修士論文　　2003 年度（平成 15 年度）

インハウスにおける現場ユーザと協調した開発アプローチと開発支援ツール

On-site-user Collaborative Approach in In-house Development and Support Tool for the Development

慶應義塾大学大学院　政策･メディア研究科

小林　孝弘

修士論文要旨　　2003 年度（平成 15 年度）

インハウスにおける現場ユーザと協調した開発アプローチと開発支援ツール

論文要旨

近年、ユーザが快適に利用できるシステムが少ない。その理由として、ソフトウェア開発の現場においてアウトソーシングが横行し、ユーザと開発者の関係が疎遠になってしまっていることが挙げられる。

ユーザが快適に利用できるシステムを開発するには、インハウスにおいてユーザと開発者が密に協調した開発を行う必要がある。作成されるシステムは、ユーザの仕事をより快適なものにするための良き手段とならなければならない。そしてそのようなシステムの開発は、それを実際に利用するユーザの参加なしには成し得ない。

OCD（On-site-user Collaborative Development）は、インハウスの利点を活かし、ユーザと協調して業務支援システムを開発するための開発アプローチである。その目的はユーザの業務を快適にする有効な手段となる、ユーザが快適に運用できるようなシステムを開発することである。OCD では、システムを分割し、反復的に要求を収集し、その中で現場ユーザを教育することによりこの目的を達成する。

Toriaezer は、日本語によるシステムの要求定義から Java のソースコードを生成することにより、動作可能なシステムを素早く作成することを可能にするツールである。Toriaezer は OCD におけるユーザから反復的に要求を収集する過程において非常に効果的である。

キーワード

インハウス開発　ユーザとの協調　反復的な要求収集　医療情報システム　迅速な開発

慶應義塾大学大学院政策・メディア研究科

小林　孝弘

Abstract of Master's Thesis Academic Year 2003

On-site-user Collaborative Approach in In-house Development and Support Tool for the Development

Summary

In recent years, there are few systems that users can use comfortably. As the reason, outsourcing overruns in the spot of software development, and it is mentioned that the relation between a user and a developer is estranged.

In order to create the system that users can use comfortably, it is necessary to perform development with which the developer cooperated with the user densely in the in-house. The system created must serve as a good means for making user's work more comfortable. And development of such a system cannot be accomplished without participation of the user who actually uses it.

OCD (On-site-user Collaborative Development) is the development approach for developing an operating support system in harmony with users taking advantage of in-house. And the purpose is developing the system that serves as an effective means that makes user's business comfortable, and users can operate comfortably. In OCD, this purpose is attained by dividing a system, collecting demands repetitively and educating users in it.

Toriaezer is a tool that makes it possible to create quickly the system which can operate by generating the source code of Java from the demand definition of the system in Japanese. Toriaezer is very effective in the process in OCD that collects demands repetitively from a user.

Keywords

In-house Development, Collaboration with users, Repetitive demand collection, Healthcare Information System, Rapid Development

Keio University Graduate School of Media and Governance

Takahiro Kobayashi

第 1 章 はじめに 8
第 2 章 現場ユーザとの協調的開発アプローチ OCD 10
1. はじめに 10
2. OCD 12
2.1. 概要 12
2.2. 第 1 段階：業務知識の収集 14
2.2.1. 概要 14
2.2.2. 業務上の役割を軸とした観察 16
2.2.3. 業務の文脈の中での質問 17
2.2.4. 業務を支援するモノの調査 17
2.2.5. 業務に関わる現場レイアウトの調査 18
2.2.6. 業務モデルの構築 19
2.2.7. 現場ユーザとの信頼関係の構築 20
2.2.8. インタビューで業務知識を得ることの問題 21
2.3. 第 2 段階：要求の収集 22
2.3.1. 概要 22
2.3.2. ボトムアップな議論 24
2.3.3. 考え方の収集 25
2.3.4. 要求の一貫性確保 25
2.4. 第 3 段階：システム開発単位の分割 26
2.4.1. 概要 26
2.4.2. 継続的な修正に耐えうる規模 27
2.4.3. 他の単位との低依存性 28
2.4.4. 直感的なポリシー 28
2.4.5. 要求から逸脱しないポリシー 29
2.4.6. 分割のアーキテクチャ 29
2.4.7. 業務管理者とのレビュー 30
2.5. 第 4 段階：反復的なシステムの作成 30
2.5.1. 概要 30
2.5.2. サンプルの作成と定義 32
2.5.3. サンプルの評価 32
2.5.4. サンプルの修正 35
2.5.5. 修正と評価の反復 35
2.5.6. ユーザに対する教育 36
2.5.7. 業務管理者とのレビュー 37

2.6. 第 5 段階：建増し的なシステム導入 37
2.6.1. 概要 37
2.6.2. システムの現場への最適化 38
2.6.3. 現場ユーザの理解の確認 39
2.6.4. 建増し的な導入のメリット 39
3. OCD の実践 40
3.1. 適用対象 40
3.2. 第 1 段階：業務知識の収集 40
3.2.1. 方針 40
3.2.2. 外来受付業務の流れ 41
3.2.3. 外来受付業務の文脈の中での質問 42
3.2.4. 外来受付業務で利用されるモノ 43
3.2.5. 外来受付業務に関わる現場レイアウト 45
3.2.6. 外来受付業務における業務モデル記述 46
3.2.7. 医事課の反応 51
3.3. 第 2 段階：要求の収集 52
3.3.1. 方針 52
3.3.2. 院長との議論 52
3.3.3. 理事との議論 54
3.3.4. 医事課長との議論 55
3.3.5. 考え方のすり合わせ 56
3.4. 第 3 段階：開発単位の分割 58
3.4.1. 方針 58
3.4.2. 導き出された開発単位 60
3.4.3. アーキテクチャ 62
3.4.4. 業務管理者とのレビュー 62
3.5. 第 4 段階：反復的なシステムの作成 64
3.5.1. 方針 64
3.5.2. 外来患者請求システムの定義 65
3.5.3. 初期の要求 65
3.5.4. 最初のサンプル 65
3.5.5. 現場ユーザとの議論のポイント 67
3.5.6. 現場ユーザ教育のポイント 69
3.5.7. 最終的なシステム 70
3.5.8. 業務管理者とのレビュー 71
3.6. 第 5 段階：建て増し的なシステムの導入 72

3.6.1. 方針 72
3.6.2. 追加要求 72
3.6.3. 現場ユーザの理解確認 73
3.6.4. 外来患者請求システムの現状 74
4. OCD の評価 75
4.1. 評価方針 75
4.2. ステークホルダーの満足度に関する評価 76
4.3. 現場ユーザのシステムに対する理解度に関する評価 76
4.4. 今後の課題 77
5. まとめ 78
第 3 章 開発支援ツール Toriaezer 79
1. はじめに 79
2. 背景 80
2.1. 要求定義とクラス設計の間のギャップ 80
2.2. クラス設計実装時の問題 81
3. Toriaezer 82
3.1. 概要 82
3.2. Toriaezer における要求定義 84
3.2.1. 要求定義ファイルの記述 84
3.2.2. サービス指令文の記法 85
3.2.3. サービス指令文によるクラス設計 88
3.2.4. サービス指令文によるクラス設計の利点 90
3.3. ソースコード生成のメカニズム 91
3.4. Toriaezer を利用した開発手順とその効率 93
4. Toriaezer の利用例 94
4.1. 適用対象 94
4.2. 外来患者請求システム 95
4.3. 要求定義ファイルの記述 96
4.4. ソースコードの生成 98
4.5. Main プログラムの作成 101
5. Toriaezer の評価 102
5.1. 評価方針 102
5.2. 開発効率改善の度合い 102
5.3. 生成されたソースコードの品質 104
5.4. 今後の課題 106
6. まとめ 106

第 4 章 まとめ 108

第1章 はじめに

今日、ユーザが快適に利用できる情報システムは非常に少ない。筆者は現在、東京中野区の小原病院において、当院の業務を支援するシステムの開発に携わっているが、その中で他の医療機関のシステム導入状況や、使用しているシステムの評価などを耳にする機会が多く、そのように考えるに至った。

一般的に出回っている医療業務支援システムパッケージは、どれも高価であるにもかかわらず、本当に良いといえるものは非常に少ない。個人診療所レベルの医療機関が個人レベルでそれなりに便利に利用できるものは存在するが、複数の病院スタッフが協調して業務を行う上で便利に利用できるものは、非常に少ない。

また、ソフトウェア開発会社に自院で利用する業務支援システムの開発を委託するにしても、それを行うことで成功している病院は少ない。億単位の巨額の費用をかけて導入したシステムが結局使い物にならず、システムを導入することで逆に業務効率が落ちてしまったり、導入されたシステムの利用をやめてしまったりするケースが後を絶たない。

小原病院も、導入したシステムが使い物にならず、苦労している病院のひとつである。当院では 4 年前に、それまで利用していたレセコン（保険者に対する診療費の請求書であるレセプトを作成することを目的としたコンピュータ）を破棄し、現在の医療業務支援システムパッケージを導入した。当初の目的は、現状業務を物理メディアの制約から解放し、業務の効率化を図ることであった。

しかし導入したシステムの品質は劣悪であり、4 年がたった今でも、不備が絶えない。患者の診療情報が突然消える、データベースの番号体系が突然変更されるなどといった、致命的な問題がしばしば発生している。また、ユーザビリティは劣悪で、画面のレイアウトは煩雑を極め、ちょっとした誤操作が容易に致命的な問題に発展する。

そのため、病院業務を完全にそのシステムに依存させることができず、結局紙伝票や紙カルテといった物理メディアが全て残されることとなった。そしてシステムはそれと並行する形で利用されることとなった。レセコンを破棄してしまったため、レセプト作成業務をそのシステムによって行わなければならなかったためである。

従って、小原病院において現状の業務はきわめて非効率なものとなっている。

紙とシステムを併用しているため、相互に情報を移す作業が発生してしまっているのである。そしてそれに付随したミスも頻発するようになってしまった。導入したシステムは小原病院の業務を支援するどころか、逆に足を引っ張るものとなってしまったのである。

小原病院のみならず、このような状況は様々な場所で発生していると筆者は考えている。つまり、現状において導入先の組織に真の利益をもたらす情報システムは本当に少ない。

筆者は、この原因はシステム開発の現場からユーザが離れていっていることにあると考える。すなわち、ソフトウェア開発の現場では、ユーザのいないところでシステムに対する要求が議論され、ユーザのいないところでシステムが作成されるというやり方が普通に行われている。そして、近年の日本のソフトウェア業界におけるアウトソーシングの風潮はこれを助長している。

システムはユーザのために作られるということは当たり前であるが、最近のソフトウェア開発ではその前提が揺らいでいるように筆者は考える。ユーザにとって見れば、システムは自らの仕事を快適にこなすことを支援する手段である。開発者にとっての目的は、そのような手段を提供することでなければならない。そしてそのようなシステムの開発は、それを実際に利用するユーザの参加なしには作成し得ない。

このような考えのもと、筆者は小原病院において当院で利用する業務支援システムを自力で開発することを考えた。巷には良いシステムはほとんど出回っていないし、開発を委託することで良いシステムを得ることにも望みがない。

筆者はこうして、ユーザがその業務を快適に遂行するためのよき手段を提供すべく、ユーザと密に協調した開発を小原病院にて展開した。筆者はその中で、ユーザとの協調を効果的にする上で重要となる様々な知見を得ることができた。そして筆者はその知見を OCD（On-site-user Collaborative Development）という開発アプローチとしてまとめ、以降の開発の中で実践した。これは、開発の効率化およびリスク削減を主眼に置く近年主流の開発プロセスとはことなり、ユーザが快適に運用できるようなシステムを開発することを第一目標とした開発アプローチである。

本論文ではまずインハウス開発において現場ユーザと協調した開発アプローチ OCD について述べる。そして、OCD を実践する上で非常に有効なツール Toriaezer について述べる。本ツールは、動作可能なシステムをオブジェクト指向に基づいて素早く作成することを支援するためのツールである。これは、OCD を実践する中で、現場ユーザと協調した開発を行う際に効果的である。

第2章 現場ユーザとの協調的開発アプローチ OCD

1.はじめに

近年、ユーザが快適に利用できるシステムは少ない。特に医療情報システムの分野に関しては、電子カルテを始め医療業務の電子化が叫ばれて久しいが、医師や事務員が快適に利用できる情報システムは、未だ存在しないといっても過言ではない。莫大な費用をつぎ込まれて開発された医療情報システムが、導入先の病院で使い物にならず、それを導入することで逆に業務効率が落ちてしまったり、導入を取りやめてしまったりする例が後を絶たない。

その原因は、システム開発がそれを利用するユーザから遠ざかってしまっていることにあると考える。近年、ソフトウェア業界はシステム開発の効率化に躍起になっており、開発において最も重要であるはずのユーザからの要求収集を軽視しているかのようである。システム開発現場におけるシステムの要求に関する議論は、多くの場合上流工程の担当者とユーザ企業の窓口担当者との打ち合わせの中で行われる。実際に開発されたシステムを利用することになるユーザと、そのシステムの開発に携わる開発者が接触することはほとんどない。実際にシステム開発をしない上流工程の担当者と実際にシステムを利用しないユーザ企業の窓口担当者がシステムの要求を定義し、その開発自体は別の企業にアウトソーシングされている。

このような状況で、ユーザが快適に利用できるシステムは作成されない。ユーザが快適に利用できるシステムを作成するには、開発者とユーザがシステムの要求についてよく議論する必要がある。アウトソーシングの風潮の中でそのようなことを行うことは難しい。インハウスにおけるユーザと協調した開発を見直す必要がある。

OCD（On-site-user Collaborative Development）は、インハウスの利点を活かし、ユーザと協調して業務支援システムを開発するための開発アプローチとして筆者が考案したものである。OCD の目的は、ユーザの業務を快適にする有効な手段となる、ユーザが快適に運用できるようなシステムを開発することである。

ユーザが快適に運用できるシステムとは、全ての立場のステークホルダーがシステムに対して満足し、システムサポート対象業務の現場で実務に携わるス

テークホルダーが使いこなせるようなものでなければならない。このシステムサポート対象業務の現場で実務に携わるステークホルダーのことを以降現場ユーザと呼ぶ。全ての立場のステークホルダーが満足するようなシステムを作成するには、彼らの要求をよく収集し、それを忠実にシステムに反映させる必要がある。現場ユーザが使いこなせるようなシステムを作成するには、現場ユーザをよく教育し、システムを本質的に理解させる必要がある。

OCD には以下のような特徴がある。

A　反復的な要求の収集
B　現場ユーザの教育
C　システムの分割

Aについて、OCD において開発者は、システムサポート対象業務を管理する立場にあるステークホルダーからはインタビューまたはレビューという形で、現場ユーザからはシステム作成の過程における議論という形で、反復的にステークホルダーの要求を収集する。すなわち、業務管理者に対しては初期の要求に関するインタビューおよび開発における各段階の成果物についてのレビューを行い、現場ユーザに対してはともに協調してシステムを作成する中でその要求について議論することにより、ステークホルダーの要求を漏れなく確実に捕捉する。このシステムサポート対象業務を管理する立場にあるステークホルダーのことを以降業務管理者と呼ぶ。

Bについて、開発者は、現場ユーザと協調してシステムを作成する過程で、必要に応じてシステムおよびそれを実現するための主要な技術について現場ユーザを教育する。そして、システムに関してそれを踏まえた議論を行う中で、現場ユーザはシステムを本質的に理解できるようにする。

Cについて、OCD では業務をサポートするシステムを分割し、それぞれのシステムについて作成および導入を行う。システムを分割することにより、フォーカスが絞られるのでステークホルダーとシステムについて議論しやすくなる上に、開発者は開発の成果が短期間のうちに得られるために精神衛生上良い状態で開発に取り組むことが可能となる。また導入時には、システム不備などといったリスク要因による被害を小さくすることができるので、安全で着実な導入が可能となる。

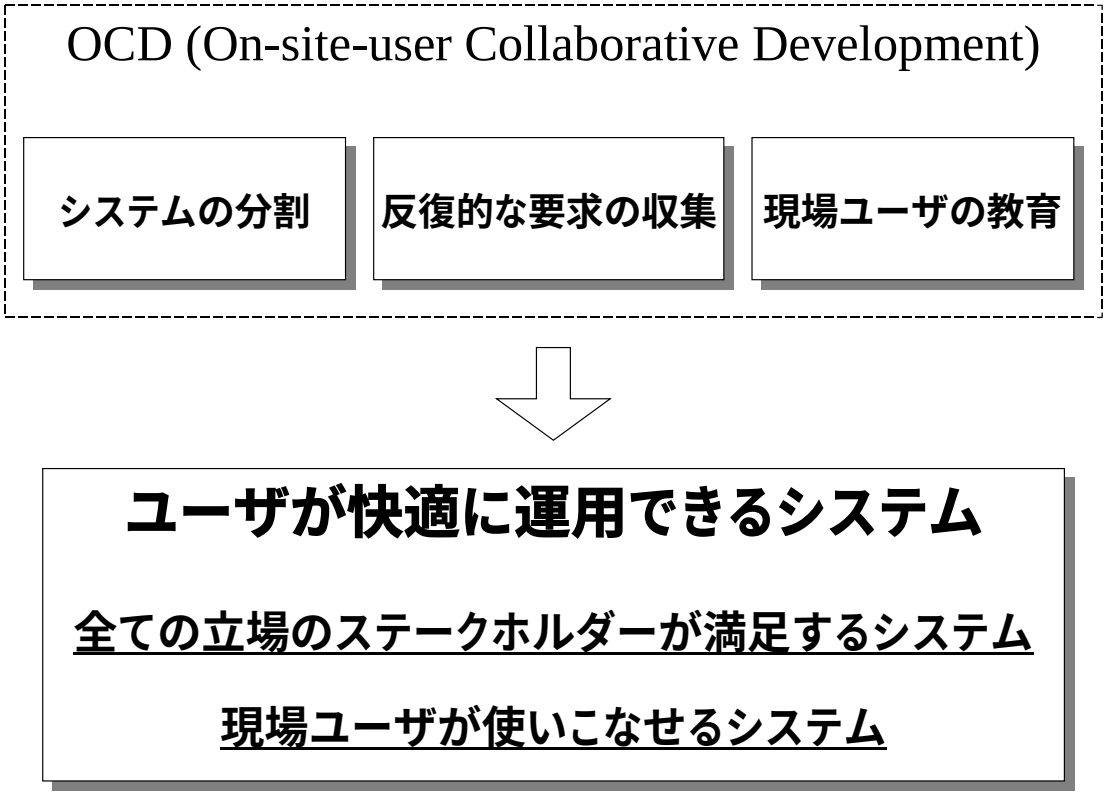


図 1　OCD (On-site-user Collaborative Development)

2.OCD

2.1.概要

OCD による開発は、開発者がまず現場で実際の業務を観察することにより、ステークホルダーと議論できるだけの知識的背景を得ることから始まる。このとき、開発者は業務上の役割の所在を軸とした観察を行い、質問があれば業務の文脈の中で現場ユーザに対して行う。観察の中で、業務を支援するモノおよび業務に関わる現場レイアウトについて合わせて調査を行う。このようにして得られた業務に関する知識をもとに業務モデルを構築する。それにより、開発者は自身に現場の業務についてのメンタルモデルを構築し、業務を本質的に理

解することが可能となる。なおこの過程は現場ユーザとの信頼関係を構築するという意味でも重要である。

ここで得た業務知識をもとに、開発者はシステムと関係の深い業務管理者と議論し、その要求を収集する。業務管理者から要求を収集する際にはインタビューを行うが、その際議論はボトムアップに行い、業務管理者の知識だけでなく考え方を収集するように気をつけなければならない。そして、立場の異なる業務管理者の要求の一貫性を確保するために、開発者は業務管理者間の意志の疎通を仲介する必要がある。

その後開発者は現場ユーザと協調して作成を行う上で扱いやすい単位にシステムを分割する。分割単位は、継続的な修正に耐えうる規模で、他の単位との依存性が低く、業務管理者の要求から逸脱しない直感的なポリシーのもとに識別されてなくてはならない。このとき、分割単位が稼動するためのアーキテクチャも同時に考慮しておく必要がある。分割単位は業務管理者とともにレビューし、そのポリシーが彼らにとって直感的で、かつ彼らの要求から逸脱していないかどうか、そしてアーキテクチャがその企業において実現可能であるかどうかを確認する。

分割されたそれぞれの単位について、現場ユーザと協調して反復的に開発を行う。その際には、個々の開発単位について、サンプルの作成と定義、サンプルの評価、サンプルの修正という手順を反復的に行う。この過程において、現場ユーザの細やかな要求を収集してシステムに反映するとともに、システムおよびそれを実現する主要な技術に関する本質的な知識について現場ユーザを教育する。この反復は多く行う程よい。反復の中で現場ユーザとよく議論することで、システムも現場ユーザも着実に成長する。

完成し次第現場に導入し、既に導入されているシステムに接続する。このとき、不備に対処したり、拾い切れていなかった実務上の要求を収集してシステムに反映したりといったシステム最適化のための作業を行う。また、この過程は現場ユーザのシステムに対する理解を確認する機会である。開発者は、現場ユーザが不備の原因を自分の分かる範囲で特定できているか、自分の追加要求をきちんと説明できているかといったことから、現場ユーザのシステムに対する理解度を確認する必要がある。

整理すると、OCD は以下のような段階から構成される。

第 1 段階：業務知識の収集
第 2 段階：要求の収集
第 3 段階：システム開発単位の分割
第 4 段階：反復的なシステム作成

第 5 段階：建増し的なシステム導入

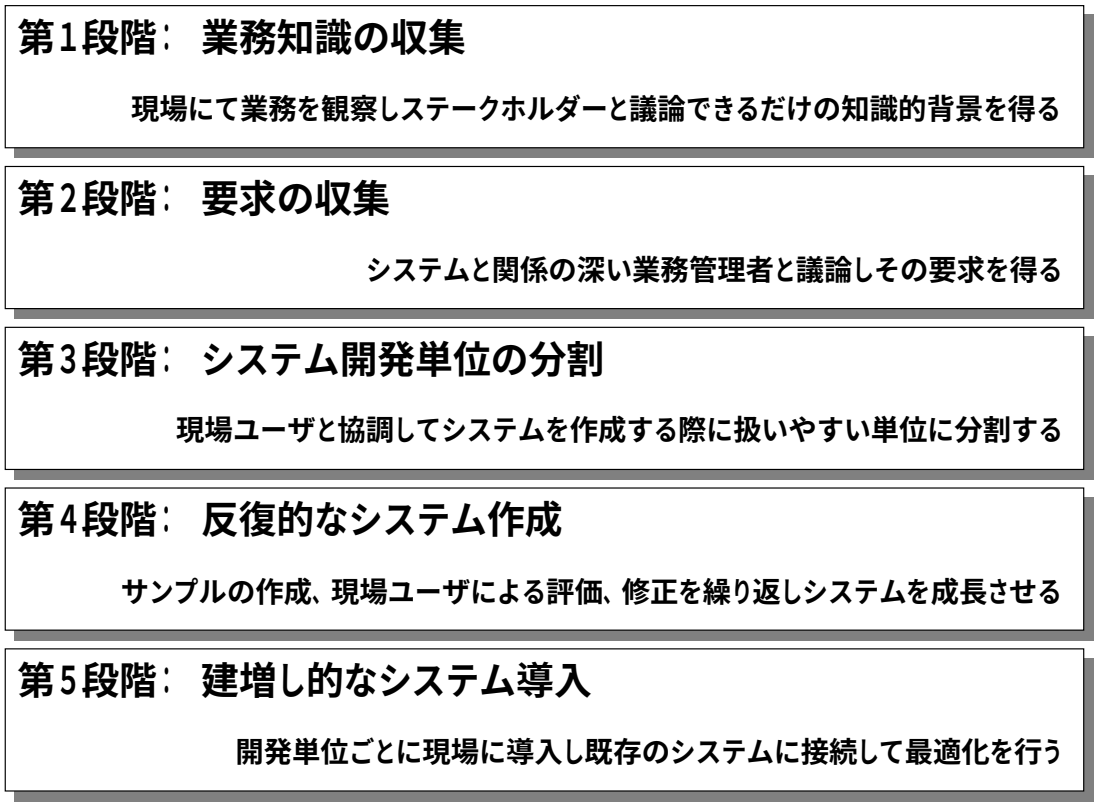


図 2　OCD の段階

2.2.第 1 段階：業務知識の収集

2.2.1.概要

この段階において開発者は、ステークホルダーと協調して効果的な開発が行えるだけの、知識的背景を身につける必要がある。すわなち、業務管理者の要求を理解するには、彼らの要求の裏づけあるいは適用対象となる現場の業務における事実を具体的にイメージできなくてはならない。現場ユーザとともにシステムを作成する際は、彼らの実務上の細やかな要求を理解するために、現場

あるいは業務における事実に基づいた問題点について、よく網羅された知識が必要である。そして、良いシステムを作り上げるために彼らと有意義な議論を行うには、業務に対する正しい本質的な理解が必要である。

このような知識的背景を身につけるためには、開発者は実際に業務が行われている現場に身を置き、自分の目で業務を観察しなくてはならない。現場において実際の業務を観察することで、開発者は業務に関する正しい知識を得ることができる。

そうして得た業務知識をモデルとして記述し、業務における抽象的な構造を帰納的に見出していくことにより、自身にメンタルモデルを構築することができる。そうして開発者は本質的に業務を理解できるようになる。

その理解を現場ユーザに問いかけることにより確認することで、自らの業務に対する理解を確認できるだけでなく、現場ユーザから業務のよき理解者としての信頼を得ることができるようになる。

現場での業務を観察するにあたり、以下の点に留意しなくてはならない。

① 業務上の役割を軸とした観察
② 業務の文脈の中での質問
③ 業務を支援するモノの調査
④ 業務に関わる現場レイアウトの調査
⑤ 業務モデルの構築
⑥ 現場ユーザとの信頼関係の構築

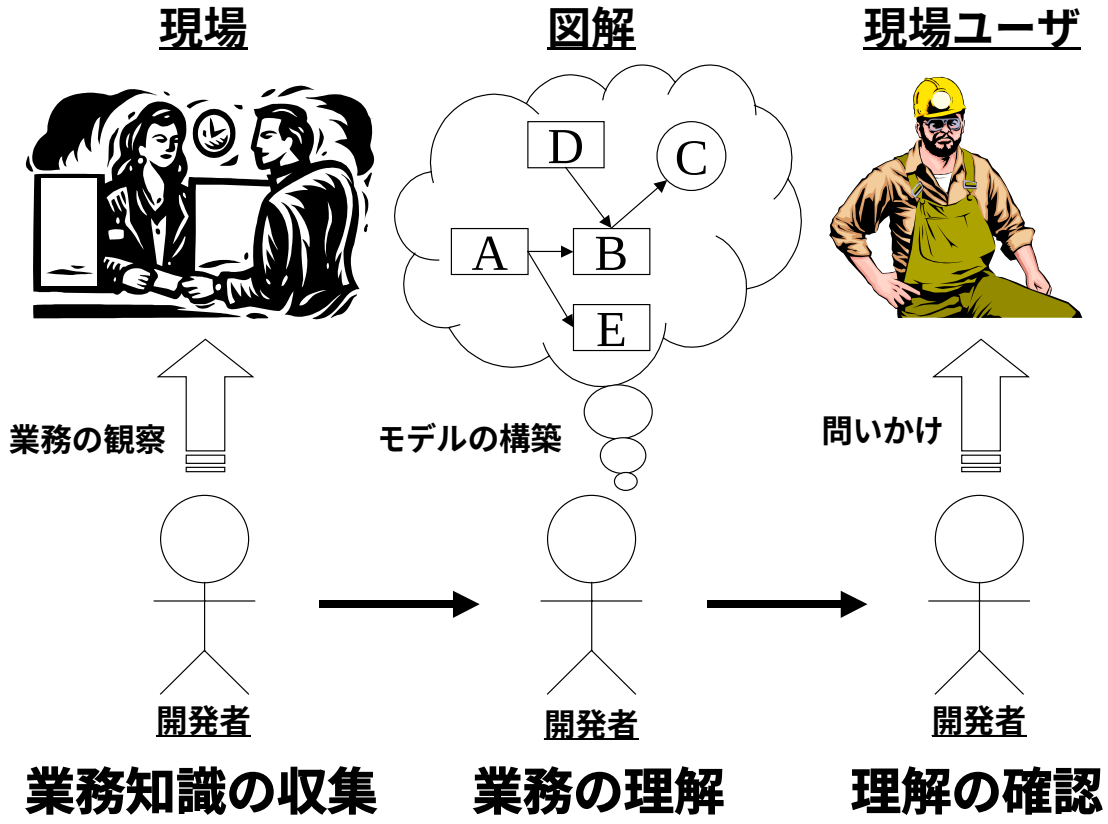


図 3　業務知識の収集

2.2.2.業務上の役割を軸とした観察

業務を観察するにあたり、業務上の役割が割り当てられた対象を軸として業務を見ると良い。これにより、役割が責任を持って遂行すべき、ある目的を持った一連の作業を観察できるので、業務に対する理解が深まる。業務を理解するには、その目的と、それを達成するための一連の作業を関連付ける必要があるからである。

業務上の役割を割り当てられた対象には、人および場所がある。例えば、受付係といった場合、その肩書きを与えられる人に受付を行うという役割が割り当てられることになる。また、受付窓口といった場合、その名前が与えられる場所に同様の役割が与えられることになる。無論、これらが両方存在することもある。受付窓口に、そこで専門的に業務にあたる受付係が存在する場合である。

もし役割が与えられた対象が一つしか存在しないのであればそれを軸とし、2

つ以上存在する場合は、より観察のしやすい方を軸として観察を行うと良い。これは現場の状況に応じて判断する必要がある。

例えば、ある業務において、役割は人と場所に割り当てられているが、現場ユーザはあまりに忙しそうに動き回っているので、観察がしにくいばかりでなく、観察することで業務の邪魔になる可能性がある場合、場所を軸にするべきである。逆に、その場所には数多くの現場ユーザが存在し、彼らによって業務が同時並行で行われている場合は、場所に張り付くよりも、個々の現場ユーザを軸として観察を行ったほうが効率的である。

2.2.3.業務の文脈の中での質問

業務に対する質問や、業務について自分が得た知識の確認は、その業務が行われる文脈の中で行う。それにより現場ユーザは、質問に対して精度の高い説明を返してくれるようになる。

その理由は、文脈の前提があるために、現場ユーザにとって答えなくてはならない範囲が明確であり、自分なりに説明を工夫する必要性が最小限になるからである。文脈の前提がない場合、質問を受けた現場ユーザがその質問の答えに相応な文脈を仮設定し、さらにその中で言い含める範囲を適切に判断して説明を行わなければならない。

文脈のないところで質問を受けた場合、現場ユーザは自分なりに事実を要約し、説明することになる。要約することで、現場ユーザにとって説明はしやすくなるし、開発者にとっても要点のみ知ることができるので一見効率が良い。

しかし、要約における、事実を解釈し重要な部分を取り出し再構成するという過程で、その現場ユーザの主観が多分に混入することになる。また、そもそも現場ユーザが要約した内容を開発者に対してきちんと説明できる保証はない。従って、得られる情報の正確性が損なわれてしまう可能性が高い。

2.2.4.業務を支援するモノの調査

業務の中ではしばしばモノが利用される。これらのモノとは多くの場合、業務を支援する目的で発生した、情報伝達あるいは記録のための手段であり、その業務において重要な役割を演じている。

例えば病院業務においては、カルテ、検査伝票、処方箋、診察券、付箋などが挙げられる。特に、カルテには、当該患者に対して行われた医療行為に関す

る情報を記録するという役割があり、検査伝票には、行った検査項目を関係者に伝達するという役割がある。

既存のモノのこのような役割を識別することは、業務を理解するうえで重要な要素となる。すなわち、その業務においてそれに関わる人々がどのように協調し、その間をどのような情報がどう行き交っているかを把握することができるためである。

モノの中でも特別な存在として、レガシーシステムがある。システムは業務の中で発生する作業の一部または全部を支援する目的で導入されるため、現状の業務モデルと密接に関係する。業務によっては、システムのサポートする業務モデルに合わせて人が動くというようなこともある。従って、レガシーシステムについて調査することは、現状の業務モデルを理解する上で非常に重要である。

このとき注意すべきこととして、開発者はレガシーシステムが業務の文脈にどのように位置づけられ、その中でどのような機能が利用されているかという観点に立って調査を行わなくてはならないということがある。レガシーシステム自身を調査することから始めてはならない。そもそもレガシーシステムの機能が業務の中で全て利用されているとは限らないし、機能単体で見ても、そのシステムの業務における意義を見出すことはできないからである。

また、レガシーシステムが業務に与えている影響についても調査しておく必要がある。そのシステムが業務においてボトルネックとなっている部分があるのであれば、その原因を調査し、新しく作るシステムにおいてその過ちを繰り返さないようにしなくてはならない。逆に、非常に便利に使えている機能があるのであれば、それを大いに参考にする必要がある。

2.2.5.業務に関わる現場レイアウトの調査

現場ユーザは業務の中で利用するモノに対するアクセスについての説明を行う際、物理的な位置関係でそれを表現することがある。特に、モノをおいた場所に名前をつけることができない場合、これは顕著である。

例えば、筆者が開発に携わった病院の医事課の会計窓口では、患者のカルテをその状態ごとに分類され、決まった場所に積まれていた。請求金額が確定していないカルテはモニタの上、請求金額が確定し会計の順番を待っているカルテは机の上、といった具合である。この例では、それぞれの状態のカルテ置き場に名前をつけることができない。従って、議論の際、現場ユーザは一貫して「机の上」あるいは「モニタの上」という表現を使っていた。

開発者は、このような表現に対応できるだけの知識を蓄えておく必要がある。現場ユーザが議論の際にこのような表現を使ったときに、すぐに具体的な場所を連想できなくてはならない。そのためには、現場を空間的に把握し、その中でもポイントとなる場所をよく認識しておく必要がある。

また、現場のレイアウトはしばしば業務におけるボトルネックとなる。例えば、プリンタがある場所から離れた場所で、頻繁にプリントアウトが発生する業務を行うのは明らかに非効率である。また、密な連携を必要とする部門がそれぞれ別の部屋に作業場を設けるのは効率が悪いだけでなく、部門の仕事の品質を落としかねない。

このような現場レイアウトにおけるボトルネックの解消は、開発の中で取り組むべき課題である。業務が行われる環境である現場レイアウトが適切でなければ、よいシステムを導入してもその効果は半減する。

2.2.6.業務モデルの構築

業務を本質的に理解するためには、開発者が自らの中に業務に対するメンタルモデルを構築しなければならない。そのためには、収集した業務知識を俯瞰的に眺め、その中から業務における原理原則を見出すということを行う必要がある。現状の業務の様子をモデルとして記述することが、これを行う上で有効である。

モデルの記述は継続的に行わなければならない。現場で業務に関する新たな知識を得るたびに、随時モデルを更新する必要がある。これを行うことで、モデルは洗練され、精度の高いものとなっていく。なお、OCD においてどのようなモデル記述言語を使用するべきか、現段階においては規定していない。ただその選択の指針として、具体的な事象を漏れなく記述するための表現力に富み、その記法は直感的であることが望まれる。

モデル化の作業には開発者の主観が多分に入り込むことになるので、その正しさについて現場にて確認する必要がある。すなわち、現場にて確認したい業務が発生した際に、開発者はその業務について自分の理解の正しさについて現場ユーザに尋ねるのである。そうすると現場ユーザは、今発生した業務に対する開発者の理解が正しいかどうかという明快な観点に立って考えることができるので、精度の高い返答が可能になる。これを受けて、開発者は必要に応じてモデルに修正をかける。

現場ユーザの確認の際は、自分の言葉で確認することが重要である。開発者が業務に対する自分の理解を現場ユーザに対してきちんと説明でき、その説明

を現場ユーザが正しいと判断することが重要である。これにより、開発者は自分の理解が正しいことを確認することができ、現場ユーザは開発者がその業務を理解したことを確認することができる。

なお、ここで注意すべきことは、モデルはあくまで業務理解のための手段であるということである。開発者は、モデル記述の作業を通して、自身の中に業務に対するメンタルモデルを構築する。従って、ここで作成したモデルは今後開発者が業務知識を思い出すために利用することはあっても、システムの作成に直接結びつくことはない。従って、モデル自身の完成度を上げることにこだわることは本質的でない。それよりも、それが表現するモデルの妥当性を確保し、自身が業務を本質的に理解することに集中するべきである。

2.2.7.現場ユーザとの信頼関係の構築

この段階において、開発者が知識を得ることは非常に重要であるが、それと同じくらい重要なことは、現場ユーザが開発者を、自分たちの業務の良き理解者として認め信頼することである。OCD では、現場ユーザの開発に対する積極的参加が必要である。しかし、本来の自分の業務がある中で、開発に積極的に参加してもらい、慣れない議論につき合わせるには、現場ユーザに相当のモチベーションが必要である。

現場ユーザが開発に参加するモチベーションとして、現場ユーザと開発者の信頼関係が必須である。常日頃、使いにくいシステムの利用を押し付けてくる業務管理者の手先などではなく、自分達の味方となり、自分達の業務を自分達に最適な形で支援するシステムを作ってくれる「正義の」開発者として信頼されるようにならなければならない。現場ユーザと協調した開発は、それによって初めて成功を見る。この信頼関係がないままに OCD を進めても、そのうち現場ユーザとの関係は途切れ、従来となんら変わりない開発手法にいつの間にかシフトしていくであろう。

そのような信頼関係を得るためには、この段階において、現場にて真摯な姿勢で業務を観察し、現場の実情を理解することに取り組むことが必要不可欠である。その中で、業務に対する自分の理解を絶えず現場ユーザに確認し、現場ユーザに観察の成果を知らしめる必要がある。そうして、現場ユーザは自ずと開発者を信頼し、開発に積極的に協力してくれるようになる。

2.2.8.インタビューで業務知識を得ることの問題

現場ユーザに対してインタビューすることによりこのような知識を得るというやり方も考えられるが、これは奨励できない。第３者の説明によってこのような知識を得ることは、以下のような理由により非常に難しい。

- 主観の混入
- 文脈の伝達
- 日常化した作業の伝達

（1）主観の混入

インタビューにおいて得られる情報には、インタビューイの主観の混入を避けられない。人に何かの説明を求める以上、その説明には必ずその人の考え方、あるいは解釈が反映される。その上、インタビューにおいての関係は一方的で、インタビューアはインタビューイの言うことを受け入れるしかない。すなわち、インタビューでは、得られる情報に主観が入り込むことを前提としなくてはならない。

インタビューで得られる知識は、事実であるというよりはむしろ、事実に対するインタビューイの解釈である。この解釈が正しい保証はないし、そもそもインタビューイがその解釈を正確に説明できる保証すらない。このような状況で、正確な業務知識を得ることは難しい。

（2）文脈の伝達

文脈のないところで文脈に特化した作業について説明することは難しい。先にも触れたが、現場ユーザの業務は、その文脈に大きな影響を受けるため、インタビューの際に自分が説明しようとする業務についての文脈を説明しなければならない。

車のないところで車の運転の仕方を説明するのが難しいのと同じように、全ての文脈を網羅した漏れのない説明をインタビューアに期待することはできない。特に、システム化によって効率化を図る必要のある業務であるため、業務には無駄や冗長な部分が多く想定される。そういったものをありのままに説明するのは至難の業であり、膨大な時間を要す。

（3）日常化した作業の伝達

日常化もしくは身体化してしまった作業について説明することは難しい。業務で非常に重要な作業を、現場ユーザが無意識のうちにこなしているということがよくある。現場ユーザにしてみれば、そのような作業はもはや習慣になっており、特に関心を払う必要がなくなっているため、このような事態が起こる。

例えば、筆者が開発に携わった病院の医事課では、外来患者を受付けた際に受け取った診察券を、その患者のカルテの中で、医師が本日分の診療情報を記入するページに挟み込む。これは医師がその患者を診察する際に、記入すべきカルテのページに素早くアクセスできるようにするための医事課の配慮である。この事実から重要な要求を導くことができる。すなわち、医師は診察開始時にすぐにカルテを記入できるような状態が必要であるということである。

筆者は、OCD の考え方をもとに開発を行う以前に、現場ユーザに受付業務の流れについてインタビューをしたことがあったが、このような情報を得ることはできなかった。診察券をカルテに挟み込む作業が現場ユーザにとってもはや習慣化しており、取るに足らないことになってしまっていたためである。後に、OCD に従って現場を観察した際に、この事実を発見することができた。

2.3.第 2 段階：要求の収集

2.3.1.概要

この段階において開発者は、業務管理者がシステムに望むことあるいは実現したい世界観について聞き出す。その目的は、システムの業務支援の方向性および、システムが満たすべき条件を得ることである。

この段階では、システムの導入に関して権力を持っているステークホルダーから要求を収集する必要がある。現場ユーザに関しては、多くの場合この限りではない上に、システムを作成する段階で十分な議論を行うこととなるので、ここで要求を収集する対象とはしない。

ここで得る内容には、どの部分にどのような業務モデルに基づいてシステムを導入し、どのようなアウトプットが得られる必要があるかといった、システムを開発する上で不可欠な要求から、システムを導入した後でどのような効果を期待するかといった感覚的な要求か含まれる。従ってそのような情報を得る

ためには、システムに対して重要な役割を担う立場にある全ての業務管理者の意見を収集する必要がある。

業務管理者の意見を収集するには、業務上のそれぞれの立場における代表者に対して個別にインタビューする方法が現実的である。要求は現状における事実を、それぞれの立場において解釈することにより発生するので、開発者がこれを現場にて自ら得ることはできない。但し、解釈対象となった事実を理解できるだけの知識を現場にて得ておくことは、彼らの解釈の仕方を理解し、そこから導き出された要求を評価する上で非常に重要である。

インタビューの対象者は、ユーザ企業側に、業務におけるそれぞれの立場から満遍なく適任者を選出してもらうと良い。例えば開発するシステムが医療情報システムである場合、理事の代表者、院長、システムと密接な関係のある課の課長などが挙げられるであろう。

業務管理者に対するインタビューを行う際には、以下点に留意する必要がある。

① ボトムアップな議論
② 考え方の収集
③ 要求の一貫性確保

図 4　要求の収集

2.3.2.ボトムアップな議論

インタビューに際して、具体的な内容からボトムアップに議論を展開すると、質の高いインタビューが可能になる。インタビューイはふつう、突然「どんなシステムが欲しいですか？」と聞かれてもうまく応えられない。どの範囲のことをどのくらいの具体性を持って話せばよいか分からないからである。

現場の業務に関する具体的な議論からはじめることによって、インタビューイは常日頃考えている自分の思いを漏れなく引き出すことができるようになる。特に、現場を観察する中で発見した問題意識をもとに議論すると効果的である。インタビューイが自分では気がついていなかった要求を、開発者が抱いた客観的な問題意識に関する具体的な議論の中で自分の要求を連想し、引き出すことができるようになる。

2.3.3.考え方の収集

考え方の裏づけとしての知識は重要であるが、それがメインになってしまってはいけない。その裏づけを踏まえ、インタビューイがどう考えるかが最も重要であり、それを収集することがインタビューの目的である。そういう意味で、インタビューイの言っていることが知識なのか意見なのかに注意しておくことが必要である。知識の議論に傾倒しているようであれば、必要に応じて流れを引き戻さなければならない。さもないと、インタビューが世間話で終わってしまうことすらある。

特に、現場に近い立場にある業務管理者は、業務における例外に関する知識について話したがる傾向がある。業務に対する外的な要因の影響力が強い場合、この傾向は顕著である。例外に関する知識は重要であるが、要求の収集という目的にはそぐわない。業務管理者の話す例外の詳細についてはその知識が得られる情報へのポインターをもらうにとどめ、そのような例外に対して業務管理者がどのように対処したらよいと考えているかを聞き出さなければならない。

2.3.4.要求の一貫性確保

OCD を行う際のリスクとして、立場の違う業務管理者による要求の食い違いが発生するという問題がある。特に、OCD では全ての立場のステークホルダーから要求を収集し、それを全てシステムに反映させることを目標としているため、このような食い違いの発生する確率は高い。

業務管理者の要求は常に妥当であるとは限らない。特に現場と離れたところにいる業務管理者には、現場に対する理解が不十分であることがある。同じように、現場に近い立場にある業務管理者には上層部の真意が伝わっていないことは十分ありえる。

開発者はこの中で、両者のパイプ役となることが重要である。すなわち、インタビューの際に、インタビューイである業務管理者に対して、立場の違う業務管理者の考え方を伝える必要がある。そしてその上で、インタビューイあるいは食い違った要求を申し立てた立場の異なる業務管理者の要求が妥当であるかどうかについて議論する必要がある。従って、この過程は必要に応じて反復的に行う必要がある。すなわち、業務管理者同士の要求の食い違いを発見し次第、相互の意見を伝えるということを行わなくてはならない。

またこの段階において開発者は、現場ユーザの代弁者としての役割も担う必要がある。すなわち、現場で得た実際の業務に関する知識をもとに、現場に対

する理解が希薄な業務管理者に対し現場の実情を伝えるのである。
　このように開発者は、ステークホルダーの要求を首尾一貫したものにするために、ステークホルダー間のパイプ役として奔走しなければならない。そうすることで、業務管理者同士の意思疎通が可能となり、システムの方向性をひとつにすることが可能となる。

2.4.第 3 段階：システム開発単位の分割

2.4.1.概要

　このステップにおいて開発者は、システムの開発単位を適切な単位で分割する。その目的は、ユーザにとっても開発者にとってもシステムを扱いやすくすることである。すなわち、ユーザにとってはシステムの単位が小さくなることによってフォーカスが狭まり、要求を考えやすくなる。開発者にとっては作成の単位が小さくなるので成果を早く得ることができ、精神衛生上良い状態で開発を行うことができる。これにより、作成の段階においてシステムが成長するスパンを早めることが可能になる。

　システムを分割する単位は以下のような条件を備えている必要がある。
① 継続的な修正に耐えうる規模
② 他の単位との低依存性
③ 直感的なポリシー
④ 要求から逸脱しないポリシー

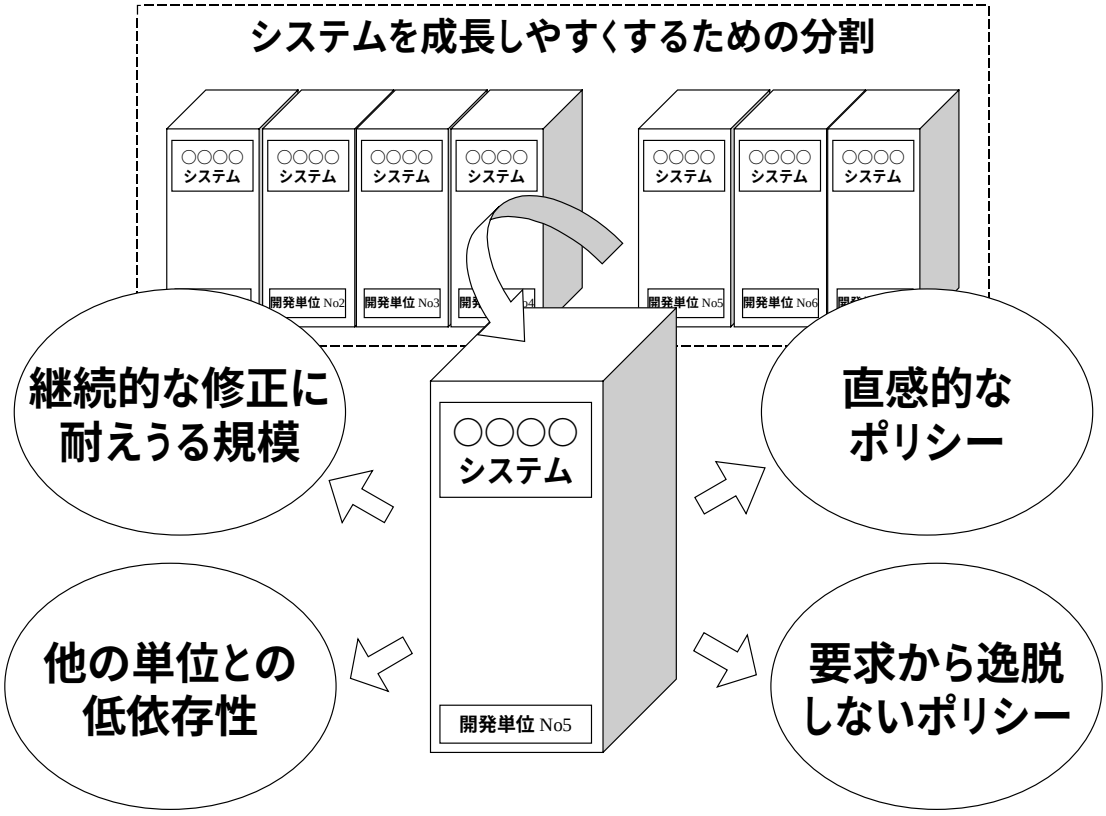


図 5　開発単位の分割

2.4.2.継続的な修正に耐えうる規模

システムが成長するスパンを短くするためには、開発者のシステム作成における負担を軽減することが必要になる。そのためには、開発者にとって手ごろな規模の単位を識別する必要がある。これはシステムの品質を保つ上でも重要である。すなわち、分割単位は開発者が品質にこだわる余裕を十分に確保できる程度にコンパクトであることが望ましい。

また現場ユーザにとっても、規模が大きいシステムを評価するのは難しい。そのようなシステムは理解するのに時間がかかる上に、システムの全体像を頭にとどめて置くことが難しい。そのため、質の高い意見を述べることができなくなる。

分割規模の具体的な目安としては、大体７つ前後の本質的な画面で構成される単位が手ごろである。現場ユーザにとっては、画面＝システムである。その数が頭の中で十分に管理できる範囲に収まっていることで、現場ユーザはシステムに対し精度の高い評価を行うことができるようになる。そして開発者にと

っても、システムのインターフェースである画面が自分の頭の中で管理できるため、全体像を見失わない円滑な開発が可能である。

システムに対する要求や論理的な構造といった抽象的な基準での規模の測定は、開発者の恣意性を多分に含むことになるため避けるべきである。また一般的によく用いられるソースコードの行数といった基準も、システムを作成する上で本質的でないので避けるべきである。

画面数という基準は開発者にとっても現場ユーザにとっても具体的で分かりやすく、解釈のずれが発生する可能性が低い。すなわち、ある画面数のシステムに対し、開発者と現場ユーザが同程度のイメージを持つことが可能である。そして７つ前後の画面は、現場ユーザにとっても開発者にとっても現実的な基準である。

2.4.3.他の単位との低依存性

これは現場ユーザとの議論の際、考慮すべきシステムの制約条件を減らすことができるという意味で重要である。ある分割単位についてシステムを作成する際、密接に関連している他の分割単位がある場合、現場ユーザは議論の際にそれを考慮に入れなければならなくなる。システム開発の経験がない現場ユーザにとって、システム同士の関連をイメージさせることは難しい。従って、こうした制約条件は、現場ユーザと協調作業を行う上で有害である。

また OCD では、それぞれの開発単位が完成し次第、順次建て増し的に導入していくというアプローチをとることになるが、これはその際のリスクを軽減するという意味でも重要である。すなわち、導入対象のシステムを既に導入されているシステムと関連付ける際に、それらの結合の度合いが低ければ低いほど、関連付けの作業は楽になるし、トラブルが発生した際の被害を小さくすることができる。逆にそれらのシステムが密接に関連していた場合は、関連付けの作業に多くの時間をとられるようになり、トラブル時にはそれによって致命的な被害を受ける可能性が高くなる。

2.4.4.直感的なポリシー

現場ユーザがシステムをよく理解でき、積極的に意見できる状況を得るためには、分割単位を現場ユーザによって直感的なものにしなければならない。そのために分割単位は、適切な粒度で、意味的によくまとまったものである必要

がある。

システムの粒度が高すぎると、その解釈が広がるため、現場ユーザにとってシステムを具体的にイメージすることが難しくなる。その結果、議論の都度、開発者に対してそのシステムのイメージを確認しなければならなくなる。これはすなわち、開発者が自分の都合でシステムの解釈をコントロールできてしまうことを意味する。つまり、開発者がシステム作成に際して立場的に優位に立つようになってしまうのだ。

これは現場ユーザと協調した開発を行う上で、避けるべきことである。現場ユーザから要求を徹底的に収集し、彼らが快適に運用できるシステムを開発することが OCD の主旨であるのにも関わらず、開発者が優位にシステムを作成してしまっては本末転倒である。

意味的に十分にまとまっていないと、これも同様の問題を引き起こす。すなわち、現場ユーザが直感的にシステムの役割を理解できないため、システム作成における開発者の優位性を助長してしまうことになる。

現場ユーザと開発者の力関係を対等にするということは、協調作業する上で非常に重要である。従って分割単位は、それを最大限保証するものでなくてはならない。

2.4.5.要求から逸脱しないポリシー

分割単位はシステムが実現する世界観に大きく影響を受ける。

例えば、医療事務で利用するシステムにおいて、患者に対する請求項目をその発生源にて自動判定し、請求金額を自動的に算出する自動会計の考え方を導入する場合と、請求項目を事務員が一旦確認し、適切な修正を加えた後に請求金額を算出する手動会計の考え方を導入する場合では、分割の方針は変わってくる可能性が高い。

すなわち、この２つの業務モデルについて、これまで述べてきたことを考慮しながら分割を行った場合、その結果は異なるはずである。従って、分割を行う際には、ステークホルダーの要求をよく吟味し、その方向性から逸脱しないように注意する必要がある。

2.4.6.分割のアーキテクチャ

このような基準の下で行った分割を実現可能なものにするために、分割した

システム同士が協調動作するためのアーキテクチャについても、この段階で考慮しておかなければならない。すなわち、分割したシステム郡を機能させるために、その論理的な協調関係および、それらが稼動するために必要なインフラについて考慮しておく必要がある。

2.4.7.業務管理者とのレビュー

分割作業を終え、それぞれの開発単位についてシステムの作成を始める前に、分割の方針とアーキテクチャについて、業務管理者のレビューを受ける必要がある。分割の方針が業務管理者の意向に沿っているかどうか確認し、分割のアーキテクチャがその企業において実現可能かどうかを確認するためである。また、分割の方針が業務管理者にとってイメージしやすいものであることを確認することも、現場ユーザと協調したシステムの作成を始めるにあたって重要である。この結果を受けて、必要に応じて分割方針の修正を行い、業務管理者の了承を得た時点でシステムの作成を開始することになる。

なお、この段階における分割方針は、あくまで現時点での最適解であるという位置づけである。今後現場ユーザと協調してシステムを作成していく中で、変更が発生することは十分にありえる。ただ、この分割方針において業務管理者と合意に達したという事実が、後々に変更を考慮する際の軸となるという意味で重要である。

なお、要求収集のインタビューのときと同様、この段階で現場ユーザのレビューを受ける必要はない。これまで集めてきた業務管理者の意向を、現場ユーザにとって最も良い形で実現するということを、次の段階において行うためである。

2.5.第 4 段階：反復的なシステムの作成

2.5.1.概要

この段階において開発者は、それぞれの開発単位について、業務管理者の要求を満たすシステムを現場ユーザと議論しながら反復的に作成する。その目的は、現場ユーザの細やかな要求を織り込みながらシステムを成長させるととも

に、現場ユーザがシステムを本質的に理解できるようにすることである。

この段階では、開発者と現場ユーザが密に協調しながら開発を進める。その協調の仕方とは、サンプルを囲んだ議論を反復的に行うことである。それにはまず開発者がサンプルを作成し、それを現場ユーザに提案する形で見せる。現場ユーザは、そのサンプルに表現されたシステムが実際の業務の中で機能し、かつ自分らが快適に利用できるかどうかを評価する。サンプルが現場ユーザの納得のいくものになるまで、これを繰り返し行う。

具体的には以下のような手順を踏む。

① サンプルの作成と定義
② サンプルの評価
③ サンプルの修正
④ 修正と評価の反復

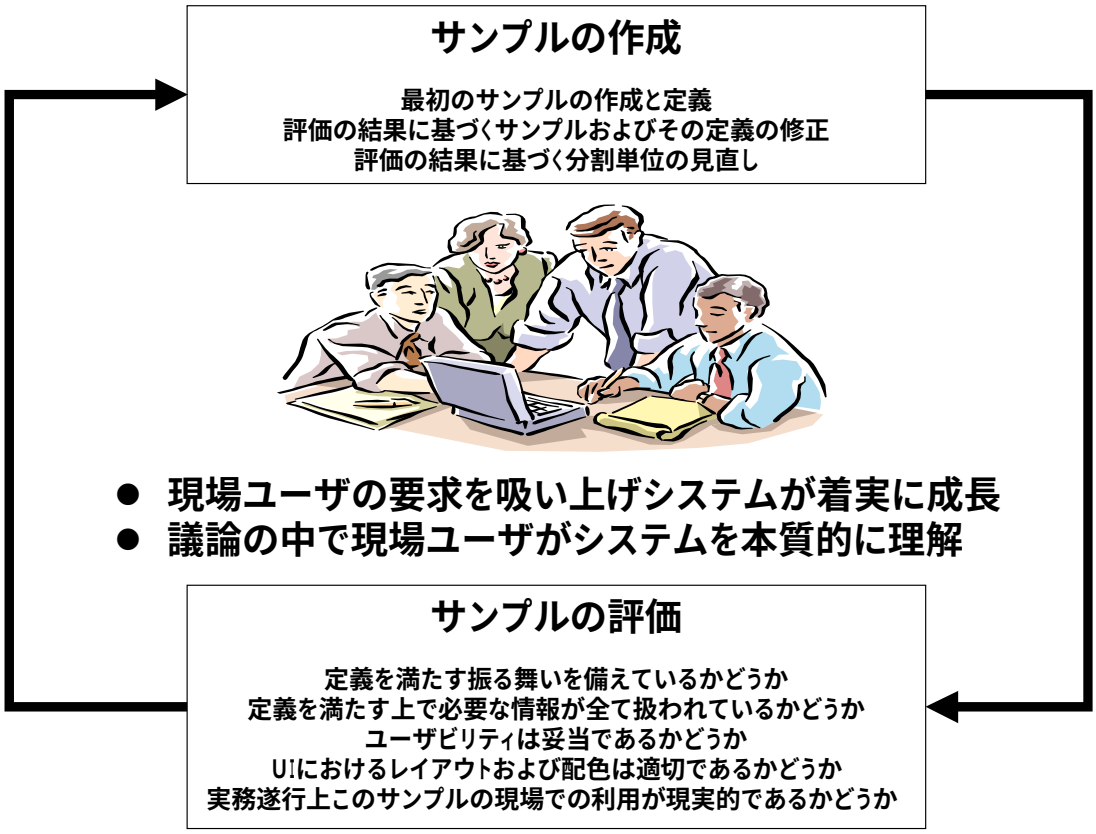


図 6　反復的なシステム作成

2.5.2.サンプルの作成と定義

現場ユーザとの議論に先立ち、システムの具体的なサンプルを用意することは必須である。これがないと現場ユーザはシステムを具体的にイメージすることができないので、システムに関する議論は不可能であると言っても過言ではない。

また、このサンプルには現場ユーザにとって明確で直感的な定義を与える必要がある。この定義とはすなわち、そのシステムが業務において何をサポートして何をサポートしないかについての境界について定めたものである。また、もしサンプルの機能に制限がある場合は、その制限についても定義しておく必要がある。

この定義が曖昧だと、やはり現場ユーザはシステムをうまくイメージできず、効果的な議論ができない。現場ユーザにとって理解が難しい定義であった場合も同様である。この定義がどうしてもうまくいかない際は、分割方針が適切でない可能性があるので、前の段階にもどって分割を再検討する必要がある。

このステップでは、これまで得てきた業務知識と業務管理者の要求をもとに、ひとつの開発単位について、開発者が自分で最初のサンプルを作成する。このサンプルの理想的な形は、実際に動作可能なシステムである。しかしそれが困難な場合は、少なくともシステムの振る舞いを忠実に再現した画面のモックアップを用意する必要がある。サンプルは実際に即していればいるほど、次のステップにおいて確実な評価を得ることができるようになる。

2.5.3.サンプルの評価

このステップはシステムを成長させる上で最も重要である。このステップにおいて、現場ユーザとともにサンプルを詳細に評価し、ステークホルダーの意向のもと、現場ユーザが最大限快適に利用できるようにシステムを成長させるための案を練る。そしてその中で、現場ユーザにそのシステムを快適に運用できるための能力を養わせる。

現場ユーザとともにサンプルを評価する際の指針には、以下のような項目について検討する必要がある。

Ａ　定義を満たす振る舞いを備えているかどうか
Ｂ　定義を満たす上で必要な情報が全て扱われているか
Ｃ　ユーザビリティは妥当であるかどうか

Ｄ　ＵＩにおけるレイアウトおよび配色は適切であるかどうか
Ｅ　実務遂行上このサンプルの現場での利用が現実的であるかどうか

これらの項目について、病院の受付業務を支援するシステムのサンプルの例を想定して説明を行う。なお、このシステムの定義は、「受付した外来患者の情報を管理するシステム」であるとする。そして、このシステムは WEB ブラウザをインターフェースとした WEB アプリケーションシステムとして実現するものとする。

まずＡについて、この項目の目的はシステムに必要な機能の漏れを捕捉することである。仮にサンプルには外来患者の個人情報および保険情報の登録機能しか備わっていなかった場合、おそらく現場ユーザは、過去半年分の保険証確認年月日を管理する機能を要求するだろう。

Ｂについて、この項目の目的はシステムで扱うべき情報の漏れを捕捉することである。仮にサンプルでは患者についての個人情報と保険情報しかを扱うことができなかった場合、おそらく現場ユーザは、コメントを扱うことができるようにすることを要求するだろう。そしてそれに付随して、コメントは永続的なものとその日限りのものを用意し、それらを編集できる機能を要求するだろう。

Ｃについて、この項目の目的はシステム操作上の不快感を捕捉することである。サンプルのインターフェースは WEB ブラウザであるため、入力フィールドの移動にはデフォルトではタブキーを利用することになる。しかし、現場ユーザはおそらく、リターンキーでの移動を可能にするよう要求するであろう。

Ｄについて、この項目の目的はシステムにおける見た目の不快感を捕捉することである。仮にサンプルは 10.5 ポイントのフォントを使用して表示を行っていた際、現場ユーザはおそらく 12 ポイントのフォントを使用することを要求するだろう。

Ｅについて、この項目の目的は現場ユーザの現時点におけるシステムに対する期待度を捕捉することである。現場ユーザはこのような状況でシステムの利用を開始するのは難しいと判断するに違いない。そこで現場ユーザは管理者に対し、これまで述べてきた内容どおりにサンプルを修正することを要求することになる。

ここで注意すべきことは、業務管理者の意向に沿いかねるような提案を現場ユーザから受けたときである。

上の例において、仮に、第 2 段階で収集した業務管理者の要求の中に、受付業務を最大限自動化することが含まれていたとする。つまり、業務管理者は現

場ユーザがシステムに情報を入力することを極力減らしたいと願っていたとする。

ここで問題となるのは、Ｃについてサンプルを検討した際に現場ユーザが要求した、リターンキーでの入力フィールドの移動である。入力を極力減らしたいという業務管理者の要求がある中で、入力フィールド移動について入念なサポートを行うことは、一見業務管理者の意向を無視しているかのように映る。

開発者は、現場ユーザはあくまで現状の業務モデルの中で、自分らの作業を効率化したいと願っていることを考慮しておかなければならない。すなわち、リターンキーによって入力フィールドを移動したいという要求は、診察券や保険証の情報を転記しなければならないという現状の業務モデルを反映している。転記を行うには、左手に診察券や保険証を持たなければならないので、右手でシステムの操作が完結するような状況を願っているのである。

しかし、業務管理者の意向を実現すれば、入力フィールドの移動はほとんど必要がなくなる可能性がある。診察券や保険証から直接情報を読み取れるような仕組みを用意すれば、ほとんど転記の必要はなくなる。

異なるレベルのステークホルダーによる、このような一見相反する要求に対し、OCD では、抽象度の高い要求の枠の中で、より具体的な要求を実現するというアプローチをとる。この例では、業務管理者の受付自動化という要求は、現場ユーザの入力フィールド移動に関する要求よりも抽象度が高い。従って、最大限入力を自動化するが、自動化しきれない部分についてはタブキーでの入力フィールド移動を可能にする、という結論になる。

ここでの妥当な判断として、開発者は、リターンキーによる入力フィールド移動の可能性を残しつつ、とりあえずは入力フィールドを最大限なくす方針でサンプルを作り直すことになる。その中で転記を撲滅することができれば、リターンの移動は考慮する必要がなくなる。しかし、いくらか入力フィールドが残ってしまった場合、リターンキーでの移動を検討する必要が出てくる。そのように現場ユーザに説明する必要がある。

また、この過程は現場ユーザに対して教育を行うタイミングである。開発者は、この過程における現場ユーザとの議論の中で、システムに関する必要な知識をユーザに提供するように心がけなければならない。この過程を通して、現場ユーザはシステムを本質的に理解できるようになる。この詳細については後述する。

2.5.4.サンプルの修正

ここでは、現場ユーザとの議論の中で得られた要求を、サンプルに反映する作業を行う。これには必要に応じてサンプルの定義、またはシステムの分割方針をも見直す。

修正作業はできる限り素早く行う必要がある。現場ユーザが前回のサンプルに対する印象を忘れないうちに、次のサンプルを見せるようにしなければならない。そのためには、現場ユーザとの評価の中で得られた要求を達成するサンプルを、とにかく作り上げるという割りきりが必要である。すなわち、ドキュメント作成やパフォーマンステストは後回しにし、ユーザの要求を満足した、見せることのできるサンプルを作成することに全力をつぎこまなくてはならない。

また、この段階において特に注意すべきことは、開発者が作業に取り組んでいる間にも、現場ユーザとのつながりを絶やさないことである。

サンプルを素早く作るよう努力することは重要であるが、修正が難しいときや、その量が多いときなどは、この作業にはどうしても時間がかかってしまう。その間現場ユーザとの間に何のやりとりもないと、自ずと現場ユーザの中で前回のサンプルに対する印象が薄れてしまう。現場ユーザには、現在作成中のサンプルを、前回のサンプルと比較して評価してもらう必要があるため、これは避けなければならない。また、あまりに修正の期間が長くなってしまうと、現場ユーザが開発に対して不信感を抱くようになってしまう。現場ユーザとの良い信頼関係を保つ上で、このような状況は好ましくない。

このような状況を回避するには、WEB やメールを活用し、現場ユーザに対して頻繁に進捗報告を行うようにすると良い。すると、現場ユーザに対して開発者が何を行うことに時間がかかっているのかを明らかにすることができるので、不信感を抱かせる可能性は低くなる。

2.5.5.修正と評価の反復

この反復の中で、システムは徐々に成長し、現場ユーザはシステムおよびそれを実現する主要な技術についての理解を深めていくことになる。従って、この反復をできる限り数多く行えるよう、サンプルの作成または修正作業を効率化する必要がある。

システムが十分に成長した段階においては、現場ユーザとともに実業務での利用を想定したテストを行い、実用性を検証するとよい。これには、現場ユー

ザにシステムを利用した実際の業務をシミュレーションしてもらうと良い。その中で、システムに必要なパフォーマンスや負荷耐性といった要求を把握することができる。

現場ユーザから十分に実用に耐えうるという評価を受けたら、業務管理者に導入の許可を仰ぐ。業務管理者は、そのシステムがその定義の範囲において自分らの要求を満たしているかどうかを確認し、必要に応じて修正を要求する。ここで業務管理者の承認が得られた場合、この段階は終了する。

2.5.6.ユーザに対する教育

この段階は、現場ユーザがシステムに対する理解を深める上で非常に重要である。特に、サンプルの評価の過程における開発者と議論を通して現場ユーザは知識を蓄えていく、逆に開発者は、現場ユーザとの議論を通して、必要に応じて現場ユーザを教育しなくてはならない。

例えば、前述の WEB アプリケーションとして実装する「受付した外来患者の情報を管理するシステム」の例において、受付された患者のリストをリアルタイムで閲覧したいときに、頻繁に画面の情報を更新しなくてはならないという問題に現場は気がつくであろう。これはサンプルが表現しているシステムが採用したアーキテクチャが WEB ベースであったということに起因する制約である。

このアーキテクチャを採用する以上は、ユーザは定期的に再読み込みを実行しなくてはならない。定期的に、自動で再読み込みさせることはできるが、仮に同じ画面に入力フィールドがあった場合、入力をしている最中に再読み込みが実行されてしまうと、入力しかけの情報が消えてしまったり、入力フィールドのアクティブ状態が解除しまったりするという意味で非常に厄介である。

この問題はまさしく教育のタイミングである。すなわち、WEB アプリケーションというアーキテクチャにおける制約と、それを踏まえたシステムの実装についてである。WEB アプリケーションはサーバとの接続状態を保持しない方式で通信を行うので、何か情報が必要なときは、クライアントがその都度 WEB サーバに要求を発行しなければならない。サンプルではこれを踏まえ、ユーザが要求の発行を実行する形で実装されている。自動的に要求を発行することも可能だが、それだと入力フィールドが初期化されてしまう可能性があるためである、といった具合である。

2.5.7.業務管理者とのレビュー

ここでは、最終的に完成したシステムが十分に業務管理者の意向を反映しているかどうかを確認する。もし業務管理者がシステムの現状に納得しないようであれば、その原因を明らかにし、前の段階に戻って現場ユーザとともに作成をやりなおす必要がある。

2.6.第 5 段階：建増し的なシステム導入

2.6.1.概要

導入は、開発単位ごとに建て増し的に行う。すなわち、それぞれの開発単位の作成が終了するたびに順次導入し、既に導入されているシステムと関連付けるということを行う。そして、少しずつ現場の業務がシステムに置き換わっていくという形にする。

建増し的にシステムを導入することのメリットは、導入時におけるシステム不備に対するリスクを軽減するだけではなく、開発期間中におけるステークホルダーおよび開発者を精神衛生上良い状態に保つという意味がある。

この段階において、開発者は以下のようなことを行う必要がある。

① システムの現場への最適化
② 現場ユーザの理解確認

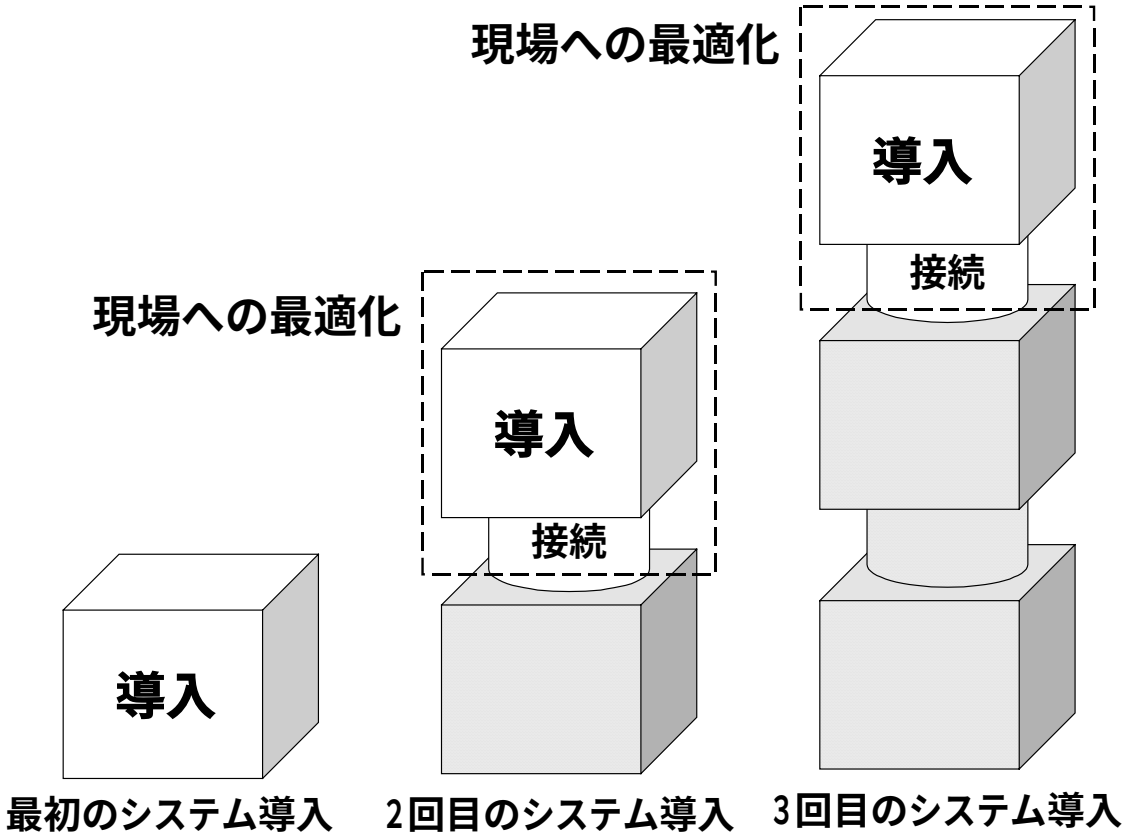


図 7　建増し的なシステム導入

2.6.2.システムの現場への最適化

この段階において開発者は、システムを現場に導入し現場ユーザがそれを快適に運用できるように調整する。その目的は、システムのサポートによって現場ユーザの業務が確実に支援され、かつステークホルダーの要求を確実に満たすように最適化するとともに、現場ユーザがシステムに対する理解を確認することである。

OCD においては、導入時も、現場ユーザの要求を収集する過程である。すなわち、現場に導入して実際に利用することで初めて見えてくる要求は必ず存在する。そしてそのような要求は非常に本質的であることが多い。そのような要求を捕捉し対応することが、現場ユーザに彼らの業務を快適なものにするためのシステムを提供する上で重要である。

2.6.3.現場ユーザの理解の確認

現場ユーザがシステムに対する理解を確認するうえでこの段階は重要である。実際にシステムを利用し始めることで、システムに対する理解が必要な場面は必ず訪れる。

例えば、病院の医事課において、システムを利用してある書類を印刷しようとした際、想定外のプリンタから出力されてしまったとする。このプリンタは別の用途で利用していたものであり、この書類を出力するために必要な書式がセットされていない。このような場合、そのシステムにおいてプリンタの選択がどのようにして行われているかに関する知識が必要である。仮にそのシステムが WEB アプリケーションであり、WEB ブラウザの機能を利用して印刷を実現していた場合は WEB ブラウザの印刷設定を修正する必要がある。あるいは、そのシステムが独自に印刷機能を備えていた場合は、その設定を修正する必要がある。

このような事態が発生した際に、現場ユーザが自分の知識を活用して対処できるかどうかを、この段階において確認する必要がある。

2.6.4.建増し的な導入のメリット

建増し的にシステムを導入することのメリットは、導入を分割して行うことで不備が発生した際の被害および復旧にかかるコストを軽減することができるというだけではない。導入結果を早く得ることでステークホルダーを安心させるとともに、開発者の開発に対するモチベーションを維持させることができるという意味がある。

ステークホルダーに、OCD による開発アプローチでの成功を早期段階に体験させておくことにより、彼らの開発に対する信頼度を向上させることができる。現場ユーザは早期段階で自分らがコミットした開発に対する結論を得られるため、自分らのコミットが無駄になっていないということを確認することができる。業務管理者にとっては、システムが現場に導入されていくたびに開発が失敗に終わる可能性が小さくなるため、安心して見ていられるようになる。これはステークホルダーの協力体制を維持するために非常に重要である。

また開発者は、導入を経るごとにノルマをひとつひとつクリアしているという実感を得ることができるので、開発を通してモチベーションを維持する上で非常に有効である。また、システムが導入されればされるほど開発失敗のリスクが減っていくので、安心して開発に取り組むことができるようになる。開発

者が精神衛生上良い状態を保たれることは、結果として品質の高いシステムの作成を可能にする。

3.OCD の実践

3.1.適用対象

筆者はこれまで、110 床の療養型病床を持つ小原病院にて、その業務を支援するシステムの開発に取り組んできた。その中で筆者は特に、同病院医事課における業務を支援するシステムの開発を担当しており、その際に OCD のアプローチのもとに開発を行った。以降では、医事課業務を支援するシステム開発において、OCD を実践した例について述べる。

3.2.第 1 段階：業務知識の収集

3.2.1.方針

医事課は、患者の受付・会計、カルテの管理、保険者への診療報酬請求、入院相談、病室管理といった、小原病院における全ての事務的な業務を担当する部署である。筆者は、開発にあたりこの医事課において現場業務の観察を行った。

医事課において、業務上の役割は場所に対して割り当てられていた。すなわち、受付に関する業務は受付窓口付近にて行われ、会計に関する業務は会計窓口付近で行われるといった具合である。個々の事務員には明確な役割が存在せず、それぞれの場所で発生する業務を、全員で臨機応変に対応することで業務が成り立っていた。

従って筆者は、医事課内の場所に張り付き、観察を行うことにした。医事課において役割が割り当てられた場所には、大きく分けて、外来受付窓口、会計窓口、入院受付窓口、会計計算デスクが存在した。以降では、特に外来受付窓

口で行われている業務について観察した際の例をもとに、説明を行うことにする。
なお、医事課にて収集した業務知識の詳細については付録１の「業務調査報告書」を参照されたい。

3.2.2.外来受付業務の流れ

外来受付窓口において業務を観察したところ、そこで行われる業務の流れは、次のようになっているということが分かった。

患者が外来受付窓口にやってくると、事務員はまずその患者が初診であるかどうか判断する。初診であると判断した場合は診療申込書への記入および保険証（老人医療受給者証あるいは何らかの公費負担医療受給者であることを示す証明書を含む）の提出を求める。特に、患者が内科の受診を希望している際は、問診表への記入を求める。再診であると判断した場合は、診察券を受け取り、月初めであれば保険証の提出を求め、希望受診科についてたずねる。このとき患者は必要に応じて、希望の受診形態を申告する。例えば、処方の希望やリハビリの希望がこれに該当する。
事務員は次に、その患者のカルテを準備する。初めて小原病院にかかる患者（新規患者）の場合は、診療申込書の情報をもとに新しいカルテと、診察券を作成する。カルテには必要に応じてその患者の会計に関する属性を表すハンコが押される。そして、患者の ID を記載した番号札を大小 2 枚作成し、カルテ裏表紙のポケットに入れる。既に小原病院にかかったことのある患者（既存患者）の場合は、外来受付窓口背後にあるキャビネットから、その患者のカルテを取り出す。
またカルテには、必要に応じて医師に伝達するための情報を添付する。患者に希望受診形態がある場合は、そのことを示す札をカルテに挟み込む。前回の診察時に行った検査の結果が出る場合や、実行を指定された診療行為がある場合は、同様にそれを示す札を挟み込む。また、会計上定期的に算定が可能な診療行為がその日に算定可能な際は、その旨を示すハンコをカルテに押す。その他に特に伝達する必要のある情報がある場合は、付箋を使用する。最後に、その患者の診察券を、カルテにおける今日の診察情報記入欄があるページにはさみこむ。
これと同時に事務員は、システムに受付した患者の情報を登録する。新規患者の場合は診療申込書の情報をこのシステムに転記し、新規患者登録を行う。

そしてその患者が希望している診察科に、受診登録を行う。再診の患者の場合は、患者 ID を入力し、同じく受診登録を行う。

この過程が終了すると、事務員はその患者のカルテを受診科に届ける。その際事務員はカルテから 2 枚の番号札を取り出す。その番号札には患者の ID が記されており、診察順の管理に利用される。事務員は小さい方の番号札を患者に渡し、大きい方をその患者の受診科の掲示板に掲げる。なお、この掲示板にはその科での受診を待っている全ての患者の番号札かかかっており、一人診察が終了するたびに、看護婦が並び順を修正している。番号札の処理が済むと、カルテはその科の看護婦に手渡される。

3.2.3.外来受付業務の文脈の中での質問

筆者は外来受付業務を観察するに当たり、積極的に質問を行った。最初に一通り業務を見て、次に見たときに納得のいかなかったものに関しては、すぐに近くの事務員に質問した。以下はその例である。

（1）初診の判断基準に関する質問

最初に質問の対象となったのが、初診の判断である。診察券を提示したにも関わらず、初診の扱いを受けた患者を見たときに、初診再診の判断の基準について、そのときの担当事務員に聞いてみた。すると以下のような答えが返ってきた。

初めて小原病院で受診する患者は例外なく初診となるが、そうでない患者の場合、場合によって初診にも再診にもなりえる。患者が始めて来院した際は無条件に初診となる。しかし、患者がある病気で整形外科に通院中だが、風邪をひくなどして内科にかかった場合、これは内科について初診という扱いとなる。また、通院中に３ヶ月以上間隔が空いてしまった場合も、初診という扱いになる。例外的に、再診としてある科を受診したときに、別の病気が見つかった等の理由で、その科の医師に別の科を受診するよう指示があった場合も、その科について初診となる。この場合は後に看護婦から報告を受け、会計計算デスクの事務員が処理を行う。

ここでさらに、初診再診はなんのために識別する必要があるのかと聞いて見たところ、初再診はそもそもレセプト上の請求項目であるとのことだった。すなわち、初診であるか再診であるかによって保険者に請求できる額がかわってくるので、識別する必要があるのだ。

（2）黒いラインの入ったカルテに関する質問

新規患者のために新しいカルテを作成する際、カルテの表紙の上端に黒いラインを引いている場面に遭遇した。その作業を行っていた事務員にその目的について聞いたところ、以下のような答えが返ってきた。
自費で受診する場合はその診療情報をレセプトに反映してはいけない。これをやってしまうと、後に診療報酬明細書を書き直すなどといった非常に面倒なことを行う必要があるため、保険診療としての診療情報と明確に区別しなければならない。従って、保険に加入しているが、特別な事情によって自費で診察を受けなければならない場合、既にその人のカルテが作成されていても、新たに自費用のカルテを作成する。

このように文脈の中で質問を行うことで、質問したことに対して非常に分かりやすく詳細にとんだ返答を受けることができた。また、こちらとしても目の前で起こったことについての説明であるため、理解が容易であった。

3.2.4.外来受付業務で利用されるモノ

外来受付業務では以下のようなモノが利用されていた。

名前	役割
保険証	保健医療を受診する資格があるかどうか調べる
老人医療受給者証	老人医療を受診する資格があるかどうか調べる
番号札小	患者が自分のIDを把握する
番号札大	患者が（番号札小のIDと照らして）自分の待合順を把握する
カルテ	患者に関する全ての診療情報を記録する
診察券	再診患者が受診するにあたって自分のIDを事務員に提示する
問診表	内科を受診する初診患者が自分の様態を記述する
診療申込書	新規患者が自分の個人情報を記述する
診察券代用	患者が診察券を忘れた際に診察券の代わりに使用する
カルテキャビネット	カルテを収納する
順番待ち掲示板	患者に診察順を通知する
診察券箱	再診患者が診察券を投入する
保険証箱	再診患者が月初めに自分の保険証を投入する
Professional Doctor	現状業務で利用されているレセプト作成を支援するシステム
プリンタ	新しいカルテを印刷する
薬札	患者が処方を希望している旨を医師に伝える
リハビリ札	患者がリハビリを希望している旨を医師に伝える
採血札	採血をする必要のある患者を示す
検査結果札	本日診察時に検査結果が出る患者を示す
各種ハンコ	患者に関する会計情報や指導料の有無などについて必要に応じてカルテに押される

表 1　医事課で利用されているモノ

外来受付業務には、システムに対して受付患者の情報を登録するという作業が含まれている。このシステムは病院業務においてどのような位置づけにあり、登録情報がこれ以降の業務の中でどのように利用されるかについて調査を行った結果、以下のようなことが分かった。

小原病院では、4 年ほど前に導入された、Professional Doctor という医療業務支援システムが稼動している。小原病院では、このシステムを基幹業務システムとして採用しているが、実際のところ、このシステムは業務において十分に機能していないようであった。

本システムは、紙メディア主導の病院業務を電子メディア主導に切り替えることで、業務効率を向上することを目的としていた。すなわち、カルテや伝票といった情報を全て電子的に扱うことで、職員の情報に対するアクセシビリティを高めるとともに、情報の物理的移動の必要性をなくすことで、職員同士のコミュニケーションを円滑にし、結果として業務スピードが向上することを期待していた。

しかし、本システムはそのような目的を達成する上で十分な機能と品質を備

えておらず、本システムの現状の主な用途は、医師に対する外来患者受付情報の通知、検査の依頼、処方箋の発行、毎月末のレセプト作成といった、非常に断片的な業務サポートにとどまっている。従って、このシステムのみでは業務を完全に遂行することはできないため、結局のところ紙メディア主導の業務は変わらず、そこにシステムを上乗せした格好になってしまっている。

受付業務において登録した患者情報は、システムを利用した業務全てにおいて利用される。すなわち、医師に通知する受付情報、レセプト作成、検査依頼、処方箋発行に必要な患者情報には、ここで登録した情報が利用される。

このシステムに対する現場の評判は非常に悪い。システムに対する情報の入力はレイアウトに問題があるため非常に面倒であるし、ちょっとした誤操作がデータ紛失やシステムのダウンといった致命的な問題に容易に発展するのだ。従って現在では、本当に限られた機能を安全がある程度確認された方法で利用しているという状況である。それでもアーキテクチャ上の根本的な問題により、致命的な問題が発生することがあるという始末である。従ってこのシステムは明らかに業務におけるボトルネックとなっている。

それでも、処方関係の機能は便利に利用できているようである。いわゆる約束処方という機能で、患者の処方箋を作成する際に、過去に処方した内容が自動的に呼び出されるという仕組みである。この機能は、外来診察においてなくてはならない機能という位置づけになっているようである。

また、毎月のレセプト作成はこのシステムがなければできない。そのための機能自体の評判は悪いが、レセプト作成業務はこのシステムに完全に依存している。逆に言えば、これ以外の業務においてこのシステムが依存されているということはない。レセプト作成業務以外は全て紙メディアによってカバーされているためである。レセプト作成業務は病院経営上最も重要な業務であるため、このシステムが否応なしに利用されているというのが実情である。

3.2.5.外来受付業務に関わる現場レイアウト

受付窓口は、医事課の中で最も病院入り口に近い位置にあった。受付窓口にはロビーに面した方に患者用のカウンターがあり、医事課内にはカウンターと向き合う形で数台のデスクが配置されていた。

カウンターは患者が受付に必要な作業を行う場所である。そこには診察科ごとに区切られた診察券箱と、診療申込書、問診表、そして患者が診察券を忘れた時に利用する用紙が配置されていた。

事務員はデスクにて受付に関わる作業を行う。デスクには、一台の端末と、カルテ作成時に使用する各種ハンコ、白紙の番号札、等が配置されていた。

医事課内から見てデスクの背後と左側には、カルテを収納するためのキャビネットが並んでいた。デスクの右側の方には、診察券作成機と診察形態を示す札が納められた筒が配置されていた。そして新規カルテを印刷するためのプリンタは医事課において、受付窓口とちょうど反対側の壁沿いに設置されていた。

このことから分かることとして、プリンタの位置は明らかに不適切である。現状において、出力された新規カルテをとりに行くためには、席を立ち、様々なものでひしめきあった医事課中央部を通り抜けなければならない。およそ１０歩の道のりであるが、通路は狭く、医事課では絶えず事務員が動き回っているので、プリントアウトされた新規カルテをとって帰ってくるまでに数十秒の時間を要してしまう。これは明らかに業務におけるボトルネックとなっている。

3.2.6.外来受付業務における業務モデル記述

モデルの記述に際して、筆者は Contextual Design[1]で紹介されている Work Model のアプローチを採用した。Work Model は、特定の現場ユーザの業務を５つの視点から記述するものである。すなわち、組織構造を俯瞰し人々の責任およびコミュニケーションパスを表現する Flow Model、ある目的を達成するための作業の手順を表現する Sequence Model、業務で利用されているモノの構造における目的あるいは戦略を表現する Artifact Model、業務に対する影響力を表現する Cultural Model、そして業務が行われる物理的環境を表現する Physical Model である。

Work Model は、現場で行われている業務を、５つの観点において漏れなく忠実に表現することができる。また、その記法は単純かつ直感的であり、学習のための時間がほとんど必要ない。

筆者はこのアプローチに基づき、Flow Model、Sequence Model、Physical Model を業務観察の中で随時作成した。その際には、業務知識を収集するたびにモデルを更新し、モデルを最新の状態に保った。

最終的には外来受付業務を表現するモデルとしては、以下のようなものが完成した。

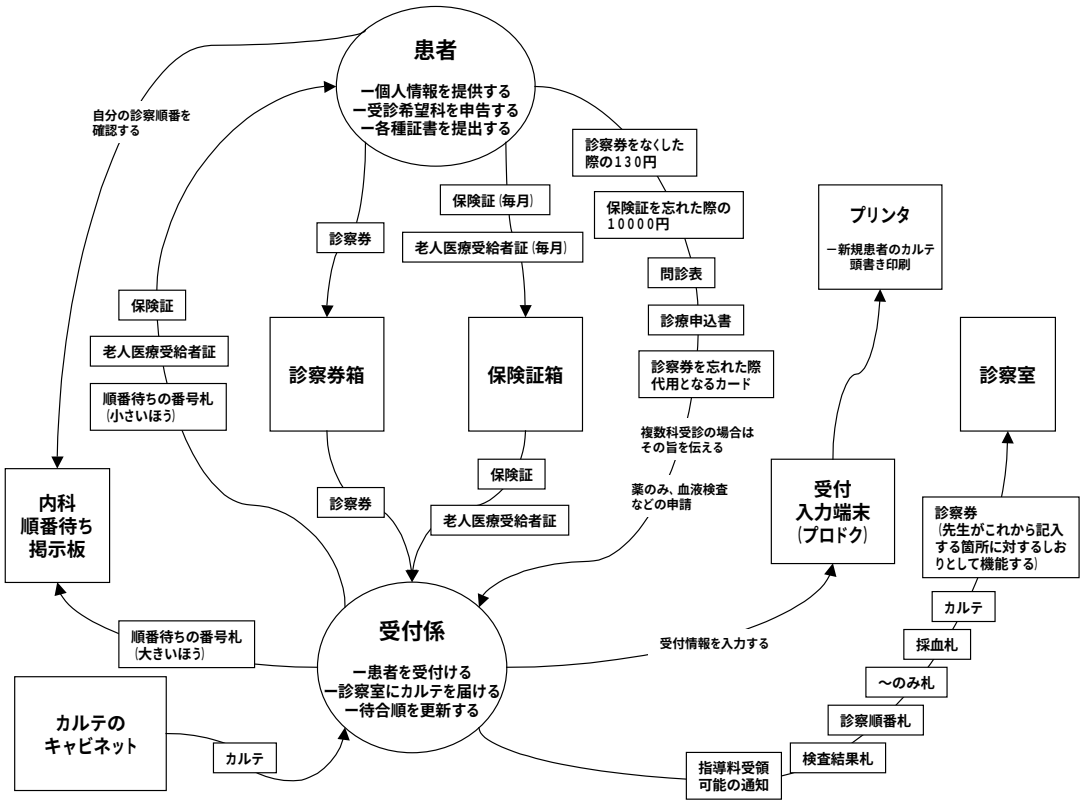


図 8　外来受付業務の Flow Model

このモデルは Work Model における Flow Model により外来受付業務を表現したものである。Flow Model は業務における組織構造を俯瞰し、それを構成する人々の責任およびコミュニケーションパスを表現するものである。

図上の円は業務に関わる人あるいは人のグループを表す。この円の中には、それが表現する人またはグループの責務が記述される。大きな矩形はその業務を遂行するために出入りする必要のある場所あるいはコミュニケーションを仲介する場所を表す。その中にはその場所の責務が記述される。円と大きな矩形を結ぶ矢印は、コミュニケーションパスである。その上に配置された小さな矩形は、その矢印が示すコミュニケーションパスの中でやり取りされる文書あるいは伝票といった有形のモノを表す。同じく、矢印の上に配置されるテキストは、そのコミュニケーションパスの上でやりとりされた会話あるいは行動といった無形のモノを表す。

例えば上の図中に円で表現された患者と受付係の間にはコミュニケーションパスが５つ存在する。そのうちのひとつ、左から３番目のコミュニケーションパスでは小さな矩形で表現された診察券がやりとりされているが、その際には大きな矩形で表された診察券箱がコミュニケーションの仲介役となっている。

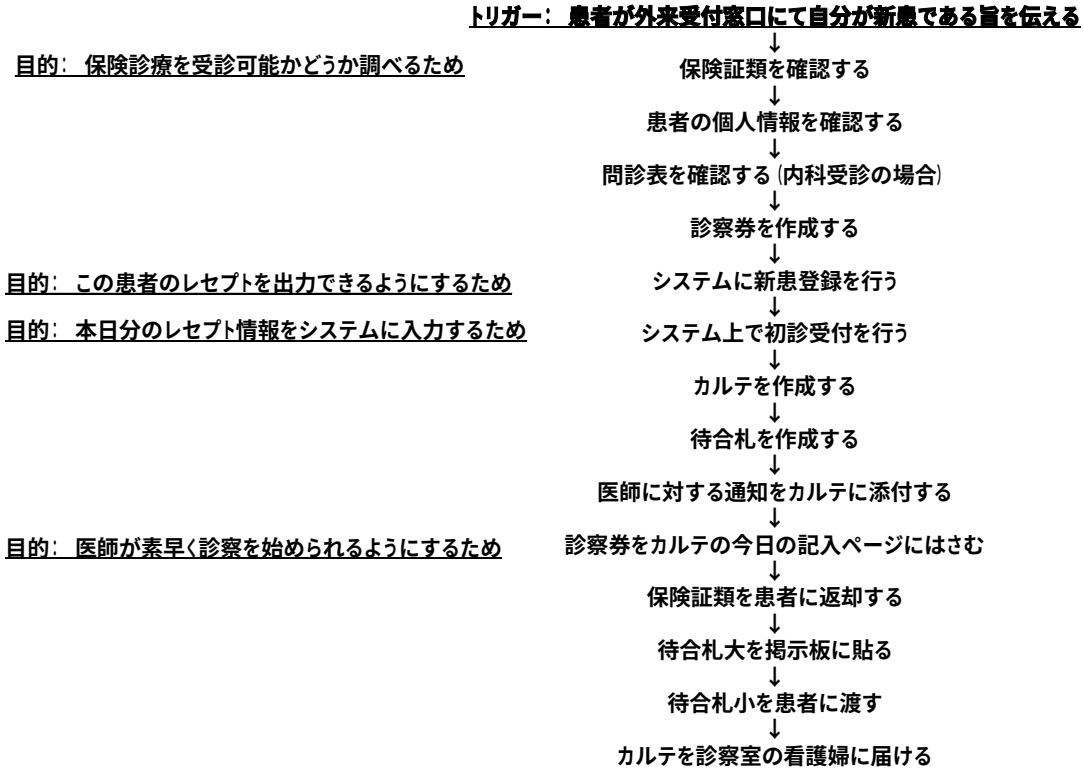


図 9　新規患者受付業務の Sequence Model

すなわち、患者が診察券箱に診察券を入れることで、受付係に診察券が受け渡されるということである。

このモデルは Work Model における Sequence Model により外来受付業務における新規患者受付の様子を表現したものである。Sequence Model はある目的を達成するための作業の手順およびそのきっかけを表現するものである。

Sequence Model において、先頭に「トリガー：」という表記のある文はその Sequence Model が表す一連の作業が発生するきっかけを表す。トリガーを起点として、作業のステップを示す文が矢印で接続されて作業の流れを表現する。ステップの左には必要に応じて目的を記述する。「目的：」という表記が先頭にある文がそれにあたる。

例えば上の図は、新規患者受付の様子を表現した Sequence Model である。新規患者受付に関わる一連の作業は患者が外来受付窓口にて自分が新患である旨を伝えることがきっかけとなっている。その後事務員は、患者の情報を確認し、システムに対して新患登録を行い、新しいカルテを作成するといった作業を行っていくことになる。

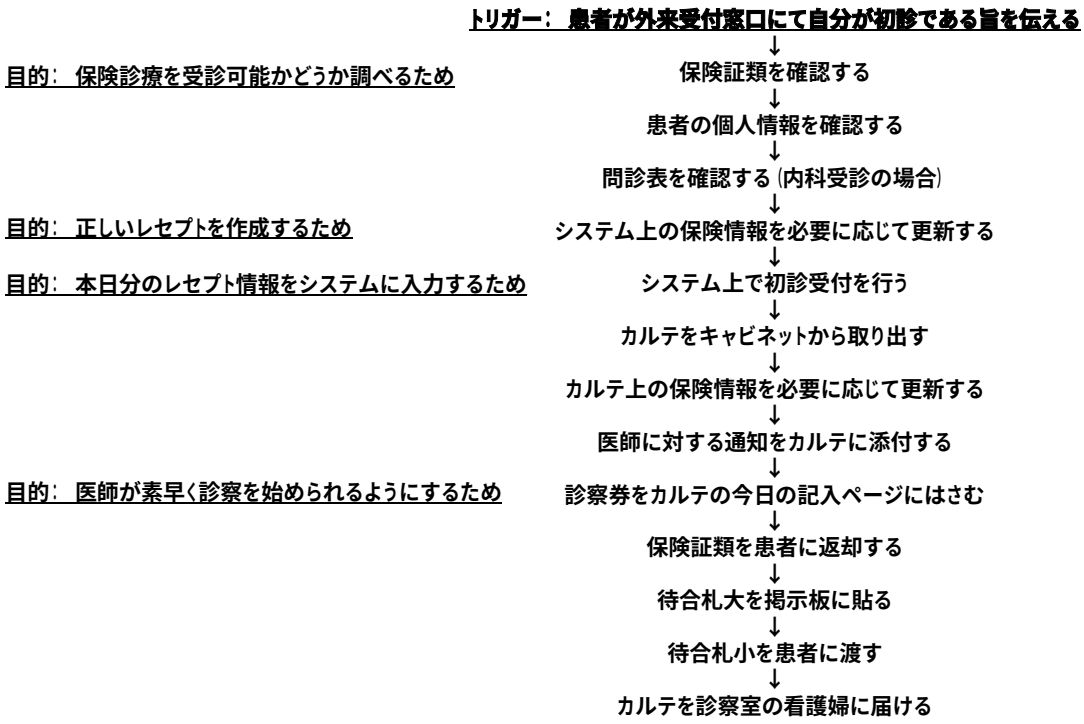


図 10　初診受付業務の Sequence Model

同様に、外来受付業務における初診患者受付および再診患者受付を Sequence Model で表すと図 10、図 11 のようになる。

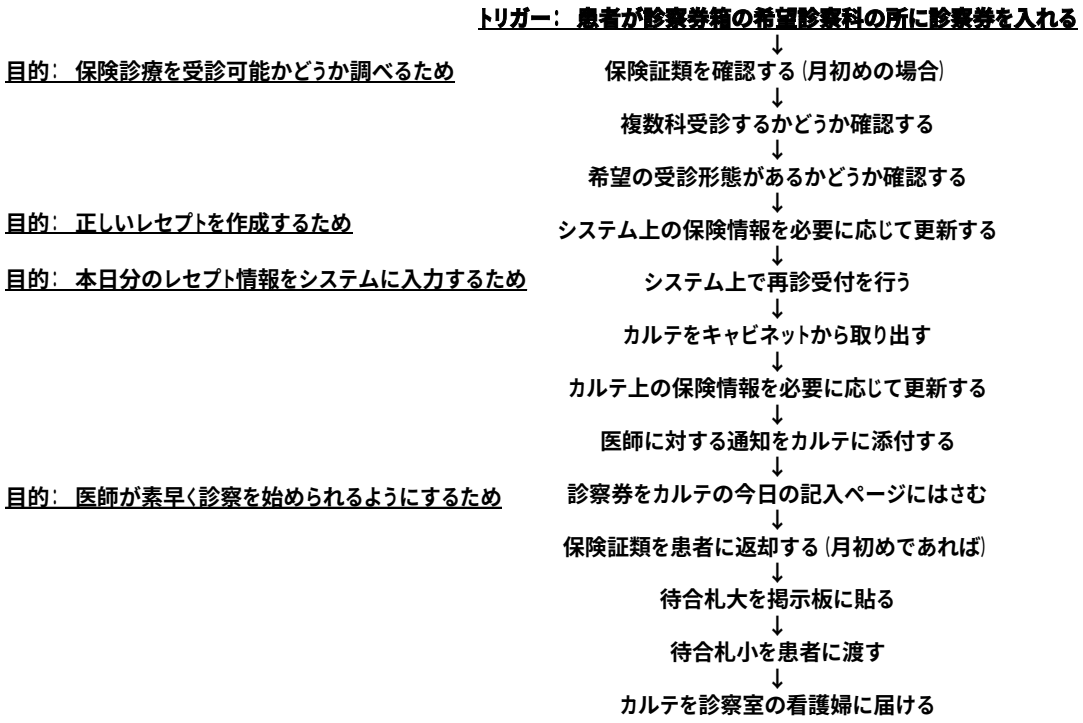


図 11　再診患者受付業務の Sequence Model

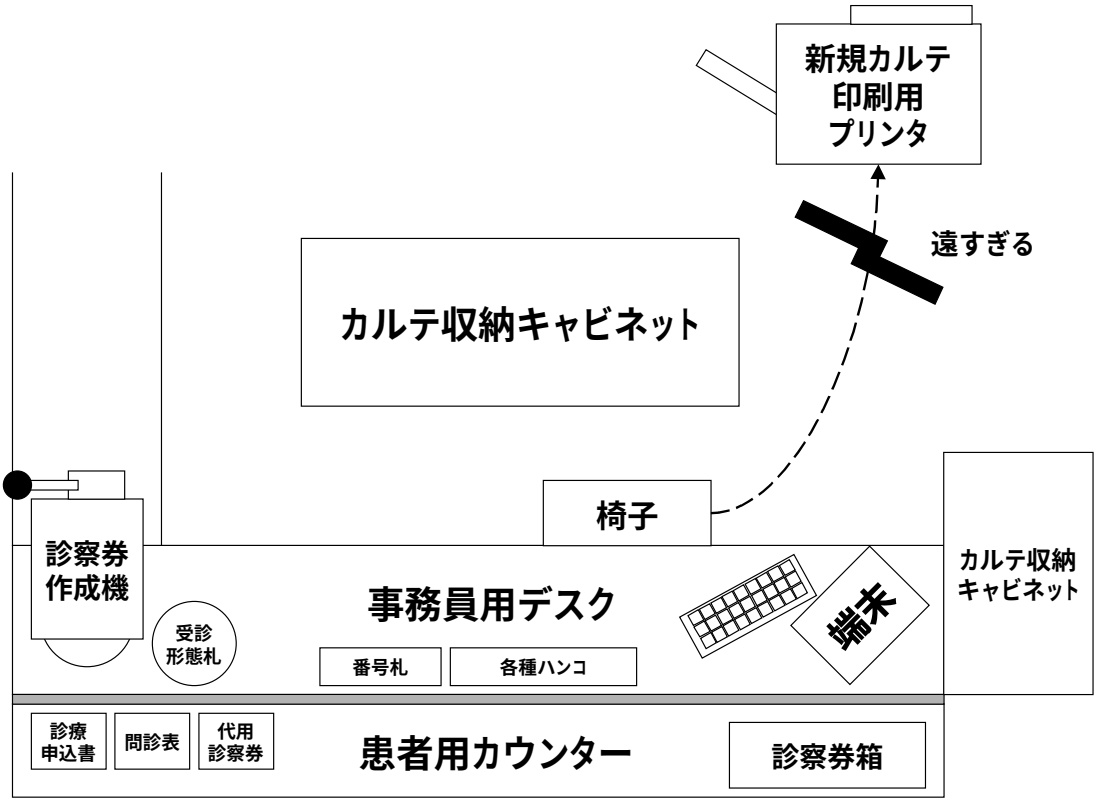


図 12　受付窓口の Physical Model

このモデルは Work Model における Physical Model により外来患者受付窓口の様子を表したものである。Physical Model は、業務が行われるスペースにおける構造、レイアウト、モノ、といった要素を表現する。

Physical Model では、特にその記法を定めていない。強いて言えば、矢印によって人あるいはモノの移動、コミュニケーションを表現する。また、そこに雷マークを付記することにより、その矢印の示す移動またはコミュニケーションに問題があることを示す。上の図では、受付係が座って業務に当たる椅子から新規カルテが印刷されるプリンタまでの距離が遠すぎることを示している。

3.2.7.医事課の反応

業務知識収集のために、最終的に約 1 ヶ月の間医事課に出入りすることとなった。その中で業務を観察し質問するということを繰り返しているうちに、事務員の方々と親睦を深めることができた。

時間がたつにつれて、事務員の方々ならびに医事課長は、筆者が行う質問の

中で筆者の知識の収集度合いを確かめ、業務が分かってきたと認めてくださるようになった。

3.3.第 2 段階：要求の収集

3.3.1.方針

要求の収集にあたり、医事会計システムに直接的な影響力のある業務管理者に対して、システムに何を求めるかについての意見を求めた。その際には、これまで得てきた業務知識をもとに、議論のポイントを用意した。そして、そのポイントについて議論し、システムにどのような効果を期待するかについての考え方を収集した。以下はその議論のポイントの例である。

待たせすぎる（患者の会計待ち時間の平均は１０～２０分)、ミスが多いなどといった患者の不満が医事課に対して向けられている
仕事の質の割に人（事務員）が多すぎるのではないか
診察時間は患者一人あたり３分程度であるにもかかわらず、一時間に６～８人程度しか処理できていないが、効率が悪いのではないか
業務におけるミスとして、受付ミス、処方ミス、請求漏れが目立つ
システムと紙を併用するのは効率が悪いし、ミスの要因となるのでは
レガシーシステムのマウス主導のインターフェースは扱いにくくないか
会計を全て自動化することは現実的か（実際そのような医療機関があることについて）

3.3.2.院長との議論

院長と議論した結果、以下のような結論を得ることができた。

業務の中で、思い立ったときに出来る限り素早く簡単にシステムにアクセスできることが重要である。システムの敷居を低くし、認証さえすれば誰もが手軽にシステムに対して情報を入力したり、システムから情報を引き出したりできるような状況が欲しい。

そういう意味で、システムは最も本質的な機能のみ備えた単純なものが良い。極端な話、電子カルテシステムについては、患者に対して誰がいつ何を行ったかについての記録が残っていればよい。オーダリングシステムについては誰がいつ誰に対して何を指示し、誰がいつそれを実行し、誰がいつそれを承認したかが記録されていればよい。医事会計システムに関しても、何の病気に対していつ何を使って何を行ったかが記録されていれば請求は可能である。無駄に巨大なシステムは結局使い物にならない。

入力方式に関しても同様のことが言える。医師の負担を軽減しようと様々な入力方式が提案されてはいるが、結局のところ、最もプリミティブなキーボード入力が一番である。プルダウンメニューやコンボボックスといった GUI は、一見便利楽に操作できそうではあるが、マウスを動かして小さな入力フィールドを特定し、その入力フィールドをアクティブな状態にした上でキーボードを使って入力するということを行わなければならないため、明らかに効率が悪い。音声認識では風邪を引いただけで声をうまく拾ってくれなくなるし、ペンタブレットによるフリーハンド入力は入力した後の情報を検索できないため非常に不便である。

診察中にキーボードをたたくことが患者に不快感を与えるといったような意見があるようだが、気にする必要はない。キーボードをたたくことによって、患者も幸せになる。つまり、医師が素早く確実にその患者の情報にアクセスできるようになるので、診察の質が向上する。紙カルテの場合、他の医師が書いた記録が、字が汚いなどの理由により読めないことが多々ある。また、物理メディアの制約により、過去の情報へのアクセスが容易でない。すなわち、近い過去であればすぐに対応できるが、遠い過去の情報についてはそうはいかない。診察情報が全て電子媒体で保存されるようになれば、記録が読めないという問題は発生しないし、どんなに遠い過去の情報でも記録されてさえいればすぐにアクセスできるようになる。

また、キーボード入力に抵抗を感じる医師に気を使う必要はない。メールは普通に書くのだから、出来ないはずはない。キーボード入力は今後医師にとって必須のスキルとなる。キーボード入力が出来ない医師は必要ない。

近年の医療情報システムの失敗は無駄に巨大なものを作ろうとしていることである。巨大なシステムを作ったはいいが、結局カメラ付き携帯電話の方が手軽で便利に使えたといったような例が後を絶たない。医療情報システムにおいて、本質的にやりたいことは非常に単純である。

誰でも簡単にシステムにアクセスし、情報を出し入れできるような環境が整えば、病院スタッフの仕事は減るはずである。例えば医事課にしてみれば、会計情報を発生源入力させることにより、入院レセプト作成の手間はほとんど省

ける。看護課にしてみても、情報を流すための転帰の必要性はなくなる。すると人のやる必要のない作業に無駄に時間を費やすのではなく、もっと本質的な仕事に時間を費やすことができるようになる。すなわち、医事課は、その分の時間を収益性の高い患者を連れてくることに費やすことができるようになり、看護課は患者と接することに費やすことができるようになる。

この結論から分かることは、院長のシステムに対する理想像は、最も単純で本質的な機能を提供することであり、誰もが簡単に利用できるアクセシビリティを備えているということである。それにより、結果として業務は支援され、病院スタッフはもっと本質的な仕事に時間を費やすことができるようになる。

3.3.3.理事との議論

理事と議論した結果、以下のような結論を得ることができた。

現状において最も問題なのは、各部門におけるセクショナリズムである。病院では多数の高度な専門家集団が同居している。それがゆえに、自分の部門のことに他の部門が口を出すことを許さない風土がある。従って、各部門にはその部門の人しかアクセスできないか、アクセスはできるが解読できない情報が存在してしまう。

医事課も例外ではない。例えば、保険者に対して請求した診療報酬の額と、実際に振り込まれた額が異なることはよくある。この理由は、レセプト記載ミスによる返戻のためであったり、伝票が届いていないことが原因での記載漏れだったりする。

経営側としては、請求額と振り込まれた額の差額と原因を把握し、経営に役立てたい。しかし現状においてこれは難しい。その理由として、その記録は医事課に残っているが、それが医事課しか解読できない形で残っているからである。

システムには、このようなセクショナリズムの壁を取り払うことを期待する。すなわち、部門を越えた業務環境をシステム上に構築することで、コミュニケーションを円滑にしたい。厳密な原価計算やタイムマネージメントにより利益を追求することは、業務におけるコミュニケーションの問題が解決してから取り組むべきことである。逆に、コミュニケーションが円滑になれば、金と時間は意識の問題でどうにでも解決できる。

最近病院内のメーリングリストが頻繁に使用されるようになり、部門を越え

た意見交換が活発になってきた。同じようなことを、業務レベルで行いたい。すなわち、日常業務において、各課がそれぞれの持つ情報を開示し、積極的なコミュニケーションのもとに業務を遂行するような状況を実現したい。システムはそれをサポートするものであって欲しい。

このようなシステムを機能させるには、そのシステムは現場が納得するものでなくてはいけない。現場が最も快適なシステムを作成し、その上で部門間コミュニケーションを促進するサポートを行うのが良い。現場がシステムを使わなければ意味がない。レガシーシステムを利用して上述したようなことを行おう思えばできないこともないが、ユーザビリティがひどいため、結局使われていない。そのような状況を避けるには、現場の意見を良く聞き、彼らが最も使いやすいシステムを提供する必要がある。

この結論から分かることとして、理事がシステムに期待することは病院職員のコミュニケーションを円滑にすることである。すなわち、各部門に独自の情報を抱え込ませず、病院に対して開放させるような仕組みを整えたい。そしてそのようなシステムは現場が最も使いやすいものである必要があると考えている。

3.3.4.医事課長との議論

医事課長と議論した結果、以下のような結論を得ることができた。

システムにレセプトを自動的に作成させることは現実的でない。なぜならレセプト作成の規則には無数の例外が存在するし、しかもその規則は頻繁に更新される。大改正は２年おきに行われ、小さな改正はそれとは無関係に頻繁にしかも突然行われる。規則の中でも診療行為、医薬品、調剤行為に関する標準名称、発生点数などを定めたマスターは、医学が進歩すれば変化せざるを得ないものなので、規則が今後一定に定まることも期待できない。このような例外および変化に柔軟に対応するには、医事課事務員が頭を使うしかない。

従ってシステムには、医事課事務員が効率よくレセプトを作成できる機能が欲しい。すなわち、最近のマウスを多用するインターフェースではなく、キーボードで操作が完結できるようなものが欲しい。マウスで目的の箇所にヒットさせるのは思いの他難しく、業務効率を落とす原因になっている。

また画面の切り替えもできる限り押さえるようにして欲しい。画面が切り替わると、頭の中まで切り替わってしまうからだ。

さらに、レセプトの用紙を画面上に再現し、その上で編集ができるような機能が実現できればうれしい。現状のレセプトの確認作業は非常に面倒である。一度紙で出力して確認した後、修正情報を入力しなくてはらないからだ。あまりに面倒なので、用紙の上で修正を行ってしまうこともある。レセプトを画面上で確認ができるようになれば、この出力は最後の一回だけで済むようになるため、非常に楽になる。

同様に、レセプト以外のシステムにおいても、マウスを多用するインターフェースは極力避けて欲しい。業務処理速度を向上するには、キーボード上の操作で完結するシステムであることが望ましい。それも、右手の使用だけで完結することができればさらに良い。紙カルテからシステムへ転記を行う際など、左手でカルテを持ちながらキーボードをタイプするためである。

この結論から分かることとして、医事課長がシステムに求めることとは、業務を最大限効率化するためのユーザビリティである。医事課長はシステムによる業務の自動化にはむしろ否定的で、業務モデルはそのままに、その効率を向上させることをシステムに期待している。

3.3.5.考え方のすり合わせ

３人の業務管理者から収集した要求には、いくらか食い違いが存在する。この様子を示す際に、Contextual Design の Cultural Model が便利に使えた。Cultural Model とは仕事が行われている環境の文化を、影響者と影響という形で表現するものである。文化は環境に対する期待、要求、価値観、姿勢などといった要素を定義する。

図 13 は、小原病院における医療業務支援システム開発という環境において、その影響者すなわち３人の業務管理者と開発者がどのように影響しあっているかについて示した図である。

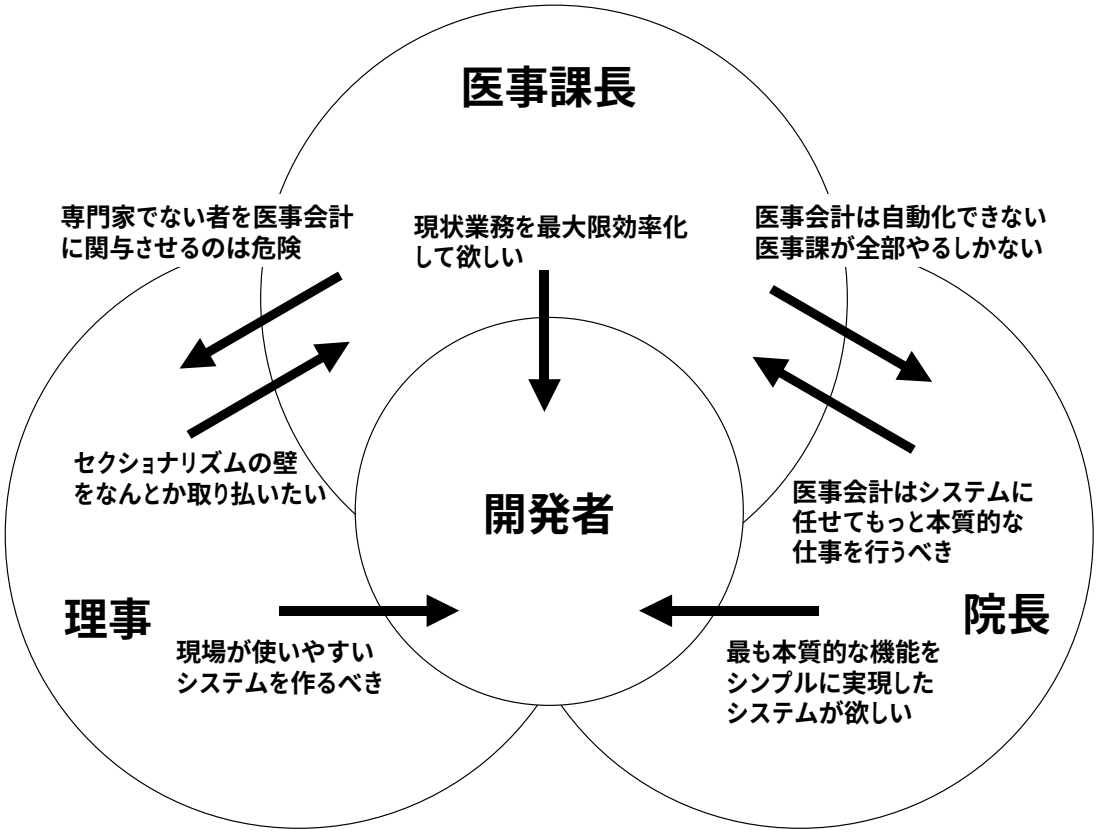


図 13　３者の要求を示す Cultural Model

医事課長と院長および医事課長と理事の意見の間には、それぞれ食い違いが存在する。医事会計業務のシステムによる自動化は可能であるという院長の立場と、不可能であるとする医事課長の立場は食い違っており、セクショナリズムの壁を取り払いたいと願う理事の立場と、専門家でない者を医事会計に関与させることに抵抗を持つ医事課長の立場は食い違っている。このような問題に対し、筆者は以下のように対応した。

院長と医事課長の意見の食い違いについて、どちらも極端であることが問題である。確かに医事会計業務には無数の例外が存在し、国や地方自治体の頻繁な制度改定の影響を多大に受けるため、システムによる完全な自動化は難しい。しかしそのような例外や外的要因の中で、小原病院に直接関係があるものはそこまで多くないのも事実である。実際、診療報酬明細書を作成する際、小原病院が過去に算定したことのある明細項目のバリエーションはわずか数百である。診療報酬明細書の中で使用可能な明細項目を収めたデータベースには数万件の明細項目が登録されていることを考えると、制度改定の影響を頻繁に受ける可能性はさほど高くないし、仮に影響を受けたとしてもその修正は難しいことではない。また、入院患者に対する会計に関しては、小原病院における病棟は定

額制の療養病床であるため、多くの場合決まりきった請求パターンしか行わない。そのため自動化の可能性は十分に存在する。

このような状況において筆者は、とりあえずは現状において明らかに自動化できる業務のみ自動化し、その後状況を見計らって医事課業務支援という立場から継続的に自動化を考えていくこととした。ここで重要なことは、自動化を考えるにあたってその時点の業務を支援するという立場を貫き、現場の事務員とともにそのやり方を考えることである。医事課長の懸念は、完全な自動化がなされてしまうと例外が発生した際に対応できなくなってしまうことである。自動化は行っていくが、それが必要に応じて徐々に行われるのであれば、そのような問題は発生しない。

理事と医事課長の意見の食い違いに関しては、セクショナリズムを取り払うことに関する認識の相違が存在する。医事課長の懸念は、他の部門の者が医事会計の業務に直接タッチすることが引き起こす混乱であり、理事の主張は情報開示である。

この問題はシステムレベルで解決できることである。単に発生した情報の見せ方を工夫すればよいことだからである。すなわち、各課で発生した情報に対し、他の部門の者が閲覧しても理解可能なインターフェースを用意すればよいことになる。

3.4.第 3 段階：開発単位の分割

3.4.1.方針

筆者は、開発単位を分割するに当たり、医事課業務に関するユースケース分析[17]を行った。

ユースケースとは、ユーザの目的に照らしたシステム利用のシナリオである。ユースケースはシステムを外部から見たときのシステムの振る舞いを記述するものであり、システムの要求を記述する上で利用される。ユースケースはユースケースモデルとユースケース文書によって記述される。ユースケースモデルはユースケースを俯瞰的に表すための図解であり、アクター、システム境界、ユースケースという要素から構成される。ユースケース文書は個々のユースケースに対して詳細な定義を与えるものであり、個々のユースケースについてユースケース名、アクター、目的、事前・事後条件、イベントフロー、シナリオ

といった内容について記述する。

OCD における開発単位分割の目的は、現場ユーザと協調してシステムを作成する過程を円滑にすることである。従って、分割された単位は適切な規模を持ち、他の単位との依存性が低く、直感的で業務管理者の要求から逸脱しないポリシーである必要があった。

ユースケースは、システムに対してある役割を持ったユーザが、ある目的のもとにシステムを利用する際のシナリオであり、このシナリオは、上記指針に合致した開発単位を抽出する際の強力な軸となる。すなわち、シナリオの規模が適切で、シナリオ同士の依存性が低く、シナリオが直感的で業務管理者の要求から逸脱しない場合、そのユースケースはそのまま分割単位となる。また、単体では分割単位として不十分であるが、別のユースケースと結合することにより分割単位として適切になるのであれば、結合したユースケースを開発単位として抽出すればよい。シナリオはユーザがシステムを利用する際の具体例であるため、そのような評価が行いやすい。

これを行うには、業務を一挙にサポートするシステムを想定してユースケースを記述すると良い。その際の指針は、一般的なユースケース分析の方法論に従う。すなわち、ユースケースは明確な目的と直感的な名前、そして適切な粒度を持ち、適切なアクターと関連付けられ、業務を網羅してなければならない。そうして記述されたユースケースをもとに、開発単位を導き出すのである。

筆者は上記アプローチのもと、医事課における全ての業務を一括で支援するシステムについてユースケースを作成した。そしてその個々のユースケースについて OCD の分割の指針に照らし合わせて考慮し、必要に応じて分割または統合を行うことによって開発単位を抽出した。

ユースケースは、これまで得た業務知識と、業務管理者から収集した要求をもとに作成した。以下は、この結果導き出されたユースケースモデルである。なお、ユースケースの詳細については、付録 2 のユースケース分析書を参照されたい。

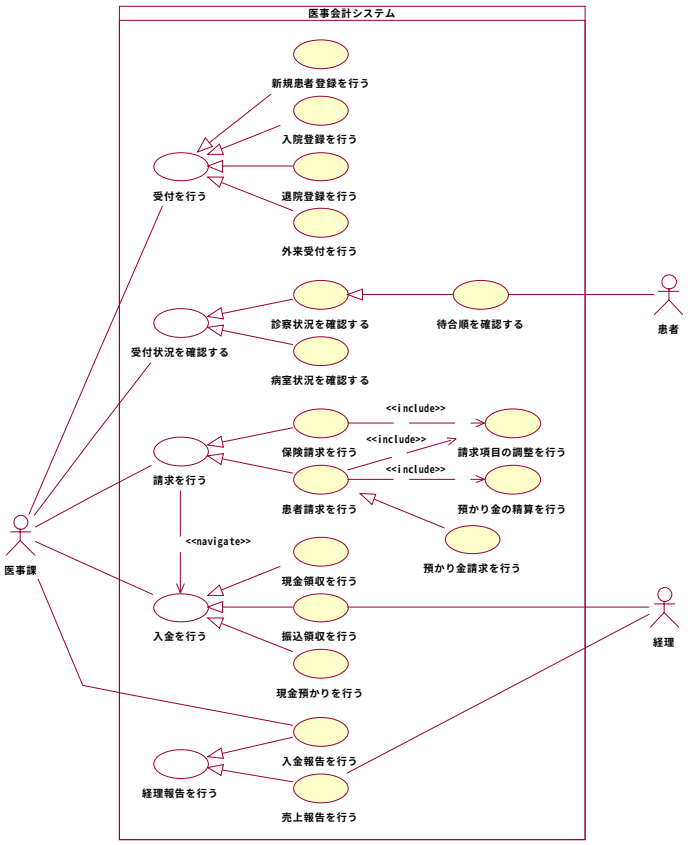


図 14　医事課業務支援システムのユースケースモデル

3.4.2.導き出された開発単位

図 15 は、図 14 のユースケースモデルから導き出された開発単位を示したユースケースモデルである。

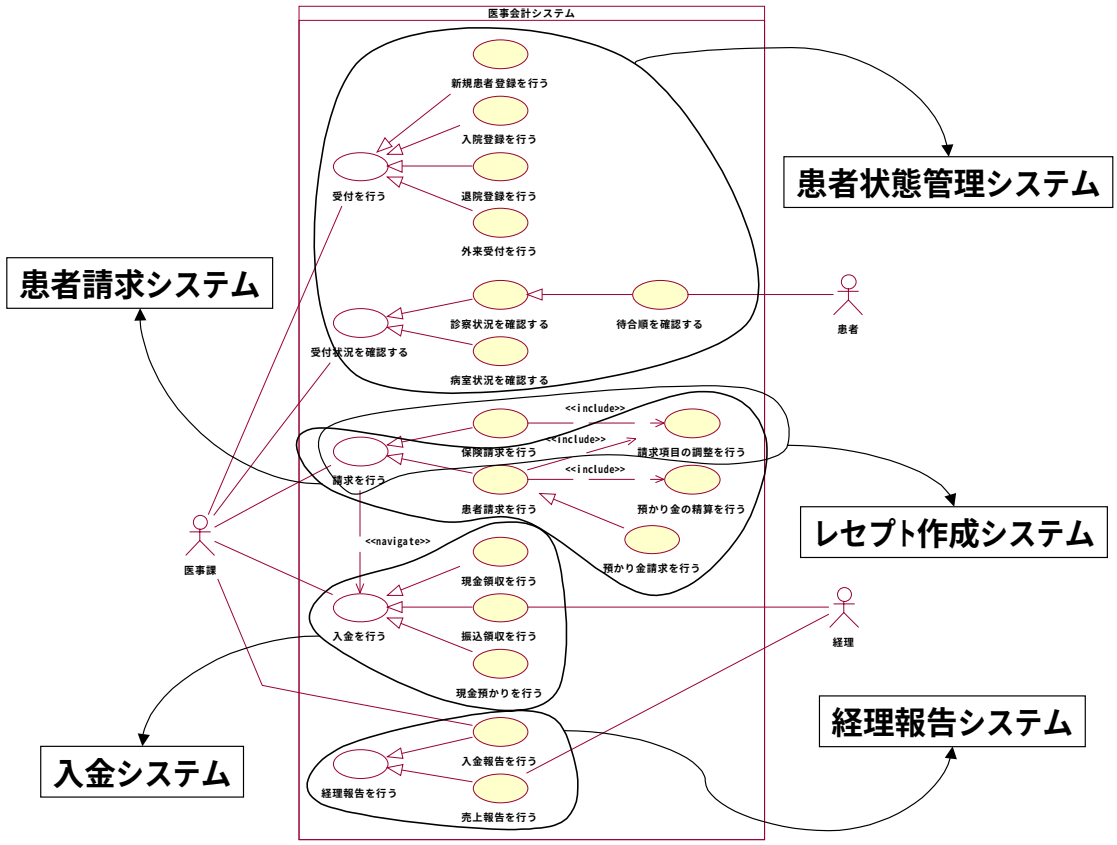


図 15　開発単位を示すユースケースモデル 1

外来患者状態管理システムは、入院および外来患者の院内における物理的状態を追跡する。入院患者に関しては、入院登録されてから退院登録されるまで、どの病室のどのベッドに入院しているか、および会計の状況を管理する。外来患者に関しては、外来受付されてから会計が終了するまでどの診察科を受診しているか、および会計の状況を管理する。

レセプト作成システムは、入院および外来患者のレセプトを作成する。その際必要に応じて請求項目の調整を行うことができる。医事課事務員はこのシステムを使って、毎月末に個々の入院および外来患者のレセプトを発行する。

患者請求システムは、入院および外来患者に対する請求金額を算出し、請求書および領収書を発行する。預かり金に関する処理も合わせて行う。

入金システムは、患者あるいは保険者から入金された額を管理する。経理も必要に応じて、このシステムを使って振り込み領収の処理を行う。

経理報告システムは、入金に関する情報を入金報告あるいは売上げ報告としてフォーマットし、出力する。医事課事務員はこのシステムを利用してその日の入金を経理に報告する。経理はこのシステムを利用して売上げ報告を作成す

る。

3.4.3.アーキテクチャ

アーキテクチャとしては、基本的に WEB アプリケーションのフレームワークを採用することとした。すなわち、クライアントサイドは WEB ブラウザ、サーバサイドにはアプリケーションサーバ（WEB サーバ＋WEB アプリケーションサーバ）とデータベースサーバという３階層モデルである。そして、WEB サーバには Apache、WEB アプリケーションサーバには Tomcat、データベースサーバには PostgreSQL を採用することとした。

分割したシステム同士の協調には、WEB サービスのフレームワークを利用することとした。すなわち、WEB サービスサーバにおいて個々のシステムが汎用的な機能を公開することで、それを他のシステムが利用できる形にした。この WEB サービスサーバには Axis を採用した。

3.4.4.業務管理者とのレビュー

分割方針が定まったところで、インタビューを行ったそれぞれの業務管理者に対して報告を行った。

それぞれの業務管理者から、分割のレビューに関しては問題ないという評価をいただいた。但し、医事課長からは、現状業務において外来会計時の正式な領収書発行が急務であるため、外来患者請求システムをまず開発することをリクエストされた。また理事からは、クライアントサイドの実装にあたり場合によっては表現力豊かな Visual Basic を利用する可能性があることを示唆したことについて、それはやめるようにとの指示を受けた。WEB ブラウザ上で完結することが、このシステムに対外的な説得力を持たせる上で重要であるとのことだった。

この結果を受けて、外来患者請求システムを最初の開発単位とし、クライアントサイドは WEB ブラウザに統一することとした。ただし、HTML の表現力は、複雑なレセプト作成機能を実装するにあたりどうしても不十分であるため、必用に応じて Applet を利用することとした。

またレビューの際に各開発単位について説明する中で、筆者は、患者状態管理システムと患者請求システムそしてレセプト作成システムの３つのシステムについて、開発単位としては少し規模が大きくなりすぎているという感覚を得

た。

その原因は、３つのシステムが、それぞれ入院と外来を区別していないことである。実際、入院と外来で本質的に行うことは同じである。すなわち、受付を行い、受付の状況を確認し、保険者および患者に対して請求を行う。３つのシステムはこのことを忠実に反映している。

しかし実際問題として、本質的には同じ業務でも、その中身は随分異なる。例えば患者請求に関して、入院患者に対する請求は毎月１５日とその月の末日と約１５日おきに行われ、一方外来患者に対する請求は患者が受診を受けた日ごとに行われる。さらに入院と外来で請求書のフォーマットは異なるし、請求書を作成する際に外来の場合は主にカルテを参照するが、入院の場合はほぼ伝票しか参照しない。つまり３つのシステムは、本質的に同じ振る舞いではあるが、全く別の中身を持つ２つの業務を同時にサポートしていることになる。

筆者は、これら３つのシステムについては、それぞれ入院と外来に分割することにした。本質的には同じ振る舞いをサポートするシステムでも、入院と外来において全く異なる中身を実装しなくてはならなくなるため、結局のところ規模は２倍になるし、全く異なる中身は相互に依存する可能性が低い。また、現状の分割方針のレビューにおいてそのポリシーに問題がないことが分かったため、それをさらに入院と外来という明確な区切りで分割することに問題はないはずである。

こうして、開発単位は最終的に図 16 のような形に落ち着いた。

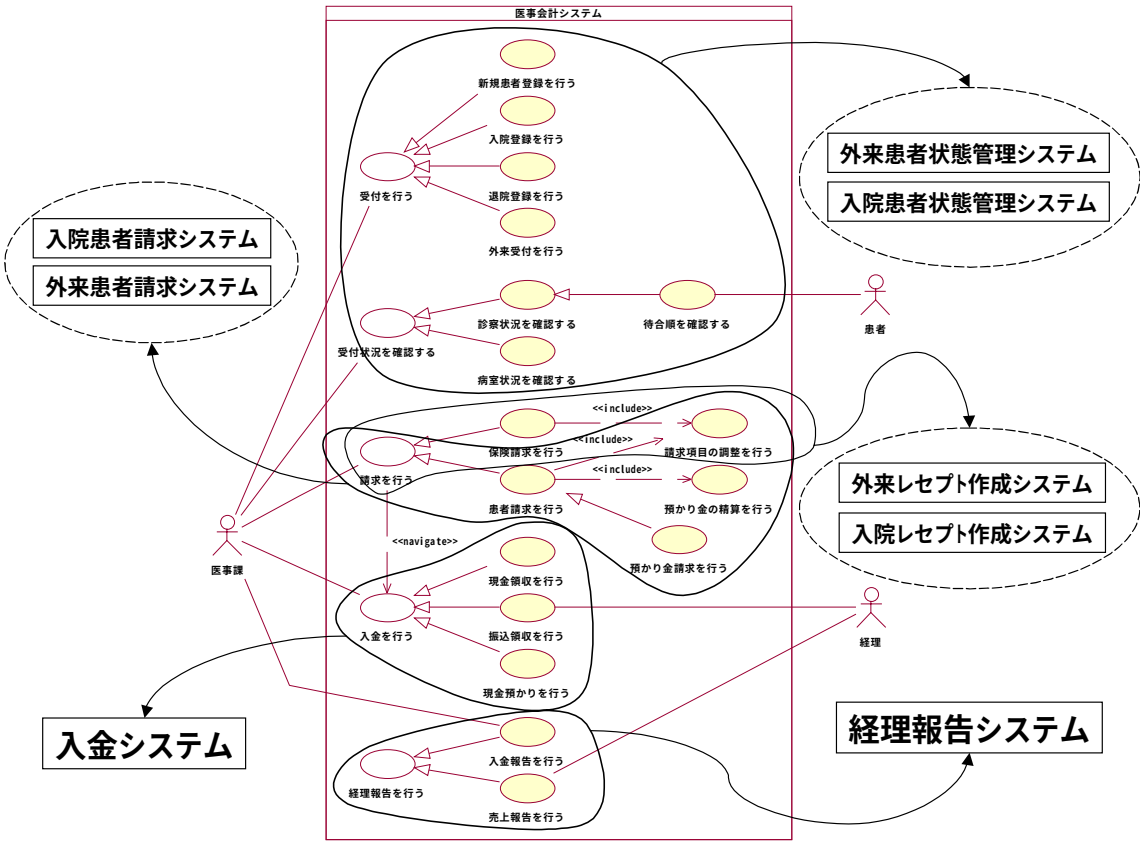


図 16　開発単位を示すユースケースモデル 2

3.5.第 4 段階：反復的なシステムの作成

3.5.1.方針

作成にあたっては、OCD のアプローチに従い、個々のシステムについて現場ユーザと協調した反復的な作成を行った。その際には、サンプルを作成しては医事課に赴き、医事課長のデスクに事務員全員を集めてサンプルについて議論を行うということを行った。またサンプル作成の過程においても、医事課に頻繁に赴き、サンプル作成のアプローチに関して質問したり、確認をとったりした。

以下では、外来患者請求システムについて上記アプローチを行った経過について述べる。

3.5.2.外来患者請求システムの定義

目的の外来患者についてその日発生した会計情報を収集し、その患者が支払うべき一部負担金の額を算出するとともに、その明細が記載された領収書を発行する。

3.5.3.初期の要求

本システムの開発が急務である理由は、現状の領収書に対する患者の苦情である。現状において患者に渡している領収書はレジが出力するレシートであり、その印字は薄い上に、明細がほとんど記載されていない。この問題に対しては兼ねてから患者の苦情が来ていたが、会計情報がレガシーシステムに蓄積されているため既製品を導入するわけにはいかず、対処されずに放置されていた。

従って、本システムに対する要求は、レガシーシステムから会計情報を収集し、きちんとした領収書を出力することである。そしてその際には、会計情報の収集先を容易に変更できるようにしておく必要がある。すなわち、今後診療業務をサポートするシステムが新しく導入されることを見越し、レガシーシステムから新しいシステムに容易に繋ぎかえることができるような実装を行う必要がある。その上で、現状の業務においてボトルネックとならないようなユーザビリティを備えるようにしなければならない。

3.5.4.最初のサンプル

図 17 および図 18 は、最初に現場ユーザに提示したサンプルシステムのスクリーンショットである。

図 17　最初のサンプルにおける診療明細編集画面

図 18　最初のサンプルにおける領収書画面

3.5.5.現場ユーザとの議論のポイント

以下は作成の過程において、現場ユーザと議論を行った際のポイントの例である。

（1）業務モデルの問題

現状業務においては、患者と金のやりとりをする際に、必ずしもレガシーシステムに対して会計情報入力を済ませておく必要はなかった。レジが出力するレシートは、レガシーシステムの持つ会計情報とは無関係に発行されていたからである。従って、実際の業務では、患者との金のやりとりが終了してから会計情報を入力するということが可能であり、実際にそのようにする場面が多々あった。事務員がその金額を暗記してしまっているくらい頻出する請求パターンというものがいくつかあり、その場合会計情報入力は後回しにし、直接レジをたたいてしまっていた。そうすることで問題となっている患者の待ち時間を

少しでも短縮することが出来るからである。
しかし、本システムを業務で利用する場合、このようなやりかたは出来なくなる。本システムにおける、レガシーデータから会計情報を収集しなくてはならないという制約上、患者との金のやり取りは、レガシーシステムに会計情報を入力し本システムを利用して領収書を発行してから行うというモデルに固定化されてしまう。これは患者に対してきちんとした領収書を渡せる半面、ただでさえ問題となっている会計時の患者の待ち時間をさらに延ばすことになってしまいかねない。
そのような問題をどう考えるか現場と議論したところ、以下のような結論が得られた。
きちんとした領収書を患者に渡すことは、情報開示という意味で重要である。患者からの請求金額についての問い合わせ、あるいはクレームに対処する際に重要な証拠となるためである。そういう意味で、患者の待ち時間の問題よりも、重要度は高い。従って、患者にきちんとした領収書を渡すために、患者の待ち時間が増えてしまうことはある程度仕方ない。
患者の待ち時間の問題に対して、特に急ぎの患者については領収書を後日渡すということで対処することにする。また、領収書専用のプリンタを購入し、それを配置する位置を工夫することで業務の効率化を図る。

（2）会計情報の修正に関する問題

現状業務において、レガシーシステムに対する会計情報の入力は基本的に事務員がカルテを見ながら行っているが、時としてレガシーシステムが自動的に挿入するものもある。そのようなものは、レガシーシステムの不備が原因で、情報に誤りがあることが多い。従って、レセプトを発行する際には、全てのレセプトに関してそのような誤りがないかどうかのチェックを行っている。
この問題は、領収書を発行する際にも影響する。すなわち、領収書はレガシーシステムのデータをもとに作成されるので、その情報に誤りが含まれている可能性が高い。従って、領収書を作成する際にはレガシーシステムから収集した会計情報に修正を加えなければならないことを考慮しなければならない。
これに関して、どのようなポリシーで修正を行うべきかについて現場ユーザと継続的に議論を行った結果、以下のような結論を得られた。
基本的には全ての会計情報をレガシーシステムから収集する。但し収集した情報は全て編集可能とし、修正後の情報をもとに一部負担金を再計算することを可能にする。
この際、修正した情報をレガシーシステムに保存しなおす必要はない。レガ

シーシステムの中身はきわめて複雑になっており、その修正には相当の手間がかかり多くの時間がとられることが予想される。領収書は緊急の課題であるためそれだけの時間をかけることはできない。

本システムの作成にあたり、反復は５回行った。その中で上記以外にも、領収書上の表記、領収書における「その他」項目の実装、一部負担金算出のロジックなどに関して様々な要求を得ることができ、システムを着実に成長させることができた。

3.5.6.現場ユーザ教育のポイント

レガシーシステムでは一貫して、入力した情報が入力と同時にシステムに保存されるというポリシーであった。その理由は、レガシーシステムの採用しているアーキテクチャが、ユーザの入力がそのままデータベースのレコードに挿入されるというポリシーであるからである。

しかし本システムのアーキテクチャである WEB アプリケーションのフレームワークにおいては、ユーザによって入力された情報は、明示的に WEB サーバに送信されなければデータベースに保存されない。すなわち、ユーザは情報を入力した後で、サーバにその情報を送信するための操作を実行しなくてはならない。

レガシーシステムのユーザビリティに慣れている事務員にとって、これは大きな変革である。筆者は事務員との議論の中で、会計情報を修正する画面の入力フィールドにおいて間違ってデータを消してしまった場合どうなるのかとの質問を受けた際、この事実に気がついた。

レガシーシステムのポリシーでは、入力フィールド上でデータを消すと、それがそのままデータベースに反映されるので、データベース上のデータも消してしまうことになる。しかし WEB アプリケーションではある情報を消したということがサーバに送信されるまで、データベースのデータは消えない。そして、そもそも本システムにおいて、扱う情報をどこかに保存するということはしない。ただレガシーシステムから情報を収集し、それを必要に応じて修正したものを領収書としてフォーマットして出力するだけである。

筆者は、このことはシステムについて非常に本質的な知識であると判断し、現場ユーザに対して説明を行った。

図 19　最終的なサンプルにおける診療明細編集画面

3.5.7.最終的なシステム

図 19 および図 20 はこの段階における最終的なサンプルのスクリーンショットである。

図 20　最終的なサンプルにおける領収書画面

3.5.8.業務管理者とのレビュー

医事課長および事務員と最終的なレビューを行った。院長および理事は、このシステムは暫定的なものであるため、現段階においては医事課長の許可が得られれば良いとのことであった。医事課長はシステム作成の段階においてよく議論に参加されており、その過程でシステムを良く理解していたので、特に改まったレビューを行う必要はなかった。

レビューの代わりに、実際の業務における利用をシミュレーションする形で、数人の患者の領収書を実際に作成するテストを行った。また、本番稼動に備えてプリンタの配置および設定といった作業を合わせて行った。

比較的単純なシステムである上に、現場と十分に議論を行って作成したシステムであるため、特に問題なくテストを完了することができた。

3.6.第 5 段階：建て増し的なシステムの導入

3.6.1.方針

導入は、個々の開発単位の作成が完了し次第、随時行った。そしてその都度既存のシステムと関連付けるということを行った。

外来患者請求システムについては、導入に際してレガシーシステムと関連付けることとなった。すなわち、稼動中のレガシーシステムから会計情報を収集し、領収書を出力するよう設定した。そして実際の業務の中で稼動させ、システムの最適化を行った。以下ではそのときの様子について述べる。

3.6.2.追加要求

現場に導入してまもなく、領収書における「その他」という項目の実装が問題となった。領収書における「その他」項目は、文書、健康診断、各種予防接種といった、保険請求が行えない項目についての請求額を記載するために用意されている。ここまでの実装では素直に「その他」という項目を置いているだけであったが、その内容を領収書に記載する必要性が生じてきた。

その理由は、企業や団体といった組織から健康診断や予防接種を委託するケースが増えてきたためである。この際には、その組織の患者個人個人に対して、健康診断あるいは予防接種を行った旨が記載されている領収書を発行する必要があるのである。

このようなことに対応するためには、「その他」項目は必要に応じてその名前を編集可能であり、かつ伸縮自在である必要がある。すなわち、自由に項目を作成可能である必要がある。このような報告を現場から受け、システムの実装を変更することとなった。

この追加要求を、現場ユーザは以下のように筆者に対して申し立ててきた。「その他」項目と同じ位置づけで、「健康診断」という項目を追加してほしい。ある企業から健康診断の委託を受け、その企業の患者には健康診断を受けた旨が記載された領収書を発行する必要があるためである。ただ、今後同じようなことが発生する可能性が高い。例えば、インフルエンザの時期になると健康診断と同じように企業から予防接種の委託を受けることになるので似たような要求が発生する。

従って、「その他」項目を自由に編集できる機能が欲しい。必要に応じて項目を追加したり、削除したりできるようにして欲しい。そのような機能があれば、上記のようなことをその都度要求しなくて済む。

システム作成の過程における議論の中で、現場ユーザがシステムにひとつの機能を実装する際の開発者の考え方を身につけたため、このように非常に明快な要求を申し立てることができた。

3.6.3.現場ユーザの理解確認

稼動して直後、筆者は現場ユーザから以下のような要求を受けることとなった。

会計情報を修正する画面において、誤って情報を編集してしまったり、誤って全消去してしまったりしたときのために、レガシーシステムからデータをリロードする機能をつけて欲しい。

これは現場ユーザが本システムのアーキテクチャを良く理解しているために可能となった要求である。すなわち、本システムはレガシーシステムから会計情報を収集し、領収書というフォーマットに当てはめるだけであり、その際に行う会計情報の修正はレガシーシステムに反映されない、ということを現場ユーザが理解しているのである。

また、領収書をプリントアウトする際に、想定外のプリンタから出力されるという問題が発生した。これは開発者がプリンタの設定を行うのを忘れたために発生した問題であるが、これに対して現場ユーザは自力で対処していた。すなわち、本システムは独自にプリントアウトの機能を実装しているわけではなく、プリントアウトの際には WEB ブラウザの機能を利用しているということを現場ユーザは理解していた。そのため、現場ユーザはこの問題が発生した際、開発者を呼ぶことなく WEB ブラウザのデフォルトプリンタの設定を修正し、目的のプリンタに領収書をプリントアウトさせることに成功していた。

図 21　現在稼動中のシステムにおける診療明細編集画面

3.6.4.外来患者請求システムの現状

本システムは昨年６月中ころに導入されたが、上記のような要求に対処し終わった７月以降は目立った不備も発見されず、安定して稼動している。図 21 および図 22 は、現在稼動中のシステムのスクリーンショットである。

図 22　現在稼動中のシステムにおける領収書画面

4.OCD の評価

4.1.評価方針

ここでは、小原病院において OCD を実践した例を評価軸として設定し、OCD の有効性を評価することにした。その際、作成されたシステムが OCD の目標である以下の基準をどれだけ達成したかについて考察した。

① 全ての立場のステークホルダーが満足するシステムを得る
② 現場ユーザが使いこなせるシステムを得る

4.2.ステークホルダーの満足度に関する評価

OCD ではあらゆる立場のステークホルダーの要求をシステムに反映することを目指し、その要求を積極的に収集する。その中で、要求は常に変化し、また変化することによって成長していくものであるという立場をとる。そのため、業務管理者に対しては初期の要求に関するインタビューあるいは開発の要所におけるアウトプットのレビューという形で、現場ユーザに対しては反復的なサンプルシステムの評価という形でその要求を継続的に収集する。

小原病院での実践では、ステークホルダーは OCD によるそのような開発アプローチに対して満足していた。出来合いのシステムの利用を押し付けるのではなく、開発者とステークホルダーが一丸となって、その業務に最適なシステムを作り上げようという姿勢がステークホルダーによって評価された。

特に、反復的なシステム作成の段階において、現場ユーザは非常に積極的に参加してくれた。そもそも現場ユーザに、自分の使うシステムは自分でデザインしたいという欲求が多分にあった。それに加え、自分の要求がサンプルシステムに忠実に再現されて短期間のうちに出来上がってくるので、自分の欲しいシステムが本当に得られそうだという実感につながり、開発に参加する上での強力なモチベーションとなったようである。同時に現場ユーザは、自分が責任を持ってシステムを評価しなければ良いシステムを得ることはできないという意識を持つことになり、それは議論のクオリティを向上させることとなった。

この結果、最終的なシステムは必然的にステークホルダーの要求をよく網羅するものとなった。反復的な要求収集の中で、不適当な要求は徐々に淘汰され、本当に必要な要求のみがシステムに反映される形となった。

このように OCD はその実践にこそ意味があるといえる。上述したように、ステークホルダーの満足は、OCD のアウトプットではなく、OCD の開発アプローチによるものであった。OCD のアプローチに従えば、そのアウトプットは必然的にステークホルダーの要求を忠実に反映したものになる。

4.3.現場ユーザのシステムに対する理解度に関する評価

OCD では、ユーザが導入されたシステムを快適に利用できるよう、システムおよびその主要な技術について教育する。これは現場ユーザとの反復的なシステム作成段階にて行われる。ユーザとのシステムの要求についての議論の中で、

必要に応じて教育を行い、その後それを踏まえた議論を行っていくことで、現場ユーザはシステムを本質的に理解するようになる。

小原病院での実践では、このアプローチによって現場ユーザは自分の出来る範囲でシステムの不備に対応できるようになり、かつ現実的な追加要求を開発者に対して申し立てることができるようになった。この結果、導入システムについて呼ばれる際は、ある程度深刻な問題が発生したときか、十分に検討された追加要求が用意された際に限られるようになった。

このことは、現場ユーザにとっても開発者に対してもメリットをもたらす。現場ユーザにとってみれば、ちょっとしたシステムの不備のために、開発者がそれに対応する間業務を停滞させることがなくなる。また、必要に応じて現実的な追加要求を自分で考え、すぐさま開発者をその要求の実現に動かすことができるようになる。すなわち、業務の中でシステムに追加したい要求を発見した際、現場ユーザが開発者に対して曖昧な要求を投げかけてしまったがために、それについて開発者が分析し何度も現場ユーザに確認をとるといった無駄な手続きを経る必要がなくなる。開発者にとってみれば、些細なシステムの不備のために現場ユーザから呼び出されることが少なくなるし、追加要求が発生した際その打ち合わせに無駄に時間をとられることなくすぐに作業を開始できるようになる。

小原病院には現在このような状況が存在し、現場ユーザ、開発者ともにそのメリットを享受している。

4.4.今後の課題

OCD には現状において次の２つの課題が存在する。

一つ目は、開発にかかる全体的な期間の見積もりが難しいことである。その理由として、OCD では開発の各段階の境界が明確に定められていないためである。すなわち、第１段階および第２段階の終了のタイミングは、開発者が必要な情報を十分に収集できた実感を得たときである。そして第３段階と第４段階の終了のタイミングはユーザがその段階における開発者の提案に満足したときであり、第５段階に関しては明確な終了は存在しない。なぜなら、OCD はインハウスにおいて成長させるシステムの開発を前提としているからである。従って、現在のソフトウェア開発現場の中で OCD を実践可能なものにするには、開発期間、コストなどに関する更なる吟味が必要である。

二つ目は、反復的なシステム作成の段階において、議論のたたき台となる動

作可能なサンプルシステムを用意することが難しいことである。たたき台となるサンプルが動作可能であればより広範囲の要求を収集することができる。すなわち、表面的な画面レイアウトや機能要求のみならず、システムの挙動やパフォーマンスに関する要求なども収集することができるため、より効果的な議論が可能となる。その一方で、反復的な開発において、そのスパンを短く保つことは、前のバージョンのシステムに対する現場ユーザの印象を劣化させないために非常に重要である。そして動作可能なシステムを素早く作成することは、よほど小さなシステムでない限り簡単なことではない。現状において、OCD を実践するにあたりこのような葛藤を回避することができない。

この問題を解決するために、筆者は動作可能なシステムを素早く作成することを支援するツール Toriaezer を開発した。これについては次章で述べる。

5.まとめ

このように、OCD はステークホルダーおよび開発者をともに幸せにする開発アプローチである。ステークホルダーは自分の要求に忠実なシステムが得られ、特に現場ユーザはそのシステムを快適に運用するための知識を得られる。開発者は、ステークホルダーにその要求に忠実なシステムを提供し、特に現場ユーザにはシステムに関する本質的な知識を授けることで、導入後のクレームあるいは追加要求などに対応する際の負荷を最小限にすることが可能である。

第3章 開発支援ツール Toriaezer

1.はじめに

ユーザの要求に忠実に応える良いシステムを開発するには、ユーザからの反復的な要求の収集が欠かせない。要求ははじめから固定的にユーザの中にあるものではなく、度重なる開発者との議論の中で徐々に形作られていくものだからである。この議論を繰り返せば繰り返すほど、要求は確かのものになっていく。

良いシステムを作るために、開発者はこの議論を効果的なものにすることを考えなくてはならない。特に、ユーザとの議論の際に、議論のたたき台としてのサンプルを用意することは非常に重要である。たたき台があることにより、議論のフォーカスが定まる上に、ユーザがシステムを具体的にイメージできるようになるため、効果的な議論が可能になる。

このとき、たたき台は動作可能なシステムであることが望ましい。すなわち、たたき台はその時点におけるユーザの全ての要求を網羅した、実際に操作可能なシステムであることが望まれる。そのようなたたき台を囲んで議論することにより、システムの機能やインターフェース設計といったシステムの外面的な要求はもちろんのこと、挙動やパフォーマンスといった内面的な要求も合わせて吸い上げることができるようになり、より効果的な要求収集が可能になる。

また、たたき台としての動作可能なシステムは、要求を実現しうる最もシンプルな構造のもとに作成されていることが望ましい。開発者は、ユーザとの議論の結果を逐次システムに反映し、システムを成長させていかなければならないからである。その際に発生するシステムの修正作業の手間を軽減するには、その構造が単純で分かりやすいものであることが必要不可欠である。

たたき台は、毎回の反復において素早く作成されなくてはならない。さもないと、前回の議論に対するユーザの印象が薄れてしまい、要求収集作業において手戻りが発生してしまう。すなわち、ユーザが前のたたき台について評価した際の議論の内容を忘れてしまい、同じ内容について何度も議論しなくてはならなくなってしまう。また、たたき台の作成に時間がとられることによって、

開発に膨大な時間がかかってしまったり、反復が十分にできずユーザの要求が十分定まらないうちに納期を迎えてしまったりする可能性がある。

しかし、そのような動作可能なシステムを素早く作成することは簡単なことではない。オブジェクト指向に基づきこれを行う際、システムの要求定義とクラス設計の間のギャップが問題になる。要求定義からクラス設計を導く際に一般的に行われている方法は精度が低い上に、効率が悪い。

また、クラス設計が定まってから、その実装が完了するまでの過程も問題である。最近の CASE ツールはクラスモデルからソースコードを生成できる機能を備えていることが多いが、この機能により生成されるソースコードは実装のテンプレートに過ぎない。すなわち、クラスモデルにおける個々のクラスの中に初期化されていない属性と空のメソッドが定義されたソースコードが生成される。開発者はこのようなテンプレートの中身を逐一実装していかなくてはならないため、依然として負担は大きい。

Toriaezer は、動作可能なシステムをオブジェクト指向に基づいて素早く作成することを支援するためのツールである。オブジェクト指向システム作成の際の一般的な問題である、要求定義とクラス設計の間のギャップを解消するとともに、実装を半自動化することにより、完成されたシステム作成の過程を効率化する。

2.背景

2.1.要求定義とクラス設計の間のギャップ

オブジェクト指向システムを作成するにあたり、要求定義からクラス設計を導き出す際には名詞抽出法がしばしば用いられる。名詞抽出法[14]とは、ユーザの要求を矛盾または過不足なく端的に表す文章から名詞を抽出し、その妥当性を吟味することでクラスを識別する方法である。

オブジェクト指向システムを設計するに当たり、名詞抽出法は様々な形で利用されている。例えば、ユーザがシステムを利用する際のシナリオからオブジェクトを識別し、それらの構造を UML におけるオブジェクト図により記述する

ことにより、クラスを識別しようというやり方も、名詞抽出法の拡張に他ならない。同様に、シナリオから識別したオブジェクトが協調する様子を、UML におけるシーケンス図、あるいは CRC カードを利用して記述することによりクラス設計を行うやり方も、名詞抽出法の拡張であると捉えることができる。

名詞抽出法は便利であるが、その問題点は、名詞抽出法を適用する対象となる適切な要求記述を用意することが難しいことにある。一般的にこの文章は非形式的に記述されるため、要求が網羅されているか、矛盾がないかといったチェックを行うことは難しいということがその理由である。

要求定義とクラス設計の間のギャップは、ここに存在する。すなわち、クラス設計を行うにあたり、要求を矛盾または過不足なく端的に表す記述を用意することは難しい。

2.2.クラス設計実装時の問題

最近では、UML モデルからオブジェクト指向プログラミング言語のソースコードを生成することができる CASE ツールが普及してきている。これらの CASE ツールが生成するソースコードは、多くの場合クラスに初期化されていない属性および空のメソッドが定義されたものであり、いわば実装の雛形である。当然のことながら、モデルから実装を生成することはできない。

従って、開発者は生成した雛形をもとに改めてプログラムを書いていかなければならないことになる。雛形をもとにプログラムを記述する作業は、実装すべきメンバが明らかになっているため、何を実装すべきか分かりやすいという利点があるが、作業量的には大した効率化につながらない。

また、プログラムを記述する際、似たような実装を行うことは多々ある。例えば、属性を取得するメソッドはほとんどのクラスについて存在することになる。属性を設定するメソッドについても同様である。

このような同じ種類のメソッドは、必然的に似たような実装になる。それはつまり、アルゴリズムは同じだが、そのアルゴリズムによって処理される対象が異なるということである。実際のプログラムの中にはこのような状況が数多く存在する。これは同じことの繰り返しが多分に含まれるという意味で、非常に非効率なことである。

3.Toriaezer

3.1.概要

Toriaezer は、オブジェクト指向に基づいて完成されたシステムを素早く作成するために、システムを特徴付ける機能およびデータを提供する部分のプログラム（以降コアプログラムと呼ぶ）の作成を支援するツールである。具体的には、Toriaezer は日本語で記述された要求定義とその中で使用された名詞の実装定義から、Java の実装コードを生成する。

Toriaezer を利用した開発において、開発者はまず日本語で要求を定義し、その中で使用した名詞について実装時の仕様を定義し、Toriaezer を利用してコアプログラムのソースコードを生成する。そして開発者は、その生成されたソースコードを利用してシステムを実現する Main プログラムを書く。このようなアプローチで開発を行うことにより、完成されたシステムを作成するための工期を、従来の約半分に縮めることが可能である。

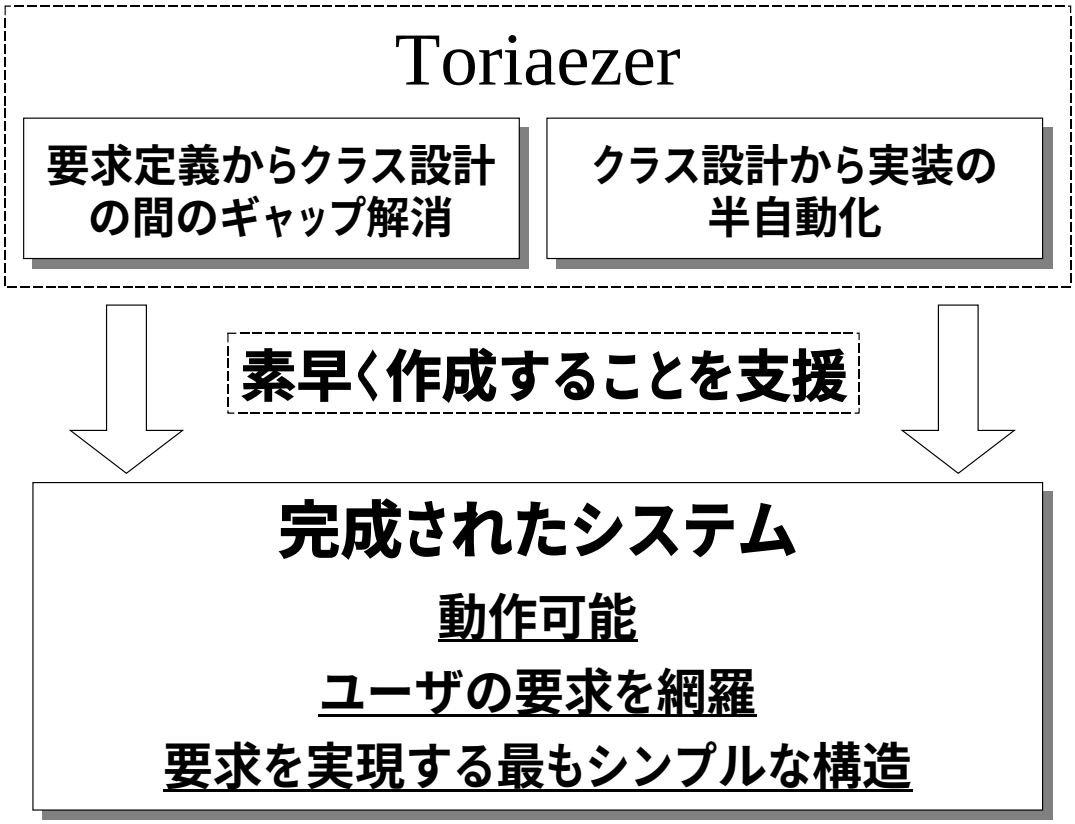


図 23　Toriaezer

要求定義ファイルはシステムに対する要求を単純な日本語で記述するものであり、クラス設計の記述も兼ねている。区切り文字定義ファイルは、Toriaezer が要求定義ファイルをパースする際に参照するファイルであり、ここに定義された区切り文字をもとに要求記述の構文の解析がなされ、クラスモデルがオブジェクト構造として Toriaezer 内部に生成される。名詞定義ファイルは、Toriaezer がソースコードを生成する際に参照するファイルであり、Toriaezer はここに記述された名詞の実装情報をもとにクラス設計の実装を行う。

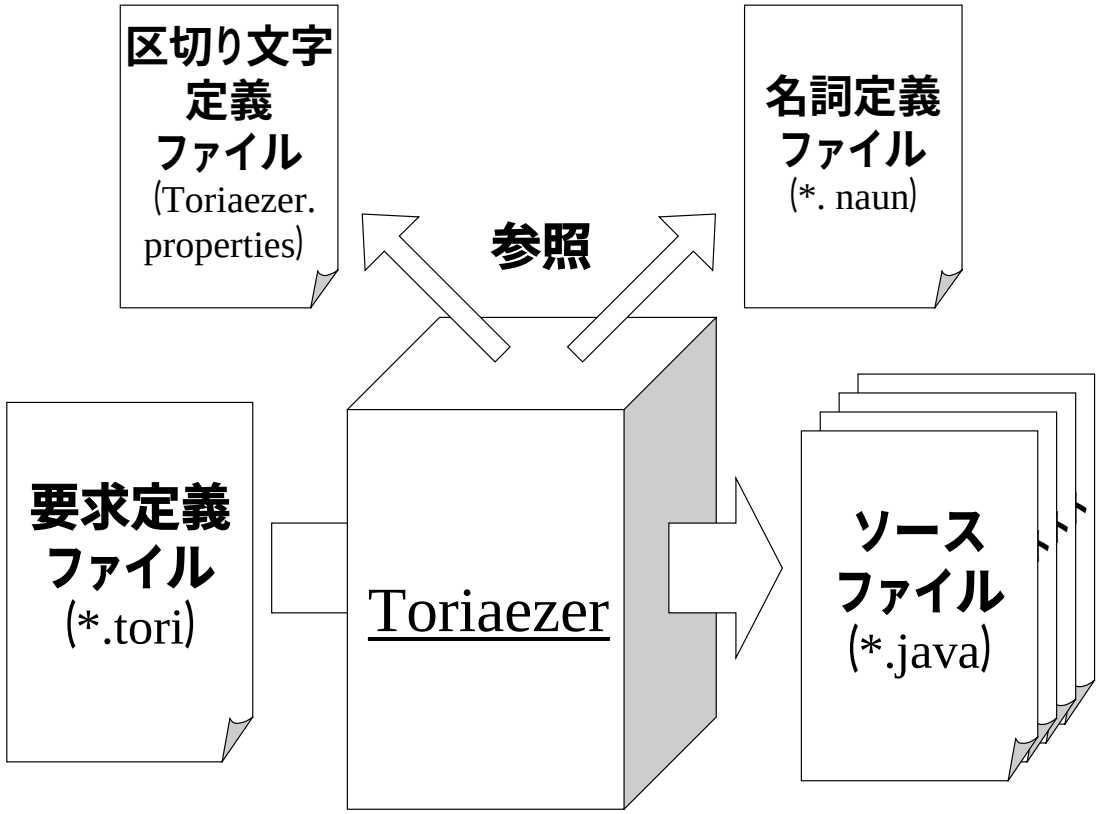


図 24　Toriaezer の挙動

3.2.Toriaezer における要求定義

3.2.1.要求定義ファイルの記述

Toriaezer における要求定義は、システムがユーザに対して提供するサービスおよび、サービスを提供するためにシステムが実行すべき操作であるサービス指令を要求定義ファイルに記述することにより行う。この中で、一つのサービスを定義した文のことを特にサービス文とよび、同様に一つのサービス指令を定義した文のことをサービス指令文と呼ぶ。

サービス文は、システムがユーザに提供する高レベルのサービスを非形式的に記述するものである。これはすなわち、システムを利用して「誰が何をするのか」についての記述である。

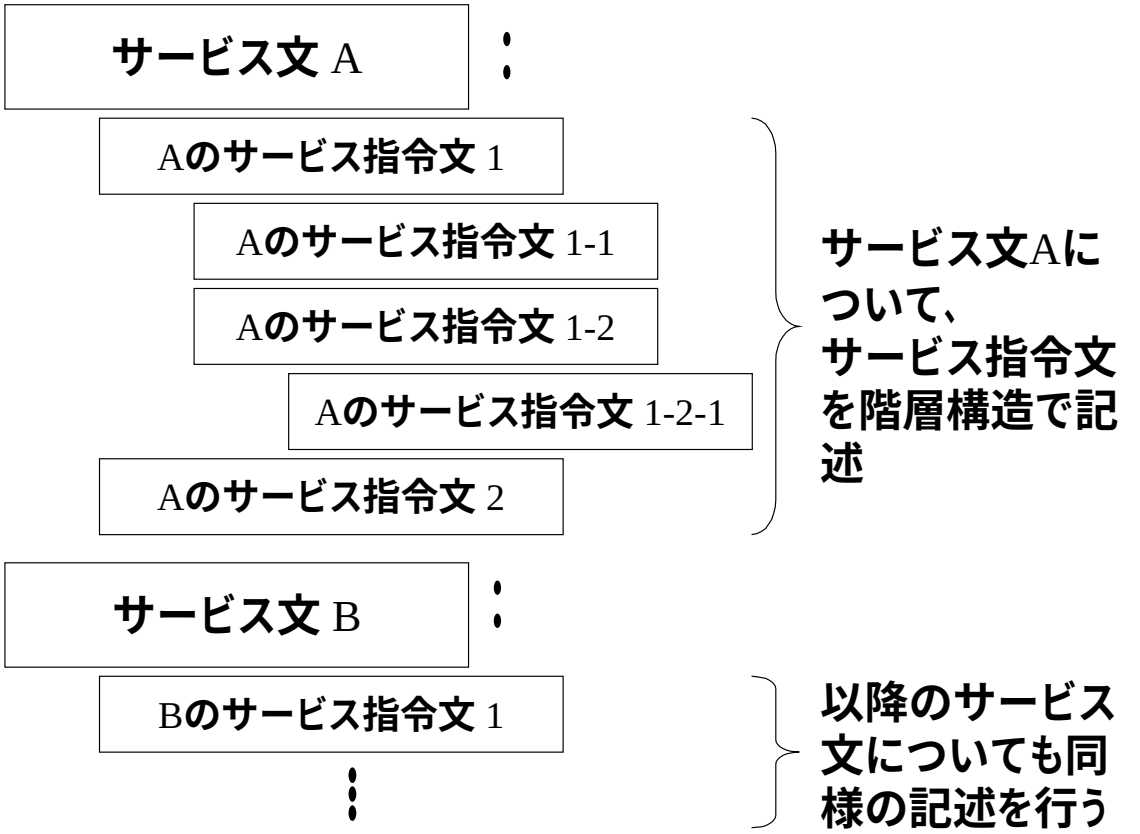


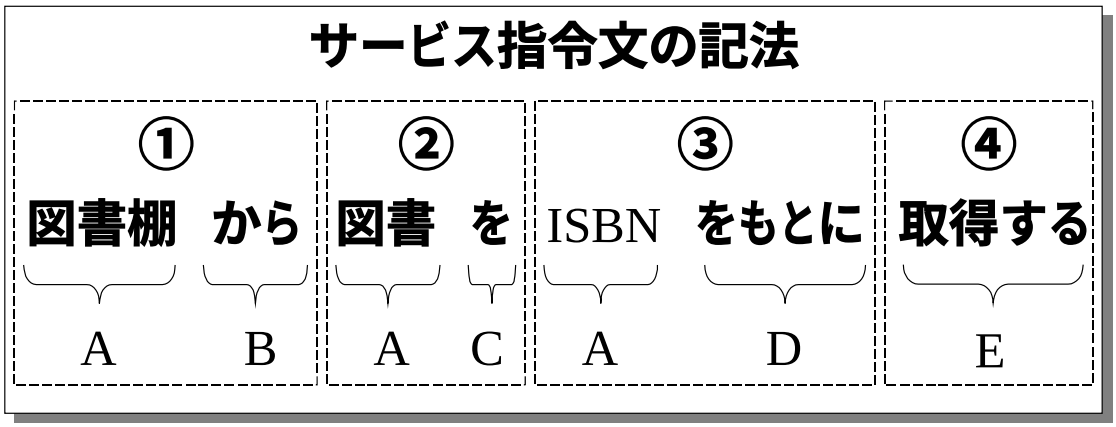
図 25　要求定義ファイルの記述

一方サービス指令文は、あるサービスを実現するためにシステムが行うべき操作を形式的に記述するものである。これはすなわち、サービスを実現するためにシステムが「何をどうするか」についての記述である。

サービス文は、一群のサービス指令に目的を与えるものであり、要求定義の最上位階層に位置づけられる。そしてそれぞれのサービス文に追従する形で、サービス指令文が階層構造で記述される。

3.2.2.サービス指令文の記法

図 26 はサービス指令文の記法を表したものである。以下ではこれに基づいて説明を行う。



① 「環境」を表す名詞句
② 「目的」を表す名詞句
③ 「引数」を表す名詞句
④ 「操作」を表す動詞句
※1 ①は最上位階層でないサービス指令文では記述しない
※3 ②と④は必ず一つずつ記述
※4 ③は省略可かつ複数記述可
※5 ①～③は任意の順で記述可（④は必ず文の末尾に記述）

A 名詞
B 前に来る名詞とともに「環境」を表す名詞句となる助詞
C 前に来る名詞とともに「目的」を表す名詞句となる助詞
D 前に来る名詞とともに「引数」を表す名詞句となる助詞
E 動詞
※6 B、C、Dに使用する文字は「Toriaezer.properties」にて変更可

図 26　サービス指令文の記法

図中の番号は、サービス指令における高レベルの構成を表している。ここで①は「環境」、②は「目的」、③は「引数」を表す名詞句であり、④は「操作」を表す動詞句である。環境とはこのサービス指令の適用対象であり、目的とはこのサービス指令の操作対象を表現する。そして引数はサービス指令を実行する上で考慮する必要のある物事を表現し、操作はそのサービス指令により引き起こされる振る舞いを表現する。

なお、環境はサービス文直属のサービス指令文以外は記述しない。親サービス指令の目的が子サービス指令の環境として引き継がれるためである。図 27 はその様子を示している。要求記述の表記上、子サービス指令に環境は記述されないが、実質的には環境は存在している。

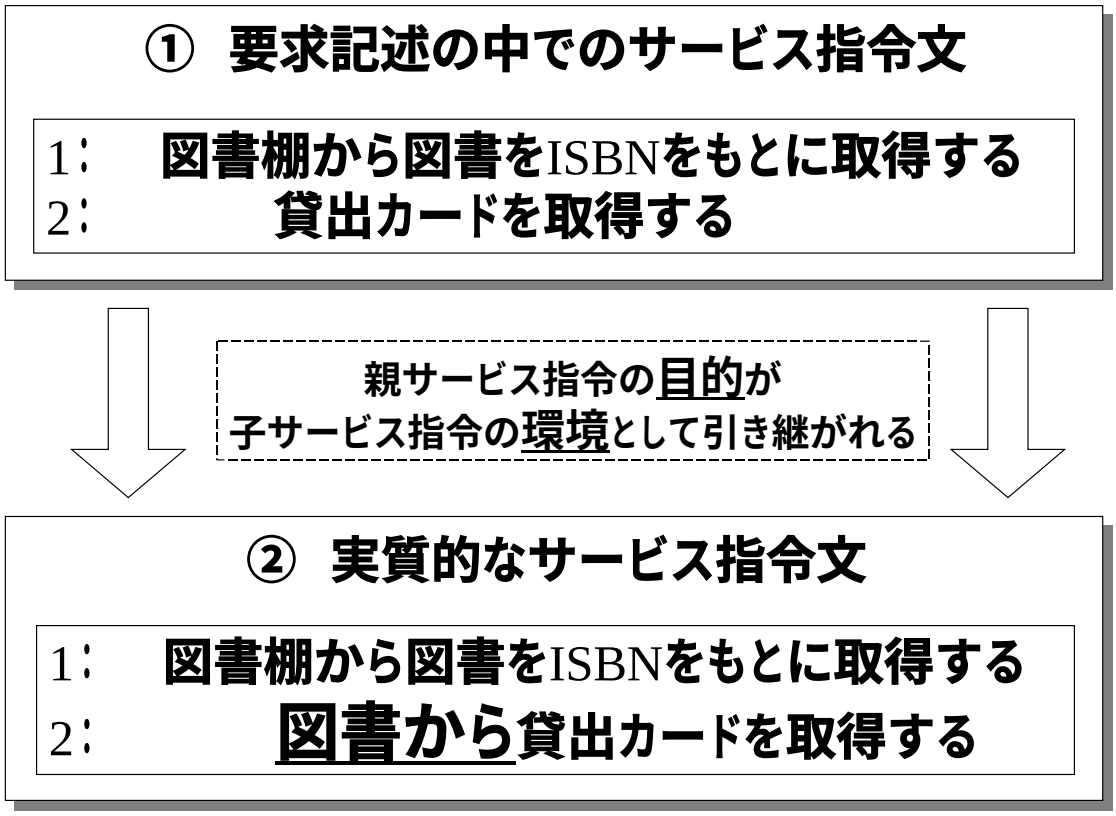


図 27　サービス指令文の親子関係

目的と操作は、「何をどうする」に相当しサービス指令に不可欠な要素であるため、サービス指令文には必ずこれらをひとつずつ含まなくてはならない。引数はサービス指令にとってオプショナルな要素であり、省略可能でありかつ複数記述が可能である。環境、目的、引数は任意の順番で記述することができるが、操作は必ずサービス指令文を構成する最後の要素でなければならない。すなわち、以下のような表記はどれも妥当である。

- 図書棚から図書を ISBN をもとに取得する
- 図書を図書棚から ISBN をもとに取得する
- ISBN をもとに図書棚から図書を取得する

図 26 のアルファベットは、サービス指令文における低レベルの構成を表している。ここで A は名詞であり、E は動詞である。後述するが、名詞については名詞実装時仕様定義ファイル（*.naun）にて実装時の仕様を定義することになる。現状において、動詞については標準で備わっているものを利用する必要があり、それらは「取得する」「設定する」「追加する」「削除する」である。標準でこれ

らの動詞を用意した理由は、これらの動詞がそれぞれ、オブジェクト指向システムにおいて非常に本質的な振る舞いである、関連先の取得、対１関連の構築、対多関連の構築、関連の削除に対応しているからである。これらの動詞の実装時仕様は、Toriaezer が内部的に保持している。

B から D はそれぞれ助詞であり、前に来る名詞を高レベルの構成、すなわち環境、目的、引数のいずれかとして割り当てる役割を担っている。すなわち、B は前に来る名詞とともに、そのサービス指令文における環境を示す名詞句となる。同様に C は目的、D は引数を示す名詞句となる。この助詞には、区切り文字定義ファイル（Toriaezer.properties）において任意の文字を設定することができる。

なお、要求定義ファイルに実際にサービス指令文を記述する際には、名詞あるいは動詞を示す言葉を「」でくくる必要がある。日本語は英語と違い、分かち書きを行わないために、言語レベルでトークンの切り出しが不可能であるためである。従って「」によってトークン明示しなければ、サービス指令文において助詞と名詞あるいは動詞の判別を行うことができない。

3.2.3.サービス指令文によるクラス設計

サービス指令を記述することとは、作成しようとしているシステムのクラス設計を記述することに他ならない。Toriaezer はサービス指令を解析することによりクラス設計を導き出すためである。

Toriaezer は、それぞれのサービス指令文を解析することによりクラス設計を行う。すなわち、環境を表す名詞はクラスとなる。そしてそのクラスに対し、環境を除いたサービス指令文がメソッドとして登録される。そして Toriaezer はそのようなクラス設計をオブジェクト構造として内部的に生成する。

要求定義ファイルの中では、同じ環境を持つサービス指令文あるいは全く同じサービス指令文が複数定義されることが想定される。そのような場合、Toriaezer は最初の一回だけクラスあるいはクラスに対するメソッドを登録し、以降は無視する。すなわち、何度同じ環境が要求定義ファイル中に出現してもそれに対するクラスは一度しか作成されない。また、同様に何度全く同じサービス指令文が出現しても、それに対するメソッドは一度しか作成されない。

なお属性、メソッドのパラメータあるいは戻り値に関しては、クラス設計の段階で明らかにされない。そのようなメソッドに関する詳細は、Toriaezer が内部に保持している動詞の実装テンプレートに規定される。同様に、属性の詳細

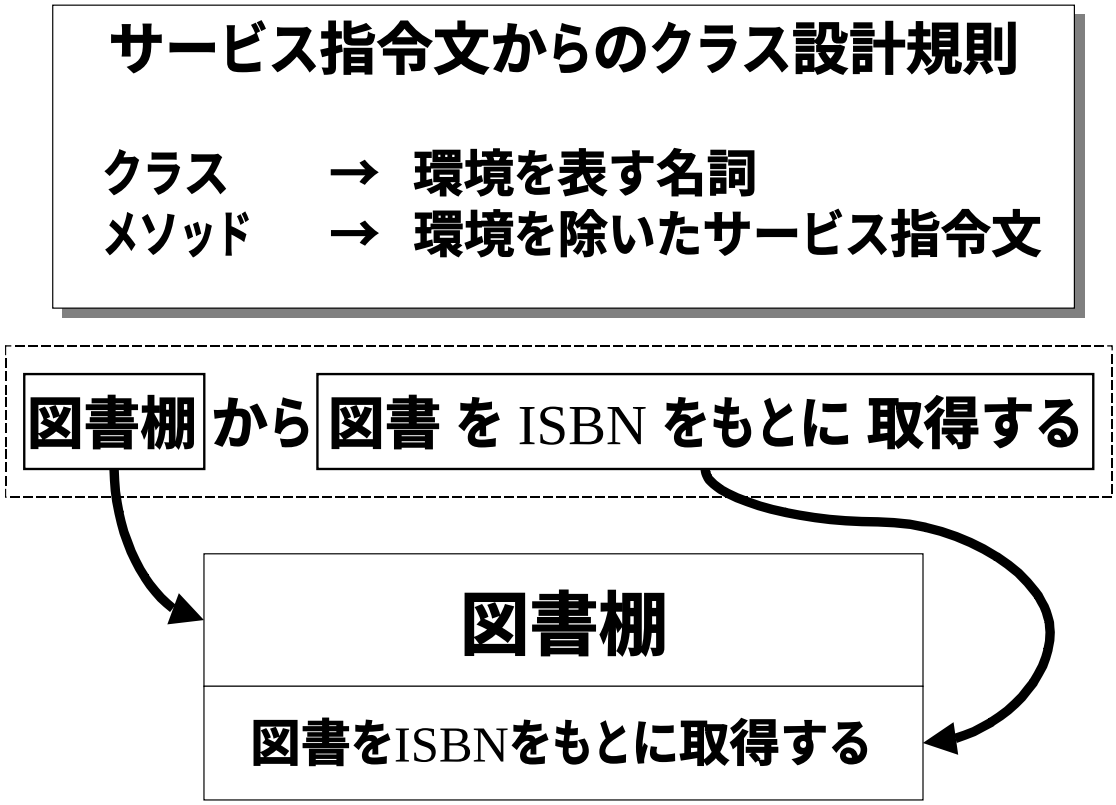


図 28　サービス指令文から生成されるクラス設計

に関してもこのテンプレートに規定される。これについては後に詳しく説明する。

サービス指令の階層構造は、システム動作時におけるオブジェクト同士のメッセージパッシングの抽象である。すなわち、サービスの階層構造は、あるサービスをユーザに提供する際、システムがクラスレベルで行うメッセージの受け渡しの様相を規定する。図 29 はその様子を UML のシーケンス図を用いて記述した例である。

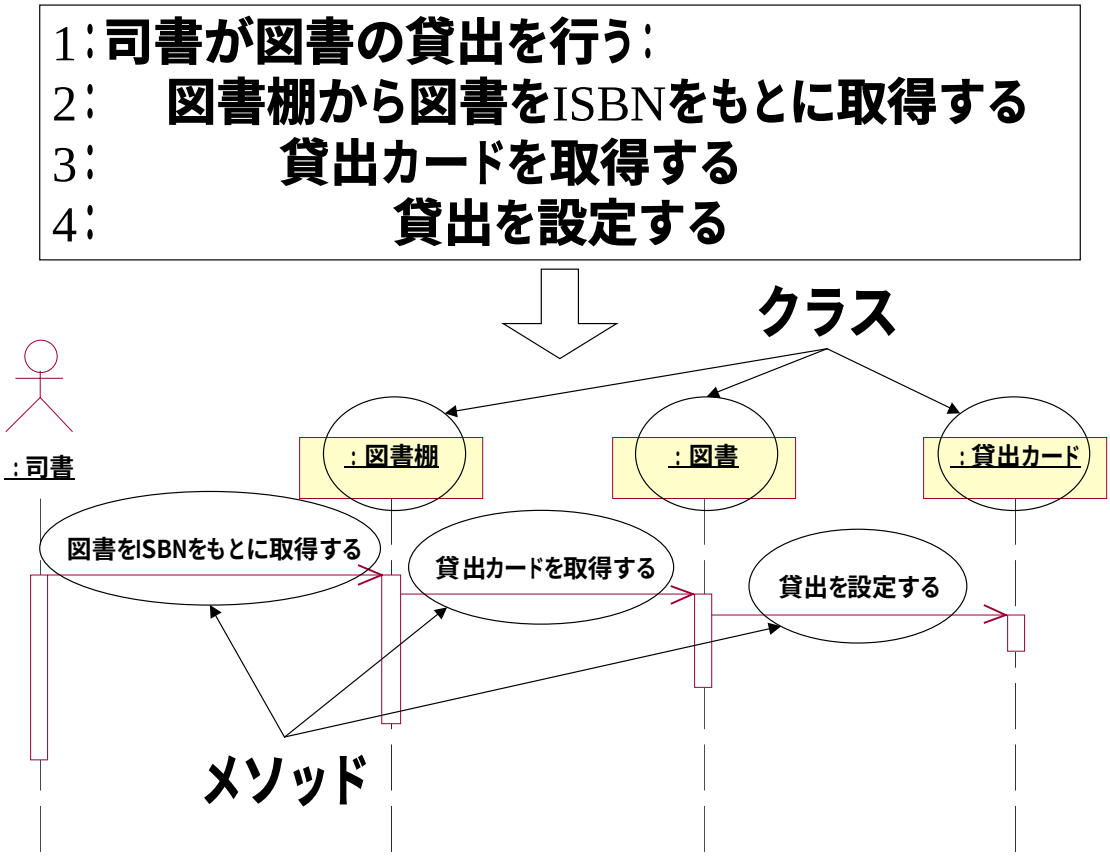


図 29　サービス指令文の階層構造が表現するクラスレベルのメッセージパッシング

図 29 における要求記述の 1 行目はサービスの記述である。2 行目以降はそのサービスを実現するためのサービス指令が階層構造で記述されている。この記述は、オブジェクトレベルでは以下のように解釈することができる。

司書が図書の貸出を行うためには、まず図書棚から貸出対象の図書を xxxx という ISBN をもとに取得する。そしてその図書から、最新の貸出カードを取得する。そしてそのカードに対してその図書が貸出された旨を設定する。

3.2.4.サービス指令文によるクラス設計の利点

サービス指令文を記述することによりクラス設計を導き出すことは、クラス識別の発想を助け、必要最小限のクラス構造を導きだすことを支援するという意味で効果的である。

オブジェクト指向システム開発におけるクラスの識別はいわば発明である。適切なクラスを導き出すには、要求を実現する世界を適切に識別し、その中からシステムに必要な概念を抽出しなくてはならない。

これを図解を用いたモデリングによって行うのは難しい。図解には表現力がありすぎるためである。オブジェクト指向開発の経験が浅い開発者にとっては、発想のよりどころとなるものがないため、常に導き出したクラスの妥当性に不安を感じながら作業を行うことになる。また熟練した開発者にとっても、いろいろなものが見えすぎるがために必要性を超えたクラスを識別してしまったり、技術的好奇心のため無駄に高度に抽象化されたクラスを識別してしまったりするという弊害がある。

サービス指令を階層構造で記述するという行為は、図解に比べクラス識別時における発想の幅を狭めるが、その分確実なクラスの識別の仕方が可能である。そのため、初心者または熟練者に関わらず良いクラスが導き出せる可能性が高い。開発者は「このサービスを実現するには、〜したい」といった発想を繰り返すことで、クラスを導き出すことができる。こうして導き出されたクラスは、サービスを実現するために必要であるという強力なモチベーションに支えられており、無駄や冗長さが入り込む余地を少なくする。

また全てのサービス指令の記述は、その最上位階層に存在するサービスを満足するものでなくてはならない。逆に言えば、サービスを満足する目的でサービス指令を構成しなくてはならない。従って、導き出されるクラス設計を必要最小限の構造を備えたシンプルなものにすることができる。

3.3.ソースコード生成のメカニズム

Toriaezer を利用してソースコードを生成するためには、要求定義の記述の中で使用した名詞に対する実装時の仕様を定義する必要がある。それは名詞定義ファイル（*.naun）の中で行う。名詞定義ファイルでは、一つの名詞について４種類の情報を定義する。それらはすなわち、継承する名詞、実装時の型、実装時の名前、唯一性である。

ソースコードの生成は、Toriaezer が要求定義ファイルから生成したクラス設計をもとにして行われる。Toriaezer が生成するクラス設計では、サービス指令文中の環境に対応する名詞がクラスとして、そのサービス指令文から環境を除いたものがそのクラスのメソッドとして定義されている。

クラスの実装は、クラス設計におけるクラス名に対応する実装情報を名詞定義ファイルから得ることにより、機械的に行われる。

クラスに対するメソッドおよび属性の実装は、クラス設計中のメソッド名に

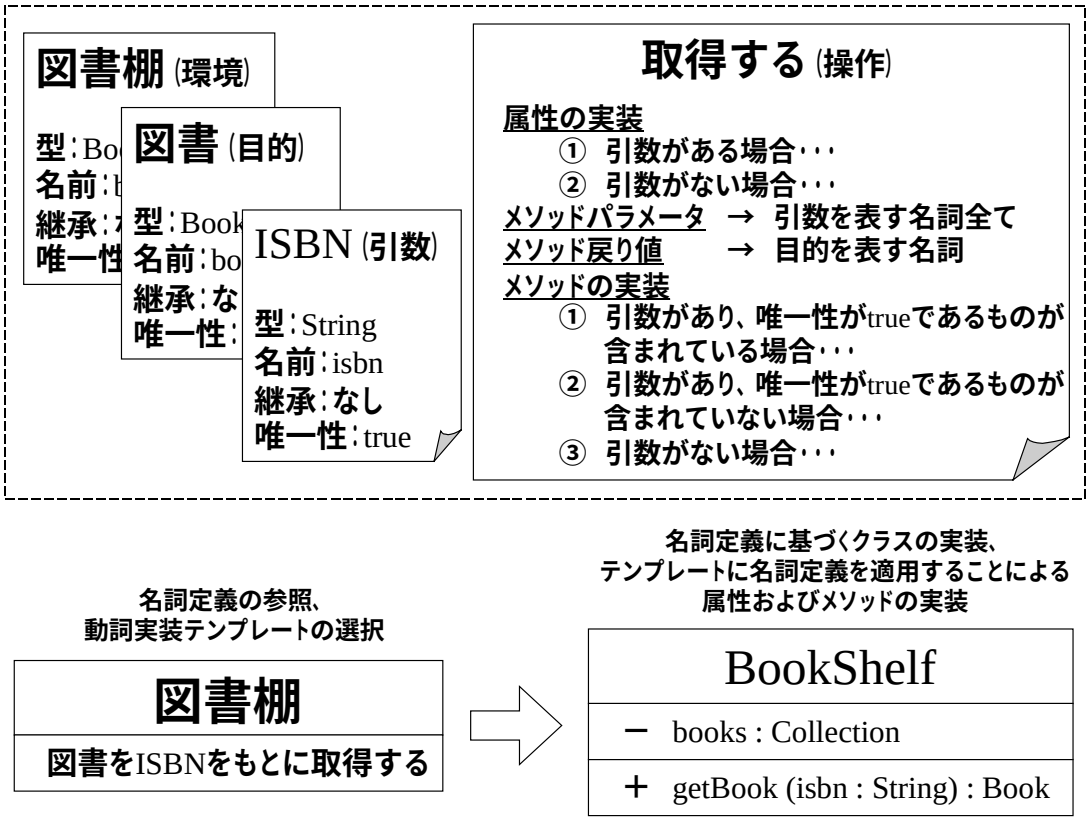


図 30　Toriaezer のソースコード生成メカニズム

おける操作を表す動詞の実装テンプレートに、目的および引数を表す名詞の実装情報を適用することにより行われる。動詞の実装テンプレートは、現状においては Toriaezer が内部的に保持している。Toriaezer は、クラス設計におけるメソッド名から操作を表す動詞を抽出し、それに対応する実装テンプレートを選択する。この実装テンプレートは実装対象のクラスに対してクラス設計におけるメソッド名の目的および引数をもとに属性およびメソッドを実装するための雛形を規定している。図 30 はその様子を表現している。

このように、Toriaezer はクラス設計に基づいて名詞定義および動詞の実装テンプレートを利用することによりソースコードを生成することになる。

なお、Toriaezer によって生成されるソースコードは、開発者がそれを利用して Main プログラムを作成することを前提としており、出来る限り可読性の高いものを出力するようにしている。これを実現するソースコードの整形フォーマットは現状において Toriaezer が内部に保持している。

3.4.Toriaezer を利用した開発手順とその効率

開発者が Toriaezer を利用してシステムを作成するには、次のような手順を踏むことになる。まず、要求定義記述の中で利用する区切り文字を必要に応じて Toriaezer.properties 上で編集する。そしてその区切り文字を使用して要求定義ファイル上にサービスおよびサービス指令を記述することで要求を定義し、その中で利用した名詞について名詞定義ファイル上で実装時の仕様を記述する。そうして作成された要求定義ファイル、名詞定義ファイルを、区切り文字定義ファイルとともに Toriaezer に読み込ませることによりソースコードを生成する。そして開発者は、生成されたソースコードを利用してシステムを実現するための Main プログラムを作成する。

要求定義は日本語で記述できる上に、非常に単純な記法であるため、その記述に時間はさほどかからない。また、同様の理由で修正も容易である。

区切り文字定義ファイルおよび名詞実装時仕様定義ファイルは完全な再利用が可能である。区切り文字定義ファイルに関しては一度使いやすい区切り文字を定義してしまえば後に変更する必要が発生することはまれである。仮にそのような変更の必要性が発生しても、定義する内容自体が少ないので大した作業量にならない。名詞実装時仕様定義ファイルに関しては一度その仕様を定義した名詞を同ファイルに蓄積しておけば自ずと名詞の実装辞書が出来上がる。これは開発者が複数のソフトウェア開発において一貫した概念の実装を行うことができるという意味でも意義深いことである。

また、生成されたコアプログラムを利用する Main プログラムの作成に関しても大した時間はかからない。コアプログラムが作成されれば、それを動作させるための環境としての Main プログラムの作成は容易である。その理由としては、その作業はコアプログラムの作成ほど創造性を要さない作業であることに加え、プログラムの実行環境を提供するフレームワークが一般的に広く普及しており、それを利用することで目的に応じたコアプログラムの実行環境が容易に得られるからである。

従って完成されたシステムを作成するに当たり Toriaezer を用いることで、大幅な開発効率の向上が期待できる。

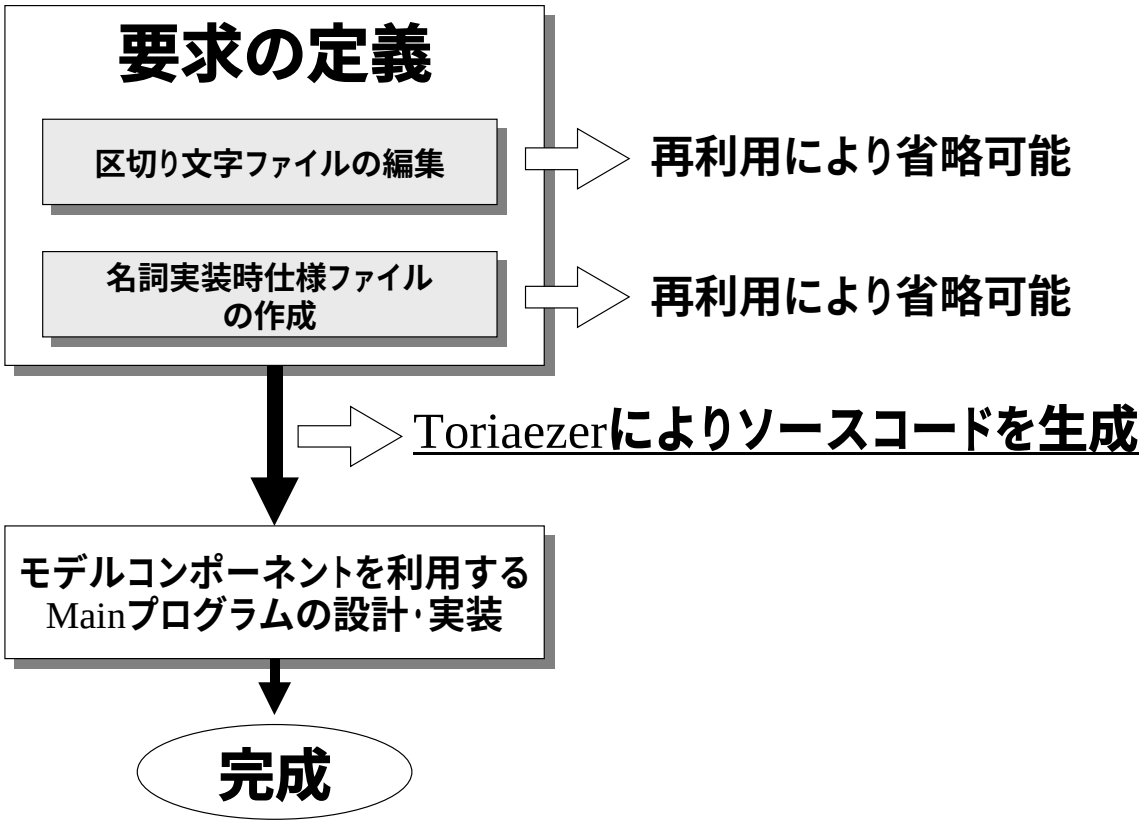


図 31　Toriaezer を利用した開発手順

4.Toriaezer の利用例

4.1.適用対象

筆者は、東京都中野区の小原病院においてその業務を支援する情報システムの開発に携わっている。その開発はユーザと密に協調した形で行っており、特にシステム作成時は、そのシステムの直接的なユーザとなる現場の方々から反復的に要求を収集し逐次システムに反映するということを行っている。すなわち、開発者が作成したサンプルをユーザとともに評価し、その結果を受けてサンプルを修正あるいは作り直すといった作業を反復的に行っている。

外来患者請求システムは、こうして開発されたシステムの一つである。このシステムは、外来患者のその日の診療にかかる一部負担金額を算出するととも

に、その領収書を発行するためのシステムである。このシステムは現状業務において即利用する必要があったため、ユーザとの入念な議論が必要であった。この際、動作可能なサンプルシステムをたたき台として作成し、それについてユーザとともに評価を行った。

筆者はこのシステムについて、Toriaezer を利用して再度動作可能なシステムを作成してみることとした。以下ではその様子について述べる。

4.2.外来患者請求システム

実際の外来患者請求システムはレガシーシステムから診療明細に関する情報を吸い上げ、それをユーザが必要に応じて編集し、患者の一部負担金計算および領収書の出力を行っているが、サンプルシステムはスタンドアロンシステムとしてこれをシミュレーションすることとした。そして、Toriaezer を利用してそのサンプルシステムを作成するにあたり、以下のような要求を設定した。

Ａ　診療報酬請求の内容を編集する
Ｂ　未収金請求の内容を編集する
Ｃ　診療外費用請求の内容を編集する
Ｄ　診療外請求項目を管理する
Ｅ　請求金額を計算する
Ｆ　領収書を発行する

Ａについて、その目的は患者の診療報酬点数を必要に応じて修正することである。レガシーシステムの持つ診療明細に関する情報は現状において信頼性が低く、誤りがしばしば存在するため事務員が随時修正する必要がある。
Ｂについて、その目的は患者からの未収金を合わせて請求することである。診察時の中で、検査結果が出てからでないと診療報酬として算定できない特別な検査が行われた場合などは、後日未収金として請求する必要がある。

Ｃについて、その目的は保険請求ができない費用について請求することである。文書料や予防接種、健康診断などは診療報酬として算定できないため、診療外請求として請求する必要がある。

Ｄについて、その目的は上述の保険請求ができない項目をすぐに領収書に設定できるようにしておくためである。そのような請求が発生するたびに診療外請求項目名をキーボードで入力するのでは効率が悪い。使用する項目をシステ

ムで管理することで、診療外請求項目名入力の手間を軽減する。
Ｅについて、その目的は患者に対して本日の受診にかかった金額を請求することである。患者の保険の受給状況によっては診療点数が請求金額に全く反映されない場合があるので、一部負担金を事務員が設定できる機能も合わせて必要である。また、自費の場合は一部負担金の計算法が異なるので、それにも対応する必要がある。
　Ｅについて、その目的は患者に請求金額の明細を提示することである。兼ねてから患者から領収書のクオリティの低さが指摘されていたため、きちんとした領収書を出力するようにする必要がある。

4.3.要求定義ファイルの記述

　要求定義ファイルの記述にあたっては、上述した要求をそのままサービス文として採用した。そしてそれらのサービスを実現するためにシステムが行うべき操作を、サービス指令文として記述していった。以下は作成された要求定義ファイル（Receipt.tori）である。

```
1:  診療報酬請求の内容を編集する：
2:      「ドキュメント」から「領収書」を「患者 ID」をもとに「取得する」
3:              「診療点数」を「設定する」
4:              「投薬点数」を「設定する」
5:              「注射点数」を「設定する」
6:              「処置点数」を「設定する」
7:              「手術点数」を「設定する」
8:              「検査点数」を「設定する」
9:              「レントゲン点数」を「設定する」
10:             「処方点数」を「設定する」
11:             「一部負担金額」を「設定する」
12: 未収金請求の内容を編集する：
13:     「ドキュメント」から「領収書」を「患者 ID」をもとに「取得する」
14:             「未収項目名」を「設定する」
15:             「未収点数」を「設定する」
16:             「未収金額」を「設定する」
17: 診療外費用請求の内容を編集する：
18:     「ドキュメント」から「領収書」を「患者 ID」をもとに「取得する」
19:             「診療外請求」を「追加する」
20:                     「診療外請求項目」を「設定する」
21:                     「課税の有無」を「設定する」
22:                     「数量」を「設定する」
23:                     「金額」を「設定する」
24:     「ドキュメント」から「診療外請求項目」を「診療外請求項目名」をもとに「取得
する」
25: 診療外請求項目を管理する：
26:     「ドキュメント」に「診療外請求項目」を「追加する」
27:             「診療外請求項目名」を「設定する」
28:     「ドキュメント」から「診療外請求項目」を「削除する」
29: 一部負担金を計算する：
30:     「ドキュメント」から「領収書」を「患者 ID」をもとに「取得する」
31:             「一部負担金額計算の有無」を「設定する」
32:             「自費計算の有無」を「設定する」
```

例として 1 行目のサービス文とそのサービス指令文について説明する。ここでは、診療報酬請求の内容を編集するには、「ドキュメント」から「患者 ID」を指定することにより「領収書」を取り出し、それに対して「診療点数」や「投薬点数」などといった情報を設定することを行う必要があることを示している。

これを解析することにより得られるクラス設計としては次のようになる。まず「ドキュメント」および「領収書」がクラスとなる。そして「ドキュメント」クラスに対しては「『領収書』を『患者 ID』をもとに『取得する』」というメソッドが登録され、同様に「領収書」クラスには「『診療点数』を『設定する』」、「『投薬点数』を『設定する』」などのメソッドが登録されることになる。

以下は解析の際に利用された区切り文字定義ファイル（Toriaezer.properties）である。

```
#######################################
# 複数指定可（カンマ区切り）、文字の長さは任意
#######################################

# 名詞を「環境」として型付けする助詞
toriaezer.document.delimiter.semantic.TEnvironmentPostfix = から,に
# 名詞を「目的」として型付けする助詞
toriaezer.document.delimiter.semantic.TObjectPostfix = を
# 名詞を「引数」として型付けする助詞
toriaezer.document.delimiter.semantic.TParameterPostfix = をもとに，を指定して
# 「引数」を表す複数の名詞を連結する助詞
toriaezer.document.delimiter.semantic.TParameterCombinationInfix = と

#######################################
# 複数指定可（カンマ区切り）、それぞれ１文字
#######################################

# サービス指令文の階層を示す記号
toriaezer.document.delimiter.syntax.TCommandLayerSymbol = ¥t
# 無視可能な記号
toriaezer.document.delimiter.syntax.TNegligibleSymbol = 、, 。
# サービス文の末尾に付ける記号
toriaezer.document.delimiter.syntax.TServicePostfix = ：
# コメント行の先頭に付ける記号
toriaezer.document.delimiter.syntax.TCommentPrefix = #, #

#######################################
# 複数指定可（カンマ区切り）、それぞれ１文字、同じ文字を使うことは不可
#######################################

# 言葉（名詞または動詞）の開始を表す記号
toriaezer.document.delimiter.syntax.TWordPrefix = 「
# 言葉（名詞または動詞）の終了を表す記号
toriaezer.document.delimiter.syntax.TWordPostfix = 」
```

なお、筆者はこの要求定義ファイルにおけるサービス文直属の全てのサービス指令文は「ドキュメント」という環境を持つよう構成した。これにより、どのサービスを実行するに当たっても「ドキュメント」からオブジェクト構造を辿ればよいことになるため、Main プログラムの作成が容易になるからである。

4.4.ソースコードの生成

要求定義ファイル中の名詞は以下のように実装時の定義を行った。

作成した要求定義ファイルをもとに、Toriaezer はまず図 32 のようなクラス設計を行うことになる。

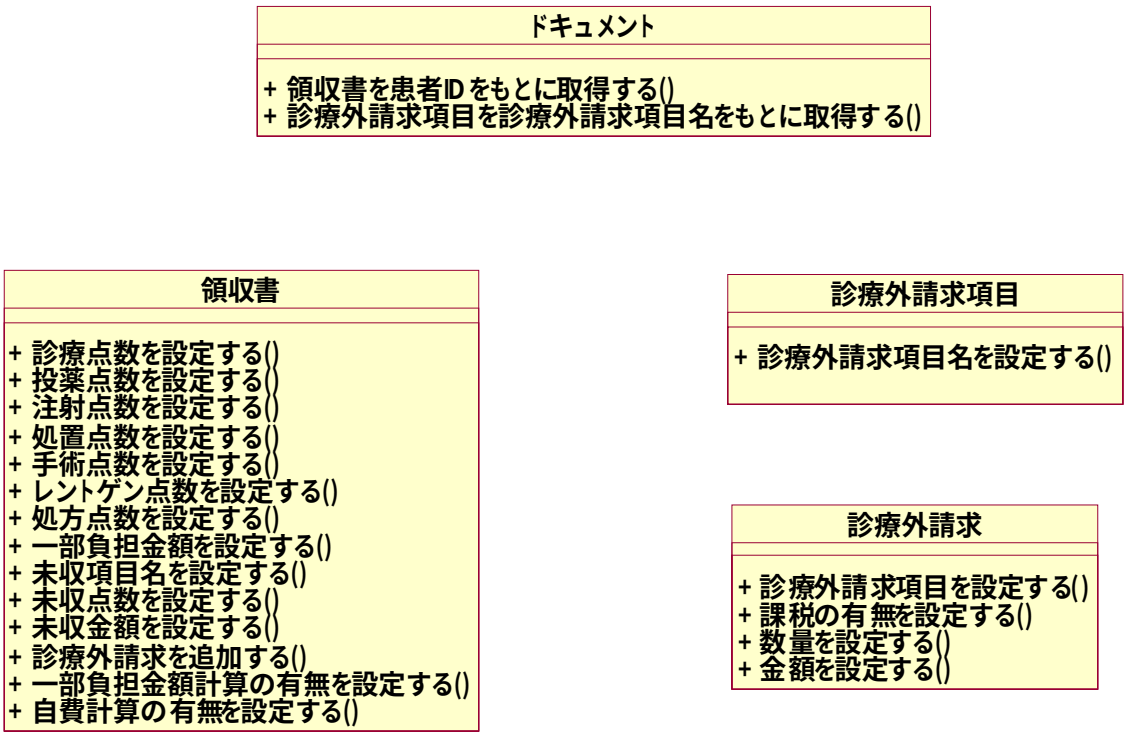


図 32　要求定義から導き出されたクラス構造

これに対し、以下のような名詞定義ファイルを読み込ませると、コアプログラムのソースコードは図 32 のような構造のもとに図 33 のクラス設計に対する実装を生成されることになる。なお、名詞定義ファイルの記法として、カンマ区切り構造の 1 番目に要求定義ファイル中での名詞、2 番目に要求定義ファイル中の継承する名詞、3 番目に実装時の型、4 番目には実装時の名前を記述する。

領収書, , Receipt, receipt, true
患者 ID, , int, patientId, true
診察点数, , int, diagnosisPoint, false
投薬点数, , int, medicationPoint, false
注射点数, , int, injectionPoint, false
処置点数, , int, treatmentPoint, false
手術点数, , int, operationPoint, false
検査点数, , int, examinationPoint, false
レントゲン点数, , int, xRayPoint, false
処方点数, , int, prescriptionPoint, false
未収項目名, , String, unearnedItemName, true
未収点数, , int, unearnedPoint, false
未収金額, , int, unearnedFee, false
数量, , int, amount, false
金額, , int, fee, false
一部負担金額, , int, totalFee, false
課税の有無, , boolean, isTaxed, false
一部負担金額計算の有無, , boolean, isCalculatable, false
自費計算の有無, , boolean, isPrivate, false
診療外請求, , NonMedicalCharge, nonMedicalCharge, true
診療外請求項目, , NonMedicalChargeItem, nonMedicalChargeItem, true
診療外請求目名, , String, nonMedicalChargeItemName, true

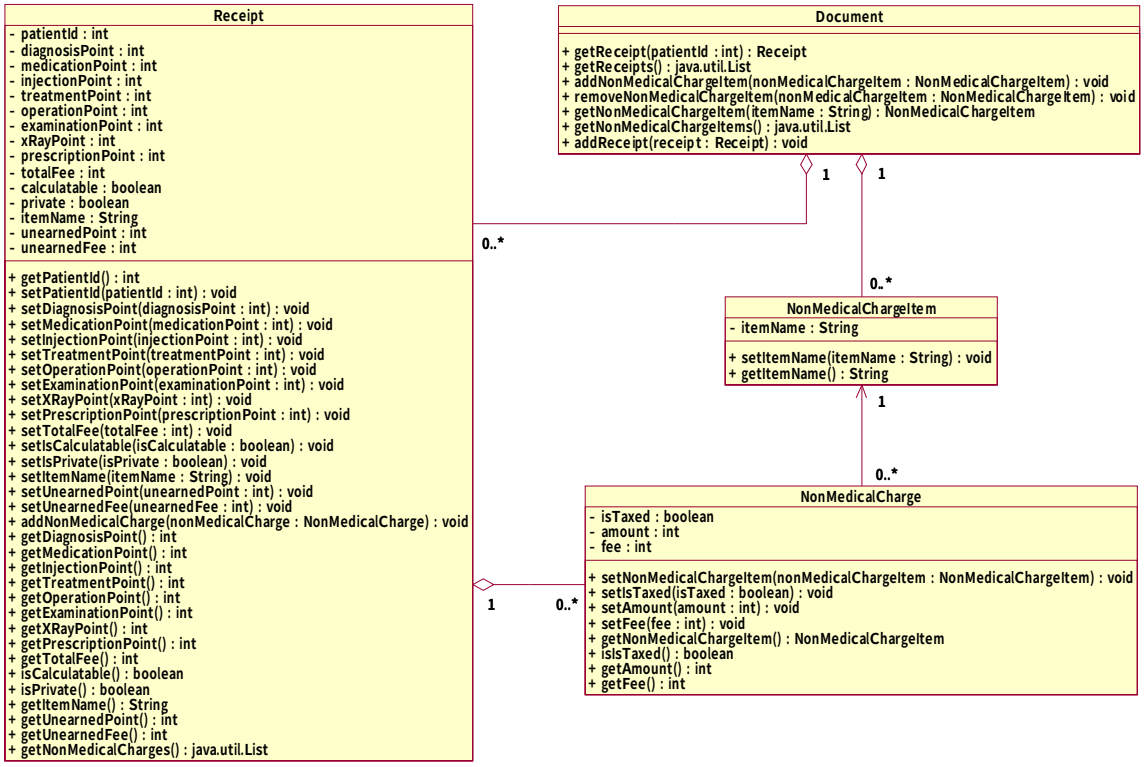


図 33　外来患者請求システムの実装時のクラス構造

要求定義ファイルのみから Toriaezer が生成したクラス構造と、最終的に生成されたコアプログラムのクラス構造を比べると、メソッドの数が格段に増えていることがわかる。これは Toriaezer が内部に保持する動詞の実装テンプレートに、ひとつの動詞に対して複数のメソッドを生成するものが含まれているためである。

例えば Toriaezer の現在の仕様では「設定する」という動詞の実装テンプレートは、その所属するサービス指令文の目的を表す名詞を属性として実装し、それに対して値あるいは参照を設定するメソッドを実装するとともに、その属性の値あるいは参照を取得するメソッドも合わせて実装するようになっている。その理由として、「設定する」のテンプレートが、「設定する」必要性の背後には、「取得する」という目的があるからであるという思想を反映しているからである。

そのため、要求定義ファイルにおける「『診療点数』を『設定する』」というサービス指令に対し、setDiagnosisPoint(diagnosisPoint : int)というメソッドと、設定された診療点数を取得するための getDiagnosisPoint() : int というメソッドが実装されている。

4.5.Main プログラムの作成

筆者は生成されたソースコードを利用して外来患者請求システムを WEB アプリケーションシステムとして実装した。

この際、WEB アプリケーションシステムを開発する際に利用される一般的なアーキテクチャである MVC モデル 2 に基づいて開発を行い、Toriaezer が生成したソースコードを MVC モデル 2 におけるモデルコンポーネントとして位置づけた。MVC モデル 2 とは、オブジェクト指向に基づき GUI を備えたシステムを作成する際によく利用される、MVC アーキテクチャパターンを WEB アプリケーション用にカスタマイズしたモデルであるが、その構成および振る舞いは MVC アーキテクチャパターンとほぼ同じである。すなわち、コントロールコンポーネントが外部の入力をモデルコンポーネントに伝播しモデルコンポーネントの状態が変更された旨をビューコンポーネントに通知する。そしてビューコンポーネントがその通知を受けてモデルから最新の情報を取得し、出力するという仕組みである。

モデルコンポーネントが完成している状態で、それに依存したコントロールコンポーネントおよびビューコンポーネントを作成することは容易であった。

モデルコンポーネントが適切であるという前提のもとに開発を行えることも効果的で、結果として、以前同じシステムを作成した際にかかった時間の約 2 分の 1 の所要時間で作成を終えることができた。

5.Toriaezer の評価

5.1.評価方針

Toriaezer の評価には、Toriaezer を利用して外来患者請求システムを作成した際の例を評価軸と設定する。そして、以下の 2 点について評価を行った。

① 開発効率向上の度合い
② 生成されたコアプログラムの品質

5.2.開発効率改善の度合い

外来患者請求システムの作成において Toriaezer を利用した結果、以前作成したときに比べて明らかにその開発効率は向上した。
その要因として、要求分析とクラス設計がほぼ同時に行えたということが大きい。ユーザ要求をサービス文として、それを実現するためのクラス設計をサービス文に追従するサービス指令文の階層構造として表現することは、動作可能なシステムを素早く作るうえで無駄が生じにくく、非常に効率的であった。すなわち、サービス文は要求を漏れなく表していることが保証されれば、その粒度を厳密に管理する必要はない。そしてサービス指令文に関しては、単純な記法のもと、サービス文に記述された要求を満たすために必要な操作を、階層構造のもとに記述すればよい。

また、生成されたコアプログラムのソースコードをもとに Main プログラムを記述することは非常に容易であった。外来患者請求システム作成の際には、生成されたソースコードを既存の CASE ツールを利用してリバースエンジニアリングを行うことでクラス図を得て、それを参照することでそのソースコードを

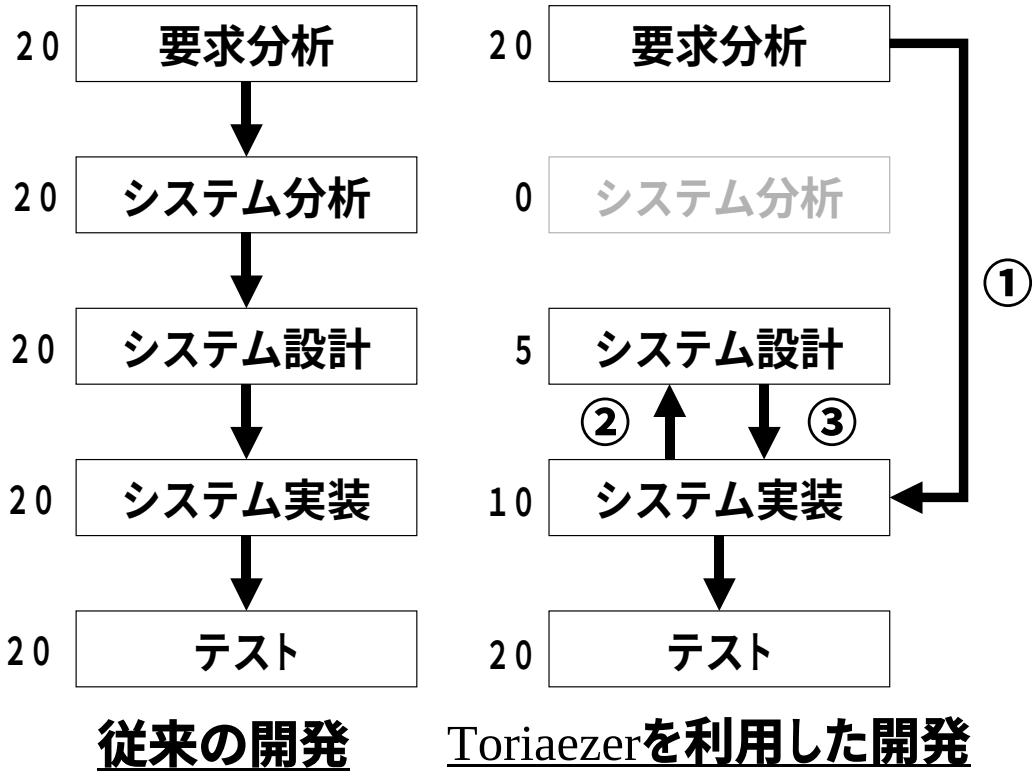


図 34　従来の開発の効率と Toriaezer を利用した開発の効率の比較

利用するプログラムの開発を行った。

その際には、WEB アプリケーションのフレームワークに基づいて MVC モデル 2 におけるコントロールコンポーネントおよびビューコンポーネントの開発を行ったため、大した設計を行う必要もなく、簡単に開発を行うことができた。

以上から、図 34 に示すよう結論を導くことができる。

あるシステムを作成する際の、従来の開発における５つの開発ステップ（要求分析、システム分析、システム設計、システム実装、テスト）に、それぞれ２０の労力で合計１００の労力が必要であるとする。これに対し同じシステムを作る際に Toriaezer を利用した場合では、約半分の労力で済む。

その理由として、要求分析は同程度の労力を要するが、Toriaezer を利用した開発ではシステム分析の過程は要求分析の過程に組み込まれているため、システム分析にかかる労力は０である。そして Toriaezer は要求定義からコアプログラムのソースコードを生成するため、コアプログラムに関してシステムの設計を行う必要はない。Main プログラムについて設計を行う必要はあるが、これは上述のとおり困難なことではないため、５の労力で行うことができる。システ

ム実装に関しても、Main プログラムの実装のみでよいが、システムによってはある程度作業が発生する場合があるので、１０の労力がかかるとする。テストに関しては Toriaezer のサポート対象ではないので従来の開発ステップと同じく２０の労力がかかる。

このことから、従来の開発では１００の労力がかかるところを、Toriaezer を利用すると５５の労力で済むということが分かる。Toriaezer はこのように、動作可能なシステムを作成する過程を大幅に効率化する。

5.3.生成されたソースコードの品質

Toriaezer では、要求定義と同じタイミングでクラス設計がなされる。そこには、要求定義とクラス設計の関係を出来る限り密接にしようという意図がある。すなわち、クラス設計は要求を達成する目的でなされるべきであり、さらにその要求を達成する必要最小限の構造を持つように行われなければならない。Toriaezer における、サービス文に対してサービス指令の階層構造を記述することにより要求定義およびクラス設計を行うアプローチは、生成されるソースコードが要求を網羅し、かつ最もシンプルな構造を持つものであることを最大限保証するものである。

外来患者請求システムが生成されたコアプログラムのソースコードを利用して簡単に作成できたのも、この理由によるところが大きい。すなわち、Toriaezer が生成したコアプログラムのソースコードは要求を網羅していることがある程度保証されているため、開発中にその修正を行う必要性はほとんどないという安心感が、開発を素早く行うことができた要因となった。

また Toriaezer は、生成したソースコードは開発者に利用されるという前提に立ち、そのフォーマットにも気を配っている。すなわち、可読性を最大限高め、開発者が快適に利用できるようなソースコードを提供している。

以下は Toriaezer が生成した、外来患者請求システムの「領収書」クラスの実装コードである。

```
import java.io.*;
import java.util.*;

/**
 * Receipt
 *
 * @author Toriaezer
 */
public class Receipt implements Serializable {

        private int diagnosisPoint; //診察点数
        private int medicationPoint; //投薬点数
        private int patientId; //患者 ID

        ～ 省略 ～

        private List nonMedicalCharges = new ArrayList(); //診療外請求リスト

        /**
         * 診察点数を設定します。
         *
         * @param diagnosisPoint 診察点数
         */
        public void setDiagnosisPoint(int diagnosisPoint) {
                this.diagnosisPoint = diagnosisPoint;
        }

        /**
         * 診察点数を設定します。
         *
         * @return 診察点数
         */
        public int getDiagnosisPoint() {
                return this.diagnosisPoint;
        }

        ～ 省略 ～

        /**
         * 診療外請求を追加します。
         *
         * @param nonMedicalCharge 診療外請求
         */
        public void addNonMedicalCharge(NonMedicalCharge nonMedicalCharge) {
                this.nonMedicalCharges.add(nonMedicalCharge);
        }

        /**
         * 診療外請求リストを取得します。
         *
         * @return 診療外請求リスト
         */
        public List getNonMedicalCharges() {
                return new ArrayList(this.nonMedicalCharges);
        }
}
```

5.4.今後の課題

現状において、Toriaezer には以下の 2 点の課題が存在する。

① 動詞の実装テンプレートを編集可能にする
② 生成されるソースコードの整形フォーマットを編集可能にする

①は現状において非常に重要であり、本質的な課題である。これが実現することにより、Toriaezer により全システムのソースコードを生成することが可能になる。また、ソースコードの使用言語に関しても、Java 言語に限定されることはなくなり、開発者が目的に応じた言語のソースコードを生成できるようになる。

但しこれは、非常に難しい。動詞のテンプレートに定義すべき内容は非常に多くかつ複雑であるため、仮に編集可能にしたとしても、その編集作業が非常に困難になってしまう可能性がある。従って、この課題の解決には入念な検討が必要である。

②について Toriaezer は、それが生成するソースコードを開発者が利用することを前提に開発された。従って、Toriaezer が生成するソースコードは開発者にとって最大限分かりやすいものでなくてはならない。

現状において、Toriaezer が生成するソースコードの整形フォーマットは、Toriaezer が内部に保持している。この分かりやすさには最大限留意しているが、開発者が必要に応じてカスタマイズできることが望ましい。これは厳密なソースコード規約のもとに開発を行わなければならないプロジェクトにおいて Toriaezer を利用する際に問題となる。そのようなときに、生成されたソースコードを、そのプロジェクトに合わせた規約に適合するように修正する手間を開発者に負わせることは避けなければならない。

6.まとめ

ユーザから反復的に要求を収集するにあたり、ユーザとの議論を効果的なものにするためには、議論のたたき台を用意することが必要不可欠である。そして、そのたたき台は、ユーザの要求を網羅した、最もシンプルな構造を持つ動作可能なシステムでなければならない。Toriaezer は、オブジェクト指向システ

ムの開発における一般的な問題である、要求定義とクラス設計の間のギャップを解消し、実装を半自動化することにより、このようなシステム作成にかかる工期を、従来の約半分に短縮する。

第4章 まとめ

以上、現場ユーザと協調した開発アプローチ OCD および、開発支援ツール Toriaezer について述べてきた。

OCD は、効率とリスク削減を追い求め、ユーザと開発者の距離が離れていく傾向にある今日のソフトウェア開発事情とは逆行し、ソフトウェア開発のあるべき姿を主張するものである。すなわち、ソフトウェアとは本来ユーザが自分の仕事を快適に行うための手段となるべきものであり、その開発はユーザと開発者が密に協調することにより行われるべきである。

そして Toriaezer は、このようなアプローチによる開発を協力に支援するツールである。すなわち、ユーザから反復的に要求を収集し、その中で効果的な議論を行うには、動作可能なシステムをたたき台として議論する必要がある。Toriaezer はこの動作可能なシステムを素早く作成することを可能にし、ユーザと密に協調した開発を強力に後押しする。

筆者は、日本のソフトウェア業界が現状を見直し、ユーザと開発者が協調した開発に力を入れる方向で動き出すことを願う。そしてその際には、OCD と Toriaezer が有効に利用されることだろう。OCD に基づく開発はそのアプローチおよびアウトプットともにユーザを満足させ、Toriaezer は開発者とユーザの距離を近く保つ。このような開発が十分一般化すれば、ソフトウェアを取り巻く現状は大きく改善されることになるであろう。

謝辞

最初に、学部時代から 3 年間ご指導いただいた大岩元教授に深く感謝いたします。その中で数々の貴重な知識、考え方を身に付けさせていただきました。また、先生に小原病院での開発を推進していただいた上に、様々な形でご支援いただき、この研究を迷うことなく続けることができました。本当にありがとうございました。

SFC 研究所の中鉢欣秀さんには、enTrance プロジェクトのころからお世話になり、いろいろなこと学ばせていただきました。本研究についても貴重な意見を数多くいただいた上に、小原病院での取り組みに対しても多大なご支援をいただきました。本当にありがとうございました。

そして、有限会社小原メディカルサービスの小原さん、りーさん、中谷さん、多田さんに深く感謝いたします。常に私の研究に対して協力的でいてくださり、いつも親身にいろいろな相談に乗っていただきました。この研究を行うことが出来たのは皆さんのおかげです。本当にありがとうございました。そして、開発に協力していただいた小原病院のみなさん、特に、忙しい中システム開発に積極的に参加していただき、たくさんの有意義なご意見を下さった医事課の方々に深く感謝いたします。

久しぶりに研究室に訪れると、いつも暖かく迎えてくださった CreW プロジェクトのみなさんに深く感謝します。特に、論文作成の追い込み作業を徹夜で手伝ってくれた岡田さん、青山君、杉浦君、明石君、本当にありがとう。おかげさまで救われました。

最後に、いつも体の心配をしてくれて、研究を何も言わずに見守ってくれた両親に感謝します。

参考文献

[1] Hugh Beyer, Karen Holtzblatt, “Contextual Design :
A Customer-Centered Approach to Systems Designs”, Morgan Kaufmann, 1997

[2] Desmond Francis, D'Souza, Alan Cameron Wills, “Objects, Components, and Frameworks with UML : The Catalysis(SM) Approach”, Addison-Wesley Pub Co, 1998

[3] 電子情報通信学会 編, 岩間 一雄, “オートマトン・言語と計算理論”, コロナ社, 2003

[4] 浅海 智晴, ”Relaxer”, ピアソン・エデュケーション, 2001

[5] 結城 浩, “Java 言語で学ぶデザインパターン入門”, ソフトバンクパブリッシング, 2001

[6] 有澤 誠, “ソフトウェア工学”, 岩波書店, 1988

[7] Magnus Penker 他, 東 秀明 他訳, 鞍田 友美 監, “UML によるビジネスモデリング, ソフトバンクパブリッシング”, 2002

[8] Eric Gamma, 本位田 真一 訳, 吉田 和樹 訳, “オブジェクト指向における再利用のためのデザインパターン[改訂版] ”, ソフトバンクパブリッシング, 1999

[9] A・ゴールド, 松葉 素子 訳, “Python で学ぶプログラム作法”, ピアソン・エデュケーション, 2001

[10] C・マーシャル, 児玉 公信 監訳, “企業情報システムの一般モデル”, ピアソン・エデュケーション, 2001

[11] アレグザンダー,C.（クリストファー）, 西川 幸治 監訳, 宮本 雅明 監訳, “オレゴン大学の実験”, 鹿島出版会, 1977

[12] P･クルーシュテン, 日本ラショナルソフトウェア, 藤井 拓 監訳, “ラショナル統一プロセス入門[第 2 版] ”, ピアソン・エデュケーション, 2001

[13] アンドリュー・ハント, ディビッド・トーマス, 村上 雅章 訳, “達人プログラマー”, ピアソン・エデュケーション, 2000

[14] ペルディタ・スティーブンス, ロブ・プーリー, 児玉 公信 監訳, “オブジェクト指向とコンポーネントによるソフトウェア工学”, ピアソン・エデュケーション, 2000

[15] ウイルソン,B.（ブライアン）, 根来 龍之 監訳, “システム仕様の分析学”, 共立出版, 1996

[16] ゲリ・シュナイダー 他, 羽生田 栄一 監訳, オージス総研 訳, “ユースケースの適用：実践ガイド”, ピアソン・エデュケーション, 2000

[17] D・ローゼンバーグ 他, テクノロジックアート 訳, 長瀬 嘉秀 監訳, “ユースケース入門”, ピアソン・エデュケーション, 2001

[18] ファウラー,M.（マーチン）, 堀内一 監訳, 児玉公信 他訳, “アナリシスパターン”, ピアソン・エデュケーション（星雲社）, 1998

[19] デビッド・ベリン 他, 今野 睦 他訳, “実践ＣＲＣカード”, ピアソン・エデュケーション, 2002

[20] アリスター・コーバーン, Alistair Cockburn, 株式会社テクノロジックアート 訳, “アジャイルソフトウェア開発”, ピアソン・エデュケーション 2002

[21] プラクティススコット・W・アンブラー, 株式会社オージス総研 訳, “アジャイルモデリング―XP と統一プロセスを補完する”, 翔泳社, 2003

[22] マーチン ファウラー, ケンドール スコット, Martin Fowler 原著, Kendall Scott 原著, 羽生田 栄一 翻訳, “UML モデリングのエッセンス―標準オブジェクトモデリング言語入門”, 翔泳社, 2000

[23]ケント ベック, Kent Beck 原著, 長瀬 嘉秀 翻訳, 飯塚 麻理香 翻訳, 永田 渉　翻訳, “XP エクストリーム・プログラミング入門―ソフトウェア開発の究極の手法”, ピアソン・エデュケーション, 2000