OGIS-RI

OGIS
Scalable Agile Method

オージス総研
アジャイル白書

2013 年 5 月

技術部　アジャイル開発センター

藤井　拓

掲載事例:

・アジャイル実践事例～一括請負受託開発への適用～
・アジャイルソフトウェア開発におけるモデリングの有効性

序文

2001 年に複数のアジャイル開発手法の主唱者が集まり、アジャイル開発が重視する共通の価値と原則を定めてから、はや 11 年間の歳月が流れました。その間に、アジャイル開発は欧米では広く普及し、開発手法全体の 3 割以上を占めており、形式がしっかり定まった開発手法としては最も大きな普及率を誇るようになっています。その一方、日本を見た場合、アジャイル開発手法の普及率は 2 割にも満たないというのが現状です。

アジャイル開発は、「ビジネス競争に勝つためのソフトウェアを早く作ること」を目的としています。そのようなビジネス競争に勝つためのソフトウェアとは、以下のようなものです。

A) 機能の良し悪しが業績を左右するようなもの
B) 顧客に対するサービスの良し悪しが業績を左右するもの

A)の代表例はパッケージソフトウェアの製品やパブリッククラウドで提供されるソフトウェアです。B)のサービスとは「業務とそれを支援するソフトウェアの組み合わせで提供されるもの」の意味です。例えば、代表的なものとしてユニクロのような製造小売業を実現するための業務とソフトウェアを考えることができます。

このようなソフトウェアを開発する際には以下のような課題に直面します。

- 市場が求めている機能やサービスの予測が困難
- 競争力のあるサービスを実現するための業務やサービスの形が分からない

これらは、製品であれ、サービスであれ、「競争に勝つために必要なソフトウェアに対するニーズが不確定」だということを意味しています。これらの不確定なニーズに対処しながら、限られた予算で迅速にビジネス競争において優位に立つためのソフトウェアを開発するというのが今日のソフトウェア開発の大きな課題になっています。

不確定なニーズに対処するためには、不確定なニーズに対する仮説を立ててその仮説が正しいかどうかを検証し、その結果によって開発の方向性を変えていく必要があります。これは、日本でよく知られている PDCA (Plan-Do-Check-Act) サイクルを回して開発を進めていくということです。PDCA サイクルをソフトウェア開発で実践するためには、開発期間を「反復」と呼ぶ一定の周期に分割し、以下のことを行います。

- 反復で開発する機能を計画する
- 機能を作り上げる

- 作り上げた機能を評価する
- 評価結果に基づいて開発すべき機能の一覧を見直す

アジャイル開発は、このように反復を通じて機能を追加することで開発を進める方法です。反復で開発する機能は、開発依頼者のニーズと優先順位に基づいて決めます。また、反復の期間は数週間から 1 ケ月に設定されることが多く、短い周期でフィードバックをかけて開発が進行します。

つまり、不確定な開発依頼者のニーズ（＝仮説）は、反復で動くソフトウェアとして実現され、デモや試用により有効性を早く確認できるというのがアジャイル開発の大きな利点です。さらに、仮説が正しくなかった場合やビジネス状況の変化に対応して柔軟に開発計画を変更できるというのもアジャイル開発の利点です。

このようなアジャイル開発を進めるための価値と原則を定めたものがアジャイル宣言です。アジャイル宣言では、アジャイル開発の利点を活かすために以下の 3 点が重要であると述べています。

- 開発依頼者と開発者との密な連携
- 開発チームのチームワークと開発者の自律的な判断
- 継続的な開発方法の改善

本白書は、アジャイル開発の概説を提供するとともに、日本でのアジャイル開発を適用する上での課題に対する解決策を提示し、アジャイル開発という概念が米国で生まれるきっかけを与えた日本のメーカーの組織的知識創造に再び取り組むことの重要性を説明することを目的としています。本書は、4 部と 2 つの事例で構成されており、各部と事例の内容は以下のとおりです。

第 1 部：ソフトウェア開発のプロセスの変遷の過程でアジャイル開発が生まれた背景を説明するとともに、アジャイル開発の土台となるアジャイル宣言とアジャイル開発手法の具体例として「スクラム」を説明する。さらに、ユーザ企業と IT ベンダーのこれまでの関係がアジャイル開発の導入の阻害要因となっていることを説明する。

第 2 部：企業の IT 関連作業を全体最適化するためのプロセスフレームワークである EUP (Enterprise Unified Process)の概要を紹介する。また、ユーザ企業と IT ベンダーとの間により良い関係を築き、アジャイル開発を取り入れるための手段として測定の活用を提案している。さらに仕様変更も含めた開発規模を測定するための方法として機能規模測定手法 COSMIC 法の概要を説明する。

第 3 部：長期的な計画（見通し）、契約、ドキュメントというような日本でアジャイル開発を取り入れる際の代表的な阻害要因を挙げ、それらを解決する方法として OGIS Scalable Agile Method（以降 OSAM と略す）を提案する。OSAM は、開発手法と測定技術で構成される。OSAM の開発手法は、スクラムに含まれない開発の技術的な進め方をアジャイル UP という手法で補う。また、OSAM の測定技術では見積もりや契約のためにソフトウェアの規模や変更の規模を測定する手法である COSMIC 法を用いることを提案している。

第 4 部：OSAM 誕生の原点となった反復開発やアジャイル開発の事例を通じてアジャイル開発を成功させるために必要な要素について説明するとともに、1 つのプロジェクトを超えたエンタープライズアーキテクチャや百年アーキテクチャのような IT ガバナンスに対する取り組みとアジャイル開発の関係を説明する。また、アジャイル開発手法「スクラム」が誕生するきっかけとなった「組織的知識創造」の概念を紹介し、そのような「組織的知識創造」を実践する具体的な手段として「スクラム」を説明する。「組織的知識創造」は日本のメーカーが競争力のある新製品開発を生み出す過程を野中先生等が分析した結果として見出されたものである。さらに、「組織的知識創造」を実践するために企業風土を変えていく必要がある点を説明する。

事例その 1 ：OSAM の源流となった同僚の入江さんの開発事例を紹介する。本事例では、統一プロセスとスクラムを融合した開発手法、及び機能規模の測定値によるスコープ管理を行っている。本事例により、OSAM を適用した開発の具体的なイメージを得ることができる。

事例その 2：2004 年に実施したモデリングを組み入れたアジャイル開発事例を紹介する。本事例では、CRC カードという手軽で協調的なモデリング手段と UML とを組み合わせて 2 週間の反復サイクルの中ですばやくモデリングを行いながら開発を行っている。本事例により、OSAM におけるモデリングの実践方法のより具体的なイメージを得ることができる。

最後に、長年に渡るアジャイル開発の理解者であり、本白書の第 1 部の執筆のきっかけを与えてくれたオージス総研技術部の宗平部長、アジャイル開発センターを設立し、アジャイル開発を推進しようという判断を下した平山社長、過去 15 年以上に渡り反復開発やアジャイル開発をこれまで実践してきた入江さんを始めとする多くの同僚、また本白書の執筆を勧めてくれた営業企画部の水間さん、本白書の執筆過程で様々なフィードバックを下さったアジャイル開発センターの同僚、アジャイルモデリングやアジャイル開発の取り組みを温かく見守り、応援をしてくれた多くの同僚にこの場を借りて感謝します。また、アジャイルモデリングとアジャイル UP の考案者であり、アジャイル UP の日本語訳と引用を許可して下さった Scott Ambler さんにこの場を借りて感謝致します。

本白書が日本の企業でのアジャイル開発の活用に少しでも貢献できれば幸いです。

2013 年 5 月

藤井　拓

目次

事例1　アジャイル実践事例　～一括請負受託開発への適用～

事例2　アジャイルソフトウェア開発におけるモデリングの有効性

第1部：アジャイル開発

第１部：アジャイル開発

1. アジャイル開発が生まれた背景

ウォーターフォール型開発は、図 1 に示すように要求を開発初期段階に確定し、確定した要求に基づいて設計、実装、統合、テスティングを順次的に行う形の開発方法です。ウォーターフォール型開発は、日本のソフトウェア開発プロジェクトの半分以上で採用されており、未だ大勢を占めています。

ウォーターフォール型開発には以下のような長所があります。

A) 開発依頼者側の計画、契約の負荷（リスク）が小さい
- 開発初期段階に確定した要求に基づいて計画や契約を締結すれば、要求の実現の責任を IT ベンダーに転嫁できる

B) 分業が可能
- 開発を複数フェーズに分けて、フェーズ間で文書による引き継ぎを行うことでフェーズ毎に異なる開発者（役割）による分業が可能になる

この「分業が可能」という長所を利用して、日本ではソフトウェア開発において複数の階層にも及ぶアウトソースという形態が一般化しました。また、フェーズ間での引き継ぎのために作成される文書をレビューすることで品質を管理するというアプローチも広まりました。

ウォーターフォール型開発の短所としては、以下のような点を挙げることができます。

イ）要求確定後の仕様変更が困難
- 要求確定後に技術やビジネス状況の変化やビジネスニーズの理解が進んで要求を変更したいと望んでも、対応は次バージョン以降になる

ロ）開発終盤に開発上の問題点が顕在化して遅れる
- 統合段階やテスト段階でアーキテクチャ上の問題やサブシステム間のインタフェースの不整合などの問題が顕在化して納期が守れない

「要求確定後の仕様変更が困難」という点に付随する問題として、「文書で書き表した要求では要求の有効性が判断できない」というものがあります。つまり、開発依頼者が明確に要求を述べられる場合を除いては「要求」が IT ベンダー主導で決められていき、その結果として「真のニーズとはかけ離れたソフトウェアが納品される」ような事態に陥る危険性があります。つまり、ウォーターフォール型開発には「ビジネス価値が低いソフトウェ

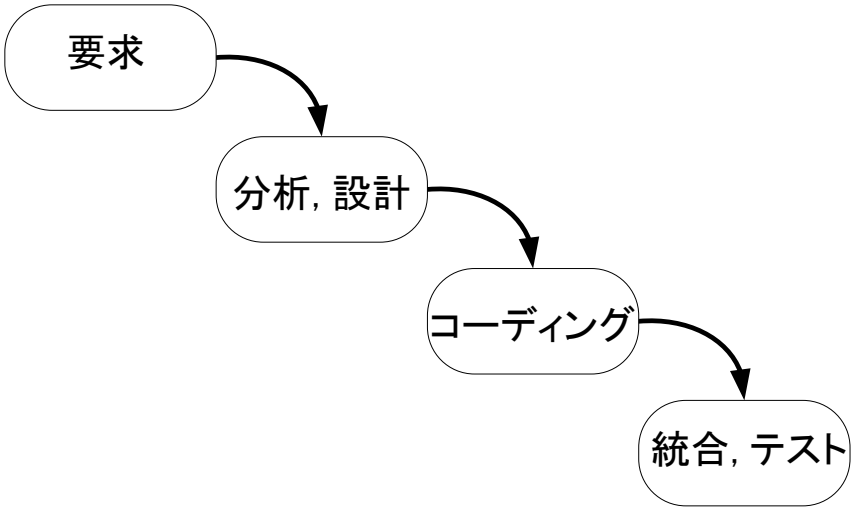

図 1　ウォーターフォール型開発

アが開発されてしまう」というリスクがあります。また、「開発終盤に開発上の問題点が顕在化して遅れる」ことで結果的には「納期が守られないことで、そのソフトウェアの運用開始が遅れてビジネス上の損害が生じる」ような状況も現実には発生します。つまり、実際には「計画や契約に伴うリスクが高い」こともありえるのです。

これらの長所と短所を考えると、開発依頼者にとってウォーターフォール型開発の長所が発揮できるのは以下のような場合だと考えられます。

- 要求が開発初期に確定できる
- アーキテクチャやサブシステム間のインタフェースの不整合などのリスクが低い

欧米ではウォーターフォール型開発を適用するプロジェクト割合が半分以下に低下し、アジャイル開発を適用する割合が増えているということも報告されています。欧米でウォーターフォール型開発が減少している原因として以下のようなものが考えられます。

- ビジネスや技術の変化にすばやく対応するための開発が求められている
- ユーザ企業がある程度開発要員を抱えている

Google や SalesForce のようにクラウドでビジネスを展開する企業も、Apple のようにハードウェアデバイスやソフトウェアでビジネスを展開する企業も、Amazon のような販売でビジネスを展開する企業も競争に勝つために「ビジネスや技術の変化へのすばやい対応」を重視しています。「ビジネスや技術の変化への対応」というのは以下のことを意味します。

- ビジネス形態（モデル）や市場の変化に対応する
- 新たな技術（IT 機器）を組み込んだり、移行したりする必要がある

これらの「ビジネスや技術の変化」を事前に予測するのは不可能に近いと言えます。しかしながら、変化に対応した製品やサービスを早く市場に投入しないと高い利益を得ることができません。このような状況に対応するためは、「ビジネスや技術の変化」に柔軟かつスピーディーに対応して開発することが求められます。その結果として、ウォーターフォール型開発からアジャイル開発への移行が進んでいると考えられます。

また、同時に欧米では自社で開発要員を抱えている場合が多いために日本ほど「開発委託側の計画や契約の負荷（リスク）」に対処する必要はありません。つまり、計画や契約という敷居が低いこともウォーターフォール型開発からアジャイル開発への移行が欧米で進んでいる理由になっていると考えられます。

2. アジャイル宣言

90 年代の終盤に、顧客と開発者との密な協力に基づいて、顧客に役立つソフトウェアをすばやく開発する手法として XP(eXtreme Programming)[1]やスクラム[2]などの複数のアジャイル開発手法が登場しました。これらのアジャイル開発手法の提案者達は 2001 年に集まり、アジャイル開発が共有する価値や原則をアジャイル宣言という形でまとめました。アジャイル宣言の価値は、以下の 4 つの項目で構成されます。

A) プロセスやツールよりも個人や相互作用
B) 包括的なドキュメントよりも動くソフトウェア
C) 契約上の駆け引きよりも顧客とのコラボレーション
D) 計画を硬直的に守るよりも変化に対応する

これらの各行で下線を引いてある右側の部分を左側の部分より重視するということが価値として宣言されたのです。また、アジャイル宣言には、さらに以下のような原則が付随しており、この宣言が推奨している開発の進め方をより具体的に説明しています。

①. ソフトウェアの早期、継続的な納品により、顧客の満足を達することが最優先である。
②. 開発の終盤であろうとも、要求内容の変更を歓迎する。アジャイルなプロセスは、顧客の競争上の優位性のため、変化を制する
③. 数週間から数ヶ月間のサイクルで動作するソフトウェアを納品する。サイクルは短い方が良い
④. ビジネス側の人間と開発者がプロジェクトを通じて日々協力をしなければならない

⑤. 志の高い開発者を中心にプロジェクトを編成する.必要な環境や支援を与え､任務をやり遂げることを信じなさい
⑥. 開発チームの内外でもっとも効率的で効果的な情報伝達を行う手段は､顔をつき合わせて話し合うことである
⑦. 動作するソフトウェアが主たる進捗の確認手段である
⑧. アジャイルなソフトウェア開発は､持続的な開発を促す｡開発資金の提供者､開発者､ユーザは、必ず一定のペースを守れるべきである
⑨. 技術力と良い設計に絶えず気を配ることで、機敏さは向上する
⑩. 不必要なことを行わない技能である簡潔さは、本質的である
⑪. 自己組織化されたチームから最善のアーキテクチャ、要求、設計が生まれる
⑫. 定期的に、チームはもっと効果的になる道を考え、開発の進め方を調和させ、調整する

アジャイル宣言の原則の内容を大別すると、以下の 5 点にまとめることができます。

A) チームのあり方: ⑤, ⑥, ⑧, ⑪, ⑫
B) 個人の心構え: ⑨, ⑩
C) 動くソフトウェア: ①, ③, ⑦
D) 顧客とのコラボレーション: ④
E) 変化への対応: ②

A)と B)は、アジャイル開発の価値である"個人や相互作用"を個人に関する原則とチームに関する原則に分解したものです。アジャイル宣言の①、③、⑦ の原則は反復的な開発アプローチに関するものです。そのため、"動くソフトウェア"の価値と対応すると考え、C) に分類しました｡さらに､D)と E)は顧客のニーズの変化に即応した開発を行うことであり、顧客本位のソフトウェア開発を行うという方向性を示すものです。

これらの原則から､アジャイル開発にとって"顧客のニーズの変化に即応した開発"が大きなテーマであり、A)と B)と C) がその実現手段だといえます。すなわち、顧客のニーズの変化に即応するためには、一定の作業内容や作業手順に従うだけでは不十分であり、機動性に富んだチームと個人の自律性が必要になるということです。

3. アジャイル開発手法

3.1. アジャイル開発手法の分類

アジャイル開発手法を個別に見た場合、以下のようにプロジェクト管理に特化したものとそうではないものの 2 種類に分類できます。

A) プロジェクト管理中心
- スクラム
- 適応型ソフトウェア開発

B) より包括的な開発手法
- XP
- Crystal Clear
- アジャイル UP

A)の 2 つの開発手法はプロジェクト管理的な作業やチーム運営などに特化したものです。前節で説明したようにアジャイルなソフトウェア開発の原則の多くはプロジェクト管理に関するものであり、スクラムや適応型ソフトウェア開発[3]はそれらの原則のより具体的な適用イメージを理解するために参考になります。その反面、 A)の手法を適用する場合には、プロジェクト管理以外の開発作業について B)のプラクティスを取り入れたり、独自に定義する必要があります。

B)の 3 つの開発手法は、プロジェクト管理以外のプラクティスを含むようなより包括的な開発手法です。XP (eXtreme Programming)は、Beck らにより提案された開発手法であり、開発者 2 名がペアでプログラミングを行うペアプログラミングやプログラミングよりも先にテストを考えるというテストファーストなどの原則やプラクティスで構成されています。Crystal Clear [4]は、Cockburn により提案された開発手法であり、2004 年 に解説書が刊行された比較的新しいアジャイル開発手法です。Cockburn は、プロジェクトの規模などに応じて成果物や作業などの構成を変えるべきだという考え方を提案しており、Crystal は規模に応じた複数の手法からなる開発手法ファミリーになっています。Crystal Clear は、その Crystal ファミリーの最軽量な開発手法です。Crystal Clear は、XP 以外の包括的な開発手法の選択肢として注目されますが現時点では XP よりも低い普及率に留まっています。また、アジャイル UP という開発手法は UP (Unified Process)[5], [6]をアジャイル化したものとして Ambler 氏により提案されたものです。アジャイル UP では、UP のライフサイクルモデルと、軽量にモデリングを行うアジャイルモデリング(AM: Agile Modeling)[7], [8]、テスト駆動開発(TDD: Test Driven Development)[9] を組み合わせることでプロジェクトのガバナンスと機敏さを両立させることを目指しています。

アジャイル開発手法のように具体的な開発方法を示すものではありませんが、「トヨタ生産方式」と「トヨタ製品開発システム」の両方の考え方をソフトウェア開発に応用するために考案されたリーン開発[10]というものも提案されています。

3.2. スクラム

スクラムの価値

スクラムは、Ken Schwaber と Jeff Sutherland、 Mike Beedle によって考案されたア

ジャイル開発手法です。 スクラムという開発方法論の名称は，ラグビーのスクラムにちなんで名づけられました。 スクラムは、野中先生達が 80 年代に日本の製造メーカーの新製品開発において欧米のメーカーを凌駕した要因の研究をまとめた「知識創造企業」[11]などに触発され、Schwaber らがいくつかの失敗プロジェクトを立て直す経験を通じて生み出されたものです。

スクラムは、他のアジャイル開発手法と同様に動くソフトウェアを順次作り、そのソフトウェアを発展させながら開発を進める反復的な開発アプローチに基づいています。アジャイル開発手法は一般的に開発チームで共有すべき価値や作業の指針となる原則やプラクティスで開発の進め方を定めていますが、スクラムが定めている価値は以下の 5 点です。

- 確約（Commitment）
  - 約束したことを確実に実現すること
- 専念 （ Focus）
  - 確約したことの実現に専念すること
- 隠さない（Openness）
  - たとえ自分にとって不利なことでも包み隠さないこと
- 尊敬（Respect）
  - 自分と異なる人に敬意を払うこと
- 勇気（Courage）
  - 確約し、隠さず、敬意を払うために勇気を持つこと

スクラムでは、このような価値以外にも開発の進め方や開発チームにおける役割についても言及していますが、言及している範囲はプロジェクト管理作業に偏っています。 スクラムは、プロジェクト管理的な作業に特化しているため、XP や UP など設計、実装、テストの実践形態を規定している様々な反復的な開発手法と組み合わせやすい開発手法です。

スクラムにおける開発の進め方

スクラムでは、 1 週間から 4 週間のサイクルで動くソフトウェアを作りながら開発を進めます。 図 2 は、スクラムによる開発の流れを示したものです。図中の用語の意味は、以下の通りです。

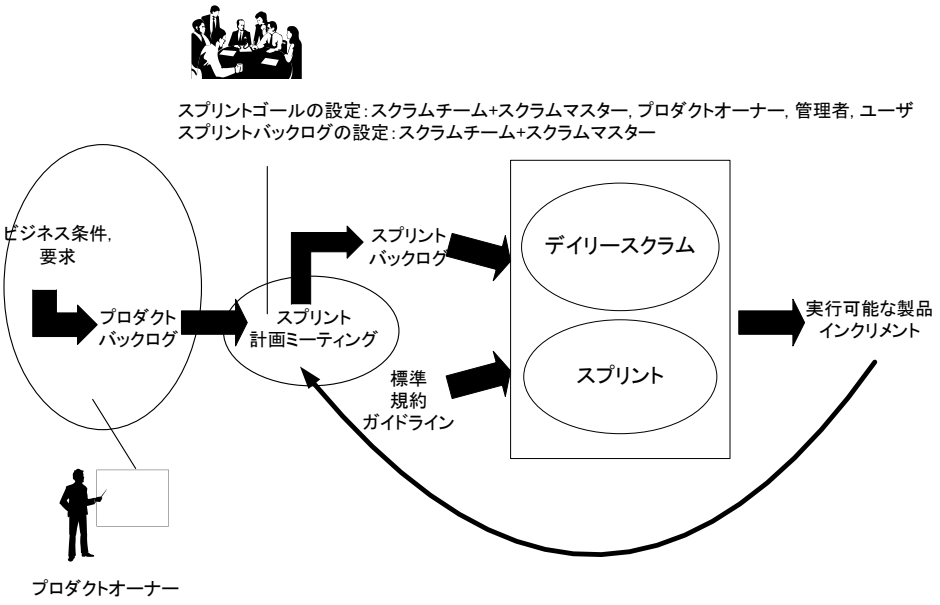

図 2 スクラムにおける開発の流れ

- プロダクトバックログ: 開発対象のソフトウェアに対する要求のバックログ
- スプリント: 1 週間から 4 週間サイクルの反復
- スプリント計画ミーティング: スプリントの開発目標（ スプリントゴール ）とスプリントバックログを設定するミーティング
- スプリントバックログ: スプリントゴールの達成に必要なタスクのリスト
- デイリースクラム: 日毎の進捗確認ミーティング
- 実行可能なプロダクトのインクリメント: スプリントの結果として作成される実行可能なソフトウェア

図 2 に示されている標準、規約、ガイドラインは、開発組織において守ることが求められている標準、規約、ガイドラインを意味します。
スクラムでは、開発は以下のようなステップで進行します。

1. ソフトウェアに要求される機能とその優先度をプロダクトバックログとして定める
2. プロダクトバックログからスプリントで実装するべき目標（ スプリントゴール ）を選択する
3. スプリントゴールをより詳細なタスクに分解したスプリントバックログを作成し，タスクの割り当てを行う
4. スプリントの間, 毎日決まった場所及び時間で開発メンバーが参加するデイリースクラムというミーティングを開催する

5. 1 回のスプリントが終了すると，スクラムレビューミーティングを開催し，作成されたソフトウェアを評価する
6. 次回のスプリントに備えてプロダクトバックログの内容と優先度の見直しを行う

図 2 のスプリント計画ミーティングは、2、 3 の 2 ステップとして実行されます。
スクラムでは、一般の開発メンバーに加えて以下の 2 つの管理的な役割が定義されています。

- プロダクトオーナー: プロダクトバックログを定義し，優先度を決める人
- スクラムマスター: プロジェクトが円滑に進むように手助けする人

スクラムマスターは、通常のプロジェクト管理者のように開発者への作業の割り当て、計画策定、進捗管理等を行うことではなく、開発を阻害する様々な障害を解決することを主任務とします。

スプリント計画ミーティングでは、スプリントゴールとスプリントバックログが決められます。 スプリントゴールは、プロダクトオーナーが中心になって顧客、管理職、開発チームとの議論により決定されるスプリントの大雑把な目標です。 例えば、スプリントゴールは"システムを構成する永続性や通信メカニズムを決定する"などと設定されます。スプリントゴールが設定された後、開発チームのメンバーが主体になってスプリントゴールを達成するために必要なタスクをリストアップしていきます。各タスクは、4-16 時間で完了できる粒度で定義されます。さらに、抽出されたタスクから開発チームのメンバーが話し合いによりスプリントで達成するタスクの割り当てを決めます。 開発チームのこのようなタスクの定義や割り当ては、顧客やプロダクトオーナーの介入なしに開発メンバーにより自律的に行われます。

このような開発チームのメンバー間の自発的な議論を通じて、メンバー間の連携が自然に形成されます。このチーム内の連携が自律的に形成される過程を Schwaber らは「自己組織化による真のチーム形成」と呼んでいます。

最終的に割り当てされたタスクの集合がスプリントの詳細目標であるスプリントバックログになります。 スプリントの途上では、スプリントバックログの消化状況がグラフ化されてプロジェクトの全体の作業進捗状況として共有されます。

デイリースクラムは、スプリントの期間中に毎日決まった場所及び時間に開催され、開発チームのメンバー全員が参加するミーティングです。 デイリースクラムでは、スクラムマスターが開発チームの各メンバーに以下の 3 点を質問します。

- 前回のデイリースクラム以降の作業内容
- 次回のデイリースクラムまでの作業予定

- 作業を進める上での障害

デイリースクラムは、チーム内のコミュニケーションを促進するとともに、チームメンバー全員がプロジェクトの現状についての認識を共有し、メンバーの連帯を深めるのに有効です。また、スクラムマスターはデイリースクラムで報告された障害を解決することを支援します。

スクラムの大事な点は、スプリントの期間中は開発チームに外乱を与えず、開発に専念できるようにすることです。そのような外乱の発生を防ぐために、デイリースクラムには開発メンバー以外の人々も参加できますが、意見や要望を述べたりすることは許されていません。

3.3. リーン開発

リーン開発は、「トヨタ生産方式」と「トヨタ製品開発システム」の両方の考え方をソフトウェア開発に応用するために Mary Poppendieck と Tom Poppendieck により発案されたものです。

リーン開発の骨格をなすのは、以下のような 7 つの原則です。

- ムダをなくす
  - 顧客に対する価値を付加しない活動を「ムダ」と捉え、そのような「ムダ」をなくすことで顧客価値を提供するまでの時間軸を短縮する
- 品質を作り込む
  - 欠陥を検出するために検査を行うのではなく、最初からコードに品質を作り込む。また、欠陥は即座に修正する
- 知識を作り出す
  - 開発対象のソフトウェアに関する知識は初期に確定するものではなく、顧客のフィードバック等を通じて開発途上で作り出すものである
- 決定を遅らせる
  - 取り返しがつかないような重要な決定を行うタイミングをなるべく遅らせることができるように複数の選択肢を用意しておく
- 速く提供する
  - 顧客の気の変わるよりも速くソフトウェアを提供する。そのためには、開発を継続的に改善し、品質を作り込める人材を育成し、顧客のニーズに対応する能力を培う必要がある
- 人を尊重する
  - 競争力のあるプロダクトを作るためには、優秀なリーダー、エキスパートエンジニアとともに詳細な計画の立案とフォローを自ら行う自律的な組織が必要である

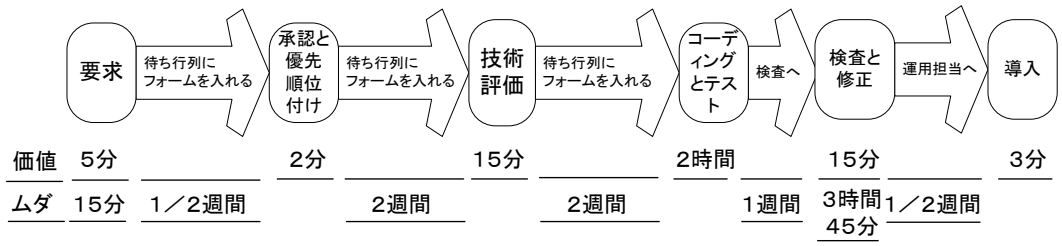

図 3 機能変更要求のバリューストリームマップの例

「リーン開発の本質」[10]から引用

- 全体を最適化する
  - リーンな組織では、顧客のニーズを満たすための注文を受けた時点から、ソフトウェアが導入され、ニーズが解決されるまでのバリューストリーム全体を最適化する

ここで「バリューストリーム」とは、顧客が注文を出してから製品のベンダーが現金を回収するまでの間に実行される一連の作業と、それに付随する原料と情報、価値の流れを表すものです。リーン開発の原則である「ムダをなくす」及び「全体を最適化する」を実行する際には、この「バリューストリーム」を図示したバリューストリームマップという図を作成します。図１は、顧客の要求した機能を追加する過程のバリューストリームマップです。このようなバリューストリームマップを作成することで、顧客に対する価値を付加する活動に要する時間（図の「価値」の行）と付加しない活動に要する時間（図の「ムダ」の行）を明らかになります。このように、バリューストリームマップを作成することで開発作業におけるムダの大きさを定量化し、改善すべき活動を特定できます。

リーン開発の原則が目指しているのは、以下の 2 点であると考えられます。

- 顧客価値を高める
- 顧客価値の納品速度を向上する

つまり、「顧客価値を高める」ために「トヨタ製品開発システム」の概念を「知識を作り出す」、「決定を遅らせる」、「人を尊重する」という原則として取り入れ、「顧客価値の納品速度を向上する」ために「トヨタ生産方式」の概念を「ムダをなくす」、「品質を作り込む」、「速く提供する」、「全体を最適化する」という原則として取り入れていると捉えることができます。

リーン開発とアジャイル開発との間には類似する点が多いと言えます。例えば、リーン開発の「顧客価値を高める」ための原則は、スクラムと共通する点が多いと考えらえます。

また、リーン開発の「顧客価値の納品速度を向上する」ための原則は XP に共通する点が多いと考えられます。その一方で、リーン開発は、具体的な開発の進め方には言及せず、「より良いソフトウェア開発の方向性を抽象度の高いレベルで示す」という点でスクラムや XP と異なります。また、「リーダーシップ」の重要性が強調されている点でもリーン開発がスクラムや XP と異なります。

「より良いソフトウェア開発の方向性を抽象度の高いレベルで示す」ことを中心に据えているために、リーン開発は開発経験のないマネジャーや経営者にとってより理解しやすいものだといえます。さらに、「リーダーシップ」の重要性が強調されている点において、リーン開発への転換におけるマネジャーや経営者の位置づけが明確になっており、マネジャーや経営者の積極的な関与を促す効果も期待できます。

4. 日本でのアジャイル手法の普及

今のところ、日本では依然としてウォーターフォール型開発が主流というのが現実であり、これはアジャイル宣言以来の 10 年間でアジャイル開発が普及した欧米とは異なる状況です。この違いが生まれた大きな原因は、「ソフトウェア開発の内製化率」の違いが影響していると考えられます。つまり、欧米では業務の遂行に必要なソフトウェアの開発を本業の一部と考え、内製する傾向がありますが、日本では本業の一部とは考えず、アウトソースする傾向があるという違いです。内製すると、ビジネスを成功させるという目的で業務側と開発者がより密に連携するというアジャイル開発を取り入れやすくなります。また、終身雇用制を前提とするか否かという雇用形態の違いも大きな影響を及ぼしていると考えられます。

日本では歴史的に業務の遂行に必要なソフトウェアの開発を本業の一部と考えない会社が多く、それが結果的に「ユーザ企業」が「IT ベンダー」に開発委託をするという産業構造を生み出しました。この産業構造は、過去 10 年間を通じて大きくは変わっていません。しかし、リーマンショック以降「ユーザ企業」の業績がかつてないほど厳しいという状況に至り、「ユーザ企業」は「IT ベンダー」のコストパフォーマンスに厳しい目を向けるようになってきています。

このような日本の状況で、アジャイル開発はその強みである以下の 2 点を切実に求める「ユーザ企業」に受け入れられていくと考えられます。

- 変化への対応
- より業務に即したソフトウェアの実現

この中で「変化」を変化量として捉えると、それは開発期間に比例すると考えられる。つまり、単位期間あたりの変化率が一定であれば開発期間が長くなればなるほど「変化」は大きくなります。つまり、開発期間が長いほど「変化への対応」が求められます。

「より業務に即したソフトウェアの実現」が難しいのは、「業務」と「システム」の両者をどのように変えたらビジネスパフォーマンスが向上するかということが予測困難だからだと考えられます。つまり、「業務」と「システム」を変えた結果の「効果」を予測することが困難だということです。そのような状況に対する論理的な対処法は、「効果」についての仮説を立ててそれを帰納的に検証しながらソフトウェア開発を行うということです。そのような帰納的な検証を可能にするという点で「アジャイル開発」は大きな魅力を持つのです。このように「業務」と「システム」を変えた結果の「効果」が予測しづらいのは、企業にとってビジネスを差別化するような戦略的な業務分野だと考えられます。

現在アジャイル開発を積極的に採用しているのは、ネットビジネスを展開している会社、ゲーム会社などです。これらの会社は変化のスピードが速く、使い勝手がよく魅力的な機能を提供するシステムを迅速に作れるかどうかでビジネス上の業績が左右される会社です。使い勝手がよく魅力的な機能を提供するシステムを迅速に作るために、短期に機能を追加し、ユーザのフィードバックを得ていきながらシステムを改良していくというアジャイル開発の特長を活かしていると考えられます。

「ユーザ企業」と「IT ベンダー」という関係の中で「アジャイル開発」が普及していく場合に、大きな課題になるのは「変化」に対応するためのコストを誰が負うかという点です。変化に対応するためのコストは以下の式のように 1 次式で表現できます。

- 開発費用　＝　{(当初の仕様）＋（変更された仕様)} / (開発生産性)} x （単価）
- 変化に対応するためのコスト　＝　{(変更された仕様) / (開発生産性)} x （単価）

ここで（変更された仕様）は開発委託者が制御するものであるのに対して、(開発生産性）と（単価）は IT ベンダー側が制御するものです。アジャイル開発を普及させるためには、この「変化に対応するコスト」を明確にし、そのコストを IT ベンダーが一方的に負うのではなく、開発委託者が受け入れることが必要になります。

一方で、IT ベンダー側には、(当初の仕様）を開発途上で変更しても（開発生産性）をなるべく落とさないことが求められます。そのためには、開発途上での変更に強くなるように開発プロセスを改善していく必要があります。

5. 参考文献

[1] ケント・ベック, XP エクストリーム・プログラミング入門―変化を受け入れる, ピアソンエデュケーション, 2005

[2] ケン・シュエイバー, マイク・ビードル, 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, 2003

[3] ジム・ハイスミス, 適応型ソフトウエア開発-変化とスピードに挑むプロジェクトマネージメント, 翔泳社, 2003

[4] Alistair Cockburn, Crystal Clear: A Human-Powered Methodology for Small Teams, Addison-Wesley, 2004
[5] イヴァー ヤコブソン, ジェームズ ランボー, グラディ ブーチ, UML による統一ソフトウェア開発プロセス―オブジェクト指向開発方法論, 翔泳社, 2000
[6] フィリップ・クルーシュテン, ラショナル統一プロセス入門 第 3 版, アスキー, 2004
[7] スコット・W・アンブラー, アジャイルモデリング―XP と統一プロセスを補完するプラクティス, 翔泳社, 2003
[8] Scott W.Ambler, オブジェクト開発の神髄-UML 2.0 を使ったアジャイルモデル駆動開発のすべて, 日経 BP 出版センター, 2005
[9] ケント・ベック, テスト駆動開発入門, ピアソンエデュケーション, 2003
[10] メアリー・ポッペンディーク, トム・ポッペンディーク, リーンソフトウエア開発~アジャイル開発を実践する 22 の方法, 日経 BP, 2004
[11] 野中郁次郎, 竹内広高, 知識創造企業, 東洋経済新報社, 1996

第2部：測定による IT 開発能力の強化

第 2 部：測定による IT 開発能力の強化

1. IT ライフサイクルとプロセス改善

1.1. IT ライフサイクルとは

第 1 部では、アジャイル開発が登場した背景と概要について説明しました。アジャイル開発は 1 つの開発プロジェクトに課せられた目的を達成するためのものです。しかし、IT 部門全体の使命を考えた場合、このような個別の開発プロジェクトを超えた以下のような取り組みも必要になります。

- システムの企画
- システムの運用/サポート
- ソフトウェア開発効率化
- 後継者の育成

これらの取り組みは、図 1 に示すようにシステム開発ライフサイクルを包含するシステムライフサイクルと IT ライフサイクルにかかわるものだと捉えることができます。このようなシステムライフサイクルと IT ライフサイクルを整理したフレームワークとしては、エンタープライズ統一プロセス(EUP：Enterprise Unified Process)[1]というものが提案されています。

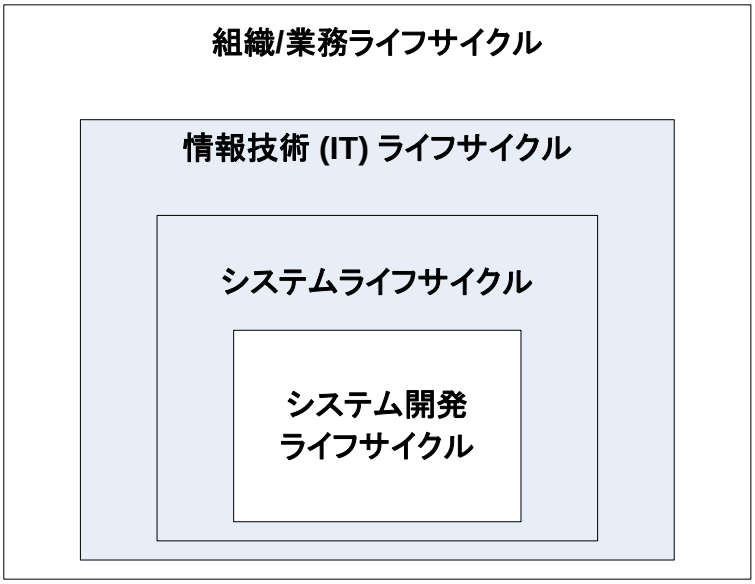

図 1 IT ライフサイクルとシステムライフサイクル

EUP は、ソフトウェア開発のプロセスフレームワークである UP [1]（Unified Process）[2], [3]を土台にして作り上げた、企業全体の IT 関連作業を網羅するプロセスフレームワークです。EUP は、ビジネスをより良く支援するために企業全体の IT 関連作業の全体像を提供することを目指しています。EUP は、企業全体の IT 関連作業を網羅するために UP を拡張したものですが､プロジェクト単位の開発手法としては UP の代わりに UP をアジャイル化したアジャイル UP を用いることも可能です。逆に、EUP を用いることで企業全体の IT 関連作業という観点でアジャイル開発も含めた IT 関連作業の全体最適化を図ることができます。

EUP は、企業全体の IT 関連作業のワークフロー、役割、成果物を UP の形式に準じて定義することで全体像を提供しています。そのために、EUP では以下のフェーズ、作業分野、作業分野群を UP に追加しています。

- 稼動フェーズ
- 引退フェーズ
- 運用及びサポート作業分野
- エンタープライズ作業分野群

ここで作業分野とは、作業を種類ごとにグループ分けしたものを意味し、作業分野群は複数の作業分野を 1 つのグループにまとめたものです。稼動フェーズと引退フェーズは、システムの開発後に行われる IT 関連作業を網羅するために UP の 4 つのフェーズに追加されたものです。運用及びサポート作業分野は、開発したシステムの運用やサポートに関する作業が定義されています。また、エンタープライズ作業分野群は複数のプロジェクトを横断するような IT 関連作業を網羅するためのものです。

稼動フェーズは、開発したシステムを運用及びサポートし、問題点報告や拡張依頼などから修正や拡張すべき内容を特定し、その開発を行い、その変更をシステムに反映していくというフェーズです。稼動フェーズでは、フェーズで行う作業の全体像や次の引退フェーズに移るためのマイルストーンが定義されています。

引退フェーズは、現在稼動しているシステムを新システムやパッケージソフトで置き換えたり、システムを廃止するためのフェーズです。引退する対象システムと他のシステムとの相互作用を分析したり、引退する対象システムのデータを抽出、変換し、新システムに移行させるなどの作業を行います。

運用及びサポート作業分野は、稼動フェーズで行われる作業とほぼ重なりますが、サポートの対応戦略、バックアップやリストア、災害への対応などが追加されています。

エンタープライズ作業分野群は、以下のような作業分野で構成されています。

---

[1] UP の概要は、本白書第 3 部の付録で説明しています。

- エンタープライズビジネスモデリング作業分野
- ポートフォリオ管理作業分野
- エンタープライズアーキテクチャ作業分野
- 戦略的再利用作業分野
- 人材管理作業分野
- エンタープライズアドミニストレーション作業分野
- ソフトウェアプロセス改善作業分野

エンタープライズビジネスモデリング作業分野では、経営方針や経営戦略から出発し、企業全体の業務モデルをエンタープライズビジネスモデルやエンタープライズデータモデルなどにより定義します。これらエンタープライズビジネスモデルやエンタープライズデータモデルは、他のエンタープライズ作業分野の情報源になります。

ポートフォリオ管理作業分野では、企業に存在する既存システム及び企画中の新規システムの全体像を把握することにより、それらのシステムの開発や拡張の優先度を決め、開発の承認を行ったり、リスクを管理するための作業を行います。

エンタープライズアーキテクチャ作業分野では、企業内のシステムで共通に使われる技術アーキテクチャを考案し、そのアーキテクチャを検証し、そのアーキテクチャを開発プロジェクトで適用するのを支援します。

戦略的再利用作業分野では、既に開発されたソフトウェアから再利用できる成果物を収穫し、それをさらに洗練することで得られた再利用できる資産を作り、その資産のプロジェクトへの適用を促進します。

人材管理作業分野では、IT 技術者のためのキャリアパスを企業内で作成したり、世代間の技術の伝承を促進し、採用を行ったり、教育を行います。

エンタープライズアドミニストレーション作業分野では、企業の施設やハードウェアの資産の管理を行ったり、データやネットワークの管理を行ったり、セキュリティを管理したりします。

ソフトウェアプロセス改善作業分野では、UP のような新たな開発プロセスを組織に作ったり、テーラリングしたり、導入します。

EUP において、これらの作業はプロジェクトマネジメントオフィス（ PMO ）やソフトウェアエンジニアリングプロセスグループ（ SEPG ）などの組織やエンタープライズビジネスモデラー、エンタープライズアーキテクト、エンタープライズアドミニストレータなどの役割で担うことが提案されています。さらに、これらのエンタープライズ作業分野群で実行すべき作業や成果物の種類などが提案されています。

EUP の特徴の 1 つは、これらエンタープライズ作業分野を行う際に、膨大な文書（マニュアル）に頼るのではなく、エンタープライズ作業を行う役割の人たちによる教育と実地

指導を強調している点です。このような考え方は、変化への対応を重視するアジャイル開発の考え方と通ずるものです。

1.2. プロセス改善と測定

日本でソフトウェア開発が欧米に日本では従来「情報システム部門が企画を行い、IT ベンダーが開発する」という形で IT 業務をウォーターフォール型開発（一括契約）で外注することが一般化してきました。ユーザ企業にとって、外注には以下のような点でメリットがありました。

- 開発上のリスクを負わない（機能や期間に関する失敗は IT ベンダーの責任）
- IT の専門家を自社で抱える必要はない

これらのメリットの反面で、外注は以下のようなデメリットを伴います。

- 業務に役立つソフトウェアを迅速かつ適切なコストで開発できない
- IT 技術について一貫性がある、現実的な自前の方針を持てない

このような IT ベンダー丸投げの状況は、IT ベンダー側のビジネスのありかたもゆがめてしまいます。

- リスクを負わない（赤字にならないようにマージンを積む）
- 開発の効率化に熱心に取り組まない

後者については、例えばオープンソース製品の採用やハードウェアプラットホームの変更について自ら判断できず、IT ベンダーのいいなりになってしまうというようなことを意味します。

このような IT ベンダーにお任せの現状に対するユーザ企業側の疑問が次第に高まってきています。しかし、それでも IT ベンダーにお任せの現状を変えることのリスクの大きさに二の足を踏むユーザ企業が多いのが現実です。

このような状況で、「ユーザ企業」と「IT ベンダー」の間の関係をよりよいものにするために「測定」というのは有力な武器になりえます。つまり、ユーザ企業が IT ベンダーの「開発能力」を客観的に「測定」できるようになれば、自分たちが付き合っている IT ベンダーの現在の実力を把握し、さらに開発効率化への取り組みがなされているのかを評価することができるようになります。

そのような測定に対する取り組みの例としては、IPA が刊行している「データ白書」[4]と JUAS が刊行している「ユーザ企業ソフトウェア・メトリックス調査」[5]が挙げられま

す。前者はIT ベンダー側のデータを収集、分析したものであるのに対して、後者はユーザ企業側のデータを収集、分析したものです。これらのデータを参照することで、IT ベンダー側やユーザ企業側の QCD に関する一般的な傾向を把握することができます。また、これらの調査で用いている開発工程やソフトウェアの種類による分類は自社で同様なデータを収集するために非常に参考になります。

これらの調査は各ユーザ企業やIT ベンダーの全体的な傾向を調べることを目的としています。そのため、各企業がこれらのデータをベンチマークとして用いようとする場合に、以下のような問題に直面します。

- 示されたデータが大きくばらついており、そのばらつきが何に起因しているのかがよく分からない
- 同様なデータを測定した場合に、自社の分布がどうなるかが分からない

例えば、開発生産性は開発プロセス、開発者のスキルや経験、アーキテクチャや開発言語の選択、仕様変更の発生頻度、プログラムコード等の品質などによって大きく影響されると考えられますが、自社の条件と同じような条件のデータを前述した白書から抽出するのは困難です。しかし、それらの要因と開発生産性の関係を明らかにならないとソフトウェア開発プロセスにおいて改善すべき点が明確になりません。また、そもそも自社のデータがなければ白書のデータとの比較は困難です。

このような問題を解決するためには、自社でもプロジェクトの測定を行い、データを蓄積する必要があります。その際に大切なのは、比較的データを取りやすい欠陥密度などの品質に関するデータを取るだけではなく、「生産性」に関するデータも積極的に取り、開発委託者と共有していくということです。ソフトウェアの品質、開発生産性、コストの 3 点に基づいてIT ベンダーの実力を定量化することで、以下の 2 点が実現できます。

- IT ベンダーの実力に基づく、より合理的な取引条件を設定できる
- IT ベンダー自身が自らの実力の向上に意欲的に取り組むようになる

このような測定を行う際に注意すべきなのは、「生産性」を測定する際の開発規模の取り扱いです。開発規模を測定する際の情報源としては、一般的に以下の 2 種類のものが考えられます。

- 開発初期段階のソフトウェア仕様書
- 納品されたソフトウェア

しかし、これらを情報源として測定した機能規模は開発の開始時、完了時の値であり、

ソフトウェア開発に「仕様変更」が発生した場合の開発労力を反映したものではありません。すなわち、「開始」または「完了」時の規模だけで開発生産性を評価すると、「仕様変更」に対応すればするほど「生産性」が低下することになります。前述したアジャイル開発のように、「仕様変更」に積極的に対応する開発方式の開発生産性を適切に評価するためには、「仕様変更」の規模も定量化する方法が必要になります。

このような「仕様変更」の規模は、COSMIC 法 [6], [7]などの機能規模測定手法を使えば測定することができます。COSMIC 法では、図 2 に示すようにソフトウェアの仕様などに基づいて測定対象のソフトウェアと外界、測定対象のソフトウェアと永続ストレージとの間のデータ移動をモデル化し、それらのデータ移動の数を集計することで規模を測定します。仕様変更が発生した場合には、変更前と変更後のソフトウェアの仕様に対応するデー

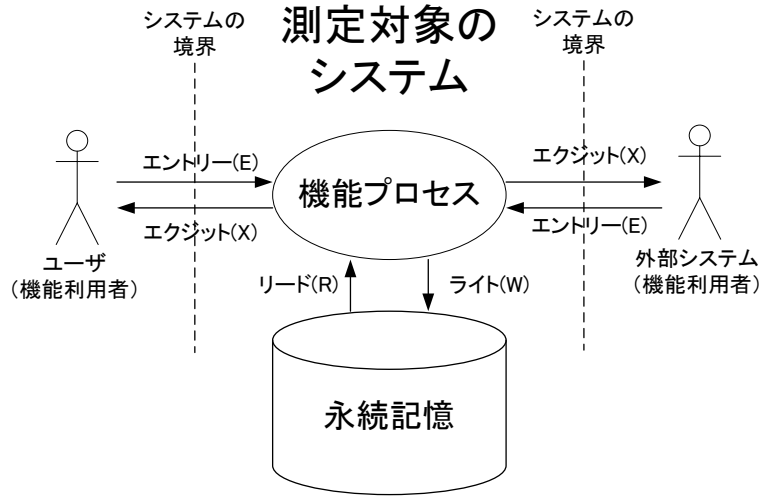

図 2 COSMIC 法の概念図

タ移動をモデル化し、仕様変更の結果として追加/削除/変更されたデータ移動数の合計値を求めることで「変更規模」を測定することができます。このように「変更規模」を測定することで、「仕様変更」の規模を定量化できます。

この「仕様変更」の規模測定と類似した問題として、「改修」の場合の開発規模や生産性をどのように測定するかという問題があります。「改修」の開発規模も、COSMIC 法のような機能規模測定手法を用いて「改修前のシステム」と「改修後のシステム」の間の変更規模により定量化することができます。「改修」の生産性を求めることで、新規開発と同様に開発ベンダーのパフォーマンスを定量化したり、改修のプロセス改善の手掛かりを得ることができると考えられます。

2. 参考文献

[1] Scott W.Ambler, エンタープライズ統一プロセス, 翔泳社, 2006

[2] イヴァー ヤコブソン, ジェームズ ランボー, グラディ ブーチ, UML による統一ソフトウェア開発プロセス―オブジェクト指向開発方法論, 翔泳社, 2000
[3] フィリップ・クルーシュテン, ラショナル統一プロセス入門 第 3 版, アスキー, 2004
[4] 独立行政法人 情報処理推進機構, ソフトウェア開発データ白書 2010-2011, 独立行政法人 情報処理推進機構, 2011
[5] (社)日本情報システム・ユーザー協会, ソフトウェア・メトリックス調査 2010, (社)日本情報システム・ユーザー協会, 2010
[6] JFPUG の COSMIC 法のページ, http://www.jfpug.gr.jp/cosmic/top.htm
[7] ビジネスアプリ開発者のための機能規模測定手法 COSMIC 法入門, http://www.ogis-ri.co.jp/otc/hiroba/technical/IntroCOSMIC/index.html

第3部：OGIS Scalable Agile Method の真髄

第３部：OGIS Scalable Agile Method の真髄(ver 1.01)

本白書では、まずスクラムの課題及び日本でアジャイル開発の普及を阻んでいる 8 つの不安について述べます。それらの課題や不安を解決する手段として、OGIS Scalable Agile Method（以降 OSAM と略す）を提案します。OSAM は、開発手法と測定技術で構成されます。スクラムとアジャイル UP を組み合わせることで OSAM の開発手法部分は高い品質と大規模化を実現します。また、機能規模測定手法 COSMIC 法を使うことで測定技術は仕様変更を組み込んだ契約やパフォーマンスベンチマーキングを実現します。最後に、OSAM の原則と OSAM がアジャイル開発の普及を阻んでいる 8 つの不安をどのように解決するかを論じます。

1. 解決すべき課題とスクラム

白書第1部で述べたように、お客様（ユーザ企業）が現在直面しているのは以下のような問題です。

- 業務に必要なソフトウェアがタイムリーに開発できない（納期や予算がオーバーする）
- 特に、開発すべきソフトウェアの規模や新奇性が大きくなるにつれて失敗する可能性が高まる

10 年前に比べてビジネスの競争が激しくなっている現在、業務に必要なソフトウェアをタイムリーに開発できないことのダメージは、戦略的な業務であればあるほど深刻になっています。しかし、開発スピードだけを追求して品質が低いソフトウェアができると将来の変更や保守のコストが増大する恐れがあります。

また、プラットホームの移行や OSS（Open Source Software)の導入によるシステムの近代化を行ったり、SOA などによりシステムの連携を行う際にも、開発に伴う技術的なリスクに対処しながらタイムリーに開発を行うことの必要性はますます高まっています。

これらの問題を解決する方法として有効なのが、アジャイル開発です。白書第1部で述べたように、アジャイル開発の特徴は以下のとおりです。

- 概ね１か月以内の短い周期で動くソフトウェアを作成し、顧客のフィードバックを得る
- 自律性の高いメンバーで構成されるチームと技術的な工夫で、顧客ニーズに柔軟に対応する

これらの特徴により、「業務に必要なソフトウェアがタイムリーに開発する」ことが可能になります。具体的なアジャイル開発手法で言えば、スクラム[1]はこれらの特徴を満たす最小限の開発手法だと言えます。

また、「周期的に動くコードを作成する」というところでスパイラルモデルを使うことで、技術的リスクに対処して開発を進めることができます。この点は、統一プロセスのような反復型開発手法のメリットとして強調されてきました。アジャイル開発手法でも同様なことは実現可能であり、このことを意識して開発を進めることが重要です。

スクラムでは解決できない問題もいくつかあります。

- 高いソフトウェアの品質を実現する
- 大規模なソフトウェア開発を行う

日本固有のユーザ企業と IT ベンダーに分かれた産業構造がここ数年間で大きく変わるという可能性が低いことを前提とする場合には、ユーザ企業と IT ベンダーの間でどのような契約を結べばよいかということも問題になります。

2. アジャイル開発に対する 8 つの不安

2000 年から 2005 年の期間に、日本にもアジャイル開発が紹介され、XP (eXtreme Programming)[2]を中心としたアジャイル開発のブームが起こりました。その際、結果的にアジャイル開発が日本では定着せず、アジャイル開発がさらに普及した欧米とは異なる道をたどりました。また、欧米では最初に XP がアジャイル開発ブームに火をつけましたが、それが次第にスクラムを採用する割合が高まり、現在ではスクラムが最も普及しています。XP は、テスト駆動開発など変化に対処するための技術プラクティスを中心とした開発手法であり、技術的な観点では非常によくできた手法です。しかし、幅広く普及しなかった原因は以下のようなところにあると考えられます。

- 開発依頼者にとってはその技術的なプラクティス中心のところが理解しづらい
- 技術的なプラクティスを一挙にすべて実行するのが難しい

これに対して、スクラムの特徴はアジャイル宣言の価値や原則を実現するために開発依頼者と開発者が実践すべき必要最小限のプラクティスに絞り込んでいる点にあります。これが、スクラムが日本以外で広く普及した大きな原因だと考えらえられます。その一方、スクラムは XP の技術プラクティスを否定するものではなく、むしろそれらのプラクティスを段階的に取り入れることを推奨しているという点も理解する必要があります。

日本でアジャイル開発が定着しなかった大きな要因は、以下のような開発依頼者の不安を解消できなかったことにあるのではないかと筆者は考えています。

A) 最終的に何ができるか分からない
B) いつ開発が終わるか分からない
C) 開発費用をどのように見積もればよいか分からない
D) 従来の開発と比べてコストアップになるのではないか
E) 開発依頼者側の負荷が高いのではないか
F) 開発されるものの品質が悪いのではないか
G) 高いスキルのメンバーしか実践できないのではないか
H) 保守のためのドキュメントは作成されるのか

このような不安のうち、A)-C)は日本固有のユーザ企業と IT ベンダーに分かれた産業構造ゆえに重要になるものです。G)は、「XP の技術プラクティスをすべて実行しなければならない」という誤解に基づくものだと考えられます。

そもそも A), B), C)を解決すべきか否かは、要求の不確定性とシステムの戦略性によって変わります。これらの点については、後の「測定に基づく契約と見積もり」の節で論じたいと思います。

D)は、以下のような 2 つの不安に分解することができます。

- アジャイル開発はウォーターフォール型開発と比べて開発生産性が下がるのではないか
- 仕様変更を受け入れると、手戻りで生産性が下がるのではないか

まず、ウォーターフォール型開発との開発生産性の比較で言えばアジャイル開発の方が 16%開発生産性が高いというデータ[3]が報告されています。この報告は、26 のアジャイル開発プロジェクトの開発生産性を 7,500 のウォーターフォール型開発プロジェクトの生産性と比較したものです。

「手戻りで生産性が下がる」という疑問に対する回答は、「仕様変更は(アジャイル開発であろうとなかろうと)開発生産性を下げる」ということです。しかし、技術的プラクティスを使うことで仕様変更に対する開発生産性の低下を抑制できる可能性があります。つまり、変更のコストを抑制できるということです。

E)については、従来の開発よりも業務に役立つソフトウェアを作るためには従来の開発以上に開発依頼者の積極的な関与が必要になるということです。従来の開発で業務に役立つソフトウェアが開発できなかったら、そのソフトウェアの改良を行うために次のリリースまで利害関係者の関与が続くことになります。アジャイル開発では、従来の開発において複数のリリースで行っていたことを前倒しで行うことになるので開発依頼者の積極的な関与が必要になるのです。但し、開発途上での開発依頼者の関与がどの程度必要であるかは、要求の不確定性が開発途上でどのように減るかによって決まります。要求の不確定性と開発依頼者の関与についても後の「測定に基づく契約と見積もり」の節で論じたいと思います。

G)は、「XP の技術プラクティスをすべて実行しなければならない」という誤解に基づくものなので

スクラムを採用することで解決できます。しかし、F)で言及されているような品質に対する不安に対処するためには技術プラクティスが必要になります。つまり、高い品質を求めるならばスクラムに技術プラクティスを加えることが必要になるのです。

最後に H)ですが、これはアジャイル宣言の「包括的ドキュメントよりも動くソフトウェア」という価値に対する誤解に起因しています。アジャイル宣言が言っているのは、「包括的ドキュメント」を作成することで開発が進捗していると判断したり、顧客のニーズを確認するのではなく、「動くソフトウェア」で開発が進捗していると判断したり、顧客のニーズを確認した方がよいということです。また、アジャイル開発は「顧客の成功」を重視しますので、顧客が望めば「開発ドキュメント」を作成することを禁じているわけではありません。

これまで述べたことをまとめると、日本のお客様の抱くアジャイル開発への不安を解消するためには、以下の点が鍵になるということになります。

- スクラムの採用
- スクラムを補うための技術的プラクティスの採用
- お客様に受け入れられるような契約や見積もり方法の提案

3. OGIS Scalable Agile Method とは

OGIS Scalable Agile Method（以降 OSAM と略す）とは、前節で述べた不安を解消し、以下のゴールを達成するための開発フレームワークです。

お客様と開発者がより良いパートナーとして連携し、限られた時間と予算の枠内で、要求の変化への対応と技術的なリスクの管理を行いながらお客様のビジネスに役立ち、かつ高品質なソフトウェアを早く開発すること

このゴールを達成するために、OSAM は以下の 2 つの要素で構成されています。

- スクラムとアジャイル UP を組み合わせた開発手法（OSAM の開発手法部分）
- 機能規模測定手法 COSMIC 法[4]に基づく測定（OSAM の測定部分）

アジャイル UP は、付録で説明されている統一プロセス(Unified Process)をアジャイル化したプロセスフレームワークです。アジャイル UP は以下の点でスクラムを補います。

- 技術的プラクティスを補うことで成果物の品質を高め、変更への対応コストを低減する
- 開発途上にマイルストーンを設けて開発が全体にうまく進行しているかを確認できる

アジャイル UP は、XP の実装中心の技術プラクティスだけではなく、アジャイルモデリングというモデリングに関する技術プラクティスからも構成されています。このモデリングに関する技術プラクティスは、プロジェクトの規模が大きくなるにつれて問題となるコミュニケーションの精度の問題の解決に役立ちます。アジャイル UP の概要とスクラムとの関係については後の節で説明します。

機能規模測定手法 COSMIC 法は、白書第1部で説明したように通常のファンクションポイント法よりも簡易な機能規模測定手法です。OSAM の測定部分では、COSMIC 法を用いてスコープ変動の定量化やパフォーマンスの評価を行います。

3.1. アジャイル UP

アジャイル UP とは、アジャイル開発の技術プラクティスを取り入れて UP（Unified Process）をアジャイル化した開発プロセスフレームワークであり、考案者は Scott Ambler 氏です。アジャイル UP は、UP と、アジャイルモデリング, XP などのアジャイル開発で提唱された技術プラクティスを融合させたものになっています。なお、UP の概要は本書の付録で説明されています。

アジャイル UP の全体像は、図 1 で表現されます。

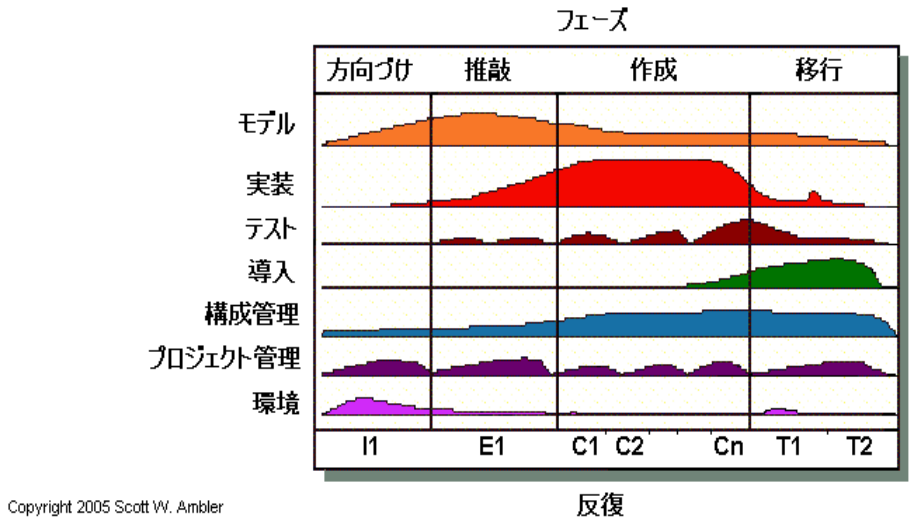

図 1　アジャイル UP の全体像

この図からアジャイル UP は、統一プロセスと同じ 4 つのフェーズで構成されているが、作業分野は 9 つから 7 つに減っているということが分かります。アジャイル UP の 7 つの作業分野の目的は、表 1 に示されます。

表 1 アジャイル UP の作業分野の目的

作業分野名	目的
モデル	組織の業務とプロジェクトの対象となる問題を理解し、問題に対処するための実現可能な解決策を明らかにする
実装	モデルを実行可能なコードへ変換し、基本的なテスト（特に単体テスト）を実行する
テスト	客観的な評価によって品質を保証する。 不具合の発見、システムが設計通りに動くことの検証、要求が満たされていることの確認などを行う
導入	システムの導入計画を立て、実施し、エンドユーザがシステムを使えるようにする
構成管理	プロジェクトの成果物に対するアクセスを管理する。 成果物のバージョンを継続的に追跡するほか、成果物に対する変更を制御、管理する
プロジェクト管理	プロジェクトで生じる作業を指示する。 リスクの管理、要員への指示(タスクの割り当て、進捗の管理など)、プロジェクト外部の人やシステムと協調して予算内でスケジュール通りに導入できるよう気を配ることなどが含まれる
環境	適切なプロセスやガイダンス(標準と指針)やツール(ハードウェア、ソフトウェアなど)をチームが必要な時に使えるようにして、ほかの作業を支援する

アジャイル UP では、これら 7 つの作業分野に対するワークフローを提供しています。アジャイル UP のワークフローは、開発作業、役割、成果物の関係を大雑把に示すものにとどまっており、UP を具体化したラショナル統一プロセスで数百ページに及んでいた以下の記述のほとんどが省かれています。

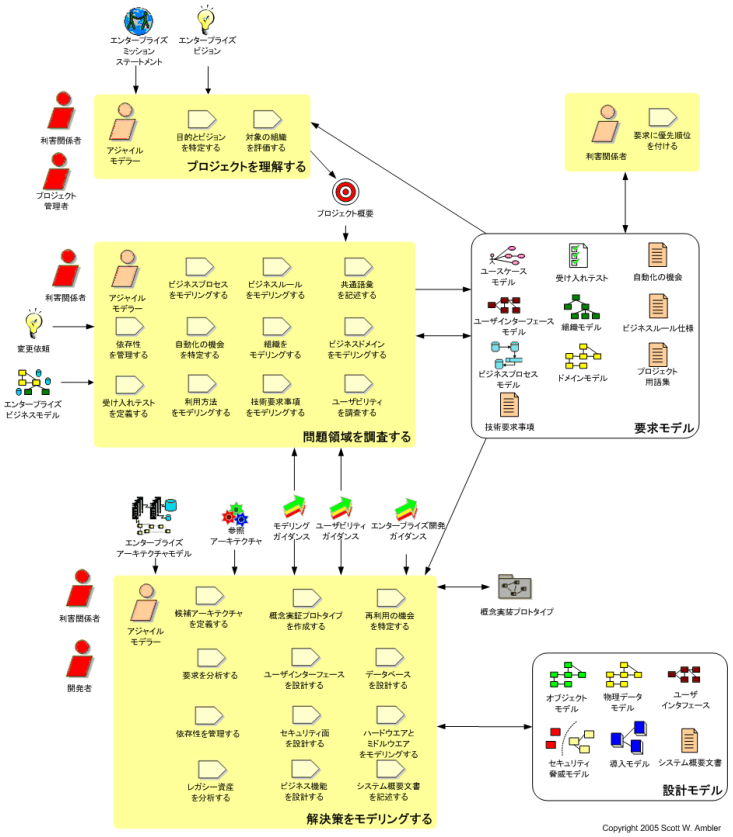

図 2　モデル業分野のワークフロー

- ワークフローや役割の詳細
- 成果物テンプレート
- 作業のガイドライン
- ツールによる自動化の説明

このため、アジャイル UP は非常にシンプルになっています。

アジャイル UP では、ラショナル統一プロセスでは膨大な文書で説明されていた開発作業の具体的な進め方をアジャイル開発の技術的なプラクティスで置き換えています。そのために、アジャイル UP が採用している技術プラクティスとしては以下のようなものがあります。

- 長期的なマイルストーンと反復
- アジャイルモデル駆動開発[7], [8]
- テスト駆動開発（テストの自動化）[9], [10]
- リファクタリング[11]
- 継続的なインテグレーション

長期的なマイルストーンは、開発を進める途上で開発が約束通り進んでおり、開発依頼者の期待している方向と一致しているかどうかを確認するチェックポイントです。また、各フェーズでは力点を置くところが以下のように変わります。

- 方向づけ：開発の初期構想やアーキテクチャ案の策定
- 推敲：アーキテクチャの検証と要求の獲得/モデル化
- 作成：アプリ機能の実装
- 移行：システムテストと本番稼働の準備

方向づけ以外の各フェーズでは、複数の反復を実行し、反復毎に動くコードで進捗を判断します。

ユーザ企業と IT ベンダーという枠組みの場合は、方向づけまでの検討はユーザ企業が主導し、推敲フェーズ以降の開発作業を IT ベンダーにアウトソースするという場合も多いでしょう。そのような場合には、IT ベンダー側は推敲フェーズ以降の 3 フェーズで開発を進めればよいことになります。

アジャイルモデル駆動開発（AMDD: Agile Model Driven Development）は、アーキテクチャや設計を考えたり、要求を表現するために軽量にモデリングを行いながら開発を行うものです。アジャイルモデル駆動開発は、以下のような特徴を持ちます。

- 数日から数時間程度という比較的短い時間でモデリングを行う
- 誰でも描ける軽いモデルを用いてモデリングする

誰でも描けるモデルを利用して、要求、アーキテクチャ、設計などを比較的短時間に集中してまとめたり、考えることで開発を効率し、ソフトウェアの品質を高めるというのがアジャイルモデル駆動開発の狙いです。

テスト駆動開発（TDD: Test Driven Development）は、各クラスが備えるメソッドのコードを書く前に、そのメソッドの動作を検証するためのテストコード(テストケース)を先に書くべしというプラクティスを元々意味しています。つまり、開発対象のソフトウェア本体のコードを書く前に単体テストのテストコードを書くことを求めています。また、単

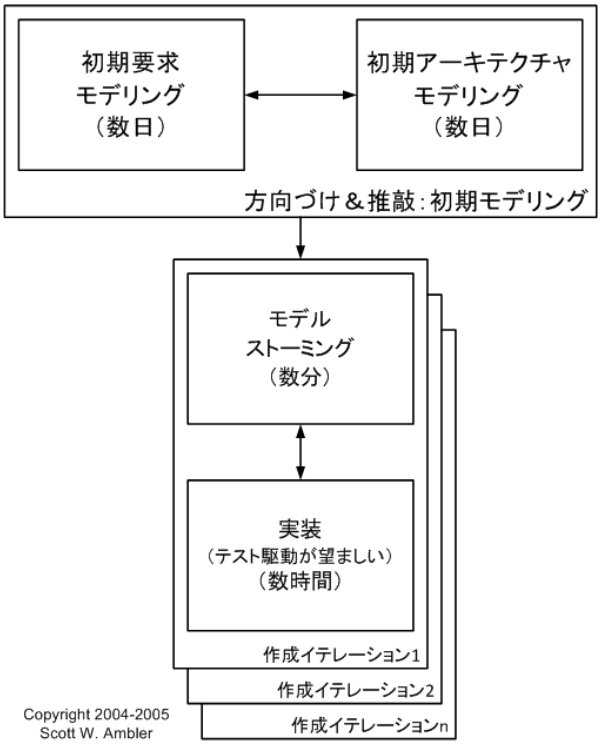

図 3　アジャイルモデル駆動開発の概念図

体テストはテストを自動化するための　xUnit（x は開発言語の種類に対応する文字が入る）というテスティングフレームワークを使って実装することが推奨されています。

テスト駆動開発の狙いは、以下の 2 つの点です。

- プログラムコードの各ロジックを書く際に単体テストのテストコードを書くことを習慣づける
- 単体テストを自動化することにより、プログラムコードを追加/変更した際に発生する回帰エラーをすぐに検出し、その修正コストを削減する

特に、後者は仕様変更に対する開発コストの増加を抑制するための鍵となるものであり、アジャイル開発の定石の 1 つと言えます。

リファクタリングは、既に存在するプログラムコードの機能を変えずに品質を改良するための作業です。例えば、2 つ以上のクラスの間にコードやインタフェースの共通性を共通のクラスに括りだしたり、1つのメソッドの複雑度が非常に高く保守性が悪いコード部分を分割して保守性を高めるなどの作業を指します。リファクタリングを安全に行うためには、テストの自動化を行うことが必要になります。

継続的なインテグレーションは、常にプロジェクト全体のソースコードにより実行コードをビルドする環境を整え、運用するというものです。継続的なインテグレーションを行う際には、以下のような 2 つの環境を導入することも多いです。

- 構成管理や変更管理を行うための環境
- ビルドで作成される実行コードのテストを自動化する環境

例えば、個人の環境で単体テストを実行して合格したコードだけを構成管理システムでチェックインすることを許し、チェックインされた最新のコードを定期的にビルドし、作成された実行コードの動作を自動化されたテストで自動検証するというような環境を構築します。また、ビルドエラーを発生した部分やテストの不合格の部分はプロジェクトメンバーにメール等で通知するようにします。

アジャイル UP に関して注意すべき点としては、「アジャイル UP のプラクティスを一挙にすべて取り入れるのは難しい」という点が挙げられます。特に現在の開発方法と大きく異なるプラクティスについては注意が必要です。そのために、アジャイル UP の各作業分野を開発メンバーに説明し、とりあえず実行できそうなプラクティスを話し合いで決めていき、段階的に取り入れるというアプローチが必要になります。

3.2. スクラムとアジャイル UP を組み合わせることの利点

OSAM の開発手法部分では、スクラムとアジャイル UP を以下のように組み合わせます。

- スクラムでアジャイル UP のプロジェクト管理作業分野を実装する
- アジャイル UP のプラクティスによりスクラムを補強する

後者は、アジャイル UP の各プラクティスの概要で説明したように開発を長期的に制御したり、ソフトウェアの品質を向上させることができるということです。さらに、アジャイル UP を組み合わせることには以下のような利点もあります。

- スクラムを実行する際の開発作業や成果物を考えるためにアジャイル UP で定義された作業や成果物を参考にすることができる
- プロジェクトの規模が大きくなり、複数チームに分かれた場合に問題になるコミュニケーションの精度をモデリングによって改善できる

これらの利点とスクラムでアジャイル UP のプロジェクト管理作業分野を以降説明します。

アジャイル UP にはプロジェクト管理作業分野が定義されていますが、そこで行う作業は以下のようなものです。

- プロジェクトや反復を計画する
- プロジェクトの状態を報告する
- 予算を管理し、リソースを確保する
- リスクを評価する

この「プロジェクトや反復を計画する」や「プロジェクトの状態を報告する」という作業について、詳細な作業の進め方について説明はありません。この部分にスクラムの開発の進め方を使うことができます。逆にスクラムを実行する際に、アジャイル UP の定義を以下のように活用することができます。

- 開発を始める際に、アジャイル UP の作業分野、役割、成果物を参考にして、自分たちの開発作業の進め方について合意を形成することができる
- スクラムの各スプリントのタスクを洗い出す際に、アジャイル UP のワーフフローに定義されている開発作業や成果物を参考にすることができる

アジャイルではない開発手法に慣れたメンバーがアジャイル開発を段階的に取り入れる場合には、このような参考資料が非常に役立ちます。

スクラムで大規模なソフトウェアを開発する場合には、以下のようなアーキテクチャやチーム横断的な設計をどのように行うかを考える必要があります。

A) アーキテクチャ構成要素の連携の設計

B) アーキテクチャ構成要素のモジュール性の確保
C) 情報の共有
D) 全体的な設計の一貫性の確保

A)は、各反復の実装対象となるエンドユーザ機能（外部機能）を実現するためのアーキテクチャ構成要素の連携（インタフェース）の設計をどうするかという点です。
B)は、アーキテクチャを構成するコンポーネント（パッケージ）、サービスなどをどのように独立性良く、変更が波及しないように設計するかという点です。
C)は、アーキテクチャの全体像、ドメイン（データ）モデル、共通機能などをどのようにチーム全体で共有するかという点です。
D)は、パッケージの階層、クラスの命名、クラスの連携パターンなどに一貫性を持たせて設計やプログラム構造の理解容易性を高めるという点です。
これらの課題を解決する方法として、以下の 3 つの方法が考えられます。

- リファクタリングと暗黙知を使う方法
- モデルを使う方法
- ドキュメントを使う方法

「リファクタリングと暗黙知を使う方法」は、先行してあまりアーキテクチャを考えず、リファクタリングにより徐々に独立性をよくし、一貫性を高めるという方式です。また、開発者が担当する機能を固定化せずにローテーションすることで、開発者がアーキテクチャの全体像をモデルなし（暗黙知）で理解できるようになります。この方法が有効なのは、小規模なものからソフトウェアを徐々に発展させることができ、同じ開発者が比較的長く同じソフトウェアを開発し続ける場合です。このような場合がよく当てはまるのは、ソフトウェアを内製する欧米の企業やネット企業、パッケージソフトウェア製品やクラウドのベンダーです。
「モデルを使う方法」は、先行してある程度アーキテクチャを考え、アーキテクチャの全体像、データモデル、共通機能などをモデルで表現するものです。それらのモデルにより、早い段階から独立性や一貫性を高めたり、チーム内で理解を共有します。この方式が有効なのは、比較的短期間に一時的に集まったプロジェクトメンバーでソフトウェアを開発する場合です。このような場合がよく当てはまるのは、日本の大規模な業務用アプリケーション開発のように IT ベンダーが一時的に多くのプロジェクトメンバーを集めてユーザ企業向けに開発を行う場合です。
「ドキュメントを使う方法」は、先行してある程度アーキテクチャを考え、アーキテクチャの全体像、データモデル、共通機能などを文書で表現するものです。それらの文書により、早い段階から独立性や一貫性を高めたり、チーム内で理解を共有します。この方式の利点は「モデルを使う方法」とほぼ同じです。但し、文書はモデルに比べて変更コスト、特に整合性を維持するコストが高く、アジャイル開発が登場した背景となった「変化に対応する」という点では「モデル」ほど適していません。

表 2 開発形態と設計手段

設計の考案と共有手段	開発の継続性の前提	開発者の継続性の前提	変化への対応	ソフトウェアの種類
リファクタリングと暗黙知	あり	あり	強い	パッケージクラウド
モデル	なし	なし	強い	業務アプリ
文書	なし	なし	弱い	業務アプリ

さらに、文書を増やせば増やすほど文書の変更や整合性維持のコストが高まるという点が短所になります。

以上の議論をまとめると、どの設計手段が適しているかは以下の 3 つの条件によって決まるといえます。

- 開発の継続性
- 開発者の継続性
- 変化への対応

「開発の継続性」とは、対象となるソフトウェアの開発作業が1つのリリースに留まらず、複数のリリースの間で継続するかどうかを意味します。また、「開発者の継続性」とは複数のリリースの開発を通じて開発者の大半が継続して開発に従事するかどうかを意味します。

日本での大規模な業務アプリ開発を考えると、短期に一時的な開発要員が集まるという点で「開発の継続性」も「開発者の継続性」も「ない」ということになります。そのような状況において開発途上で変化に対応することが求められた場合には、前述した A)-D)の課題を解決する手段としては「モデル」が有効だと考えられます。

A)-D)の課題の解決が問われるのは、2 つ以上のチームでアジャイル開発を行う場合です。1 チームを 10 名以下と考えると、10 名を超えるチームでは「モデル」を使うことの有効性が高まると考えられます。

「モデルを有効」とする考え方に対する反論としては、「モデリングできる人（モデラー）が少ない」というものや「モデリングできる人（モデラー）のコストが高い」というものが考えられます。しかし、前述した A)-D)の課題を解決する人をプロジェクトの一部のメンバーに限定したり、アジャイルモデリングで推奨されるような敷居の低いモデリングテクニックを使うことで、これらの問題は致命的ではなくなると考えられます。

3.3. 測定に基づく契約と見積もり

この節では、以下の点について論じ、機能規模測定手法 COSMIC 法に基づく測定が見積もりや契約にどのように役立つかについて説明します。

- 開発途上での要求の不確定性と見積もり/契約/開発依頼者の関与
- スコープの変化を許容する開発契約

開発途上での仕様変更に対応し、顧客のビジネスに役立つソフトウェアを開発するのがアジャイル開発の意義ですが、それは同時に本白書の 2 章で述べたような以下の不安を生みだす原因になっています。

A) 最終的に何ができるか分からない
B) いつ開発が終わるか分からない
C) 開発費用をどのように見積もればよいか分からない

これらの不安にどの程度対処できるかは、開発当初における要求の不確定性の大きさと、その不確定性が開発途上でどのように変化するかによって決まります。

まず、要求の不確定性の大きさを以下の 3 段階に分類します。

- 不確定性が大きい　→　要求の詳細のほとんど（50%以上）が決まっていない
- 不確定性が中ぐらい　→　要求の詳細の 50%から 80%ぐらいが決まっている
- 不確定性が小さい　→　要求の詳細の 80%ぐらいが決まっている

ここで要求の詳細とはユースケース記述のイベントフローや画面定義などを意味します。また、「要求の X%が決まっている」とは、開発途上での要求の詳細の変化は(100−X)%以内に収まるということを意味するとします。

表 3　不確定性の変化の起こりえる場合

開発当初の要求の不確定性	推敲フェーズ終了時点での要求の不確定性
大	大
大	中
大	小
中	中
中	小

開発初期段階で不確定性が大きい場合と中ぐらいの場合に、3.1節で説明したアジャイル UP の推敲フェーズのようなフェーズを 1-3 ケ月程度設けることで要求の不確定性がどの程度変わりうるかを考えます。

不確定性が開発当初に小さい場合と不確定性が開発とともに大きくなる場合を除くと、起こりえる場合は表 3 の 5 つの場合になります。これらの 5 つの場合について、以下の観点で考えてみましょう。

- 見積もりが可能か
- 一括（請負）契約が可能か

- 開発依頼者の関与がどれくらい必要か

以降、推敲フェーズ終了時点での要求の不確定性の大きさで場合分けして議論します。

表 4　推敲フェーズ終了時点の不確定性と見積もり/契約/関与

	推敲フェーズまで			作成フェーズ以降		
	既存の見積もり方法の精度	契約形態	開発依頼者の関与	既存の見積もり方法の精度	契約形態	開発依頼者の関与
不確定性が小さくなる場合（大→小または中→小）	悪	準委任（SES）契約	大または中	良	一括（請負）契約	大
不確定性が中ぐらいの場合（大→中または中→中）	悪	準委任（SES）契約	大または中	やや悪	準委任（SES）契約？	中？
不確定性が大きいままの場合（大→大）	悪	準委任（SES）契約	大	悪	準委任（SES）契約	大

まず、不確定性が小さくなる場合（大→小、中→小）では、不確定性が小さくなる作成フェーズ以降は従来の見積もり方法で費用を見積もり、開発スコープと費用を定めた一括（請負）契約が可能になり、開発依頼者の負荷も減ります。

不確定性が中ぐらいの場合（大→中、中→中）では、不確定性が中ぐらいで残るため従来の見積もり方法で精度よく費用を見積もることが困難であり、契約上の選択肢は準委任契約になります。また、不確定性を解決するために開発依頼者の継続的な関与が必要になります。

不確定性が大きいままの場合（大→大）については、開発スコープが定まらないために、開発スコープと費用を定めた一括（請負）契約を結べないことになります。その結果、選択しうる契約形態は準委任契約になります。ただ、準委任契約を選んだ場合、IT ベンダーが妥当なパフォーマンスで開発を行っているかどうかという点について不安を持つことになります。また、不確定性を解決するために開発依頼者の継続的な関与が必要になります。

これまでの議論では、「従来の見積もり方法」を前提としましたが、「見積もる」のではなく、「発生する変更の規模を測定する」という考え方に基づく別の解決策も考えられます。その別の解決策とは、以下のようなものです。

- スコープの変化を許容する契約
- パフォーマンスベンチマーキング

これらの実現には、ソフトウェアの規模や要求変化の規模を定量化する機能規模測定手法

の適用やプロジェクトの測定データの蓄積が必要になります。

機能規模測定手法を用いて、スコープの変化を許容するソフトウェアの調達方法としてはsouthernSCOPE[12]というものが提案されています。この方法は、オーストラリアのヴィクトリア州における情報システムの調達において開発されたものです。

southernSCOPE の実行手順は以下のとおりです。

① ソフトウェアを調達する際に、顧客はスコープマネジャーを雇う
② スコープマネジャーは、FP (Functional Point)法により初期の費用とスケジュールを見積もる
③ 顧客は、機能概要や制約を記述した RFP を作成し、入札にかける
④ 顧客は、調達先と機能規模あたりの単価に合意する
⑤ 分析フェーズを実行することで、要求仕様書を作成する
⑥ 要求仕様書に基づいてスコープマネジャーが機能規模を算出し、顧客は予算とスケジュールを考慮してどの機能が必要かを決める
⑦ 開発途上での変更について、 スコープマネジャーが費用に対する影響を算出し、顧客と開発者が合意する
⑧ 開発が完了した時点で、顧客は納品されたソフトウェアと変更に対する費用を支払う

southernSCOPE を導入してから、予算超過したプロジェクトが 84%から 10%以下に減ったと報告されています。フィンランドの Finnish Software Measurement Association (FiSMA)は、このsouthernSCOPE に「プロジェクトのデータを蓄積する」などのステップを追加するなどの改良を加えた northernSCOPE[13]という手法を提案しています。

southernSCOPE や northernSCOPE は、開発依頼者と開発会社が以下のようなことに合意していることを前提にしています。

- ソフトウェアの規模及び仕様変更の規模は機能規模で測定できる
- 開発時の契約時に機能規模あたりの単価に合意する
- 仕様変更への対応費用は、仕様変更の規模に機能規模あたりの単価を掛けたものとして算出する

これをスコープの変化を許容する契約と呼びます。

OSAM では、このスコープの変化を許容する契約を以下のように実装することを提案します。

- 機能規模の測定方法として COSMIC 法を使う
- スコープマネジャーの役割は IT ベンダーが担う

COSMIC 法は、白書第 2 部で説明したように以下のような特徴を持つ機能規模測定手法です。

- ISO や JIS 標準の機能規模測定手法である
- 比較的容易にソフトウェアの規模や仕様変更の規模を測定できる
- 測定手法のマニュアル等が Web 上で公開されており、誰でも使える

このような特徴を持つため、機能規模の測定の際に作成したデータ移動モデルを IT ベンダーが提供すれば、機能規模の根拠を開発依頼者が確認することが可能です。

OSAM の契約スキームを使うことで、開発途上で受け入れる仕様変更の規模（上限）を契約時点に設定して開発契約を締結することが可能になります。これを仕様変更上限設定型契約と呼びます。

また、委任契約のように開発者の工数を契約する場合にも、各時点で開発した機能規模と実績工数（費用）から機能規模あたりの単価を求めて開発のパフォーマンスを評価することができます。

不確定性が中ぐらいの場合（大→中、中→中）と不確定性が大きいままの場合（大→大）には、このような仕様変更上限設定型契約やパフォーマンス評価を用いることで、ユーザ企業と IT ベンダーとが公平なパートナーとして前向きにアジャイル開発を推進できるようになると期待できます。

また、アジャイル開発において開発方法を改善したり、プロジェクトが大規模化した際の全体の状況を把握（共有）するためにも、機能規模を用いたプロジェクトの測定が役立ちます。

3.4. OGIS Scalable Agile Method の原則と不安の解消

これまで説明したスクラム、アジャイル UP、COSMIC を融合した OSAM の全体像は、以下のような原則で表現されます。

1）開発の大きな方向性を顧客と合意するために長期的なマイルストーンを持つべし
補足）特に、技術リスクには早期に対処すべし
2）動くコードで客観的に進捗を判断し、顧客からのフィードバックを得るべし
3）メンバーの自律性と個性を尊重し、チームとしてのパフォーマンスを高めることに力を注ぐべし
4）要求の変化の影響を最小限に留め、品質のよいソフトウェアを実現するための技術的な方策を適用すべし
5）要求を理解し、解決策を考案し、コミュニケーションの精度を高めるためにモデリングを行うべし
6）要求の規模とその変化を測定し、その測定結果に基づいて顧客と開発スコープの設定や変更について合意すべし

5）で述べられている「技術的な方策」とは、テスト駆動開発(TDD)、アジャイルモデル駆動開

発(AMDD)、リファクタリングを意味したものです。

OSAM の大きな利点の 1 つは、3.2 節で述べたようにプロジェクトの規模の拡大に対応することができる点です。この特徴が、“Scalable”と命名した所以です。それに加えて、OSAM は以下のような不安に対応し、アジャイル開発を活用して業務に役立つソフトウェアを早く作ることを支援します。

A) 最終的に何ができるか分からない
B) いつ開発が終わるか分からない
C) 開発費用をどのように見積もればよいか分からない
D) 従来の開発と比べてコストアップになるのではないか
E) 開発依頼者側の負荷が高いのではないか
F) 開発されるものの品質が悪いのではないか
G) 高いスキルのメンバーしか実践できないのではないか
H) 保守のためのドキュメントは作成されるのか

A), B), C)は、アジャイル UP の長期的なマイルストーン、COSMIC 法、仕様変更上限設定型契約で解決します。F), D)は、AM, TDD, リファクタリングにより高い品質を実現し、変更コストを低減することで解決します。E)は、推敲フェーズを通じて可能な限り要求の不確定性を減らすことで、開発依頼者の負荷を減らします。G)は、スクラムのような敷居の低い開発手法から出発し、アジャイル UP のプラクティスを徐々に取り入れることで解決できます。H)は、ドキュメント作成をアジャイル UP の移行フェーズの作業として位置付けることで解決できます。

4. 今後の課題

OSAM の今後の課題としては、以下の点が挙げられます。

- 要求の確定を促進するための技術
- 開発メンバーの知識や習熟度のバラツキへの対処

要求の確定を促進するための技術とは、開発者も含む複数の利害関係者の要望や制約をうまく調整し、迅速に要求としてまとめるための技術です。このようなことを実現するために有望な技術としては、Joint Application Development (JAD) の流れを組んだ要求ワークショップ[14]というものが提案されています。今後、要求ワークショップの方法論を研究するとともに日本での要求ワークショップを効果的に開催するためのノウハウを蓄積していく必要があります。

知識や習熟度のバラツキを克服する方法とは、複数の会社のメンバーから構成されるような大規模プロジェクトの場合に、以下のような点においてプロジェクトメンバーのバラツキが大きくなることへの対処方法という意味です。

- プロジェクトで求められるドメインや技術に関する知識
- 開発手法と技術プラクティスに対する習熟度

大規模プロジェクトでは、プロジェクトメンバーの大半がこれらの知識や習熟度が少ないメンバーで構成される場合もあります。そのように知識や習熟度が少ないメンバーが多い場合にスクラムのようなアジャイル開発を単純に適用しただけでは、高い開発生産性を達成するのは困難です。このような状況で高い開発生産性を確実に達成する方法の1つは、以下のように従来の開発手法のやり方を取り入れることです。

- ドキュメントの作成
- 作業の定型化、単純化
- 判断の集中

しかし、このような方法を使うと自己組織化されたチームが自律的に開発するというアジャイル開発の理想像からは外れてしまい、変化への対応力が損なわれる恐れがあります。

自己組織化されたチームが自律的に開発するという理想像の実現を大規模開発で目指す場合、以下の 2 点を整備することが必要になります。

- プロジェクトのメンバーの大半が、開発開始時点でアジャイル開発の基本的な知識を習得している
- プロジェクト固有のドメイン知識や開発技術を効率的に習得するための仕組みを提供する

これらの実現のためには、トレーニング教材や方法を整備するとともに、ペアプログラミングのようなメンバー間での技術移転の仕組みをうまく活用していく必要があります。

5. 付録

5.1. 統一プロセス

統一プロセスとは UML の策定の中心となった 3 人衆（Three Amigos）が提案した反復型開発プロセスフレームワークです。統一プロセスは、以下のような特徴を持ちます。

- マイルストーンを持つ反復型開発プロセス
- UML のモデルを中心にした開発作業で構成される

統一プロセスの全体像は、図 4 のようなハンプチャートという図で表現されます。

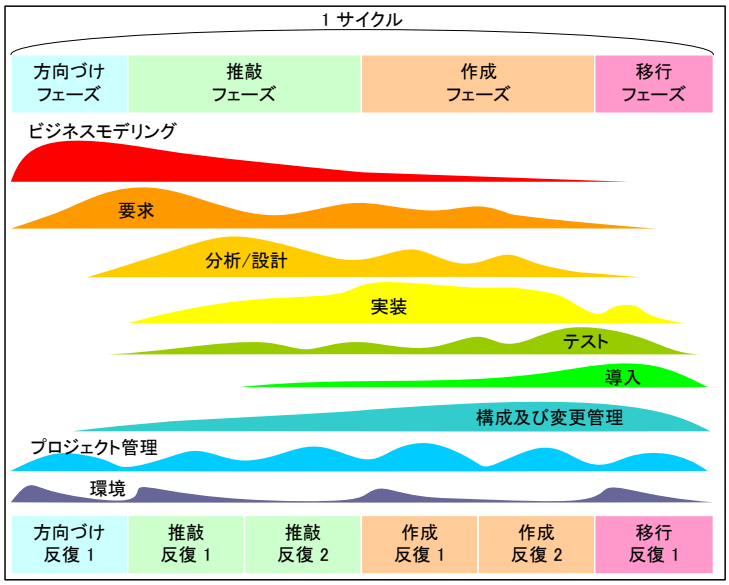

図 4 統一プロセスの全体像

この図の縦方向には、統一プロセスを構成する 9 つの開発作業のグループが色分けされた波うつ図形として示されています。これらの開発作業のグループを作業分野と呼びます。この図の横方向には、左から右への開発のライフサイクルが示されています。図の上部に並んだ色分けされた 4 つの矩形がライフサイクルを構成する 4 つのフェーズを表し、図の下側に並んだ 6 つの矩形が「反復」を表しています。統一プロセスでは、方向づけ以外の各フェーズは複数の反復で構成されていいます。

統一プロセスは、これらの作業分野を実行するための以下のようなものが定義されています。

- 役割
- ワークフロー
- 成果物

また、統一プロセスを具体化したラショナル統一プロセス (RUP: Rational Unified Process) では、これらの作業を進めるためのガイドライン、成果物テンプレート、作業を支援するためのツールの使い方などから膨大な HTML ファイルが提供されています。

統一プロセスとアジャイル開発手法の大きな違いは、「開発手法としての複雑さ」にあります。アジャイル開発手法は、「開発者の自律性を活かすために約束事をなるべく少なくする」という考え方に基づいています。そのため、少数の価値、原則、プラクティスで開発のやり方を定義しています。

それに対して、統一プロセスは大規模開発までを視野に入れたために、大規模にも対応できる開発方法をマニュアル化し、開発プロジェクト毎に必要な部分を取捨選択するという考え方で作成されています。そのために、開発手法の定義が複雑になってしまったのです。この点が、統一プロ

セスの短所になっています。

6. 参考文献

[1] ケン・シュエイバー, マイク・ビードル, 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, 2003

[2] ケント・ベック, XP エクストリーム・プログラミング入門―変化を受け入れる, ピアソンエデュケーション, 2005

[3] Mah, Michael., How agile projects measure up, and what this mean to you, Cutter Consortium Agile Product & Project Management Executive Report 9(9), 2008

[4] http://www.jfpug.gr.jp/cosmic/top.htm

[5] イヴァー ヤコブソン, ジェームズ ランボー, グラディ ブーチ, UML による統一ソフトウェア開発プロセス―オブジェクト指向開発方法論, 翔泳社, 2000

[6] フィリップ・クルーシュテン, ラショナル統一プロセス入門 第 3 版, アスキー, 2004

[7] Scott W.Ambler, オブジェクト開発の神髄~UML 2.0 を使ったアジャイルモデル駆動開発のすべて, 日経 BP 出版センター, 2005

[8] スコット・W・アンブラー, アジャイルモデリング―XP と統一プロセスを補完するプラクティス, 翔泳社, 2003

[9] ケント・ベック, テスト駆動開発入門, ピアソンエデュケーション, 2003

[10] Janet Gregory, Lisa Crispin, 実践アジャイルテスト テスターとアジャイルチームのための実践ガイド, 翔泳社, 2009

[11] マーチン・ファウラー, リファクタリング―プログラムの体質改善テクニック, ピアソンエデュケーション, 2000

[12] http://www.egov.vic.gov.au/victorian-government-resources/e-government-strategies-victoria/southernscope/southernscope-avoiding-software-budget-blowouts.html

[13] http://www.fisma.fi/wp-content/uploads/2008/09/northernscope-brochure-v152.pdf

[14] エレン・ゴッテスディーナー, 要求開発ワークショップの進め方 ユーザー要求を引き出すファシリテーション, 日経 BP, 2007

第4部：アジャイル開発がもたらすもの

第４部：アジャイル開発がもたらすもの

1. OGIS Scalable Agile Method の原点

当社では、1995 年に反復開発を取り入れた基幹系システム開発を皮切りに、以下のようなアジャイル開発と反復開発適用を推進してきました。

- 1995-2000 年：反復開発
- 2000-2005 年：統一プロセスに基づく反復開発
- 2006 年-2011 年：アジャイル開発（アジャイル UP とスクラム）

反復開発もアジャイル開発も、一定の期間毎に動くコードを作成しながら開発を進める手法です。この両者は、一定の期間毎に動くコードを作ることで客観的に進捗が評価できたり、お客様のフィードバックを得ることができるという点で共通しています。

これら各時期の代表的なプロジェクトである A, B, C プロジェクトの概要をまとめたものが表 1 です。これらのプロジェクトの結果からアジャイル開発や反復開発で以下のことが達成できたことが分かります。

- 大規模化
  - 30 名以上のプロジェクト規模でも開発を成功させることができた
- 納期厳守
  - 当初計画した納期を守って開発を完了できた
- 高い品質
  - 基幹系のシステムとして長期間に渡り、稼働し、機能拡張を行うことができた

これらのプロジェクトを成功させるために大きく影響した要因としては、以下のようなものを挙げることができます。

- お客様のニーズ
  - 3 つのプロジェクトとも「ビジネス上どうしても必要なので、従来の開発方法と異なるものだが挑戦しよう」という強い意志を持たれていた
- お客様の積極的な関与
  - 新しい開発方法を自ら体験し、理解するとともに、ベンダーと一緒にソフトウェアを作ろうという姿勢で開発を進めた

表 1 当社の代表的なアジャイル開発と反復開発の事例

	A プロジェクト	B プロジェクト	C プロジェクト
業種	製造業	製造業	リクルート業
前提条件、動機	全世界の工場に展開できるソフトウェアを開発したい（技術的なリスク高）	メインフレームからオープン系の移行で、納期とコストを守って開発したい	既存システムのドキュメントがないが、リリース日は厳守したい
開発手法	反復開発	統一プロセスに基づく反復開発	アジャイル開発
チーム規模	20 名程度	30 名程度	10 名程度
要求の不確定性	大	中	中 （仕様追加あり）
契約	準委任	準委任	請負 機能規模で契約
お客様	積極的な関与	積極的な関与	形よりも実質重視
開発結果	納期通りリリース	納期通りリリース	納期通りリリース
リリース後の状況	全世界の工場に展開 現在も稼働	従来よりも障害が少 現在も稼働	現在も稼働

- 反復開発
  - 一定の期間毎に動くコードを作ることで客観的に進捗が確認できたり、お客様のフィードバックを得ることができた。その結果として、理想と現実を俎上に載せて現実的な判断を行うことができた
- 技術力（モデリングやテスティング）
  - 開発途上での変更への対応や、開発後長期間に亘る機能拡張が可能になり、規模の比較的大きなプロジェクトでコミュニケーションを取れたのは、ドメインモデルやアーキテクチャをある程度先行して考えた結果であり、またテストの自動化（＊）も有効だったと考えられる

＊：A プロジェクトはテストの自動化を行っていない

これらのプロジェクトの経験をまとめたものが、白書第 3 部で説明した OGIS Scalable Agile Method なのです。

2. 企業の IT ガバナンスとアジャイル開発

今後アジャイル開発が普及する過程で、「かつての RAD (Rapid Application Development)の轍を踏まない」ことに注意する必要があります。つまり、「早く、安く、うまい」ことだけをよしとしていると、以下のような点に足を掬われる可能性があるということです。

A) 局所最適なアプリケーションの乱立
B) 経時的な拡張や保守コストの増加

誤解がないように言えば、現時点でもこれらの問題は存在します。そのため、これらは「アジャイル開発の普及」で新たに発生する問題ではありません。これらの問題のうちで B)については、第 3 部「OGIS Scalable Agile Method の真髄」で説明した以下のような技術プラクティスを適用することで１つの開発プロジェクトのレベルでの低減を行うことが可能です。

- アジャイルモデリングやリファクタリングによる設計品質の向上
- テスト駆動開発（テストの自動化）による回帰テストのコスト削減

しかし、B)の問題の中には例えば「１つの業務要求の変更が複数のシステムの変更に波及する」というように１つのプロジェクトという枠内では解決できないものもあります。このような問題や A)のような問題を開発するためにはアジャイル開発であろうとなかろうと、１つのプロジェクトや１つの開発ライフサイクルを超えた観点で以下のことを考える必要があるのです。

- 経営戦略を効率的に実現するためのシステムのあるべき姿
- あるべき姿への移行の道筋と推進体制

つまり、現在のビジネスに勝つための手段としてアジャイル開発を活用する一方で企業の IT 業務の全体最適化を行うためのガバナンスも考えていく必要があるということです。より具体的には以下のようなことが必要になります。

- 経営戦略に基づいたシステムの企画（優先順位付け）の実行
- 経営戦略に基づいたシステムを効率的に実現するためのアーキテクチャや開発方法の改善及び IT 担当者の育成
- 企業全体でのアーキテクチャの統一や開発方法の共有

アジャイル開発とも比較的整合する形で、このようなガバナンスを実現するためのフレームワークとして提案されたのが白書第 2 部で紹介したエンタープライズ統一プロセス（EUP）[1]です。また、企業システムのあるべき姿とその姿への移行シナリオを提案するものとして百年アーキテクチャ[2]があります。

エンタープライズ統一プロセスでは、経営戦略に基づいて企業全体で効率的に開発を行うために以下のようなエンタープライズ作業分野群を設定しています。

- エンタープライズビジネスモデリング作業分野：経営方針や経営戦略から出発し、それらを実現するための企業全体のビジネスプロセス、ドメイン、組織をモデル化する
- ポートフォリオ管理作業分野：企業全体のプロジェクトやプログラムの企画を行うとともに、それらのポートフォリオを管理する
- エンタープライズアーキテクチャ作業分野：企業のアーキテクチャに対するニーズに基づいて参照アーキテクチャを作成し、それらの参照アーキテクチャのプロジェクトへの適用を支援する
- ソフトウェアプロセス改善作業分野：プロジェクトのニーズに合うようにプロセス（アジャイル開発手法）をテーラリングするとともに、それらのプロセスのプロジェクトへの適用を支援する

プロジェクトの外でこれらの作業を行うことで、アジャイル開発で局所最適なソフトウェアを作ることを防ぎ、企業戦略に沿ったソフトウェアを開発することができます。（図 1）エンタープライズ統一プロセスの考え方は、白書第 3 部で紹介したアジャイル UP のモデル作業分野のワークフローに組み込まれています。また、エンタープライズビジネスモデリング作業分野での具体的なモデリングの進め方については[3], [4]が参考になります。

エンタープライズ統一プロセスの書籍（原書）が刊行されたのが 2005 年でそれ以降にクラウドや仮想化のような新たな技術が登場したり、アジャイル開発や OSS（Open Source Software)の普及が大きく拡大しました。そのため、エンタープライズ統一プロセスにはこれらの技術やアジャイル開発や OSS を考慮していなかったり、考慮が足りなかったりするという点で不十分な面があるということに注意する必要があります。

百年アーキテクチャでは、A), B)のような問題を「アーキテクチャ成熟度ステージ」という観点での成熟度の不足と捉えて、SOA（Service Oriented Architecture）、OSS、モデリングの活用により成熟度を向上させ、ビジネス上のニーズにより柔軟に対応できるような企業システムの実現を提案しています。このような方法を適用することで、エンタープライズ統一プロセスのエンタープライズアーキテクチャ作業分野の目指すべき大きな方向性を定めることができます。

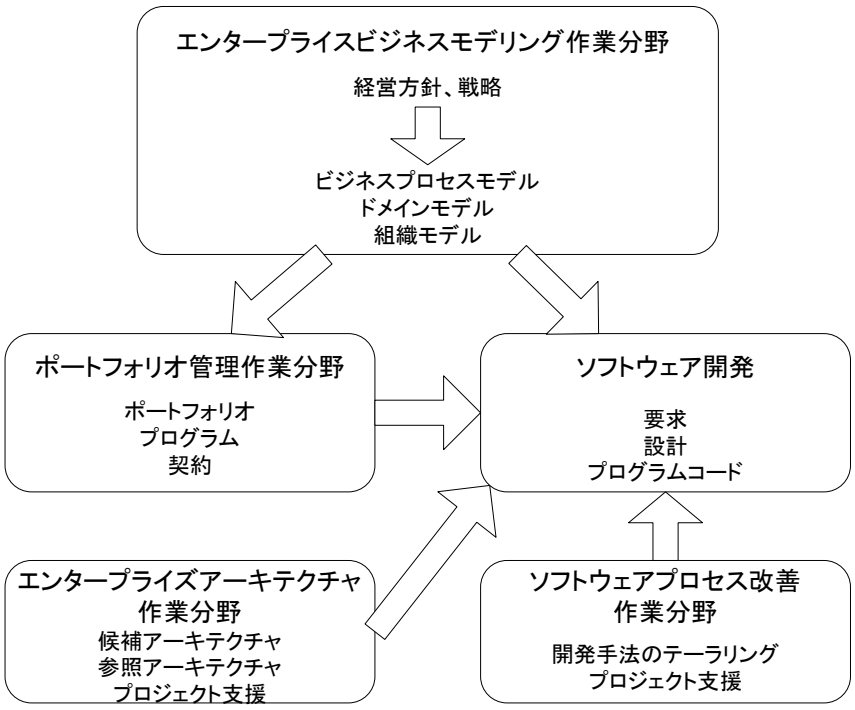

図 1エンタープライズ作業分野とソフトウェア開発の関係

アジャイル開発に「早く、安く、うまい」ことだけを期待すると、従来から存在した「局所最適なアプリケーションの乱立」などの問題が残るという点を見失いがちになります。そのような問題を解決するためには、エンタープライズ統一プロセスや百年アーキテクチャで提案されているような1つのプロジェクトや1つの開発ライフサイクルを超えた観点（ガバナンス）も必要になることを理解した方がよいでしょう。

3. スクラムの理論的な背景

白書第1部で紹介したスクラム[4]を生み出す理論的な背景となったのは、野中らの「組織的知識創造」[5]です。この「組織的知識創造」は、日本のメーカーがホームベーカリーや低価格コピー機などの画期的な製品開発の原動力となったものです。

「組織的知識創造」は、図 2 で示されるような日本の文化で重視されてきた言葉や文章で表現することが困難な「暗黙知」と西洋の文化で重視されてきた言葉や文章で表現できる「形式知」を相互に変換する SECI プロセスとそれを促進するコンテキストを土台としています。

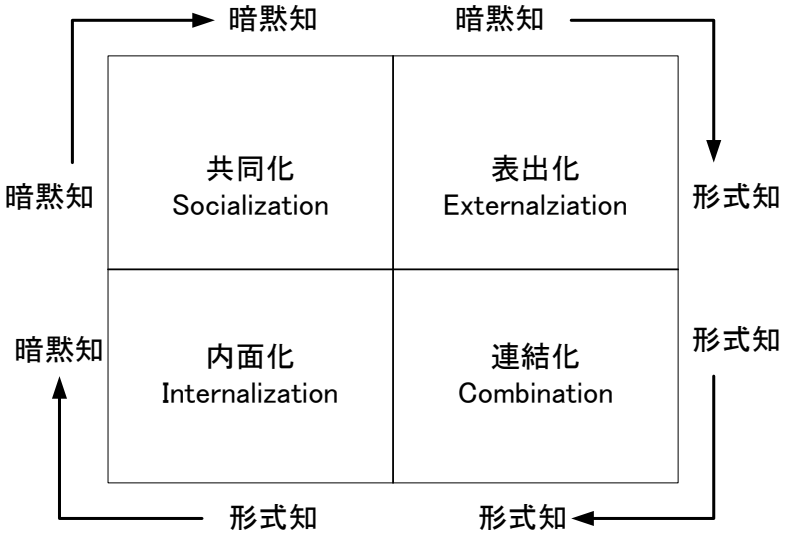

図 2 SECI サイクル

- SECI サイクルの知識変換モード
  - ➢ 共同化
  - ✧ 経験を共有することで、メンタルモデルや技能などの暗黙知を創造する
  - ➢ 表出化
  - ✧ 暗黙知をメタファー、アナロジー、仮説、モデルなどのコンセプト（形式知）に変換する
  - ➢ 連結化
  - ✧ コンセプトを組み合わせて１つの知識体系を創り出す
  - ➢ 内面化
  - ✧ 形式知を実践（具体化）してみて、それを追体験したり、実感することで暗黙知を生み出す

- SECI サイクルを促進するコンテキスト
  - ➢ 知識創造の目標やチームを生み出す「組織の意図」
  - ➢ チームのメンバーに自由な行動を認める「自律性」
  - ➢ 組織と外部環境との相互作用を刺激する「ゆらぎと創造的なカオス」
  - ➢ 組織に組み込まれた意図的な情報の「冗長性」
  - ➢ 複雑で多様な環境に対応するために組織のメンバーも同程度の多様性を持つ必要があるという「最小有効多様性」

このような「組織的知識創造」は、「競争力のある独創的な新製品（＝知識）を生み出す」

という点で有効だったと言えるでしょう。

スクラムは、このような「組織的知識創造」の考え方を取り入れて、ソフトウェアに対するニーズ（要求）という暗黙知をソフトウェアプロダクトという形式知に変換することを中心に据えた開発手法と捉えることができます。また、スクラムで重視している「自己組織化」という組織のあり方はSECIサイクルを促進するコンテキストの「自律性」や「最小有効多様性」に基づいていると考えることができます。つまり、「役割分担」や「上からの指示」に基づく組織ではなく、チームに与えられた課題を解決するという共通の目標のために、メンバー１人１人が自律的に自分がなすべきことを考えるという組織です。

「組織的知識創造」に対して、スクラムで追加された特徴としては以下のようなものがあります。

- 比較的短い周期でのSECIサイクルの繰り返し
- SECIサイクルを通じた継続的な改善を促進するためのスクラムマスターという役割の導入

前者はユーザニーズに即し、市場の変化に機敏に対応するために役立つものであり、後者はチームの開発能力を高めるために役立つものです。

スクラムがアジャイル開発手法として世界的に普及してきたという事実は、ビジネス競争力に役立つソフトウェアプロダクトを開発する上で「組織的知識創造」が有効だったことを示していると考えられます。

4. 「組織的知識創造」を阻害する要因とその克服

「組織的知識創造」を現在に活かすためには、「組織的知識創造」という概念が提案された1980年代と現在の間の状況の違いを理解する必要があります。1980年代までの日本の状況は、以下のように特徴づけられます。

- 追う立場、似た企業間での激しい競争→危機感
- 勤勉さや努力を重んじる価値観
- 多くの人に対する教育の機会の提供
- 現場の重視（実践的、チームワーク）
- 守るべきもの（既存のシステム、成功体験）が少なかった→型にはまらなかった
- 若手へのチャレンジと成長の場の提供

このような背景の中で、独創的な新製品を考案し、それを高品質、低コストの優れた製造技術で生産したのが日本の製造業の大きな成功要因だったと考えられます。

そのような状況は過去20年間で以下のように変化をしました。

- 製造技術や工場が海外に広まり、移転
- 価値観も仕事中心から家庭重視に変化
- さらなる競争の激化：追う立場、似た相手　→　追われる立場、異質（低コスト）な相手

これらの変化は、欧米でもすでに起きていることなので避けがたいものと言えるかもしれません。この「さらなる競争の激化」は、欧米でのアジャイル開発普及の大きな原動力になっていると考えられます。

また、以下のような変化も生じたと考えられます。

- 守るべきもの（既存のシステム、成功体験）の増加　→　事なかれ主義
- 若手へのチャレンジと成長の場の提供の減少

このような「事なかれ主義」と「チャレンジと成長の場の減少」は、「組織的知識創造」を阻害するものであり、ひいては「新たな可能性（ビジネスチャンス）への挑戦」を困難にします。また、過去 20 年間のさらなる競争の激化により「ビジネスや技術の変化」のスピードが 20 年前よりも大幅に速まっています。そのような「ビジネスや技術の変化」のスピードアップにより「チャレンジと成長の場」は若手だけに必要なものではなく、ベテラン社員にも必要になっています。これらをこのまま放置すると、時代の変化に対応できなくなり、ビジネス競争から脱落していくことになります。また、これらは「IT 業務」の革新を阻み、「アジャイル開発」の導入を阻むものでもあります。

厳しいビジネス競争で勝ち残るためには、「事なかれ主義」を排除し、若手とベテランの両方に「チャレンジと成長の場」を提供して「組織的知識創造」を活性化することが今後求められます。そのような取り組みを行う際に、「組織的知識創造」を具体化したものとしてスクラムのフレームワークを活用することができます。

5. 最後に

近年、欧米ではアジャイル開発がプロジェクトレベルで使われるのは当たり前になりつつあり、企業全体のアジャイル化ということがホットな話題になりつつあります。そのような企業全体のアジャイル化を行う上で以下のような点が強調されています。

- 「アジャイル開発を行う」ことが目的でなく、「顧客満足度を高めてビジネス競争に勝つ」という目的を忘れてはならない
- そのような目的を達成するために開発のやり方を改善する際に、「アジャイル宣言」を指針としたらよい

企業全体のアジャイル化を行う際には、現状の開発のやり方がマチマチの組織がスクラムや XP などの書籍に書いてあるとおりの開発方法を一気に取り入れられる訳ではありません。現実的には現状の開発のやり方を少しずつアジャイル化していくことになります。このように段階的にはアジャイル化を行う際に、これらのアドバイスを念頭に置くことで大きな方向性を見失うことを防ぐことができるでしょう。

6. 参考文献

[1] Scott W. Ambler, John Nalbone, Michael J. Vizdos, エンタープライズ統一プロセス, 翔泳社, 2006

[2] 平山輝, 宗平順己, 明神知, 大場克哉, 池田大, 今井英貴, 谷上和幸, 百年アーキテクチャ – 持続可能な情報システムの条件, 2010

[3] 森雅俊, 宗平順己, 左川聡, ビジネスモデル設計のための UML 活用 – 企業改革とシステム構築へのアプローチ, 毎日コミュニケーションズ, 2006

[4] エレン・ゴッデスディーナー, 実践ソフトウェア要求ハンドブック, 翔泳社, 2009

[5] ケン・シュエイバー, マイク・ビードル, 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, 2003

[6] 野中郁次郎, 竹内広高, 知識創造企業, 東洋経済新報社, 1996

事例1 アジャイル実践事例 ～一括請負受託開発への適用～

株式会社オージス総研　ソリューション開発本部
エンタープライズソリューション第三部 入江 茂喜

アジャイル実践事例
～一括請負受託開発への適用～

1. はじめに

エンタープライズ向けのシステム受託開発では、予算や納期の制約をクリアすることが重要な課題です。しかし、予算や納期をクリアしたとしても、ユーザニーズに合わないシステムが出来上がることもしばしば起こります。当事例では、計画駆動の確実性、安定性と、アジャイル開発の柔軟性、適応性の両者を活かし、予算、納期を厳守しつつ、よりユーザニーズに応えるシステムを開発した事例としてご紹介いたします。(筆者は、プロジェクトマネージャおよび要件定義担当として参画。)

2. プロジェクト概要

プロジェクトは、PowerBuilder と COBOL で構築されたクライアント・サーバー型の旧システムを再構築する目的で開始されました。旧システムは改修に次ぐ改修で、プログラムの有効ステップ数は 27 万ステップを越える規模まで膨れ上がっており、それでもビジネスニーズとのアンマッチが発生している状況。そのため、これ以上の改修は不可能と判断され、また H/W 老朽化も問題であったため、再構築することとなりました。

再構築に当たり、お客様の RFP（提案依頼書）で提示された主なプロジェクト要件は下記のとおり。

- 業務が滞りなく遂行可能な旧システム同等機能を備え、かつ、使い勝手を向上させてスリム化
- 主要な連携他システムのリリースが約 9 ヶ月後に予定されており、新システムも同時にリリース
- C/S 型システムから Web システムに変更。オープンソースソフトウェアを活用し、言語を統一
- 保守運用はお客様自身が実施。一括請負によるシステム構築のみが契約範囲

3. 開発手法　～アジャイル ＋UP（統一プロセス）～

旧システムの規模から換算すると、完全に再構築する場合、約 200 人月はかかると試算されましたが、お客様の IT 予算枠の制限、また、機能をスリム化するという目的もあったため、使える人的リソースは半分の約 100 人月が上限でした。さらに、要件定義～リリースまで 8 ヵ月強という工期は、JUAS の標準工期[1]と比較すると約 30%短縮しなければならないという短納期でした。

よって、短い工期の間に徐々に機能を洗練してゆくアジャイル開発、特に自社に事例があったスクラム[2]を適用することを計画。さらに、要件定義含めた一括請負というリスクを軽減するため、アーキテクチャ中心、リスク駆動といったプロジェクトを安定化させるプラクティスを UP (Unified Process)[3] から取り入れて、ハイブリットな手法とすることに決めました。また、開発中では、スクラムのみでは技術的プラクティスが足りないと感じ、テスト自動化、継続的インテグレーションなどの技術的プラクティスを XP (eXtreme Programming)[4]から取り込みました。

表 1　アジャイルと UP のハイブリッド開発手法

開発手法	採用したプラクティス	狙い
UP (Unified Process)	・アーキテクチャ中心 ・リスク駆動	早期にアーキテクチャを安定させリスクを低減
アジャイル開発 (スクラムおよび XP)	・スプリント計画、スプリントレビュー ・自己組織化チーム、デイリースクラム ・テスト自動化、継続的インテグレーション ・シンプルな設計、リファクタリング	柔軟に仕様変更を取り込みつつ、短納期を実現させる
UP およびアジャイル	・インクリメンタルな反復型プロセス（UP） ・反復型でフィーチャ（機能）ベースの提供（ｱｼﾞｬｲﾙ）	反復毎にユーザーにレビューし、フィードバックを貰う

4. プロジェクト計画

マスタースケジュールは UP の４つのフェーズを意識して計画し、作成フェーズのみを反復することにしました。推敲フェーズとして要件定義、外部・基本設計を 1 ヵ月強、作成フェーズとして反復開発を 5 ヶ月、移行フェーズとしてシステムテスト・移行に 2 ヵ月という計画でした。（お客様の RFP によって方向性は定義されていたため、方向付けフェーズは設けず。）

反復開発はスクラムのスプリントを参考にし、1 ヶ月ごとのタイムボックス型開発を採用。各反復終了時に、反復で実装した実際に稼動するシステムをお客様の環境にセッ

トアップし、レビューしながら機能を確認して頂き、改善要望などのフィードバックを頂くことを計画しました。

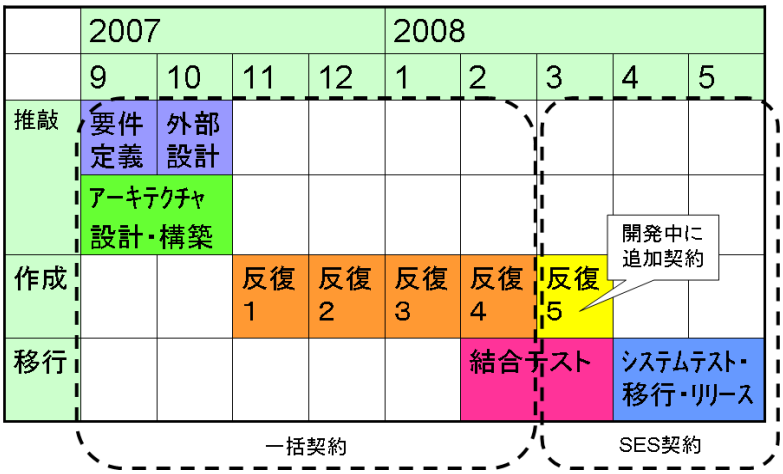

図 1　マスタースケジュール

5.　全体反復計画

各反復で実装する機能を前もって決めることはせず、機能一覧の中から、ストーリーとしてユーザーに意味ある機能のかたまり、業務的優先度、リスクなどを考慮して、実装する機能や作業タスクを各反復の開始時に計画することとしました。

また、UP のような作業分配（反復が進むごとに、作業の割合を要求→分析→設計→実装→テストと順に増やす）を意識し、各反復で開発する機能数（TFP：トランザクションファンクションポイント）が、レイリー曲線的な山型になるように全体で計画。この全体計画によって、各反復計画時に、前半は設計に注力したタスクが組みやすく、後半はテストに注力するタスクが組みやすくなることを意図しました。

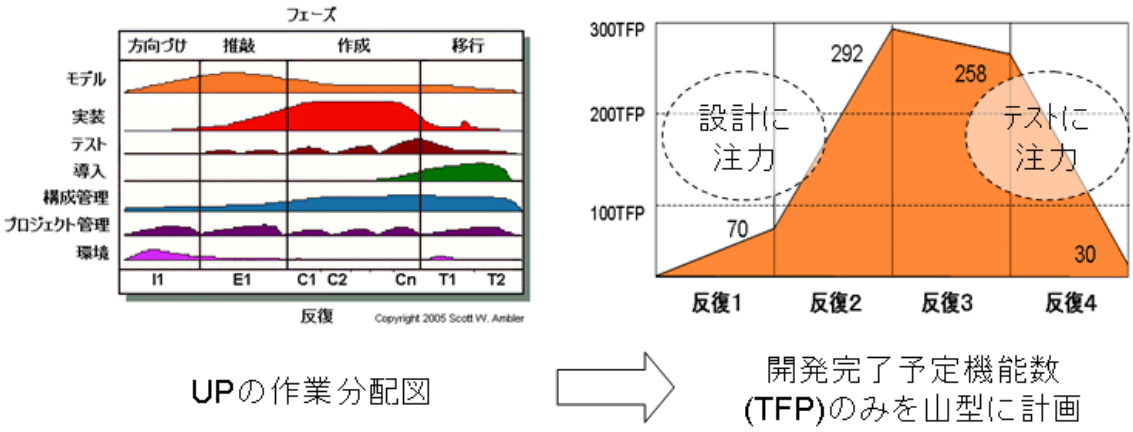

図 2　UP の作業分配図と今回の全体反復計画

6. スコープ管理、全体進捗管理

FP（ファンクションポイント）を元に見積もりを行ったという理由もあり、仕様変更などが発生した場合は、FP を都度算出し、お客様とスコープ調整を実施しました。ただし、”要件定義で決めた機能の実現を約束する”のではなく、”要件定義で決めた FP は固定するが、システムが基本要求を満たし、より良くなるのなら最初に決めた機能に固執しない”ことを最初にお客様と合意し、途中で新しいアイデアが生まれ、機能が追加された場合でも、重要度の低い機能と入れ替えで取り込むことを基本姿勢としました。

全体の進捗管理も、当初は WBS(Work Breakdown Structure)を元に管理していましたが、反復開発が開始されたあたりから、計画した予定 TFP(図 2 参照)と完了 TFP の予実差で進捗を管理しました。

7. 推敲フェーズ　～要件定義、外部設計、基本設計の実施～

推敲フェーズでは、機能仕様やアーキテクチャを（感覚的には）全体の 6～7 割程度まで確定する姿勢で実施し、システムのベースラインを作成しました。

7.1. 要件定義

再構築ですので、まずは旧システムを分析することから要件定義を開始しました。旧システムは規模も大きく複雑で、新たな改善要望もあったため 1 ヵ月では時間が足らず、予想どおり、確定できた仕様は全体の 7 割程度でした。主な成果物は、要件定義書、機能概要、アクティビティ図によるシステム化業務フロー。どの成果物も納品対象でしたので、“使い捨て”はしませんでしたが、出来るだけ簡素な記述で作成することを心がけ、特に要件定義書は、USDM（Universal Specification Description Manner）表記法[5]を採用し、変更し易いフォーマットで作成しました。

7.2. 外部設計

画面、帳票設計は、粗いスケッチ程度のドキュメンテーションに止めて、後に変更が発生してもドキュメントは修正せずに、JSP や PDF レイアウトなどの製造成果物を直接修正しました。ただし、複雑で操作性が重要な画面は HTML で使い捨てのモックを作成し、お客様にレビューして事前確認を行いました。他システムとのインターフェース設計については、そもそも他システムも構築中であったため仕様が確定で

きず、反復開発中まで仕様検討と設計を継続しました。

7.3. 基本設計・アーキテクチャ構築

アーキテクチャ設計・構築と論理 DB 設計も同時に行いました。Seasar2 ベースの既存自社フレームワーク（OJF: OGIS Java Framework）をカスタマイズし、非機能系の共通コンポーネントを実装。DB は PostgreSQL を採用し、各外部設計担当者の要望を集めて、専任の DB モデラーが論理 DB まで設計しました。これらも 1 ヵ月では完了できるボリュームでなかったため、反復開発中は、当初予定していなかったアーキテクチャチームを編成し、各反復で引き続き、非機能系の設計・実装、物理テーブル設計を行いました。

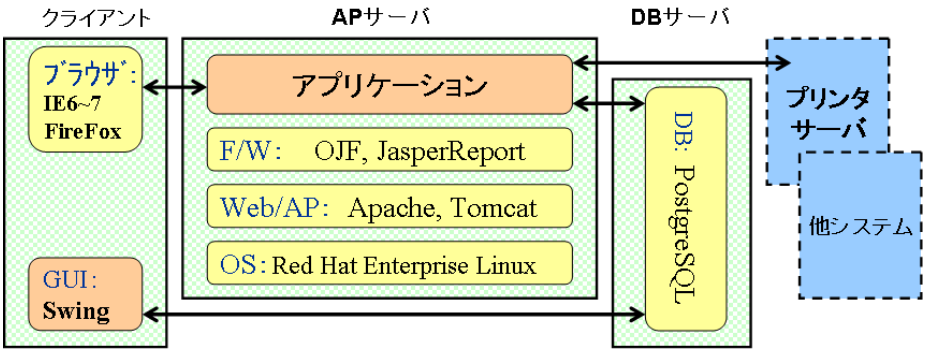

図 3　アーキテクチャ概要

8. 作成フェーズ　～各反復の実施～

作成フェーズでは、いよいよ本格的にアジャイル型の反復開発を実施しました。各反復内のプロセスは、スクラムを参考にし、下記のフローを実践しました。

- <u>反復計画</u>

お客様が重要付けした機能一覧、課題一覧を、スクラムで言うところのプロダクトバックログとして扱い、PM および各チームリーダーがその中から当該反復で何を開発するか決定し、作業タスクまで落とし込む。作業タスクは時間単位で開発者自身が見積もりし、チームリソース（理想時間（6h /日）×作業日数×チーム人数）に収まるタスクを作業範囲とする。ただし、純粋なスクラムとは異なり、前述した全体反復計画の完了予定 TFP を達成目標とした。

- **毎朝、毎昼のスタンドアップミーティング**

毎朝、各チームでスタンドアップミーティングを実施し、各自、前日実績、本日予定、課題問題点を報告して情報を共有。また、チーム間の情報共有として、毎日昼会を実施し、リーダー間でも短いミーティングを行った。当時、特に意識はしていなかったが、これがスクラムで言うところの Scrum of Scrums と同等な機能を持ったと考えられる。

- **顧客レビュー**

詳しくは後述の「8.5. 各反復の終了～顧客レビュー～」を参照。

- **ふりかえり**

各チームで、反復中うまくいったこと、改善が必要なことを検討する。他チームでうまくいったやり方を取り込むなど、チーム間でもノウハウを共有した。

実際の開発作業ですが、フレームワークに付随する DDL 生成ツール、コード生成ツールや自動テストツール等を活用したため、大まかな作業フローは決まっていました。この点では、UP のアーキテクチャ中心のプラクティスが活きて、ある程度、プロセスの安定性を得ることができました。

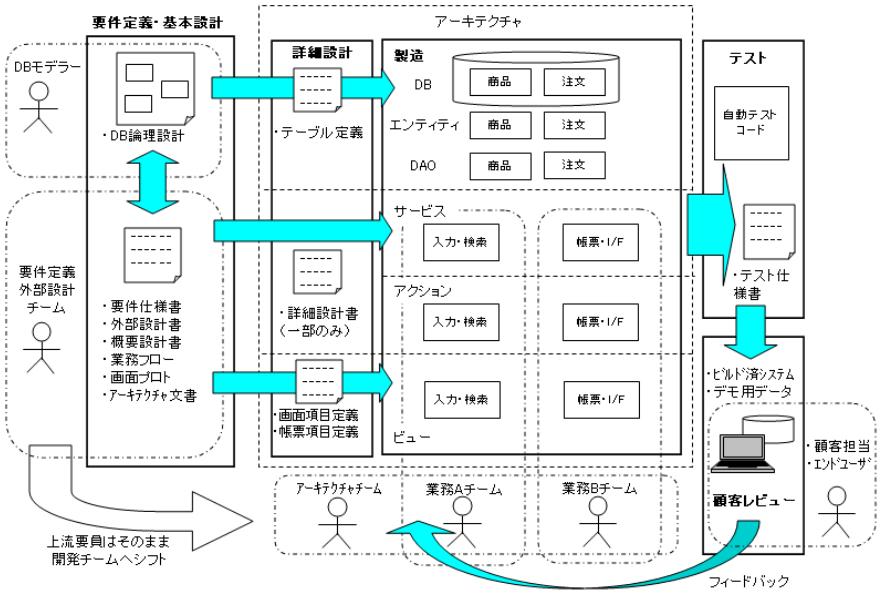

図 4 作業フロー

8.1. 各反復の特徴

下記表が各反復の特徴（計画時に設定したゴールと実際の実施内容）です。前述したように、各反復で何を開発するのか、事前に決めていたわけではありません。しかし、例えば反復１で“最初にアーキテクチャを横断する小さな機能をまず開発し、作業の感触をつかもう”、“仕様が確定してないが、ユーザーのフィードバックを得るため、早めに一部実装してレビューして頂こう”など、自然と全体を意識した戦略的な計画も盛り込まれました。

また、スクラムのプラクティスに則り、反復中はユーザー変更要望を受け付けないことが原則でしたが、設計変更が避けられないような情報（例えば他システム I/F 仕様が確定した等）を反復中に得た場合は、“今の反復で設計を変えたほうがよい”と判断して、作業タスクを変更することもありました。

このように、その都度得られた最新の情報、判断を基に、戦略や予測を含んだ計画と臨機応変な適応行動を織り交ぜて、各反復を進めていきました。

表 2　各反復の特徴

反復	各反復の特徴
反復１	・アーキテクチャの全要素を横断するログイン機能を開発し、作業と技術のベースを確立 ・重要度Ａ、複雑で、仕様曖昧な機能の一部を開発し、フィードバックを得る
反復２	・反復１で実装した重要度Ａ機能に対するフィードバックを反映し洗練 ・他の重要度Ａ、Ｂの機能を開発
反復３	・反復２で得たフィードバックの反映と仕様変更を取り込む ・残りの重要度Ａ、Ｂの機能を全て開発
反復４	・反復３で得たフィードバックの反映と仕様変更を取り込む ・残りの重要度Ｃの機能を全て開発し、一括請負分の納品に向けた包括的なテストを実施
反復５ （追加契約）	・追加機能の開発 ・仕様変更を取り込み、運用機能を洗練

8.2. 自律的、自己組織化チーム

反復開発中のプロジェクト全体ピーク要員数は 19 人。PM、PMO がトップレベルの管理を担い、その下に業務機能開発チームＡ、チームＢ、そしてアーキテクチャチ

ームの３チームで体制を構成しました。各チームは１名のリーダーと 3～7 名のメンバーで構成され、基本的に各メンバーは設計、実装、テストを一貫して実施しました。

大枠の方針は PM が明示して厳守させましたが、各チームの方針も尊重し、チーム毎に最適な管理方法、作業方法の採用を認めました。各チームは様々な手法を駆使しましたが、代表的な例は以下のとおりです。

- **業務機能開発チームＡ**

  別名、画面チームと呼ばれ、複雑な画面機能を主に担当。最も仕様が不明確だったため、テストに重点を置き、仕様変更に対応できるようにテスト自動化を推進。頻繁な変更要望に対しては、先にテスト仕様を作り、そのケース消化を管理して、仕様モレを防ぐなどの特徴を持っていました。

- **業務機能開発チームＢ**

  別名、帳票・I/F チームと呼ばれ、こちらは複雑なバッチ処理を多く引き受けました。チームＡと異なり、内部設計が不安定で作業ボリュームが予測し辛いため、毎日、残作業時間と残チームリソース（人時）のグラフをバーンダウンチャートとして可視化しました。バーンダウンチャートを採用したのはこのチームが最初で、その後、他のチームも採用しました。

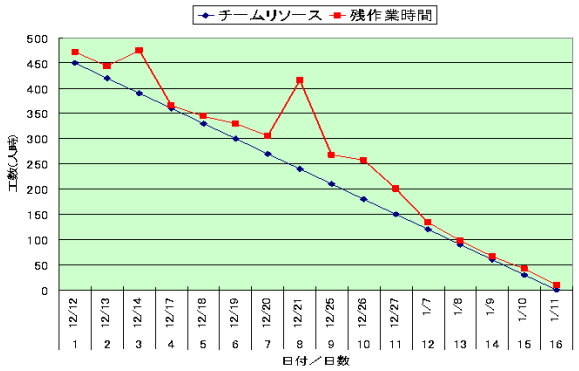

図 5 バーンダウンチャート

- **アーキテクチャチーム**

  アーキテクチャチームは、非機能系のコンポーネント開発、環境構築を主に担当。また、他チームからのテーブル変更要請を受けて、それに関わる DDL、DAO、Entity のコード自動生成と管理を受け持ちました。このチームは、他のチームからの依頼が不定期にあがってくるため、手書きのタスクかんばんボートでタスクを可視化す

るやり方を途中から実施しました。

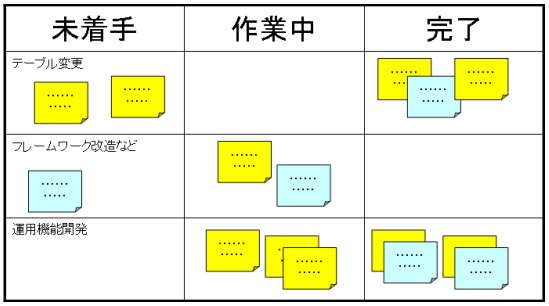

図 6 タスクかんばんボード

開発中、チームを横断する課題が発生したときは、タスクフォース的な小チームを編成することもありました。また、ある特定作業が得意なメンバーが他のチームを支援するなど、メンバーの役割は自己組織的に、ダイナミックに変わりながら、チームの垣根を越えて協力することも多々ありました。

8.3. コミュニケーション

- 暗黙知と形式知のバランス

暗黙知と形式知のバランスは意識的にコントロールしました。永続的で核となる仕様や設計、手順的な「How」はドキュメント化し、一時的で文書化し辛い「Why」は、頻繁に対話することでプロジェクトに浸透させました。暗黙知を意識的に作り上げたことはプロジェクトの推進に大きく貢献しました。特に、” 何故この機能が業務に必要か”、”何故このような設計になっているか”といった「Why」を各自が理解することで、システムと業務の理解が深まり、仕様誤認ミスを減らし、後発的な仕様のアイデアも生まれるといった効果がありました。

- オープンワークスペースとコミュニケーション

専有プロジェクトルームは必要不可欠でした。毎朝、毎昼のスタンドアップミーティングをその場で実施し、進捗状況や課題を共有。入り口付近に、バーンダウンチャートやタスクかんばんボードが毎日張り替えられ、進捗を見える化。３つのホワイトボードを備え、設計の議論、障害対応策などが行われ、一つの場所で議論することで、自然と暗黙知もプロジェクト全体に広がりました。

ただ、コミュニケーションのオーバーヘッドが無い代わりに、うるさくて集中できないデメリットもあります。よって、開発前半は積極的に対話することを推奨しまし

たが、後半はコードやテストにじっくりと向き合えるよう、対話の頻度を減らすことを指示しました。

8.4. 技術的プラクティス

スクラムは主に管理をカバーするプラクティスの集まりなため、XP などから技術的プラクティスを取り入れて実施しました。

- テスト自動化

  完全な TDD（テスト駆動開発）やテストファーストではありませんでしたが、テストの自動化は徹底しました。ここはフレームワークが提供するテストツールがテスト効率化に大きく貢献しました。実は途中、開発者がテスト自動化に熱中しすぎている様子を感じ、”自動化はほどほどにし、他に注力すべきことがあるのでは？”と問いかけましたが、チームリーダーは断固反対。結果的に、反復毎に大幅な仕様変更を受け入れることが出来たのは、この自動化された回帰テストコードがあったおかげです。

- シンプルな設計、設計の改善

  機能が最低限実現できる程度のシンプルな設計から始めました。例えば、処理に時間がかかる一括データ更新機能は、初めはボタンを押して完了を待つような同期処理でしたが、非同期処理にして、後からユーザーが結果を確認できるような機能に変更しました。

  また、反復を重ねる毎に、継続的にコードをリファクタリングして改善しました。これもテスト自動化とあわせて、仕様変更を受け入れ易い下地を作ることに貢献しました。リファクタリングにより、後半はコード量が減っていく現象も観測されました。

- 継続的インテグレーション

  １ヵ月毎に動くシステムを提供するため、頻繁なビルドと自動回帰テストを実施し、完全にコードが統合された状態を保つことが必要不可欠でした。そのため、自動化されたビルドスクリプトを整備し、Subversion からチェックアウトしてスク

リプトを動かすだけで、完全に動くシステムがどの PC 環境でも手に入れるようにしました。PM の筆者自身が要件定義担当でもあったため、仕様を正しく実装しているか都度確認するのに非常に役に立ちました。また、ソースコード以外にも、デモンストレーション用のための整合性が取れたデータ、そしてドキュメントも全て Subversion で一元管理していました。

- ペアプログラミング

  ペアプログラミングは一部メンバーのみ限定的に実施しました。これは、Java、Seasar2 に不慣れな開発者にベテランが指導するといった教育的意味での実施でしたが、技術力の底上げもさることながら、不慣れな開発者が担当したコードの品質も高まりましたので、プロジェクト全体としても有意義でした。

8.5. 各反復の終了　～顧客レビュー ～

反復が終わる毎に、お客様を訪問して開発した機能をレビューしました。レビューの内容は、開発者による数時間のデモンストレーションと、お客様開発担当者とエンドユーザーからの質疑応答と改善要望のヒアリングです。動くシステムが目の前にあることの効果は絶大で、要件定義や外部設計レビューでは出てこなかった具体的な要望が合計で 100 件近く引き出すことができました。また、未決定だった仕様もレビューを通じて徐々に固めることができました。

また、お客様の PC 環境にデモンストレーション環境を短い時間で構築しなければならないため、デモ用のデータとアプリケーションの移行をスムースに行えるように、インストーラーや移行作業自体が毎回改善されていく、といった副次的効果もありました。

そして、セッティングしたデモンストレーション環境を、お客様方にそのまま置いておくことで、継続的にシステムに触って頂き、レビュー時以外に気付いたことも都度フィードバックして頂くことができました。

9. その他実施したこと

- 要員教育

  ほとんどの要員が、Java の経験はあっても、フレームワークや DBMS の PostgreSQL は未経験でした。よって、メイン開発者に、社外、社内の技術講習を受講させました。今回、アジャイル開発は密なコミュニケーションによって新技術

へのキャッチアップも早い、ということを実感しましたが、初期にスキル水準を迅速に上げるためには、やはり計画的に教育を組み込むことが有効だったと思います。

- **数値による品質管理、進捗管理**

  従来のように数値目標を設定する品質管理・進捗管理も実施しました。開発中は、テストカバレッジ率、テスト消化率、テスト密度、バグ発生率、日次の有効ステップ数測定、完了済み TFP などを計測し続けました。ただしこれは、あくまで問題を早期に発見するため、システムの成長度、成熟度の指標として使用するに止め、必達目標にはしませんでした。また、上位の社内管理者やお客様の管理者にとって、アジャイル開発は馴染みがなかったため、従来の数値管理を併用することで、プロジェクトのアカウンタビリティを補完できたと思います。

10. メトリクスで見る実績

参考のため、プロジェクトに関するメトリクス実績を図 7 に記載します。それぞれ、細かく解説はしませんが、納期は計画どおり達成し、生産性も計画を上回る実績になりました。また、品質面でも、特にシステムテスト・リリース後の障害件数はほぼ計画どおりであり、一括請負開発後、保守・運用はお客様へ引き継いだのですが、現在に至るまで瑕疵は発生していません。

プロジェクト計画:実績

	計画	実績
納期(リリース日)	2008年6月	2008年6月
投入要員(人月) (※1)	78.5	95.8
顧客満足度(※2)	4	4
生産性(FP/人月)	12.8	15
規模(有効ｽﾃｯﾌﾟ数)	-	16万7千

※1 途中で追加契約、※2 満点5

品質計画:実績

	計画	実績
単体テストケース数	530	1,600
UTカバレッジ(C0基準)	90%以上	80%
結合テストケース数	320	400
不具合件数(※1)	(予測せず)	63
障害件数(※2)	5	6

※1 結合テスト中の不具合、※2 ST、リリース後の顧客起票障害

図 7 プロジェクト計画・実績、品質計画・実績

また、開発の過程も、計画の戦略どおりの実績となりました。図 8 の A 付近では、思惑通り最初はアーキテクチャ等の内部コードが成長し、B 付近では、リファクタリングの効果のため機能の完了数が増加してもコードはあまり増加していません。そして、C 付近では、最初はテストが少なく、設計や実装に注力している様子を、D 付近では、テストコードの成長が本体コード上回り、テストに注力する様子が見て取られ、UP の

作業配分を意識した全体反復計画どおりにプロジェクトが進行したことを裏付けています。

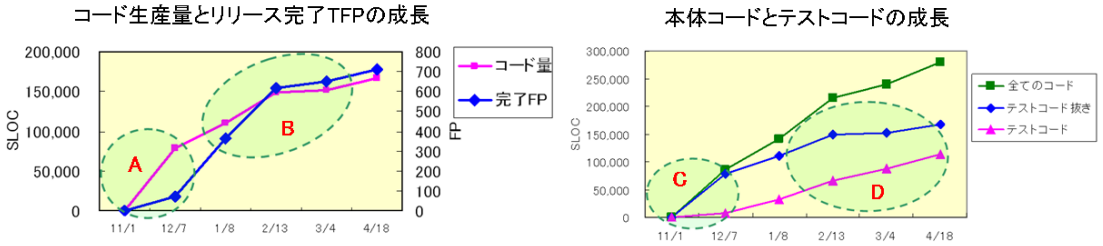

図 8 コード生産長とリリース完了 TFP、本体コード・テストコードの遷移

11. アジャイル+UP の効果と成功要因分析

- アジャイルの柔軟性がシステム価値を最大化

最も効果があったのは、ユーザーのフィードバックを果敢に取り込み、納期、予算が限られた中で、ユーザーにとって最も価値あるシステムを提供できたことだと思います。極端に言えば、このシステムをウォーターフォール型で開発していたなら、納期、予算の制約はクリアできたとしても、システム自体が全く異なったものになったのは間違いありません。前述したように、開発中に発生した 100 件近くの改善要望は、ウォーターフォール型ならば、開発側は受け入れなかったでしょうし、そもそもユーザーが改善要望を開発中に見つけることはできなかったでしょう。

その点、アジャイル開発の様々なプラクティスは、小さな変更に対応可能な柔軟なチームとシステムの構築を可能にします。そして、小さな変更や要望を、反復毎に安全に取り込み、それを繰り返すことによって徐々にシステムの価値を最大化することができたのだと思います。

- UP 等の計画駆動がプロジェクトの制約をクリア

今回、短い納期、予算固定、一括請負契約という制約をクリアすることができたのは、アジャイルが持つ柔軟さを損なわない程度に、プロジェクトを安定させる全体計画を慎重に予測して組み込んだことが成功要因だと思います。具体的に言えば、UP のリスク駆動、アーキテクチャ中心、作業分配の概念を取り入れることで、短納期プロジェクトにとって致命的な初期の混乱を避け、早期に安定化させることができましたし、従来のプロジェクトマネージメントが得意としていた、リスク管理、スコープ管理、目標管理なども取り入れることで、計画と実績の乖離をコントロー

ルすることができたのだと思います。

- システムを作りだすシステム（すなわちプロジェクトチーム）の改善

  プロジェクトの成功の鍵は、やはりシステムそのものを改善する人、すなわちチームの作業自体がアジャイルの効果により改善されたことだと思います。短い反復で動くシステムをユーザーに提供することを目標にすると、早期に全ての工程を実際に経験することができ、まずいことがあれば、チームは以降の反復で自分たちのやり方を徐々に改善してゆきました。

  しかも、改善するのは全ての作業工程であるというのが重要です。テスト進捗が遅れがちだったら、テスト方法を見直しますし、反復後に顧客レビューするために、移行工程も改善します。また、変更要望を受け入れるためには、保守性を向上させなければならないことにチームは早い時点で気付きます。保守性は未来への投資ではなく、今の推進力に必須であると受け止めるからです。

  そして、最大の効果はメンバーのモチベーションが向上することでした。システムを顧客に都度レビューすることで、ほど良い緊張感と終わったあとの達成感を感じることでき、チームメンバーはその自信を糧に次ももっとうまくやろうと思うからです。

12. 最後に

当事例は、2007～2008 年に実施したプロジェクトでしたが、当時のアジャイル手法、特にスクラムが、製品開発向けの手法と考えられていたころの[6]、“どのように受託開発向けに適用するのか”という課題に対する一つの解決事例としてご紹介させていただきました。

近年、アジャイルは既にキャズムを超え、“アジャイル手法イコールずさんなやり方[7]”という初期の誤解は無くなりつつありますが、特に失敗が許されない受託開発への適用に不安を感じる企業はまだ多いと思われます。しかし、当事例のように、アジャイルの経験的プロセスを活かしながらも、従来的な計画駆動を組み込んで、受託開発を安全に遂行することは可能であると思います。

弊社では、当事例の経験を活かし、スクラムとアジャイル UP を組み合わせてブラッシュアップした OGIS Scalable Agile Method という開発手法を提唱[8]しており、アジャイル手法に興味があるが、実際の適用に躊躇している企業を支援することで、日本の

アジャイル開発の普及に貢献してゆきたいと考えます。

参考文献

[1] 日本情報システム・ユーザー協会（JUAS）が発表した式“標準工期 = 投入人月の立方根の 2.4 倍”, http://www.juas.or.jp/

この JUAS の工期の計算式は、Boehm 氏が"Boehm, B. W., Software Engineering Economics. Englewood Cliffs, N.J.:Prentice-Hall, 1981."で提案した COCOMO の式に基づくものである

[2] ケン・シュエイバー, マイク・ビードル: 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, 2003

[3] イヴァー・ヤコブソン他: UML による統一ソフトウェア開発プロセス, 翔泳社, 2000

[4] ケント・ベック: 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

[5] 清水吉男氏,: 要求を仕様化する技術・表現する技術, 技術評論社, 2005

[6] オージス総研 藤井拓, オブジェクトの広場・アジャイルモデリングへの道・第２回：スクラム組んで開発しよう！, http://www.ogis-ri.co.jp/otc/hiroba/technical/IntroASDooSquare/chapter2/IntroScrumCaseStudyMay2005.html

[7] クレーグ・ラーマン：初めてのアジャイル開発, 日経 BP 社, 2004

[8] オージス総研 藤井拓, オブジェクトの広場・OGIS Scalable Agile Method の真髄, http://www.ogis-ri.co.jp/otc/hiroba/technical/OSAMDistilled/OSAMDistilledPart1Sep2011.html

事例2 アジャイルソフトウェア開発におけるモデリングの有効性

株式会社オージス総研　技術部ソフトウェア工学センター*
藤井 拓　　鶴原谷 雅幸

*ソフトウェア工学センターは 2004 年当時の部署の名称です

アジャイルソフトウェア開発におけるモデリングの有効性

Effectiveness of Modeling in Agile Software Development

株式会社オージス総研　技術部ソフトウェア工学センター
Software Engineering Center, IT R&D Dept., OGIS-RI Co., Ltd.
〇藤井　拓　　鶴原谷　雅幸
Taku Fujii　　Masayuki Tsuruharaya

Three kinds of modeling activities are proposed to mitigate risks in software development and improve design qualities in agile software development; conceptual design modeling, collaborative detailed design modeling, and collaborative design refactoring. Since the proposed modeling activities are both lightweight and easy to learn, those activities are suitable to agile software development projects. This design technique was applied to an internal software development project, and improvements on design qualities are observed after collaborative detailed design modeling.

1. はじめに

近年，開発途上での顧客ニーズや技術の変化に対応しながら開発を行うことを目指すアジャイルな開発手法〔1〕と呼ばれるソフトウェア開発手法が提案されている．アジャイルな開発手法は，開発方法論の観点では要求，設計，実装，テストのサイクルを重ねながら開発を進める反復型開発アプローチの1種であるが，価値や原則やプラクティスで開発手法を定義している点で通常の反復型開発プロセスと異なる．本論文では，要求，設計，実装，テストを通じて動作するソフトウェアを作り上げる 1 サイクルを反復(iteration)と呼ぶ．

これらのアジャイルな開発手法の中で最も知名度が高いのが Beck らにより提案された XP（eXtreme Programming)〔2〕という手法である．XP では開発を進める上でのプラクティスを複数定義しているが，特に設計に関するプラクティスとしては以下のようなものを提案している．

- シンプルな設計
- 設計改善
- メタファ

XP では，このような設計のプラクティスを他のプラクティスとともに適用することで，開発されたソフトウェアの変更コストが時間とともに増大することを抑制できると主張している．

オブジェクト指向設計方法論の主張から考えると，XP が主張するようなソフトウェアの変更コストの抑制を実現するためには，開発成果物の設計品質が高いことが求められる．しかしながら，XP ではプラクティスで提唱している以上のモデリング作業の実践方法について言及しておらず，高い設計品質が実現できるために必要

技術部ソフトウェア工学センター　〒560-0083 大阪府豊中市新千里西町 1-2-1 TEL: 06-6871-7993
IT R&D Dept., Software Engineering Center 1-2-1 Shinsenri-Nishimati, Toyonaka, Osaka

表 1 XP の提唱する3つの設計プラクティスとその意味

プラクティス	意味
シンプルな設計	将来の拡張を見越した複雑な設計を行わず，現在の反復で目標とされている機能だけを実現するシンプルな設計を推奨すること
設計改善	重複したプログラムコードが複数箇所で現れた場合にそれを継承で一本化したり，可読性が悪いプログラムコードをメソッドの分割や再配置で改良すること
メタファ	アーキテクチャなどの設計概念を共有するために，それらのメタファを用いること

なモデリングスキルや経験等の前提条件は示されていない．

筆者らは，XP を始めとするアジャイル開発手法に軽量なモデリング手法を導入することで開発の伴うリスクを早期に軽減したり，変更コスト抑制の前提となる設計品質の改善が達成できると考えた．さらに，このモデリング手法をなるべく平易なものにすることでモデリング経験が少ないメンバーでも簡単に取り入れることが可能になると考えた．

本論文では，まず XP の設計に関するプラクティスについて説明する．ついで，オブジェクト指向設計の方法論の立場から考えた設計品質と設計品質の測定方法について説明する．さらに，筆者らが提案する設計手法とその期待効果について説明する．最後に，筆者らが提案する設計手法をモデリング経験が少ないメンバーからなる開発プロジェクトに適用した結果，設計品質がどのように変化したかという点を報告する．

2. 設計のプラクティスと設計品質

2.1. XP の設計に関するプラクティス

前節で言及した XP の提唱する設計に関する3つのプラクティスの意味をより詳しく説明したものが，表 1 である．これら3つのプラクティスのうち前者2つは密接に関連しており，予測に基づいたソフトウェアの設計を慎み，重複したプログラムコードが現れるなど実害が発生してから設計を改善すれば良いという設計思想を表している．このような設計改善は，一般的には対象となるコード部分あるいはソフトウェア全体の提供機能を保ったまま設計及びプログラムコードだけを改善するものであり，リファクタリング(refactoring)〔3〕と呼ばれている．

また，XP では開発に際して JUnit に代表される単体テスト自動化フレームワークにより製品本体のプログラムコードの単体テストを自動化することが推奨されている．製品本体のプログラムコードと自動化された回帰テストスイートが同時並行して開発されることにより，開発の任意の時点でリファクタリングのような設計やプログラムコードの改善を安全に行うことができる．

このようなプラクティスから成る XP を適用するメリットとして Beck が主張しているのは， 図 1 に示されるように変更コストを時間とともに飽和させることができるという点である． 著者らは，このような変更コストの飽和が実現するためには，以下の 2 点が必要だと考えた．

- 変更が局所化するような設計構造
- 変更後に低コストで動作検証が行える仕組み

これら 2 点のうちで後者は単体テストやシステムテストの自動化を行うことにより実現できると考えられる．一方，

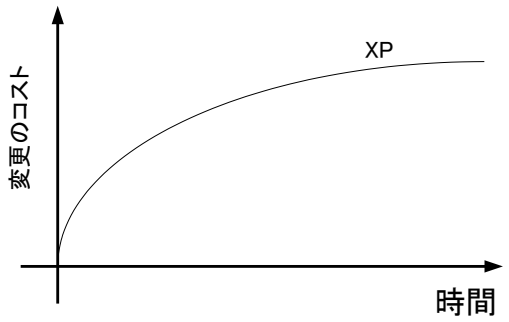

図 1 XP を適用した場合の変更コスト

前者が実現されるためには設計の基本単位であるクラスのレベルで良好なモジュール化が行われている必要がある．すなわち，オブジェクト指向設計の観点で高い設計品質に至っている必要があると考えられる．

クラス名：	
サブクラス：	
スーパークラス：	
責務	協力オブジェクト

図 3 CRC カードの形式

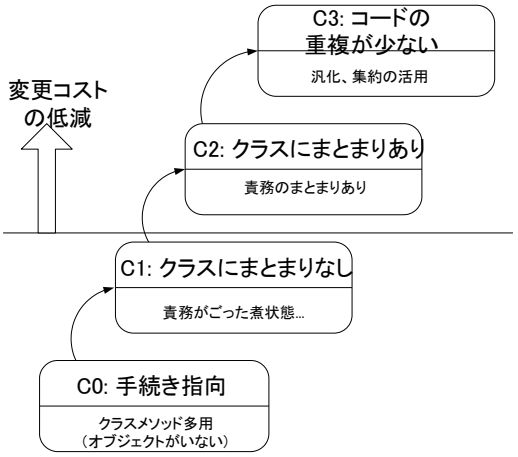

図 2 クラスの設計品質のレベル

2.2. 責務主導設計の観点での設計品質

オブジェクト指向設計の基本的な考え方として歴史的に定評があるのは，Wirfs-Brock により提案された責務主導設計(Responsibility Driven Design)手法〔4〕である．責務主導型設計は，図 3 に示される CRC(Class Responsibility Collaborator)カードを用いてクラスを責務の集合と他のクラスとの協調を中心に設計モデルを考える手法である．CRC カードは，クラス(class)毎に作成され，そのクラスが他のクラスに提供するサービスを責務(responsibility)として，責務を果たすのに必要な他のクラスが協力オブジェクト(collaborator)として記入するものである．責務は，一般的に各クラスの操作に対応する．このようなカードを用いることで責務主導設計が目指す各クラスが適度な数のまとまりのある責務を持ち，ソフトウェアの機能が責務の種類が異なる複数のクラスの協調として実現する良好な設計モデルの作成を促すことができる．また，CRC カードはグループワークに適しており，グループワークを通じて設計を行うことにより設計に対する共通の理解を形成したり，設計スキルの向上を達成できると期待できる．

以上説明したような責務主導設計の立場から考えた設計品質のレベルをまとめたものが図 2 である．この図にはC0からC3までのクラスの設計品質レベルが示されている．責務主導設計の観点で最低限許容できる設計品質のレベルは C2 だと考えられるため，著者らは C2 を変更コストが抑制されるための必要条件だと考えた．

2.3. 設計品質の測定手法

本研究では，前節で説明した C0 及び C1 に相当する設計部分を表 2 に示されるいくつかの測定量を用いて，それら測定量の閾値を超えるかどうかにより検出を試みた．

C0 の検出に用いた McCabe のサイクロマティック複雑度(CC)〔5〕は，メソッド内の制御フローのノード数(n)とエッジ数(e)より以下の表で求まる測定量である．

$$CC = e - n + 2$$

CC は，元々は分岐複雑度を定量化する測定量であるが，オブジェクト指向設計やプログラムが手続き的である度合いを示す指標として用いられている．これは，複雑な責務を複数のより小さな責務や協調に分解せず 1 つのメソッドに実装すれば結果として分岐複雑度が高まるとの考えに基づいている．メソッドの命令行数(NOS)も同様の理由で手続き的なメソッドの検出に用いられる．また，クラス操作は一般的に手続きとデータを

表 2　C0 及び C1 に相当する設計品質の部分の検出方法

測定対象	測定手段	閾値
手続き型のコード(C0)	メソッドの命令行数 (NOS: Number Of Statements)	20
	McCabe のサイクロマティック複雑度 (CC: Cyclomatic Complexity)	4
	クラスメソッド数 (CM: Class Method)	
クラスのまとまりのなさ(C1)	クラスのメソッド間凝集度の欠如 (LCOM: Lack of Cohesion Of Methods)	40
	他のクラスに対する関係数 (Ce: Efferent Coupling)	25

モジュール化したインスタンスを基本とするオブジェクト指向の考えから外れるものであり，クラス操作の割合の高さは手続き的であるという度合いを示していると考えた.

一方，C1 の検出のためには，クラスのメソッド間凝集度の欠如(LCOM)と他のクラスに対する関係数(Ce)を用いた. LCOM は，属性とメソッドの対応関係から 1 つのクラス内のメソッド間の凝集度の欠如を定量化する測定量である. LCOM については複数の測定方法が提案されているが，本研究では Henderson-Sellers の提案した以下の計算式〔5〕に基づく測定方法を用いた.

$$LCOM - HS = \frac{\left(\frac{1}{a}\sum_{j=1}^{a}\mu(Aj)\right) - m}{1 - m}$$

m: メソッド数, Aj: j 番目の属性, a: 属性数

$\mu(Aj)$: j 番目の属性にアクセスするメソッド数を求める関数

Ce は，各クラスの他のクラスへの関係による参照数を測定したものであり，協調が多すぎて蛸足状態のクラスを特定するために有効な測定量である．蛸足状態のクラスは，変更の影響を広範囲に波及させる可能性が高いため，変更コストを抑制する設計である基本的な条件だと考えられる.

本研究では，表 2 の C0 に対する各測定量において閾値が示されているものについては閾値を越えたメソッド数を計測し，それらのメソッド数のプログラムコード全体でのメソッド数に対する割合を求めることによりプログラムコードの C0 に相当する部分の割合の定量化を行った．クラスメソッド数については，直接プログラムコード全体でのメソッド数に対する割合を求めた．また, C1 の各測定量については，測定量が閾値を超えたクラス数のプログラム全体のクラス数に対する割合を求めることによりプログラムコードの C1 に相当する部分の割合の定量化を行った.

Java プログラムが測定対象の場合，表 2 に挙げられているクラスメソッド数以外の測定量は無償提供されている Java の統合開発環境である Eclipse の中で無償提供されている Eclipse Metrics Plugin〔6〕により求めることができる．また，表 2の測定量に対する閾値は Eclipse Metrics Plugin において初期設定されている値を採用している．

3. 本研究で提案する開発フェーズと設計手法

表 3 本研究で設定した4つの開発フェーズ

開発フェーズ名	開発フェーズの目的
方向付け	初期要求の策定と検証，技術調査
推敲	アーキテクチャの確定
作成	機能の作りこみ
移行	ソフトウェアの評価，改良，導入成果物等の作成

筆者らは，アジャイルな開発においても統一プロセス〔7〕のように複数のフェーズを設けることがプロジェクトのリスクを管理したり，作業の労力配分を制御したりするために有効ではないかと考えた．そのため，統一プロセスのフェーズを参考にして表 3 に示されるような4つのフェーズを開発ライフサイクルに設定した．方向付けフェーズでは，初期要求の策定と検証，アーキテクチャに使われる技術候補の選定のために必要な期間実施され，開発対象のソフトウェアの大雑把な機能及びその実現に使う技術を確定させることを目指す．推敲フェーズ以降では，フェーズの目的の達成を長期目標として，基本的に 2 週間単位の反復で追加すべき機能やその他の目標を設定し，プログラムの拡充及びテストを行う．開発要員の点では，統一プロセスと同様に開発に伴うリスクが高い方向付けフェーズではなるべく少人数でスタートし，以降のフェーズでリスクを軽減しながら順次開発者を増員していくことにより開発に伴うリスクと投入する費用とのバランスを取ることができる．なお，各反復での目標設定等の作業の進め方については，文献〔8〕に記した方法を用いている．

筆者らは，このような 4 フェーズに基づくアジャイルソフトウェア開発方式において開発のリスクの早期軽減と設計品質の改善を行うために表 4 のような 3 種類のモデリング作業を考案した．これらのモデリング作業の考案にあたり，重視したのは以下の 3 点である．

- 軽量さ
  - 2 週間というアジャイルな開発の短期の反復サイクルに収まること
- 容易性
  - 最低限の指導でモデリング経験がほとんどない開発メンバーが自ら適用できること
- 設計改善
  - 実装前に設計に多くの時間悩むより，とりあえず実装し得られた設計を改良すること

これらのモデリング作業のうちで概念設計モデリングの目的は，要求の実現性や技術的なリスクを明らかにすることである．概念設計モデリングは，開発対象となるソフトウェアの要求事項のリストがまとまった時点で， 要求を取りまとめるメンバーと実装技術を理解したメンバーの2者がシステムの内部動作を大雑把にモデリングすることで実施する．協調的な詳細設計モデリングの目的は，推敲フェーズ前半で作成されたプログラムの設計品質を責務主導型設計の観点から改善するとともに，作成フェーズで実装するプログラムを先行的にモデリングすることである．協調的な詳細設計モデリングは，ソフトウェアの実現に使う技術がプロトタイピングで実証された時点で，開発メンバー全員が参加するグループワークの形で実施する．協調的な設計改善モデリングは，作成フェーズで実装されたプログラムコードの設計をもう一度調査し，設計品質の改善を行うことを目的とする．協調的な設計改善モデリングは，実装作業がほぼ完成した段階で開発メンバー全員が参加するグループワークの形で実施する．これらのモデリング作業をモデリングに不慣れな開発者から成るプロジェクト

表 4 本研究で提案する 3 種類のモデリング作業

適用時期	モデリング作業名	モデリング作業内容	所要時間
方向付けフェーズ後半	概念設計モデリング	システムの主要機能について内部機能の相互作用を UML のシーケンス図で大雑把にモデリングする	1 主要機能につき 0.5-1 日間
推敲フェーズ終盤	協調的な詳細設計モデリング	システムの主要機能について相互作用の詳細を UML のコラボレーション図でモデリングし，クラスに送信されるメッセージを CRC カードに書き込む	1 主要機能につき 0.5-1 日間
作成フェーズ終盤	協調的な設計改善モデリング	当初の設計に対して適用された拡張を UML のクラス図でレビューし，クラスの分割の見直し等を行う	1 反復につき 1-2 日間

に導入する場合には，最初導入する時点でモデリングを一緒に実践する形で指導を行う必要がある．

以上説明したようなモデリング作業を行うことで，開発終了時に概ね C2 以上のレベルの設計品質に到達できると期待される．

4. 適用結果

本研究で提案したモデリング手法を社内向けのツール開発において適用した．開発対象のソフトウェアは，Java 言語で実装され，Java の分散オブジェクト技術である RMI(Remote Message Invocation)や Java の XML パーサなどの技術を適用している．

図 4 は，モデリングの手法を適用したプロジェクトのスケジュールの概要とモデリングの適用時期を示したものである．開発メンバーは，推敲フェーズの反復#2 まで専任メンバー兼プロジェクトリーダ1名と要求定義者兼コーチ1名（兼務）の体制で進行し，反復#3 で専任メンバー2名を増員し，反復#3 から反復#8 まで3名の専任メンバーと要求定義者兼コーチ1名（兼務）で開発を行った．基本的な機能の実装が完了した反復#9 以降，専任メンバーが1名減少している．また，図 4 に示されるように方向付けフェーズで概念設計モデリングを実施し，推敲フェーズの反復#3 で協調的な詳細設計モデリングを実施した．これらのメンバーで要求定義者兼コーチ1名を除く3名は，オブジェクト指向分析設計のトレーニングは受講したものの実開発でのモデリング実践経験はほとんどない状態であった．そのため，コーチ1名が概念設計モデリングでモデルをレビューしたり，協調的な詳細設計モデリングの実施初期段階にモデリングの実演を行う形で指導を行った．

図 5 と図 6 はモデリング手法を適用したプログラムコードを Eclipse Metrics Plugin で測定した結果である．同時に測定されたクラスメソッドの割合は，反復#2 において 50%を超えていたが反復とともに単調に減少し，反復#9 では 1.2%に低下した．また，他のクラスに対する関係数が 25 を超えるクラスは反復#2 の時点で1クラス検出されたが，反復#3 以降は検出されなかった．図 5 から，C0 に相当する手続き指向のメソッドの割合が協調的な詳細設計モデリングを実施した反復#3 以降で減少したものの，反復#6 以降で再び増加していることが分かる．また，図 6 から反復#3 ではメソッドの凝集度が低いクラスの割合は低いものの，反復#4 以降で徐々に増えていることが分かる．図 6 において，反復#2 でのメソッドの凝集度が低いクラスの割合は一見低いように見えるが，これは LCOM-HS の測定からクラスメソッドが除外されているため見かけ上良くなっているものと考えられる．

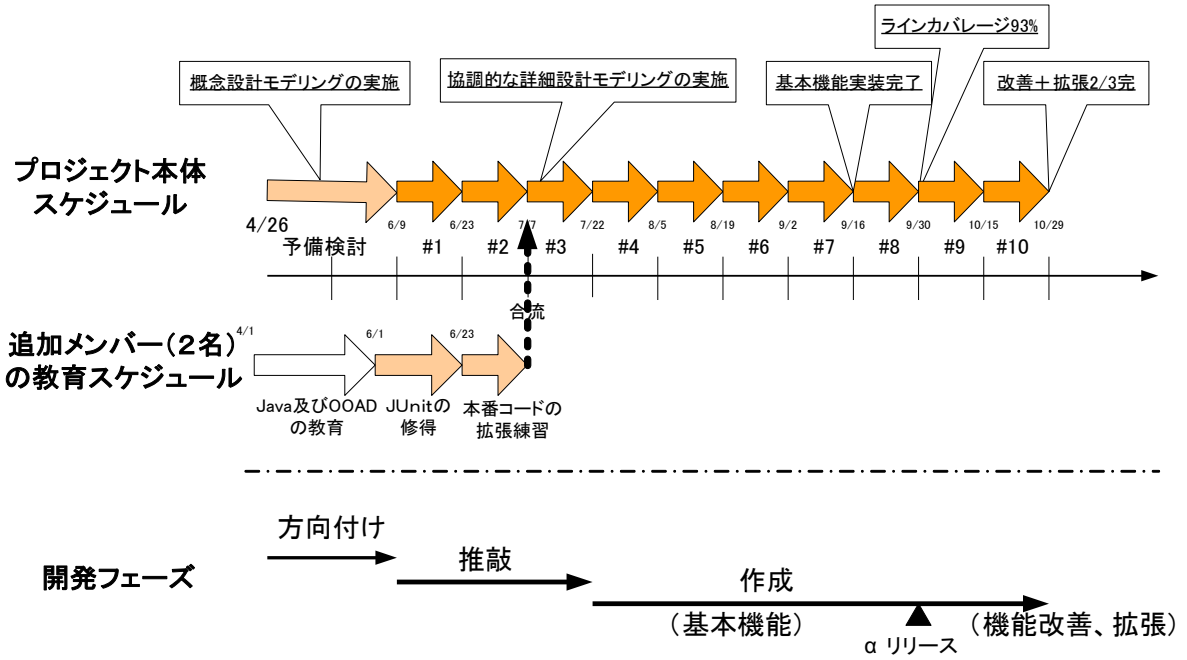

図 4 開発スケジュールとモデリング作業の適用時期

これらの測定結果から協調的な詳細設計モデリングにより，モデリング直後に全体的な設計品質は改善したものの，モデリングした時点以降で設計品質が再び低下したことが分かる．このような品質低下は，反復#3 で作成された設計モデルにおいて反復#5 以降で足りない機能がいくつかあるのが判明し，それらを実現するために設計部分を拡張した箇所において発生したものと思われる．このような当初の設計に対する設計拡張では，既に存在するクラス定義をどのように拡張すべきかということを問われる点で当初の設計を考えるのとは異なる．そのため，協調的な詳細設計モデリングのやり方を単純に適用することでは対処できなかったと考えられる．このような設計品質の低下に対処するためには，実装が進んだ段階でコーチの協力の下で設計改善を実施する必要があると考えられる．

これらの測定結果及びプロジェクトの進行状況から概念設計モデリング及び協調的な詳細設計モデリングの有効性は以下のように評価できる．

- 概念設計モデリング：アーキテクチャで実現すべき機能と相互作用の大筋が明らかになったことで，後続するアーキテクチャ上のリスクを洗い出すことができた．相互作用の大筋を協調的な詳細設計モデリン

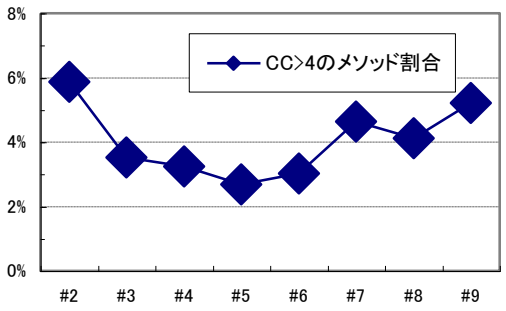

図 5 CC>4 であるメソッドの割合の推移

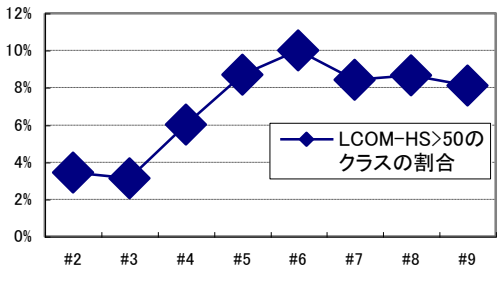

図 6 LCOM-HS>50 であるクラスの割合の推移

グの出発点にすることにより，モデリング作業が効率化した．

- 協調的な詳細設計モデリング：設計品質を改善し，結果的に実装作業も効率化した点で有効だと考えられる．

但し，開発完了の時点まで設計品質が良好な状態を持続するためには，協調的な詳細設計モデリングで得られた設計の拡張に伴う設計品質の低下を改善するような設計改善作業を実施することがさらに必要になると考えられる．

5. まとめ

本論文では，アジャイルなソフトウェア開発の特長である2週間程度の短い反復期間において実践可能で，かつ開発に伴うリスクを低減したり， 設計品質を向上させることを狙った3種類のモデリング作業を提案した．すなわち，開発初期段階に要求内容を検証するための”概念設計モデリング“，アーキテクチャの確定時点で責務の分割の点で良好な品質な設計を作るための”協調的な詳細設計モデリング“，実装が概ね完了した時点において開発途上で劣化した設計品質を改善するための”協調的な設計改善モデリング“である．これらのモデリング作業は，概ね半日から数日間で実施することができ，モデリング経験がほとんどない開発者でも少ない指導で実践できるように考案された．

本研究で提案した”概念設計モデリング“及び”協調的な詳細設計モデリング“を社内向けソフトウェア開発を行うプロジェクトに適用した．その結果，”協調的な詳細設計モデリング“を実施することにより，手続き的なコードを減少させたり，クラスに対する責務の割り当てを向上するなど設計品質の改善が達成された．その一方，”協調的な詳細設計モデリング“で得られた設計を拡張した部分において設計品質が低下する傾向が見られた．このような設計品質の低下に対処するためには，実装作業がほぼ完了した段階でさらに”協調的な設計改善モデリング”を実施し，さらに設計改善を行うことが必要だと考えられる．

謝辞

本研究で提案しているモデリング作業の実開発への適用においてご協力頂いた同僚の矢田貴也さん及び舛屋康典さんにこの場を借りてお礼をさせて頂きます．

6. 参考文献

〔1〕 アリスター・コーバーン，*アジャイルソフトフェア開発*，ピアソン・エデュケーション，2002

〔2〕 ケント・ベック，*XP エクストリーム・プログラミング*，ピアソン・エデュケーション，2000

〔3〕 マーチン・ファウラー，*リファクタリング：プログラミングの体質改善テクニック*，ピアソン・エデュケーション，2000

〔4〕 Rebecca Wirfs-Brock and Alan McKean, *Object Design – Roles, Responsibilities, and Collaborations*, Addison-Wesley, 2003

〔5〕 Brian Henderson-Sellers, *Object-Oriented Metrics – Measures of Complexity*, Upper Saddle River, 1996

〔6〕 http://www.teaminabox.co.uk/downloads/metrics/index.html

〔7〕 フィリップ・クルーシュテン，*ラショナル統一プロセス入門*，ピアソン・エデュケーション，2001

〔8〕 藤井他，*普通のプロジェクトへの適用を目指したアジャイルな開発手法の構築と適用結果*，情報処理学会研究報告 2004-SE-145, pp.15-21, 2004

オージス総研

本社／岩崎コンピュータセンター
〒550-0023 大阪府大阪市西区千代崎 3 丁目南 2 番 37 号 ICC ビル
TEL.06-6584-0011（代） FAX.06-6584-6497

東京本社
〒108-6013 東京都港区港南 2-15-1 品川インターシティA棟 12,13F
TEL.03-6712-1201 FAX.03-6712-1202

千里オフィス
〒560-0083 大阪府豊中市新千里西町 1 丁目 2 番 1 号 クリエイティブテクノソリューション千里ビル
TEL.06-6831-0531（代） FAX.06-6872-9404

名古屋オフィス
〒460-0003 愛知県名古屋市中区錦 1 丁目 17 番 13 号 名興ビル
TEL.052-209-9390 FAX.052-209-9391

豊田オフィス
〒471-0034 愛知県豊田市小坂本町 1 丁目 5 番地 10 号 矢作豊田ビル
TEL.0565-35-6722 FAX.0565-35-6723

尼崎オフィス
〒660-0892 兵庫県尼崎市東難波町 5 丁目 29 番 50 号
TEL.06-6489-2300（代） FAX.06-6489-2307

http://www.ogis-ri.co.jp/

2013. 5