	26.95
	50	609.727	209.615	127.06	92.845	72.34	52.825
	55	672.589	268.835	180.12	143.825	125	86.85
	60	734.496	346.04	245.1	211.585	171.76	133.075
	65	794.894	420.37	326.64	288.62	240.54	194.8
	70	857.823	505.265	402.54	373.645	322.14	250.025
	75	914.996	587.67	505.72	454.39	392.32	312.4
	80	979.062	696.66	597.06	539.385	474.04	377.075
	85	1043.148	792.795	699.7	639.965	564.34	436.1
	90	1102.991	909.735	801.8	722.39	644.08	499.95
	95	1164.744	1018.26	898.5	810.84	722.5	562.25
	100	1225	1125	1000	900	800	625

色凡例	
～	306
～	613
～	919
～	1225

6.3.2 デリバリサイクル１～５０の結果

6.3.1 では５０週でリリースが１回になってしまうため、デリバリサイクルを１～２５とし、５刻みに設定したデータをもとに考察していた。ここでは、改めて外観をつかむためにデリバリサイクル１～５０を１刻みにした３Ｄのグラフ（図 43）で、更に特徴をつかんでいく。横軸にデリバリサイクル期間、縦軸に獲得利益、奥行き軸に成功率を設定している。

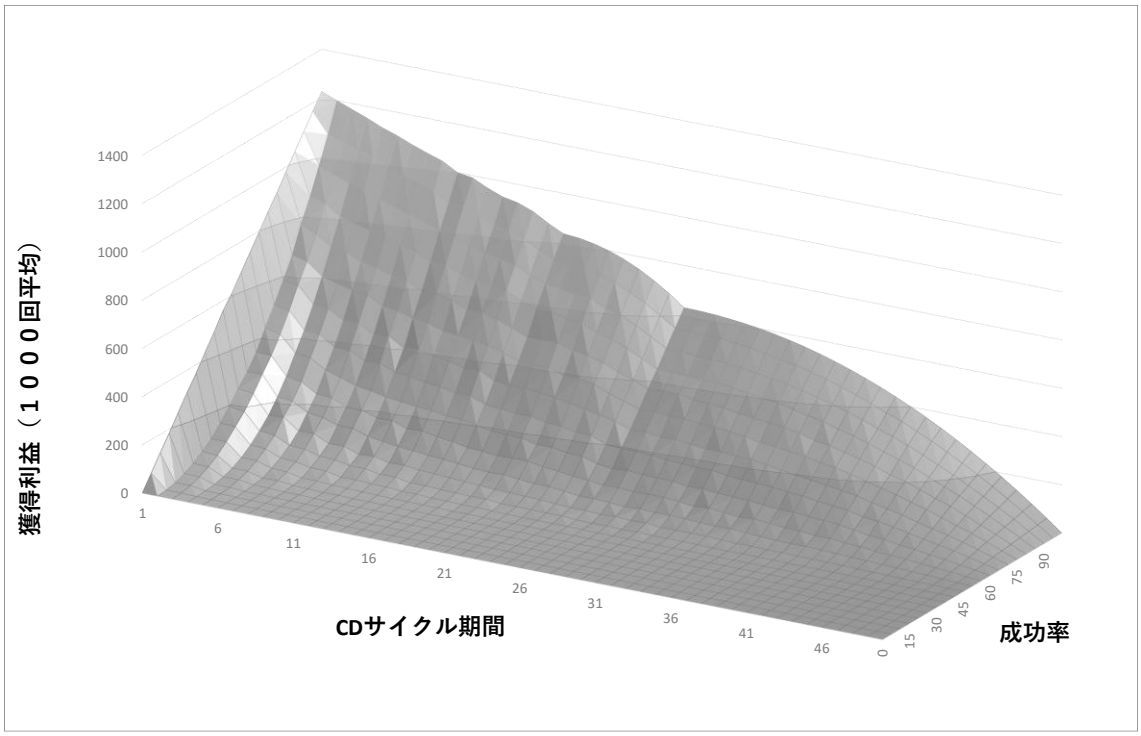


図 43 シミュレーション結果全体

まるで、山の様な形状をとるが、ポイントは下記の点である。

- デリバリサイクル１は線形を取り、左端で急激な崖のような存在となる。
- 成功率１００％（山の稜線部分）は、年間のリリース回数によって凹凸のある形状となる。図中央のデリバリサイクル２５では、谷のような形状をとっており、５０週（年間）で２回リリースか、１回リリースなのかの境界となる。その後も、デリバリサイクル１６付近に、２回リリースか３回リリース化の境界が、うっすらと見える。それ以降は、リリース回数が急激に増加していくため、細かい凸凹となって表れている。

6.3.3 シミュレーション結果のばらつきについて

ここまで、シミュレーション結果は、それぞれ１０００回の試行平均で扱ってきた。実際には、6.2 で言及しているとおり、成功と失敗がランダムで発生する為、それぞれの試行においては結果が異なっている。そこについて確認していく。

6.3.1 同様に、デリバリサイクルを１，５，１０，１５，２０，２５と設定した場合の状態について、１０００回の獲得利益の最小値と最大値をグラフの誤差として表現した（図 44)。

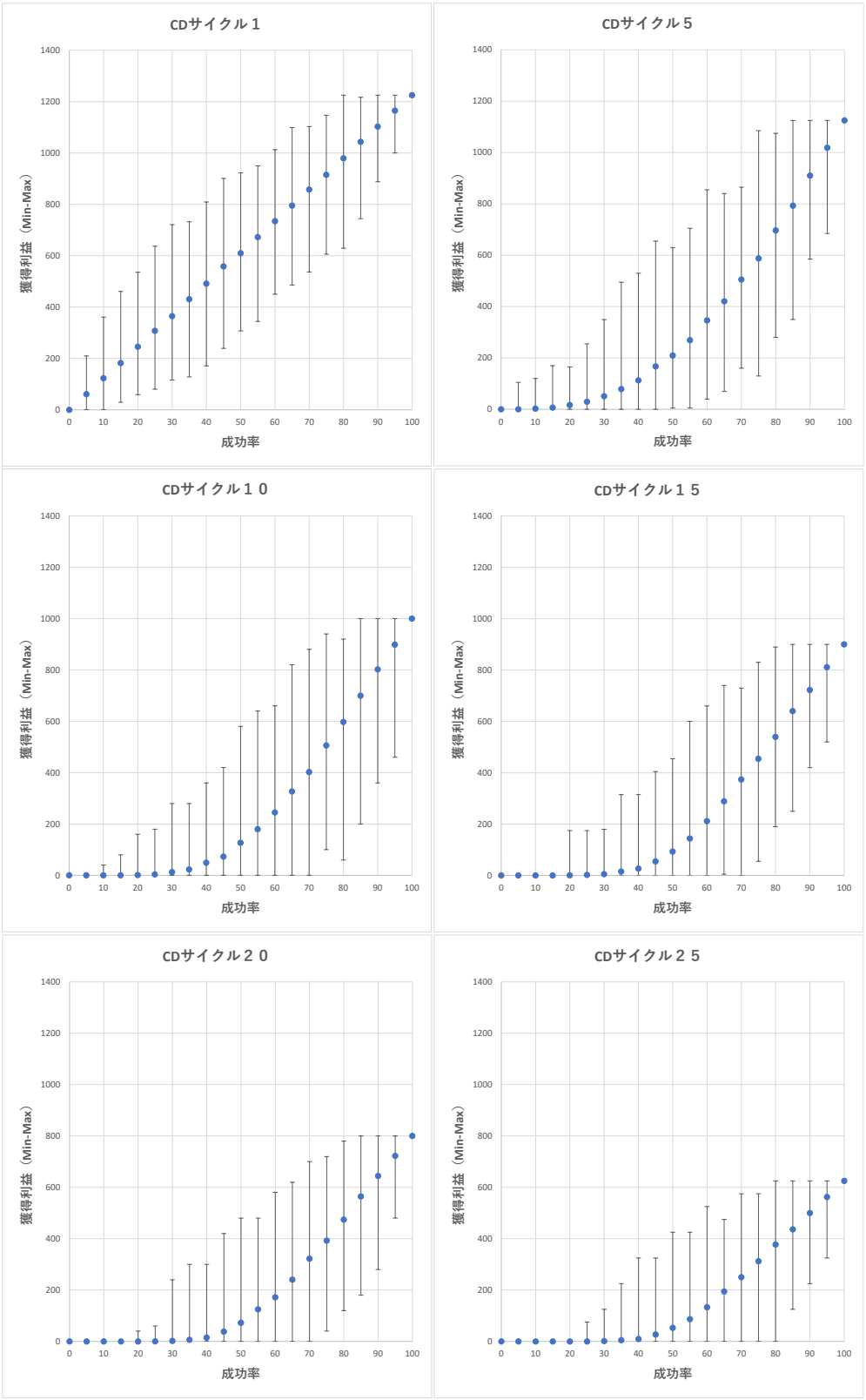


図 44 誤差範囲を反映した結果

下記のような特徴が読み取れる。

- 成功率０％と１００％は、当然であるが誤差が無い。
- 測定の結果、マイナスとなれば除却されるため、獲得利益がマイナスになることは無い。
- 逆に、成功率１００％の獲得利益を超過することもない。
- シミュレーションプログラムが正しくランダムで挙動していることを示しているだけだが、成功率７０－８０％付近の最小値がマイナスにならず、最大値が成功率１００％に到達しない付近では、平均値が誤差範囲の中央になる。

ここまでは、最小値と最大値が、どの様に発生したかを示したので、それぞれの標準偏差を計算したものをグラフに表す（図 45)。

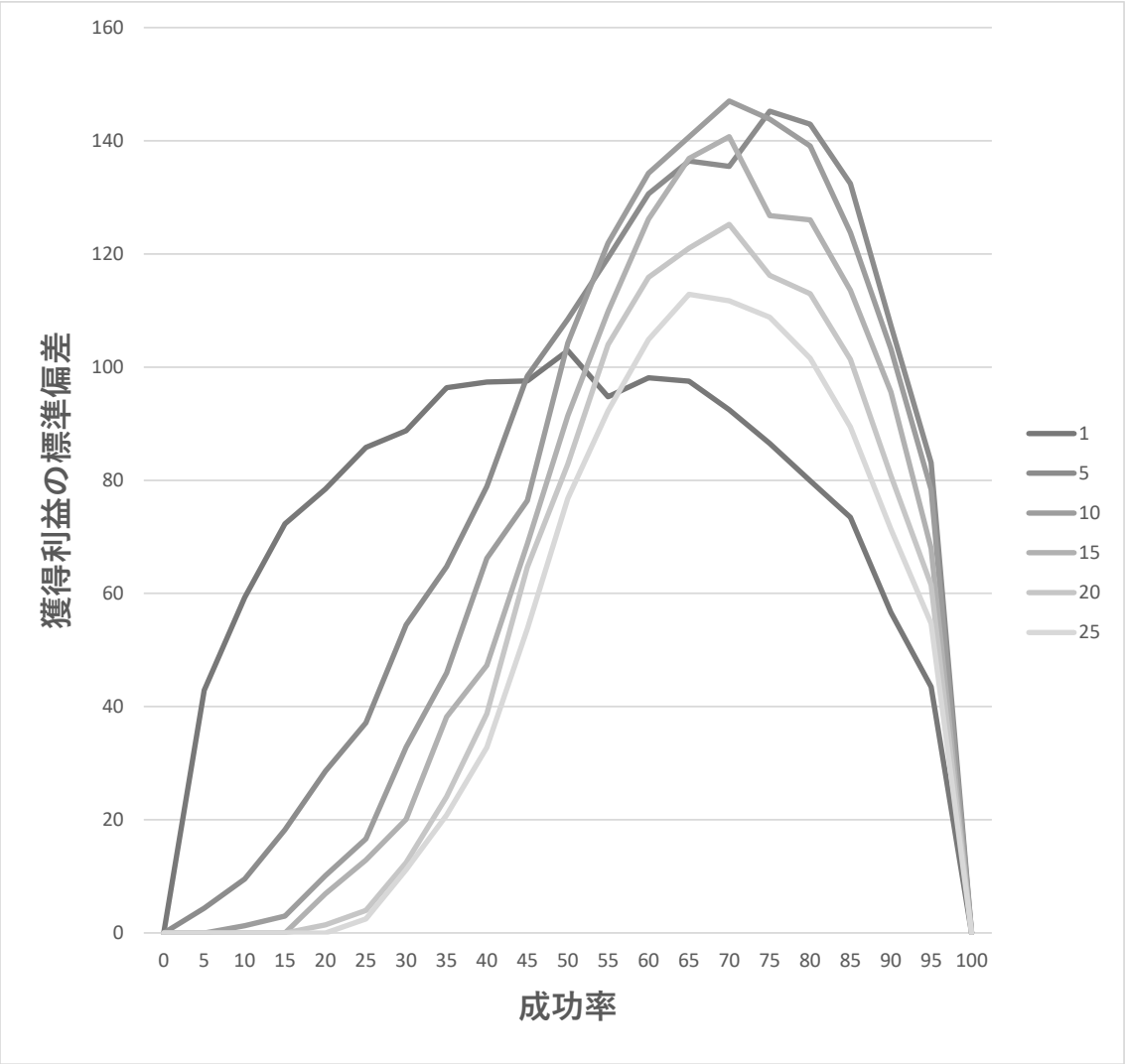


図 45 シミュレーション結果の標準偏差

ここでも、デリバリサイクル１は特殊な形状をとる。

- 偏差が一番大きいのは成功率５０％付近であり、左右対称な形状をとる。
- その一番大きい成功率５０％でも、他と比較すると偏差自体は小さい。
- 一方で、デリバリサイクル１以外は、同様の特徴を持っている。
- 偏差が一番大きいのが成功率７０％付近となり、左右非対称となる。
- 成功率７０％以下は最大に向かって、なだらかに上昇し、頂点を超えると急激に下落する。
- デリバリサイクルが小さくなると偏差が大きくなる。

この特徴は、図 44 と組み合わせることで、その構造が読み取れる。測定結果が 0 以下となり除却が発生する可能性が高い低成功率では除却によって偏差が小さくなる。また、デリバリサイクル 1 が巻き込まれて除却される改善も、改善に隠れてしまい残り続ける改悪マイナスも存在しないため偏差が小さくなる。逆に言えば、巻き込み除却や、隠れ改悪という何と同セットになったのかという事によって、運の要素が大きくなる、つまり不確実性が拡大させていると言える。

6.4 統合的な分割益についての定量的評価

本研究の結果として明らかになったことは、デリバリサイクルは、ビジネスアウトカムに対して大きな影響を及ぼすという事である。本研究では、2 つのメカニズムによって、その影響差が発生することを示した。1 つは、第 4 章で明らかにした不確実性の高い環境下において改悪の除却漏れによって暗黙的な損失が発生する影響、もう 1 つは第 5 章で明らかにした本来であれば獲得できた改善効果を逸失したことによって機会損失が発生する影響である。どちらも興味深いのは、デリバリサイクルを短期化させることで獲得利益が拡大するのではなく、デリバリサイクルを長期化させると本来獲得できた利益を逸失することになることで、ビジネスアウトカムに対して影響が発生するという点である。このことを、図 46 に図示する。6.2 のシミュレーション設定において図 39 として例示したデリバリサイクル 3 とデリバリサイクル 1 で、利益差分が発生した箇所を図示したものである。ここでは、3 つのパターンで利益差分が発生していることが見て取れる。1 つ目は、第 5 章の内容と合致する内容であるが、本来利益を獲得できていた期間を逸失している（図 46 の赤枠に該当）。2 つ目は、第 4 章の内容と合致する内容であるが、本来改悪となってしまっている施策が、他の改善施策と混ざってしまい除却する機会を逸失している（図 46 の黄枠に該当）。3 つ目は、2 つ目の逆で本来改善となる施策が、他の改悪に巻き込まれて除却されてしまい改善施策自体が逸失している。第 6 章のシミュレーションでは、これらのパターンがランダムに発生していた結果として、最終的な獲得利益に差が発生していた。

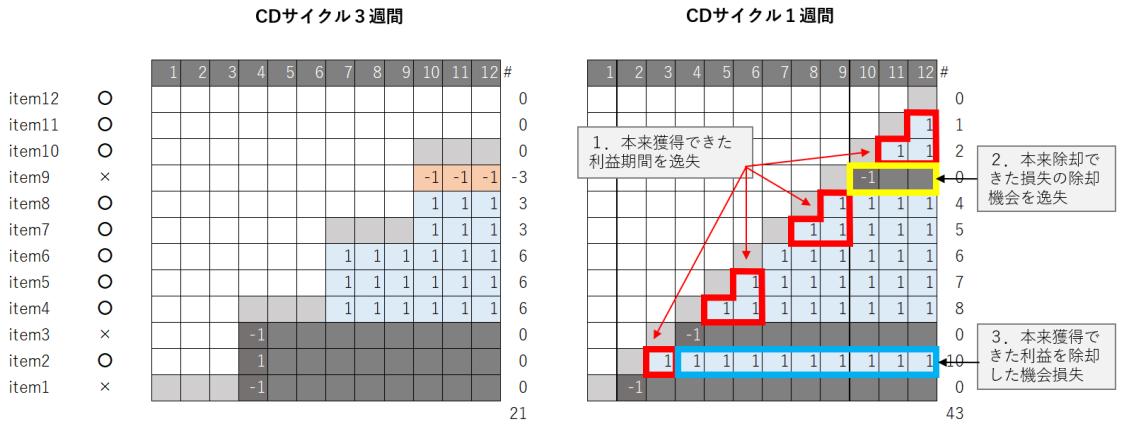


図 46　逸失した利益の発生個所

繰り返しになるが、デリバリサイクルを短くすることで獲得利益が増大するのではなく、デリバリサイクルが長くなると３パターンの機会損失によって獲得利益が減少するというのが正しい。一方で、デリバリサイクル２５（現実には半年毎にバージョンアップする運用を意味する）や、デリバリサイクル８（隔月でバージョンアップする運用）からすれば、デリバリサイクルを短期化させると獲得利益が増加していく様に視点が逆転してしまう。

ここまで、利益差分が発生したメカニズムについて検討を行ったが、ここからは、その差分の定量的な規模について検討を進める。第４章では、ビジネスアウトカムに対して１０倍程度の影響があることが分かった、しかし時間軸という概念がない成功率だけのシミュレーション結果であった。第５章では、２倍程度の影響があることが分かった、しかし成功率という概念がない時間軸だけのシミュレーション結果であった。第６章では、それらを統合したシミュレーションを行った。第３章の事例調査で明らかにした成功率４０％での設定では、デリバリサイクル１と２５で５０倍ほど獲得利益に差が発生する。これは、現実世界に置きなおすと。デリバリサイクル１とは、年間５０週間に毎週改善施策をリリースすることを意味しており、デリバリサイクル２５とは半年に１度まとめてリリースすることを意味している。この５０週で開発した施策は、どちらも同じであり週に１つトータル５０の施策を開発している。そして、投下した開発者の人件費も同じである。つまり、全く同じものを、全く同じ人が作り、全く同じ改善・改悪の成功率だったとしても、そのリリースを小分けにしたか、まとめたかというだけで、投資効率が５０倍変わるという事である。もしも、仮に同レベルの投資効率まで成功率

を上げようとすると図 47 の様に４０％の成功率を９０％まで向上させる必要がある。そもそも、失敗しようと思って改善を行っているわけではない前提において、成功率を２倍以上に引き上げるというのは非常に困難である。このことは、収益性を高める影響が大きいことだけでなく、代替も難しいということを示している。

		CDサイクル					
		1	5	10	15	20	25
成功率	0	0	0	0	0	0	0
	5	61.216	0.355	0	0	0	0
	10	123.117	2.405	0.04	0	0	0
	15	182.308	6.92	0.16	0	0	0
	20	245.559	16.94	1.48	0.52	0.06	0
	25	307.573	29.49	4.02	2.075	0.32	0.1
	30	364.828	51.125	12.64	5.45	1.8	1.525
	35	430.621	78.68	23.04	15.435	6.66	4.725
	40	491.299	112.725	49.26	27.025	14.98	9.65
	45	558.354	167.14	72.86	54.895	38.24	26.95
	50	609.727	209.615	127.06	92.845	72.34	52.825
	55	672.589	268.835	180.12	143.825	125	86.85
	60	734.496	346.04	245.1	211.585	171.76	133.075
	65	794.894	420.37	326.64	288.62	240.54	194.8
	70	857.823	505.265	402.54	373.645	322.14	250.025
	75	914.996	587.67	505.72	454.39	392.32	312.4
	80	979.062	696.66	597.06	539.385	474.04	377.075
	85	1043.148	792.795	699.7	639.965	564.34	436.1
	90	1102.991	909.735	801.8	722.39	644.08	499.95
	95	1164.744	1018.26	898.5	810.84	722.5	562.25
	100	1225	1125	1000	900	800	625

色凡例

～	306
～	613
～	919
～	1225

CDサイクルを25から1にすることは、成功率を40%から90%へ向上させるのと、同等の効果がある。

図 47 50 倍の投資効率を実現する場合

6.5 分割損を加味した定量的評価

ここまでは、デリバリサイクルを短期化することによる利点サイドについて扱ってきたが、ここでは、分割損といった不利益サイドについて扱う。実際問題として、デリバリサイクルを２から１にすると、単純にリリース作業・リグレッションテスト[6]などのリリースに伴う作業は２倍になる。図 47 で行っているのは、収益サイドで５０倍の増加が見込むには、デリバリサイクルを２５から１にする必要があるという比較である。

6 新しく追加した機能自体の動作をテストするのではなく、追加によって既存機能に悪影響がないことを確認するテストのこと。

この場合は、このリリースに伴う作業についてコストサイドで２５倍増加するという構造である。仮に、デリバリサイクル２５とは、開発作業の全量が２５×５＝１２５人日となるので、その３%がリリースに伴う作業（３－４人日程度）であるとすると、２５倍で７５－１００人日となるというのが単純計算であり、開発作業自体は２５人日しか残らないので、作るモノ自体が減ってしまうので、そもそもリリースするものが無くなる。このことが意味することは、そもそも計算が合わないという事であり、デリバリサイクル１まで短期化するのは現実的ではないということである。そのため、分割損というよりは、不可能要因という方が正しい。

このことは、先行研究においてCD導入によって品質がトレードオフとなり、それが導入障壁であるという報告(Schermann, Cito, Leitner, and Gall 2016)とも同義である。つまり、この報告の前提は、リリース付帯作業として挙げたリグレッションテストを減ってしまうことを意味している。他にも増加する作業工数のために開発者を増員するという方法も考えられる。その増員の仕方も、DevOpsといったテーマとも関係性がある。その場合は、人件費が増加するが、収益性が５０倍になるのであれば、選択可能であろう。一方、CSEの枠組みでは、Continuous Integrationというテーマで、リグレッションテストの自動化が扱われており、Continuous Deployというテーマで、リリース作業の自動化が扱われている。どちらも、自動化がテーマとなっている。まさに、この分割損を自動化によって最小化させるというアプローチである。

この様に、そもそもソフトウェア開発とは、あらゆる企業における投資である以上は、デリバリサイクルを短期化させてソフトウェア開発の収益性を高めるというのはCSE導入の根幹となる目的といえる。ただし単純に短期化させることは難しい。そのためには、DevOpsやCI/CDといった様々なCSEテーマが必要になる。そのような関係であることを示唆しているとも言える。

6.6 本章のまとめ

第３章では、インターネットサービスの改善というのは継続的に行われ続けており、そのサイクルを短くする努力が行われていること、一方で改善の勝率というのは高くなく成功率５０%を切るような不確実性の高い環境であることを明らかにした。第４章では、改善施策を小分けにしてデリバリサイクルを短くすることで、低い成功率であって

も利益期待値を向上させられることを明らかにした。第 5 章では、同じくデリバリサイクルを短くすることで獲得利益の機会損失を低下させられることを明らかにした。

第 4 章・第 5 章ではメカニズムを明らかにするために、敢えてシンプルなシミュレーション設定としたが、現実には同時に発生しており、デリバリサイクルを短くする効能というのは、両者の合算によって表現されるべきである。そのため、本章では 6.2 にて両者を組み合わせるシミュレーションモデルを設定し検討を行った。

なお、第 5 章の階段型の様なシミュレーションモデルに、第 4 章のような成功失敗を設定すると、どれが成功もしくは失敗としたかによって恣意的なシミュレーションとなってしまう。そこで、本章では改善アイテムの成功・失敗がランダムに決定されるシミュレーションプログラムを作成し、その試行回数を増やすことで、両者のメカニズムが働いた際のビジネスアウトカムへの影響値を定量的に明らかにするアプローチをとった。

結果として、成功率 40%という環境下では、毎週サイクルでリリースしていく場合と、半年サイクルでリリースする場合、50 週間（1 年を想定）の獲得利益で 50 倍の差がつくことが明らかになった。これは、成功率を 40%から 90%へ上昇させるのと同等の効能であることが明らかになった。これは、マーケティングコストなどを投下して成功率を向上させるよりも、安価にデリバリサイクルを短期化させることが出来れば、その方が合理的であるということを示している。

第7章 考察

7.1 学術的な背景

DX への取り組みのため、高まる不確実性への対応のために、アジャイル開発や DevOps の活用が重要である(経済産業省デジタル産業の創出に向けた研究会 2021）とされ、ソフトウェア開発の変革は多大な期待をされている。しかし、このアジャイル開発自体は多様な方法論があるが、明確な定義が存在しない。唯一存在する共通的な定義は、「アジャイル開発宣言(agilemanifesto.org 2001a)」であり、極めて抽象度の高いマインドセットと１２の原則が存在するのみである。一般的なワードとしては広く普及しているのだが、そこに共通の定義はなく、学術的にも取り扱っている例は少ない。

インターネットサービスのソフトウェア開発は何が正解か分からない中でも、改善と失敗を小さく高速に積み重ねながら、ソフトウェアの価値を向上させ続けている。これは、一般的にはアジャイル開発と呼ばれるものである。この永遠に続く価値向上活動のことを「完成することのないソフトウェア」として、それが完成させるための知識体系である従来のソフトウェア工学ではカバーできていないという学術的な課題認識から構築されている知識が CSE である。CSE では、アジャイル開発のような方法論だけでなく、経営学や情報システム学といった学際的な知識の統合として扱うべき(Fitzgerald and Stol 2017）とされている。実際に、多角的な学術的研究が展開され、その知識の体系化や、研究するべきテーマの提言が活発に行われている(Johanssen et al. 2018; Klotins and Gorschek 2022)。

本論文は、この様な学術的な背景の中で、まさに多角的な視点かつ学際的な研究に取り組んだ。

7.2 デリバリサイクル短期化による不確実性への対応力向上メカニズム

CD と隣接する Continuous Experimentation(CE)というテーマが存在する。これは、ソフトウェア開発と AB テストといった手法を組み合わせて探索的かつ継続的にソフトウェアを改善し続ける分野を扱ったテーマである。しかし、この CE においても、そ

のスケジューリング方法に遺伝的アルゴリズムを用いる手法提案(Schermann and Leitner 2018）や、CE 導入における困難についての研究(Auer et al. 2021）が行われているにとどまっている。そもそも、CE で扱っているソフトウェア開発と AB テストが一体化したプロセスが、何に有効であり、それが何故であるかについては研究が進んでいない。本論文の第 4 章では、このソフトウェア開発と AB テストが一体化したプロセスにおいて、リリースサイクルの設定が不確実性への対応力を向上させることをにて示したことは、本研究の理論的意義を持つと言える。

7.3 デリバリサイクル短期化による投資効率の改善メカニズム

Facebook（現 Meta）社のような先進的で大きく成長した企業が実践している(Feitelson, Frachtenberg, and Beck 2013）こともあり、社会や企業は CD に注目している。そのような背景もあり、比較的新しい研究分野である SCE の中でも、CD についての研究は古く、そして多く展開されている。CD に注目している企業が多いが、一方で導入に困難があることも多く。組織構造の問題(Leite et al. 2021）や、アーキテクチャの問題(Schermann, Cito, Leitner, and Gall 2016）や、トレードオフとなる品質の問題(Schermann, Cito, Leitner, and Gall 2016）についての研究が進んでいる。

一方で前述のとおり、既に実践している企業が多いことも事実である。しかも、その企業群は、デリバリサイクルを月次、週次、日次‥と短期化を行い続けている(Feitelson, Frachtenberg, and Beck 2013）。それは CD の導入・デリバリサイクルの短期化にメリットがあることは言うまでもない。

しかし、そのメリットについての研究は進んでいない。研究論文においても、ＣＤや、サイクル短期化の利点について、ジャストインタイムの方が良いという説明(Rico, Sayani, and Sone 2009）や、競争環境において競争相手よりも早い方が良いという説明(Schermann, Cito, Leitner, and Gall 2016）にとどまっており、漠然とした利点への言及に留まっている。本論文において、デリバリサイクルの短期化によって、利益獲得の機会損失を低下させるメカニズムを示したことは、本研究の理論的意義を持つと言える。

7.4 プログラムマネジメントの視点

第 2 章の先行研究レビューにも記載の通り、本研究のテーマは持続的に価値を高めていく点において、プログラムマネジメントと類似である。一方で、プログラムマネジメントには、全体使命と呼ばれる、大きな目的・ゴール設定が重要とされている。これは、インターネットサービスにおけるソフトウェアは完成せず、永遠に続くという文脈とは必ずしも一致しない。また、プログラムは複数のプロジェクトから構成されるが、本研究対象で扱っている 1 つの改善を 1 つのプロジェクトと呼ぶには、想定サイズが違う。

一方で、複数のプロジェクトから構成されるプログラムにおいて、そのプロジェクトの粒度を設計し実行していくことは、まさにプログラムマネジャーの中心的仕事であり、その目的はアウトカムの拡大のためである。これは、サイズこそ違うが、本研究と同じ問題と向き合っていると言える。どの様に、プロジェクトを分割すれば受益は拡大するのか？という問題である。そのため、本研究をプログラムマネジメントの視点から考察することには価値がある。

プログラムマネジメントの視点でとらえるために第 3 章の図 17 改善プロセスを 3S モデルにマッピングした。

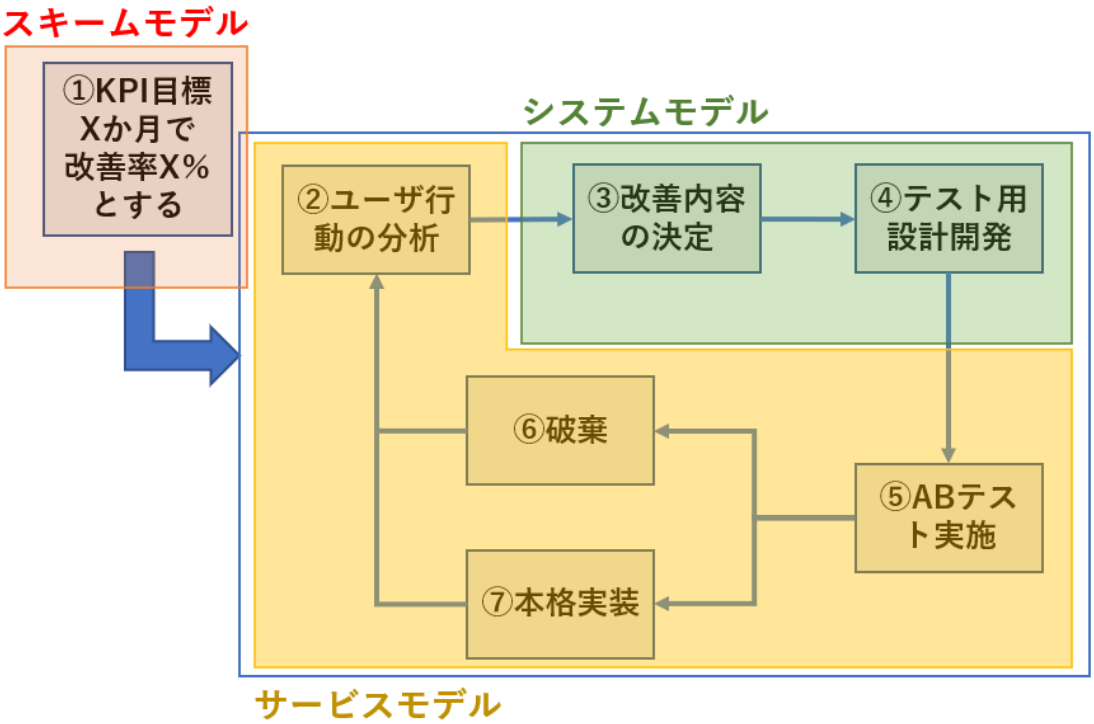


図 48　改善プロセスを３S モデルで表現

で調査した事例でいえば、①がスキームモデルに該当し、③④がシステムモデルに該当し、②⑤⑥⑦がサービスモデルに該当すると言える。ただし、②～⑦がサイクリックに回り続けている。これを、一般的な３S モデルと比較する様に図 49 にて図示する。

■一般的な３Sモデル

■本研究の３Sモデル

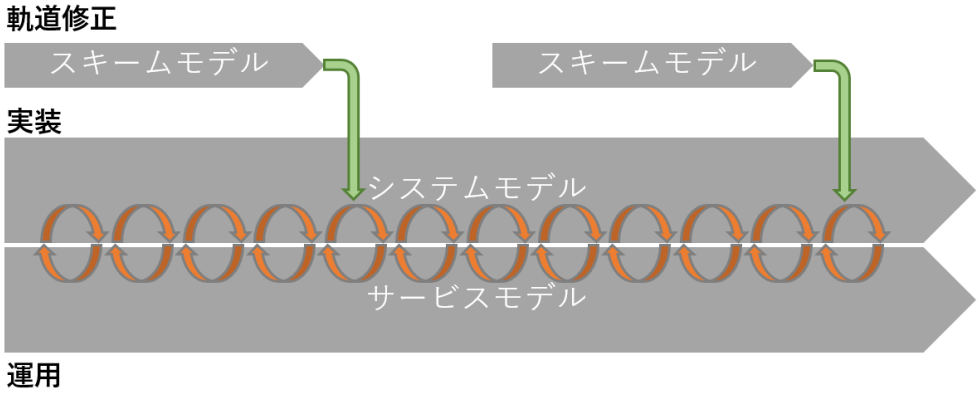


図 49　一般的な３S モデルとの比較

この様に、スキームモデル・システムモデル・サービスモデルがシーケンシャルに並ぶのではなく、システムモデルとサービスモデルが継続的に動き続けていて、スキームモデルが、それを KPI 目標の設定という形で軌道修正していると捉えることが出来る。

もともと３S モデルは、スキームモデルでプログラムミッションを定義し、そのミッションを達成するためのシステムをシステムモデルで実装し、そのシステムがサービスモデルで稼働させることで、スキームモデルで定義したミッションが達成される。この様に、前後関係のあるシーケンシャルな関係性である。そのため、プログラムには開始と終了が存在する。これは、前述の通りインターネットサービスの改善に終了がないということを表すには不向きで合った。本考察で提案する３S モデルでは、システムモデルとサービスモデルが常時稼働しており、スキームモデルが軌道修正するという関係性となることで、永遠に終了しないモデルとして表現できたと考える。これは、終了しないプログラムマネジメントの３S モデルとしてプログラムマネジメントの知識体系に

新しい概念を提案するものであると考える。

7.5 本研究の他分野への応用

ここでは、本研究の知識をソフトウェア開発以外の分野へ応用する視点についての考察を行う。本研究の中心的メカニズムは、ソフトウェア改善を可能な限り小刻みに行うことは、2つの機会損失を低減させる。それは、意図しない改悪を認識し除却する機会と、統合によって本来であれば利益を創出できた期間を獲得できていた機会の2つである。また、本研究では、これらの機会損失によるインパクトは大きいということが判明した。

機会損失の低減は、小売りにおいて在庫管理を適切に行い販売機会の逸失を回避するといったように、様々なビジネスにおける課題であった。むしろ、機会損失という言葉を聞いて想起されるのは、そのような在庫・出店・販売・・といった一般的なビジネス課題である。一方で、本研究における機会損失というのは、除却する機会・利益獲得期間という機会の2つである。これはこれで、独特な機会の類型を扱っている可能性がある。例えば、製造業においても、とある商品Aに対して改善しようと思った変更が、実は改悪であったということは起こりうる。ただ、「改悪だったので変更を捨てる」という発想にはならない。そもそも改悪であるかを正確に判別できない可能性がある。そして、仮に判別できたとしても、変更を捨てる・元に戻すということが出来ない。また、利益獲得期間についても、すぐに変更を反映させられるから機会が失われていると認識できるのだが、そもそも「すぐに反映」ということ自体が特殊である。つまり、本研究で取り上げた機会損失とは、すぐに変更を反映可能であり、改善や改悪が定量的に把握可能であるという、デジタルサービスの特徴が前提となっている。つまり、言い換えれば、本研究ではデジタル化によって発生した新たな機会損失の類型を提示しているとも言える。昨今の電子機器ではネットワーク経由でバージョンアップが可能になっているし、ゲームなども同様にネットワーク経由でコンテンツの追加などを行える。ある意味でインターネットサービスの特徴に近づいているともいえる。本研究の成果は、このようなシーンに活用可能となる可能性がある。

次に、成功と失敗を繰り返しながら価値を拡大させていくことの意味について考察を深めたい。類似なことに、リーンスタートアップにおけるピポットなどがある。結局、

何が正解か分からないから、ピポットしながら正解を探すといった文脈で用いられることが多いが。裏返せば、ピポットするたびに、それまでのことを除却していると捉えることも可能であろう。いうまでもなく、スターアップのピポットには根本的なモノからマイナーチェンジに近いものまである。ビジネスモデルが確定した後のスタートアップビジネスのグロースにおいても、本研究の知識を活用できる可能性がある。

7.6 知識科学への貢献

最後に知識科学への貢献について考察する。本研究は、インターネットサービスの改善を効率よく行うことがテーマである。ここでいう改善とは、不特定多数である利用者の過半数が好むデザインや機能を探し当てることを意味する。そして、その改善がサービスに組み込まれ継続的に利益を生み続ける。このことは、知識科学の視点でいえば、利用者の好みという知識の表出化させ、サービスに対して蓄積し続けていると言える。本研究では、この知識の蓄積を効率よく行う方法と、その効率を定量的に示し評価することを試みていると解釈できる。

これは、良い知識（改善）を上手に想像させる発想ではなく、悪い知識（改悪）が混入することを前提としており、それを除却する方法を洗練させることで、良い知識だけを残し資産化させていくという方法論を示唆している。また、失敗可能性の高い知識創造において、試行と失敗の除却を可能な限り小刻みに行うことで、知識創造の生産性を大きく向上させることも示唆している。これは、「有用な失敗の仕方」と言い換えることが可能であり、この逆説的ともいえる失敗への着目が、知識科学における新しい研究テーマになりうることを示唆している。

また、これらを期待値計算やシミュレーションによって定量的に評価している点も特徴である。本研究の様に、知識の良し悪しを定量的に規定し、数理モデル化することができるということも、知識科学における新しい研究アプローチとなりうることを示唆している。

第8章 結論

8.1 本研究のまとめ

全ての企業にとって、ソフトウェア開発とは投資である。いうまでもなく、ソフトウェアを完成させることが目的ではなく、完成したソフトウェアから利益を生み出すことが目的である。ソフトウェア開発の生産性を上げることは、ソフトウェアが重要なビジネスモデルにおいては、重要なテーマであり永遠の課題でもある。一方で、ソフトウェア開発の生産性とは、主にコストサイドの問題と認識されることが多く、利益サイドは、ソフトウェア開発の方法というよりは、何を開発するのかという内容で決まると考えられていることが多い。本研究では、仮に作るモノが同じだったとしても、ソフトウェア開発の方法が利益サイドに影響があることを明らかにした。

第 2 章では、先行研究レビューでは、従来の SE がソフトウェアを完成させることを目的とした知識体系であったこと。一方で、インターネットサービスには、完成しないソフトウェア開発があるということ。そのためにも知識体系の拡張が必要であり、CSE という概念が生まれたこと。その CSE には経営学や情報システム学や組織学といった学際的な知識統合が必要であるということを述べた。

第 3 章では、実際にインターネットサービスのソフトウェア開発の事例調査を通じて、そもそも使いやすさという曖昧な要求仕様が改善テーマであり、ユーザという不特定多数ステークホルダーの投票で決まる特定があること。その曖昧な要求仕様の実現がビジネス的に大きな影響を持っているという事、一方で、その改善の成功率が３０－４０％と低く、失敗した場合はゼロではなくマイナスになるということを明らかにした。

第 4 章では、成功率が低いという特徴に着目してシミュレーション設定を行い、開発施策を分離することで、成功率５０％という環境であれば１０倍程度の収益性差分が発生するということを明らかにした。これは、金融工学や経営学で利用される、不確実性下における投資理論であるリアル・オプション理論にて説明可能なメカニズムが働いていることを明らかにした。

第 5 章では、デリバリサイクルを短期化させる影響を長い時間軸で評価するために、階段型のような形状をとるシミュレーションモデルを考案しており、その結果として成功率１００％の同期間でも、短期化させるだけで 2 倍程度の収益性に差分が発生することを明らかにした。その差分は、利益を獲得する機会を逸失してしまうメカニズムから発生することを明らかにした。

第 6 章では、第 4 章と第 5 章を統合するためのシミュレーションプログラムを作成し、２つのメカニズムが同時に働いた場合のビジネス的な影響値を明らかにした。第 3 章の事例にあった成功率４０％では、デリバリサイクルを２５とするか、1 とするかで、年間５０倍程度の収益性影響があることが確認された。作るモノが同じで、作る人も同じだが、収益性が５０倍も変わるというのは、大きな影響があることを意味している。また、５０倍の収益性向上を行う場合、成功率を４０％から９０％へ引き上げる必要があることを示し、その影響力の大きさだけでなく、同効能を得るための代替方法の困難さにも言及した。一方で、デリバリサイクルを短期化させることによって発生する分割損についても検討を行い、短期化によって増加するリリース付帯作業というのが、分割損というよりも短期化不可能要因となることを示した。また、そのリリース付帯作業を低減化させることが、CSE の他テーマとなっていることを示し、CSE における複数テーマの依存関係を示唆した。

8.2 各研究結果とリサーチ・クエスチョンに対する回答

ここでは、本論文で扱っている 3 つの研究結果と、当初設定したリサーチ・クエスチョンへの回答としてまとめ直す。

研究 1: デリバリサイクル短期化は、どの様なメカニズムで投資不確実性に有効性があり、その有効性はどの程度か？

第 4 章にて対応。不確実性の高い状況下でリアル・オプション理論の撤退オプションが働くことで、投資効率が向上するというメカニズムが働いている。有用性は最も大きな差が付くのは成功率５０％の環境であり、このメカニズムだけで１０倍程度の差が生

まれる。

研究 2: デリバリサイクル短期化は、どの様なメカニズムで獲得利益を増大化させ、その増大量はどの程度か？

第 5 章にて対応。デリバリサイクルを長期に設定すると利益獲得の機会損失が発生するメカニズムによって、逆説的に短期化は利益が増大する。単純な短期化を行えば 2 倍程度の差が発生する。

研究 3: 研究 1,2 を統合した結果として、その有効性はどの程度か？

第 6 章にて対応。研究 1,2 のメカニズムが同時に働くことで、成功率４０％の環境で、デリバリサイクルを２５（半年に１度リリース）から１（毎週リリース）へ変更すると、年間５０倍程度の収益性影響が発生する。また、同等の収益性影響を成功率で実現するには、成功率を 40%から 90%に引き上げる必要があるため、容易に代替できることではない。

RQ: デリバリサイクル短期化がビジネス上有効性（継続的な利益）を発揮する条件と、その有効性はどの程度か？

デリバリサイクルの短期化がビジネス上有効性を発揮するのは、下記の点である。

前提となる条件：この条件を満たさなければ、成立しない。

- 継続的な改善が可能[7]なソフトウェアである。
- ソフトウェアの改善がビジネス利益を生み出す。
- 改善においてビジネス利益が計測可能である。

[7] CD/DVD 等で販売するソフトウェアの場合は、そもそも不可能である。

有効性を高める条件：この条件を満たせば、より有効性が高まる。

- 改善によって生み出される利益が大きい。
- 不確実性が高く、改悪となる失敗が発生しうる。

制約を生み出す条件：この条件はデリバリサイクルの短期化の障害となる。

- リリース付帯作業に多くの工数がかかっている。

これらを満たすのは、インターネットサービスの特徴である。その有効性の程度は、研究 3 の結果にある通り、年間 50 倍程度の収益性を改善することができる影響度である。この様に強力な有効性であるから、インターネットサービス企業の一部では理論的背景無しにデリバリサイクルの短期化が受け入れられ、そのための障壁を取り除く努力に企業が取り組んできたと考える。

8.3 理論的含意

7.1、7.2、7.3 で説明したように、そもそも CD・CE の導入や、デリバリサイクルを短期化させる必要性、そのメカニズムについての研究事例は存在しない。当然ながら、それらがビジネスアウトカムへ定量的に、どの程度の影響を与えるのかといった研究事例も存在しない。様々な CSE、CD の導入障壁は報告されているが、そもそもメリットという導入モチベーションについて寄与する研究が存在していないのが現状である。本論文では、メカニズム解明を通じてシミュレーションモデルを作成し、これらのメリットを定量的に表現している。この定量的な表現及び、定量的表現を可能としたシミュレーション手法は、本研究の理論的意義を持つと言える。

8.4 実務的含意

本研究では、CSE、CD 導入の意義を定量的に表現し、その表現手法も提供している。先行研究にもある通り、これらの導入は様々な困難を伴う。程度の差こそあれ、導入を進めるにあたり資本的・人的コストを投下する必要があるということである。場合によっては、システムアーキテクチャを大幅に変更する必要があるかもしれない、他にも自

動テストの基盤を用意するなり、自動デプロイの環境を新たに用意する必要があるかもしれない。そのために、新たに雇用しなければならない可能性すらある。これらのコスト投下は、エンジニアリングコストを低下するために行うと誤解されているケースが一般的ではないだろうか。例えば、テスト自動化基盤を導入することで、年間数百万円のエンジニア人件費が低下する（しかし、レイオフでもしない限り、実際には人件費は低下しない）といった具合である。本研究の結果をみれば、それは誤解であることが明らかである。目的はデリバリサイクルを短期化することであり、更なる上位目的はソフトウェア開発の収益性を５０倍に高めることである。その障壁となるテスト工程の期間を短縮するために導入するという意思決定に代わる。1.1 の背景でも述べた通り、ソフトウェア開発の変革が求められている。この変革の目的・メカニズム・効能を経営が具体的に理解できること、そのことによって変革が推進されることが期待される。また、経営だけでなく現場においても、エンジニアは開発した機能を除却することには心理的抵抗がある。それが改悪である可能性があること自体は理解しているケースが多いと思われるが、明示されない限りは、せっかく作ったのだから、除却したくないという心理は当然である。本研究では、可能中限り小刻みに、ソフトウェアの変更による効能を数値化し、改善・改悪を明らかにすることが重要であるとしている。明示的に改悪であることが分かれば、エンジニアは除却することが当然であると考えることになるだろう。また、経営にせよ現場のエンジニアにせよ、成功確率が低いマーケットにおいて、丁寧に除却することができれば成功率が低くても、収益性が向上し事業継続が可能になるということを理解するべきである。これは、企業の競争力の１つとなりうる。成功率が低くて、投資の撤退を余儀なくされるマーケットであっても、本研究の結果を活用すれば徹底しなくて良くなる。仮に他の企業が撤退したとして唯一生き残ることが出来れば、そのマーケットを独占できる可能性がある。このケイパビリティの獲得は明確に企業競争力となる。以上のようなことを理解し、合理的な判断を企業にもたらす。それが結果的に、社会のソフトウェア開発の生産性向上へと繋がることを期待している。

8.5 本研究の限界と将来研究への示唆

本研究の限界として、シミュレーションモデルが現実よりも単純であることが挙げられる。本研究のシミュレーションモデルは、１名の開発者が、１週間で１つの開発施策を開発可能であるという設定を行っている。さらに、１つの開発施策は、常に同じビジネスアウトカムを生み出すという設定である。しかし、現実には、１名の開発者ではな

く複数名存在することが一般的である。また、開発施策のサイズは一律ではなく、その施策が生み出すビジネスアウトカムは、一律ではない。そのため、より複雑なシミュレーションモデルの構築によって研究を拡張する余地が存在する。

また、第 6 章で触れたように、デリバリサイクルを短期化することは、リリース付帯作業を増加させることになる。本研究では、デリバリサイクル短期化のボトルネックと位置付けたが、自動化などを行っても、リリース付帯作業が完全にゼロになることは無く、自動化の程度次第ではあるが、多少の分割損は存在し続ける。この分割損についても、シミュレーションモデルに組み込む余地がある。

さらに、本研究における改善は、それぞれが独立していて相互作用を与えないとしている。しかし、実際には後続の改善によって、それまでの改善効果が失われることは起こりうる。それすらも含んで、AB テストの結果としては勝敗が付くため、考慮済みである可能性もあるが、そこについて検証を行っていない。この点については、シュミレーションモデルに組み込み検討を行う余地がある。

参考文献

agilemanifesto.org. 2001a.“アジャイルソフトウェア開発宣言.” アジャイルソフトウェア開発宣言. 2001. https://agilemanifesto.org/iso/ja/manifesto.html.

———. 2001b.“アジャイル宣言の背後にある原則.” アジャイル宣言の背後にある原則. 2001. https://agilemanifesto.org/iso/ja/principles.html.

Auer, Florian, Rasmus Ros, Lukas Kaltenbrunner, Per Runeson, and Michael Felderer. 2021. “Controlled Experimentation in Continuous Experimentation: Knowledge and Challenges.” *Information and Software Technology* 134 (106551): 106551.

Boehm Barry, Turner Richard, 原幹, 河野正幸, and 越智典子. 2004. アジャイルと規律 ～ソフトウェア開発を成功させる 2 つの鍵のバランス～. 日経 BP.

Brooks Frederick P., 滝沢徹（訳）, 牧野祐子（訳）, and 富澤昇（訳）. 2002. *The Mythical Man-Monsth 人月の神話*. 株式会社ピアソン桐原.

Chen, Lianping. 2018. “Microservices: Architecting for Continuous Delivery and DevOps.” In *2018 IEEE International Conference on Software Architecture (ICSA)*. IEEE. https://doi.org/10.1109/icsa.2018.00013.

Courtney, Hugh, Jane Kirkland, and Patrick Viguerie. 2009.“不確実性時代の戦略思考.” In *Diamond Harvard Business Review*, 64–80. Diamond.

Cusumano, Michael A. 2004. ソフトウェア企業の競争戦略（監訳 サイコム・インターナショナル）. ダイヤモンド社.

Cuypers, Ilya R. P., and Martin Xavier. 2010. “What Makes and What Does Not Make a Real Option? A Study of Equity Shares in International Joint Ventures.” *Journal of International Business Studies* 41 (1): 47–69.

Darrell, Rigby K., Jeff Sutherland, Hirotaka Takeuchi, and 倉田幸信(訳). 2016.“アジャイル開発を経営に活かす 6 つの原則 臨機応変のマネジメントで生産性を劇的に高める.” In ハーバード・ビジネス・レビュー. ダイヤモンド社.

Feitelson, Dror G., Eitan Frachtenberg, and Kent L. Beck. 2013. “Development and Deployment at Facebook.” *IEEE Internet Computing* 17 (4): 8–17.

Fitzgerald, Brian, and Klaas-Jan Stol. 2017. “Continuous Software Engineering: A Roadmap and Agenda.” *The Journal of Systems and Software* 123 (January): 176–89.

Humble, Jez, and David Farley. 2010. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. 1st ed. Boston, MA: Addison-Wesley Educational.

Humble, Jez, and Joanne Molesky. 2011. “Why Enterprises Must Adopt Devops to Enable Continuous Delivery.” *Cutter IT Journal* 24 (8): 6.

Johanssen, Jan Ole, Anja Kleebaum, Barbara Paech, and Bernd Bruegge. 2018. “Practitioners’ Eye on Continuous Software Engineering: An Interview Study.” In *Proceedings of the 2018 International Conference on Software and System Process*, 41–50. ICSSP ’18. New York, NY, USA: ACM.

Klotins, Eriks, and Tony Gorschek. 2022. “Continuous Software Engineering in the Wild.” In *Software Quality: The Next Big Thing in Software Engineering and Quality*, 3–12. Lecture Notes in Business Information

Processing. Cham: Springer International Publishing.

Klotins, Eriks, Tony Gorschek, Katarina Sundelin, and Erik Falk. 2022. "Towards Cost-Benefit Evaluation for Continuous Software Engineering Activities." *Empirical Software Engineer* 27 (6): 157.

Klotins, Eriks, and Einav Peretz-Andersson. 2022. "The Unified Perspective of Digital Transformation and Continuous Software Engineering." In *Proceedings of the 5th International Workshop on Software-Intensive Business: Towards Sustainable Software Business*, 75–82. New York, NY, USA: ACM.

Kurapati, Narendra, Venkata Sarath Chandra Manyam, and Kai Petersen. 2012. "Agile Software Development Practice Adoption Survey." In *Lecture Notes in Business Information Processing*, 111:16–30. Lecture Notes in Business Information Processing. Berlin, Heidelberg: Springer Berlin Heidelberg.

Leite, Leonardo, Gustavo Pinto, Fabio Kon, and Paulo Meirelles. 2021. "The Organization of Software Teams in the Quest for Continuous Delivery: A Grounded Theory Approach." *Information and Software Technology* 139 (106672): 106672.

Michlmayr, Martin, Brian Fitzgerald, and Klaas-Jan Stol. 2015. "Why and How Should Open Source Projects Adopt Time-Based Releases?" *IEEE Software* 32 (2): 55–63.

Porter, Michael E. 2001. "Strategy and the Internet." *Harvard Business Review*, March 1, 2001. https://hbr.org/2001/03/strategy-and-the-internet.

Racheva, Zornitza, and Maya Daneva. 2010. "How Do Real Options Concepts Fit in Agile Requirements Engineering?" *8th ACIS International Conference on Software Engineering Research, Management and Applications*, 231–38.

Rico, David F., Hasan H. Sayani, and Saya Sone. 2009. *The Business Value of Agile Software Methods: Maximizing Roi With Just-in-Time Processes and Documentation*. J. Ross Publishing, Inc.

Royce, Winston W. 1970. "Managing the Development of Large Software Systems: Concepts and Techniques." *Mathematics and Computers in Simulation*, 287–93.

Rust, Roland T., and P. K. Kannan. 2003. "E-Service: A New Paradigm for Business in the Electronic Environment." *Communications of the ACM* 46 (6): 36–42.

Savor, Tony, Mitchell Douglas, Michael Gentili, Laurie Williams, Kent Beck, and Michael Stumm. 2016. "Continuous Deployment at Facebook and OANDA." In *Proceedings of the 38th International Conference on Software Engineering Companion*. New York, NY, USA: ACM. https://doi.org/10.1145/2889160.2889223.

Schermann, Gerald, Jurgen Cito, Philipp Leitner, and Harald C. Gall. 2016. "Towards Quality Gates in Continuous Delivery and Deployment." In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, 1–4. IEEE.

Schermann, Gerald, Jürgen Cito, Philipp Leitner, Uwe Zdun, and Harald Gall. 2016. "An Empirical Study on Principles and Practices of Continuous Delivery and Deployment." *PeerJ*. PeerJ Preprints. https://doi.org/10.7287/peerj.preprints.1889v1.

Schermann, Gerald, and Philipp Leitner. 2018. "Search-Based Scheduling of Experiments in Continuous Deployment." In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*,

485–95. IEEE.

Schwaber, Ken, Mike Beedle, 嘉秀（監訳）長瀬, and 睦（監訳）今野. 2003. アジャイルソフトウェア開発スクラム. ピアソンエデュケーション.

Takeuchi, Hirotaka, and Ikujiro Nonaka. 1986. "The New New Product Development Game." *The Journal of Product Innovation Management* 3 (3): 205–6.

プロジェクトマネジメント協会. 2023. *プロジェクトマネジメント知識体系ガイド（PMBOK ガイド）第 7 版＋プロジェクトマネジメント標準: PMI 日本支部 監訳*. PMI 日本支部.

入山章栄. 2019. 世界標準の経営理論. ダイヤモンド社.

内閣府. 2021. "世界経済の潮流." 世界経済の潮流. 2021. https://www5.cao.go.jp/j-j/sekai_chouryuu/sh21-01/index-pdf.html.

刈屋武昭, and 山本大輔. 2001. リアル・オプション 新しい企業価値評価の技術. 東洋経済新報社.

小椋俊秀. 2013. "ウォーターフォールモデルの起源に関する考察 ウォーターフォールに関する誤解を解く." *商学討究 (Economic Review)* 64 号: 105–35.

岸知二, and 野田夏子. 2016. ソフトウェア工学. 近代科学社.

情報処理推進機構独立行政法人. 2020. "アジャイル領域へのスキル変革の指針 アジャイル開発の進め方." アジャイル領域へのスキル変革の指針 アジャイル開発の進め方. February 2020. https://www.ipa.go.jp/files/000065606.pdf.

日本プロジェクトマネジメント協会. 2014. *P２M標準ガイドブック 3 版*. 日本能率協会マネジメントセンター.

日本情報システム・ユーザー協会一般社団法人. 2020. "企業 IT 動向調査報告書 2020_Ver.4." 企業 IT 動向調査報告書 2020_Ver.4. March 2020. https://juas.or.jp/cms/media/2021/02/JUAS_IT2020_original_Ver.4.pdf.

玉井哲雄. 2008. "ソフトウェア工学の４０年." 情報処理 49(7): 777–84.

経済産業省デジタルトランスフォーメーションの加速に向けた研究会. 2020. "DX レポート 2 中間取りまとめ（本文）." DX レポート 2 中間取りまとめ（本文）. December 28, 2020. https://www.meti.go.jp/press/2020/12/20201228004/20201228004-2.pdf.

経済産業省デジタル産業の創出に向けた研究会. 2021. "DX レポート 2.1（DX レポート 2 追補版）（本文）." DX レポート 2.1（DX レポート 2 追補版）（本文）. August 31, 2021. https://www.meti.go.jp/press/2021/08/20210831005/20210831005-2.pdf.

長尾真., 石田晴久, 稲垣康善, 田中英彦, 辻井潤一, 所真理夫, 中田育夫, and 米澤明憲. 1990. 岩波情報科学辞典. 岩波書店.

日本情報システム・ユーザー協会（juas）, 一般社団法人. 2020. "企業 IT 動向調査報告書 2020_Ver.4." 企業 IT 動向調査報告書 2020_Ver.4. March 2020. https://juas.or.jp/cms/media/2021/02/JUAS_IT2020_original_Ver.4.pdf.

業績リスト

・査読付き国内・国際論文誌（２件）

大島 將義, 内平 直志. 2022. “インターネットサービスにおけるアジャイル開発が持つ不確実性の低下メカニズム.” 国際 P2M 学会誌 17 巻 1 号: 124–40. （第 4 章に対応）

Ohshima, Masayoshi, and Naoshi Uchihira. 2025. “Mechanisms for Improving Investment Efficiency through Continuous Delivery in Internet Services.” Journal of Systems and Software. （第 4 － 6 章に対応）※現在準備中

・査読付き国際会議（１件）

Ohshima, Masayoshi, and Naoshi Uchihira. 2023. “Mechanisms for Improving Investment Efficiency through Continuous Delivery in Internet Services.” In 2023 Portland International Conference on Management of Engineering and Technology (PICMET), 1–6. （第 5 章に対応）

・査読なし国内会議（２件）

大島 將義, 内平 直志. 2022. “インターネットサービスにおけるアジャイル開発が持つ不確実性の低下メカニズム” 国際 P2M 学会研究発表大会予稿集 2022 春季, p. 174-189 2022 年 4 月 23 日　第 33 回 国際 P2M 学会春季研究発表大会 千葉工業大学 津田沼キャンパス（第 4 章に対応）

大島 將義, 内平 直志. 2021. “アジャイル開発が持つ不確実性の低下メカニズム” 国際 P2M 学会研究発表大会予稿集 2021 秋季, p. 182-185 2021 年 10 月 24 日　第 32 回 国際 P2M 学会秋季研究発表大会 北陸先端科学技術大学院大学（第 4 章に対応）

・寄稿

大島 將義. 2021. “インターネットサービスにおけるアジャイルの位置づけ” 国際 P2M 学会研究発表大会予稿集 2021 秋季, p. PS_8-PS_12 2021 年 10 月 24 日　第 32 回 国際 P2M 学会秋季研究発表大会 北陸先端科学技術大学院大学（第 3 章に対応）

謝辞

本論文の執筆にあたり、多くの方々からご指導、ご支援を賜りました。ここに深く感謝の意を表します。まず、研究の指導を賜りました内平先生には、長期間にわたり温かいご指導をいただき、心より感謝申し上げます。先生の的確な助言と豊富なご知見に支えられ、本研究を進めることができました。研究が行き詰まった際にも、示唆に富んだアドバイスをいただき、最後まで粘り強く取り組むことができました。先生のご指導なくして、本論文の完成はあり得なかったと感じております。

また、最終審査において貴重なご意見をいただきました久保先生、橋本先生、藤波先生、西村先生に、深く御礼申し上げます。ご多忙のなか時間を割いて本研究をご審査いただき、鋭いご指摘と建設的なご助言を賜りましたこと、心より感謝いたします。先生方からいただいた貴重なご意見は、今後の研究活動にも大いに活かしてまいりたいと考えております。

さらに、本研究を進めるうえで、職場であるリクルートの皆様にも、多大なご理解とご支援をいただきました。本研究は、自分たち自身を理解する研究であったと考えます。私は、これからもソフトウェア開発を深く理解し、誰も見たことがないソフトウェア開発に挑戦を続けたい。そして、ソフトウェア開発を通じて、世界を少しでも良くし続ける存在でありたい。それを共に歩んでくれる仲間たちが居ることを誇りに思うとともに、その存在に大変感謝しています。

最後に、これまで支えてくれた家族に心より感謝いたします。明るく楽しい家庭であること。そして、やりたいことを尊重できる家庭であること。それらなくして、この道のりを歩み続けることはできませんでした。これからも、それぞれが自分のやりたいことを精一杯できて、それを助け合える家庭でありたいと願っています。