知働化研究会誌 創刊第１号

EXEKT Review Volume One (November 2010)

アジャイルプロセス協議会

知働化研究会

知働化研究会
Executable Knowledge and Texture Laboratory

2010.11.3 編集

本誌掲載の記事は、アジャイルプロセス協議会 / 知働化研究会の活動成果をまとめたものです。著作権は、各作品の著者に属します。各記事の内容は、知働化研究会サイト（http://www.exekt-lab.org/）に掲載、あるいは、関連しております。各作品・記事に関するお問い合わせ、ご質問などは、個別にお願いいたします。また、本誌全体に関するお問い合わせは、ご意見などは、otsuki.s@exekt-lab.org（大槻）までお願いいたします。

知働化研究会誌 創刊第１号

EXEKT Review Volume One (November 2010)

アジャイルプロセス協議会
知働化研究会

序 6

知働化前史 山田正樹 7
巻頭言：知働化研究会誌発行に寄せて

知働化研究会活動経緯 大槻繁 15
ポスト・アジャイルプロセス起動

論説 23

知働化プロセス 山田正樹 25
知働化プロセスの一つのあり方についての試論

リアルウェア 濱勝巳 36
ソフトウェアは空

対話『リアルウェア』 大槻繁 50
Ｈ氏とＯ氏との対話

システム・エンターティナー 時本永吉 57
システム開発を楽しむ

ゆるっと行こう 本橋正成 87
ゆる思考のすすめ

ΛＶモデル 大槻繁 105
Ｖ字モデルからの意味論的転回

クラウドコンピューティングにおける
アーキテクチャの進化の方向性 萩原正義 132
頭脳のアーキテクチャからのアプローチ

寄稿 149

アジャイル開発プロセスと契約　高橋雅宏 151
契約はアジャイル開発プロセスに追いついたか

小論 165

知働化が切り開くソフトウェア工学の価値創造　竹内雅則 167
ソフトウェアと価値

なぜ、あなたはモデルが描けないか　天野勝 171
概念モデルと業務分析モデルの関係

ソースコードを書くことに似ていること　中村裕樹 176
ソースコードという文

新しい知識のカタチ　羽生田栄一 182
知識に対する新しい取り組み

編集と知働化　野口隆史 188
新時代の知識編集

結 195

編集後記 197

入会のご案内 202

序

知働化前史　山田正樹

巻頭言：知働化研究会誌発行に寄せて

知働化研究会活動経緯　大槻繁

ポスト・アジャイルプロセス起動

知働化研究会のコンセプトリーダである山田正樹氏は、「知働化」の前身である「実行可能知識と様相」というテーマで、ソフトウェア開発の実践やビジネスを行いながら思索を深めていました。『知働化前史』は、知働化研究誌創刊号の巻頭言という位置づけで山田正樹氏に知働化のコンセプトの背景や想いを書き下ろしていただいた作品です。

知働化研究会は、2009 年夏に発足し、定期的な会合を行いながら継続的に交流してきました。運営リーダの大槻繁氏が『知働化研究会活動経緯』でその活動の一端を紹介しています。

知働化前史

知働化研究会誌発行に寄せて

山田正樹（やまだまさき）

有限会社メタボリックス
アジャイルプロセス協議会 副会長
知働化研究会 コンセプトリーダ
masaki@metabolics.co.jp

はじめに…… 8
Ⅰ．すべての出発点…… 9
Ⅱ．オブジェクト…… 11
Ⅲ．モデル…… 12
Ⅳ．知識…… 13
Ⅴ．知働化…… 13
Ⅵ．言語…… 14
〔プロフィール〕…… 14

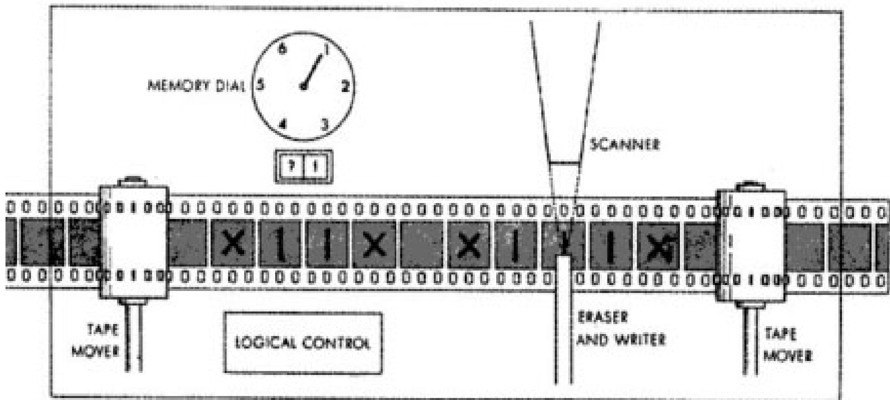
Copyright 2010, YAMADA Masaki, All rights reserved.

はじめに

編集長氏から、「知働化研究会誌発行に寄せて」何か書けとのご下命を賜った。編集長氏の意にはそぐわないかも知れないが、これをいい機会として「知働化」誕生以前の一つの物語を手短にまとめておこうと思う。ただしこれはある一人のメンバにとっての知働化前史に過ぎない。すべての知働化研究会メンバ一人ひとりにそれぞれの知働化前史があることだろう。これからは知働化研究会という紡ぎ車が、それらの糸を紡ぎ、やがては大きな織物へと織り上げられることになるはずだ ...

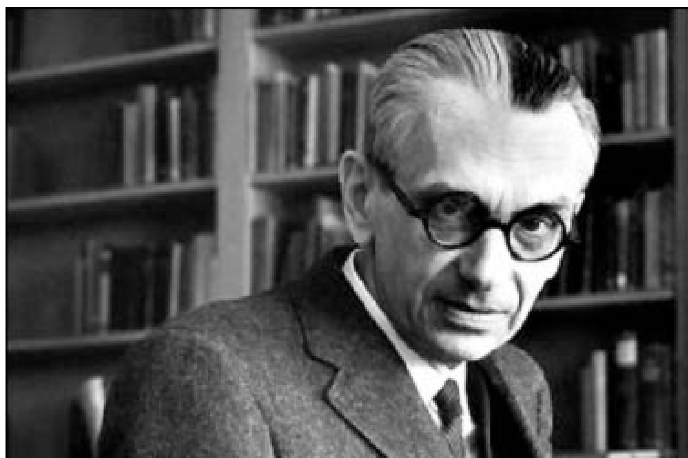
Ⅰ. すべての出発点

人類のすべてのミトコンドリアをたどると、十数万年前にアフリカの大地溝帯に生きた一人のイブにたどり着くように、我々の思考の旅も（ひとつではなく）二つの偉大な思考にたどり着く。一つは Alan Turing のチューリング・マシン (1936) である。



チューリング・マシンは「コードを実行する」ということのもっとも原始的な意味を与えている。ちなみに Turing と同じくイギリス人である Michael Jackson も "computer" とは言わず、"machine" という用語を使う。そこには「実行」というイメージが強く反映されているのではないかと思われる。

　そしてもう一つはやはり、Kurt Gödel の不完全性定理 (1931) である。もっともここで重要なのは不完全性定理がもたらす結論そのものではなく、その証明の過程で用いられたコード化なのだが。Gödel のコード化は、「世界をコード化する」ということの端的な (そして多分人類史上初めての) 例となった。



こうして「世界をコード化し」「コードを実行する」ことの原初的イメージが誕生した。

Ⅱ．オブジェクト

そして時代は一挙に 50 年を飛ぶ。「オブジェクト」は今やくだらないものの代名詞だが、1980 年にはそうではなかった。オブジェクトには大きく分けて三つの源流がある。

- **抽象データ型**
- **シミュレーション**
- **認知科学**

「クラス」、「スクリプト」、「属性」などを初めとする多くの用語や概念は認知科学の分野からもたらされている。オブジェクトはまさに知識表現の単位だったのである。だからソフトウェアが実行可能な知識であることは、30 年も前から (なんなら 50 年前の Lisp まで遡ってもいいのだが) わかりきったことだったのだ！

そう考えれば、オブジェクトのコミュニティから生まれたアジャイルなソフトウェア・プロセスは、ソフトウェア構築の手法と言うよりは (実行可能な) 知識集約の手法であることがよく分かるはずだ。

Ⅲ．モデル

数学基礎論における「モデル」は、おおざっぱにいえば公理系の具体例である。一方、ソフトウェアにおける「モデル」はむしろ具体例を抽象的に表現したものと捉えられているように思われる (数学基礎論とソフトウェアは本質的には歳の離れた兄弟か従兄弟のようなものだが、この例のように同じ用語が反対の用例として用いられることがままある。妥当性確認と検証もそのような例の一つ)。プラトン的な意味でのモデル (流出源) はむしろソフトウェアのモデルに近い (ただし向きが反対かも知れない) し、数学基礎論のモデル理論が前述の不完全性定理よりは Gödel の完全性定理により関連しているのも興味深い。それはさておき。

ソフトウェアが実行可能な知識であることがほとんど忘れ去られていた時代に、Stephen Mellor は「実行可能 UML」と言い始めた (2002)。もっとも Mellor は正確に「UML」と言ったのであって「モデル」とは言っていないし、ましてや「知識」とは口を滑らせても言わないだろう。彼にとってのモデルは解決領域のモデルであって問題領域のモデルではないのだから。

だが、そこで口を滑らせれば「実行可能知識」はそこにある (かもしれない)。それを確認するために 2004 年 2 月から 2007 年 8 月にかけて @IT 連載「変幻する実行可能知識」を書くことになる。

Ⅳ．知識

知識というと「どこかに何らかの形で (例えば百科辞典の中にことばとして) 存在するもの」を想像しがちではないだろうか？ しかし「識」という字を見れば分かるように、知識は「知」という作用であって、存在ではない (「知」が分析的なものか綜合的なものか、暗黙的なものか明示的なものか、意識的なものか無意識的なものかなどということは今は議論しないでおこう)。一方､モデルは存在するものだとしたら「実行可能モデル」から「実行可能知識」への移行は連続的な変化ではなく、何らかの亀裂を飛び越えてジャンプしてしまったことになる！

だから、ソフトウェアは機能を単に実現したものではないし、逆に情報やデータを集めただけのものでもなく、ソフトウェア自身が作用であるということを明確にするために「機能はただである」(2008 年 6 月)､「ソフトウェアは様相 (texture) を紡ぐ」(2008 年 9 月) と言わざるを得ない羽目に陥ったのである。

Ⅴ．知働化

かくして、2009 年 4 月 8 日、横浜中華街にて大槻繁氏と (既に弊社メタボリックス内に存在していた)「実行可能知識と様相研究所」をより広い場で組織化する懇談がなされ、6 月 2 日、本橋正成氏により「知働化」というキータームが提案され、8 月 27 日に第 1 回知働化研究会が開催されたいきさつについては皆さん、ご存知のとおりである。

Ⅵ . 言語

我々人間も含めて、生命はそれぞれの認識 (フレーム) を通してしか、世界を感じ、世界に参加することができない。いや、世界というひとつの確固たるものがあって、それをそれぞれが部分的に認識するというよりは、我々がそれぞれに認識する世界の集合がこの世界をかたちづくる。中でも人間は、言語というフレームに縛られる。言語化した途端に世界はずれ始めるが、言語化しなければそもそも世界を認識できないのである。

ソフトウェアも、いくら実行可能知識などと言い換えてみたところで、言語 (という認識フレーム) の束縛から逃れることはできない。いやむしろ、ソフトウェアは言語のみから構成される人工物である。ソフトウェアは、言語と同様に世界を構築し、破壊し、自己言及的であり、嘘をつくことができ、オートポイエーティックに増殖する。人間という格好の実行機械を見つけた言語は、今度はソフトウェアという形に変容して生き続けるのではないだろうか…

〔プロフィール〕

NHK の人形劇 (ぶーふーうーとか) に影響され「小さな世界」構築を妄想する、20 世紀半ばに生まれた不器用な子どもは、やがて計算機械というおもちゃを手に入れ、妄想が止まらないまま大人になり、誰に知られることもなくしずかな人生を送った。

序

〔知働化研究会誌創刊第１号〕
知働化研究会
Executable Knowledge and Texture Laboratory

知働化研究会活動経緯

ポスト・アジャイルプロセス起動

大槻 繁（おおつき しげる）

株式会社一（いち）副社長
アジャイルプロセス協議会 フェロー
知働化研究会 運営リーダ
otsuki.s@1corp.co.jp

Ⅰ．アジャイルプロセス協議会のあゆみ…………………………… 14
Ⅱ．知働化研究会のあゆみ………………………………………… 17
Ⅲ．会合開催等経緯………………………………………………… 18

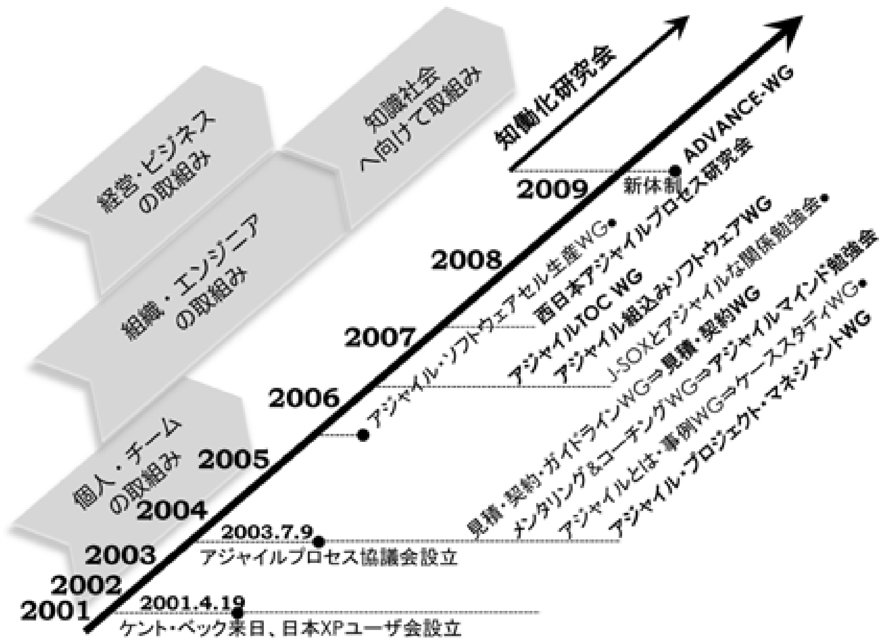
Copyright 2010, OTSUKI Shigeru, All rights reserved.

Ⅰ．アジャイルプロセス協議会のあゆみ

テクノロジックアートの長瀬氏と共同で、筆者がエクストリーム・プログラミングで有名なケント・ベックを招聘したのは 2001 年 4 月のことです。この時の盛り上がりで日本 XP ユーザ会（通称 XPJUG）が立上がりました。これが日本でアジャイルプロセスが周知されたきっかけになっていると考えられます。以降、雑誌や著作が多く出版され、急速に普及していきました。

XP のプラクティスに代表されるように、当初のアジャイルプロセスは、どちらかと言うとエンジニアや開発者の中での実践に留まったものでした。普及していくにつれて、ビジネス領域との関係、クライアントとの契約や見積りといった企業活動としてのアジャイルプロセスについて検討していく意識が高まってきて、2003 年 7 月に組織や企業参加を前提としたアジャイルプロセス協議会を設立しました。

アジャイルプロセス協議会は、WG（ワーキンググループ）活動が主体となっています。設立時から見積り、契約、プロジェクトマネジメント等がテーマとして設定されていて、組織に関わる事項が採上げられているのが特徴です。



2006 年頃からは、ソフトウェアセル生産、TOC（Theory of Constraint)、組込みソフトウェア、J-SOX といったビジネスマインドや実務領域の課題に挑戦する WG 活動が活発になってきました。どの企業も、海外の書籍で紹介されているようなものを味見する「なんちゃってアジャイルプロセス」の時代は終了し、各企業独自の文化にとけ込んだアジャイルプロセスを構築し始めています。

そして、今までの取組みが、どちらかと言うと第２の波へのアンチテーゼ的な位置づけだったのに対し、本来の知識社会の価値観や世界観に基づくアプローチとして始まったのが『知働化研究会』です。

前述のアジャイルプロセスのエンジニア領域内のものから、ビジネスや経営領域への拡大は、狭義のアジャイルから広義のアジャイルプロセスへのシフトと言えます。

狭義のアジャイルプロセス

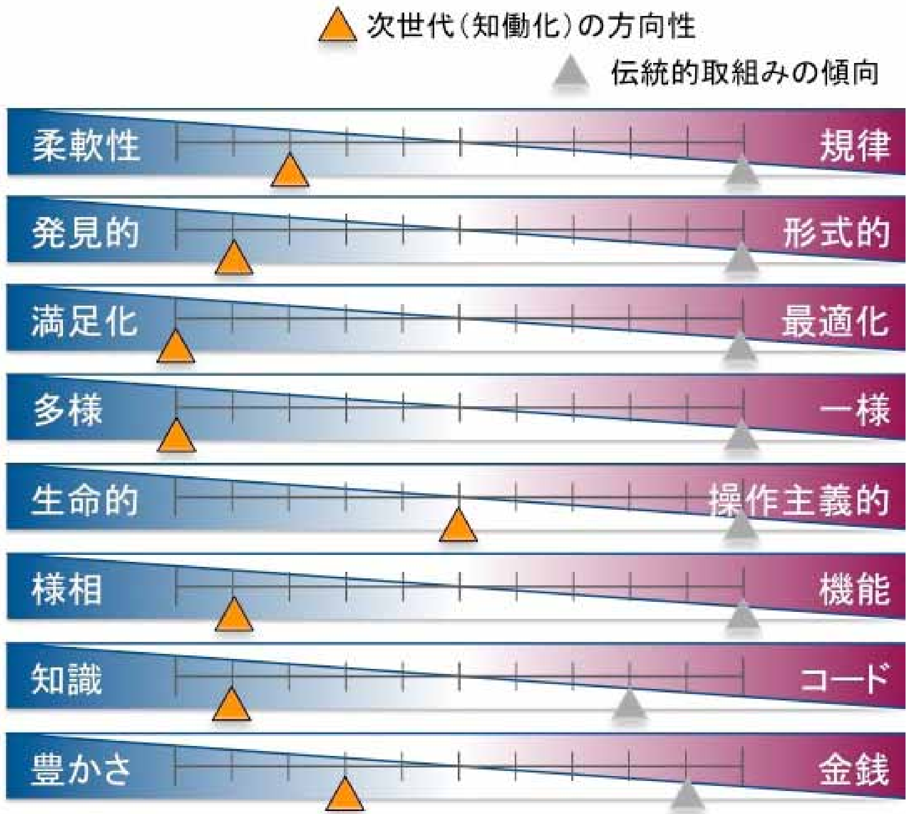
開発者側中心の視点
顧客側とのコミュニケーションや確認を重視してはいるものの、あくまでも受動的

広義のアジャイルプロセス

顧客側と開発側との同期
全体での最適化、ビジネスプロセスを能動的に考慮

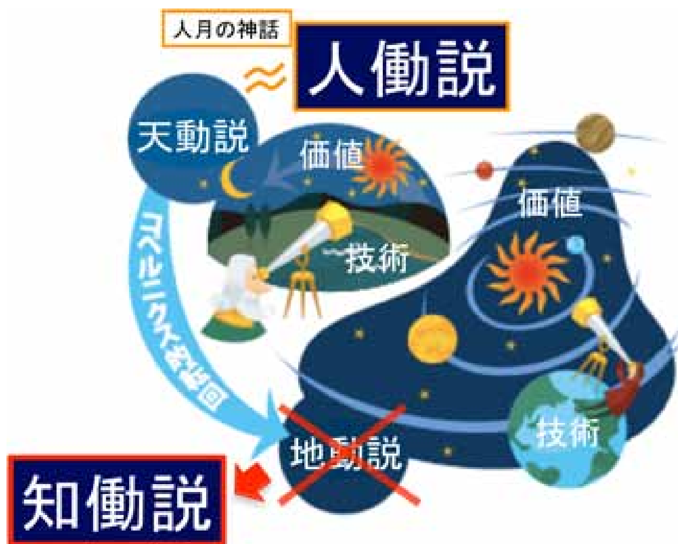
上図の広義のアジャイルプロセスの歯車の図は、本資料の表紙にも掲げたデザインです。ちなみに、上図の狭義のアジャイルプロセスの歯車の図では、歯が内側に向いたデザインになっていることに注意してください。

インターネットの普及、技術革新、社会・経済原理の変貌等によって、ソフトウェアやシステムを取巻く世界観も新しいパラダイムに移行してきています。



Ⅱ．知働化研究会のあゆみ

本章では、ソフトウェアエンジニアリングの第３の波に対する本格的な取組みの一つとして『知働化』について解説します。知働化研究会は、アジャイルプロセス協議会の WG として、2009 年 6 月に設立された先端的な活動です。コンセプトリーダとして山田正樹氏（メタボリクス社）、運営リーダとして筆者（大槻）、2010 年 4 月時点で、メンバ数 25 名の方々が参加しています。



『知働化』のコンセプトは、かなり本格的なパラダイムシフトです。いわば、コペルニクス的転回と言えます。今までのソフトウェア開発に対して、『人月からの脱却』を標榜しつつなかなか糸口が見つけられませんでしたが、知働化はその可能性を秘めています。考え方、手法、コンセプト、アプローチ、哲学とありとあらゆる事項が再定義（脱構築）される新しい体系です。

Ⅲ．会合開催等経緯

・山田−大槻懇談（2009 年 4 月 8 日於横浜中華街）
・山田−大槻懇談（2009 年 4 月 27 日於大磯ロングビーチ）
・本橋−大槻懇談（2009 年 5 月 7 日於錦糸町）
・アジャイルプロセス協議会に WG 設立申請提出 / 承認（2009 年 6 月 8 日）
・知働化研究会準備会合（2009 年 7 月 23 日於横浜 CIJ 社）

・第１回（キックオフ）会合：テーマ「問題提起ショー」

（2009 年 8 月 27 日於竹芝 TIS 社プリゼンテーションルーム）

０１	山田正樹	機能はただである
０２	野口隆史	知働化研究会への期待
０３	飯泉純子	問題は「問題」にある
０４	塩田英二	問題提起
０５	濱勝巳	ソフトウェアはソフトウェアに非ず
０６	本橋正成	無題
０７	羽生田栄一	新しい知識のカタチ
０８	萩原正義	概念プラグインの進化プロセス

・第２回会合：テーマ「知働化への期待とアプローチ」

（2009 年 10 月 15 日於竹芝 TIS 社プリゼンテーションルーム）

００	大槻繁	クリッペンドルフの『意味論的転回』読書感想
０１	本橋正成	ゆるいについてなど
０２	竹洞陽一郎	第 1 回会合に参加しての感想と考察
０３	綿引琢磨	知働化に期待すること
０４	山田正樹	もし (株) 知働化があったら , それはどんな会社か？
０５	野口隆史	EMZero ＋知働化
０６	大槻繁	知働化への接近
０７	佐藤真央	知働的システムの実験的実装のための解釈・具体化
０８	濱勝巳	ソフトウェア哲学事始め
０９	時本永吉	プログラマの戒
１０	羽生田栄一	雑感など

・第３回会合：テーマ「年の終わりに」

（2009 年 12 月 7 日於竹芝 TIS 社プリゼンテーションルーム）

０ ０　大槻繁　　言語ゲーム的転回の年表
０ １　野口隆史　知働化研究誌の進め方
０ ２　中村裕樹　ソースコードの文芸的読み方
０ ３　塩田英二　自動化、自働化、知動化、知働化のこころ
０ ４　茨木希　　情報から知識へ
０ ５　高野明彦　自発的な学びを育む連想的情報アクセス技術
９ ９　時本永吉　エンターテイナーと哲学（知働化サイト上での提示）

・第４回会合：各メンバの知働化への取組み

（2010 年 3 月 2 日於竹芝 TIS 社プリゼンテーションルーム）

０ ０　大槻　　　言語ゲーム入門
０ １　野口編集長　知働化研究誌の進め方
０ ２　野口　　　編集と知働化
０ ３　羽生田　　新しい知識のカタチ
０ ４　時本　　　ウォータフォール開発ですべきこと
０ ５　久保秋　　研修から見えるソフトウェア開発
０ ６　濱　　　　リアルウェアその後
０ ７　萩原　　　これからのソフトウェア
０ ８　本橋　　　エゴレス開発プロセス＆ちょっとした LT
０ ９　綿引　　　知働化におけるリーダシップ
１ ０　佐藤　　　アプリケーションにおけるコンテクストの使用
１ １　大槻　　　ΛＶモデル
１ ２　塩田　　　おもてなし、おまかせなど
１ ３　中村　　　知働的ソースコード読解
１ ４　飯泉　　　問題フレームとパターン
１ ５　荒川　　　デザイン指向のとりくみ

・第５回会合：知働化研究誌の取組み状況
（2010 年 6 月 2 日於竹芝 TIS 社プリゼンテーションルーム）
本研究誌の各メンバの検討状況について披露

・ソフトウェアエンジニアリングの呪縛 WG
（2010 年 6 月 9 日～ 11 日於横浜開港記念館）
SEA（ソフトウェア技術者協会）主催のソフトウェアシンポジウム SS2010
アジャイルプロセス協議会（知働化研究会）協賛

このWG では、ソフトウェアに関わる技術者、研究者、経営者、コンサルタントなどの多方面の方々に参加いただき、クールにソフトウェアエンジニアリングの現況を把握し、限界を見極め、課題を整理し、次のステップに進むための準備をしました。その成果として『新ソフトウェア宣言』という形でメッセージをまとめました。

詳細は、知働化サイト http://www.exekt-lab.org/Home/newsoftdecl に掲載されています。

・第６回会合：知働化研究会１周年記念会合
（2010 年 8 月 18 日於竹芝 TIS 社プリゼンテーションルーム）
「ソフトウェアとは何か？」舟越和己、渡辺滋氏による討論
本研究誌の各メンバの検討状況について披露

論説

知働化プロセス　山田正樹
知働化プロセスの一つのあり方についての試論

リアルウェア　濱勝巳
ソフトウェアは空

対話『リアルウェア』　大槻繁
Ｈ氏とＯ氏との対話

システム・エンターティナー　時本永吉
システム開発を楽しむ

ゆるっと行こう　本橋正成
ゆる思考のすすめ

ΛＶモデル　大槻繁
Ｖ字モデルからの意味論的転回

クラウドコンピューティングにおけるアーキテクチャの進化の方向性　萩原正義
頭脳のアーキテクチャからのアプローチ

論説の章では、比較的長編のまとまった論文を掲載しています。知働化研究会では、あえて「知働化」の言葉の定義は明確にせずに、参加者の方々の個別の解釈にお任せし、広がりを持たせるようにしてきました。それぞれの方々の自由な研究の方向性や成果が提示されています。

『知働化プロセス』は、山田正樹氏による知働化の一つのイメージの提示です。「受容」と「参与」という立場の間のやりとりのプロセスとして、知識のアクティビティの枠組みを提唱しています。

『リアルウェア』は、濱勝巳氏がかねてから検討してきたソフトウェアに関する哲学的省察です。対話形式の作品になっています。これに続く『対話：リアルウェア』は、同様の対話形式による大槻繁氏によるコメント作品です。

『システム・エンターティナー』は、時本永吉氏によるソフトウェア開発に対する姿勢やマインドの課題と役割に関する論説です。開発現場に携わっている開発者の置かれている課題が浮き彫りになっています。

『ゆるっと行こう』は、本橋正成氏による「ゆるい」というコンセプトの提案です。文化的背景を押さえつつ、これからのソフトウェアのあり方について探求しています。

『ΛVモデル』は、大槻繁氏によるパラダイム提案です。開発の世界が「V字モデル」として整理できるのに対し、問題領域を「Λ字モデル」として位置づけ、それ等を接合して全体を見る世界観を提示しています。

『クラウドコンピューティングにおけるアーキテクチャの進化の方向性』は、萩原正義氏による、巷でにぎわっているクラウドコンピューティングをテーマに、ガイア的とでも言う頭脳のモデルやモジュール化に基づく新世代のアーキテクチャ論です。

論説

〔知働化研究会誌創刊第１号〕
知働化研究会
Executable Knowledge and Texture Laboratory

知働化プロセス

知働化プロセスの一つのあり方についての試論

山田正樹（やまだまさき）

有限会社メタボリックス
masaki@metabolics.co.jp

はじめに……………………………………………………………… 26
Ⅰ．知働化プロセスに関わる二つの立場……………………………… 26
Ⅱ．知働化プロセスのかたち…………………………………………… 27
Ⅲ．知働化プロセスの主要なアクティビティ………………………… 28
　知働化プロセスのアーキテクチャ………………………………… 28
　参与する立場の主要なアクティビティ…………………………… 29
　受容する立場の主要なアクティビティ…………………………… 32
　その後のアクティビティ…………………………………………… 33
Ⅳ．知働化を現状において活用するシナリオ………………………… 34
Ⅴ．まとめ………………………………………………………………… 34
　〔プロフィール〕……………………………………………………… 35

Copyright 2010, YAMADA Masaki, All rights reserved.

はじめに

本稿は知働化のプロセスをひとつの仮想的な例を通して、記述する試みです。

本稿では、仮想的な例として、医療機器ベンダの商品カタログを取り上げます。つまり医療機器ベンダが顧客と交換する「モノ」という様相を知働化するプロセスを考えます。通常の「IT 化」のプロセスでは、必要とされる「機能」の洗い出しから始めて、「機能」の実現をプロセスの目標とするわけですが、知働化のプロセスでは「機能」は後からついてくるものと考えます。知働化のプロセスの場合に中心になるのは、どの「様相」を知働化するか、ということです。様相は、繊維を縒って糸にし、糸を織って布とし、布を縫い合わせて (例えば) 敷物とするように、多層的で連続的なものですが、その中のある部分を人為的に取り出して、対象となる様相とします。

Ⅰ. 知働化プロセスに関わる二つの立場

知働化のプロセスに関わる二つの立場があります。

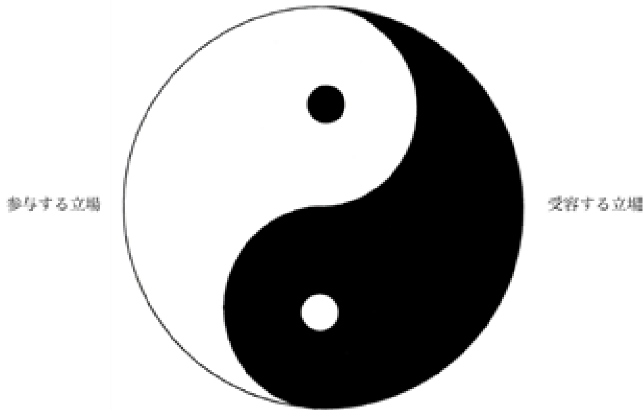
一つは、知働化の対象となる場で実際に継続的に活動しているひとや組織の立場です。例えば今回の仮想的な事例では、医療機器ベンダやその特定の部署、あるいはそこで働いたり、関係を持つ人の立場です。

もう一つは、日常的にその場に関わっているわけではない、あるいは今までその場に加わっていなかったが、新たにその場に加わって、知働化を推し進めるひとや組織の立場です。ここでは暫定的に前者を「受容する立場」、後者を「参与する立場」と呼ぶことにします。

参与する立場に立つ人や組織は、外部の目、心と手を持って知働化のプロセスに参加します。比喩的に言えば文化人類学におけるフィールドワーカのような立場ですが、「観察」を超える、より積極的な意味があります。IT 化のプロセスでは「開発者 / チーム」のように呼ばれてきた役割に相当しますが、その名前から分かるようにその働きは大きく異なります。

受容する立場に立つ人や組織は、知働化のプロセスの対象となる場の永続的な構成者で、知働化のプロセスに参加し、知働化による場の変化や成果を受け取り、自分たち自身が変容します。もちろん単なるインフォーマントや要求提出者ではありません。IT 化のプロセスでは「顧客」、「ユーザ」などと呼ばれてきましたが、知働化ではより主体的な視点から「受容する立場」と呼ばれます。

「受容」という単語からは一方的な受け身のイメージが浮かぶかも知れませんが、ここではより積極的に変化を受け入れ、自身が変容することを意味しています。



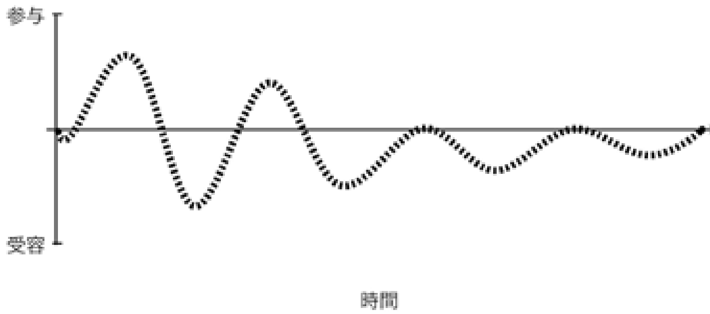
Ⅱ．知働化プロセスのかたち

知働化のプロセスは、ウォータフォール・プロジェクト的なプロセスではありません．つまり、期限 / 目標 / 予算が予め明確に決まっており、それに向かって単線的に行われるプロセスではありません。

同時に知働化のプロセスは、繰返し的 / 漸増的なプロセスでもありません。つまり、最終的な目標を細分化し、細分化されたサイクルの目標を達成することを繰り返して、最終的な目標を達成するプロセスではありません。

知働化のプロセスは、減衰する振動から構成される、定常的で持続的なプロセスです。局面に応じて小さな目標はその場で作られたり変更されたり破棄されたりしますが、それは大きな目標を達成するためではありません。その場をよりよく (と思われる程度に) 生き延びるだけです。また小さなフィードバック・ループは至る所に存在しますが、それは PDCA のように形式化されたものではありません。

知働化のプロセスは、実際には異なる側面を表す多数の振動から構成されますが、本稿ではその中でも主要な部分についてのみ述べます。下の図は、参与する立場と受容する立場がそれぞれ関わる、知働化の主要なアクティビティのおおよそを図示しています。



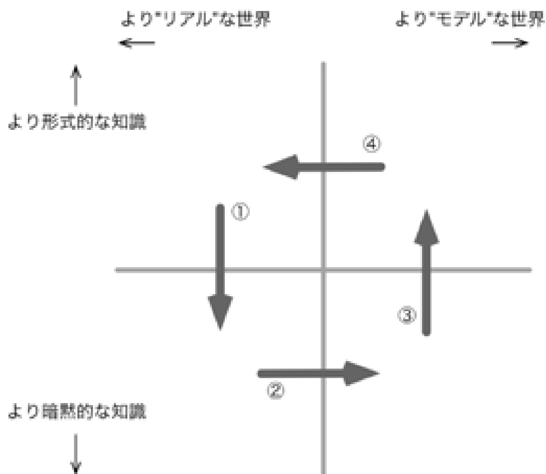
Ⅲ．知働化プロセスの主要なアクティビティ

本章では、知働化プロセスの主要なアクティビティについて全体を説明し、次いで参与する立場、受容する立場のそれぞれにとってのアクティビティの内容を述べます。

知働化プロセスのアーキテクチャ

参与する立場からも、受容する立場からもここの具体的なアクティビティの内容はもちろん異なりますが、その位置付けは変わりません。ここではその両者に共通する、プロセスの構成について述べます。

参与する立場であっても、受容する立場であっても、知働化プロセスの一回の振動はおおよそ次の図のような構造をしています。



上の図で４つの太い矢印がそれぞれのアクティビティを表しています。どこから始まって、どこから終わるか (知働化プロセスはエンドレスな繰り返しではありません)、具体的なアクティビティは何か、などは立場や局面によって異なります。

上の図で、アクティビティ①は、現実の世界において明示的な知識を暗黙的な知識へと内面化するアクティビティです。アクティビティ②は , 現実の世界に関する暗黙的な知識をモデルの世界に抽象化するアクティビティです。アクティビティ③は、モデルの世界において暗黙的な知識を明示的な知識に形式化するアクティビティです。アクティビティ④は、モデルの世界に関する明示的な知識を現実の世界に実現するアクティビティです。

それぞれ①内面化、②抽象化、③概念化、④埋込と仮に名前を付けます。

参与する立場の主要なアクティビティ

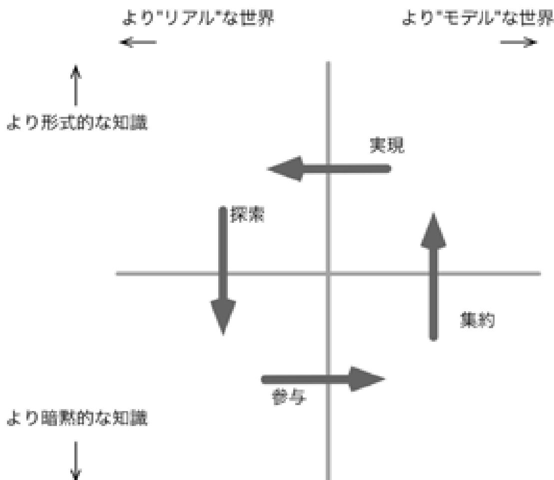
参与する立場の主要なアクティビティは、以下のようなものです。

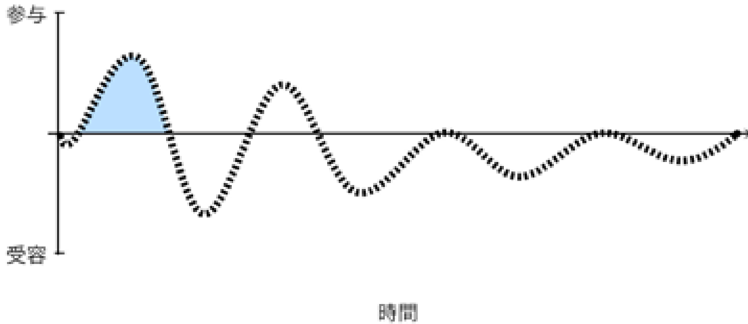
* 探索 (内面化)

* 参与 (抽象化)

* 集約 (概念化)

* 実現 (埋込)





探索のアクティビティでは、その業務、業界、製品などに関するさまざまなドキュメントや書籍、情報などを収集し、読み込み、理解する作業を行います。

医療機器商品カタログの例では、医療機器に関する書籍、医療機器業界の商慣習や内実を示す参考文献、Web 上のさまざまな情報、実際の現場で使われているマニュアルやカタログなどを集めます。特に指針はありませんが、これから行う知働化の領域に関する表層的な知識を、自分たちの内部に蓄積していくことが目的です。このアクティビティで作成する成果物も特にありませんが、収集し、読み込んだ情報を何らかの方法で整理し、保存し、アクセスできるようにします。明示的な成果物はなくても、この作業そのものに意味があります。例えば 2 週間程度の期間を割り当てます。

参与のアクティビティでは、参与する立場のひとが受容する立場の現場に入り込み、業務を共に (一対一の関係で) 行います。その過程でさまざまな情報や体験を得て、何らかの形で定着させます。

医療機器商品カタログの例では、商品が関わる複数の現場 (例えば営業、ヘルプ・デスク、製品企画など) に参加し、いくつかの仕事を専門家と一緒に行います。もちろん彼らと同等の仕事をできるわけではありませんが、ペアあるいは師弟として仕事に参加します。その過程でさまざまな細かい情報や暗黙的な知識を収集し、記録します。ムービー、写真、メモ、走り書きの図などが重要です。例えば 2 週間 ~1 ヶ月程度の期間を割り当てます。

集約のアクティビティでは、参与のアクティビティで蓄えた多くの明示的 / 暗黙的な知識を出し合い、チームのワークショップを通してそれらを概念化します。

医療機器商品カタログの例では、探索のアクティビティで収集され、読み込まれたさまざまな情報、参与のアクティビティで収集され、体験された情報をまずレビューします。

その中には商品情報がどのように扱われる / 手渡されるか、異なる業務ごとに何が求められるか、どのような変異や捨てられてしまう情報があるか、なども含まれています。

その次に、これらの情報からキーとなる概念を抽出し、その概念の元に情報を分類し、概念の間に関係をつけます。「概念」には名詞的な概念 (オントロジ) もありますし、動的な概念 (プロセス) もありますし、制約的な概念 (ルール) もあります。それ以外にも領域固有の種類の概念がある場合もあります。例えば医療機器の商品には、カタログ上の「見栄え」と顧客との関係や商品のライフサイクルやバリエーションなどを扱う必要があるかも知れません。

次に概念を表すのに適した「言語」を考えます。この言語はその概念に属する具体例を適切に表すことができなければなりません。また今後現れるであろう変異も柔軟に表せることが望まれます。

最後にこの様相が持つ価値付けを行います。この例では、商品カタログがリアルタイムで更新され、それを関連する全部署で単にアクセスするだけではなく部署ごとの使い方にリンクできること、機器ユーザの要望 / 評価 / 事例を紐付けることによって、様相の持つ価値を活かすことができると考えました。

また、これらのアクティビティを通じて、必要に応じて受容する立場のひとなどに対するインタビュなどを行う場合もあります。

これらの表現方法はいろいろありますが、例えばカードや手書きの図でも構いませんし、UML その他のモデリング技術、あるいはその拡張を使っても構いません。どんな形にせよ、成果物はリポジトリに格納します。

このアクティビティに対して、例えば 2 週間から 1 ヶ月の期間を割り当てます。

実現のアクティビティでは、集約のアクティビティで構築した概念と言語を元に、実際に動くモデルを作ります。ただし、単なる IT 化とは異なり、IT システムを作るだけとは限りません。仕事のスタイル、情報のデザイン、他の仕事との繋がり具合、IT システム、非 IT 的な道具なども含めたものです。

医療機器商品カタログの例では、バックエンドを Grails、フロントエンドを Flex を用いて実現します。Grails を使用することによって、前のアクティビティで構築したオントロジ、プロセス、ルール、言語などをそのまま実現することができます。

また通常の IT 化とは異なり、オントロジ、プロセス、ルール、言語などに (アプリケーショ

ンを通してではなく）直接アクセスする仕組みを入れます。ユーザ・インタフェイスも部分的にユーザのレベルで変更することができます。また、実現されている機能とそれに対応する価値付け、価値の評価もシステムに組み込みます。また、「最小限での実現」の原則を守ります（例えば既存のデータを丸ごと移行するのではなく、アダプタを介してアクセスできるようにする、など）。

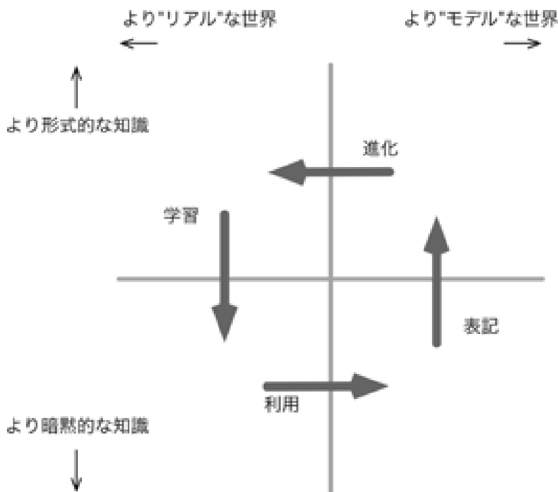
そして、これらのスタイル、デザイン、システム、道具を受容する立場に埋め込めるように準備します。

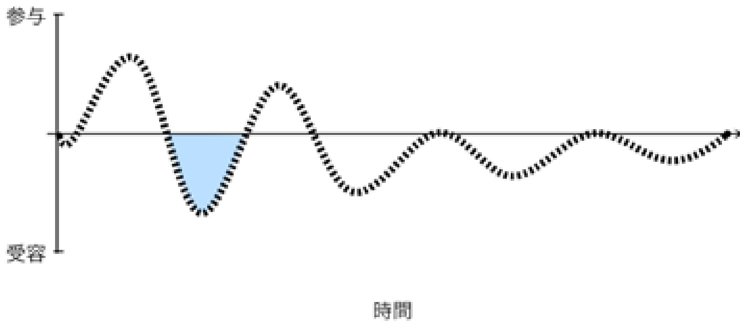
このアクティビティに対して、例えば 1 ヶ月から 2 ヶ月の期間を割り当てます。

受容する立場の主要なアクティビティ

受容する立場の主要なアクティビティは、以下のようなものです。

* 学習 (内面化)
* 利用 (抽象化)
* 表記 (概念化)
* 進化 (埋込)





学習のアクティビティでは、受容する立場のひとは参与する立場のひとから、チュートリアル、トレーニング、テキストなどの形で、提案された知働化 (システム、スタイル、プロセス、ルールなど) を学びます。

利用のアクティビティでは、実際の現場において、受容する立場のひとが参与する立場のひとと共同で、提案された知働化を運用します。

表記のアクティビティでは、受容する立場のひとびとが自分たちの視点、立場から、自分たちのことばを用いて、知働化を記述します。例えばマニュアルを書く、書籍を作る、チュートリアルを作るなどです。その一部には、知働化で提供された IT システム自体を使って、IT システムの使い方を IT システムの中に作り込むことが含まれる場合もあります。

進化のアクティビティでは、受容する立場のひとびとが参与する立場のひとびとの力を借りながら、知働化を進めていきます。

その後のアクティビティ

その後の振動がどうなるかは、知働化の事例ごとに異なります。前記のアクティビティが何度か繰り返される場合もありますし、一度きりの場合もあります。ただし、いずれにせよ、長期的には参与する立場のひとびとは徐々に現場を去り、受容する立場のひとびとが主体的に知働化を日常的、継続的に進めていくことになります。

Ⅳ. 知働化を現状において活用するシナリオ

知働化があまり人口に膾炙していない現状では、直ちに「IT」などに代わって知働化を導入するのは困難な場合があります。そのような場合には、現状の「IT 化」の文脈において知働化を進めるために次のようなシナリオが考えられます。

最初の参与する立場のアクティビティ、受容する立場のアクティビティが終了した段階で、知働化の成果を「IT 化」への提案と「動く仕様」としてまとめます。これを従来のシステム・インテグレータに発注する場合には、知働化の成果は RFP、要求仕様書となります。この時点で既に稼働する要求仕様が存在しているわけで、発注元には大きな利点になります。つまり「知働化」を (極めて強力な)「IT 化」の前段階として利用する方法です。

もちろん、ここまでできているのならば新たなシステム開発は不要であるとも考えられます。この場合に知働化を IT 化の文脈に押し込めるためには、知働化の成果の一部である IT システムをベースとして通常のプロジェクト化し、品質向上、機能追加、文書化、運用などの作業を追加する方法もあります。

Ⅴ. まとめ

本稿では、知働化のプロセスを (知働化とは何かを定義することなく) 構想してみました。仮想的な例を元にしたとは言え、欠けている部分、具体的でない部分があまりにも多く存在すると思います。また、概念的にもまだまとまりきっていない部分、よりよい名前付けや概念化が必要な部分もあります。それらについては、今後ともできる限り具体的な案件に基づいて、解決を図っていきたいと思います。

ただ、これによって知働化のひとつのイメージを頭に思い浮かべることができれば、今後の議論の種になり得ると考え、未完未熟ながら本稿を上梓しました。今後とも読者の皆様のご指導ご鞭撻のほど、よろしくお願いいたします。

最後に知働化研究会リーダの大槻氏、研究会誌編集の労を執られた野口氏を始め、本稿執筆の機会を与えていただいた知働化研究会の全メンバに感謝します。

〔プロフィール〕

1981 年頃 Lisp を、その後続いて Smalltalk-80 を知る。これらに共通する (実行可能知識的に) 重要な性質は “self descriptive” であること、つまりソフトウェアが自分で自分自身の「意味」について記述していること、また外の世界からその記述にアクセスできることである。その後、Objective-C、(貧乏人のための Objective-C であるところの) Java、(よりマシな Java であるところの) Groovy などを仕事で使ってきたが、残念ながらどれもそのような性質は持っていなかった。それにはソフトウェアを構築するプロセスと構築されたプロダクトが表裏一体の､入れ子の関係になっていなければならないからだ。そのような性質を持つプログラミング言語 / 環境 / ソフトウェア・システムの復活を望みたい。

論説

〔知働化研究会誌創刊第１号〕
知働化研究会
Executable Knowledge and Texture Laboratory

リアルウェア
ソフトウェアは空

濱 勝巳（はま かつみ）

株式会社アッズーリ
アジャイルプロセス協議会 会長
katsumi.hama@azzurri.co.jp

概要……………………………………………… 37
知働士と技術者との会話………………………… 37
解説……………………………………………… 46
〔参考図書〕……………………………………… 48
〔プロフィール〕………………………………… 49
読者からのコメント……………………………… 49

Copyright 2010, HAMA Katsumi, All rights reserved.

概要

本論文は、ソフトウェアを知らないままソフトウェアについて語る事ができないという前提のもと「ソフトウェアとは何であるか？」という問題を掲げ、それに答えることによって新しいソフトウェアの姿とこれまでのソフトウェアを取り巻く世界の問題を知働士と技術者の対話とその解説によって提示しています。

知働士と技術者との会話

A　こんにちは、なんか蒸々して暑いですね。こう暑いと冷えたビールでも飲みたくなりますねぇ。今日はあなたが経験のあるソフトウェア技術者だという事を聞いて、ちょっとだけ話を聞いてもらおうと思って来ました。お忙しいところすみませんが、少しだけお時間よろしいですか？

B　次があるのであまり時間はありませんが、少しだけなら構いませんよ。

A　ありがとうございます。あなたは、実務経験も長いソフトウェア技術者であるということですので、ソフトウェアについて十分熟知していると思ってもよいですね？

B　ええ、それなりに色々なプロジェクトにも参加してきまして、自分で云うのも何ですが、それなりの経験は積んで来たつもりです。技術者足るもの日々スキルアップしていかなければならないので色々と勉強しています。これまで、何人もの後輩に指導もしてきましたので、大概の事であればわかっているとは思いますが…

A　それは良かった。いやね。実は、ソフトウェアについてお話をしたいのです。

B　はい？それは、何かのプログラミングについてですか？それともオブジェクト指向ですかね？そうじゃなければソフトウェア工学についてとか？ソフトウェア製品の使い方ってことではないですよね。“ ソフトウェアについて “ ではわからないので、具体的にソフトウェアの何について話をしたいのかもう少し詳しく聞かせてもらえますか？

A　いいえ、いいえ、そういった類の物ではなく、ソフトウェアについて教えてもらいたいのです。ソフトウェアの “ 何か ” ではなく、ソフトウェアが何なのかあなたの意見を聞かせてもらいたいのです。

B　ソフトウェアが何であるかだって、ハハハッ。そんなの簡単ですよ。えーっと…ううん…

A　どうしました？ソフトウェアって何ですか？

B　ソフトウェアってプログラミングですかねえ？いや、それはおかしい。プログラミングソースのことかな？それも違うな。じゃあ、実行モジュールやバイナリコードのことか？構造やロジック？うーん、ソフトウェアねえ。ソフトウェアは、ソフトウェアだよ。おかしいな、わかっているはずなのに…

A　困りましたね。ソフトウェア技術者のあなたがソフトウェアを知らないなんて。

B　はあ、お恥ずかしい。ソフトウェアが何であるのか、簡単に答えられないなんて、私はこれまでソフトウェア技術者として何をしていたのか分からなくなりました。ああ、そんなことは当然の事、常識だと思っていたのに…、まったく、考えたこともありませんでした。だって、ソフトウェアはソフトウェアなのですから、それ以上でもそれ以下でもないですよね。

A　まあ、そんなに落ち込まないでください。私自身、ソフトウェアとは何であるかがわからずに自問自答して、ある答えを見つけました。そこで、経験のあるあなたにソフトウェアが何であるかを確認してみたかったのです。これからソフトウェアが何であるのか、一緒に考えて行きたいと思っていますがよろしいですか？たぶん、私も含めて多くの人があなたと同じようにソフトウェアを漠然と捉えていて、良く分からないまま扱っているのです。不思議なものです。

B　そうですね。落ち込んでいても始まらないので、ソフトウェアが何であるのが一から真剣に考えてみることにします。でも、ソースコードや実行モジュールを作るのが私たちソフトウェア技術者の仕事ですから、ソースコードやコンパイルされた実行モジュールが「ソフトウェアだ」とも言えるような感じは、そう間違っていないような気がします。でも、どうもそれだけでもないような気がします。その部分がどうも引っかかっているので、はっきりと「ソフトウェアとは是々である」と言いきれなくなっています。

A　では、ソフトウェアが、あなたの言うようにソースコードやコンパイルされた実行モジュールだとした場合、ソフトウェアの対極にあるハードウェアとの違いはどこにありますか？

B　それなら簡単です。ソフトウェアは、手で触れない物であって、ハードウェアは手で触れられる物ですから、ソースコードやコンパイルされた実行モジュールはソフトウェアであると言えます。解った。そうですよ、手で触れられない物がソフトウェアですよ。

A　まあ、それでも良いのですが、ちょっと待ってください。あなたの言う通りだとするとソフトウェア技術者は“手で触れられない物”の技術者ってことになりますよね。“手で触れられない物”の技術とは、一体、どんな技術なのでしょう？電波や音波等の波、量子のような捕まえることのできないもの、宗教や政治、易や占い等も手で触れることはできません。あまりにも、ソフト トウェア技術者が扱う範囲が広すぎることになりやしませんか？ソフトウェア技術者が何をしなければならないのかとても曖昧になってしまいます。

それに、印刷されたソースコードは“手で触れられる物”なのかどうなのか？印刷されたソースコードを“手で触れられる物”とするならソフトウェア技術者は ハードウェア技術者になってしまう。その分け方はちょっと無理があっておかしいですよね。

B　なるほど、おっしゃることはわかります。でも、「触れられる物」と「触れられない物」という尺度で判断できないとなると、ソフトウェアは何だと言えるのでしょう。また、解らなくなってきました。

A　話を少し変えますが、あなたは、ソフトウェア技術者とご自身の事を言いますが、ここで云う技術は、科学技術を指すと思いますの で、ソフトウェア技術者とは、コンピュータ科学やソフトウェア工学と言われる科学技術を駆使する者であると考えてよろしいですね？

B　はい。私自身、論理学や代数学といった数学を始めとして、コンピュータ科学やソフトウェア工学もずいぶん勉強してきました。これらの技術。ああ、科学技術ですか。その科学技術を組み合わせてソフトウェアを作ることがソフトウェア技術者の仕事であると言えると思います。

A　 私の感覚では、数学やコンピュータ科学といった部分は、科学技術としてしっくりくるのですが、ソフトウェア工学となると眉唾なものの感じがします。工学自身が応用や統計といった物を使って現実世界の諸現象から導き出すものだから仕方ないと言えば仕方ないのですが、ソフトウェアが何だかも解らないのに、そのソフトウェアの工学だなんて、一体何が科学なのでしょうか？

B　そう言われてしまうと、ソフトウェアについて答えられない私には何とも言えません。

A　話しを元に戻しましょう。

B　はい。ソフトウェアが何であるかということですね。

A　ソフトウェアが何であるかという問いに、先ほどあなたは、“手で触れられない物”がソフトウェアであると言いましたが、私は、それは強ち間違っているとも言えません。

B　どういうことですか？“手で触れられない物”がソフトウェアであると。

A　はい。

B　でも、それだと範囲が広すぎて曖昧だと言いましたよね。印刷したソースコードの話からも手で“触れられる物”と“触れられない物”では分けられませんでした。それは、あなたが言った事ですよね。でも、今度は、“触れられない物”がソフトウェアとも言っています。なんだか解らなくなってきました。

A　“触れられない物”の範囲が広かったのですから、それを狭めて、ソフトウェアを“人間の心”としてみてはいかがでしょうか？心は、手では触れられない物ですよね。

B　人間の心だって？確かに手では触れることができませんが、人間の心がソフトウェアだなんてますます、解らない。それなら、ソフトウェア技術者が、人間の心の技術者になる。そんなものは精神科医にまかせておいたらいい。ソフトウェア技術者の仕事ではありません。そんな話をしに来たのですか？

A　そんなに怒らないでください。心というと確かに、突飛な感じがすると思います。…では、心ではなく「知識」としたらいかがですか？私は、ソフトウェアとは、人間の心、特に意識とか知識と言われるもの、そのものであると考えます。仏教の世界でいうと、有名の般若心経や唯識に見られる眼耳鼻舌身といった五根の感覚器官から第六識である意識の部分と自我を作るとされる末那識の部分を「知識」と呼びたいと思います。

B　仏教の世界は解りませんが、知識ですか？それならなんとなく解るような気がします。でも、ソフトウェアが知識だとすると、私たちソフトウェア技術者は何をしていることになるのでしょう？

A　ある制約や条件を付けることで思考を停止させ、知識を言語化し記述しています。その記述言語が、私たちが日常的に利用する自然言語であり、C や Java のようなプログラミング言語も記述言語です。打ち合わせでの会話や仕様書を記述したりする行為は自然言語で、コンピュータを操作するためには、プログラミング言語で記述します。デザインパターンやアルゴリズムといったものは、慣用句やことわざ、熟語であると考えることもできるでしょう。記述は、文章として書くことだけではありません。声に出したり、歌ったり、表情や仕草で伝えたり、絵を描くといったものも含まれると

考えています。無意識でないならば、人が物を考える時は全て言葉や映像、音といった記述可能なものになっていると言っても言い過ぎではないでしょう。

B　知識がソフトウェアであるとすると言語化された記述はソフトウェアではないですよね。私には知識を全て記述できるとは思えません。記述されたソースコードや実行モジュールは、知識そのものではなく、知識の写像であると言えるのではないでしょうか。

A　そうなんです。ソースコードや実行モジュールは、知識を写像し、複写可能にしたものです。コンピュータの上で解釈された言語化によって写像された知識は、ディスプレイやプリンタ等を通して人間が認識可能となります。知識は写像され複写されたということになるです。ただ、ここで一つ注意しなければならないことがあります。それは、知識が個人に依存するものであるということです。

B　どうしてですか？象を見れば誰でも象だと認識する知識があります。日本語の知識を使って私たちは会話できている。四則演算の知識は誰がやっても同じになります。その知識が個人に依存してしまうようであれば、知識とは呼べない物になってしまう。そんな物は知識でも何でもなく個人の身勝手、思い込みです。ですから知識は、摂理や法則と呼んだ方が適切だと思います。摂理や法則を含む実世界の写像がソフトウェアですよ。

A　いやいやそれはおかしい。誰も実世界を記述することはできませんよ。実世界から受けた認識を個人が必ず媒介して言葉にしています。そこには、個人の経験を元にした知識しかない。あなたの言うような絶対の知識なんてものは存在しません。結局語っていることは、それぞれ個人の知識でしかないのです。窓の外に山が見えていたとします。その山が本当にそこに存在しているかどうか実は解りません。しかし、目に映る映像や過去の経験から山が見えるとしているのです。先ほどの仏教でも、「各々の心は異なる」という大前提があります。ただ、その心の関係性、因縁が世の中を作り出している。仏教にある空（くう）の思想も、個人の心をなくせば結局は何もなくなってしまうということから来ていると考えることができます。「色即是空、空即是色」です。また、20世紀初頭にヴィトゲンシュタインも著書『哲学探究』の中で家族的類似性として似たような事を論じています。あくまでも閉ざされたある家族、社会、世界の中でそれぞれの認識する概念が類似しているだけであって決して同じであることはないとしています。まるで言語ゲームをしているみたいだとね。会話が成立しているが、その意図がまったく別の形で伝わっていたことも、一度や二度じゃなく経験したことがあると思います。

B　確かに、そういった経験はあります。しかし、科学の発展によってそれらの発見されてきた知識は誰もが疑う余地のない正しい物・・・

A　科学が正しい知識だって？本当にそんなことを信じているのですか？これまでの科学は、帰納法による累積的な蓄積した知識でしかありません。これらは、世界の知識ではなく、科学者の世界の知識です。科学は、その世界の中で矛盾がないのであって、実世界を正しく語る知識ではありません。論理学の世界では、ある式を否定したものをもう一度否定すると元に戻るとしています。否定の否定でその意味が元に戻らずに変わってしまうものは世の中にたくさん存在しますが、論理学の世界では、変わってしまうものは取り扱わず、否定の否定は元に戻るとする矛盾ない世界のために思考を停止しています。

特に、現在の科学は、絶対神を頂点とする西洋の世界観から生まれてきたものであることを忘れてはなりません。体系化、構造化された世界が科学なのです。私たちは確かに科学の発展による恩恵を受けています。このことは否定するつもりもありません。今後も多くの科学者に科学的な接し方で記述してもらいたいと思っています。だた、科学を無条件に正しいと受け入れることは、科学という宗教の盲目な信者のようであり、とても危険なことです。

B　そう言われると辛いですね。わかりました。もう少し、広い視点で考えてみます。

A　話が少し外れてしまいましたので元に戻します。累積的な蓄積した知識である科学という制約によって矛盾のない世界があるからこそ矛盾なく記述することができます。しかし、個人の中にある矛盾だらけの心を、心のまま記述することはできません。心は矛盾があっても受け入れることはできますが、記述するためには、ある種の制約や条件を置く必要があります。家族的類似性に見られるような国家、教育、文化的な背景や業界や職場といったその人がそれまでに培った経験によって制約や条件が置かれます。これまで教わってきたコンピュータ科学やソフトウェア工学ものもその一つです。常識であると思っている事の多くは社会環境とそこで培った自身の経験によって支配されています。本当の心を記述することはできない。制約を設けて思考を停止させることで始めて記述可能になります。だから､心そのものの記述ではなく､心の写像、知識の写像と言うのです。

B　もし、こういった制約がなければ当然会話も成り立たなくなってしまうということですね。

A　そうです。また、記述言語の自由度によっても制約や条件は変わります。記述言語が物理法則に大きく影響されるもの、所謂、ハードウェアといった物は、その記述の自由度は低くなります。自然言語と表情、身振り手振りといったものを使うことが最も高く自由に記述できる言語でしょう。

B　これまでのお話を聞いていて、芸術家は、木や粘土といった自由度の低い制約の中で知識を記述している人なのではないかと思いました。自由度が低い作品から創作者の知識を受け取ることが難しくなりますね。送り手と受け手が同じような類似性を持っていれば知識を受け取ることができるのでしょうが、類似性が低いとその知識を受け取ることは難しいですね。私も美術館で古い陶器を見た時に、その良さがさっぱりわからなかったという経験があります。馬の耳に念仏です。

A　知識を送り手から受け手に渡すことは、言語を複写することによって知識を伝えることであるとも言えないでしょうか。言語は複写可能です。

B　ソフトウェアが知識のことであるという主張はわかりましたが、ソフトウェアが個人の知識であるとすると、これまでのソフトウェアという概念が大きく変わってしまうことになります。ハードウェアとソフトウェアの間にある、記述の部分をどのように考えればよいのでしょうか？この絵（図 1）の真ん中のところです。ソフトウェアの写像の部分です。

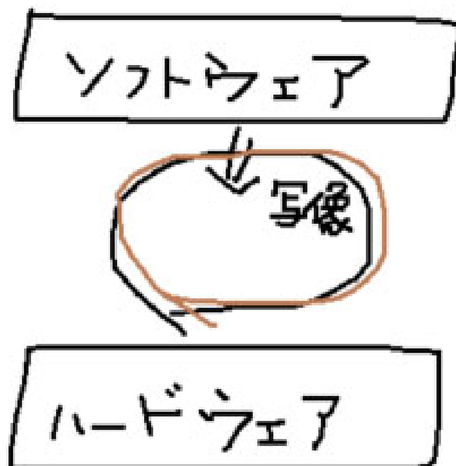


図 1

A　はい。私はこの写像の部分を「リアルウェア」と呼んで、ソフトウェアやハードウェアとは一線を引きたいと思っています。この言葉は、私が作り出した言葉です。

B　「リアルウェア」ですか…

A　はい。リアルウェアは、心、知識であるソフトウェアの記述、写像です。リアルウェアは複写可能であり、当然、人工物です。リアルウェアは、心とは違う外側の世界に存在します。マイケル・ジャクソンのプロブレムフレームのマシンと同等のものであると考えると解りやすいでしょう。ですから、リアルウェアは、ハードウェアを含みます。この絵（図 2）のように、これまでのソフトウェア／ハードウェアの境界を、ソフトウェア／リアルウェアの境界に変えて考えるようにします。このような境界で考える事の方が、ソフトウェア／ハードウェアと分けることによって発生する機能／非機能といった言葉に見える曖昧性がなくなるのです。

知識の写像となるリアルウェアが、仕様書なのか、プログラムソースなのか、コンピュータの上で動く機能なのか、絵なのか、ハードウェアなのか、ネットワークなのか、クラウドなのかということは問題ではありません。大切なことは、リアルウェアが送り手の知識を写像し、複写することで、知識を受け手に伝えることです。

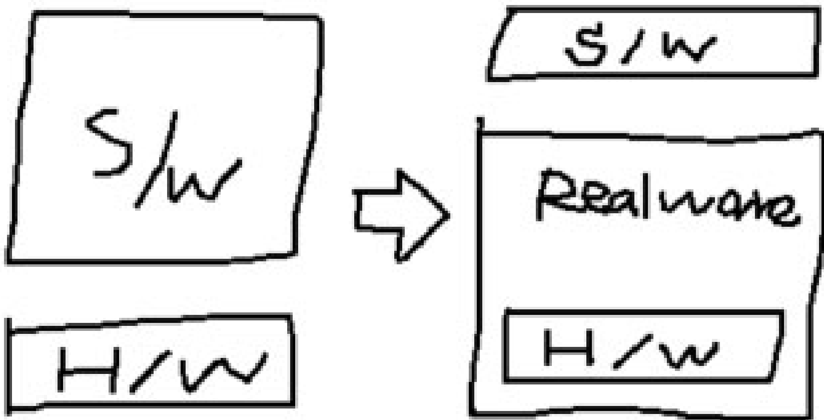


図 2

B　そう考えると、わかりやすいです。ソフトウェア技術者は、ある人の知識や集団の知識をリアルウェアに写像するということですね。ソフトウェア技術者と言うよりは、ソフトウェア通詞者といった方がぴったりくるのではないでしょうか。通詞者ですから、ある自由度の中で可能な限り適切に知識を記述できるように、たくさんの言葉や言い回し、即ち、仕様書の書き方からプログラミング言語、API やパターンやアルゴリズム、ミドルウェアの設定と言った慣用句を身につける必要があるのではないでしょうか。まるで、赤ちゃんが日本語を覚えるようなものですね。ソフトウェア技術者として一人前になるために多くの時間と経験が必要であることが納得できます。

A　さらに、知識は個人の経験によってのみ形成されると先ほども言いました。赤ちゃんが日本語を覚えたことも経験です。知識が経験によって形成されるということは、ある時点の知識をリアルウェアとして写像すると、受け手は、リアルウェアやその周辺から受けた経験よってこれまでの知識に影響を与え、知識を変化させます。その影響がわずかであったとしても知識の形成は、カオス的に変化します。ソフトウェアを知識とすると、これまでのソフトウェアにあったような静的な物の見方では対応できません。ソフトウェアを扱うためにアジャイルプロセスのような考え方が生まれてきたのは、至極当然のことです。この世は、諸行無常なのです。

B　ソフトウェアを知識として捉えると、これまでのソフトウェアに対しての考え方が変わってきますね。

A　これからのソフトウェアは、できる限りソフトウェアのまま扱う必要があります。ソフトウェアのまま扱うということは、知識のまま扱うことです。知識は一度記述してしまうと多くの知識が欠落し不完全な知識となってしまいます。知識のまま扱うには、記述しないことが一番ですが、リアルウェアとするためには知識はなからず記述しなければなりません。ですから、制約の高い言語ではなく、柔軟で自由度の高い自然言語のような言語を使って知識の欠落を抑えて記述することが必要となるのです。

B　送り手の知識を知識のまま扱えるようにすることは、もっと追究しなければならない点ですね。個人もそうですが家族的に類似した知識として扱えるようにならなければなりません。私たちはこれまで、当たり前のようにたくさんの仕様書やソースコードといった物を作ることを生業としてきました。しかし、作る、記述するということは知識を欠落させることになるから記述せずに、そのまま扱う方法を考えること、作らないということが私たちの生業になるのですね。

A　そうです、ソフトウェアを知識とすると、新たな発想や可能性に期待することができます。新たなソフトウェアの学問も必要となってくるでしょう。これまでのソフトウェアとハードウェアという関係性ではなく、ソフトウェアとリアルウェアという関係にすることで、ソフトウェアやリアルウェアの研究が進むとも思っています。

B　すみません。まだ、聞いてみたいこともあるのですが、会議の始まる時間になってしまいましたので失礼します。今度は冷たいビールを飲みながらやりたいですね。

A　それはいい考えですね。今日はお相手していただいてありがとうございました。

解説

これまでのソフトウェアは、科学の一分野としてコンピュータ科学やソフトウェア工学といった帰納法による累積的な知識蓄積の上に成り立つ常識の上で理解、探求されてきました。私は、これらの常識を疑うこともなく正しいソフトウェアの扱い方であると信じ、利用してきましたが、数十年前のコンピュータという高度な機械を操作する技術者を必要とする時代から現代へとコンピュータ技術が発展するとともに、ソフトウェアの扱い方に対する大きな疑問を抱くようになりました。

まず、第一点に、ソフトウェアの本質的困難とされる不可視を利用した曖昧な理論が多すぎるということです。システムやモデル、アーキテクチャ、要求等といった語彙からも解るように、現在のソフトウェア工学では、各々が各自の経験や語彙を用いて議論を進めています。ビジネス上の優位性を出すために、バズワードを含む様々な語彙を都合よく組み合わせ使うような場合さえあります。帰納法による累積的な知識蓄積の上で成り立っている学問であるはずが、語彙が曖昧すぎるため反証すらできません。これでは工学としての信憑性を欠くことになります。プロジェクト管理等の分野でも同様に見られます。

このことは問題ではありますが、これまでのソフトウェア界が発展していく上で、過去に囚われずに自由な議論をする時代には必要な事であったと思います。私自身もこの中に含まれ、曖昧な語彙を多用せざる負えない状況にいます。今後、科学的にソフトウェアを発展させるためには、きちんとした語彙の整理が行われるべきです。

第二点目は、第一点目にある科学的アプローチだけがソフトウェアを扱うための方法であるのかという疑問です。これまでの科学は、西洋思想の絶対神を頂点に広がるヒエラ

ルキー構造を基に考えられていることは知られています。神という絶対の存在を定義し、思考を停止することで体系的に扱うことが容易になります。全体を掌握するためには思考を停止しなければなりません。思考を停止せずに制限がなければ、語る事すら不可能になります。複雑性、不可視性といった本質的困難を特徴とするソフトウェアも無限であり体系的に扱うことに無理があると考えるべきです。

科学的なソフトウェアへのアプローチの中のひとつに、ソフトウェアの観察者自身を含まないという「思考停止」があります。観察者自身を含まないことで、ソフトウェアを静的に捉え易くなり、体系化や理解を容易に進められます。しかし、実世界にあるソフトウェアが、実世界にいる観察者からの影響を受けないということは考えられません。最近になってアジャイルプロセスやソフトウェア・ライフサイクル、コンテクスト、知働化といったことが議論されるようになったということは、観察者自身にまで思考の範囲を広げていかなければならないといった事の表れであると考えます。

私は、これまでと同じような科学的アプローチをやめろと言うつもりはありません。これまで同様に続けて探求すべきことであると思います。しかし、科学だけを絶対の物とするのではなく、世の中の全ては「思考停止」の上で語られてきたことを知り、思考を停止する範囲を広げて考えることが必要なのです。

近年では科学の世界では、西洋思想だけでもなく仏教や儒教、リゾームといった東洋思想による思考が大きな役割を持っています。西洋思想だけで考えることの限界が見えて来た今日の科学で、東洋的な思想も使って考えることは当たり前の事となっています。ソフトウェア界を見ると、そこだけが、未だニュートン時代の思想で思考を停止しているかのようです。絶対的な西洋思想に比べて、東洋思想は相対的で捉え難く、解り難い部分がありますが、これからのソフトウェアを扱う上でも、東洋的な思想は大いに役に立つと考えます。現在の科学がそうであるように、これまでの神中心のソフトウェアから人間中心のソフトウェアへと人間化していかなければなりません。

私は、先のような疑問から今一度ソフトウェアの原点に立ち返りたいと考えました。ソフトウェアとは何であるのか、ソフトウェアを知らずして、ソフトウェアを語る事はできません。これまでのソフトウェアという言葉をもう一度考え変え直してみたいと思いました。

ソフトウェアが何であるかということを追求することは、西洋のデカルト的な発想であるとの見方もできるため本論と矛盾するよう思えますが、私は、ソフトウェアが何であるかと問われれば「空」であると答えます。空の思想はデカルト的な発想とは違い、

どちらかと言えばヴィトゲンシュタインの言語ゲームのような発想です。ソフトウェアという存在はありません。あくまでも個人の主観の中にある世界であると捉えます。デカルト的な発想で有ればそこに存在を見出さなければならないことになるでしょうが、空であればその存在はあるともないと言えないのです。

書籍や論文といった章、項、段のように著作は往々にして体系立てて書かれることが良い文章であるとされています。これには西洋的、科学という背景が前提となっています。私たちは、幼少の頃から教育され、このことに慣れ切っていて疑うことすらありません。しかし、不可視であり複雑な世界を表現するために体系立てて記述することは欠落を生みます。一人の人が全てを読むことができないぐらい多くの経典が存在する仏教は、世界を表現することの難しさを表しています。そこで、本論では、プラトンやファイヤアーベント、仏教経典に倣い体系化しない問答の形式で表現したいと考えました。現在において、これが到達したいソフトウェア記述に最も近い形です。

〔参考図書〕

・Frederick Phillips, Jr. Brooks 著 , 滝沢 徹訳 , 富沢 昇訳 , 牧野 祐子訳 、人月の神話―狼人間を撃つ銀の弾はない、ピアソンエデュケーション、2002/11
・マイケル・ジャクソン著，榊原　彰監訳，牧野裕子訳、プロブレムフレーム ソフトウェア開発問題の分析と構造化 IT Architects' Archive ソフトウェア開発の課題 3、翔泳社、2006/5
・大槻 繁、ソフトウェア開発はなぜ難しいのか ~「人月の神話」を超えて、技術評論社、2009/10
・朝永 振一郎、鏡の中の物理学、講談社学術文庫、1976/6
・高橋 昌一郎著、理性の限界、講談社現代新書、2008/6
・野家 啓一著、パラダイムとは何か クーンの科学史革命、講談社学術文庫、2008/6
・Paul K. Feyerabend 著，村上陽一郎訳、知についての三つの対話、ちくま学術文庫、2007/ 7
・黒崎 宏著、ウィトゲンシュタインから道元へ、哲学書房、2003/02
・中村 元著、『般若経典』(現代語訳大乗仏典)、東京書籍、2003/02
・中村 元著、『法華経』(現代語訳大乗仏典)、東京書籍、2003/04
・中村 元著、『華厳経』『楞伽経』(現代語訳大乗仏典)、東京書籍、2003/02
・石飛 道子著、ブッダと龍樹の論理学―縁起と中道、サンガ、2007/9
・太田 久紀著、唯識の読み方―凡夫が凡夫に呼びかける唯識、大法輪閣、2000/09

・松久保 秀胤著、唯識初歩—心を見つめる仏教の智恵、鈴木出版、2001/11
・佐々木 閑著、犀の角たち、大蔵出版、2006/07
・末木 文美士著、日本仏教史—思想史としてのアプローチ、新潮文庫、1996/08

〔プロフィール〕

株式会社アッズーリ代表取締役社長。メーカ系ソフトウェア会社でプログラマを経て、フリーとして独立、1999 年に同社設立。経営やプロジェクト管理の視点からアジャイルプロセスを推進する。アジャイルプロセス協議会会長。同協議会、見積＆契約ＷＧ及び知働化研究会メンバー。ソフトウェアファクトリ研究会。

読者からのコメント

大槻氏：いくつかのセクション（話題）に区切るとよいと思います。

はま：今回のところセクションにしたくないんです。セクションを作ってしまうと境界を設定することになるので、私の意図に反します。

大槻氏：A,B じゃなくって、人物像（キャラ）を設定してそれらしい名前で呼ぶのがよいかも。放浪の修行僧「知道」とカリスマハッカー「賢徒」との対話なんちって。

はま：キャラ設定は必要かなとも思いましたが、これも読み手がなんとなーーく勝手に設定してもらって良いと思います。ただ、A と B じゃ味気ないので、キャラ設定が匂うそれっぽい固有名詞をつけてみようかと思います。

論説（コメント）

〔知働化研究会誌創刊第１号〕
知働化研究会
Executable Knowledge and Texture Laboratory

対話『リアルウェア』

Ｈ氏とＯ氏との対話

大槻 繁（おおつき しげる）

株式会社一（いち）
otsuki.s@1corp.co.jp

本作品は、濱勝巳氏の論文『リアルウェア』の下案を読んだ後に、濱氏と意見交換をしました。その折のやりとりを回想して、対話形式でまとめたものです。『リアルウェア』も対話形式で書かれているため、これに合わせた方がコメントとして親和性が高いと考えたからです。

Ｈ氏とＯ氏との対話

2010 年４月６日（火曜）、新宿京王プラザホテル喫茶ラウンジにて。

〔O 氏〕は、とあるソフトウェアエンジニアリングをコアとするコンサルティングファームのコンサルタント、〔H 氏〕は、新進気鋭のソフトウェア開発企業の社長さんです。時々、こういった浮世の対談をしているようです。

〔O 氏〕 送っていただいた『知働化研究誌』の原稿、面白かったです。対話形式というのが新鮮です。

Copyright 2010, OTSUKI Shigeru, All rights reserved.

〔H 氏〕　話の境界を作らずにだらだらと書いて行くスタイルが気に入っています。ファイヤアーベント（Paul Karl Feyerabend）の本『知とは何か：三つの対話』の影響です。

〔O 氏〕　なるほど、科学哲学を語る新しい方法を発見したようで何よりです。

さて、ソフトウェアエンジニアリングについて、その後の探求はいかがでしょう？　「否定」や「引き算」という見方は面白いと思っています。

〔H 氏〕　「作らない」ことだと思うんですよ。

〔O 氏〕　宮本武蔵みたいですね。戦わずして勝つ。作らずして価値を生む。

「非ソフトウェアエンジニアリング」とでも呼びましょうかね。

〔H 氏〕　今までのソフトウェアエンジニアリングというのは、生産性を上げるとか、大きなもの、複雑なものをどうやって開発するかという問題設定だったのですが、「小さくする」エンジニアリングというアプローチがなされていないということに気がついたのです。

〔O 氏〕　今どきの「エコ」ってやつですね。「Eco ソフト」。なんか有り難みがわきますね。ユーザとベンダとの取引きでは、作ってなんぼということだったので、作ることに焦点が当たり過ぎていたんですね。

「リファクタリング」は、機能を保持したまま良構造化することですが、こういったものにも価値があるという見方になると思います。もっと突き進めれば、使われていない機能を落としてしまって本当に使うものだけにする「広義のリファクタリング」という引き算の考え方も良さそうです。

〔H 氏〕　社会的にもお掃除とか、ゴミを無くすことって価値のある活動ですから、ソフトウェアだって同じでしょう。機能を食べるソフトなんていうのがあってもよいかもしれません。ソフトウェアそのものには価値はないのであって、それが置かれているコンテクストや様相に価値があるのです。企業活動でも、それだけでは価値を生まないのです。

〔O 氏〕　ΛＶモデルの「Λ」が中心ということですね。Λの部分は、意味（価値）・行為（要求）・感覚（テスト）で、「意味」の部分は主観世界の奥深いところにあるので難しい問題です。少なくとも、ユーザ／ベンダという取引き関係という見方をすると、たくさん要求をもらって、実現に勤しむことによって利益を得るという図式があるので、本来の目的である「意味（価値）」に焦点があたらなかったと言えます。昨今の内製化のトレンドというのは、要求を大きくする志向性を排除できるので、本来のあるべき姿なのかもしれません。

〔H 氏〕 従来の意味でのソフトウェア開発ベンダの時代というのは終わっていると思います。最近のベストセラー『Free』風に言えば、無料ソフトウェアの時代かもしれません。無料でパッケージ提供してしまうことや、Web サイトでクライアントが直接、プログラミングをしてしまうようなシームレスな流れが重要になってくるでしょう。最終的には従来の意味でのベンダは無くなり、ユーザ価値をもたらすコンサルティングやビジネススキームをデザインすることが主要な仕事になるのだと思います。また、金銭的価値とそれ以外の価値も多様化していくので、単純な価格設定はできなくなるし、一見「無料」の世界ももっと広がっていくような気がします。コンサルティング料は「お布施」のような位置づけになっていくかもしれません。

〔O 氏〕 かつて札幌の見積り・契約 WG 合宿の折にまとめた SPC（特定目的会社：Special Purpose Company）が理想的なソフトウェアに関するビジネススキームだと思っています。ユーザ企業と開発企業、さらには、保守企業などが出資して一つの企業体を作り、そこで継続的に開発や保守、広義のリファクタリングなどを行い、そこから得られた利益を出資企業に還元するというものです。これは広い意味での内製化とも言えますし、先ほど述べた要求や開発対象が大きくなってしまうリスクを避けることもできます。

〔H 氏〕 元来、アジャイルプロセスというのは保守に近いですし、ソフトウェアのライフサイクルを見守ることが本質に思えています。「アジャイル SPC」がよいのかもしれませんね。ビジネス領域でよく言われるスマイルカーブで言えば「保守ビジネス」がこれから主題になっていくと思っています。

アジャイルプロセスについても、今までの世の中の議論では、作り手側の話ばかりでしたが、ユーザ側のアジャイルプロセスというのがあまり議論されてきていません。

〔O 氏〕 確かに、その通りですね。Λで言えば、意味の獲得や修正方法、要求の発し方、実装されたソフトウェアの動作からの感覚を得た時の対処の方法、要求とテスト、あるいは、テストから次の要求へのプロセスなどは、ほとんど解明されていませんし、規範もないというのが実状です。

〔H 氏〕 「テスト」という言葉は､あまり好きではないんですよ。五蘊（ごうん）で「見る」とか､「心眼」のような感じ。

〔O 氏〕 Λ V モデルで要求 R、テスト T ととりあえず V 字モデル側の制約で「テスト」としてのですが、私もこの言葉は適切ではないと思っています。言い訳っぽく T は Tracing や Tracking とも言うとしています。

〔H 氏〕 僕がテストを好きでないもう一つの理由は、開発においてもテストって本当はいらないんじゃないかと思っているからなんです。

〔O 氏〕 テストがいらないというより、いくらやってもきりがないということですよね。原理的にプログラムが仕様を満たしていることをテストによって証明することはできません。

所詮テストは自己満足の域を出ません。一部の性質、例えば、関数等価性なら数学的証明によって論証することはできます。「ソフトウェアクリーンルーム手法」の原理です。関数等価性以外の性質についてもなるべく数学的証明でやるというのが正しいアプローチです。それがだめなところはモデル検証とか実行確認に頼らざるをえないかもしれません。

さらに、仕様そのものが、そのソフトウェアが置かれているコンテクストで、要求に適合しているかどうかということは、要求を欲した本人しか確認することができません。

〔H 氏〕 ソフトウェアのテストや実行に、きりが無いというのは、結構本質的な問題です。コンテクスト含めて変化し続けるものです。要求というか、解くべき問題というのが次から次へと出現し、それをソフトウェアで解き続けるというライフサイクル的観点が必要だと考えています。

〔O 氏〕 その意味で言うと「狭い意味でのソフトウェアづくりは創造的活動ではない」ということですよね？

〔H 氏〕 ソフトウェアがソフトウェアだけで価値を生まないのと同じように、それを作るだけでは価値のある知的活動だとは言えないでしょう。いくらプラクティスをいっぱい取入れてアジャイルプロセスをやったとしても、それだけでは価値は生まれない。

〔O 氏〕 価値というのは経済学的に見てもダイヤモンドのような希少性と、水や空気のような必要性の両面があります。ソフトウェアが、実世界にある問題を解くものであるという観点からすると、同じ問題を解き続けても、価値は生まれません。新しい問題を設定し、新しい解を得ることに挑戦し続けなくてはならないでしょう。これが経済学的な原理から言えることです。

無論、労働価値説は通用しないので、価値とコストとは本来関係がありません。現実には、いまだに人月ベースの取引きが多いので、ソフトウェア業界は未だに労働価値説にあると言っても過言ではありません。

〔H 氏〕 とりあえずセル・月まで行きましたけどね。問題は、その先です。よいものをつくれば価値が高くなるというわけでもなさそうです。

〔O 氏〕「よいとは何か？」という問題ですね。よく言われるのが、技術的に優れていても、マーケットシェアが確保できるとは限らないということです。

〔H 氏〕 よいプログラミング言語は流行らないとか。

〔O 氏〕 まぁこれは戦略とかビジネスモデルの問題とも言えます。よいソフトウェアを、価値が高くなるようにしていくことが、価値指向の戦略として重要だということになります。

〔H 氏〕 そのような観点から言うと、そもそも世に言う技術指向の「理系」の人がこの業界に向いているというのも幻想だと思っているのです。

〔O 氏〕 なかなか過激な見解ですね。「理系」の定義にもよると思いますが、数学的素養は必要でしょう。欧州では文学部哲学科の中に数学があるようですし、数学は「文系」かもしれませんよ。

〔H 氏〕 ソフトウェアエンジニアリングが「記述」に関する活動ですから、文学や社会常識、文化的素養が求められると思います。プログラミング言語の記述は手順を表現したもので済みますが、仕様や制約記述をしていくことも大切なことです。

〔O 氏〕 そしてたぶん、実世界を観察し、問題を発見し、表現する「抽象化能力」が、ソフトウェアエンジニアリングで最も必要な能力です。

もう少々範囲を限定して述べるならば、アジャイル・ソフトウェアセル生産のシェフには抽象化能力が必要ということです。コックの世界はソフトウェアづくりが創造的活動ではないという見解に賛同します。あえて補足しておきますが、創造的でない仕事だから地位が低いとかそういった話ではありません。

〔H 氏〕よく「定形業務」と「非定形業務」といった分類がありましたが、「手順的」と「創造的」というのが対応しているのかもしれません。

でも「創造的」な活動というのは、何をやっているのでしょうね？

〔O 氏〕 難しい質問です。

強いていえば、問題を作るということかな。問題の前提となっている世界や空間を作ることも創造的な活動だと思います。

〔H 氏〕　そうなってくると先ほどの「抽象化能力」がますます重要ですね。

〔O 氏〕　実を言うと、最近、「記述」という活動そのものが何かというのも気になっていて、数学的思考や抽象、定式化といったことの本質が何かがよくわからないのですよ。

〔H 氏〕　あるものを、あるがままに述べたり、書いたりすること。だと思います。

〔O 氏〕　おぉ、まさに仏教的ですね。石飛道子先生の『ブッダと龍樹の論理学』をここのところ読んでいるのですが、まさに「ぶっ飛び」です。西洋論理学とはあまりに違う。

〔H 氏〕　僕にはすんなり腑に落ちます。

〔O 氏〕　流石ですね。私はどうしても西洋論理とブッダ論理を翻訳しようとしてしまうので、なかなか難解なところがあります。語の順番に意味があるとか、接続詞の用法も異なります。A かつ B が、B かつ A とは違うと言われてもねぇ。

〔H 氏〕　そりゃ違うでしょう。頭に思い浮かぶ順序が違うんだから。

〔O 氏〕　そうそう、そこなのですよ。言語ゲーム的に考えれば、結構すんなりと対応することが最近わかってきました。つまり、西洋論理というのは、表層表現と深層の論理関係とを区別していて、これに対してブッダ論理は言語化（あるいは認識）と論理関係をいっしょにしている体系なのです。そうならそうと言ってくれよと思いますが、そこはなかなか奥ゆかしい。

〔H 氏〕　ということは、ブッダ機械（マシン）をつくれるということですか？

〔O 氏〕　オートマトン理論で言う「計算」まで行けるかどうかはわかりませんが、原理的にはつくれると思います。面白いところは、「認識と記述の同時進行」の体系になるということになることです。これについては、しばし探求の時間をください。数年は楽しめそうです。

〔H 氏〕　「実行」とは「記述」することだったりして。

〔O 氏〕　深いですね。「実行可能知識」の定義にもこの実行と記述との関係について入れていきたいところです。

〔H 氏〕　人工物（アーティファクト）の中でソフトウェアがソフトウェアであるゆえんは、この「実行」概念が握っていると思っても過言ではありません。花屋を経営するのと、ソフトウェア企業を経営することの違いは、ここにあるのかもしれません。次世代の花屋はもっとソフトウェア的かもしれませんけどね。

〔O 氏〕　認識の問題とかもいっしょに考えていくと、「暗黙知」と「形式知」との関係も関わってきそうです。

〔H 氏〕「暗黙知」にこそ価値があると思っているのです。「形式知」を「暗黙知」にしていくところに、価値が生まれるといってもよいかもしれません。

〔O 氏〕　共同化（暗黙知→暗黙知）、表出化（暗黙知→形式知）、連結化（形式知→形式知）、内面化（形式知→暗黙知）というサイクルを繰り返しながら発展していくという SECI モデルで、重要なところは「内面化」であるというのに私も賛成です。

〔H 氏〕そうですよね。暗黙知を形式知化することに価値があるという人もいるのですが、僕は逆の発想なんですよ。

〔O 氏〕　シュレーディンガーが「学習とは無意識化することである」という定義を述べているのですが、このことだと思うのですよ。形式知、手順、規範といったものを一生懸命学習し身につけることによって、それが次第に無意識にできるようになって、その状態で、本当の暗黙的が熟成されていくのですよ。

〔O 氏〕　ということは、ソフトウェアエンジニアリング、否、非ソフトウェアエンジニアリングでは、「暗黙知の手法」が望まれていますね。

〔H 氏〕　ひたすら座していよとか。スリッパを揃えるのも修行とか。あるいは、「道」のような形式をマスターすることによって、その奥にあるものをつかむといったアプローチになるでしょうね。

〔O 氏〕　おっと、そろそろ時間ですね。本日は、とても有意義な対談でした。どうもありがとうございます。通奏低音として、仏教の思想、否定や引き算の観点、形式知と暗黙知との関係、言語ゲームの考え方があり、リアルウェアやΛVモデルといった枠組みも定まってきましたね。

〔H 氏〕　旧来のパラダイムの呪縛から解き放たれ、神話をまとめ、次世代の未解決問題を設定するという「ソフトウェアエンジニアリングの呪縛 WG」も本日のような対談の延長線上に位置づけられるでしょう。

〔O 氏〕　そうですね。「未解決問題」は新たに設定されるとして、旧来と新世代とにまたがる普遍的な上位レベルの問題設定として「本質的困難」があると思っています。

〔H 氏〕では、お互い現世に戻ることにしましょう。どうもありがとうございました。

論説

〔知働化研究会誌創刊第１号〕
知働化研究会
Executable Knowledge and Texture Laboratory

システム･エンターテイナー

システム開発を楽しむ

時本 永吉（ときもと えいきち）

合同会社カモス
eikichi63@exekt-lab.org

概要……………………………………………………………… 58
はじめに………………………………………………………… 58
Ⅰ．欲望を捨てよ、欲求に従え……………………………… 59
　Ⅰ．Ⅰ．ジェネラリストを目指して……………………… 59
　Ⅰ．Ⅱ．欲望を根拠とする………………………………… 61
　Ⅰ．Ⅲ．欲求に従う………………………………………… 62
Ⅱ．道化師の視点で…………………………………………… 65
　Ⅱ．Ⅰ．要求よりも欲求を捉える………………………… 65
　Ⅱ．Ⅱ．妄想よりも事実を根拠とする…………………… 66
　Ⅱ．Ⅲ．解決よりも認識を優先する……………………… 68
　Ⅱ．Ⅳ．完全よりも妥当に処置する……………………… 68
　Ⅱ．Ⅴ．管理よりも制御をすること……………………… 70
　Ⅱ．Ⅵ．責任よりも覚悟を持つこと……………………… 71
　Ⅱ．Ⅶ．チームを築くか、パーティーを演じるか……… 72
　Ⅱ．Ⅷ．欲求に妥当か、契約に完璧か…………………… 73

Copyright 2010, TOKIMOTO Eikichi, All rights reserved.

Ⅲ．エンターテイナーへの道 …… 74
Ⅲ．Ⅰ．苦行契約を脱却する …… 74
Ⅲ．Ⅱ．システムエンジニアであること …… 75
Ⅲ．Ⅲ．システム・エンターテイナーであること …… 75
Ⅲ．Ⅳ．パーティーのプロセスづくり …… 76
Ⅲ．Ⅴ．パーティーの参加料 …… 77
Ⅳ．システム開発を楽しむ …… 78
Ⅳ．Ⅰ．アジリティ・ウォーターフォール・モデル …… 79
Ⅳ．Ⅱ．ホウレンソウ・モデル …… 82
Ⅳ．Ⅲ．物語と辞書 …… 83
Ⅳ．Ⅳ．確認と根拠 …… 83
Ⅳ．Ⅴ．マニフェスト …… 85
〔参考図書〕 …… 85
〔プロフィール〕 …… 86
読者からのコメント …… 86

概要

自分自身がソフトウェア開発を楽しむものとして、システム・エンターテイナーを名乗る。さまざまな手法やマインドなど「顧客満足のため」とか「作業の効率化」などが挙げられるが、結局は「自分が満足できているか」を考え、実践することが重要である。「顧客満足のため」という思い込みから脱却し、どう考えていくかという思いを述べる。

はじめに

自分の技術力を磨くと、苦しみが増えた。いろんなことを知りたくて、いろんなことを楽しみたくて、ある時は自分のために、また、ある時はみんなのために、自分を磨いてきた。技術力とは実装やテストだけではない。モデリングやデザイン、リスクコントロール、プロジェクトマネジメントなど、システム開発に関わるあらゆる能力のことを指す。しかし、それは政治的な理由や大人の事情というもので受け入れられることは多くなかった。それでも、私が担当すると早く品質の高いものを実装することが出来ていたので、それなりの評価は得ていた。そして、「お前は設計よりも技術向きだから」と言われ、殆どのプロジェクトで実装担当になっていた。参加するプロジェクトの殆どは途中参加で、文化や顧客と

の関係が出来上がっていたため、参加時のロールが変わることはなかった。そこで、私はそのロールのまま、他のロールに手や口を出すことにした。あまりにも突っ込むところが多すぎて、一部の人からは「すぐ文句をいう男」と称されもした。みんなのためを思ってのことだったのに、どうすれば良いのか別の会社の人に相談すると、「変えることはできない」とか「そんなことを考えるのはおこがましいことだ」と言い放たれてしまい、私の苦しみは一層増していった。

この日々の思いは私を否定的にさせてきた。しかし、私はただシステム開発を楽しみたいだけだ。楽しくないのであれば、楽しくなれるよう考えたい。仕方ないと言って自分を正当化しようとしても、私は満足できない。みんなで楽しめることを楽しいこととする。そういったマインドを持ち、振る舞うものとしてシステム・エンターテイナーを名乗ることとした。エンターテイナーに至るまでの想いと、今後、どのように振る舞っていきたいかをこのエッセイで述べる。

Ｉ．欲望を捨てよ、欲求に従え

システム開発をあらゆる面で楽しみたかった私はスペシャリストではなく、ジェネラリストを目指した。しかし、自分の信念を貫く欲求と、自分は人の役に立っていたいという欲望の区別ができていないことが私を苦しめた。

本章では、システムエンジニアとして生きてきて感じた、この苦しみについて述べる。

Ｉ．Ｉ．ジェネラリストを目指して

私は無能で愚かなシステムエンジニアである。

私は誰かに何かを与えることはできない。私はただ自分の思いで行動しているだけである。つまりは、その覚悟を持って生きることを決断した。

私は救いを求めている。本文は私の懺悔に過ぎない。この行いによって、私の今が改善されることだけを願っている。私が私として生きていられるための行いであり、誰かを救うためではない。そもそも私が誰かを救えるわけがない。みんな求めるものは異なり、その過程として求めるものも異なる。「最終的に同じ結果」であったとしても、その過程は大きく異なり、「何をどのように求めるか」が価値観の違いである。そして、私は他人のそれを理解できるとは思わないし、他人が私のそれを理解できるとも限らない。もしかし

たら、近しい価値観を持つものと共感することはできるかもしれない。それが「趣味が合う」という現象だろう。しかし、完全一致するわけではない。それは思い込み（個人の期待すること）でしかない。

同じウサギ好きでも、「鼻をすぴすぴさせるのが好き」な者もいれば、「ご飯を抱きかかえて必死に食べる姿が好き」な者もいれば、単純に「ふわふわしたものが好き」な者もいる。みんなウサギが好きなので動物園でウサギに会いに行くまでは仲良くとも、ウサギの魅力を談義すると意見が分かれる。時には喧嘩になることだってあり、「あいつは何もわかっていない」と言い出すかもしれない。

私はウサギが好きなので、ウサギを例え話にしたが、意識の相違は人と人が付き合う上で必ずといっていいほど起こる現象である。当然、システムエンジニア間においても起こる。いや、システムエンジニア間だからこそ、特に起こる現象なのかもしれない。顧客とシステムエンジニア、またシステムエンジニア間で個々の思い込みでシステムが作られていく。思い込みで創られるため、その根拠が曖昧になる（思い込みで作ったといえば、自分の責任になるから言わない。相手が言ったことにする。あるいは相手に言っただけで了承を得ていないことを、「確認済み」と言ったりする。その結果、言った・言ってない、の言い争いが起こる）。そして、「完璧なものは作れない。状況にあわせて妥当に作るべきだ」という考えのもと行われるのが、現在多くのシステムエンジニアが行っている方法論だ。

方法論とは目的を達成するための手段である。「どのようにしたいか」を考えるのではなく、「どのようにすべきだ」となっている現状では、こんな方法論とはいえないようなことも方法論になりかけている。そして、これに対してさらに「方法論」が形成され、「すべき」の上に「すべき」が築かれ、修正不可能な「すべき」方法論が生まれている。そして、すべての事象は複雑になる。「複雑なものは高尚なもの」と認識されてしまう。

色んなものが複雑になり、一人がすべてをこなすのは困難だと思われる。そしてチームが形成され、チーム内で役割が割り当てられる。やがて、作業が分担されることは当たり前になり、作業ごとに業種が作られる。「高尚なもの」に垣根が築かれる。その結果生まれたのが、「アーキテクチャのわからないプログラマー」や「テストのできないデザイナー（設計者）」、「マネジメントのできないテスター」などである。それらすべてができてこそ、「システムエンジニアのプロフェッショナル」であるはずなのに、一芸だけで「プロフェッショナル」を語っている。「歌って踊れる」ことまで求めないが、プロフェッショナルのシステムエンジニアとは、ジェネラリストであるべきだと捉えている。「ゴッドハンド輝」という医療漫画でこの言葉を目にしたとき、そう感じた。世間一般には、会社内でジェネ

ラリストというと、経営陣・役員などをイメージされるようだが、この作品では「全身科医」のことをあらわしている。すなわち、内科だから内科の領分しか診断できない、外科だから外科の領分しか診断できない、ではなく、すべてにおいて精通していること。それにより、診断ミスや発見の遅延を避けることができる。ジェネラリストはすべてにおいて誰よりも優れている必要はない。ある程度のことはできるべきだが、やはり自分より優れた医者は存在する。それはスペシャリストと言われる。たとえば、脳外科のスペシャリストや麻酔のスペシャリスト、縫合のスペシャリストがおり、症状に応じて、さまざまなスペシャリストと協力して手術を行う。これはシステム開発においても見習うべきところがある。すべてを見渡して考えられる人はシステム開発において重宝されるだろう。たとえば、ただの一芸エンジニアより、ジェネラリストははるかに多くのリスクに気づくことができる。リスクとは「危機」であり、「機会」と「危険」という言葉からなる。プロジェクトの障害となるもの、プロジェクトをうまくまわしていくための方法を気づくことができる。プロジェクトマネージャーがこうあることは最良だが、ただの一メンバーがこれであってもかまわない。ただし、メンバー間でそういった意見交換ができる場が用意されていることが必須である。

Ⅰ．Ⅱ．欲望を根拠とする

私が何を求めているのか、まだよくわからない。自分でもわからないことなのに、どうして救ってほしいと言えるのか。何かをしていただいたとき、私はどのようにして・何を根拠に「救われた」と判断するのか。あるいは、何かをしていただいたことに対して「何かを解決していただいたが、私はまだ満足していない」と述べるのか。おそらく、それが一生続くだろう。何かを満たしても、次の欲望が生まれる。それはよりよいものを求め続けるという意味では進化と呼べるが、進化し続けるということは欲望が止め処ないことを意味する。

このことより、進化は「満たすべき欲望」が何であるかに気づくことが第一となる。そして、それを「どのようにして満たすか」を考えることが第二であり、「満たしたことにより今後どのように進んでいくか」をフィードバックしていくことが第三となる。すなわち、問題・課題に気づくことが最大の問題であり、それを解決する方法を考えるのが次の問題である。そして、その考えた方法を実行することが、続く問題となる。システムエンジニアは主に第三の領域を対象とする。第二の領域も対象とすることはあるが、別会社がやっていることが多い。

第一の問題である「満たすべき欲望が何かに気づくこと」は非常に難しい問題である。システム開発では「要求定義」とか「要件定義」とか言われるが、これらがシステムエンジニアによって「定義」されたものである時点で、「欲望」とは程遠いものになっているだろう。すなわち、「こうしなければならない」という思い込みで整理されてしまっているからだ。これは顧客側に書かせても同じ結果になる。顧客側も今までのシステム開発依頼の経験上、「システム開発はこういうものなのだ」、「これを実現するには困難・不可能だから仕方ないのだ」と刷り込まれているため、自分の欲望を抑えて「要件」を述べてしまっている。そして、記録されるものは、この整理されてしまったものでしかない。「こうしなければならない」を除いた、「こうしたい」という文章はどこにも残っていないのだ。

そのため、顧客も「期待通りのシステムができていない」と言えない。要求定義・要件定義書に書かれていることができていなければ言うことができるが、逆に書いてあるが、それは欲求を満たせないものだった場合、言うことができない。そういう場合、大抵は顧客が諦めるか、仕様変更・運用保守中の機能改修としてシステム屋に依頼を出す。その結果として、顧客はシステム屋に予定以上のお金を支払わなくてはならなくなってしまう。

欲求を満たさないから、バグとして対応しろ、とは言えないし、システム屋にそんな無茶は言えない。そんなことを許すと、大変なことになってしまう。しかし、システム屋はもっと早い段階で「欲望を満たしている」が検証できなかったのだろうか。要求定義・要件定義をする際、そういったことを確認しなかったのだろうか。あるいは、確認する手段を持たないのだろうか。定義することはできないため、必須とは言わないが、この手段、というかその覚悟を持つこと、すなわち、「私はシステムを提供することをミッションとするのではなく、顧客の欲求を満たすため、システム開発をその手段の一つとして持っている」という理念を持っているかどうかが、他のシステム屋との差別化となるだろう。

Ⅰ. Ⅲ. 欲求に従う

欲望と欲求は異なる。自分の価値観に基づいた自己実現を求めることを欲求といい、社会価値観に基づいた評価を望むことを欲望という。たとえば、「英語で日常会話できるようになりたい」は自己成長として欲求になるだろうが、「英語で日常会話できるようになって、他の人に尊敬されたい・されるかもしれない」は他人にそれをゆだねることから欲望となる。いずれにせよ、「欲」であることに変わりはないが、たいていの欲には欲求の側面と欲望の側面を持つ。

社会で生存するため、殆どの業務は欲望を満たすためにある。しかし、欲望は概して、即物的であることが多い。なぜならば、社会的価値観とは現状において価値があると評価されるものである。これは目に見えるものである。たとえば、「売上を上げる」や「無駄を排除する」など、数値で見ることができる。この背景として、「経営陣につつかれた」とか「経営戦略を実施するための下準備」などがあり、何らかの動機に直結することから、価値よりも実現することに重きをおくことになる。そのため、実現したことを確認するために、即物的になってしまう。

しかし、社会的価値観に基づくため、社会状況が変われば、すぐ次の欲望を求める。そして、システムエンジニアが整理した要求定義とは、それを得るためだけのものである。欲望は社会に柔軟に対応するが、コンピューターシステムはこれに柔軟に対応することはできない。そのため、社会状況が変われば価値を失い、作りなおしとなる。

コンピューターシステムを作りなおすとき、既存システムの仕様書と現在の顧客の欲望をもとに新しい要求定義を行う。このとき、既存システムの要求定義には、根拠とした欲望について書かれていないため、「何を求めたか」しか分からず、「何を価値として（根拠として）、求めたか」が分からなくなる。そのため、「価値観」という価値を失う。企業の価値観は、その企業の文化に値する。これにより、「今まで出来ていたことができなくなってしまった」ということも多い。「出来ないように変更した」のではなく、「出来なくなった」のだ。

たとえば、C/S システムから Web ベースシステムに乗り換えると、「キーボードだけでの入力が困難になった」、「値のコピー＆ペーストができない」、「応答時間が長くなった」などのことが起こった。問い合わせても、「こういうものだ」という回答しか得られず、顧客側がなれるしかなかった。そして、この状態で要求定義は残った。時代は流れ、Web ベースでも C/S システムと遜色なく、または仮想化技術の革新による別解が、用意できるようになった。しかし、以前の欲望を満たすことはできない。欲望はどこにも記述されておらず、満足できない状態の要求定義しか残っていないからだ。そして、システムエンジニアは既存システムの要求定義を考慮する。前回苦渋を味わった顧客は「コンピューターシステムとはこんなものだ」と思いこんでしまい、また、既存システムに慣れてしまい、そんなことを言わなくなってしまった。欲望は大きくなり続けるはずなのに、コンピューターシステムに対する欲望は小さくなってしまったのだ。

欲求は欲望と異なり、実現することよりも価値を求める。システムエンジニアは要求定義から「顧客の価値」を見出してあげるべきだったろう。そして、「顧客の欲望」を見出

すのだ。これを満たすことがシステムエンジニアとしてのプロフェッショナルなのではないだろうか。いろんな技術やいろんな業務知識は、ただこれのためだけにある。「理解が早い」とか「聞かなくても分かる」というのは二の次だ。そして、システムエンジニアのプロフェッショナルはこれを自己形成・自己の価値として抱くべきだ。すなわち、システムエンジニアは「顧客の欲望を満たす」ことを欲求としてもつべきだ。

他人の欲望を自分の欲望としてもってはいけない。欲望としてもつことは他人に評価されるために、自分の思いを込めてしまい、別物となってしまう。その結果、顧客の欲望を満たすことができなくなってしまう。

自分の思いを込めてしまうと、相手（顧客）を助けている気分になってしまう。「私はあなた（顧客）のために精一杯がんばります」、「あなたのために、この方法を提案します」、「なぜ提案したとおりになさらないのですか」、「そのやり方だと、こんな問題がありますよ」、「助言したのに、なぜ今になってそんな事を言うのですか」、「あなた方では理解できなかったのだから仕方ないですね」、「あなた方がそう決定されたのだから仕方ないですよね、私は出来るだけのことはしました」・・・顧客にためにと始めた行為が、「よく評価されたい」という欲望によって次第にねじ曲がっていき、苦しみ、嫌な思いをする。「このプロジェクト、早く終わらないかな」、「あとXか月の我慢だから」などと思ったことはないだろうか。こんな思いをしたくないのであれば、システムエンジニアは自分の仕事に欲望を抱いてはいけない。

かといって、作業は人と人の付き合いなので、「この人たちのためにがんばろう」という欲望が生まれるだろう。私もそんな欲望を否定するつもりはない。ただし、欲望と欲求は別物であることは認識すべきだろう。そして、相手（顧客）の価値観を否定してはいけない。顧客の価値観を自分が決定してはいけない。それを理解した上で、自分の欲求を満せる欲望を抱くことは現実解として妥当である。

Ⅱ．道化師の視点で

欲望であろうと欲求であろうと欲であることに変わりはない。欲を持てば、苦しみの原因となる。欲望を捨て、欲求に従うことでいくらか軽減されるも、完全になくなることはない。全ての欲には欲望的側面と欲求的側面があり、それを常に意識しつつ欲望を捨てることは不可能だからだ。仏様は解脱し、欲を捨てておられるようだが、私は人間でいたい。私は常に楽しくありたいのだ。私の行動規範は「それは自分にとって楽しいと感じられるか」である。常に「楽しい」と感じる行動をとり、「楽しい」と感じることができないことはしたくない。

私は欲望にまみれて生きている。「楽しい」と感じることを求めるが故に、欲望に溺れて生きることを選択している。それが苦しむ原因だと知りながら、マゾヒストのような自分を嘲り笑う。そのような道化を演じながら、「ああはなるまい」と誰かに感じ取ってもらえればいいな、という欲望を抱く。

本章では、自分を苦しめてきた経験に対して、道化師の視点で問いかけたことを述べる。

Ⅱ．Ⅰ．要求よりも欲求を捉える

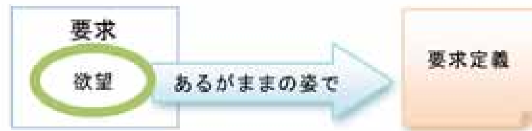
システム開発は要求定義からはじまる。システムエンジニアが顧客から「要求」を聞き、整理したものを要求定義書といい、この作業を要求定義という工程で行う。この要求を満たすことができれば、顧客は満足するはずだが、どうにもうまくいかないことが多い。後になるにつれ、要求が変更することがよくあるのだ。そういうときは次フェーズに回したり、追加請求をして受けたり、受けなかったりするのだが、これは顧客に問題があったのだろうか。

顧客はビジネスのプロフェッショナルだが、コンピューターシステムのプロフェッショナルではない。また、ビジネスのプロフェッショナルであっても、革新的・適合的なソリューションを持ち合わせているわけではない。そんな顧客に、プロジェクトの早い段階から「要求」という整理された状態の知識を導き出てもらえることを望むのは酷な話である。

システムエンジニアは、どう表現すればよいか悩んでいる顧客の要求を引き出してあげるべきだろう。そうして引き出した要求を満たせば、顧客が満たされるだろうか。残念ながらうまくいかないだろう。なぜなら、「どう表現するか」を考えてしまっているためである。これに限らず、考えてしまうと、もともとの形から変形してしまうことがある。多いのは、「自分はこんなにすごいことをやっているのだ」という思いから難しくしてしま

うことと、「自分はこんなに物事を単純にとらえられるのだ」という思いから簡単にしてしまうことだろう。また、システムエンジニアにもこの傾向はあるので注意が必要だ。顧客の変形されていない要求を単純にそのまま捉える必要がある。

顧客の変形されていない要求は、顧客の欲望である。欲望を満たしたとき、顧客は初めて満足できるのではないだろうか。また、欲望さえ捉えておけば、多少の要求変更にも耐えられるだろう。あるいは、それは書式が変わった程度の問題で、「変更」と言うべきものではなかったかもしれない。



Web ブラウザを利用したシステムが主流になったが、「ブラウザでは困難なのでできません」という言い訳を与えることになってしまっている。もともと、Web ブラウザは C/S システムのように各端末にソフトをセットアップしなくてよく、各端末の設定に依存しないことで配布の容易性の向上・設定コストの低減が魅力であったが、最近では「ブラウザ依存（バージョン違い含む）」を要求し、各端末に色々な設定を要求している。それでも C/S システムの時代に比べれば随分ましだが、「昔と比べてマシだから」ではなく、「今、満足したい」のだ。昔より色々な便利を知ったなら、今より便利になることを望むようになるのは当然のことである。そこで、「何を望むか」を考えるのである。考えたものが要求となる。望むことが欲望となる。

「どうしたい」･「どうなりたい」を表現する。たとえば、「売上を向上したい」でも、「無駄な作業はしたくない」でも何でもよい。それに対して提案はいくらでもできる。つまり、考えることはいくらでもできる。逆に、要求を最初から言われてしまうと、考えることができない。思い込みで、「こうしかない」が決められてしまう。そして手段は目的となり、目的を見失い、作ったはいいが、面倒くさいルールに縛られた使い勝手の悪いインタフェースで構成されたシステムが構築される。

Ⅱ．Ⅱ．妄想よりも事実を根拠とする

人の考えはすべて妄想であり、現実に起こったことだけが事実である。ここでいう妄想とは、他の人が「根拠になる」と合意できないものを根拠とした考えのことを言う。例え

それが論理的であったとしても。

論理は時に非常に暴力的である。構文論的に正しければ意味が異なっていても、論理的である。思い込みを利用すれば、あらゆる事象を恒真（トートロジー）として仮定することで、事実とは異なっていても、意味論としても論理的となる。たとえば、「他の人を手助けするために自分の作業を出来るだけ早く終わらせなければならない。したがって、残業して自分の作業を早く終わらせなければならない。」は正しいだろうか。何をもって「正しい」というのかという議論もあるし、「個人の進捗ではなくチームの進捗」という考えも分かる。けれど、何か違和感はないだろうか。

根拠は誰もが認識できる「事実」であれば、疑う余地がない。たとえば、設計書は動かして、業務を遂行できることを確認したという事実があって、はじめて完成したと言える。また、プログラムは実行して期待通りの動きをすることを確認したという事実があって、はじめて完成したと言える。しかし、特にウォーターフォールでは、「確認した」はテストとして別の工程で行われる。そのため、「遂行・動作することを確認していなくても、レビューアーが合格と言えば完成したと言える」というレビューアー・トートロジー論が成立してしまっている。レビューアーは顧客ではない。最終的には顧客にもレビューしてもらうが、そこで意識違いや思い違いが発覚すると、手戻り作業が発生する。

そう考えると、テストとは試験・検証ではなく、根拠の提示・意思確認・意識合わせといわれた方がしっくりくる。「動けばいい」と言われたとき、「動く」とはどういうことなのか分からなければ、「動いた」ことを保証できない。何をもって「動いた」と判断したか、を述べるための作業、あるいは認識合わせの作業が必要だろう。

では、「動いた」という根拠なしに「顧客の言ったとおり≠要求通り≠設計通りに作った」（すなわち、まやかしの要求をもとに作られた妄想の設計書を元に作ったチェックリスト）をもとにテストするどうなるのか。当然、欲望と異なるため、変更依頼がくる。顧客にとっては満たすべき欲求は何も変わらないが、「要求定義書と異なる」ため、仕様変更扱いになる。加えて、「各工程」で各人の「思い込み」が組み込まれているため、また、その確認工程が離れた工程で実施されるため、意識違いの不具合は確実に出る。

それによって起こる単純な結末は残業・休出地獄だ。そして、不具合対応はスケジュールの延長・見直しではなく、残業でカバーされる。その責任を負わない人が、その負担を負うことになる。そして、プロジェクトはこれに対応するために赤字になる。

Ⅱ．Ⅲ．解決よりも認識を優先する

私は問題を解決したがる。けれど、ワインバーグ氏が著「コンサルタントの秘密」で教えてくれたように、問題を解決して満足していると、次の問題に足元をすくわれる。本当に問題なのは問題に気づけないことなのだ。解決したいのであれば、この問題を解決したい。

これはリスクに似ている。プロジェクトには様々な「想定外」のことがあり、計画通りにいかないことはよくある。この「想定外」を「リスク」（危機＝危険 (danger/hazard)＋機会 (opportunity)）という。リスクはすべて解決するものではない。影響度が低いため対応しないとか、影響度が高くても発生頻度が低いため対応せず受容するという選択もある。あるいは活用することで利益を得ることもできる。何も考えずに解決してしまうと、機会を失う。また、「解決した」と認識してしまうと、別の「想定外」が牙をむく。かといって、すべてを解決すること、完全に解決することはできない。それこそ、認識していない問題に気づくことは難しいし、仮に気付いたとしても、もうほかに問題がないことは証明できないのだ。

問題は解決するのではなく、「この件に関してはこのように対処することにした」と認識する程度がいいだろう。つまり最初から、解決できないのだから、この問題とうまく付き合っていこう、という制御可能 (Controllable) にしようと認識するのがよい。

Ⅱ．Ⅳ．完全よりも妥当に処置する

「完璧なものなんて作れない」または「100% のものを作るのは困難だし、労力もかかる」と言われてきた。確かに、昔、信頼性を求めるときも、「95% 以上または 99% 以上を 100% とみなす」という判別があることを工学で習った覚えがある。

この私の考えを打ち壊したのが、佐藤正美氏の TM（T 字形 ER）だ。TM は事象をあるがままに書き表すモデリング手法であり、無矛盾で完全なものを表現する。よくみる ER 図に類似するのだが、「関係」をただ書きあらわすだけでなく、意味も書き表すことで、意味論として完全なものを表現する。当然、ヒアリングや確認のため、それなりに時間がかかってしまうのだが、言葉さえ理解できていれば、システムエンジニア以外が見ても理解できるし、あるがままに書き表しているため、言葉は顧客の文化圏で使われるものとなる。また、「見える化」の意味合いもある。すなわち、現在の顧客のビジネスの状況や所有する資源を明瞭にし（知識を知り）、革新や社会情勢への適合させるためにはどのよう

にすればよいか考えられる状態（知恵を働かせる）にできるところに、TM の本当の価値がある。

しかし、ワインバーグ氏が著「コンサルタントの秘密」によって、またも私の考えを見直さなければならなくなった。私はシステムエンジニアらしく、「問題を解決したい病」に犯されていたようだ。何をもって、完璧というのか。TM の否定ではないが、ただ、TM の価値を最大限に味わうためには、あまりにも高い能力や経験が必要に思えた。顧客の言葉や文化をあるがままに書けば、顧客の現状が表現できる。この時点で「完璧に表現できた」と言ってはいけない。なぜならば、あらわしたのは業務プロセスや制度でしかない。人に依存していた作業やちょっとした工夫などは盛り込まれていない。そして、これを考慮したうえでモデルを見直すことができなければ、現在の仕事のやり方を完全に表せたとは言えない。しかし、書き表したことは間違っていないのだが、状況を表現できたという意味で「妥当な状態」を表現したといえよう。その上で、「原則はこうだが、この場合はこういう手段をとっている」ということを考慮することができる。これがモデル上に表現できないのは、あまりにも文脈に対する適合性が多岐にわたるためだ。これをすべて考慮すると、すべてのモデルを表現しなければならず、何を書いているかわからないものになる。情報量が多すぎて、情報として活用が困難なものになってしまう。把握すべき言葉も多くなってしまう。程よい数の語彙、文脈で意味が理解できる程度の曖昧性があったほうがよい。文脈（顧客文化、業務状況）に適用することで適合できる語彙（解決法）を利用することを妥当性とする。逆に、完全性は適用できる文脈が確定的であり、それ以外の文脈に適用することの効果を保障しない。そのため、企業という大きな世界において、完全性を求めることは、どんな状況にも対応できる、想定外のことも含めた無限の種類の解決方法を持つということである。

無限への対応はできない。何ができていないかがわからないからだ。私がわかるのは、今、わかっていることだけであり、わからないことはできない。あるいは、できないかも知れないが、それも含めて対応するという覚悟を持つことはできるが、いずれにせよ、今、完全でないことの証明にしかならない。私はただ、その文脈において、妥当に対処することしかできない。

妥当な対応とは、恣意的なその場限りの対応ではない。文脈に適合するためには、論理的に結びついた対応でなければならない。論理的でなければ、いずれ意味論的に矛盾が発生したり、論理の欠如による追加作業が発生したりして、私を苦しめてくれる。その場限りの対応だけでなく、今まで、そしてこれからに適合できる妥当な解決を行いたい。

Ⅱ．Ⅴ．管理よりも制御をすること

プロジェクト管理するということは、納期を守るようにプロジェクトを運営していくこと。納品するものをいつ納品するか顧客と契約を結ぶ。チームメンバーの日々の進捗を把握し、進捗通りにいかないことに対して対処する。

進捗の管理の仕方はこうだ。「この機能の規模と複雑度はこれくらいだから、５日で計画する。一日 20% 進むはずだ。今日は何 % まで進んだ？」。ここで、「今日は２日目で 40% まで進んでいます。スケジュール通りです。」と回答すれば、マネージャーは満足する。あるいは、「後でいろいろあるかもしれないから、スケジュール上では５日にしているけど、４日でやってくれ。だから、今日は最低でも 50% は進めてほしい。残業してくれ。」と言われるかもしれない。顧客待ちの問い合わせが多くて、仕様が後で変更して手戻りになる恐れがあるプロジェクトだとよくあることだ。あるいは、プロジェクトを達成するために必要な技術力に対して、メンバーの技術力が足りなくて、バグが多く発生する可能性があるとき、などだ。そうであれば、その理由をリスク管理し、イテレーション１（作成）は５日、イテレーション２（手戻り対応）は２日と計画すればよい。イテレーション１の日数を少なくしてはいけない。イテレーション１でやろうと思っていた作業（たとえば、リファクタリング、パフォーマンス対応など）をイテレーション２にまわすのである時に限り、必要分少なくするには構わないが、作業量が変わらないのに日数を少なくしてはいけない。なんの対策も考慮していないのに理由なく日数を減らしても進捗どおりにいくということは、マネージャーの見積もり能力に問題があったことを意味する。あるいは、毎日数時間の残業で日数を狭める、という対処も論外だ。残業しなければ完遂できない見積もりしかできないのだから。しかも、最初から残業時間を含めてスケジュールを考えるものもいるが、そういうやつは大抵、残業時間のためにどれだけのコストがかかるか考慮していない。そういうやつがマネージャーをやると、大抵のプロジェクトは残業代のため、利益をほとんど失う。

そもそも最初からきちんとしたスケジュールを引ければいいのだが、最初からすべてを把握することはできないし、やはり想定外の問題はいつでも発生する。そのため、常に進捗状態を把握し、場合によっては担当替えや縮小などを行い、リスケジュールする必要がある。残業するにしても、「今日残業しなければならない理由。明日対処できないかもしれないと思う根拠」を明確にしなければいけない。そうしなければ、残業代を稼ぐメンバーが登場するかもしれない。このように日々の加速度（作業スピード）と進捗をもとに実行可能なスケジュールを計画していくことを制御といい、スケジュールを制御できてようやく、管理できていると言える。

制御するためには、状況がわかっており、いつでも手を加えることができなければならない。たとえば、「毎日 90%」や「君が手を加えているから、他の人に分担することができない」という状況は、とても制御できているとは思えない。それは管理できているのではなく、状況を知っているだけでしかない。

Ⅱ．Ⅵ．責任よりも覚悟を持つこと

私は責任を取ることはできないし、取るつもりもない。自分の預かり知らぬところで、ある日急に「責任を取れ」と言われても困る。

私は誰かに何かをしてあげられるとは言えない。もしそれでうまくいかなったら逆に相手に迷惑をかけることになるだろう。少なくとも私につき合わせた時間を無駄にさせてしまい、責任を感じてしまう。

責任を取りたくなり私は、だからこそ、「何かをしてあげよう」などという考えを排除する。しかし、私は欲張りなのだ。責任を取りたくないし、誰かに何かしてあげたいとも思う。それが、私が求める「楽しいと思うこと」につながるからだ。

「責任を取りたくない」というと誤解されてしまうかもしれない。しかし、厳密に言えば、「責任を取らせてくれない」のだ。誰が責任を取ろうとチーム全体に影響を及ぼすことには変わりはない。それはリスクと捉え、そのリスクを回避する方法をとる。すなわち、「リスクがあることに気づかない」方法を取る。

何かをやるとき、「このリスクが発生する恐れがある」と考えるのではなく、それも含めてこなすことを考える。それをやった経験がないとか、やるための技術を持っていないとか、問題に気づけないとか、自身がないとか、そういう問題ではない。自分が何をするか、何をしてみせるか、という覚悟を持っているか、という問題だ。わからないことはわかるようになれば良いし、一人で対処できないことは誰かに救援を求めれば良い。自分がやると決めたことを達成するためにやるべきことをただやるだけである。

だから、私はこういう言い方をする。「あなたのそのやり方ではこのようなリスクがあります。あなたは自分の言動によって起こるすべての事象に対して責任が取れますか。私は自分の言動によって起こるすべての事象に対して対処する覚悟を持っています。」と。

Ⅱ．Ⅶ．チームを築くか、パーティーを演じるか

私は組織の中の一平卒に過ぎない。それはまるで、兵士の様である。組織という軍隊に所属し、開発プロジェクトという戦地に赴き、目標を達成するために一丸となって奮闘する。理不尽な、あるいは（チームの能力上）達成困難な契約により、デスマーチという苦境に立たされる。それでも、「お客さん（祖国）のために」を魔法の言葉として胸に抱き、奮闘する。やらなければやられるので、ただやるしかない。すべてに真正面に向き合うと精神が病んでしまうため、「そういうものだ」と納得し、心を無にして目の前の問題に対処する。そのような兵士たちは命の危険が危うくなったとき、自分たちの命よりも苦楽をともにした仲間の命を気にやる傾向があるらしい（ただし、あまりにも精神が病んでいない状態にあるものだけだが）。

システム開発のチームはこれに似ている。強力なチームを築くための最良の方法は何度も何度も同じメンバーを同じプロジェクトに投入して、苦境を乗り越えさせることだ。レクリエーションでチームビルディングなどできない。楽しみを与えてはならない。彼ら自身に、自分の精神を守るために楽しみを見出させるしかない。生かさず殺さず程度のストレスを与えるべきだ。もちろん、人道的な主張により、そんな意見は却下されるだろう。であれば、やらなければいいだけだが、この事実は変わらない。

しかし、残念ながら、この効果が確認されるのは稀である。大抵、プロジェクトのチームはそのとき余っている人材を適当に集められ、結成される。組織としては、あるいは工学的には、「メンバーそれぞれが一定の能力を持っており、メンバーは代替可能であるべき」としているためである。だからこそ、人月という考えは成り立つ。メンバー間にスキルの差はなく、メンバーは協調性があり、誰とペアを組んだとしても一定の成果を発揮できると考えるため、後は作業量＝人数 × 月数の計算が成り立つのだ。

これは仕方のないことだ。だから、そのプロジェクトで新しいチームが出来上がる。すなわち、強力なチームを築きあげたいのであれば、そのようにするべきである。そうしないのであれば、プロジェクト中に築き上げられていくだけなので、築きやすいような制御をするだけだ。

あるいは、いっそのこと、チームビルディングをしようとは考えない方がいいかもしれない。様々な文化を過ごしてきた人が集まって急に、チームという強固な体制が築けるとは思えない。また、強固な体制が築かれたときには、そこの文化に染まってしまい、それまでの持ち味が薄れてしまうかもしれない。もう少しゆるい、同じ目的と同じ価値を共有するだけの関係をパーティーと呼ぶ。

パーティーでは道を外れないように、目的を共有できるように、司会者が進行する。参加者は自分の持ち味を生かし、目的を達成するために自律的に自分のロールを演じる。

Ⅱ．Ⅷ．欲求に妥当か、契約に完璧か

「顧客のために」は魔法の言葉だ。これを言えば、すべてがまかり通る。どんなに困難なことであっても、がんばらなければならないと思い込める。しかし、それは本当に顧客のためになっているのだろうか。

例えば、営業が困難な契約をとってくる。「顧客のために」安価で高品質なものを短納期で納める契約をとってくる。安価にするためには人件費を抑え、作るものを画一化（フレームワークの適合など）をしなければならない。しかし、短納期のために人を大量に投入しなければならない。また、高品質、すなわち顧客が満足するもの（非機能要件も含む）を作るためには、顧客の要求に耐えられるよう、フレームワークをうまく活用する技術を所持していなければならない。言い出せばきりがないが、最終的には、とにかく契約を守ればよいと考える。すなわち、契約で決めた納品物を納品日に納品することを優先する。大量のドキュメントを作成する契約だったが、「作業効率化」の名目で中途半端な内容の大量のドキュメントを納める。品質を保証したコードを作成するため、ドキュメントに書いたことを盾に、「要求通り作った」ことを主張する。「品質向上・品質の作り込み」としておかしなテストケースの設計方法が考えられ、結果として品質悪化となる。

本当に顧客のためを思うのであれば、確かなものを確実に納品できるよう契約すべきだろう。できないことを出来ると言って契約をとり、「やっぱり出来ませんでした」と言ってしまっては顧客を満足させることなどできない。直接この言葉を言うことは少ないだろうが、顧客からより具体的なヒアリングをすることで「想定外でした」とか「それをして納期を守るためには機能削減が必要です」という言い訳は多い。

この問題に対して、アジャイルの考え方が有効だ。アジャイルはフィードバックを早くする。顧客が満足しているかどうか早く確認できるように、確認できる形で提供する。また、確認してもらった結果をすぐに受け、フィードフォワードする。納品物を短期間納品するのではなく、想いの伝達を早くする。システムエンジニアはそのために、確かなものを確実に納品できるよう、契約を結ぶのだ。

Ⅲ．エンターテイナーへの道

システム開発を楽しむ SE をシステムエンジニアとは違うものとして、システム・エンターテイナーと称する。また、従来のシステム開発とは異なるものとして、プロジェクトではなく、パーティーと称する。システム・エンターテイナーはパーティーを開催し、プロセスづくりによりパーティーを実施する。

本章では、エンジニアからエンターテイナーに移行して、どのように振る舞っていくかを述べる。

Ⅲ．Ⅰ．苦行契約を脱却する

修行僧は苦行を行うことで自らを高めようとするが、システムエンジニアは苦行を行うとどうなるのだろう。苦行とは、知を働かせることなく、事務作業のようなシステム開発を行い、制御できずに不要なはずだった残業に身をゆだねることを言う。もっと直観的にいうと、「楽しくない」作業を強いられる、大人の対応で（周りに歩調を合わせて）がんばる（残業する）ことを強いられる作業だ。これによって賃金が払われる契約を苦行契約と呼んでみる。

苦行契約は人の心理をうまく突いてくる。他のメンバーに迷惑をかけないようにがんばらなければいけないような気になってくる。根を上げると、自分よりも苦しい立場の人を紹介され、「お前はまだマシな方なのだ」と、えた・ひみん制度がある時代の農民の様な扱いを受ける。しかし、苦行を受けた分の給与は支払われるため、「自分は奴隷ではない。これはきちんとした契約関係なのだ」と思わせ、騒動に発展することを抑制している。

苦行契約の悪しき点は、システムエンジニアが技術革新についていかないことである。顧客の要求が多様化し、短期間リリースを望んだとしても、システムエンジニアの人数を増やす必要はない。技術革新によって生まれた便利なツールやフレームワーク、自動化などを適合すれば、人数を増やすことなく、多様化・短期リリースに応えることができる。「将来、システムエンジニアの数が足りなくなるだろう」が正しいとするならば、それはシステムエンジニアが変わらず技術力で作業し、多様化・短期リリースのために作成するドキュメントが爆発的に増えてしまっているせいだろう。

Ⅲ．Ⅱ．システムエンジニアであること

システムエンジニアのスペシャリストとは、何にでも対応できる能力（あるいは、誰に尋ねればよいか知っていること）を持つ者のことであり、ジェネラリストであることを述べた。「何でも」とは、例えば、プログラミングができること、設計ができること、要求を引き出せること、契約について折衝ができること、チームとコミュニケーションできること、リスクコントロールが出来ること、プロジェクトの見積もり・コントロールができること、事業戦略が立てられること、仕事をとってこられること、など挙げればきりがない。全てを完璧にこなせる必要はないが、せめて、環境整備・設計・実装・テスト・本番環境へのデプロイくらいは一人ですべてできるようになりたい。

システム開発は複数人でやることが多い。プロジェクト内でロールを割り当てられて仕事を進めている。それは良いのだが、自分が担当していないロールのことが全く分からないのはどうだろう。例えば、「自分は実装担当なので、DB 環境が整えられません」、「テスト手法を知りません」、「計画を立てられません」など。その結果、何か大切なことを決定する人はみんなの意見を聞けることなく、一人で決定しなければならなくなる。もっと色んな事を議論し合いたいのに。

ジェネラリストと名乗る覚悟をまだ持てない私は、自分のことをシステムエンジニアと呼ぶことはやめ、プログラマーと呼ぶことにした。しかし、おそらく、あらゆることに対応できるようになったとしても、システムエンジニアとは呼ばないだろう。

Ⅲ．Ⅲ．システム・エンターテイナーであること

楽しくあるためには、自分だけでなく、自分に関わる場を楽しくしなければならない。そうしなければ、すぐに楽しくないことが自分に雪崩れ込んでくる。しかし、私はみんなを楽しくすることはできない。何を楽しいと感じるか、を決定するのは本人しかできない。私に出来ることは、私自身が楽しくあること、その場を楽しむ様を演じるだけである。もしも、場にいるみんなに楽しめることを共感してもらうことができれば、私は非常に楽しく過ごせるだろう。このように演じるものをエンターテイナーと呼ぶ。

エンターテイナーは時に道化師のように問いかけ、時に伝道師のように道を説く。システム開発は思ったより難しいものではない。全てが当てはまるわけではないが、難しいと思うのは本人が難しいと思い込んでいる可能性がある。そんなときは、思い込みを含んだ考えを捨て、再度見つめ直してみると、実は単純なことだったことに気付ける。これは契

約に対して悪影響を与えることがある。システム開発は難しい、大変なものだ、費用対効果が低い、回収に時間がかかる、など顧客に思い込ませると、その分、発注しづらくなる。発注者が決済権限を持つ者に説明して承認を得るのも困難だ。新規開発よりも保守開発の方が決済承認を取りやすいから、なんとなく保守契約は続く。COBOL の大規模システムが未だにたくさん残っている理由がこれなのか、本当に COBOL が優れているのかは知らない。

この状況をなんとかしなければ、システム開発の仕事は少なくなってしまう。少なくなるから、とれる契約はどんどんとり、お金をとれるところからはガンガン絞りとる。その結果、さらに仕事は少なくなる。自分で自分の首を絞めているのだ。システム開発はもっと単純なものであることをシステムエンジニアに、そして顧客に気づいてもらわなければならない。その様なパーティーを演出することが、エンターテイナーの使命の一つである。

パーティーは「システム開発」というイベントと、イベントの参加者の両方の意味を持つが、達成する目的は同一である。私たちは目的において共同体なのだ。

ここまでの記述で、エンターテイナーはパーティーの司会者だと思うかもしれない。その通りの場合もあれば、そうでない場合もある。あるいは、顧客からヒアリングした物語に対する辞書を実装する場合もあれば、他のメンバーが考えることに集中できるように面倒くさい（実装都合上の）部分の作りこみや、ツールの作成をする場合もある。システムエンジニアと同じだと言われれば、その通りだ。ただ、「意識を変えたもの」の名称として、「システムエンジニアではない」ことを主張するために、システム・エンターテイナーと命名した。

Ⅲ. Ⅳ. パーティーのプロセスづくり

パーティーはプロジェクトとプロジェクトメンバーの両方を意味する。これは、プロジェクトを集合体として捉えているためである。そこには個人心理だけでなく、集団心理も存在し、目的に向かって各個人が働きかける心理はこの二つの心理から生み出される。そのため、メンバーの思考を縛りつけるルールはなるべくゆるく、しかし秩序だったものである方がよい。ルールとは、みんなが最良だと考える方法がぶれて悪影響が起こることを防ぐために、価値観を共有し、何を主軸とするかを認識するための最低限の決めごとである。

パーティーを円滑に開催するためには、戒律を定めておくと良い。戒とは前述したルールのことで、パーティーの趣旨のようなものであり、集団を戒めるものである。律とはパー

ティーのマナーのようなものであり、己を律するものである。戒律とは集団心理と個人心理を考慮している。

このような心理を促すプロセスとして、オルフェウス・プロセスというものがある。これは、オルフェウス交響楽団のスタイルであり、指揮者を持たない。しかし、それぞれがプロとして最高の演出をすることによって､非常に素晴らしい結果を生み出す。プロとは、その社会（状況）に求められるものに応え、行動し、結果を演出する。

パーティーには決まったやり方はないが、目的別に参考となるものはいくつかある。システム開発にもX駆動といったように、目的に応じたプロセスづくりの考え方が提案されている。たとえば、要件があふれるようであれば、チケット駆動を参考にすればよい。業務を見直したいのであればユースケース駆動を参考にすればよいし、現行の組み換えをしたいのであればユーザー駆動を参考にすればよい。

目的、すなわち何を主軸に置くかを決定し、状況に適合すべく計画づくりをすることをプロセスづくりと称する。プロセスづくりは全員が同じ目的を共有し、それに向かって働きかける意思の一体化が必要となる。そのため、メンバーとプロジェクト（他の全てのメンバー）が一体化するよう戒律を定め、個人と集団を同じものとしてとらえるべく、パーティーという一語で称した。

Ⅲ．Ⅴ．パーティーの参加料

システム開発は非常に高価だ｡それはシステム屋の知恵にお金を払っているためである。すでにある知を認識させるだけで十分のはずなのに､システム屋に頭を使わせようとする。具体的に言うと､「設計書は嘘です、またはありません。ソースが真です。」と言って、レガシーコードから仕様書を起こさせることだ。レガシーコードとは社会情勢に適合できない、または適合困難なコードである。具体的には、実行可能なテストコードが存在しないコード、またはリファクタリングできないコードのことである。このようなコードは変更後に正しいことを確認することが困難であるし、名前や文脈もデタラメであるため、読むこともままならず、リファクタリングしたときに機能が変わっていないことを確認できない。そして、仕様書を起こして顧客に確認しても､「業務が動けば正しいのでは？」程度の認識しかもらえない。コードが仕様書のように読めるようなものだったり、理解するためのモデルが存在していたりすれば、まだ良いのだが、レガシーコードにそんなものが存在することは稀だ。さらに、そのようなプロジェクトの場合､歴史を繰り返すことが多い。すなわち､「見える化」､「最適化」と称してガチガチに書かれた社会情勢に適合困難なコー

ドである、別のレガシーコードを生みだす。

　システム屋の知恵を使かっても、知識としないため、次のプロジェクトでは別の知恵を使わなければならない。だからシステム開発はお金がかかるし、「またお金がたくさん必要になる」と思い込ませてしまい、その結果、新規開発の発注を渋る。システム屋は保守開発で食い繋げようとする。それもお金は随分かかるし、何もしなくてもお金がかかる。特に受託開発の場合は保守契約を結んでいなければ、どんな不具合であろうと「仕様通り」で乗り切られるため、続けざるを得ないこともある。

　そのプロジェクトが生んだ知識を次のプロジェクトに利用することができれば、どれだけ安価にシステム開発ができるであろうか。それはコードの再利用性というレベルの話ではなく、業務プロセス・業務戦略まで対象とする。システム屋の知恵を使うことなく、顧客側が自分たちの知識を以て社会情勢への適合を考えたとき、「どこが、どのように、どれだけ変わるのか」が理解できれば、必要なシステム屋の知恵は抑えられるようになるし、考えるためにシステム屋に声をかける機会も増えるだろう。この状態こそ、「見える化」であり、ただ現状を表現しただけでは「見える化」と言うべきではない。どれだけ知恵を触発させる知識を生み出したかが「価値」であり、そのお手伝いをすることがシステム開発の費用（パーティーの講演料）となる。

Ⅳ. システム開発を楽しむ

　私はシステム開発を楽しみたい。「仕事は楽しむものではない」とは思わない。楽しくないより楽しい方が断然良い。それで成果が落ちるのであれば問題あるかもしれないが、成果が上がるのであれば、もし成果が上がるのであれば、ぜひ、楽しくするためにはどうするか考えるべきだ。

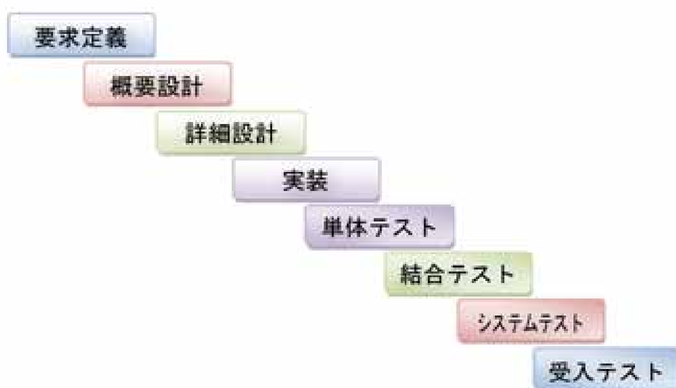
　もしかして、「仕事は楽しむものではない」と思っている人は、「楽しい＝楽」だと思っているのではないだろうか。「楽しい」と「楽」は完全一致しない。それどころか、むしろ、「仕事は楽しむものではない」という人が指揮する仕事こそが「楽」だ。契約に縛られた仕事は得てして事務作業的システム開発になりがちだ。知を働かせる必要がなく、兎に角書類に決まったことを書き綴るだけの作業になる。特に論理的に考える必要がなく、本来必要とされた能力を使わずに出来るのであれば、「楽」な仕事だと言えるだろう。ただし、体力は要求されるが。逆に、「楽しい」仕事はシステムエンジニアとして必要とされる以

上の能力が必要になる。「楽」ではない仕事になりがちだが、それが自ら望んだ道であれば納得もできる。納得していなければ命を懸けることも出来ない。

本章では、エンターテイナーとして「楽しくありたい」という欲求を満たすために使用する思考ツールを紹介する。

Ⅳ. Ⅰ. アジリティ・ウォーターフォール・モデル

規模が大きく、複数人で行うプロジェクトはウォーターフォール・モデルによるシステム開発になることが多い。ウォーターフォール・モデルの特徴は、作業を各工程で区切り、作業が完了しなければ次の工程に進まない(並行作業をしない)。多少の並行作業はあるが、少なくとも次工程に悪影響（手戻りの恐れがある）を与えない程度に抑えることで、安定した品質や納期を確保することができる。



しかし、実際にはこの通りに出来るのは困難になってきている。各工程を並行作業しており、手戻りは頻繁に起こる。概要設計・詳細設計・実装・単体テスト・結合テストを並行作業したことがあるが、仕様変更や設計不良のバグ票をたくさん書いた。

アジャイルのような反復モデルでよく言われるのは、「状況の変化により顧客の要求は頻繁に変わる」ということだが、私が経験した状況においては、次のことが原因となっている。

* （目に見えるものが）何もない状態ですべてを決定しきれない
* 顧客レビューが滞っているが、納期がきまっている

そのため、工程の終了条件は時間となり、各工程の責任は次の工程へたらいまわされてしまう。

最初に何かやりたいことがあって、システム開発を依頼するのだが、その時点ではそれが自分の欲望を満たすものなのか、あるいはもっと良いやり方があるのか、といったことはまったくわからない状態である。そんな状態のときに、何か月・何年かかけて作るシステムを、しかもそれを検証する術なく想いだけで、完全に考えきるのは、あまりにも負担をかけすぎている。そう思うと、何でもかんでも「仕様にありません」といって切り捨てるのに躊躇するのか、「顧客のために」出来る限りの要件は取りこんでいこうとしていく(あまりにもひどいものは再見積もりか保守対応になるが)。そうすると、結局、各工程を並行作業することになる。

また、顧客が通常業務のため時間があわないとか、代理レビューアーを用意できない、あるいはそもそも、レビュー前の成果物を作るために必要な問い合わせの回答がいつまでたっても貰えないが、納期は決まっている場合、並行作業を強いられる。

手戻りの発生する並行作業をすることはウォーターフォール・モデルのもともとの考えと異なるものになってしまっている。特に問題なのは、手戻りがあることではなく、手戻り対応の工数を見積もっていないことにある。たとえば、「手戻り対応を含めて５日」と見積もられたとき、手戻りを想定して３日で終わらせたとすると、残りの２日は前倒しのリスケジューリングにより消滅する。そして、しばらくして手戻りが発生したとき、余

分工数は存在しないので、作業を詰めてみたり、がんばってみたりすること（残業）になる。最初から、手戻り対応という作業があることを認識し、これを作業工程として組み込んでおけばよかったのだ。

このモデルはアジャイルに、顧客が、自分が考えたことをこのまま進めて良いか、確認しながらシステム開発を進めていけるよう、フィードバック（結果確認）・フィードフォワード（見直し）を早くすることを目的としている。

まずは要求定義をしっかり行い、それに対して受入テストを行う。まだ何も作っていないので、テストを実行することはできないだろう。ここでは、テストを実行できる状態にすることを目的とする。たとえば、チェックリストを作る、など。自動テスト環境ができれば尚良い。少なくとも、「レビューしたよ」と口頭で終わらせるのではなく、「次の工程に進んだときに前工程の手戻りが発生しないことを確認した根拠」の検証をしておきたい。これが受け入れテスト Phase1 である。

次に、概要設計を行うのだが、その前に受入テスト Phase1 の結果をもとに要求定義を見直す。見直しが終わったら、概要設計を行い、それに対するシステムテストを行う。そして、今回考えた概要設計が要求定義に則しているか受入テスト Phase2 を行う。

続いて、受入テスト Phase2 の結果をもとに要求定義・概要設計を見直す。見直しが終わったら、詳細設計・実装・単体テスト・結合テストを行う。ここで動くものができるので、実際に動かして確かめる。そして、システムテストを実行し、受入テストを実行する。アジャイル同様、いくつかのモジュールを作らなければ実行できない場合は、モジュールがそろうまでシステムテスト・受入テストを実行する必要はないが、これまでと同様の観点でテストはしなければならない。

あとはこれを何回か繰り返し実行する。回数は規模や顧客の確認間隔にもよる。また、

上図では要求定義・概要設計は１回で、あとは見直しだけになっているが、詳細設計・実装のように何回も繰り返し行って良い。これも顧客の確認間隔によって見直せばよいだろう。

Ⅳ．Ⅱ．ホウレンソウ・モデル

システムエンジニアとしてやるべきことは、顧客に対して確かなものを確実に提供することである。確かなものを作るためには、顧客に確認すればよい。確実に提供するためには、状況を把握し、コントロールすればよい。

これはシステム開発に限った話ではない。仕事をきちんとこなしていくためにやるべきことは同じだ。仕事をきちんとこなしていくための合言葉に「ホウレンソウ」があるが、これをシステム開発に適合してみる。

相談：欲求を伝える、フィードフォワードする

報告：要求に応えた結果を返す、フィードバックする

連絡：相談・報告の内容をいつでも誰でも参照可能にする

欲求を伝えるのは顧客からシステムエンジニアでも、システムエンジニアから顧客でも、どちらでも構わない。前者なら要件の伝達になるし、後者なら仕様の確認になる。顧客もシステムエンジニアも同じチームのメンバーなので、報告・相談し合えば良い。

そして、すべての情報はプロジェクトの成果物になるので、いつでもだれでも、すべての情報を参照できるよう開示すべきであり、そのための連絡網を用意すべきだ。少なくとも、必要な情報が「誰かの頭の中」にしかない状況はなくしたい。

このモデルは開発プロセスというよりはコミュニケーション管理の考え方だ。しかし、

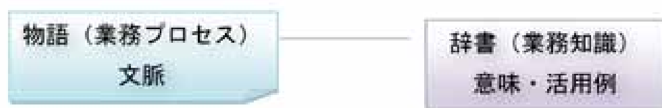
システム開発が人と人の対話によって行われる以上、コミュニケーション体制こそが開発プロセスとなるのではないだろうか。そして、今まで開発プロセスと呼んでいたものは、ただの契約形態である。

Ⅳ. Ⅲ. 物語と辞書

システム開発の一例として、物語と辞書の作成を挙げる。

まず、顧客から業務プロセスをヒアリングし、文脈をまとめる。これを物語とする。そして、物語に出てくる言葉を業務知識として確認し、意味と活用例をまとめる。これを辞書とする。辞書は意味と活用例から成る。意味とはその単語自身の抽象的な意味であり、活用例とはどんな文脈に適合できるかという関係情報と、文脈における具体的な意味から成る。

新しい単語を定義したとき、あるいは新しい様相を追加したとき、物語を読みなおし、誤解釈しないか確認することがテストとなる。



言葉と論理は人間が認識できる抽象的な方法である。どんな方法論やツール・フレームワークを使うよりも、言語と論理の方がより少ない作業でこなすことができる。より感覚的に行うことができるのだ。

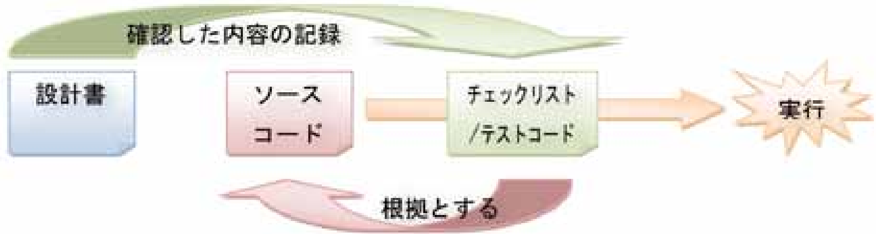
Ⅳ. Ⅳ. 確認と根拠

テストが品質を保証するもの、すなわち要求通りに作ったことを保証するものであるならば、作る前に要求通りとはどういうものかを確認する工程は何故ないのだろうか。「今、自分は要求通りに実装を進めていますよ」という根拠は何だろうか。

作業をする前にその前工程までの資料を参照したり、詳しい人に尋ねたりするが、分かっただけで終わる。分かったことを記録しないから、結局、どんなにドキュメントがあったとしても、「人の頭の中」にある、担当者しかわからないという状況は変わらない。例えば、実装・単体テストの担当者が同じだった場合、実装前に仕様を確認し、単体テスト前

にチェックリストを作成するため、再度、仕様を確認しなければならなくなるのは、「確認したこと」が過去の自分の頭の中にしかないからだ。

実装前に確認したことは記録しておきたい。できればテキストファイルへのメモではなく、チェックリストやテストコードなど、実行可能な知識として記録したい。



これはテストファーストの考え方と同じようだが、確かなものをつくるためにしたいと思うことを述べているだけだ。なぜテストファーストをしたいかという理由にしてもよい。これにより、次の嬉しいことが起こる。

* 先にチェックリストやテストコードを書いておくと、その実行対象として作る関数やメソッドの単位が意識しやすく、命名も明瞭になる
* 先にテストコードを用意しておけば、要求通り動いていることを確認しながら実装できる

これは、「実践アジャイルテスティング」が主張する次のことを満たす。

* 品質を確保する最上の方法は、テストしやすいコードを書くことである
* レガシーシステムとは、自動レグレッションテスト環境が（ほとんど）整っていないシステムのことをいう
* テストと品質がアジャイル開発の中心である

Ⅳ．Ⅴ．マニフェスト

アジャイル・マニフェストに倣い、マニフェストを考える。

＊　個人との相互作用を推進するプロセスづくりを行う

チームビルディング（パーティーの進行）がプロセスとなる。状況に応じてチーム体制が変わるように、プロセスも状況に適応し続ける（場の流れに応じて、パーティーの進行を適応させる）。

＊　動作するソフトウェアが、ドキュメントの包括的に用意する

動作するソフトウェアがドキュメントとなるが、契約によっては、ドキュメントのフォーマットに適合する必要がある。ドキュメントからソフトウェアを作ることは困難だが、ソフトウェアからドキュメントを作ることは容易である。

＊　顧客との協調を契約内容とする

システムエンジニアも顧客も同じチーム（パーティー）のメンバーであるため、双方同程度のリスクを背負う。チームとしてどれだけのことを望むか、認識を共有していくこと（パーティーの目的、あるいは最低現のマナーの取り決めなどを共有すること）。

＊　変化に対応するための工程を計画に組み込む

手戻りが発生する恐れがあるものはリスク管理する。それにより、次の計画づくりにおいて、手戻り対応を計画に組み込むこと。

〔参考図書〕

・西部 邁、『昔、言葉は思想であった　語源からみた現代』、時事通信社、2009.11

・G.M. ワインバーグ、『コンサルタントの秘密―技術アドバイスの人間学』、共立出版、1990.12

・Janet Gregory, Lisa Crispi、『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

・スコット・ローゼンバーグ、『プログラマーのジレンマ　夢と現実の狭間』、日経 BP 社、2009.5

〔プロフィール〕

2003 年 株式会社日本システムディベロップメントに入社後、おもに社内業務システムの受託開発を担当する。

読者からのコメント

時本さんのソフトウェア技術者マインドに熱く感動しました。ただ、技術者が、ぼんやりと手足を動かしているだけでは、ソフトウェアの利用者は喜んでもらえませんよね。また、技術者が独りよがりであっても同じことになると思います。

全体を読んでいて感じたのが、時本さんの不満が論文の源泉になっているのではないかということです。不満（問題）から物事を考えることは良いことですが、それが論文に出てしまうと、敵を作る事にもなりかねません。時本さんの主張ですので、他人の共感を得る必要はありませんが、ただの批判と受け止められてしまうのではないかと心配になりました。

もし、敵を作ってもかまわないということであればもっと過激にした方が面白いかと思います。(何があっても責任は取れませんよ)

もうひとつは、時本さんがソフトウェアエンジニアリングの呪縛にはまってしまっていると感じました。現状の呪縛、神話では、こうあるべきだというような主張に思えます。時本さんのマインドがあるのであればパラダイムシフトして今の状況を見てみたらいかがでしょう。

論説

〔知働化研究会誌創刊第１号〕
知働化研究会
Executable Knowledge and Texture Laboratory

ゆるっと行こう

ゆる思考へのおさそい

本橋 正成（もとはし まさなり）
masanari@motohasi.org

はじめに？合意の難しさ …… 88
I. ゆるへのご招待 …… 89
ゆる組に入りませんか？ …… 89
「ゆる」って何ですか？ …… 89
ゆるの目的って何ですか？ …… 90
ゆるの適用範囲やスコープはなんでしょう？ …… 90
ゆるじゃないものって何ですか？ …… 90
ゆるっていいんですか。 …… 91
ゆるの困難は、何でしょう。 …… 92
ゆるは、儲かりますか？ …… 92
ゆると知働化の関係性は …… 92
ゆるの先へ …… 92
ゆるマップ …… 93
「かたっ」と「ゆるっ」 …… 94
「かたっ」の世界観 …… 94
「ゆるっ」の世界観 …… 95
応用 …… 96

Copyright 2010, MOTOHASHI Masanari, All rights reserved.

II. 知働プロセス …… 97
知の発生と知働化…… 97
アジャイルと、その次へ…… 98
成功と失敗、正しいと誤り…… 98
III. ソフトウェア？阿吽モデル …… 99
究極のモデルを求めて…… 99
ご縁モデル：その１…… 100
ご縁モデル：その２…… 101
ゆるいソフトウェアに向けて…… 102
知働化に向けて…… 103
〔プロフィール〕…… 103
読者からのコメント…… 104

はじめに？合意の難しさ

東京の電車には、構内ではタバコを吸ってはならない、化粧をしてはならない、リュックを後ろに背負ってはならないなどのような様々なマナーについての広告が掲載されています。ある外資系企業でも、カジュアル・ウェアで出社するときのシャツのボタンは上から三つ目まではあけてよい、シャツはズボンに入れるなどの様々なルールが存在します。テレビを見ると、オリンピック選手が飛行機に乗るときに洋服をきちんと着ていないことがニュースになっていました。そのことについて、どちらが正しい、どちらが迷惑など私自身の意見を主張するつもりはありません。どのように合意を取っていくのがいいのでしょうか。

たとえば、声が大きい人が、オリンピック選手が飛行機に乗る際の服装ルールを明確に決めて従わせ、強制する方法も一方ではあります。確かにルールを決めることは大切ですし、ルールに従うことも大切です。また、一度決まったルールに対して何も考えずに従うことは「楽」です。しかしながら、起こりえることや条件をすべて想定した的確なルールを決定・合意できるのでしょうか。補欠の人はどのような取り扱いなのでしょうか。パラレンピックも同様のルールにするのでしょうか。準備に携わるスタッフはどのようになるのでしょうか。税金を使うという場合、誰が税金を使ったかというは明確なのでしょうか。ほんの少し援助をもらった場合の服装はどのようにすればいいのでしょうか。さらに飛行機に乗るときはきちんとした服装をする、という価値観もあるのでしょう。一方で、エコ

ノミック症候群にならないよう楽な服装が好きな人もいるのでしょう。文脈に従えば、どちらの意見も正しいでしょうし、どちらの意見も間違っています。どちらの意見を主張していても平行線をたどり、いつまでも解決しないでしょう。

ここではルールを例に挙げましたが、それ以外の状況や条件でも発生しうると考えています。たとえば、自転車の修理を考えてみましょう。マニュアルにはネジを外すと書いていますが、そのネジを力一杯かけても外れなかったこともあります。力を入れすぎて、ねじを壊してしまうこともあります。目安になるトルクは書いてあっても、マニュアル通りに行かないことがたくさんあります。むしろマニュアル通りにならないことの方が多いのではないでしょうか。あらゆることを想定して、ルールを作るのは若干「かたい」感じがします。

同じような構造から発する問題が世界中を駆け巡り、行き詰まりを感じています。客観的で明確なルールやマニュアルも重要です。しかし、もう一度考え直してゆるっといきませんか。そして、なぜ「ゆる」でうまく行くのか、ご一緒にゆるっと考えていきませんか。

I. ゆるへのご招待

ゆる組に入りませんか？

世の中には、いろいろな人たちがいらっしゃいます。様々な問題や苦しみがたくさんございます。その中で「これが正しい！」「これこそが真実だ！」って主張するのも大切ですが、ここでゆるっと参りませんか？もしかして、その問題も、そもそも適切じゃないのかもしれませんしね。

「ゆる」って何ですか？

ぜひご一緒にゆるっと考えて参りませんか。

感じとしては、日常生活における水や空気が近い気がします。水や空気は、形も色もないからこそ強さや有用さがあります。本当の究極は実現しえないのですが、取り組むことによって限りなく究極に近づけます。

ゆるの目的って何ですか？

ゆるっと楽しく参りましょう、ぐらいでしょうか。あえて書くならば、目的を持たないことによって、大いなる目的を自然に達することが目的です。

目的を明確にすることの大切さは、目的を考えることで考えをまとめたりできる効用もあります。しかしながら、目的を意識した時から、その目的を達成することが難しい逆説的な状況が多くあります。その目的を決めることによって、その目的にとらわれ、あるがままを見つめるときの邪魔をします。成果だけでなく過程も大切にして、良いことを求めていきたいです。

ゆるの適用範囲やスコープはなんでしょう？

「ゆる」は形容詞ですので、何にでも当てはまります。既存の考え方と似ているところがありますが、組織やプロセス、プロジェクト、ビジネスや経営、ソフトウェア、要求定義やテスト、アーキテクチャ、合意形成、哲学や言語学などの領域などでゆるい特性が観測されます、または観測できるといいな、と期待しています。

ゆるは「度合い」や「アプローチ」です。そのため「これはゆるですか？」について、あらゆるものに対して「はい、ただし究極ではありません」が典型的な回答になります。ドメインについても、ゆるは適用されます。あらゆることはつながり影響しあっていて、何らかの行為は影響を与えます。戦争や契約をメタファに基づいた文化においては、ドメインや領域を明確にすることが求められています。それを否定するものではありませんが、あくまで便宜上のものであることを認識することです。その関連性を含めて複雑なものを複雑なものとしてゆるっと取り扱いたいです。

ゆるじゃないものって何ですか？

あらかじめ完全な計画を立てる、完璧な管理を行うことは、ゆるとは別の方向性ではないでしょうか。その結果、創造的な活動がスポイルされる、発生したゆらぎを吸収するためにムリが発生する、ムリが出てコストが増え成果に結びつかないことがあります。そのため、実際の活動をよく観察すると、ゆるの特性は発見されることが多いようです。たとえば、この庭は気持ちいいと気づき、その庭の気持ちいい点を伸ばそうとします。

自分自身の「正しさ」や「真実」にこだわっているときは、ゆる度合いが低い可能性が

あります。「正しい」や「真実」だと認識しているものは、あくまでその人と、その人が属している文化や共通意識のなかでの「正しさらしさ」でしかありません。相手の考えやとらえかたを尊重し、その考えの違いを相対的にとらえることによって、つながりや構造をとらえて行きます。その結果、良いことが悪い結果を生んだり、悪いことが良い結果を生んだりしますし、結果が原因になったり、原因が結果だったりします。

過去の経験や知識からとらえた統一的なものは、ゆる度合いが低い可能性があります。同じパラダイムであっても 99.99999…% の「もっともらしさ」はあるけど、100% の正しいことはありません。たとえば、力学法則も、実際の実験結果は「ぶれ」や「ゆらぎ」があります。その「ぶれ」や「ゆらぎ」を含めて大切にします。さらに、真実と思われてきたものはどんどんパラダイムが変わってきた歴史を謙虚に受け止めます。そのために過去の経験や知識をとらわれないように、過去の経験や知識をいったん横において、謙虚にあるがままを見つめ、それを大切にしてします。とらわれないであるがままに従って、進めていきます。

目標を明確にし、その最短距離を直線で進むことにこだわるときは、ゆる度合いが低くなる可能性があります。むろん、そのように進めればいいのですが、現実はそんなに単純で静的ではありません。われわれが取り扱う創造的で複雑な目的自体が変わり、アウトプットによって状況が変わります。そこで､その都度の生成されるものを大切にしていきます。

数値化や形式知化にこだわりすぎることも気をつける必要があります｡状況に応じては、暗黙的で定性的な認識がリーズナブルであることもあります。今まで以上に大量のデータ処理を扱い、最大限活用します。だからこそ、数値化や形式値化し、分析する際に排除しえない恣意性などを考慮する必要があります。

ゆるっていいんですか。

「究極」や「最強」を目指しています。それが持つゆるさ故に究極はあり得ないのですが､あるしきい値を超えたときに、究極を感じる人が多いようです。

最初は、ゆるめることになりますので、無駄をそのまま許容するように感じます。しかし、振り返っているといいところにハマってくるので、ムダやムリがなかったなという印象を感じる人が多いです。

ゆるの困難は、何でしょう。

やはり、最初の計画や予測を確率事象として扱う必要があります。事前に決定論的な事前の考え方とは大きく違います。確実に約束された未来や計画を前提にされていると、手放す怖さがあるのではないかと思っています。この場合の、この手放すは放任ではありません。

ゆるは、儲かりますか？

使い方なので儲かることを約束することはできません。ただし、今後の商売やビジネス、戦略を考える上で、とても重要な考え方になってきます。既存の方法では、ゆるに太刀打ちできないのではないかとも感じています。

ゆると知働化の関係性は

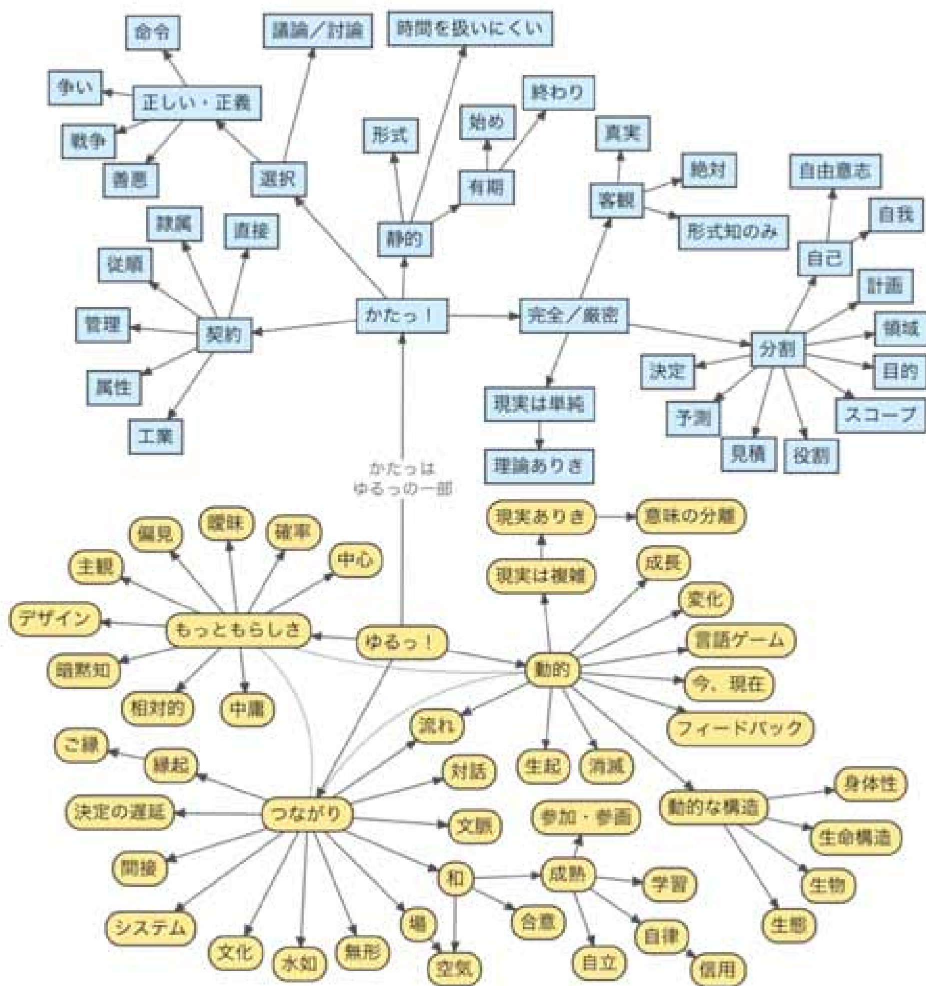
あるといいのですが。

ゆるの先へ

もちろん、何が何でもゆるく行くべし、というのもゆるさが足らないってことになります。ゆるの先があるのですが、それはこれから考えて明らかにしていきます。

ゆるマップ

キーワードをゆるっとまとめてみました。「かたっ」についても、ゆるとは反対でもありますが、含まれる概念でもあります。



「かたっ」と「ゆるっ」

「かたっ！」は「ゆるっ！」の別のパラダイム（考え方）です。「かたっ！」より「きつっ！」と表現した方が良いのかもしれませんね。なぜ、「かたっ」と「ゆるっ」という二つの方向性があって、その「かたっ」に重きがあったものの、「ゆるっ」にも注目し、考え方の重心が移動していく傾向が見て取れます。

20世紀においては、「かたっ」のパラダイムが重宝されてきました。かたっの世界観に基づいた経営やソフトウェア作り、科学技術や工学が発展してきました。かたっの世界観は、非常に有用ですし、決して否定するべきものではありません。それだけでなく、これからも引き続き有用であり、成果を出していくと考えています。しかしながら、このかたっの世界観にとらわれていると非常に危険です。たとえば、経営や運営をこの世界観だけで組み立てるとリスクが増大し、利益を得るなどの本来の目的を達成しにくくなります。無理（ムリ）が強くなって、関係者は疲弊するわりに長期的な目的から乖離するようになります。

かたっの世界観は、ゆるっの世界観の一部、と考えています。ゆるっは「かたっ」すらを飲み込んでしまう上位概念です。かたっの世界観からは、ゆるっの世界観を理解することが困難です。おそらく、ゆるっの世界観についての印象は「こんなんで大丈夫？」という不安なのではないでしょうか。なぜなら、ふわふわしていて、一見とらえどころがなく、いくつかのものを手放しているように見えるからです。しかしながら、ゆるっの世界から見ると、逆に堅苦しく、不適切さや小ささを感じる事でしょう。理解のため、必要なものも捨象してしまうとも感じます。かたっの世界観を捨てているわけではなく、限界や弱点をよく知って、使えるところでは最大限に、しかも、気をつけて利用します。

「かたっ」の世界観

「かたっ」は、完全さや厳密さを追い求めます。絶対で、真実があり、「客観があり得る」と考えているからです。なぜならば、現実は単純な規則や理論で説明できるという信念があります。そのため、あらゆる事象を分割して考えます。たとえば「個人 (individual = これ以上分割できない →個人)」であるべきと世界を捉えています。分割でき、理論があるから、すべてを予測し、決定することをが可能であると考えています。同時に、排他的な役割や領域を想定し、物事をとらえます。

「かたっ」は、静的にとらえる傾向があります。静的であるということは、時間の要素

が少ないため時間や因果関係を扱うことが比較的苦手です。物事をはじめと終わりがあるように仮定し、その中で物事を進めて行く必要があります。この制約は、変化に対応するために、大きなコストを必要とします。大きなコストがあるけれども、変化に対応できるのではないか、とも考えられますが、

たとえば、プロジェクトなどではムリをして合わせていくだけで、ほとんど変化への対応は困難ですな現場が多いようです。

絶対的な正義がある、絶対的な論理がある、という世界観を持っています。しかしながら、絶対的な正義が文化や個人によって異なっているため、どちらが正しいか、どちらが善であるのかをめぐって争いや戦争の火種になることが多々あります。勝った方が負けた方へ従順さをめぐります。

さらに、基本的な思想としての契約に基づいています。これをしたら、こんなリターンを保障する、これをしてはならない、と、直接的で、一度決めた契約にはどのように世界が変わっても従う必要があります。管理という考えかたは、この契約に従っているかを確認することになります。次に述べる「ゆるっ」の世界における管理にかわるものは、根本的なところが違って、それぞれが主体的に判断し、行動や選択を行うのですが、その際に中長期的な他人からの評価に基づきます。その「場」が制約になります。

「ゆるっ」の世界観

現実は複雑であるととらえ、複雑なまま扱っています。たとえ、あらゆる事象や構造についても、生まれては消えていく変化を前提にしています。そのため、ひとつの事象においても文脈やとらえかたによって違ってきます。たとえば、言葉を使う際にも、きちんとした論理体系に基づく以前に、お互いの背景や文脈の交換によって、意味や使い方の共有を行っていきます。「今、ここの文脈」を大切にして、その意味を育てていきます。

ゆるっの世界観では､善悪の判断も文脈や文化に基づいて相対的に動的に判断されます。そのため、確率的であり曖昧です。主観や偏見を大切にしますので、立場によって違ってきます。しかし、その違いが多ければ多いほど、より「もっともらしさ」が増すだろう、と考えています。客観的なデータであっても、主観に基づいて選択され取得された可能性を否定しません。その中で相対的にちょうどいいところを求めていきます。

さらに、絶対的に変化しない個体や単体を想定しません。つながりの中で、その意味や反応が異なってくると考えています。絶対的な正しい理解や認識を疑います。あくまで文

脈や場、文化によって、とらえかたが変わってきます。そのため、間接的に見えます。ある悪い事象も、他ではいい結果をもたらしたり、全体的なつながりを大切にしています。

そのため「場」に重きが移ってきます。たとえば「空気」や「対話」のように文脈に文脈を重ねていきます。そのため、ユーザや関係者の参加や参画が前提になり、その個人個人の成熟が求められます。成熟のためには自立や学習によって、その後ろ盾が重要視されます。その結果、相互の信用に基づいた複雑なネットワークが機能します。

応用

ゆるっとかたっは、一見、反対のように見えますが、相補的に見るのが有用です。片方により過ぎると全体としては弱い形になってしまいます。便利さや有用さを基準にとらえていくことが大切です。

抽象的な概念だけでなく、実際の現場にどのように応用していくのかを簡単なに説明します。ゆるっは、文脈や状況に応じて臨機応変に実装します。たとえば、ソフトウェア実現のためのプロセス作りや、ソフトウェアのアーキテクチャなどの仕組み作りも、この世界観に基づいて作ることができます。たとえば、場で同じ様に話すためには社内勉強会のような仕組みづくりは有用ですし、ふりかえりは、取得したい

ふりかえりは、取得したい価値や目的に向けて変化・成長した状況や状態にあわせて調整するという価値がある。たとえば、ふりかえりなしでは、目的からは外れてしまうリスクが高くなる。では、いかが？

ふりかえりは、ひとつ「メタ・上位」もしくは「間接的」な調整を担っている。インジェクションやてこ入れする場を探して、プロトタイプ的な実行に結びつけて、システム（系）を調整しながら作っていく。あたりかな。

II. 知働プロセス

ひとつのプロセスの中に発生しうるモデルを考えてみました。

- ＊ 見：あるがままに見ましょう。かなり難しいのですが、少なくともそのように努力しましょう。
- ＊ 思：あるがままに見たものを評価しましょう。
- ＊ 語:評価に従って､出力しましょう。たとえばプログラムを書くなどもあります。
- ＊ 行：語ったものを実行するなど行いましょう。この後、行った結果を含めて、また「見」につながっているところがポイントです。
- ＊ 生：上のサイクルをぐるぐる回しましょう。

このようにひとつの系を作ります。このひとつの系を、セル（細胞）と呼ぶ人がいるかもしれません。人や機械、社会の仕組み・システムも同じ比喩を使ってとらえることができます。

ほかの系とつながっていて、関連しています。この系は閉じているのではなく、ほかの系ともつながっています。相互に関連しています。認識した状態に対して、適切だと思う反応を返す。それだけです。

知の発生と知働化

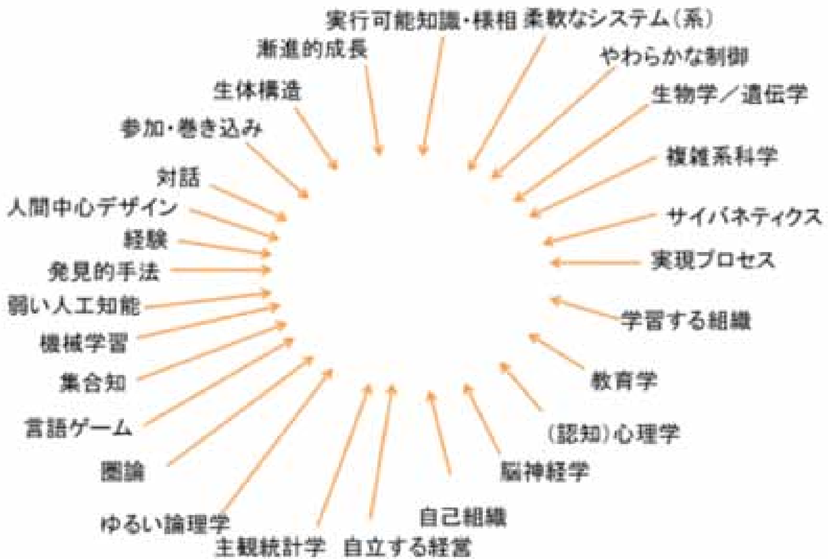
このひとつの系をまわして違いが生まれます。このちょっと違い（の集まり）から「知」が生まれます。これが知である、という明確なものではなく、この差や変化によって「ふわっ」とうきあがってくる感じです。系をまわすことを学習と呼ぶ人がいるかもしれません。また、この組み合わさった系を（まわして）動かすこと、働きます。これが私の理解している知働になります。

アジャイルと、その次へ

ソフトウェア実現プロセスにおいて、このサイクルはアジャイル開発プロセスとよく似ています。アジャイル開発プロセスでは、行ったもののフィードバックを人間が認識して新しいコードを作り出します。しかしながら、アジャイルのその次として、フィードバックを機械やソフトウェアが行ったらどうでしょうか。たとえば、実行結果の大量のログを機械が読み、集合知や機械学習などの考え方を使って判断したらどうなるでしょうか。ちょっとだけ「その次」になるかもしれません。

成功と失敗、正しいと誤り

この考え方からいろいろなことが理解することができます。たとえば、この考え方を使うことによって現在のあるがままを受け取るため成功と失敗にとらわれなくなってしまう。同じように正しいと誤りもとらわれなくなってしまう。あえていえば、あるがままで正しいのでしょう。



III. ソフトウェア？阿吽モデル

この章は、2009 年 6 月 18 日にソフトウェア・シンポジウムにて発表した「さまよえるモデリングゆるいソフトウェアを求めて」の再録で、加筆・修正しました。

究極のソフトウェアは、決定論的なパラダイムではなく、確率論的な考え方へ移行する部分を示唆しています。その結果、現実の世界によりフィットしたソフトウェアに向けての問題提起と、実現できる道筋について示します。

究極のモデルを求めて

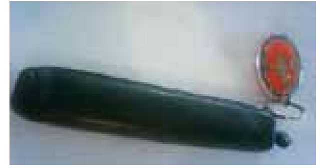
「未来を予測する最善の方法は、それを発明することだ」

みなさんご存知のアラン・ケイの言葉です。モデリングにおいて、どのような未来を作り出していくのかを探求者としての命題としてあげました。たとえ、今日時点ではファンタジーであったとしても、もし想像ができれば実現に向けてのきっかけになります。果たして、究極なモデルはどのようなものでしょうか。ぜひ、みなさまの意見をうかがいたいと思っています。

若干のユーモアも混ぜ、究極のモデルとして「阿吽（あ・うん）」を例示しました。Wikipedia(2009年7月現在)の「阿吽」によると「悉曇文字（梵字）において、阿は口を開いて最初に出す音、吽は口を閉じて出す最後の音であり、そこから、それぞれ宇宙の始まりと終わりを表す言葉」となっています。つまり、たった二文字で宇宙の始まりと終わりが記述されています。ただ、この「阿吽のモデル」は究極ですが、ちょっと規模が大きすぎて実生活やお仕事で使いにくいようです。ふたりがぴったり息を合わせて、やり取りしている「阿吽の呼吸」が、もうひとつ連想されます。今回は、この阿吽の呼吸を目指したソフトウェアやモデルを探求しました。

コンセプト・リーダの山田氏が、顔合わせのときに「ご縁」とおっしゃったのをヒントにして、ご縁モデルについて考察を試みました。このご縁とは、世界の一切は直接にも間接にも何らかのかたちでそれぞれ関わり合って生滅変化しているという考え方です。つまり、人や物に関わらず、関連しあっていることから、物事をとらえていく試みです。

ご縁モデル：その１

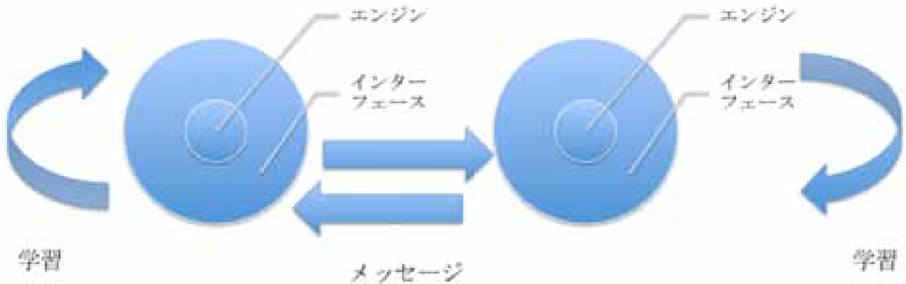


まずは、とても簡単な物理現象を取り上げてみましょう。発表時、手元にあった筆入れを落としてみます。この筆入れの落下にかかる時間は、ニュートン力学の自由落下運動の公式（りんごですね！）を使って簡単に計算できます。例えば、10 階会議室の窓から地上まで筆入れを落とした場合の落下時間は、風の抵抗も無視できなくなるでしょうから、この筆入れの投影面積を求め空気抵抗を入れて計算できそうです。しかしながら、実際に落としてみると、上昇気流のようなビル風の発生や、投げ方によってはくるくる回転（スピン）や障害物やビルへの衝突など状況が変化しますので、厳密な筆入れの落下時間の厳密な計算はおそらく難しくなるに違いありません。さらに、落下しているのが筆入れではなく、ひらひらしている A4 の紙の場合はどうなるでしょうか。もちろん計算できそうですが、さまざまな仮定をおく必要がありそうです。つまり、実際の世界では単純な落下といえども、なかなか厳密な計算は難しそうです。簡単な物理現象で既知だと考えられていても厳密な計算は難しくなっていく例として、手元にあった筆入れの落下をあげてみました。もし、計算の難しさがまだまだ足らない場合には適切な問題が他にもたくさんあるかと思います。

さて、筆入れの落下にかかる時間を予想する別の方法も検討できます。実際に落として落下時間を計測し、その結果を予想に使うことです。筆入れを 10 回落とせば、おそらく 11 回目に落ちる時間も予想できそうです。筆入れを 100 万回も落としたら、100 万 1 回目の落下時間をより的確に予測できそうです（けど、象に踏まれても壊れないぐらい丈夫な筆箱を使う必要がありそうですね）。100 万回の落下実験を行うと、そのつど、壊れ具合や状況などが変わってきますので、風の強さや筆入れの状態などの状況も記録しておくことによって、その結果、似たような状況での落下時間をある程度の精度で確率的に予想できそうです。

例の取り上げ方が少々乱暴ではありましたが、世界第二次大戦時代、ノーバート・ウィーナーが自由に飛び回る戦闘機に打ち落とすため、軍艦から打つ砲丸の制御する必要性からまとめたサイバネティックスが このような解決方法のヒントになっています。サイバネティックスは、出力した結果（の一部）をフィードバックされた入力として扱います。もちろん、サイバネティックスを使えば、シミュレーションの計算が不要になるわけでは決してありません。お互いに補完的な関係や、とらえかたによっては階層的な関係を構築できます。ただ、この計算結果を実際の世界に出力するだけでなく、実際の（実験）結果をフィードバックの入力として取り込む系を構築するところがポイントです。つまり、実社会の現象とのメッセージのやり取りがとても重要になってきます。

筆入れの落下時間を予測する際、実際の計測結果をフィードバックとして入力し、蓄積します。次の落下時間を予測するときは、その蓄積されたデータを参考にします。落下時間の実績を蓄積していくことを「学習」と表現しました。このシステムの構成は､受け取ったものを認識し、知識として取り込む学習部分と、知識から適切なものを出力する部分がエンジンとして構築されます。さらに､蓄えられた知識部分も構成されると考えています。



一般化して、自システムから出たメッセージを、他システムへ送り、他システムにおいても相互に学習するモデルにしました。

なお、コミュニケーションやメッセージなどの「関連」の重要性を訴えるために、ユーモアをこめて存在（いいやつ）は中身とコミュニケーションの二乗である e=mc2 というエイプリル・フール・ネタも合わせて示しました。

ご縁モデル：その２

現実には多システム間での連携もしています。ここでのシステムとは「相互に影響を及ぼしあう要素から構成される、まとまりや仕組みの全体。系。(Wikipedia より)」を示しているため、必然的に多システムの連携は再帰的・階層的なシステムやパターンになります。

多システムを、たとえばベイジアン・ネットワークやマルコフ連鎖を使用して、不確実で複雑な構成を（主観）確率の連鎖で記述できそうです。ベイジアン･ネットワークでは、それぞれのノードの確率の相互作用から、ある事象における「もっともらしさ」を求められます。ベイジアン・ネットワークやフィードバックの利用は、系の方向性などから階層構造や再帰構造など別のレイヤーで構成する必要があるかもしれません。

ゆるいソフトウェアに向けて

今まで、ビジネスやソフトウェアの典型的な進め方は、できる限り形式化し、それを計画に沿って厳密に構築、実行していく方法が多かったように感じます。しかしながら、現実の世界では相互作用や刻々と変わる流動的な現象であり、事前に計画することは困難です。計画が実行される時に状況はすでに変わっているため、事前にビジネスやソフトウェアを計画すること自体がもろさに結びつきやすいのではないかと考えています。

「強い」ビジネスやソフトウェアには、刻々と変わる現象を柔軟に扱う必要があります。また、計画時点では ぼんやりしているもの（暗黙知 tacit knowing）や背景にある知恵・知識も発見し、できる範囲で取り込んでいきます。今回はご縁と表現しましたが、相互作用を行い、いかにシステムを育てていくのか、という枠組み作りの視点がより重要になります。そのための解や方向性のひとつとして「ゆるさ」の観点の重要さを提案します。「ゆるい」は多義的ですので、大槻氏にまとめていただきました。

ゆるい#	定義名称	説明	典型的対象分野
ゆるい1	抽象的	設計自由度	全般
ゆるい2	不確実	確率・非決定的	予測、フィルタ
ゆるい3	曖昧	未認識・未定義	全般
ゆるい4	相対的	言語ゲーム的	社会、対話
ゆるい5	進化的	メタ・状況適応	パターン認識
ゆるい6	計算量限界	近似解	レイアウト、検索
ゆるい7	未解明項	調整・フィードバック	制御、ログ解析

事前に厳密に文書化された計画には限界があります。いろいろなモノやコトは厳密な認識は困難であるため、ゆるい、という視点を持つことによって、変化する状況に対し、より柔軟で的確に進むことを期待しています。ゆるさの視点を適用することにより、目の前の要件や要求を直接的にアプローチするだけでなく、より本質的で柔軟な間接的なアプローチを取れるのではないかと目論んでいます。その結果､価値が高く「強い」ソフトウェアやビジネスの構築に貢献できると考えています。

ご縁モデルの例として紹介したサイバネティックスやベイジアン・ネットワークは、かなり古くから研究されており、いくつかのエンジンもあります。また、すでに「ゆるさ」を必要としている分野や部品があり、すでに実装され、実際に利用されています。発表で

は、某所で開発中の製品の例から説明も行いました。しかし、開発中ということもあり、この資料では割愛させていただきます。

現実の社会には、実際のビジネスやソフトウェアの世界に「ゆるさ」がフィットする分野や問題があり、そこには「ゆるいソフトウェア」を導入していきたいと考えています。学習するご縁モデルなどよって、状況や相手に応じて「ほしいところに手が届く」、さらには「阿吽の呼吸」の状態に近づいていくと期待しています。ただ、阿吽の呼吸であっても、たまに外します。そこは確率ですのでご容赦くださいませ♪

知働化に向けて

今までは、ウォーターフォール型開発プロセスに象徴されるように計画段階で知識を形式化し、それを構築することが主でした。今後は、偏在する知識や知恵と、それをダイナミックにつむぎだすことによって、より強いシステム（系）やビジネスの構築を目指します。トヨタ生産方式にもあるカイゼン（改善）の知恵を機械やシステムに織り込み、持続的に行うためにも人が関わる「自働化 (にんべんの働)」の考え方を、より明確にして推進できる知働化が重要であると考えています。天動説から地動説へのパラダイム変化のように、今後はソフトウェアやビジネスのパラダイムも変わっていくと考えています。

今後は、実際にゆるさや知働を組み込むチャレンジを行い、経験を共有させていただけたら幸いと思っています。その中で、ゆるさ自体のパターンを抽出するとともに、ビジネスやソフトウェアを構築するために、ゆるさや知働をマッチするパターン（と、アンチ・パターン）を抽出し、知見としてまとめたいと考えています。

〔プロフィール〕

農学修士／鉄人(アイアンマン･トライアスロン)。研究所勤務、コンサルティング会社の設立･経営を経て、外資系保険会社勤務。ウェブシステム開発、テクノロジ、要件定義／要求開発、教育、マネジメントなど幅広く経験。学業や勤務のかたわら携帯 PC 技術研究所や知働化研究会で活動している。

読者からのコメント

濱勝巳氏からのコメント：

ゆるの説明の中で何度も「ゆるっとＸＸすること」がゆるだと述べられているように感じました。これだと結局のところゆるが何だかわかりません。「全ての物はゆるである」であると言いきってしまえば受け入れざる負えないのですが、「全ての物」ではないんですよね。

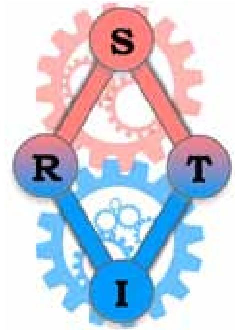
私も、過度の緊張やがんじがらめの制約や条件というものだけでは物事は上手くいかないと思っているので、ゆるという概念が必要であるとは思っています。しかし、「ぐだぐだ」は、ゆるではないと思っています。ゆるを説明するのは難しいことだとは思いますが、ゆるという言葉自体がゆるの説明の逃げ道にならないよう、ゆるに必要な軸や中心、骨格が見えてくる事を期待しています。

論説

〔知働化研究会誌創刊第１号〕
知働化研究会
Executable Knowledge and Texture Laboratory

ΛVモデル

V字モデルからの意味論的転回



大槻 繁（おおつき しげる）

株式会社一（いち）
otsuki.s@1corp.co.jp

概要……106
はじめに……106
Ⅰ．V字モデルの復習……107
Ⅱ．知働化への転回……109
Ⅲ．ΛVモデルの概要……113
Ⅳ．いくつかのケーススタディ……115
ケース１：伝統領域でのソフトウェア開発会社……115
ケース２：Web サービス領域のソフトウェア開発会社……118
ケース３：ユーザ企業内製のシステム部門……122
ケース４：新規領域開拓型研究機関……124
考察とまとめ……126
ケーススタディのまとめ……126
プロセスの特徴付け……127
なんちゃってマニフェスト……128
執筆後記と参考資料……129
〔参考図書〕……130
〔筆者の知働化関連の書き物〕……131
〔プロフィール〕……131

Copyright 2010, OTSUKI Shigeru, All rights reserved.

概要

本論文は、知働化のパラダイムに移行していくために、あえて、伝統的なソフトウェア開発プロセスの基底となっているＶ字モデルを転回して、知働化の意義を再認識してみようという試みです。一言で私の主張を集約するならば、「ソフトウェアエンジニアリングの世界で言語ゲーム的転回を進めよう」ということです。新しい世界を描く努力なくして、未来は切り開けません。

筆者は、知働化をポスト・アジャイルプロセスと位置づけており、「不確実性への対処」の次に来るものは、「進化・適応」のプロセスだと考えています。一方で、伝統的なプロセスは依然として残り続けることでしょう。そこで、これ等両者の関係を整理しておくことは、それなりに意味があると考え、「ΛＶモデル」というハイブリッドなプロセスモデルを考えてみました。

はじめに

この論文を執筆しようと思った理由は、2009年末にSEC（Software Engineering Center）の非ウォータフォール研究会のメーリングリスト上でＶ字モデルに関するやりとりをした際に、少し頭の整理をするために自分なりの見解をまとめておこうと考えたことにあります。2010年１月１日に同名の小論を発表し、これを再編集したものが本論文です。

表紙の絵柄の背景に配置した歯車の図は、ここのところ筆者が気に入って使っているものです。青で示したITシステム・ソフトウェア、あるいは、エンジニアリングの世界と、赤で示したビジネス、あるいは、実世界とが、連動してかみ合うべきだということを象徴的に表現したものです。青と赤の色使いは、全体を通して一貫させています。

全体構成は、４部構成で、ＩとⅡが前置きとか背景で、中心はⅢです。最終章ⅣでいくつかのケーススタディをΛＶモデルの観点から考察しています。

Ⅰ．Ｖ字モデルの復習

ウォータフォール型開発プロセスの原理を説明するモデルとしてＶ字モデルを復習します。

Ⅱ．知働化への転回

知働化のエッセンスを紹介し、開発プロセスの観点からどういった転回が進んでいるかを簡潔に示します。

Ⅲ．ΛＶモデルの概要

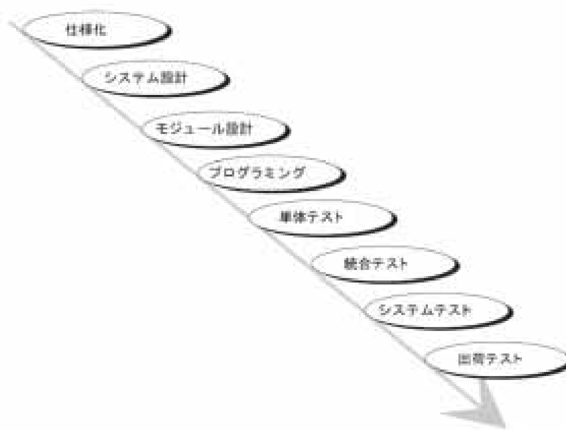
Ｖ字とそれに対峙するΛ字の部分の関係、モデルのポイントを解説します。

Ⅳ．いくつかのケーススタディ

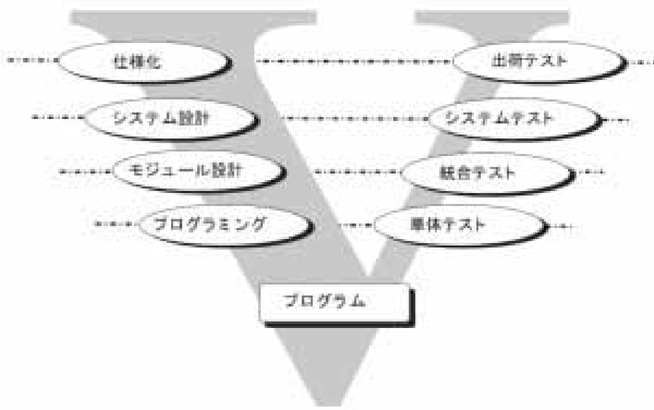
身近な事例をΛＶモデルの視点から眺めてみることにします。

Ⅰ．Ｖ字モデルの復習

ウォータフォール型開発プロセスは、前工程で得られた中間成果物を確定・固定化した後に、それを前提として該当する工程の作業を行う方式です。前工程での誤りが、後工程に持ち込まれると手戻り（フィードバック）を生じ、全体の工数も増加してしまうことになります。「ウォーターフォール型」という名前の由来も、工程を進めるということを、滝を下ることに見立てているところにあります。滝を逆に登るのが大変であることも、手戻りのコストが高いということをうまく表しています。



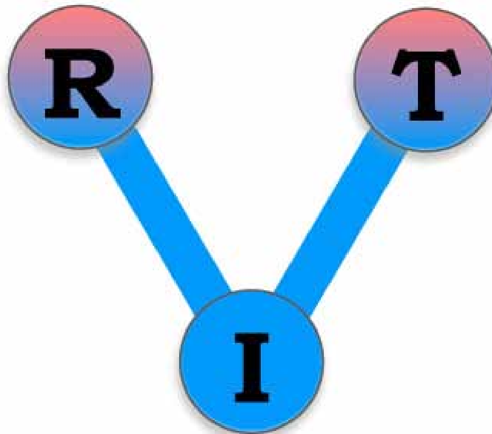
仕様化、システム設計、モジュール設計、プログラミングのそれぞれの工程の正しさを、出来上がった成果物＝プログラムに照らし合わせて確認する工程がテスト工程です。このことをよりわかりやすく表したのが、「Ｖ字モデル」です。



ウォータフォール型開発プロセスをＶ字モデルによって解釈することによって、旧来の開発プロセスの欠点が明らかになります。それは、最初に行った意思決定の正しさが最後にならないとわからないことです。

Ｖ字モデルは、各工程が左側から右側へ時間的な順序を表わしていると見なせばウォータフォール型開発プロセスになりますが、時間的な関係を捨象してしまえば、段階的モデル（Incremental Model）や進化型モデル（Evolutionary Model）などの説明のモデルとしても使用することができるようです。

Ｖ字モデルの工程の個数や名称にさほど意味があるわけではありません。そこで、簡潔に、象徴的に以下のように表わすことにします。

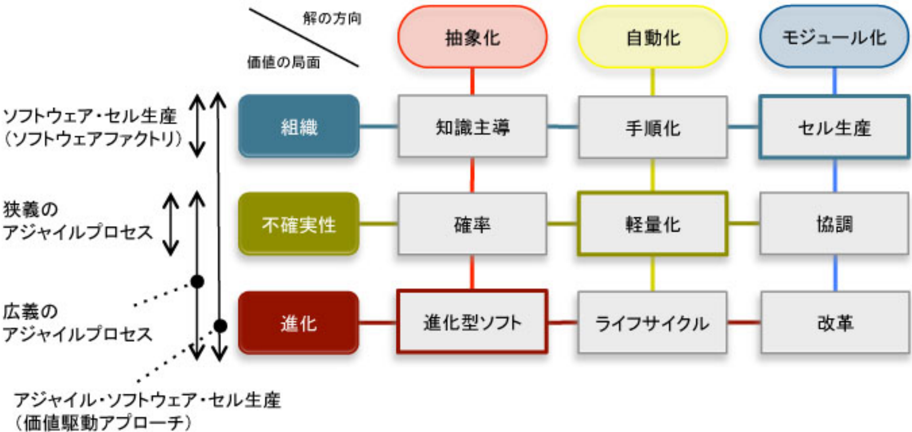


R は要求（Requirements)、I は実現（Implementation)、T はテスト（Test）を表わしています。「実現」とは実行可能なコード、および、そのコードを構築するための設計、さらに、その設計を確認するための内部テストも含みます。「要求」は実現されたコードの作用の対象となる実世界の現象に対する要望や願望です。その一部がコードに対する仕様に対応しています。「テスト」とは、「実現」が「要求」を満たしていることを、コードを実行することによって確認することです。

非常に簡単に言えば、R という期待を持ち、I を実行させ、R の期待通りかどうかを T によって確かめるということです。

II．知働化への転回

アジャイルプロセス協議会の活動の中で、アジャイルプロセスの意義や今後の方向性について、検討を重ねてきました。2008 年 6 〜 7 月にかけて、見積・契約 WG のメンバの方々と議論して、最終的に以下のような方向性を得ることができました。特に、アジャイルプロセス、および、それを遂行する組織が生み出す価値が、不確実性への対応の段階から、ライフサイクルを通じた進化（と適応）に移行していくという共通認識に至りました。



2009 年夏に発足した知働化研究会では、実世界でのソフトウェアの作用に焦点をあて、価値駆動のプロセスを検討し始めています。知働化のコンセプトは、以下のように集約されます。

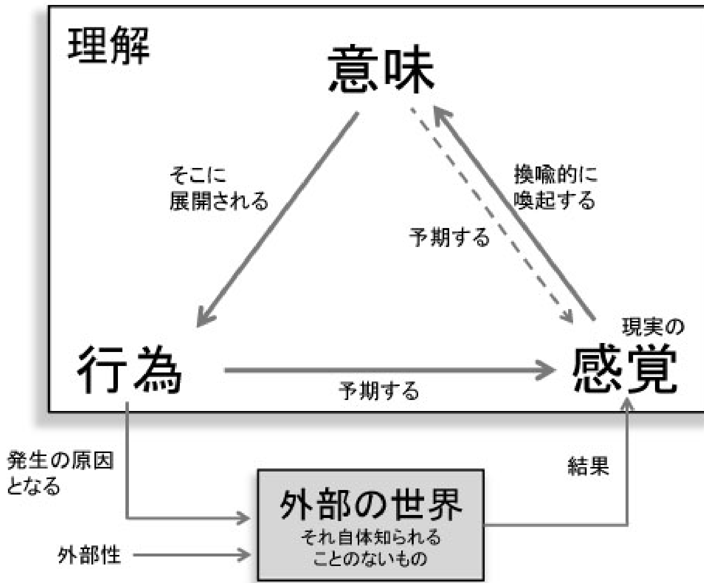
* ソフトウェアに対する新しい見方
  - ソフトウェアとは , 実行可能な知識の集まりである
  - ソフトウェアとは , 実行可能な知識を糸や布のように紡いだもの (様相) である
  - ソフトウェアを作る / 使うとは , 現実世界に関する知識を実行可能な知識の中に埋め込む / 変換する過程である
  - ソフトウェアを作る / 使う過程では , 知識の贈与と交換が行われている
  - ソフトウェアを作ることと使うことの間には , 本質的な違いはない
* 実行可能知識と様相／テクスチャ
  - 様相／テクスチャとは、「動く、問題と解決の記述」のことである
  - 「機能」を実現することから、顧客の「知識」をコンテンツ化し、実行可能にすることへ

従来のパラダイムでは、ソフトウェアがもたらす価値、ビジネスモデル、要求の発生プロセス、ソフトウェアの実行による実世界での認識の変化といった事項に対して思考停止していると考えられます。

これはパラダイムシフトです。従来の世界観や価値観は通用しません。ソフトウェアというのは、元来、実世界の問題を解決するものです。既に解かれている同様の問題を、何回も解き続けるということでしたら、従来の、工業的なパラダイムで済むでしょう。実際に、画面のレイアウトや入出力の仕様が決まったら、そのコードを書くことは、手順化されているでしょうし、自動化も可能でしょう。筆者は、このようなパラダイムは、たとえ大規模化や組織化が必要だとしても、本質的な課題はそこには無いと考えています。

実世界におけるソフトウェアのもたらす意味は、人、あるいは、組織にとって主観的であり､多様です。また､ソフトウェアは､人間が創造する人工物（アーティファクト）です。人工物の「デザイン」に関する基礎理論、哲学として、知働化のパラダイムに最も適合している考え方が、クラウス･クリッペンドルフ（Klaus Krippendorff）の『意味論的転回』です。

クリッペンドルフの主張は、「科学」というのは、過去に起こったことを分析して何か法則を見いだしたり証明したりするのに対して、「デザイン」というのは将来について意思決定していくので、ぜんぜん違う、「人間中心」の「意味」を扱う体系でなくてはならないということです。



デザインでは、人間の理解、意味とはどういったものかを明確にしておかなくてはなりません。無論、デカルト主義を排す立場では、絶対的、客観的真実というのは無いわけで、ヴィトゲンシュタインの「言語ゲーム」的に規定していくことになります。

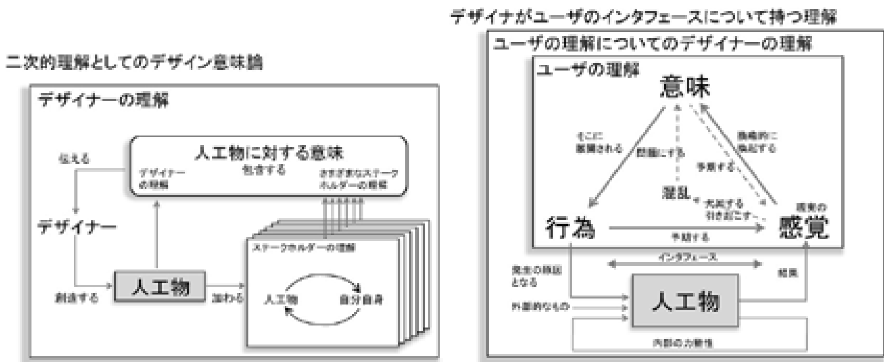
意味とは、以下のように規定されます。

- 意味は、構造化された空間、期待される感覚のネットワーク、一連の可能性である。意味は行為をガイドする。
- 意味は、いつも誰かの構築物である。
- 意味は、共有できない。
- 意味は、言語の使用の中で現れる。
- 意味は、固定されない。
- 意味は、感覚によって引き起こされる。
- 意味は、知覚されたアフォーダンスである。

図で「外部の世界」というのが、ソフトウェアの実行による作用によって影響を受ける実世界です。「意味」そのものは、直接語ることはできません。人間が持つ意味は、実世

界に対する「(言語）行為」として現れ、その行為による因果関係によってもたらされる現象が、「感覚」として帰ってきます。

人工物をデザインするには、この枠組みに従って、２次的理解を定義します。つまり、人々が「理解することを理解する」ことによって、それをデザイナーの意思決定に活かしていけるような枠組みを提示しています。

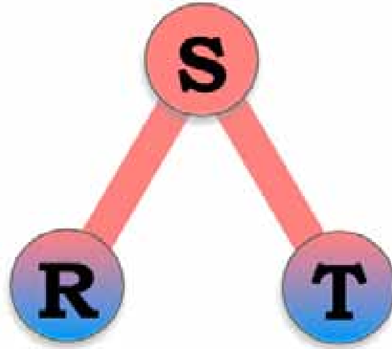


クリッペンドルフが提唱している、人工物（ソフトウェア以外も含む）に対するデザイン原則は、以下の通りです。

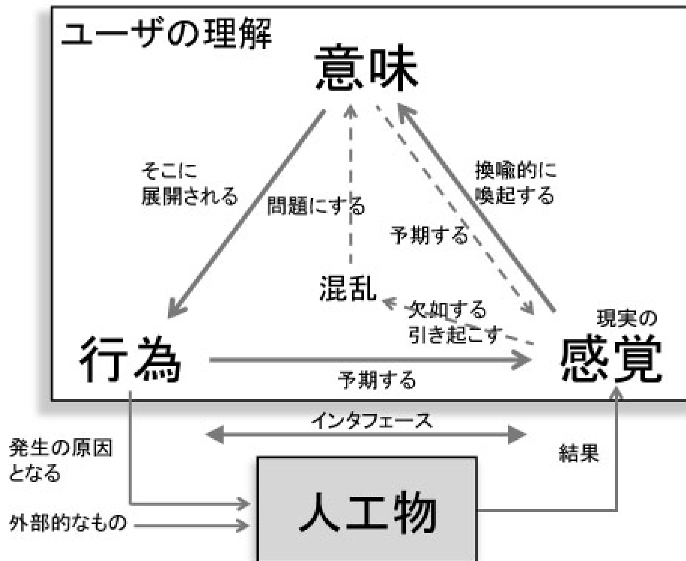
* 人間中心性：技術の概念化、理解と使用を意味の前提とする。
* 意味のあるインタフェース：認識→探求→信頼の移行を促進。
* 二次的理解：ユーザの概念モデル、シナリオを知る。
* アフォーダンス：人工物の使用における意味の基本単位。
* 制約：ユーザの注意や相互作用を方向付ける意味的な方法が望ましい。
* フィードバック：ユーザ行為の即時的、直接的フィードバックが望ましい。
* 一貫性：意味の層を定義。
* 学習可能性：ユーザが学習していくことをデザインする。
* 多感覚の冗長性：視覚、聴覚、触覚等複合的なもの。人による感覚の差。
* 変動性ー多様性：人工物の変動性は、ユーザの集団における多様性と一致。
* ロバスト性：障害や事故回避。
* デザインの委託：ユーザがデザインすること

Ⅲ．ΛＶモデルの概要

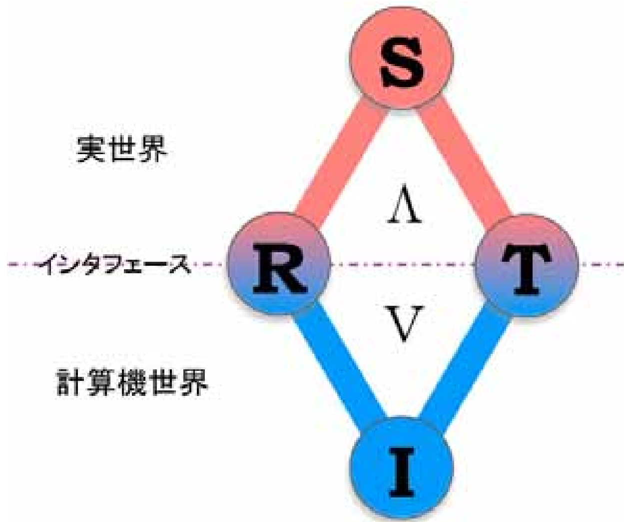
前節の意味／行為／感覚の三角形に、ソフトウェアの要求とテスト（実行の反応）を当てはめてみることにします。すなわち、S は意味 (Semantics)、R は要求 (Requirements)、T はテスト (Test) を表わしています。V 字に対峙して、Λ字モデルとでも言えるでしょう。



Λ字モデルでは、人工物としてのソフトウェアが実行することによる、認識の変化・修正を扱うことができます。(T は Test というより Track や Trace の T とした方がよいかもしれません。)



第Ⅰ節のＶ字モデルと統合すると、以下のようになります。



これがΛＶモデルです。

（１）そもそもＲなんて判らない（語れない）かもしれない。

（２）でもＲを満たすＩを作る（既にあるものかもしれない）

「実行」できるのはＩだけ。

（３）だからＩを実行させてＴをやる。

ＩがＲを満たしていない場合には、Ｉの実現方法に誤りがあることになる。

（これは、従来の伝統的な内部テストの問題）

（４）Ｔという作用を受けて<Ｓ意味（認識）が変わってＲ（要求）が変わる>

ことがある。

「想い」を発する根元であるＳ（意味：これは語り得ないものです）があって、これに基づいてＲ（要求：これは一般的にはある程度語ることができます）が発せられ、Ｔ（テストや運用）からフィードバックを得て、Ｓが修正され、新しいＲが出るといったプロセスを繰り返す。

Ⅳ．いくつかのケーススタディ

以降、４つの事例をΛＶモデルのケーススタディとして考察してみます。

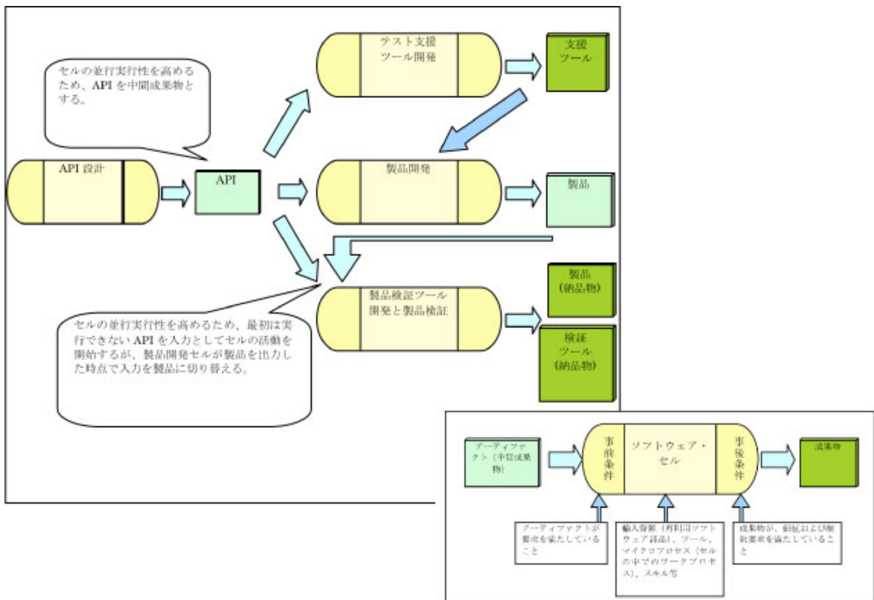
ケース１：伝統領域でのソフトウェア開発会社

＊　中堅の独立系ソフトウェア開発会社の事業部

＊　数十人月〜百人月程度の中規模案件の請負受注が多い。

＊　2005 年頃よりアジャイルプロセスの味見（なんちゃってアジャイル）

＊　2007 年頃よりアジャイル・ソフトウェアセル生産方式を確立



＊　日常的に、タスクカード（ソフトかんばん）、バーンダウンチャート、朝会、ふりかえり等のプラクティスを行っている。

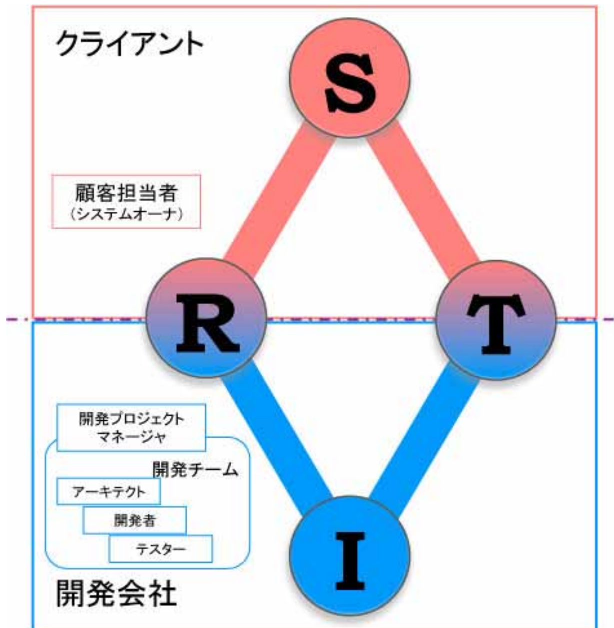
＊　新人は定常的に採用しており、アジャイルプロセスの実践チームは、人材育成のために研修的に参加することもある。

＊　進化型のフレームワーク（Groovy/Grails など）やプロセスの構築も継続的に行っている。

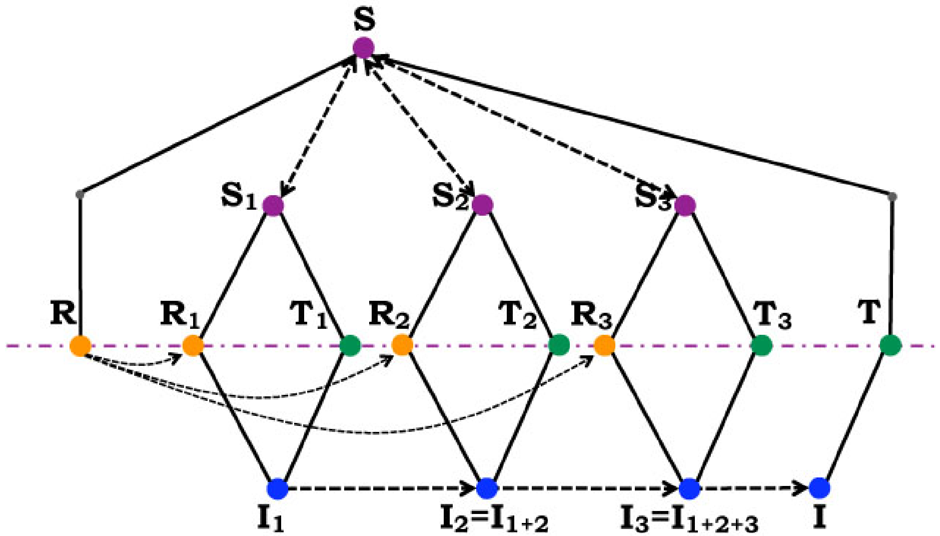
〔プロジェクト例〕

* 請負契約による約１年程度の受注
* 顧客から要件定義書が提示。要件定義の曖昧や不備などがあるが、基本的に変更はない。
* ソフトウェアは、実験などの測定データを管理する目的で使用され、顧客側のビジネスに直結するものではない。
* 開発チームは１０名程度。内５名がエキスパート、残りが新人。
* 開発プロセスは、インクリメンタル（段階的拡充）型開発プロセスとし、１～２ヶ月程度のイテレーションを繰り返すものとする。
* イテレーションごとに顧客側の確認をとり、要件の変更もある。従って、安定しているもの、重要なものから実装していくこととする。
* 実装環境は、Java, Groovy, Struts, Spring, Hibernate など。デザインパターンやアスペクト指向設計なども行う。
* テストは、イテレーションごとに回帰テストを行う。

顧客と開発企業との間の組織境界は、ΛＶモデルにおける実世界と計算機世界との境界と一致しています。従来の典型的な請負受発注と考えられます。



時系列でプロセスを俯瞰してみると、以下のようになります。



ソフトウェアの価値は、クライアントの世界や意味 S に関わることです。開発企業からは見えません。要求 R は、最初にクライアント側から文書の形で提示されます。これを受けて、開発企業では基本設計やそれに続く詳細設計が進められます。インクリメンタル型の開発プロセスをとるために、最初の段階で、全体の要求に基づいてイテレーションの分割を行い、各段階での要求を明確にします。

それぞれのイテレーションは、決められた要求 Rn（n=1 ～ 3）に対し、実現 In を開発します。Rn を意味付けている意味 Sn は、おそらくこのソフトウェアに関わる全体の意味 S の一部です。テスト Tn は、実現 In が要求 Rn を満たしていることを確認する行為です。In の実行による認識の変更も発生し、これは、次のイテレーションの要求 Rn+1 に反映されます。

イテレーション n とイテレーション n+1 との関係は、前者の拡充が後者という位置づけです。実現は、In を拡充して In+1 とします。Tn+1 は、Tn のテスト項目を回帰的に適用することになります。

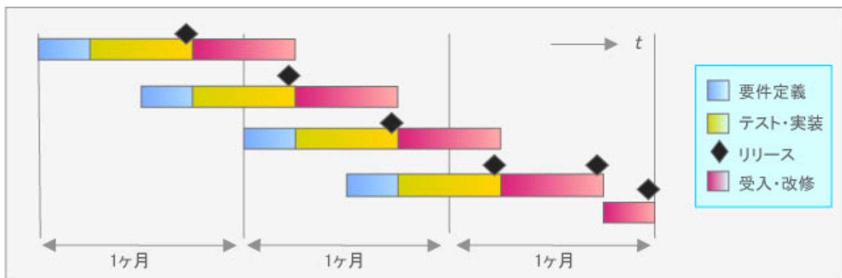
最終イテレーションの結果が、このプロジェクト全体の結果となります。

ケース２：Web サービス領域のソフトウェア開発会社

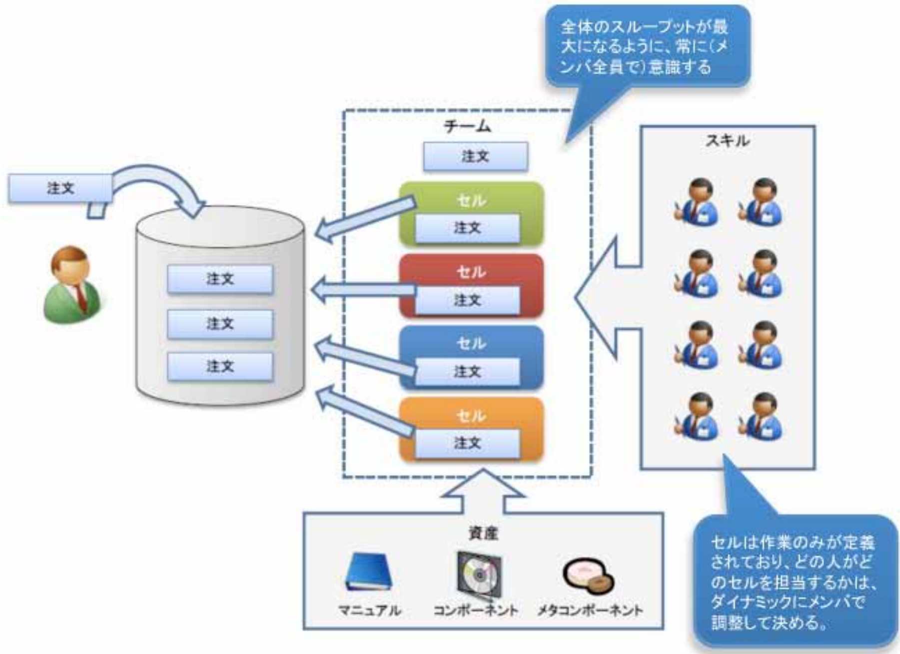
* ＊　アジャイルプロセスを積極的に取入れている小規模のソフトウェア開発会社
* ＊　Web システム開発の分野を得意としている。
* ＊　独自のアジャイル・ソフトウェアセル生産方式を確立している。
* ＊　アジャイルプロセスを、より組織化し、より大きな案件も開発できるようにしていきたいと考えている。
* ＊　アジャイルプロセスの基本的なプラクティス：朝会、振返り、イテレーション、バーンダウンチャート、週４０時間労働などは全て導入している。

この会社のアジャイル・ソフトウェアセル生産の特徴は、タイムボックス方式と呼ばれる固定間隔（２週間）のイテレーティブな方法を取入れている点です。クライアント側から見ると、定期的にリリースを受入れ、検収を行わなくてはならないことになります。イテレーションを継続していき、先々の予想は困難ですから、契約は、本来、保守契約のようなものが適していると考えられます。

- 母体の開発は1ヶ月程度で完了させる
- 実装期間を2週間に固定する
- 情報システムのライフサイクルが尽きるまで継続される
- 2週間でテスト・実装期間が全て完了していなければならない
- 2週間毎に必ずユーザへリリースし受入検収を受ける
- 要件定義と受入検収とテスト・実装の期間は重なる



もう一つの特徴が、「セル」の導入です。これは所定の作業をこなす仮想的な人格と見なすことができます。「人月からの脱却」という観点では、仕事の工数を「セル・月」で把握するということになります。このように作業（役割）と人的リソースとを切り離すことは、マネジメント上、多くのメリットを得ることができます。



（1）クライアント側に対して人員（人月）を隠蔽することができる。

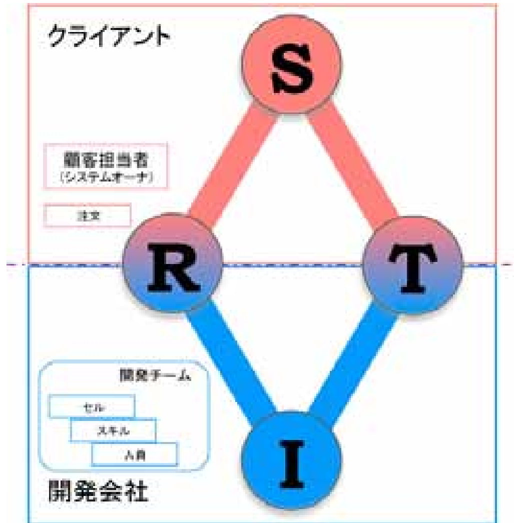
（2）セルはクライアントからの注文に直接対応しているため進捗が可視化できる。

（3）セルは並行に実行できるため段取りに気を使わなくてよい。

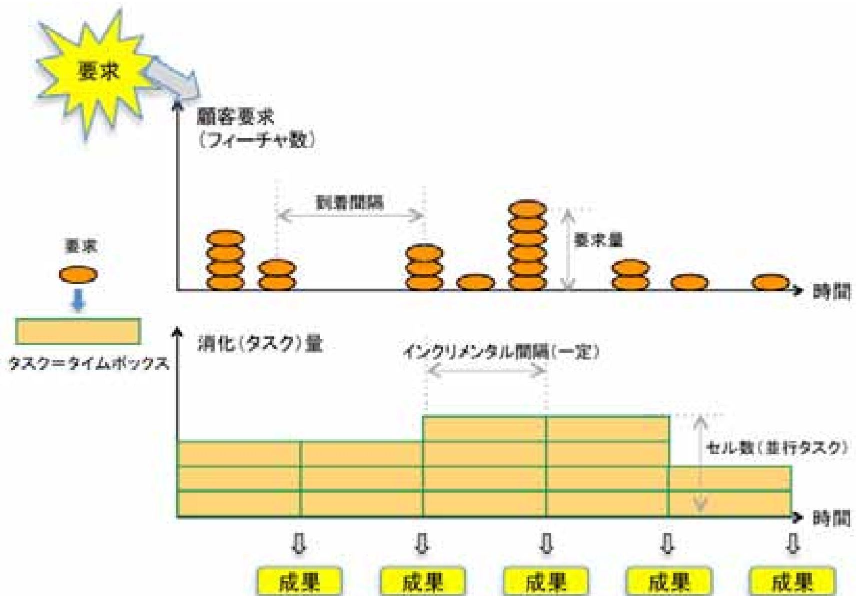
（4）チームメンバは、セルのスループットを向上させることのみに注力すればよく、上位のマネジメントに無駄がなく軽量化できる。

（5）セルの進捗を見てダイナミックに人の割当もチーム内で調整しながら進めることができる。

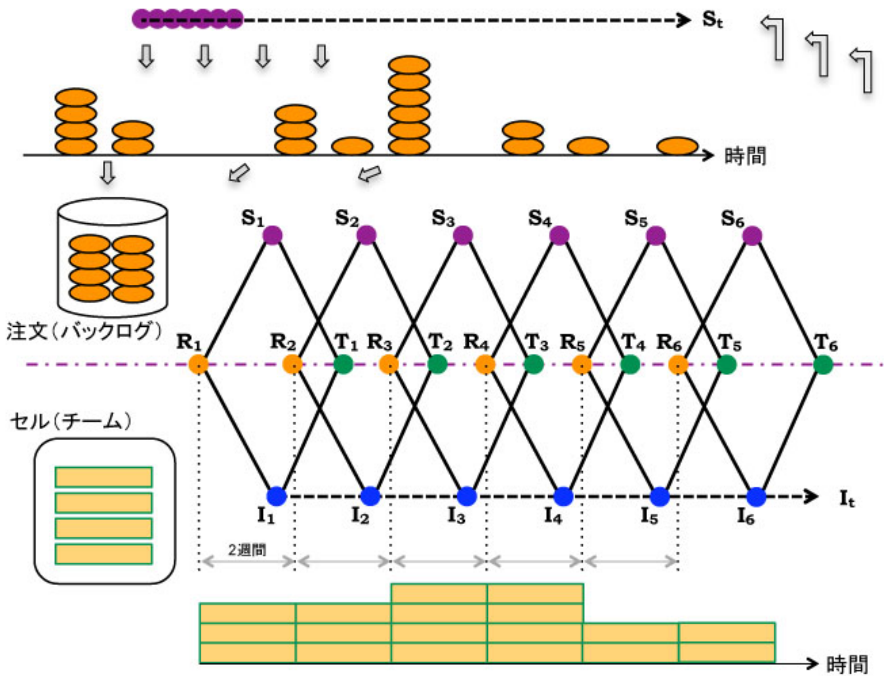
顧客と開発企業との間の組織境界は、ΛVモデルにおける実世界と計算機世界との境界と一致しています。従来の典型的な請負受発注と同様です。



クライアントからの要求は、いつどれくらいでてくるかは不確実です。アジャイルプロセスとは、明確になった要求（仕様）を、明確になった時点で速やかに実現するということです。



タイムボックスマネジメントによる、ダイナミックな要求とセルによるタスク消化の状況は、以下のように表わすことができます。



クライアント側からの要求は、時々刻々と不確実に出されます。これを注文バックログとして溜めます。これが､クライアントと開発企業との間のバッファとなります。注文は、開発チーム側でセルに割当てられ消化されていきます。注文の量は､波があると考えられ、多くなりそうな時期にはセルをたくさん用意して準備しておきます。

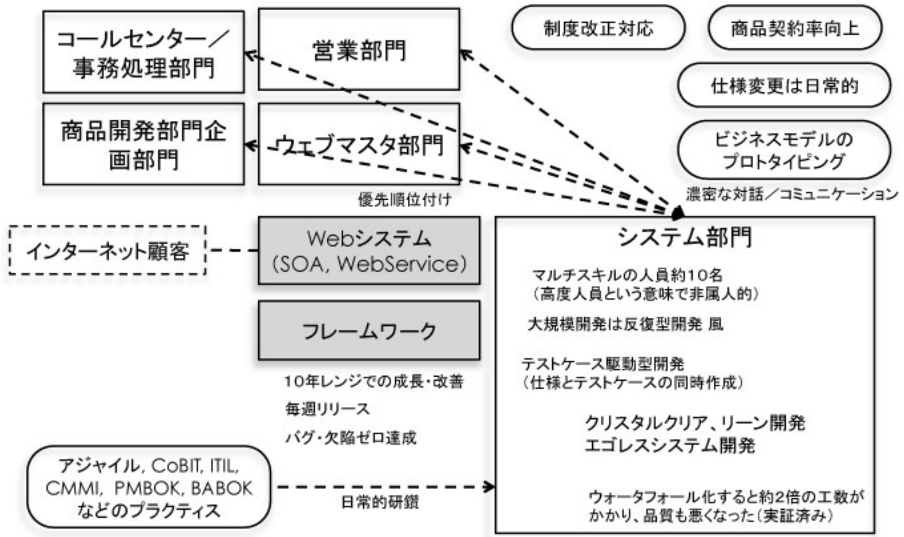
ケース３：ユーザ企業内製のシステム部門

＊　インテーネットからアクセスする顧客に対して商品案内から契約に至る支援をする Web システムを継続的に構築・維持している。

＊　ユーザ企業内製型のプロセスである。

＊　営業、業務部門などとのコミュニケーションを密にして、常時、新規フィーチャ、改善要求について検討している。

＊　プロジェクトという概念は無く、システム部門の約１０名の体制で、週１回のリリースを繰り返す方法を採っている。

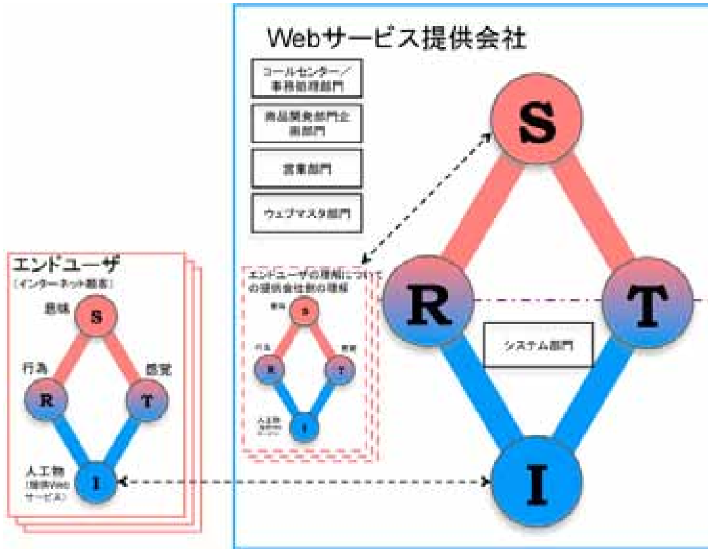


＊　開発チームは、長い年月をかけて成長・改善をしてきており、レベルが高い。

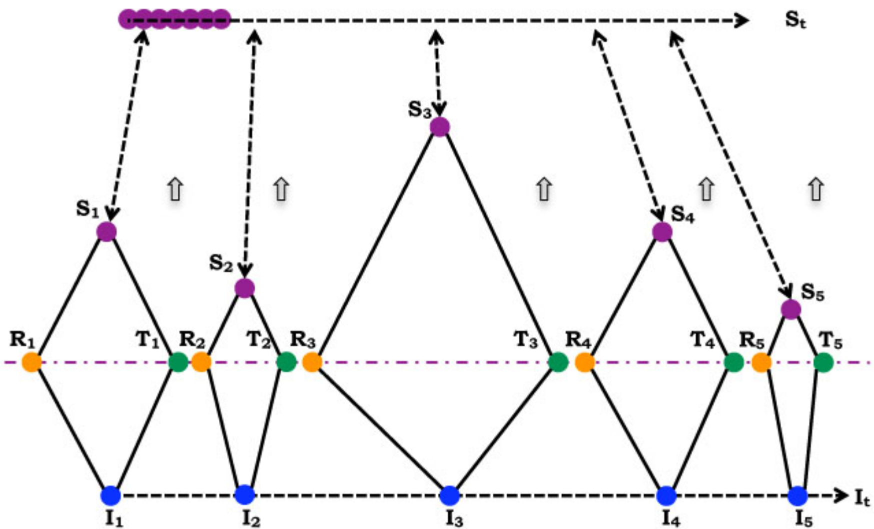
文献や書籍の読込みと、徹底した討論をすることによって、チームに導入すべきものを吟味している。

システム開発の負荷が少ない時に、育成対象の人材に責任を持たせて業務を遂行することによって、業務ノウハウの伝承を行っている。

＊　エゴレスシステム開発を実践し、チームメンバ全員がどの部分、どういった開発も可能なようにして（チーム内の）非属人化を図っている。



エンドユーザの提供 Web サービスに対する意味、文脈、理解（誤解）などの状況は、営業部門などとシステム部門とがコミュニケーションを密にとることによって、ニーズの把握、既存システムに対する改善項目など、常に的確に把握できるようにしています。すなわち、システムの意味が時間とともに変化していきます。



ケース４：新規領域開拓型研究機関

＊　「連想検索」と呼ばれる人間の創造的活動を支援するための新しい情報アクセス方式を核とした活動

＊　基礎研究によって得られた「連想検索」の方式に基づくエンジンを開発

これは３回ゼロからコードを書き直す方法で、順次、インタフェースやアルゴリズムを洗練化（約６年程度かけている)。

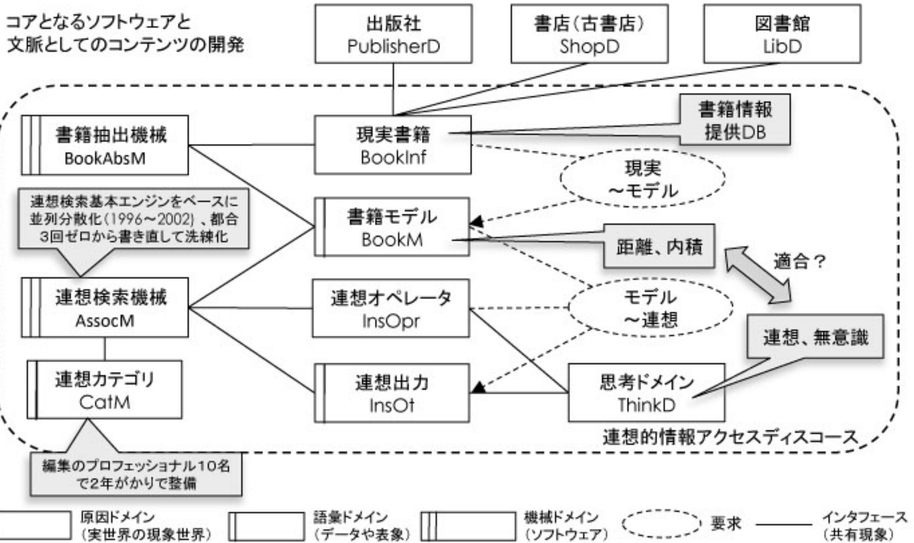
＊　連想検索エンジンを活用した Web サイトの構築をプロデュース。

（書店、図書館、古書店など）

＊　連想を効果的に、価値あるものにするために、書籍のカテゴリ分類を行う。

これは出版社の編集長レベルの人に２年がかりで作成してもらった。

＊　連想検索の効果を実証できたため、複数のサイトを連携して検索を行う仕組みを作り、実現した。

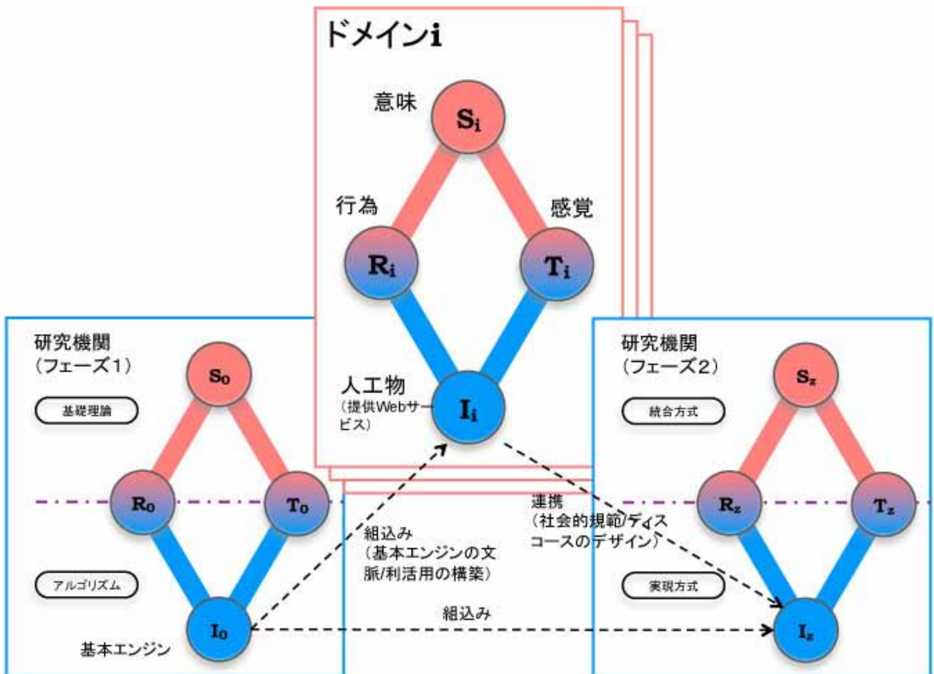


基本エンジンを開発するところは、従来の開発プロセスと考えてよいでしょう。仕様や方式をしっかりと固めてから、コードを書くというものです（次図左下部分)。次の各ドメインの Web サイトの構築は、基本エンジンを活用しながら、それぞれの分野の知識やノウハウをコンテンツとして埋込んでいく作業です。古本屋のおやじさんがどういった仕事をしているのか、図書館の司書さんの仕事、分類の仕方はどうなっているのかといったことが解明されていきます（次図中央部分)。ここで重要なことは、システムに埋込むべ

き知識が、システム外から来ることです。その中でキーになるものが、出版物のカテゴリ情報です。これを、ゼロからの発想で、出版社の編集長に参加して作ったというところが、連想検索の価値を高めています。そもそも、人間が創造的活動をして検索を行う場合、何か目的があって探し物をするわけではなく（そういう場合にはGoogleのキーワード検索を使えばよい)、何か関連しているもの、想いもよらないものなどを発想の刺激として出してほしいわけですから、出版社の編集長のような横断的に見る力というのが組込まれていることがよい結果を生みそうだと直感的に思えます。

最後のサイトを横断した連想検索を行う統合部分については、それぞれのドメインでの結果を、他の検索に活かす方式を、分散検索の方式を考え実現したものです。これは、ソフトとしてはAjaxを使って、小さな規模（数ヶ月）で実装できるものです（次図右下部分)。

所謂、ソフトウェア開発という観点からは、受発注の関係や、要求をどうするかといった観点はさほど重要ではなく、人間の発想とはどういうものかとか、今までにないディスコース（言説）をデザインしていく活動で、まさに「知働的」なものと言えます。



考察とまとめ

ケーススタディのまとめ

前節のケーススタディの４つの例は、アジャイルプロセス、あるいは、知働化のいくつかの典型的なものと言えるでしょう。

ケース１：伝統領域でのソフトウェア開発会社

「狭義のアジャイルプロセス」を実践しつつ、「セル」を定義することによって、作業の前提条件、終了条件、制約、リソースなどの管理を行い、かつ、セルの並行化によって、タイムリーな開発も可能にしています。

伝統的なユーザ／ベンダの受発注、請負開発の中で、アジャイルに対応していくベースラインを確保している好例です。

ケース２：Web サービス領域のソフトウェア開発会社

「広義のアジャイルプロセス」を実践しつつ、タイムボックスマネジメントとセルの定義、さらに、セルと人的リソースの分離も行って、システムのライフサイクルを継続的に支援していく仕組みを確立しています。

Web システムや比較的小規模案件に特化しているとは言え、伝統的なプロセスへの配慮、組織化や仕組みの確立を目指しており、アジャイルプロセスによる不確実性への対応を実践している好例です。

ケース３：ユーザ企業内製のシステム部門

ユーザ企業の Web システムを構築・維持していく内製型の開発チームです。エンドユーザの利用アクセスや商品販売の誘導などの状況を、営業や業務部門との緊密な連携によって対応しています。

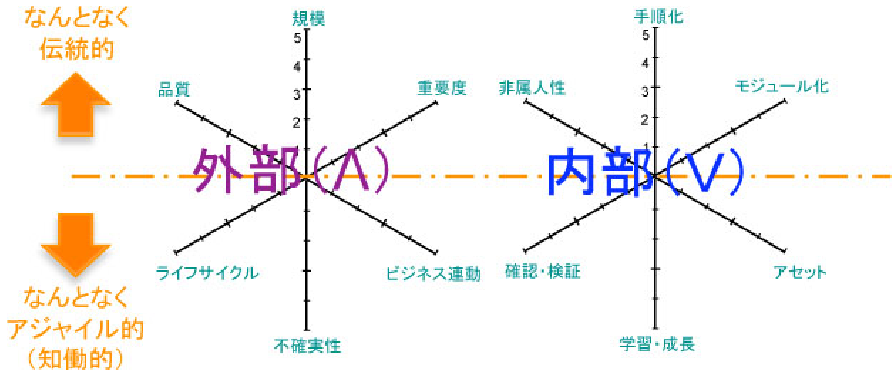
アジャイルプロセス、エゴレスシステム開発、さらには、ITIL や Cobit なども含め、自ら勉強し、検討し、議論し、実践する高度な研鑽によって、開発の質を維持している好例です。

ケース４：新規領域開拓型研究機関

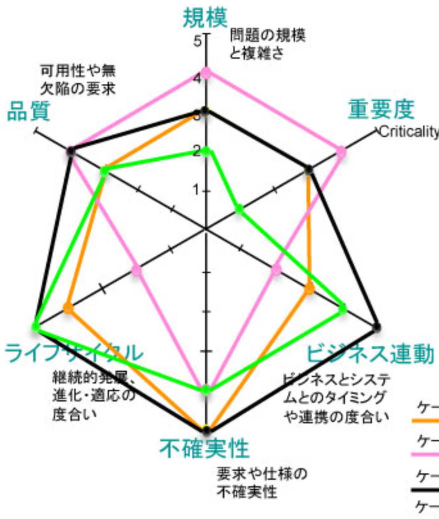
基礎研究から生まれた理論をソフトウェアのエンジンとして開発し、さらに、それを活用した社会的な仕組みや規範をデザインし、プロデュースも行っています。従来のソフト開発という範疇からは外れますが、知識をどのようにシステムに組込んでディスコースをデザインしていくかという知働的な考えの好例です。

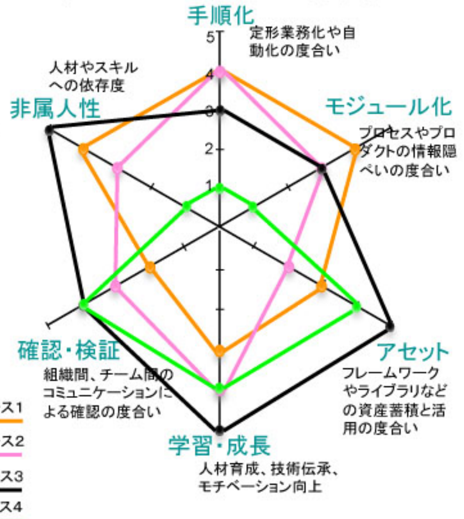
プロセスの特徴付け

これ等のケースを、強いて整理してみるとすれば、いくつかの軸によって特徴づけることができます。



左側のレーダチャートが、Λすなわち、実世界から見える特性です。もし、実世界が開発チームの外部であるということでしたら（組織境界に依存しますが）、Λ世界を「外部」と呼んでもかまわないでしょう。逆に、右側のレーダチャートはV、すなわち、開発チーム内部の特性です。





なんちゃってマニフェスト

本小論の最後に、筆者の主張を簡単なマニフェストの形で提示しておこうと思います。ここで示したものは、新しい「パラダイム」による、新しいものの見方です。伝統的な領域にとどまって、伝統的なものの見方をしていては、単なる＜改善＞に終わってしまいます。今、必要なのは＜改革＞なのです。

■工業的生産　より　実行可能な人工物のデザイン

ソフトウェアとは、実行可能な＜人工物（アーティファクト）＞です。決まりきったことを、手順化や自動化を行い、大量生産するという工業製品の＜製造＞とは異なります。ソフトウェアには製造はありません、創造的、知的活動としての＜デザイン＞が中心でなくてはなりません。デザインの対象となるソフトウェアが、一般の人工物と異なる点は、＜実行可能＞であることです。計算し、情報を処理し、実世界の現象に作用を及ぼす＜実行＞する人工物であることが本質です。

■コード　より　知識の織り込み

ソフトウェアづくりとは、知識をソフトウェアに織り込むことです。コードを書き、実行可能なコードをつくることも大切ですが、そのコードには、計算機世界の知識だけではなく、実世界や人間の知識を織り込んでやらなければ、価値はありません。織り込むべき知識がどこから来るのかということは、とても重要な観点です。クライアントからの要求や要件定義に従って、開発企業がソフトウェアを開発するという場合には、多くの知識は、クライアント側の奥深い業務領域から伝言ゲームのようにしてやってきます。多くの知識は、ΛVモデルのΛの領域に隠れていることでしょう。

■初期構築　より　ライフサイクルとフィードバック

ソフトウェアの意味は、その＜実行＞によって継続的に変化します。社会状況や世界観も変わりますし、ソフトウェアを実行させることが、利用者や関与者の認識を変えてしまうものです。従って、ソフトウェアは時間とともに変わる状況の中で生き続けなくてはなりません。伝統的な領域の概念で言えば、保守・運用をし続けるライフサイクルの観点がソフトウェアづくりの中心です。ソフトウェアは実世界に投入され、実世界の現象に影響を及ぼし、世界や人間の認識を変えてしまいます。その変化した世界や認識を、ソフトウェアづくりにフィードバックし続ける仕組みを考えていく必要があります。

■契約　より　社会的様相

二者間対峙（ユーザ／ベンダ、企業／雇用者など）の契約の改善の余地はまだまだあるでしょう。制度の問題というのは､最終的には整備しなくてはならないものです。ソフトウェアづくりに関わる組織や企業間の関係は、もっと複合的なものになっていくでしょう。織り込むべき知識の源泉、仕様化、実現、ライフサイクル維持など多くの活動が多くの組織とともに関わってきますし、組織間の関係もダイナミックに変化していきます。こういった広い意味でのソフトウェアを取巻く社会的な＜様相（texture）＞をデザインしていくことが重要です。また、全てが金銭的取引きに帰着できるわけでもありません。コミュニティ活動や地産地消的な事柄も、社会的様相として捉えていく必要があります。

執筆後記と参考資料

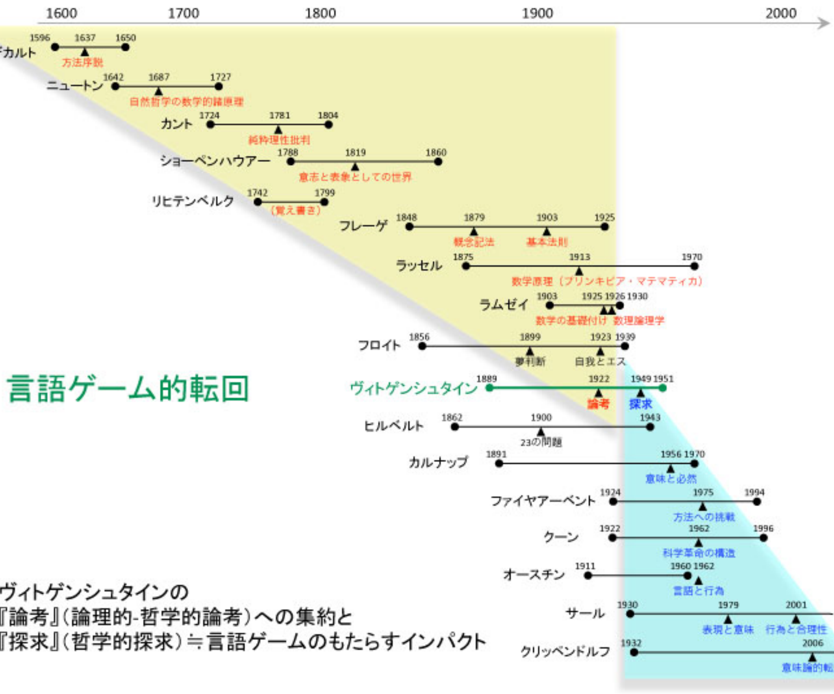
V字モデルは、1980年代初頭のライフサイクル論争の頃に、ウォータフォール型開発プロセスの欠陥を説明するために、設計とテストとの対応を整理するために提唱されたもので、現在でもSLCPの標準の中で、さまざまな開発プロセスを説明するために使われています。このモデルの本質は、要件定義や設計の正しさを確かめるテストという活動が＜実行＞するプログラム実体を前提としているということです。それ以上でも、それ以下でもないのですが、多くの誤解が生じていることも確かです。

アジャイルプロセスや知働化の検討では、あえて、V字モデルで扱う伝統的な領域とは一線を画し、その領域の言葉も使わずに、新しいパラダイムであるという立場を貫くようにしてきましたが、本小論では伝統的領域からの視点の差異を明確にするアプローチを採ってみることにしました。ソフトウェアやシステムが置かれている＜実世界＞や＜様相＞に目を向けるというのが、新しいパラダイムの第1歩です。ですから、V字の上のΛ字を描いてみました。「超上流」（この言葉は好きになれませんが）のもっと外側の世界あるよということですし、その世界もどんどん変化していくし、かつ、人によっても認識が異なる主観的なものです。

「パラダイム」という言葉が大げさな感じもするので、本小論の副題にあるように「意味論的転回」としてみました。第II章の「知働化への転回」では、クリッペンドルフのデザイン論を紹介していますが、そこで使われている用語です。

ソフトウェアエンジニアリングというのが、プログラムコードの記述に限らず、あらゆる活動を＜言語活動＞や＜言語現象＞であるとみなすという立場は、とても重要だと私は

考えています。無論、無意識や心理的な事項もあることは認めますが、最終的には言語として現れることになります。こういった哲学的な基礎付けをしたのは、20 世紀初頭に活躍した奇才、ヴィトゲンシュタインです。彼の後期哲学を『探求』とか『言語ゲーム』の哲学と称しますが、筆者はこの思想は、ソフトウェアの世界にも当てはまると考えています。昨年秋にクリッペンドルフの書籍を知りましたが、まさに我が意を得たりという感覚でした。



一言で私の主張を集約するならば、「ソフトウェアエンジニアリングの世界で言語ゲーム的転回を進めよう」ということです。新しい世界を描く努力なくして、未来は切り開けません。

〔参考図書〕

・黒崎宏 , 言語ゲーム一元論：後期ウィトゲンシュタインの帰結 , 勁草書房 ,1997.12
・黒崎宏 , 科学と人間：ウィトゲンシュタイン的アプローチ , 勁草書房 , 1977.10

・黒崎宏 , ウィトゲンシュタインの生涯と哲学 , 勁草書房 , 1980.10
・山本信 黒崎宏編 , ウィトゲンシュタイン小辞典 , 大修館書店 ,1987.7
・橋爪大三郎 , はじめての言語ゲーム , 講談社現代新書 , 2009.7
・クラウス・クリッペンドルフ , 意味論的転回：デザインの新しい基礎理論 ,
SIB アクセス発行 , 2009.4.1

〔筆者の知働化関連の書き物〕

・アジャイル・ソフトウェア・セル生産：人月から価値駆動へ（大槻繁 , 濱勝巳）,
PM Conference 2008, 2008.8.1
・人働説から知働説へ（対談：知働化研究会始動（山田正樹−大槻繁））, 2009.7.15
・知働化への接近 , 知働化研究会第 2 回会合 , 2009.10.15
・クリッペンドルフの『意味論的転回』読書感想 , 2009.10.15
・ビジネス駆動の先進的な IT システム・ソフトウェアの世界観：
知働化：不確実性への対応から進化・適応プロセスによるソフトウェアづくり ,
IPA Forum 2009 / ソフトウェア・エンジニアリングセッション , 2009.10.29
・知働化の世界観：
不確実性への対応から進化・適応プロセスによるソフトウェアづくり , 2009.11.3
・１０８の質問に対する解答（デカルト vs ヴィトゲンシュタイン）, 2009.11.4
・言語ゲーム的転回の年表 , 知働化研究会第 3 回会合 , 2009.12.7

〔プロフィール〕

大槻繁（おおつき・しげる）

日立製作所にてソフトウェアエンジニアリングの研究・開発に従事。2004 年よりコンサルタント会社一(いち)副社長。IT システム関連の調達･開発プロジェクトの見積り評価、診断・改善のコンサルティングを行うかたわら、コストモデルや経済モデルの研究・開発を進めている。

IPA/SEC 定量的マネジメント部会委員、同価値指向マネジメント WG リーダ、非ウォータフォール研究会委員などを歴任、JEITA ソフトウェアエンジニアリング技術分科会委員、アジャイルプロセス協議会フェロー、同協議会 知働化研究会 運営リーダなどを楽しむ。

著作に『ソフトウェア開発は なぜ難しいのか：人月の神話を超えて』（技評 SE 選書 , 2009 年 11 月）など多数

論説

〔知働化研究会誌創刊第１号〕
知働化研究会
Executable Knowledge and Texture Laboratory

クラウドコンピューティングにおけるアーキテクチャの進化の方向性

頭脳のアーキテクチャからのアプローチ

萩原 正義（はぎわら まさよし）
マイクロソフト株式会社

概要…… 133
1. クラウドのアーキテクチャの進化の方向性 …… 133
2. 頭脳のアーキテクチャの特徴 …… 134
2.1 入力（観測）部分のモデル…… 136
2.2 判断、意思決定部分のモデル…… 139
2.3 記憶部分のモデル…… 142
2.4 出力（運動）部分のモデル…… 145
3. まとめ …… 147
参考文献…… 148

Copyright 2010, HAGIWARA Masayoshi, All rights reserved.

概要

クラウドコンピューティングは分散、並列、非同期で動作するスケールアウトのアーキテクチャを原則とする。同様に人間の頭脳は分散、並列で動作するコンピューティングのアーキテクチャでモデル化できる。このクラウドと頭脳のアーキテクチャ上の類似性から、クラウドのアプリケーション、データアーキテクチャの進化の方向性を予見し、現在のクラウドの各要素技術の位置づけと限界を明確化する。

1. クラウドのアーキテクチャの進化の方向性

現在クラウドコンピューティングは、仮想化、P2P に基づく可用性やプロブジョニングなどのリソース管理のインフラの技術、NoSQL と呼ばれる非 RDB、分散キューイング、分散キャッシュ、分散バッチ処理、リアルタイムのイベント駆動型の実行環境、データ分析、機械学習、ESB（Enterprise Service Bus）による分散サービス間連携、統合認証や認可のプロトコルや機構などの PaaS レベルのミドルウェアが提供される。また、これらの開発の支援では、アーキテクチャスタイルとデザインパターン、プログラミングモデル、分析設計法と関連する計算モデルやパラダイム、DSL（Domain Specific Language）による抽象化したモデリングを利用する。これらの複合化した要素技術を俯瞰的に見るための参照アーキテクチャの候補として、人間の頭脳を考え、両者の類似性からクラウドのアーキテクチャの進化の方向性を見ていくことが可能である。ここで、両者の類似性から進化の方向性を導けることの根拠は以下の 2 点である。

* クラウドのアプリケーションとその他の関連するソフトウェアは人間が開発し、人間が利用する。したがって、クラウドのアーキテクチャと人間の思考形態との親和性が技術の理解の容易性につながり、クラウドの広範な要素技術の複合体に対する俯瞰的な理解を伴った進化の基盤となる。クラウドのサービスのユーザエクスペリエンスが人間の思考に従うことでより競争力の高いサービスとなり、結果としてアーキテクチャが生き残る可能性が高くなる。
* さらにより根源的な理由として、意味の認知、行動の妥当性の検証、知識の形式化といったクラウドや頭脳という物理的な場所に依存しない知に関する振る舞いの制約条件、それを支援するための記憶管理と操作、が共通していること。

2. 頭脳のアーキテクチャの特徴

すべてのシステムの振る舞いは並列実行部分と逐次実行部分に分けられる。この2つを明確に関数型パラダイムの基本操作（プリミティブ）に定義し分離している例がMapReduceである（Dean 2004)。MapReduceは、クラウドの分散並列処理の機構を利用し、並列実行をMap、逐次実行部分をReduceで行うバッチ処理のためのプログラミングモデルである。一般のバッチ処理はMapとReduceを多段階に連結したデータフローで定義可能である。またRDBに対するSQL構文は、FROM句の入力データに対してWHERE句の選択条件、ProjectionをMapで、Aggregation、Group by、HavingをReduceで、joinはMapとReduceで行うことで同等の実行結果が得られる。

人間の頭脳はMapReduceと同様に並列実行部分と逐次実行部分に分けられる。ここでは、人間の頭脳を単純化したアーキテクチャでモデル化し、その特徴的な要素だけを抽出してクラウドのアーキテクチャに採用しようと考える。人間の頭脳がIT技術の参照アーキテクチャとして実現可能であること、従来技術と異なる特長を持つことを以下で示す。

このアーキテクチャのモデルにおいては、入出力の部分と、判断と意思決定の実行部分が重要である。

* 入出力の部分のモデル：入出力は並列実行する。入力では感覚器官による観測、出力では運動系による各種の振る舞いを実行する。
* 判断と意思決定の部分のモデル：観測とほぼ同時に記憶（知識）を参照し、行動の目的に従っているかの適切さの判断を行い、知識の修正および行動（運動）への指示を実行する。

図1は頭脳のアーキテクチャのモデルである。入力部分、出力部分、行動目的の判断部分、記憶部分の4つのサブシステムを中央の意思決定部分が統合する構造となっている。入力部分、出力部分、行動目的の判断部分、記憶部分はそれぞれが並列実行され、中央の意思決定部分が逐次実行される。

これをMapReduceに対応づければ、入力部分、出力部分、行動目的の判断部分、記憶部分がMap、中央の意思決定部分がReduceの分担となる。

このアーキテクチャは汎用並列グラフ変換フレームワークPregel（Malewicz 2009）に見られるような、逐次処理の実行部のモデル、それらの複数のインスタンスを並列に実行するためのスレッドやマシンのリソース管理、通信の最適化や信頼性のための監視機構の仕組みに比べて一般的である。Pregelはグラフデータモデルのトラバースの逐次処理

と並列実行可能部分を分離し、実行の最適化のためのリソース管理を設計のゴールにおいているため、アーキテクチャとその実行の機構の汎用性は低くなる。しかし、逐次実行と並列実行部分の分離の原則は同様に守られている。

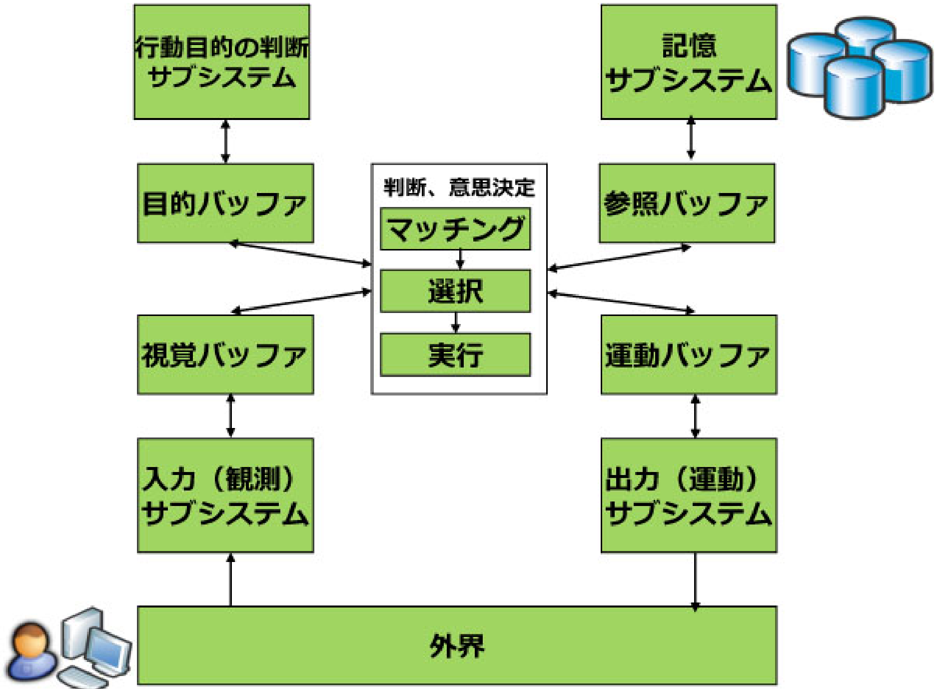


図 1 頭脳のアーキテクチャ（Anderson 2004）

以下では、この頭脳のアーキテクチャの各部分のモデルを説明し、クラウドのアーキテクチャやそのアプリケーション開発における分析設計法との関連性を説明する。なお、実体、観測、認知、表現の 4 つが分離されていることを前提とする。モデルは実体そのものの定義ではなく、観測し、意味が認知された結果を関心の分離を行い、抽象化を通じて表現した定義である。したがって、モデルは恣意的であり、客観的表現には限界を持つと考える。

2.1 入力（観測）部分のモデル

観測対象のスコープ

現在のソフトウェア開発におけるモデリングでは事前に概念モデルが開発対象の問題領域を定義する。図２は概念モデルの典型例であり、この図で表現される問題領域がソフトウェア開発の対象に関連した実体世界を表現し、分析設計の対象を決定する。この範囲はソフトウェアの個別の機能要求（ユースケース）の１つずつの表現の総和ではなく、機能要求を実現する元となった目的や解決すべき問題、より抽象的にはソフトウェア開発の戦略的な価値を表現する。図３のユースケースはそれぞれがシステムの関係者とその関係者と、システムの間の相互作用を定義することで、機能要求を実現するシステム化の範囲を示す。ユースケースもソフトウェア開発に先立ちステークホルダー間で合意の上定義される。

ここで、これまでの概念モデルやユースケース定義がシステム存在前に固定的にあるいは繰り返し型開発で漸進的に定義されるのに対して、頭脳の観測対象のスコープは、現在の関心に応じてリアルタイムに対象が定義され、関心の対象だけをフィルタリングし、コンテキスト（状況）依存である。それにより、膨大な観測対象から必要な情報だけを入力し、必要に応じて抽象化して効率的に情報処理を行う。頭脳は基本的なアーキテクチャを維持しながら、処理すべき機能の要求、要求が存在する問題領域の範囲を適応的に変更する。ここがソフトウェア開発のモデリングと異なる点である。従来のソフトウェア開発の問題領域（そこに存在する制約）や機能要求を固定してしまう前提は、すでに人間の頭脳の持つ観測スコープの動的な変更に対応した特性に制約を加えてしまう。また、コンテキストや状況への変化への適応性にも不必要な制約を加える点で、クラウドにおいてはこれらの既存のモデルの制約を緩和するモデリング言語を考えなくてはならない。

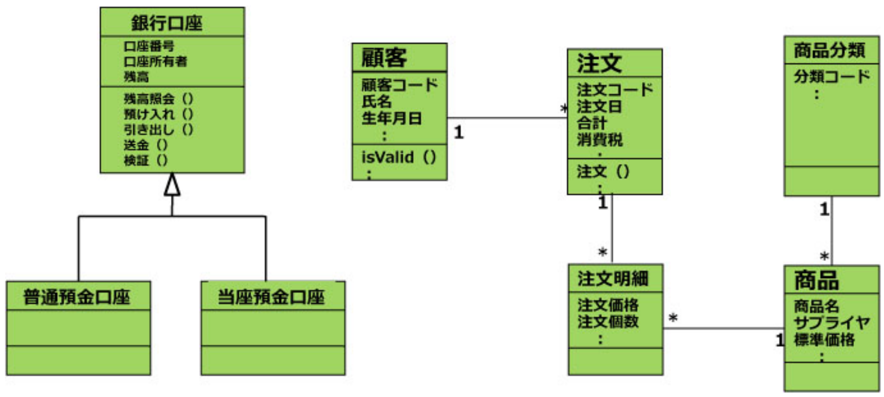


図２ 概念モデルのスコープ

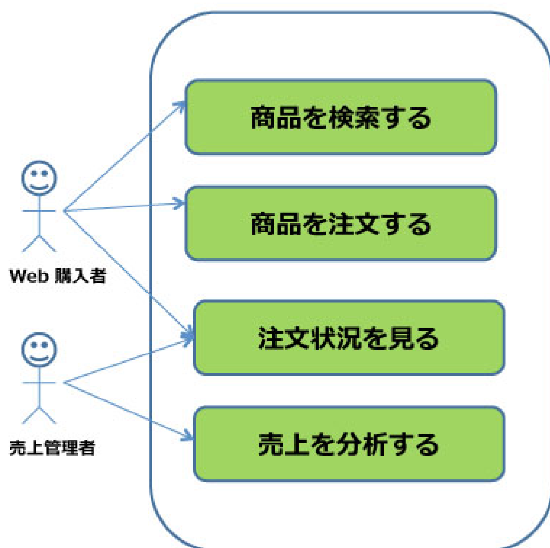


図 3 ユースケースのスコープ

活動状態の認識スコープ

ソフトウェアの振る舞いを定義する状態遷移やアクティビティのモデルは、それぞれの状態、アクティビティは特定時点でシステムが定常状態にある条件でのスナップショットを表現する。すなわち、状態、アクティビティともに時間を一時停止し、意味的に一貫性を持つと認識される範囲を定義している。この前提に基づいてモデル表現できるには、実行中のソフトウェアの状態の変化が離散的であっても連続的であっても各時点で意味の一貫性を持つことが保証しなくてはならない。したがって、振る舞いとその変化は、一貫性を持って意味づけされた範囲（集合）とその関係づけで表現される。集合は代数的、位相的に表現されてもよい。

人間の頭脳においても外部の観測情報は、ある関心の下で連続的に入力されて、一貫性を持って意味づけられた集合で認知され、それに基づいて判断、意思決定がなされる。一貫性がなく意味がとらえられない観測対象は認知、判断が不可能である。

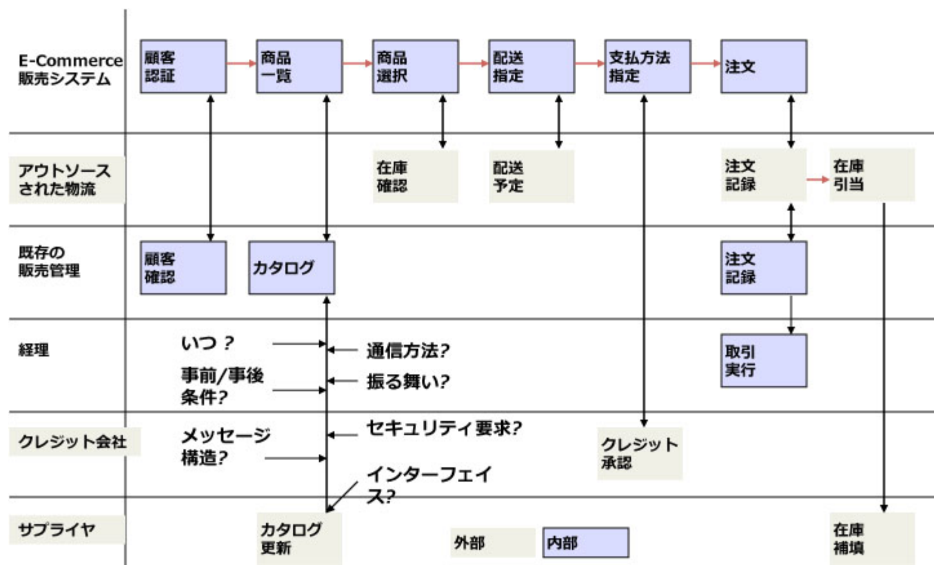


図 4 状態遷移における状態のスコープ

観測状態のイベント化

頭脳による観測対象は、意味的に一貫性を持つと認識される範囲（集合）としてとらえられ、判断、意思決定部分に入力される。これをアーキテクチャのモデルとして表現するためには、観測のデータ表現および判断、意思決定部分への入力の機構を決定する必要がある。

ここで、観測のデータ表現には、タプル（リスト）または述語を使うことが可能である。タプルは関数型パラダイムとの相性がよく、十分に汎用的である。例えば、Web サービスのヘッダーとボディのメッセージ定義はタプルで表現可能で、センサーネットワークから Web サービスにより観測データをクラウドに入力する場合のデータ表現に適切である。これと同様に、頭脳による観測の表現にも適用可能と考える。より人間の思考に近い表現としては述語による定義の採用が可能である。述語は命題を真とする集合を定義するので、観測段階の対象の定義としてはやや判断を伴う点で適切さを欠くが、判断に関係なく事実表現と考えることも可能である。述語は他の述語との関係を論理的に扱うことが可能であるので、タプルに比べると抽象的かつ論理的操作で優位性を持つ。

観測データの表現をタプルあるいは述語で定義した後、それをリアルタイムに判断、意思決定するためにはイベント生成を考えるのが自然である。イベントを受信した際に関心

によってフィルタリングするためのルールが適用される。イベントはクラウドの可用性を保証する非同期処理の原則と親和性がある。現在、クラウドにはイベント駆動型アーキテクチャでルールを適用するか（Boom)、観測データ受信をパターンマッチングで非同期処理するかの２つの実現法の選択肢が存在する。前者は論理型、後者は関数型の実現法である。図５でタプルと述語によるデータ定義の例を示す。

タプルによる定義（Haskell）：
([0,1,2,3],"abcd")
２つの要素 $a \in A$, $b \in B$ の順序づけられた組 $(a, b) \in A \times B$ 、
この順序対を元とする一般の n 組 $(n \geqq 2)$ を、
例えば $(a1, a2, a3, \ldots, an) = (\ldots((a1, a2), a3), \ldots, an)$

述語による定義（Datalog）：
employee-jones(x) :- manager(X, “Jones”)

図５ 観測状態のタプル、述語による定義

2.2 判断、意思決定部分のモデル

観測データの記憶

観測データは判断、意思決定のために一時的に（短期に）記憶しなければならない。そのためにはタプルや述語で表現される事実をできるだけ高速に記憶する書き込み優先のデータベースが求められる。観測データは時系列に参照するため、行（ロウ）指向のデータモデルが有効となる。頭脳の短期記憶も同様な機構となると思われる。

Partition Key Uid	Row Key 日時	Property 3 ユーザ名	Property 4 注文コード	….	Property N 商品名
100211	2009/3/3	萩原	11233	….	46型液晶テレビ
100211	2009/4/26	萩原	11255		ブルーレイプレイヤー
100212	2009/5/11	野村	11557		電子書籍
100214	2010/1/1	福井	11582		携帯電話
100214	2010/1/1	福井	11671		メモリカード

図６ 行指向データモデルに基づく書き込み優先のデータベース（Windows Azure Storage）

意味の認識スコープとその保護

観測データの意味を認識するためには、複数の観測データ間、記憶や知識にある情報との関係性を分析しなければならない。一般に情報の意味はこうしたデータ間の関係を構築することで定義される。例えば、概念モデルは、概念を構成する属性（データ項目）と、他の概念との関係から定義される。ここでは、他の概念との関係性から概念に必要とされる属性が決定する。商品概念モデルが製造年月日の属性を持つべきかは、商品モデルと製造モデルの関係があって決定される。概念モデルはモデル化対象に適切な概念間の関係を固定的に定義し、ソフトウェア開発に先立ち意味を固定化するが、頭脳では、観測データを関心や状況に応じて意味解釈するので、リアルタイムにデータ間の関係を動的に変更しつつ、観測データの属性データをフィルタリングして定義を決定している。この作業の間は、他の観測データを参照しながらも、データ関係づけの決定に他の無関係な動作が介在しないようにデータ保護がなされていると思われる。類似例として、リアルタイムに変化するデータを参照し、データ更新を行うため、データの変化の頻度とデータ更新のためのロックの期間との間に制約を持つ場合の設計の問題がある。これらはデータ同時性制御の高度な例と見ることができる。

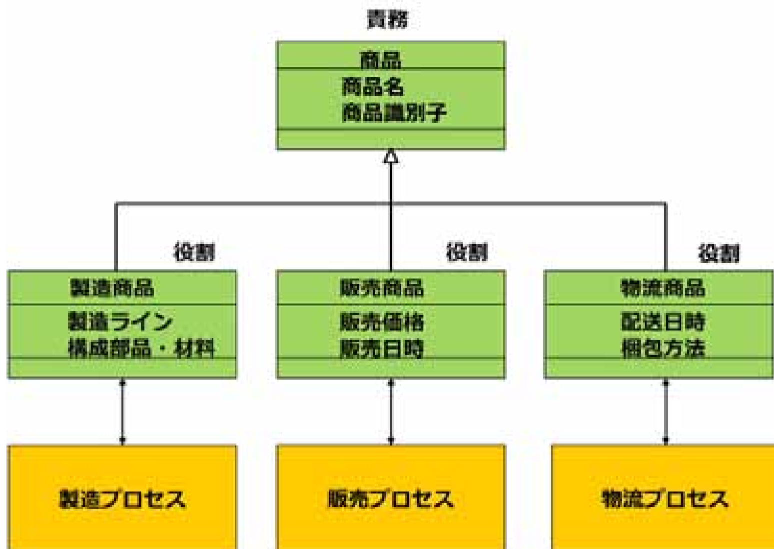


図 7 意味のまとまりの関係を構築

認識結果の判断と意思決定

観測データの意味が認識されると、その結果の分析が実行される。観測データの意味の認識では主に観測データの振る舞いに注目する。例えば、接近する自動車の観測に対して、自動車とは何かを見るのではなく、その運動に注目する。そして、その自動車が知識から経験的に交通事故を引き起こす危険なもの、その原因となるシナリオなどを分析し、結果を想定する。ここでは観測データの意味の認識と同時に判断もほぼ同時に行っている。その後、交通事故を起こさずにすむ、回避のための行動を意思決定する。回避の意思決定では、頭脳は行動目的の適切さの判断を伴う。これは IT のアーキテクチャでは一般には見られない機構であり、当然これまでのクラウドのアーキテクチャにも存在しない。行動目的の適切さの判断とは、図 1 の頭脳のアーキテクチャに見られるサブシステムで実行され、回避行動が行動として適切さを持つかどうかを判断する。IT では振る舞いは定義された通りの実行で終わり、それが適切であったかどうかの判断は伴わない。しかし、頭脳はこの判断により、経験から知識の構築が適切に行われ、行動目的の適切さ自身（優先度など）の変更が行われる。これらの判断、意思決定、知識の追加や行動目的の変更の実行も他の無関係な動作が介在しないような保護がなされていると思われる。回避行動のためには出力（運動）部分にアクションを要求する。

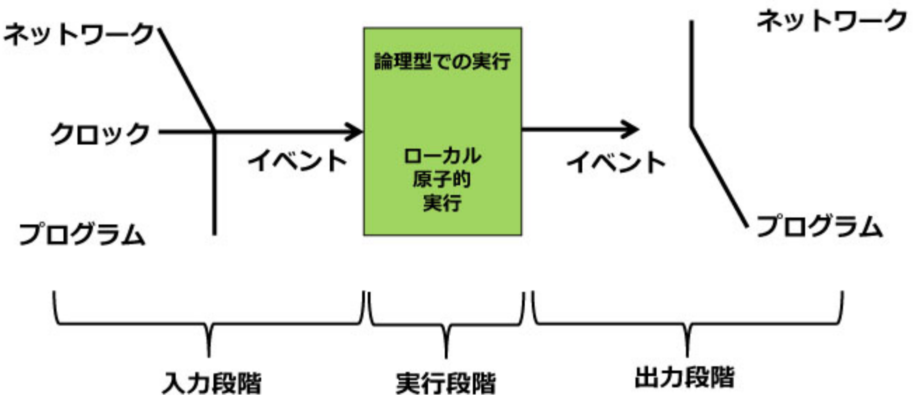


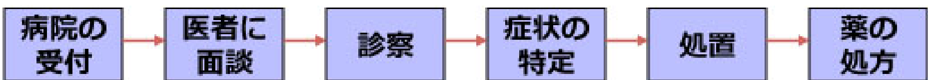
図 8 意味の認識、判断、意思決定を実行するリアルタイムアーキテクチャのモデル（Boom）

2.3 記憶部分のモデル

記憶の正規化

観測データの高速な記憶と認識や判断におけるデータ参照のためには、行指向のデータモデルが有効である。これに対して、知識として体系化される長期に記憶されるデータモデルは、時系列、連想、意味の重要性順、関心の分離による抽出のしやすさなどからグラフ構造のデータモデルが有効と思われる。この長期の記憶では、短期の記憶を総合的に分析し、過去の知識を参照しながら修正、体系化を繰り返し行っていると思われる。これらの修正や体系化は、判断と意思決定部分からの要求があって行われ受動的である。

長期の記憶における概念は、その意味を定義するために他の概念との関係をグラフで持ち、概念自体がプロセスにより定義されると考えられる。これが本来の概念の正規表現である。IT における概念モデルは構文的にはクラス構造で表現されるが、長期の記憶では概念はプロセス、つまり、振る舞い（グラフのトラバース）で定義される。例えば、交通事故という概念は、自転車、車、鉄道、飛行機に依存せず、想定外の衝突、動作、不具合などで生じる損傷とそれに伴う出来事を総称していて、それらが一連の動作としてとらえられている。そこには多様な原因、関係者、被害の結果が存在するが、それらがシナリオの一部として定義される汎用性のあるプロセス定義となっている。経験的にこのプロセス定義の解釈の範囲を拡大し、修正して概念が形成されてきたと見ることができる。



治療の概念を表すプロセス

図 9 プロセスによる概念の形成

知識化とその操作

知識の構築の過程を IT の技術で考えてみると、概念モデルの定義と永続化のアーキテクチャの面での特徴を導くことが可能である。まず、概念モデルでは、モデル間の関係性の動的な構築の機構、関係性による意味の定義の方法、その意味から価値の創造を実現可能なことが求められる。これまでの概念モデルは単なる構造定義であり、これらの要求を実現するには役不足で、より表現能力の高いモデルが必要となるであろう。その意味では概念モデルのプロセス定義化は表現能力の高さと、定義の柔軟な変更が可能となるので有

効な選択肢と考える。また、モデルから創造される価値についても、これまでの IT では価値の具体的な表現を要求から落とす方法がなかったので、関係性での表現をバリューチェインと考えていくことができると思われる。

知識の永続化のアーキテクチャは、観測データの短期記憶からの、判断や意思決定の要求に応じた知識の構築の機構であるので、キャッシュとグラフデータベース間の非同期の連携と見ることができる。キャッシュは短期記憶と判断と意思決定のための実行環境を提供するので、リアルタイムイベント駆動型アーキテクチャと、分散並列バッチ処理で実現する（図 11）。そこに処理に応じて適切な中間データ形式を採用する。また、判断や意思決定で要求される適切なデータ形式での高速な記憶の参照を想定すると、Command（書き込み）と Query（参照）を分離する CQRS（Command Query Responsibility Segregation）のアーキテクチャススタイルを採用してもよい。CQRS は従来の企業情報システムのアプリケーションアーキテクチャのレイヤ構造を大きく変える。これらはいずれも現在のクラウドの PaaS アーキテクチャの進化の方向性を考慮している。

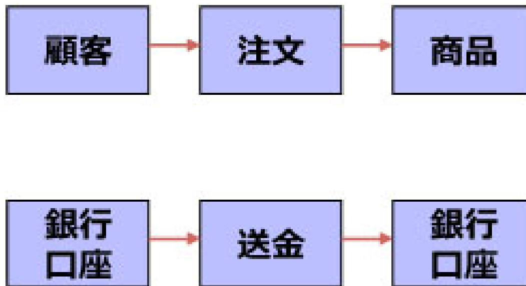


図 9 関係性が意味を定義し、価値を創造する例

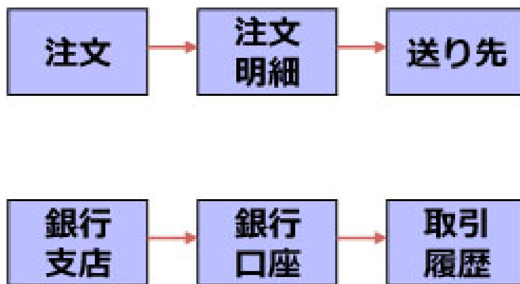


図 10 既存のソフトウェア開発で関係性が価値を創造していない例

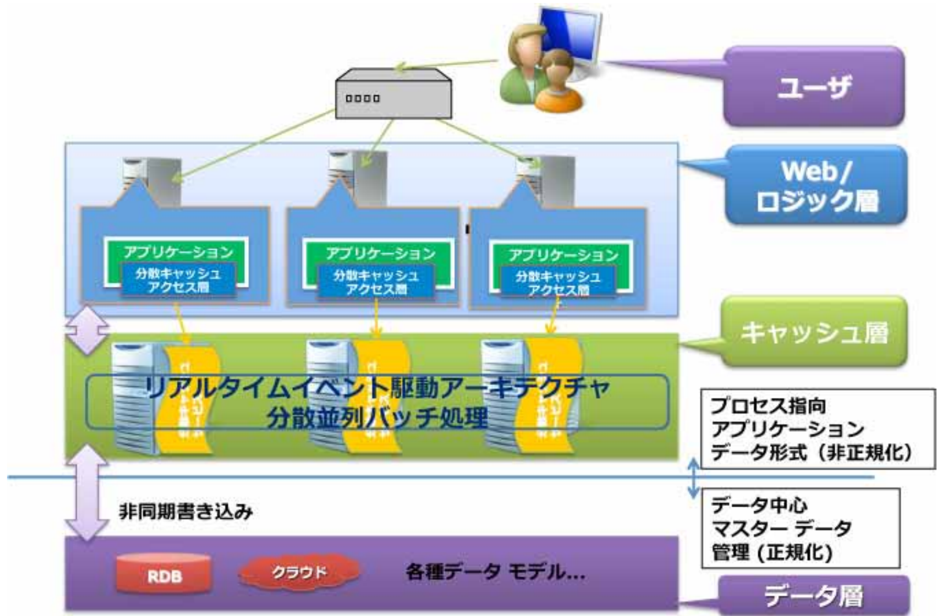


図 11 PaaS における分散キャッシュ上で動作するリアルタイムイベント駆動型アーキテクチャと、分散並列バッチ処理（図 8 との類似性）

大規模分散グラフ

長期の記憶のためのグラフデータベースは、書き込み、参照が並列実行される。これをクラウド上で実現するには、グラフ全体を複数のサブグラフに分割して管理し、それぞれサブグラフの操作が並列実行されるようにサーバーへのデータの最適配置と、検索のトラバース、参照のための到達性やホップ数の最適化、データ冗長化のための複製、隣接検索、ランク、近似解などのサポートを行う必要がある。また、書き込みでの順序一貫性、トランザクション機能なども考慮が必要である。

図 11 のリアルタイムイベント駆動型アーキテクチャと、分散並列バッチ処理の機構の上に、先の Pregel に見られるように、グラフデータモデルを使ってデータ分析、機械学習などを実現し、その状態を定期的にグラフデータベースにチェックポイントとして書き込みことで可用性の確保が可能である。何等かの障害時はチェックポイントから状態を復元して再実行することで同一の結果を得る（図 12）。

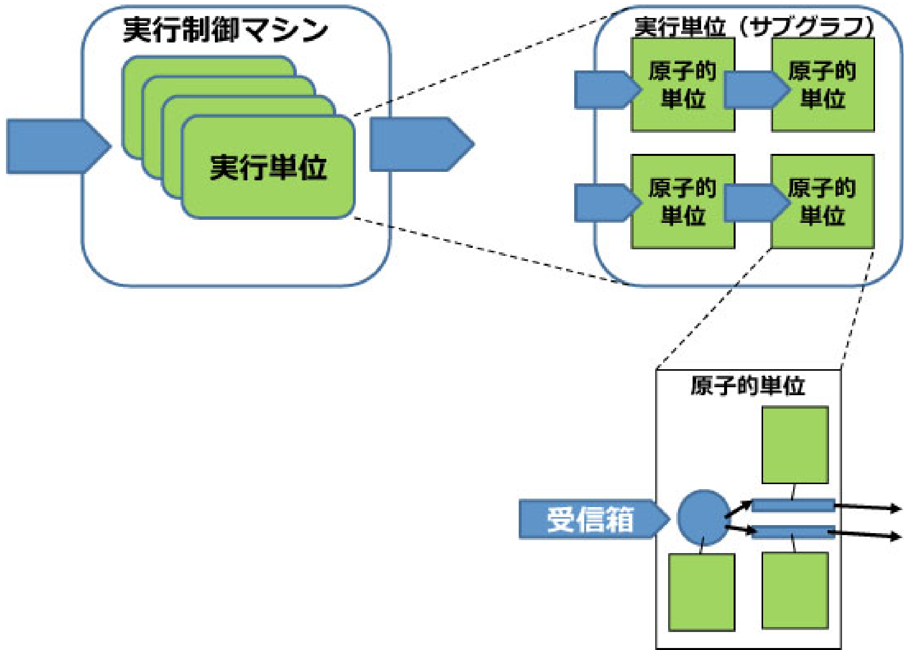


図 12 大規模分散グラフのアーキテクチャ（図 11 上で動作する）
データ分析、機械学習などに応用（Malewicz 2009）

2.4 出力（運動）部分のモデル

出力形式の選択

判断や意思決定で要求される記憶の適切なデータ形式を高速に参照し、その結果を出力形式として言葉で表現する。この出力形式化の過程も他の無関係な動作が介在しないような保護がなされていると思われる。この場合、知識や内在する概念に付随する関係性は、出力形式化で省略されてしまうので、言語表現の意味が本来の意味と異なる解釈として受け取られる可能性がある。これは内在する暗黙知の形式化の限界、表現の客観性の限界と見ることができる。たとえるなら、出力形式に存在する意味をあらわす識別子（意味のスキーマを参照する URI など）が外部に形式化できず、その意味の類似性のマッチングから、それぞれの頭脳毎に異なる定義を参照してしまうためである。

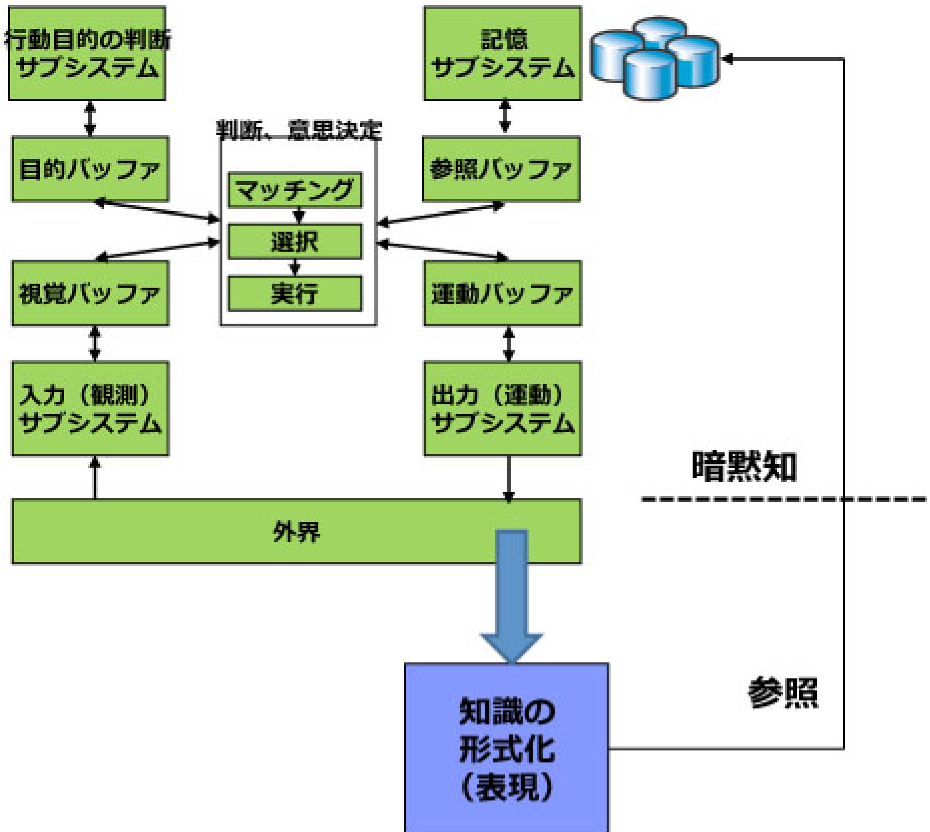


図 13 知識の形式化、表現の客観性の限界

行動

意思決定の結果の出力形式化は言葉による表現を含めて、運動系への指示と見なすことができる。例えば、顔や手による表現などである。これらの表現、具体的な行動は、その運動系へのイベント起動とタプルによるデータ定義で要求するモデルでとらえることができる（図 5、8）。

3. まとめ

現在のクラウドのアプリケーション、データアーキテクチャはまだ発展途上である。したがって、問題ドメインに対して有効なアーキテクチャの決定は既存の経験に頼る他はない。しかし、一方では、並列実行とデータの転送コストのトレードオフを考慮し並列実行の処理のまとまりを決めること、並列実行と逐次実行を分離すること、一貫性の単位を切り出し物理的にまとめること、データ要求に応じて複数のデータモデルを組み合わせること、などの物理的制約に基づく原則と、適切な抽象化を導入することによる設計、実行管理などの論理的制約に基づく原則が明らかになってきている。

頭脳のアーキテクチャのモデル化では、完全な機構のモデル化の解明はなされていないまでも、現状のモデルをクラウドの参照アーキテクチャと見たとき、参考となる多くのヒントを含んでいる。以下はこの参照アーキテクチャに特徴的な原則を示している。

表 1

原則	原則の存在背景、特徴、有効性
並列実行と逐次実行の組み合わせ	振る舞いの観測と機能要求からのモデル化。汎用的なアーキテクチャとプログラミングモデルの構築可能性を示唆。逐次実行のボトルネック。学習による効率化の改善
観測対象のスコープ	関心に合わせた入力。意味の認識、抽象化との組み合わせ。データ処理のスケーラビリティと変化への追随
意味の認識	既存の経験、知識との概念間の関係による意味の認識。グラフデータモデル。判断の同時性
意味の一貫性の制約	観測データの記憶、意味の認識、知識の参照、行動目的の適正さの判断、知識化、行動の要求に一貫性を持たせる
行動目的の適切さ	知識の価値の妥当性、振る舞いの正当性の検証。高階論理とその動的な変更によるモデル化など。計算効率のための予測による結果指向、過程に対する論理的妥当性の提示

これらの原則に基づき、適切なアーキテクチャスタイルが選択されるだろうが、まだ具体的な選択に関しては流動的である。実際、原則に従う場合でも、既存の実行環境やプログラミングモデル、設計手法を前提とする場合と、それらにとらわれない破壊的イノベーションとなりうる技術による場合があり、後者に関しては予見不可能である。いずれにしても状況の変化に効率的な方法でリアルタイムに対応可能な方法が選択可能であることが求められる。

参考文献

Anderson 2004: John R. Anderson, Daniel Bothell, An Integrated Theory of the Mind, Psychorogical Review 2004 vol. 111 No. 4 1036-1060

Dean 2004: Jeffrey Dean and Sanjay Ghemawat, MapReduce: Simplified Data Processing on Large Clusters, OSDI'04: Sixth Symposium on Operating System Design and Implementation, December, 2004.

Malewicz 2009: Grzegorz Malewicz et el, Pregel: a system for large-scale graph processing, Annual ACM Symposium on Principles of Distributed Computing, 2009

Welsh 2001: Matt Welsh, David Culler, and Eric Brewer, SEDA An Architecture for WellConditioned, Scalable Internet Services, In Proceedings of the Eighteenth Symposium on Operating Systems Principles, Oct 2001 (2001)

Boom: http://boom.cs.berkeley.edu/

CQRS: http://www.infoq.com/presentations/Command-Query-Responsibility-Segregation http://elegantcode.com/2009/11/11/cqrs-la-greg-young/

寄稿

アジャイル開発プロセスと契約　高橋雅宏

契約はアジャイル開発プロセスに追いついたか

寄稿の章では、知働化研究会の外部の有識者からの論文を掲載しています。

『アジャイル開発プロセスと契約』は、アジャイルプロセスの実務面で必ず必要になる契約の問題を、基礎知識を含めて詳説しています。著者の高橋雅宏氏は、アジャイルプロセス協議会の見積・契約 WG のコアメンバとして活躍されている方で、大手 IT 企業の調達部門の専門家です。知働化研究会誌創刊号発刊に向けて、日頃の想いを書き下ろしていただきました。

寄稿

〔知働化研究会誌創刊第１号〕
知働化研究会
Executable Knowledge and Texture Laboratory

アジャイル開発プロセスと契約

契約はアジャイル開発プロセスに追いついたか

高橋 雅宏（たかはし まさひろ）

アジャイル開発プロセス協議会
見積・契約ワーキンググループ
Takahashi.masah@nifty.com

概要…………………………………………………………………… 152
話題にするソフトウェアの位置づけ………………………………… 152
ソフトウェア開発環境の変化……………………………………… 153
開発されるソフトウェアの変化…………………………………… 157
契約や契約書の変化と進化………………………………………… 159
これからの契約書について………………………………………… 163

Copyright 2010, TAKAHASHI Masahiro, All rights reserved.

概要

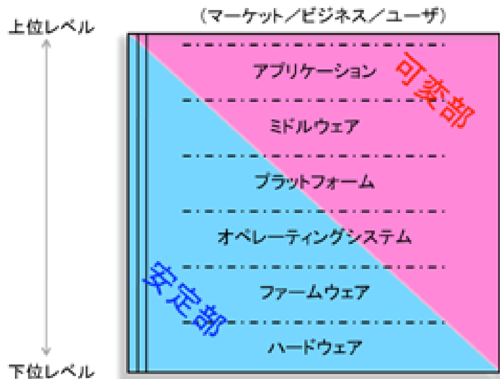
ソフトウェアは、ハードウェアとユーザとの間を埋めて、ユーザがハードウェアを使いやすくするための環境を提供する存在、と捉えることができると思います。一方、ユーザの先には、ビジネスの成功という目的が広がっています。ユーザは、この目的を果たすために、ハードウェアのスピード、ソフトウェアの柔軟性をツールとして使っている、と言うことができると思います。

このソフトウェアを「開発するための環境」というものは、ここ３０年ほどで、著しく進化してきました。ところが､ソフトウェア開発するための取引に使われている「契約書」は、この開発環境の変化とともに成長なり、進化なりをしてきたでしょうか？

ここでは、このことについて、自分が開発者だった頃の過去を振り返っていきたいと思います。そして、これからの契約書について考えてみたいと思います。

話題にするソフトウェアの位置づけ

図1 ソフトウェアの位置づけ

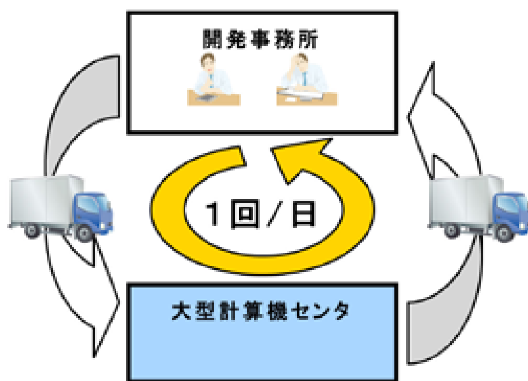


我々をとりまくソフトウェアには、多種多様なものが存在します。そこで、まず、ここで話題にするソフトウェアの位置づけをはっきりさせておきたいと思います｡図１にハードウェアから市場(マーケット）までのソフトウェアに関連する関わるものの関係を示しました。

ベース部分に位置するのがハードウェアです。ユーザとハードウェアの間を埋めるものがここで扱おうとしているソフトウェアです。ただ、このソフトウェアは、更に複数の階層に分けることができます。ハードウェアに近い下の階層では、仕様が安定していますので、仕様変更の自由度が小さいと言えます。一方、ユーザが直接操作するような業務プログラムとかアプリケーションソフトウェアと言われる階層では、仕様変更の自由度が大きくなっていると思います。通常、アジャイル開発プロセスでは、ユーザに近い業務プログラムの開発について議論されていると思います。ここでは、その部分のソフトウェアの開発に関する部分に的を絞って書いていこうと思います。

ソフトウェア開発環境の変化

図２　３０年以上前のソフトウェア開発環境



私がソフトウェア開発者だった３０年以上前の開発環境（図２参照）を振り返ってみると、開発設備としては、大型計算機、磁気テープ、パンチカード、ラインプリンタ、このようなものを使って開発していました。当時は、開発設備そのものが、すべて高価なものでしたので、大勢の開発者が大型計算機を共有して使用していました。

開発設備としての大型計算機は、物理的にも大きなものでしたので、開発者が作業をする事務所に置くわけにはいきません。必然的に開発する事務所からは、大型計算機が離れた場所にあることがほとんどでした。しかも、当時は、今で言うクライアントサーバーシステムのような構成はできませんでしたので、大型計算機を利用する場合は、大型計算センターと開発者の事務所を往復するシャトルバスに乗って直接出向いて行くか、事務所と大型計算機センタとを往復するコンテナを利用する方法しかありませんでした。コンテナを使用した大型計算機センタの利用方法は、こうです。大型計算機での実行コマンドが書かれた「パンチカード」をコンテナ形のトレーに入れておくと、シャトルトラックで大型計算機センタまで運ばれ、そこに待機しているオペレータに届けられます。そのオペレータによって、トレーに入れられた「パンチカード」が取り出され、１件１件順番に実行されます。このようにして、大型計算機で実行させる方法をとっていました。

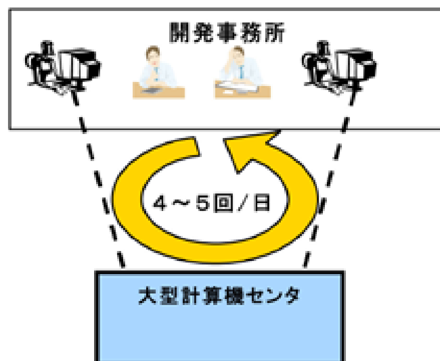
そして、大型計算機での実行結果は、再びシャトルトラックに乗って開発事務所に返ってきます。このような方法で開発すると、コンパイルの結果を確認するだけでも、丸１日必要になります。コンパイルエラーなどで、コンパイルそのものが失敗すると、さらに１日単位で作業が遅れて行くことになりました。

この頃は、人件費より、開発設備の方が高価だったために、このような開発の進め方をしていたのだと思います。当時は、大型計算機で処理する前には、十分な確認を実施しなければ、１日を棒に振ることになりましたから、相当な時間をかけて、大型計算機に依頼する前のチェックを実施していました。

今から３０年以上前（1970 年代）には、このような開発環境で、実際に開発されていました。今では、想像できないような開発環境だったと思います。

この開発環境は、今から３０年くらい前（1980 年頃）になると、大型計算機と開発者の事務所を結ぶモニターに似たディスプレイが設置されるようになりました。そのディスプレイは、直接、大型計算機と接続されていて、コンパイルの依頼や実行結果の確認（モニタするだけ）ができるようになりました。一部のディスプレイでは、コンパイルするためのソースプログラムを修正することもできるようになりました。とは言っても、現在のサーバとクライアントのような関係での接続ではなく、イメージとしては、サーバにたくさんのディスプレイとキーボードを繋いだような、そんな開発環境だったと思います。

図３　３０年くらい前のソフトウェア開発環境



このような開発環境になっても、大型計算機での処理を依頼してから実際に実行されるまでにはかなりの時間が必要でした。コンパイルを依頼して、その結果を確認するまでに必要な時間は、２～３時間ほどかかっていたと思います。３０年以上前の開発環境では、１日かかっていたのですから、それに比べれば、かなりレスポンスが向上したと言えると思います。とはいっても、開発環境は、大型計算機を多くの開発者で共有する方法でしたので、ユーザ環境でテストする場合には、さらに大変な開発環境を強いられていました。具体的には、大型計算機を占有して「ユーザ環境を構築」してからテストを実施しなければならなかったのです。計算機時間を確保するために大型計算機センタのマシン時間の予約を申請すると、深夜の計算機時間が割り当てられることも少なくありませんでしたので、徹夜勤務になりました。

この環境を大きく改善したのは、VM と呼ばれる大型計算機を複数の計算機に見せかけるシステムが登場したおかげでした。今では、PC の世界でも VM が存在します。イメージは、PC で動作する VM と同じです。この VM を使うと、大型計算機を複数の開発者で共有できます。つまり、同時に何人もの開発者があたかも自分自身が大型計算機をひとりで占有しているように使うことができました。これにより、計算機を使用するために、徹

夜勤務をする必要がなくなりました。開発作業時間帯が、ほとんど、昼間にシフトできるようになってきたのもこの頃です。当然、深夜勤務は、大幅に減りました。

また、VM のおかげで、OS(オペレーティングシステム) の振る舞いについても詳細に分析できるようになりました。これは、VM 環境下では、OS 自身がアプリケーションプログラムとして動作するので、実行状況を完全にモニタできるようになったからです。

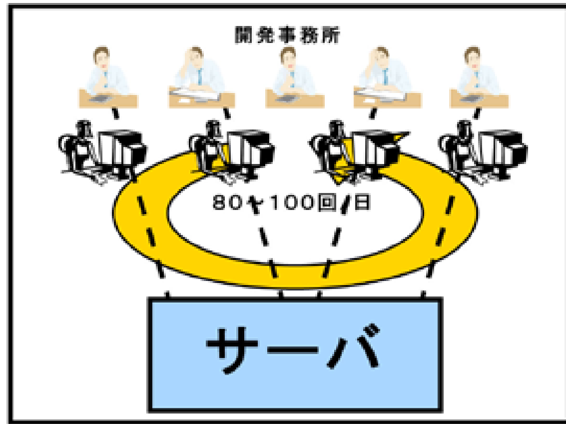
その後、２０年前（1990 年頃）くらいになると、一つのグループにワークステーションと呼ばれる、今で言えば、パソコンとサーバーの中間のようなハイスペックなマシンが割り当てられるようになりました。これにより、開発作業も格段にスピードアップしたと思います。とはいっても、開発者一人一人に自由に使えるパソコンのような開発環境ではありませんでした。現在の開発環境からすると、開発スピードはかなり落ちる環境だったと思います。

一方で、この頃は、技術者の一部で、電子メールが普及し始めた頃と思います。インターネット環境としても、回線のスピードはまだまだで、最近のような光ファイバーでの環境もありませんでした。通信料金も従量制でしたし高額でしたから、容量の大きな添付ファイルは敬遠され、あるいは、暗黙の了解として、容量の大きな添付ファイルを添付しないことがルールでした。添付ファイルを小さな塊に分解して送信し、受信側で合体させるような技術も使われていました。また、添付ファイルの容量を小さくするために、「データの圧縮／解凍」のためのユーティリティが普及しました。最近では、インターネットの環境も様変わりし、1MB 程度の添付ファイルは、頻繁に添付されるようになってきました。

Windows 95 が登場する頃になると、やっと開発者一人に 1 台ずつパソコンが配られるようになってきました。こうなると、それぞれの開発者が、自分自身の意思でコンパイルやテストを実施することが可能になりました。ただ、開発ツールなどの環境そのものがまだ未熟でしたから、ユニットテストや TDD のような手法を使うことができませんでした。現状では、PC 上で動作するコンパイラや開発ツールに至るまで、効率的な開発作業ができるように開発支援ツールが進化してきています。コンパイルに 1 日必要だった 30 年以上前の開発環境からすれば、コンパイル結果の確認は、数秒でできるようになりましたから、開発のスピードは格段に向上したと思います。

このことは、現在の開発環境（図 4 参照）では、単位時間当りに実施、あるいは実行できる作業も格段に増えたことを意味します。これは、開発者の開発作業をほんの少し妨げてしまうと、そのまま開発作業の遅延が発生してしまうということです。

図４最近のソフトウェア開発環境



開発者が開発作業の途中で、ユーザの要求仕様に対する疑問がわいたときに、ユーザから明確な応答がないと、開発者は応答があるまで開発作業をストップせざるを得ない状況になります。これは明らかに、ユーザに起因する開発の遅延につながっていることになります。開発の遅延は、ユーザのみならず、サプライヤ側にも開発スピードを落としてしまうことになり、この遅れは、最終的には、ユーザのビジネスの成功を阻害することになってしまうことになります。

ところが、サプライヤからの問い合わせに迅速に応答しないことが、開発効率を確実に低下させていることを理解しているユーザは、まだまだ少ないように思います。この事もサプライヤであるＩＴ技術者のみならず、ユーザ自身も不幸にしている要因になっていると思います。

開発環境の進化により､30 年以上前には気に留めなかったような開発上の僅かな｢分(60 秒)」という単位の「小さな遅れ」が、現在では、開発効率を大きく悪化させることになってきました。Agile JAPAN 2010 で講演した Alan Shalloway もソフトウェア開発において発生する「遅延」を無くすことが､リーンソフトウェア開発で言うところの「ムダとり」につながると言っています。

現在の開発環境では、このソフトウェア開発において発生する「遅延」を究極的に無くしていくことが求められているのだと思います。ただ、ソフトウェアを開発するのは、生身の人間ですから、遅延の無い作業を集中して連続できる時間は、限られます。あるソフトウェア開発会社での実態では、2 時間程度が限界のようで、自然に 1 日の作業では、午前中 1 回、午後 2 回のペースで開発作業が進められているようです。

開発されるソフトウェアの変化

これまでは、開発環境について振り返ってきましたが、開発の対象となっているソフトウェアについても見てみましょう。コンパイルの確認が 1 日単位で、実行されていた 30 年以上前 (1980 年頃) に開発されていたソフトウェアを見ると、ほとんどが「業務の電算化」に伴うソフトウェア開発だったと思います。この業務の電算化に伴うソフトウェアの開発では、仕様という考え方すらなかったようです。

電算化しようとしている業務そのものが「仕様」でしたので、実務担当者がいれば、その人にインタビューすることで「電算化のための実装」は可能でした。しかも、電算化に必要な仕様の変更は一切ありません。このことが、ウォーターフォールモデルでも開発が可能であった原因のひとつだったのではないかと思います。このように、業務の電算化を進めるにあたっては、開発したものに対しても、仕様を変更する必要がありませんでした。ということは、現在のように、少しずつ仕様を確定する必要がなく、すべての仕様が確定しており、しかもそれらの仕様は、開発完了まで変更する必要がありませんでした。

「一度確定した仕様が変更されることがない」という環境でなければ、コンパイル結果の確認に 1 日必要な開発環境の下で業務の電算化のためのソフトウェアを開発することは困難だったと思います。

それでは、最近、開発されているソフトウェアを見てみましょう。現在では、ユーザの先に広がるビジネスの世界が非常に多様化しています。それに伴い、ユーザがソフトウェアに求める要求も同時に多様化してきていると思います。逆に言うと、ユーザの業務の電算化というような案件は、存在しないと言っていいと思います。複雑かつ多様化しているビジネスで成功するために有効なソフトウェアを開発するためには、極端に言えば、日々、要求仕様の見直しをしなければならないことになります。そこで、頻繁に要求仕様の変更が予想される場合の開発プロセスとして考え出されたのが、アジャイル開発プロセスなのだと思います。

とはいっても、アジャイル開発プロセスそのものは万能ではありません。ただ、従来のウォーターフォール型の開発手法よりは、開発途中での要求仕様の変更に柔軟に対応できる開発手法だと思います。アジャイル開発プロセスでのソフトウェア開発は、30 年前の開発環境では実現する事はできませんでした。現状の開発環境があってこそ、成功する開発プロセスだと思います。考えてみれば、コンパイルの確認やテスト結果の確認するのに１日とか、２～３時間とか、そのような時間が必要な開発環境では、開発作業そのものが、

その結果を確認するまで先に進めないために、相当の待ち時間が発生することになります。このような待ち時間が発生する開発環境では、「テスト駆動開発」や、勇気を持って「ファクタリングする」というようなことは不可能であるばかりか、「無謀な行為」になってしまいます。

大事なのは、開発環境にあった開発プロセスを選択すべきなのだと思います。現状、それがアジャイル開発プロセスなのだと思います。現在の開発環境が、さらに進化すれば、アジャイル開発プロセスではない、全く別の新しい開発手法というのが出てくるかもしれません。あるいは開発手法という枠組みではなくて、開発支援ツールの充実という形で現れる可能性もあると思います。

契約や契約書の変化と進化

ここまでは、30 年前 (1980 年頃) の開発環境や、開発されるソフトウェア、その開発手法について、振り返ってみました。この一方で、契約や契約書の方は、どのように変化や進化をしてきたのかを見てみましょう。経済産業省、IPA、JISA など、官民の垣根を超えて、ソフトウェア開発の進化にフィットする契約書というものが、国内外で検討されてきています。ところが、それらを見てみると、契約書の条文そのものは、従来の請負契約をベースとした契約書とあまり変化のないものだと思います。これは、ユーザサイドからの利益の取り方、サプライヤからの報酬の取り方が、相反するため、その着地点を見出す方法を記載することが難しいと言うこともあると思います。

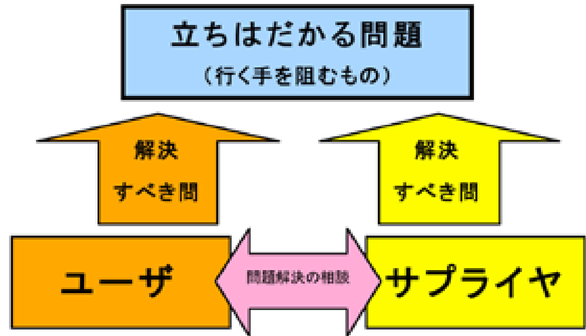
請負契約において、二種類の特徴的な契約金額の決定パターンでこのことを確かめてみましょう。二種類というのは、契約金額固定方式と、稼動実績金額精算方式です。これらを整理してみると以下のようになります。

		契約金額固定方式	稼動実績金額精算方式
契約金額決定の特徴		契約時に決定したまま変更されない	技術者の稼働実績状況に応じて契約金額を支払う
ユーザ	メリット	予算確保、社内稟議が容易（納期が遅延しても契約金額は変更されないため）	予算確保、社内稟議しにくい（開発の進捗状況により契約金額が変動するため）
ユーザ	デメリット	要求仕様を満たしていれば、契約金額全額を支払う（早期に納品できた場合においても同様） 必要工数見積を誤っても、その見積りミスは、ユーザが負担する	各技術者の稼働におけるクオリティが測定しにくいので、稼働実績の適切性が確認しにくい。プアな作業をされることなく、稼働実績が正当に請求されているのか判断しにくい
サプライヤ	メリット	要求仕様を満たすために、納期が遅延しても契約金額は変更されない（逆にペナルティを支払う場合あり）	開発途中での仕様変更も安心して受け入れられる
サプライヤ	デメリット	必要工数見積を誤っても、その見積りミスはサプライヤが負担する	開発者のプロジェクト配置計画が立てにくい（納期の変更を伴う場合は特に顕著）

ここで、注意すべきなのは、前提になっているのが何か、という事です。その前提の 1 点目は、要求仕様が細部まで明確になっていること、2 点目は、その要求仕様は原則として変更されないこと、です。この二つの前提の下であることがこの表が成立する条件です。しかし、現実にはそのような状況はほとんどありません。ユーザは、詳細な要求仕様までは提示できないまま、ソフトウェア開発プロジェクトの開始日を迎えているのが実情なのではないでしょうか。それなのに、検収条件は、要求仕様を満たしているか否か、と言う

ことになっていると思います。理不尽なようにも思えますが、そのように、契約書に書いていることがほとんどですから、どうしようもありません。要求仕様が不確定な部分は、事実上、サプライヤ側がリスクを負うことになっていると思います。このために、サプライヤの見積りには、このリスク分の保険をかけることになってしまっているのだと思います。

図５　プロジェクトを成功に導く関係



ここで、少し原点に戻って考えてみましょう。ソフトウェアを開発する上で理想的な「ユーザとサプライヤ間の意識」を考えてみるとその辺の事が少しクリアになってくると思います。開発プロジェクトを成功に導くためには、両者でどのような心構えとか想いが必要なのでしょうか。一つの答えとしては、両者の想いを先ず「運営協働体」（図５参照）とする事で、プロジェクトの成果をより大きなものにすること、そしてこれがお互いの最終的な目的である、と両者で共有して捉えることです。

そうすれば、プロジェクトは成功するでしょう。反対にそう捉えられないと、プロジェクトそのものが迷走してしまうことになると思います。さらに、プロジェクトが迷走をし始めると、ユーザとサプライヤの間では、「運営協働体」としては、全く機能しなくなります。逆に、リスクの押し付け合いになることが多くなってしまいます。こうなると、ユーザとサプライヤの関係はますます悪くなるばかりか、プロジェクトの成功のための成果はどんどん小さなものになってしまいます。

このように考えてくると、最近の新しい開発プロセスにフィットした契約書というものは存在しないのではないでしょうか。かと言って、アジャイル開発プロセスで開発できないわけではありません。逆に言うと、ソフトウェアの開発プロセスを限定するような契約書というものは、存在しないと思います。一般的な契約書に書かれているのは、ユーザの求めているソフトウェアを、決められた納期までに完成させて、ユーザに納める、ということだけで、ソフトウェアの開発にどのような開発手法を採用するか、とか、どのような開発手順で進めていくかについては、明確に特定されていることはほとんどありません。ということは、サプライヤは、サプライヤが最適と考える開発手法や手順でソフトウェア

を開発することができるということです。

それなのに、なぜ、アジャイル開発プロセスを開発手法として採用するとうまく開発できないのでしょうか。その理由は二つあると思います。一つ目は、ユーザとサプライヤの間には全く別の、極端に言えば正反対の利害関係が存在することにあると思います。二つ目は、ユーザ側が、順調に開発されているかどうか確認するために、手順を踏んだ納品物を要求しているからだと思います。これらの二つの理由は、次のように解決することができると思います。

一つ目の理由を解決するには、ユーザとサプライヤの間に存在する全く正反対の利害関係は、ユーザがソフトウェアの開発を通して解決しようとしている問題をサプライヤとともに、力を合わせて両者で解決する問題としてとらえることで、ユーザとサプライヤがともに対峙することなく、同じ問題を見つめてその解決にあたる事ができると思います。

二つ目の理由を解決するには、ユーザの開発上の不安をできるだけ初期で払拭してやることだと思います。アジャイル開発プロセスで開発できるようになるまでは、サプライヤは、ソフトウェアを完成させるまでユーザに見せる事ができなかった事が、ユーザを不安にさせていたんだと思います。その為に、ユーザが、開発が順調にいっているのかを不安に感じ、最終納品物であるソフトウェアの納品より前に、開発途中の仕様書、設計書などソフトウェア以外の納品物を要求していたのだと思います。開発の進捗状況の把握あるいは管理のために、それらの書類の提出を求めていたという側面もあったと思います。

現在の開発環境では、ソフトウェアが出来上がったところまで、実行できる状態でユーザに提供することができるようになりました。このことは、開発状況を把握するあるいは、確認するためには、非常に有効な方法だと思います。このことをユーザに理解してもらえれば、ユーザが本当に確認したかったことが、実現できると思います。

プロジェクトの遂行に当たっては、ユーザとサプライヤには、双方にリスクとそのリスクに対する報酬が、付きまとってきます。お互いに、リスクをできるだけ小さく抑えて、報酬の方はできるだけ大きくしたいのは、共通した思いだと思います。ところが、両者のリスクは、相反している場合が多く、従来、このリスクの負担はサプライヤ側が負う事が多くなっていたと思います。

今まで、機能仕様書や詳細設計書がユーザにとって必要だった理由が、「ソフトウェアの開発が順調に進んでいるかどうかを確認するもの」だったとすれば、ソフトウェアの開発状況をそのソフトウェアそのものを実際に見て、触って、確認することができるので、

機能仕様書や詳細設計書をソフトウェアの開発の途中では、必要なくなると思います。極端に言えば、これらの書類は開発途中の段階では、あくまで「開発に関わるメモ」の存在であり、最終的に実装されたソフトウェアのドキュメントが納品されれば、十分だと思います。このことをユーザに理解してもらうことで、アジャイル開発プロセスでのソフトウェア開発に対するハードルを低くすることができるのではないかと思います。

実際のソフトウェアの開発においても同じようなことが言えると思います。どういうことかというと、ユーザがサプライヤに対して提示する要求仕様と言うのは、もしかすると、ユーザ自身がそういう手順、あるいはそうしなければいけないと思い込んでいるだけで、サプライヤサイドの技術者が考えると、別の解決方法でもっと簡単に対応することができる場合があると思います。逆に言えば、ユーザが本当に実現したい物事をサプライヤが理解することで、無駄な開発をしないようにすることができると思います。

本来、ユーザとサプライヤで合意する契約においては、サプライヤは、「ユーザの示す要求仕様を実現すること」をコミットするのは間違いだと思います。ユーザの示す要求仕様の本当の要求が何かを捉えながら、開発を進められるのが優れたサプライヤだと思います。

「幸福は対抗の意識のうちにはなく、協調の意識のうちにある」(Andre Gide『文学と倫理』) と言うことも、ある程度認識しておかないといけないでしょう。ただ、仕事ですから「幸福である」と思って開発する必要はありませんが、少なくとも、スムースに開発できる環境になるためには、契約当事者が、「対抗の意識」ではなく、「協調の意識」の方がはるかに、スムースに開発できる環境である事は、確かだと思います。

これからの契約書について

いままで、開発環境の変遷や開発されるソフトウェアについて、懐古しつつ回顧してきました。最後に、これからの契約書について考えたいと思います。契約書とは、契約当事者が合意した事項が書かれた書類です。両者は、その契約書に書かれた合意内容に基づいて、各々の義務を遂行することになります。この事は、契約書に条件を書けば書くほど、両者の約束事が増えることになります。極端に言えば、「足かせにつける錘を増やす事」と同じ行為になってしまいます。であれば、契約書には、基本的な約束事の他には、余計な事は書かないほうが良いということだと思います。

契約書に、約束事を書けば、それだけ契約の条件が複雑になっていきます。たとえば、条件ごとの優先順位、例外事項が発生したときの優先順位をどうするかとか。極端な例かもしれませんが、スポーツのルールブックは、競技をするうえでは契約書といえると思います。野球のルールとサッカーのルールで簡単に比較してみましょう。

野球のルールは、9人同士で行う競技で、攻撃と守備を交互に9回行い、最終的に得点の多いチームが勝ちとなる。得点の方法は、攻撃チームのランナーが、1塁から3塁の順に進み、ホームベースに戻ってきたときに得点できる。1塁へ進む方法としては、守備チームのピッチャーが投げたボールを攻撃チームの選手（バッター）がバットを使って、そのボールを弾き返す。弾き返したボールが、ファウルとならなければ、バッターは、守備のチームが1塁のプレーヤーに、その弾き返したボールを渡す前に、1塁に到達しなければならない。到達できない場合はアウトとなり、アウトの数が3になったら、攻撃と守備の役割を交代する。と、ここまで書いても試合全体のルールの全容が分かりません。さらに、細かいルールを書こうとすると、たとえば、「内野フライ」が上がった際に、ランナーが1塁にいる時と、いない時、アウトの数などで、「内野フライ」に対してアウトにするかしないかのジャッジに差がでてきます。このように、同一の事象なのにジャッジが変わるルールは、複雑だと思います。

一方、サッカーのルールは、野球のように、同じ事象が別のジャッジとなる事はほぼ無いと思います。サッカーのルールを書いてみると、11人同士で行う競技で、手以外の体を使って、ボールを扱い、相手ゴールにボールを入れたら、自分のチームの得点になる。そして、前半、後半とも45分間競技して、得点の多いチームが勝ちとなる。特殊なルールとしては、オフサイドがあるくらいで、非常にシンプルなルールだと思います。

このように、契約書というのは、取引をする上でのルールブックのようなものですから、

たくさんの条件を書いたり、複雑にしない方がいいと思います。契約を結ぶ目的は、ユーザ、サプライヤのビジネスの成功ですから、アジャイルマニュフェストにも「契約交渉よりも顧客との調和を」と謳われています。契約書は、あくまで、ユーザとサプライヤのビジネスを成功させる、という目的を実現させるためのツールですから、以下の項目を書いておけば、事足りると思います。

要求仕様、納品物、納期、金額、検収条件

ここで、納品物と納期については、注意が必要です。設計書、仕様書の類を中間納品物にしないようにします。そうしないと、間接的に滝型の開発手法を宣言していることになってしまいます。また、開発途中での「要求仕様の見直し」ができるようにしておきましょう。もちろん、契約当事者の合意の下で変更します。仕様の見直しのプロセスを決めておくのも良いと思います。あと、大事な事は、開発プロセスを限定しないこと、です。これらを意識して契約書をつくれば、アジャイル開発プロセスでのソフトウェア開発が可能になると思います。

ここで、ちょっと視点を変えて、契約書がどこから来たのか考えてみましょう。この事が、これからの契約書のヒントにつながります。契約書そのものは、日本の外から持ち込まれたものだと思います。なぜなら、日本の文化とか習慣といった日本人の心という部分が、織り込まれていないように感じるからです。日本には、日本の文化にフィットした契約書があるのではないかと思います。そのヒントになるキーワードが、「おもてなし」とか「おまかせ」といった、日本人の心や文化に関係する言葉だと思います。

「おもてなし」は、心をこめて相手にできるだけの事をする事です。「おまかせ」は、丸投げする意味ではありません。任せる方の心構えとしては、任せたからには相手を信頼し口出ししない、という心です。また、任せられた方の心構えとしては、任せられたからには相手に満足してもらえる成果を出す、という、おもてなしに通じる心だと思います。

このような日本人の心については、「暗黙知」の存在だと思います。ですが、日本人同士なら分かり合える部分が多いと思います。日本人の間に限定すれば、形式知化した日本の文化だと言うことができると思います。ただ、この日本の文化を契約書に反映させるのは、大変です。「暗黙知」である日本人の心を形式知化する事になりますので、かなり難しい事ではありますが、日本国内での取引に限定するならば実現は可能ではないかと思います。とは言っても、契約書の中に「おもてなしの心」で開発すること、とは書けませんから、どのように表現すればいいのか、あるいは、直接表現しなくともそうしなければならないような条件が言葉として定義できるのか、これからも研究したいと思います。

（完）

小論

知働化が切り開くソフトウェア工学の価値創造　竹内雅則
ソフトウェアと価値

なぜ、あなたはモデルが描けないか　天野勝
概念モデルと業務分析モデルの関係

ソースコードを書くことに似ていること　中村裕樹
ソースコードという文

新しい知識のカタチ　羽生田栄一
知識に対する新しい取り組み

編集と知働化　野口隆史
新時代の知識編集

小論の章では、比較的短編の作品を掲載しています。多種多様な観点からの「知働化」への想いが詰まっています。

『知働化が切り開くソフトウェア工学の価値創造』は、企業経営に携わっている竹内雅則氏からの経営者の視点での「知働化活動」の果たすべき役割への期待が述べられています。

『なぜ､あなたはモデルが描けないか』は､天野勝氏の日頃のトレーニングやコンサルティングの経験からモデリングについての想いが語られています。

『ソースコードを書く事に似ていること』は､中村裕樹氏の「コード中心」の世界観によって、その読み書きの本質を問うものです。

『新しい知識のカタチ』は、羽生田栄一氏の思索の断章としての、新しい時代へ向けた取り組みです。

『編集と知識化』は、編集を生業にしている野口隆史氏の、本誌編集の経験もふまえた知識編集の神髄に迫っている作品です。

小論

〔知働化研究会誌創刊第１号〕
知働化研究会
Executable Knowledge and Texture Laboratory

知働化が切り開く ソフトウェア工学の価値創造

ソフトウェアと価値

竹内雅則（たけうち まさのり）

株式会社ヴィクサス

結論

ソフトウェア工学領域で知働化研究が始まっている。知働化研究は単にどのようにソフトウェアを設計するのかというレベルの話ではない。高度なソフトウェア工学の知識使い、新たな付加価値創造により経済的効果を生んでこそ真の知働化研究であり、ＩＴエンジニアが担う将来像でもある。

１．はじめに

ＩＴ業種は学生の就職先としての人気が落ちていると聞く。新３Ｋ業種と呼ばれて久しいが、それ以上にＩＴ企業に対して将来の期待を感じられないというのも影響している。

日本国内の市場環境から見ると、世界経済危機の影響もありマイナス成長に陥っているＩＴベンダーも多い。金融危機による投資の抑制、消費の低迷による影響も大きいが、それ以前にＩＴゼネコンと呼ばれる業界構造の中で、バブル崩壊以降採用を抑制してきたメーカ各社のＩＴ人材不足を担ってきた受託系ＩＴベンダーなどは、開発費低減を目的とした

Copyright 2010, TAKEUCHI Masanori, All rights reserved.

オフショア開発や、グローバルマーケットで成功しているパッケージベンダーなどとの競争激化の中で、構造的な問題を抱えている。

ソフトウェア開発でも他の産業同様､開発生産性を向上させることは簡単な話ではない。モデル化、部品化、そしてさまざまなツールを駆使して生産性の向上は日々努力されている。しかし、短期間で実現できる手段は限られ、手作りが主流となっているソフトウェアの開発部分では、開発者に許される開発時間が減少し、開発現場は劣悪な環境へと陥りがちとなっている。

また、ソフトウェア開発のように、一定の教育を受けた人間で可能な頭脳労働という労働集約型の作業は、簡単な作業であればあるほど、より優秀で給与水準の低い地域に移すことが可能である。分野によっては、エンドユーザーが本来直接関わる必要の無いはずのソフトウェア設計領域にまで入り込み、より低価格なオフショア開発を要求にする事例も出てきている。

このような環境で、ＩＴ企業は伸びるわけもなく、学生は日本の多くのＩＴ企業に期待を抱けないでいる。ソフトウェアの価値に対する考え方を変えてゆかない限り、日本のＩＴ産業は衰退を辿ってしまうだろう。

２．中国のＩＴ技術者の現状

経済発展が目覚ましい中国に視点を向けてみると、ＩＴ技術者は日本とは大きく異なり将来への希望に満ち溢れた職種である。初任給は２～３万円程度ではあるが、彼らの物価水準からすると実感として７～１０倍ぐらいの感覚となる。また、将来外資系の企業に転職するなどで月額２０～３０万円程度の報酬が得られる可能性があり、収入面での期待は非常に高い。また、先進国としての日本に対する憧れもあり、日本語を学びオフショア開発企業に勤め、将来日本に移り住むというのも１つの選択肢となっている。

大学もこのニーズにうまく応えている。特に大連や杭州などの地方政府がＩＴ産業を後押ししている都市では民間企業と（日系企業も含め）連携し、学生が実際の企業の製品開発を学ぶ事で単位を取得できる大学も多い。企業は教材を提供することで、自社の開発方法を学んだ優秀な学生を優先的に採用することができる。一方、学生にとっても経験を得て即戦力として採用してもらえるという環境を得られるので血眼になって勉強する。このように企業、学生、大学が共に WIN-WIN の関係を持ちＩＴ産業の活性化を推進しているのである。

日本のＩＴ技術者はこのような現実を直視しなければならない。今は日本という特殊な

文化を持った市場であることや言語障壁があり影響はゆるやかではあるが、彼らはやがて日本の市場特性や言語、文化をマスターし、日本のＩＴ技術者の驚異となってくるはずである。同じ土俵の上では生産性やモチベーションの競争ですでに負けているのである

３．ソフトウェア知識の価値とは

そもそも、ソフトウェアの中での知識の価値とは何であろうか？単にソフトウェアコードが書けるレベルから、構造設計ができる、要件定義ができる、テスト設計ができるというレベル。さらにはモデル化に基づきメンテナンス性と品質が高いシステム設計ができるというレベルもある。形式手法のようにソフトウェア言語仕様を向上させる取り組みやツールによる自動化の流れもある。これらは決められた要件に対していかに効率よくソフトウェアを設計していくのか追及していく方向である。これらの事が他人・他社よりより早く合理的に行う事ができれば、それには経済的な価値があるということになる。もう一方で、i-pod や Kindle のように新しい産業を生み出すような価値もある。これらの事例はいずれもサプライチェーンの中抜きにより､消費者により快適な生活を生み出している。これらのモデルの実現にはソフトウェアは不可欠であり、開発の効率化とは違った価値を生み出している。

このような価値を生み出す要素として、ソフトウェア工学や設計技術、さまざまなドメインでの開発経験など、知識と経験がある。他人よりより多くの知識と経験を獲得することでの価値はあるが、これだけでは十分条件は満たしていないはずである。１つの機能を生み出す事での利便性向上による価値はあるものの、世の中を豊かにするというレベルでのより大きな価値としてはよりクリエイティブなアプローチが必要になる。

４．歴史に見るソフトウェア領域の進化と知働化研究の方向性

かつて４ビットマイコンの世界の中のように限られたＨ／Ｗ上でのソフトウェアでは、限定されたデバイスの制御手段としてのソフトウェアが主流であった。その後、パーソナルコンピュータが発展し、ネットワークという手段により WorldWideWeb として進化。ソフトウェアは情報を時間と空間を越えて結ぶ手段として飛躍的に発展していった。コントロールできるデバイスは、世界中のあらゆるものとなり、システムとしてモデル化の対象となるものは実世界そのものに限りなく近づいてきているようにも思える。

このような環境の中で、どのようにすれば新しい価値を効率的に生み出す事ができるのだろうか。ソフトウェア工学としてのアプローチがよりクリエイティブ側に進化していけば、理論的に価値の高いシステム構築を行えるはずである。

５．ソフトウェアの価値創造の特徴と知働化研究

価値創造の方向性にはいくつかの特徴がある。まず対象領域が実社会そのものとなっている。特に、人間の欲望を起点として、ＩＴがその欲望を実世界に対して影響を与えるという現象も顕著になってきている。例えば、Google では、検索エンジンにより情報を発信したいという欲望と入手したいという欲望が結合された。結果として情報を知っているというレベルの知識の価値を劇的に下げた一方で大きな経済効果を実現している。また、Youtube や Twitter も個人の意識や感情・欲望がダイレクトに社会を動かすという方向に向いているのが特徴の１つである。

次の特徴として、アジャイルが挙げられる。これらの新たな取り組みは、決して最初から要件定義できるというものでは無い。大きな方向性としてプロトタイプが立ち上がり、利用者のニーズを検証しながら進化している。要件定義そのものが重要ではなく、いかにニーズに的確に迅速に対応するかが重要となっている。

さらに条件をもうひとつ挙げるならば、ソフトウェアが接するデバイスは人間そのものに近づいている。Ｋ／Ｂやマウス、タッチパネルは人間の意志を電子化する手段であり、いずれ新しい方式が生まれてくるであろう。今、普及しているシステムは、人間のワークフローの中で限られたＨ／Ｗという制約の基で最低限の手助けをしているに過ぎない状態である。Ｎ et 上で発展している新たなビジネスモデルはこのような特徴も持っている。ソフトウェアの知働化研究はソフトウェア工学の領域ではあるか、単にどのようにソフトウェアを設計するのかというレベルの話ではない。このような特徴を理解し、新たな付加価値により経済的効果を生んでこそ真の知働化といえるはずである。

6. 最後に

上記のようなソフトウェア環境の変化がある中で、実経済としては汎用機開発など、まだまだ従来型の開発が行われ、その中で日々ＩＴ技術者が活動している。しかし、顧客の要求を単に造り続けるだけではおそらく日本のＩＴ産業は衰退していくだろう。しかし、新しい産業を生み出すような価値創造にはＩＴ技術が不可欠であり、クリエイティブで優秀な頭脳が集まる知働化研究の中で、新しい理論が生み出されると共にグローバルな視野を持つ IT 技術者が育つことで、日本のＩＴ産業の流れが変わる事を期待したい。

小論

〔知働化研究会誌創刊第１号〕
知働化研究会
Executable Knowledge and Texture Laboratory

なぜ、あなたはモデルが描けないか

概念モデルと業務分析モデルの関係

天野 勝（あまの まさる）

株式会社 永和システムマネジメント コンサルティングセンター
http://sec.tky.esm.co.jp/

オブジェクト倶楽部
http://ObjectClub.jp/

概要

システムエンジニア向けの業務分析モデリングの研修の中で、業務構造の概念モデルをUMLのクラス図を描いてもらうが、研修時間内に質の高いモデルを描けるようになる人はごくわずかである。この原因を考察すると、業務知識の保有の程度がそのまま質の低さにつながっていることが見えてきた。

Ⅰ．はじめに

「このモデルであっていますか？」

筆者がモデリングの研修を行なっているときに、よく聞かれる質問である。SEと呼ばれる職種の方を対象に、要件定義の作業の質を上げるために、業務分析モデリングを行なえるようにする研修を行なっている。その中で、モデルを描く演習をしてもらっている。しかし、受講生が描くモデルを見ると、そのほとんどが「いけていない」のである。なぜ、このようなことになるのだろうか。

Copyright 2010, AMANO Masaru, All rights reserved.

講師の教え方に問題があり、そのせいで研修期間中に受講生がスキルアップできないという原因もあるだろうが、今回はこの原因は除外させていただくことにする。

Ⅱ．現状認識

受講生の多くは、システム開発・保守に何かしらの形でかかわっており、少なくとも4年以上の業務経験を持っている。

研修は主にモデリングに必要な表記法と、良いモデルを描くためのモデリングの観点を説明し、これらの知識が身についているかをミニ演習で確認し、研修の最後に受講生が関わっている情報システムで扱っている業務のモデルを総合演習として描くという構成となっている。

業務分析モデリングを行うには、当然のごとくモデルを描くための表記法などの知識が必要である。これについては、研修を充分に行っており、ミニ演習の結果を見てもほとんどの受講生が完璧とはいえないまでも理解はしていると認識している。

研修の仕上げの総合演習で、受講生がかかわったシステムで取り扱っている、主要な業務を中心に、UMLのアクティビティ図で業務フローを描き、UMLのクラス図とオブジェクト図で、業務で取り扱う主要な概念を描いてもらっている。これらのできが「いけていない」のである。

Ⅲ．考察

「いけていない」業務分析モデルを描いてしまう原因を探ったところ、その最たるものは、「業務知識がない」ということであった。受講生の多くが一つの情報システムに1年以上携わっており、いるが、その対象としている情報システムがあまりにも大きいため、明確に作業が分担されており、必要な業務知識が限定的になっており、経験年数の割には、理解している業務知識が少ないのである。そのため、業務フローを描いてもらうと、情報システムの使用方法になってしまっており、その情報システムを使っている業務のフローは描けないのである。同様に、概念のモデルを描いてもらうと、クラス図だけを描きながらモデリングを進めていくが、クラスのインスタンスを示すオブジェクト図はほとんど描かないのである。その概念が実際の業務で具体的にどのように扱うかを知らないため、オブジェクト図が描けないということなのである。これはモデリング以前の問題であり、モデ

ルが描けなくて当然といえば当然と言えるだろう。講師も得意な業務ドメインであれば指導できるが、知らない業務ドメインであれば、コメントすら困難である。

次の原因は、プログラマー視点によるものである。概念をクラスとして表現する際に、プログラミング言語のクラスの視点から抜け出せずに、概念ではなく機能を挙げてしまうのである。プログラムレベルでクラス図を描けば、そこには機能が登場するのは当然であるが、概念を描く場面では機能は登場しないのであるが、その区別がなかなかつかないようである。かなりしつこく指導をしても、区別できない人は区別できないのが現状である。

　他にも、モデリングの基礎となる考え方になじめないというものである。オブジェクト指向に限った話ではないが、汎化・特化と集約の区別がなかなかつけられないのである。これは、簡単な例をいくつか示すことで区別がつくようになるので、さほど大きな問題ではない。

　これまで、モデリングができない原因を並べてきた。しかし、研修の中でも素晴らしいモデルを描ける受講生がいるのも事実である。もともとセンスが良いというのもあるのだろうが、やはり経験によるところが大きいようである。システム開発の一連の流れを体験している受講生のモデルは全般的に質が良い。また、保守の経験しかない受講生でも質の高いモデルを描くことが多い。とりわけ、ほどほどの規模のシステムを苦労しながら一人ないし、数人で担当している場合はモデルの質が高い。これは、システム全体が見渡せているからこその結果であろう。

Ⅳ. 提言

　ここで、モデリングが行えるようになるためにはどうすればよいか、筆者なりに提言させていただく。

Step1

まずは、表記法を理解することである。表記法を共有しなければ、いくら絵を描いても、そこに描かれているものを伝えることは困難である。表記法の中には、モデルの描き方のヒントが詰まっているので、表記法の理解が深まれば、それに応じてモデルを描く際の視点などが安定してくる。クラス図であれば、どのような関連が描けるかを理解するだけでも、モデルの描き方が変わってくる。

Step2

次は、モデルのパターンを理解することである。業務知識が充分にあれば、それなりにモデリングが行えるだろうが、業務知識が無い状態では業務をどのように捉えればよいか足がかりがない。そこで、足がかりとして、基本となるパターンを理解すれば、そのパターンを切り口に業務の分析が行いやすくなる。パターンを理解するには、『アナリシスパターン』『UML モデリング入門』『UML モデリングレッスン』が良い参考書籍である。

Step3

続いて、モデルリーディングを行う。良いモデルが描けない段階では、悪いモデルを見るのは百害あって一利無しである。良いモデルを、多く見る必要がある。そして、悪いモデルのどこが悪いのかが分るようになるまで、良いモデルを見るのが望ましい。

Step4

最終的には、モデルを多く描いて訓練をする。これに尽きる。

描いたモデルを有識者によってレビューしてもらえる環境があれば、表記法を理解したStep1 の後で、Step4 までスキップし、モデルを描いてフィードバックをもらえるだろうが、なかなかそのような恵まれた環境は手に入らないだろう。通常は、Step1 から順にスキルを積み上げていくことになるだろう。

Ⅴ．おわりに

モデリングができない理由と、そのトレーニング法について述べた。モデリングができるようになるには、充分なトレーニングの時間が必要である。筆者が提供している研修は2 日間という限られた時間内で、Step1 の表記法の理解と、Step2 のいくつかのパターンを紹介しただけで、業務モデルを描かそうとしている。これでは、「いけてない」モデルとなってしまうのは当然である。

知働化の定義に関しては、諸説あるが、「知」そのものが自律すると考え、知を切り出したものがモデルとするならば、「モデリング」は、知を自律させる前に求められる一つの過程と捉えることができるだろう。しかし、自信を持ってモデリングを行えるようになるには、トレーニングが必要で時間がかかってしまう。より効果的なトレーニング方法の登場を待ち望んでいる。

〔参考図書〕

- Brian Wilson 著、根来龍之監訳、『システム仕様の分析学–ソフトシステム方法論』、共立出版、1996
- Martin Fowler 著、堀内一監訳、児玉公信、友野晶夫訳、『アナリシスパターン』、アジソンウェスレイ、1998
- 児玉公信著、『UML モデリング入門』、日経 BP 社、2008
- 平澤章著、『UML モデリングレッスン』、日経 BP 社、2008

〔プロフィール〕

天野 勝 (あまの まさる)

株式会社永和システムマネジメント勤務。オブジェクト指向による開発技術の導入や、アジャイル開発プロセスの導入に関する教育、コンサルティングに従事。オブジェクト倶楽部の事務局長や、アジャイルプロセス協議会の編集委員など、コミュニティ活動に積極的に参加。

小論

〔知働化研究会誌創刊第１号〕
知働化研究会
Executable Knowledge and Texture Laboratory

ソースコードを書くことに似ていること

ソースコードという文

中村 裕樹（なかむら ひろき）

株式会社アッズーリ
hiroki.nakamura@azzurri.co.jp

概要

ソースコードも普通の文とみなし、自然言語の文とどのように違い、どのように同じなのかについて考える。

はじめに

ソフトウェアのプログラミングをするとは、ほとんどソースコードを書くということである。プログラムを作るというよりも、ソースコードを書いているという表現のほうがより直接的で、私自身の実感にも合うし、むしろそれ以外の事をしていないという方が適切である。それではソースコードを書くとはどのようなことか。それは単に文を書いているということでいいのではないか。殊更に、特別な表現を使う必要はなく、ただちょっと風変わりな文を書いているのであるということでいいのではないかと思う。

Copyright 2010, NAKAMURA Hiroki, All rights reserved.

I．ソースコードについて

ソフトウェアを作る

ソフトウェアを作ると表現するが、その中での実質的な作業はソースコードを書くことである。設計こそが実質的な作業だという人もいるかもしれないが、設計作業からできた成果物である設計書は。決してコンピュータを動かすことはできない。ソフトウェアは「書かれたとおりに動く」というように、設計の意図どおりに動くわけではない。「いや、私の設計書は、ソースコードの中身と一対一で対応している」と言うのなら、すでにソースコードであり、そうでなくてもソースコード的である。

個人的には「だったら、直接ソースコードを書けばいいのでは」と返したい。話は逸れたが、ソフトウェアを作ることは実質的にソースコードを書くことだという点には納得してもらえると思う。

ソフトウェアを作る

それではソースコードを書くというのはどういうことか。この問いに直接の答えはない。あることとの類似によってソースコードを書くということを説明したいと思う。ソースコードを書くということは、文を書くということに似ている。ここでいう文とは日本語や英語など自然言語で書かれている文である。この類似によっていろいろなことが説明できる。

プログラミング言語の目的と自然言語の目的

プログラミング言語は、自然言語とは違って、意図をもって生み出された。この時点で、プログラミング言語と自然言語との類似が破綻していると指摘できるかもしれない。しかし、ここではプログラミング言語でソースコードを書くということと自然言語で文を書くということの類似であるので、（おそらく）問題はないだろう。自然言語は目的を持って生み出されたと言う人はいないと思う。「神が作った」という信念をお持ちの方もとりあえず納得して欲しい。プログラミング言語の目的とは、最終的にはコンピュータを動かすためだが、まずはソースコードを書くためであろう。

ソースコードを書く人と書かない人

プログラミング言語がある目的のために生まれたにも関わらず、プログラミング言語を知っていてもプログラムのソースコードを書かない人がいる。これは、仕事でプログラミングを始めたてころからの疑問であった。ここで、ソースコードを書くというのはある一定規模以上のソフトウェアのそれであって、例えば「Hello, World」を出力するだけようなの小さなものは対象としていない。

しかし、プログラミング言語でソースコードを書くことと日本語の文を書くことに置き換えてみれば、全然おかしなことではない。日本語を知っていても誰もが文章を書くわけではない。したがって、プログラミング言語を知っているからと言って、それなりのソースコードを書くわけではないのである。

ソースコードとはどのような文章か

ソースコードを書くことは、どのような種類の文章を書くことに類似しているのか。とりあえず、日本語の文章としてどのようなものがあるが挙げてみる。日記、伝記、(短編・中篇・長編）小説、風刺、寓話、戯曲、童話、論説文、随筆、紀行文、詩、歌詞、俳句、短歌、広告、嘆願書、申請書、会見記事、取材記事、評論、宣言文、意見、スピーチ原稿、案内書、手紙、願書などなど。これですべてではないが、ここでやめておく。

ソースコードは意見を表明しない。ソースコードがなんらかの方法で実行されると意見を表明することはあるかもしれないけれど。そして論を提示するものでもない。また、ソースコードでは過去に起こった事実を記すことはない。願望や予測が書かれているわけではない。

さらに、ここではソースコードは鑑賞の対象ではないとしよう。実際には、個人的に以下の C 言語のソースコードは鑑賞の対象になりうると思う。自分自身を表示するプログラムのソースコードである。

```
main(a){printf(a,34,a= » main(a){printf(a,34,a=%c%s%c,34);} » ,34);}
```

(http://www.ipsj.or.jp/07editj/promenade/4703.pdf の 303 ページより）

しかし、コンパイルして実行しなければわからないので、ソースコードが純粋な鑑賞の対象であるとは言えまい。鑑賞が読む側の態度によって決まるのであれば、鑑賞の対象といえるかもしれない。

いままでの条件から絞り込むと（短編・中篇・長編）小説、随筆、申請書、案内書、手紙、願書が残る。しかし、ここで挙げた条件の選択は恣意的なので、条件の選び方で任意の文章の種類を取り除くことができる。だから、このような絞り方は妥当ではない。ただ直感的、個人的に、小説に似ていると思っている。それは大規模な小説では構想力が必要だが、大規模なソフトウェアでも構想力（のようなもの）が必要である。小さな小説では構想力も必要ない（と思う）ように、小さなプログラムはなんとなくできてしまう。

「ソフトウェアにおける構想力とは何か」という問いに、答えを持ち合わせていない。この構想力というのは総合的な能力である。ある特定の個別の能力に依存しているわけではなく、その特定の能力を欠いたからといって構想力が無いということになるようなものではない。

いろいろ考えてきたが、ソースコードはどのような種類の自然言語の文ともぴったりと当てはめられない。

ソースコードと自然言語の文の類似

さきほど、プログラミング言語を知っていても、誰もがある規模以上のソースコードを書くわけではないということを話した。ある規模がどれくらいかを具体的な数値で表すことは難しい。ただ一つ言えるのはソースコードで命名が重要であるかどうかその基準のひとつになる。どんなに長いソースコードであっても、変数・関数・クラス・メソッドなどの命名が重要でないのならば、それは自然言語の文には似ていない。これ以後、単に「文」と表現されているときは自然言語の文を指す。

プログラミング言語でソースコードを書くことと、自然言語で文を作ることの類似により説明できることがある。たとえば、日本語を研究する人がいるように、プログラミング言語を研究する人がいる。日本語を知っているが文章を書かない人がいるように、プログラミング言語を知っているがソースコードを書かない人がいる。日本語で短い文章しか書かない人（書けない人）がいるように、プログラミング言語で短いソースコードしか書かない人（書けない人）人がいる。日本語（国語）を教えるだけの人がいるように、プログラミング言語だけを教える人がいる。

これは、IT が世の中で普及してきた結果、プログラミング言語を書かなくてもプログラミングに関われる人が増えてきたという経済的な現実だけでなく、ソースコードと文を書くこととの類似があることによる。

ソフトウェアに込められた意図

ソフトウェアは作られた「設計の意図」どおりではなく、「書かれたとおりに動く」。作られた当初はソフトウェアの意図が明確であっても、日がたつにつれて、設計者がいなくなり、実装者が変わり、そして新たに機能が付加されるにしたがって、設計の意図は不明確になる。また、大規模なプログラムになると最初から設計の意図がわからない場合もある。それは、言語が本来曖昧なものであるとか、言語でのコミュニケーションが不可能であるという言語一般の問題ではない。ソフトウェアは雑多な機能の寄せ集めで、互いに関連がないだけでなく、時には相反するような機能を含んでいるので、その意図が希薄化されてしまうためである。一般的には、ソフトウェアに込められた意図はソースコードの量に比例して薄められる。

このような性質は文章を書いたり、読んだりしているときに突き当たる問題である。自然言語においても、一般的には、文章の量が増えるにしたがって作者の意図がわからなくなる。

ソースコードを書けるようになるために

文章を書くことと、ソースコードを書くことが似ているとしたら、ソースコードが書けるようになる方法も、文章がうまくなる方法と同じように学べばよい。文法を学ぶ。字を読み、書き、覚える。辞書を引いて単語を調べて意味を理解する。熟語・慣用句・ことわざ・決まり文句を覚える。概念を理解する。文を書く。文章を書く。

プログラミング言語でも同じことをすればよい。つまり、プログラミング言語の文法を学び、言語のキーワードや関数を読み、書き、覚える。コンピュータに特有の用語や API を調べて理解する。ソフトウェアのパターン、常套句を覚える。コンピュータに特有の概念を理解する。ソースコードを書く。

さらに、自然言語の文章では書き方が重要である。文章の構成や修辞法も重要である。プログラミング言語においても同様の物が見つかるはずである。

また、どれかの知識や能力を持たないからと言って、ソースコードが書けるようにならないわけでもない。ソースコードを書くということも、総合的な能力なので、特定の能力に依存しているわけではない。言語能力の一つである。たとえば論理的なことが苦手な人でも自然言語を使ってなにかを伝えることができる。そのことはプログラミング言語でも同様である。

自然言語で文章の書き方を身に着けると、プログラミング言語でのソースコードが書けるということではない。それは、小説が上手いからといって詩が上手いわけではなく、日本語で上手に文章が書けるから英語でも上手に書けるというわけではないみたいなことである。

ソフトウェア固有の知識

ソースコードを書けるようになるには､書こうとしている対象に対する知識が必要である。それだけでなく、書こうとしている対象が必要とするあるいは利用する知識・概念も必要である。知識というのは、CPU の仕様や OS またはライブラリが提供する API の仕様のことである。また概念というのは､コンピュータ上で実現される機能のことで､たとえばセマフォとかファイルシステムとかキューと言ったメタファのことである。

II．まとめ

プログラミング言語でソースコード書くことは言語能力のひとつ

プログラミング言語でソースコードを書くことは言語能力の一つであり、何か特別な能力が必要なわけでもなく、特殊な訓練が必要なわけでもない。日本語や英語を覚えたように、繰り返し使っていけば身につくものである。そして文章を書くのが得意な人もいれば文章を書くのが不得意な人もいる。ただ、単にそういうものなのである。

〔プロフィール〕

２００２年 株式会社アッズーリに入社。現在に至る。

小論

〔知働化研究会誌創刊第１号〕
知働化研究会
Executable Knowledge and Texture Laboratory

新しい知識のカタチ

知識に対する新しい取り組み

羽生田栄一（はにゅうだ えいいち）

株式会社豆蔵

1. はじめに

((A B) C) と (A (B C)) では A,B,C の間の関係や置かれている環境や微妙に異なり、そこから異なる意味が生みだされる。「(新しい知識)のカタチ」と「新しい(知識のカタチ)」では微妙に言わんとするところが異なってくる。このようにコンテキストに依存するかたちで知識というものは働いているらしい。

あたらしい知識のあり方が今はじまろうとしている。知識というものが今までのように「持つ者からも持たざる者への流れ」として捉えられなくなってきている。知識は特定の個人の専有物なのではなく、社会や場がもつある種の共有財であり、知識は場の持つ制約とセットでそこに織り込まれていると考えることができる。知識が生みだされるのは社会活動においてであり、知識が役に立つのも社会活動においてであり、知識とその獲得も、活動とそのための知識も、ともに社会とセットであり、そのことを改めてここでは定式化してみようと考える。

Copyright 2010, HANYUDA Eiichi, All rights reserved.

2. 知識の既存理論

世間的にはおおざっぱに以下のような４種類の知識の分類が提唱されているらしい。

1. 命題知：know-that
   「xx がｙｙである」ことを知っている
2. 実践知：know-how
   「自転車の乗り」方を知っている
3. 存在知：know-what
   「自転車 EM-0」「オブジェクト指向」「塩キャラメル」とは何かを知っている
4. 感覚知：know-what-it-is-like
   実際にどんな「乗りごこち」「使いごこち」「味わい」か知っている

いろいろあるけど、どれもイマイチだよね。どこがイマイチかというと、どの知識も特定の個人という主体がまずあって、その視点から、その主体が知識を「所有する」とまったく安易に考えている点。これを乗り越えないと、知識が売り買いされるのも当たり前だということになって、著作権はそのまま納得しなければならなくなるし、現状の資本主義の悪いところを変えていく、という話にもつながらないだろうな。

3. 現象としての知識生成行為

私たちは、いつも人と会って話したり Twitter 上で無駄話をし合ったり、時にはひとりで引き籠ったりしながら、一見すると意味のない言葉も少しは意味のある言葉も含めてやりとりしたり、誰からの返答も期待せず単にポツンとつぶやいてみたり、逆にこの思いを知って欲しいのに悶々としながら人には一切知られずにいたり、といったことを積み重ねながら生活している。

最近流行が始まった Twitter 上でのやりとりを思い起こしてみよう。有名な知識人や逆に非常に身近な知人の日々のつぶやきに直にリアルタイムで触れることができるようになって、意外な感慨に囚われるようになったことはないだろうか。「1 人 1 人は言っていることも考えていることもそんなにたいしたことないなぁ」「こんな凄い著作を表わしたこの作者はこんなにオバカだったのか」「なんだか○○さんっていつも無口で何も考えてないと思っていたら、実はこんなディープな趣味をもっていたの」「なんでみんな似たようなアイデアを思いつくのかなぁ」「この考えって、自分で思ったんだっけ、誰かが言ってたのを眠い中で何となく聞いてたんだっけ」等々。(たとえば現代の日本の思想界を引っ

張る東XX､茂木YYYや竹内Zやら高橋GG郎などなど誰とは言いませんよ。）なんだか、どの人もみんな緩いアイデアをもって世界に充満しているけれど、その1つ1つは自分でも理解できるしょうもない単なることばの断片。（実はこういう斜に構えたことがいいたいのではなく、個人1人1人はたいしたことないのに、過去の知見やいろいろな人々の努力で生み出されてきた科学理論や、ネット上のあるいはコミュニティ内のさまざま人々のちょっとした気づきや一言がいかに、次のアイデアのステップの役に立っているか、なんと個人の活動は社会の多くの人の力で動かされているか、を素直に言いたかっただけなのですが、このような文章になってしまった。）

一方でTwitter上ではときどき連歌のようなことが現象として起こる。1人のつぶやきがあたかも曖昧なフォーマットを宣言しゲーム開始の勅（ミコトノリ）と誰かが認知してなって知らずにトリガーとして機能し、それに呼応するような形で次のつぶやきが先のインチキフォーマットを踏襲しつつ続き、いくつかつぶやきが続く中で、以前のフォーマットを若干破るようなつぶやきが生じ、ゆるやかにゲームの規則が変わりながらもツブヤキ作品がみんなに共有され進化していく。誰がこの言葉遊びの規則や作成ノウハウや作品群を所有しているということはなく、このような言語ゲーム自体がそれに関わる一連の知識を貯蔵しているのだといえる。つまり知識とは存在や量やどこで誰が所有しているかの問題なのではなく、そのネットワークの渦度（うずど）＝うまく渦になって皆が共鳴している状況（昔の誰かならこれをコヒーレンスのある状態というかもしれないが、ちと違う）の問題なのである。

ワークショップという仕事の進め方の形式がある。特定の目的を掲げて、それ自体がミニチュアのコミュニティでもあり教育でもあり仕事でもあるような人々の集まる場。その場を緩やかに運営するファシリテーターはいるが、基本的に参加者が目的と自分の知識とスキルを結びつけて、周りの参加者と少しずつ目的と関心とお互いの知識やスキルの異同の認識を共有しながら､メンバー間のネットワークが少しずつ作られていく｡ワークショップでは、個人個人の力の総和以上のまったく新しい価値が生み出されることが期待されるし、参加者1人1人も個人で独立して活動する場合と比較して、そこにいつにない不思議な感覚・同期感・一体感・大きな全体に包みこまれる信頼感が生まれ、自分の感覚やイメージ・アイデアとチームとしての感覚・イメージ・アイデアの区別が消滅する。ダイナミックなインタラクションが言葉と体の両方を通して同期していき、異なる背景や知識やスキルをもつメンバーから構成されるチームとしての活動が新たなアイデアや成果につながっていく。（一方で、ある種のチーム全体の愚かさを生み出す可能性にも開かれており、複数のチームでの相互チェックやある種の批判精神のようなチームの全体性に対する外部性も求められるのは確かであり、SSMのアコモデーションのような緩い全体性も含めて、そのあたりの研究も重要。）

ネットワークに死蔵されている膨大な記憶があり、何らかの刺激、きっかけによって、それらの繋がりに断層が走り、参加ノードの間に共鳴が起こり、新しい接続が生まれ、元のネットワークの上にさらに新たな網が紡ぎだされていくのと並行して新しい物語が生み出されていく。こうして馬鹿馬鹿しい幼稚な無意識の言葉のやり取りの貯蔵ネットワークが、あるいは異なる背景や観点をもった異分野のメンバーの参加したワークショップが、ある面では非常に生産的なすばらしい作品の生成の場と化していく。

4. 遺伝子と細胞環境、個人と場

同じ遺伝子があってもそれが組み込まれた細胞やその細胞の置かれた環境の違いによって機能が発現したりしなかったりする。さらに機能が発現しても他の遺伝子の機能との相互作用のタイミングが異なってくることで結果としてそこに生み出される効果は大きく異なってくる。

このことは、遺伝子が遺伝情報という形ですべての生命情報を担っているという言い方はオコガマシイということでもある。遺伝子は、その環境である細胞や細胞を構成している生体が過去から綿々とつながって来なければ、発現の場を失い機能することはできない。そういう意味では、生命を保存してきたのは、遺伝子と生体の両方であり、生体としての個体は生態系の中でダーウィン的な淘汰圧を受けつつ生き延びて遺伝子に貢献しているのだといえる。生体なしにはダーウィン的進化圧を受ける乗り物がないことになり遺伝子も進化できない。

こう考えると、遺伝子だけでなく、生体や生体が多数集まって環境に適応している生物集団自体に後世に残すべき知識が埋め込まれている、ということができる。

このことから、社会に対しても次のようなことが推論できるだろう。

『社会は単なる場ではない』

1. 社会は個人がある理由で集まっている場であるだけでなく
2. 過去の歴史性を帯びているところに意味がある
3. 場と言うとある短い時間スパンの中での個人の関係性、というニュアンスが強いが
4. 遺伝子と細胞環境のエピソードからもわかるように、知識の主体としての社会にとっては空間（空間共有個体間関係性）と時間（歴史共有世代的関係性）の両方が大事である。

これについては、別の機会に改めて考察したい。

5. 知に対するスタンスの変更

以上を踏まえて、知というもののあり方に対するスタンスの変更を無理やりくり進めてみよう。まずはコンサルタント風に、昔の「知識」観の特徴は３つあります。

旧世代の知識「観」に対して、

1. 知はカタチ（形相、フォーム、フォーマット、型）
2. 知はチカラ（権力、言語行為論への萌芽）
3. 知はミカタ（見方、ある視点での論理的な記述・モデル＝型につながる）

このことが意味するのは、知識は神様か理論が与えた型（モデル）にしたがって構成され、最終的には個人が所有するデータ（モデルのインスタンス）だということである。型や型紙をたくさんもっている者が権力を把握し、それを使って生産したデータを下々に受け止めさせることで相手に命令を発し、人々を働かせ、コンピュータを動かすことができる。（この部分を書いていて思ったのは、この３つ：カタチ・チカラ・ミカタというのは、従来の辞書や百科全書的な知の捉え方に比べれば、よっぽど新しいなぁと感じてしまったこと。そのため、この部分の論旨は若干中途半端になっていることは否めない。とはいえ、所詮は情報や知識のキャッチボールという視点だということで、乗り越えねばらない物語ではあるのです。）

新たな知識「感」の予感

1. 知はカラダ（体をメディアとする歌、言葉、おしゃべり、リズムとダンス）
2. 知はナカマ（関わる・係わる・拘わる、会話と社交、コミュニティ、育成と成長）
3. 知はサワリ（触り / 障り・アフォーダンス => 形式化されない環境知、対話知、けんか知）

では、新しい知の（見方ではなく身体的な）捉え方をしてみよう。場における個人間の情報のキャッチボールといってしまっては逆戻りである。キャッチボールではなく直接接触して触り合う・言葉で探り合う・同じ場に一緒に立つ、ことで環境や空気を共有しあい、お互いがお互いの環境になる、という状況を想定して欲しい。ここには個体と環境の区別が薄まり、全体として１つのまさに texture となり、場そのものがもつ特性 の１つにリ

ズムやその重層化したものとしてコミュニケーションが現れる、というイメージが像を結ばないだろうか。リズムは波であり、新しい意味での形を生み出す。

アフォーダンスという考え方がある。その環境内に置かれた個人が主体的にある個体の意味を認識するのではなく、その個人や個体の埋め込まれた環境自体にその個人の行動やコミュニケーションを誘発するある種のきっかけが埋め込まれている、という知識の配置の変更である。この場合、個人と個体が一般相対性理論のように環境に歪みを生じ、ごく自然に個人と個体との相互作用が生じると看做せるようなコミュニ

ケーション理論を作ることが今後の 1 つの目標となるだろう。

6. コミュニティの編集行為としての思考プロセス

新しい知識の捉え方を前提に、そのような知識の場が成立するだけでなく、4 章で述べたような遺伝子がうまく発現するための場としての細胞を生み出していくためには、コミュニティや社会を次の世代も意識して編集しながら徐々に引き継いでいくという行為が重要になってくる。コミュニティは個人の創造性の発現にとっての細胞環境になっていなければならない。本稿の続編では、その辺に焦点を当てて、「コミュニティの編集行為としての思考プロセス」が考察されることになるだろう。続く。

小論

〔知働化研究会誌創刊第１号〕
知働化研究会
Executable Knowledge and Texture Laboratory

編集と知働化

新時代の知識編集

野口隆史（のぐち たかふみ）

株式会社マナスリンク

概要

編集と知働化の関係について考えていくと、実は編集とは何かを考えてこなかったことに気づきました。編集とは何かを考えつつ、知働化、電子化、アジャイルについて今思うことを書き連ねました。

Copyright 2010, NOGUCHI Takafumi, All rights reserved.

編集とは何か

編集と知働化について書こうとするなら、まず編集とは何かを書かなければなりません。しかし、いくら編集について書こうとしても適切な定義が見つかりません。企画を立てるのが編集なのか、著者を見つけるのが編集なのか、原稿を取り立てるのが編集なのか、校正をするのが編集なのか、組版をするのが編集なのか。結局のところ、本ができるまでのすべてが編集であるとしか言えないかもしれません。

編集は本の主役ではありません。あくまで筆者や主題を引き立たせるための脇役です。黒子は何かと定義しようとしても「黒子は黒子だ」となるように、編集は編集だとしか言えないようにも思います。

本を作る上で筆者ができないことを「なんとかする」のが編集、というのがもっとしっくり感じる表現です。筆者によってできることとできないことは違いますから、当然筆者との関係で編集の役割が決まると言えます。「編集は何をするの？」と聞かれて、いつも言葉に詰まるのですが、著者によってやることが変わってくるので、一言で「何が編集」とは言えないのです。

知働化に期待すること

ここで知働化に話を移します。知働化が何か、というのは私の手に余りますので（編集が何かすら言えないので）、せめて知働化に何を期待するかを述べて、お許しを願いたいと思います。なお、編集という極めて特殊な観点からの意見であることをご了承ください。

・編集が何であるかを解明すること（理論構築）

まず１つめですが、「編集が何か」を知働化で解明できたらおもしろいと思います。編集という言葉をうまく定義できなかったのは、編集という存在をうまく定義できる言葉がたまたまなかっただけかもしれません。もしかしたら、まったく違う分野で同じような事象が起こっているかもしれません。もしそうした事象が見つかったなら、編集の定義に役立つだけでなく、その事象を扱う人と知見を交換できるようになると思います。また、互いを包括するメタ概念の存在が明らかになる（もしくは作り出す）かもしれません。

・編集を助けてくれる仕組みを作ること（直接的な助け）

２つめは１つめと関連したものです。もしメタ概念の存在が明らかになったなら、編集を助けてくれる仕組みを作ることが可能になるかもしれません。編集は地雷の連続です。原稿がなかなか仕上がらなかったり、分量が多すぎたり少なすぎたり、まったく当初と違う内容だったり、著者が急病になったり、行方不明になったり、そうした思いもかけない事象を察知して助けてくれる仕組みがあったらずいぶん編集が効率化されると思います。

・編集のプレッシャーを和らげる仕組みを作ること（間接的な助け）

３つめです。編集はかなり孤独です。あちこち出歩きますし、たくさんの人に会っているため、ずいぶん楽しそうな職種だと勘違いされることが多いのですが、企画を本としてアウトプットするまでのあらゆる責任を負うため、日々相当なプレッシャーと戦っています。編集は書くわけではないからつらいわけがないだろう、ということは決してありません。筆者ができないことが何かわからない中で、そのできないことを、いついかなる時にもやらなければいけないのですから、毎日気が気ではありません。こうしたプレッシャーを和らげてくれる仕組みができてくれると大助かりです。

・気配り（助けと気づかない助け）

４つめは、かなりわがままな要望になりますが、こうしたことを編集がそれとは気づかない形で実現してほしいということです。編集も人間としての尊厳がありますので、尊厳を損なわない形で編集の知働化がなされてくれれば言うことはありません。気配りと言い換えてもいいと思います。

編集する人がいなくなる？

ここで冷静になって考えてみると、これらの要望は実は人間に対して求めたほうが実現の可能性が高いものです。実際これまでの編集の世界は徒弟制で、先輩から後輩に仕事を通じて伝えられるものでした。徒弟制の中で上記の４つを「何となく」できるようになってきたのです。しかし現在、先輩が後輩に編集の仕方を教える余裕はありませんし、そもそも業界の縮小で後輩そのものが業界に入ってこなくなりました。案外本を買う人がいなくなって出版界が途絶える可能性よりも、本を編集する人がいなくなって（現状のスタイルの）出版界が途絶える可能性のほうが高いかもしれません。

そうだとすると編集を知働化するというのは前向きな話ではなくて、縮小する産業をどうダウンサイズして生き残らせるかという後ろ向きの話になってしまいます。

自分で気づかずに行っている振る舞い

ところで先ほど「本を作る上で著者ができないことをなんとかするのが編集」と書きましたが、実は本を作る以外でやっていることがかなりあります。それは「調整」です。とにかく本を作るには調整が多いのです。本という体裁にコンテンツを入れ込んでいく作業も１つの調整です。

それから人の調整があります。著者、出版社、デザイナー、イラストレーター、取次、書店、（雑誌であれば）広告主など、１つの本を作るにもずいぶん多くの利害関係を調整しないといけません。利害関係の調整だけでなく、場合によってはゼロサムではなく、ポジティブサムになるような調整ができるようになることが望ましいです。それがビジョンなのか、夢なのか、言葉の定義はさておき、関係者がわくわくできるような何かを作り出すということが、編集で一番大事なように思います。

こうした振る舞いは、どの業界でも何か（それがものであれサービスであれ）を作り出すときには自然にやっていることが多いと思います。そうした本人すら気づかない振る舞いを価値として存在させるのに知働化が役に立つのではと期待しています。

電子化とアジャイル的な編集の可能性

編集のもう一つの側面が「調整」（場合によっては調整以上のもの）なのではないかと述べましたが、本の電子化は調整の性質を変えていくはずです。紙の本を作って流通させるのには一定のコストがかかるため、どうしても調整にはシビアなリスク配分の色彩が強く出てきます。いつどんな内容のいくらの本を出す、ということを決めてから物事が進むため、おおむね制作作業はウォーターフォール的になります。

しかし、電子化の世界では労力以外のコストは限りなく低くなっていきます。コストが限りなく低い世界では、「いつどんな内容のいくらの本を出す」ことをあらかじめ決めてしまうのではなく、「そのときにあるものをとりあえず出してみて、次を考える」というアジャイル的なコンテンツの出し方が可能になります。物理的な「本」というパッケージにする必要もないので、いつでも改編したり、組み合わせを変えることができます。

今まではいかに本という形に収めるかという収束に向けた調整が大事だったのが、これからは他の何かとつなげて広げていく拡散に向けた調整が重要になってくると思います。

とするなら、電子化の世界はずいぶん Web の世界と近くなっていきます。書籍が Web に吸収されるのか、何か違うものとして存在していくのか、それぞれの編集者が試行錯誤して見つけ出すのが速いか、知働化がそれを見つけ出すのか、そんな競争ができるとしたら楽しみです。

「仕事」としての編集

そもそも編集という仕事が切り分けられて認識されたのはごく最近（明治くらいから）のことです。それまでは著者自身が編集者でもありました。古の文豪が編集者に原稿を取り立てられていたという話は聞いたことがありません。いったん産業という文脈の中で切り分けられた編集という仕事が、電子化や編集・組版ツールの普及によって、また書き手に戻っていくとしたら興味深いことです。

いずれ、コストフリーの世界では「やるかやらないか」が一番大きな差を生みます。出版不況なのは事実かもしれませんが、活字不況なわけではありません。多くの人に今チャンスが与えられ、文字を中心としたコンテンツ作りの世界に大きな変化が起きようとしているのだと考えます。

「仕事かどうかわからないもの」としての編集

私自身は、EM ZERO を中心としたコミュニティ支援（するほうか、されるほうかというのはさておき）活動を行っています。本知働化誌にもそうした立場で参加しています。前項と対比するなら「仕事かどうかわからないもの」としての編集ということです。これからどんな本を作るかというよりは、編集の経験やスキルをどう今後の社会的なコンテキストに置き換えていけるか興味があります。

・EM ZERO での試み

EM ZERO は有志の集まりで成り立っています。書くのも配布するのも有志によって行われています。「何かをやるときに今いる人では手の回らないことを自分がする」のが編集のあり方なので、有志は（場合によっては）自分でそれと気づかず編集的な振る舞いをしています。

実は、周りの人が何で困っているか、を見つけ出してそこに手をさしのべる、という行為は、実はコミュニティでは非常に積極的に行われています。コミュニティの運営スタッフには特に編集的な空気を強く感じます。そのため、コミュニティ支援を「する」つもりが支援「されて」いる状況になることも多々あります。

・知働化誌での試み

また、本知働化誌でも、メンバーの皆様の寛容さのおかげで、様々な試みをさせていただいています。具体的には、互いに原稿をレビューし合うペア執筆や合同缶詰会を行いました。直接的には、互いに締め切りを意識してほしいという目的があるのですが、産業化に伴って分業化してしまった「書くこと」と「編集すること」が、新しい関係を作り出すきっかけの一つになればという意図が背景にあります。

そして、その新しい関係の中で、自分がいったいどんな貢献をできるだろうかという切実な課題に答えていく作業も同時に行っています。

編集とは何なのか、自分は何ができるのか、それをゼロベースで考えられる場作りを、EM ZERO や知働化の活動を通じて実践していこうと考える今日この頃です。

〔プロフィール〕

IT 系オープンペーパー「EM ZERO」編集長。編集という仕事を 10 年続けていますが、未だに編集とは何かに悩む日々です。「仕事かどうかわからない」ものとしての編集の可能性を切り開きたいと考えています。

結

編集後記

入会のご案内

　最終章の「結」は、本研究会誌編集作業のよもやま話を『編集後記』に、座談会形式で述べています。最後は『入会のご案内』です。研究会誌をご覧になって、是非、活動に参加してみたいという方、入会のご検討をいただければ幸甚です。

編集後記

2010 年秋、都内某所駅上にあるマンション８階にて、編集長の N 氏と、運営リーダの O 氏とが、美味い珈琲を飲みながら、まったりと懇談。

〔O 氏〕いやぁ、なんとか完成しましたね。

〔N 氏〕ですね。苦節半年ってところでしょうか。

〔O 氏〕一時はどうなることかと思ったけど、継続は力だね。

〔N 氏〕本当なら昨年度末に完成する予定だったのが、この秋になってしまいましたから。原稿のフォロー、大変だったんじゃないですか？

〔O 氏〕そうだね。メーリングリストで一様にフォローしたんじゃ効果なしだったね。個別にじっくりと、辛抱強くって感じ。この手の書き物の本質的困難は、原稿執筆フォローに有りってところだよ。

〔N 氏〕あの、○△さんや、□△さんの原稿を入手できたというのは、街の出版社さんから見ても快挙でしょう。

〔O 氏〕確かに、締切り日は始める日の人とか、本当の締切り日はいつかとか、ちょっと駆け引きがあったね。そこはそれ、僕もいつもやってるし、手のうち見えてる。

〔N 氏〕最終的な分量もかなりなものになりましたね。全ての原稿が揃っていたら紙ベースで 300 頁程度になっていたかもしれません。Vol.2 へ延期した方もいらしたので、ちょうどよいです。

〔O 氏〕紙版の編集作業をやってみて、200 頁を超えると気が遠くなるね。N さんの普段のお仕事の大変さをほんの少々体感できて、改めて感謝しちゃってます。

〔N 氏〕全体の構成は何か考えがあったんですか？

〔O 氏〕おっ、いいとこついてきたね。実は、編集作業で一番時間かけてるのが、目次なんですよ。当初は、論文と随想とに分けてと考えていたんだけど、出来上がって来た作品とか、皆さんの書きっぷりを見て、論説と小論と改名したんですよ。

〔N 氏〕原稿の到着準で目次作成したってわけでもないんですね。

〔O 氏〕結果として N さんのが最後でしたが. . . 、実はそうではなくて、比較的重いというか論文調のもの、長編のものを論説として位置づけ、それ以外の軽いもの、形式に捕われないものを小論としたんだ。それぞれで後は作品の到着順だよ。

〔N 氏〕頁番号の管理とかからすると賢明な対処です。

〔O 氏〕実は、もう一つ、寄稿という章があるんだけど、ここが 1 作品しかないのが心残りなんだな。知働化メンバの外側で、いろいろ言ってほしいし、他のコミュニティとの交流をして行く上でもキーになると思っているんだ。

〔N 氏〕やはりメンバ外への原稿フォローというのは難しいんですね。

〔O 氏〕知働化の活動を続けて知名度が上がってくれば、書いてくれる人も増えるんではないかなと期待しちゃってます。

〔N 氏〕PDF 版（紙版）の目次のデザインがニョロっとした背景の図案を使ってますが、何か思入れとかあるんですか？

〔O 氏〕時間かけて構成を考えたんで、暇つぶしで描いたデザインを配しただけ。これといって意味は持たせてないよ。目次の 1 頁目と 2 頁目とで上下対称になっているとか、グラデーションのかけ方をトライしたくらい。何の意味もないものに時間をかけるのって、楽しいよ。

〔N 氏〕何か紙へのこだわりっていうのを感じますね。

〔O 氏〕今回の公開は、PDF 版と EPUB 版ということで、PDF 版は紙に印刷するにせよ、端末で見るにせよ、頁が固定されるというところに価値があると思うんだ。一方で、EPUB 版は、もう少々インタラクティブな媒体という位置づけになるね。

〔N 氏〕頁が固定される価値ってどういうことですか？

〔O 氏〕もともと知働化研究誌は、学会や研究機関が発行している論文誌のような役割があると思うんですよ。つまり、世の中の論文や書籍から参照されてなんぼのような位置づけです。「知働化研究会誌 Vol.1 Page 188 記載の◯△」といった参照のしやすさが頁固定の方がやりやすいでしょう。査読はないですが、ある程度、論文誌的な装丁の方が権威付けの効果もあるでしょう。

〔N 氏〕InDesgin の使い心地はどうでした？

〔O 氏〕昔し、PageMaker を使っていたので、さほど違和感はなかったですが、すさまじく機能がてんこ盛りで戸惑いました。最近のツールって、どんどん膨れてどうなってしまうんでしょうね。よく落ちるし。

〔N 氏〕プロはツールのバージョンを上げずに、使い込んでいますよ。素人の方々の方が最新バージョンを使っている逆転現象もおきています。

〔O 氏〕Emacs で TeX 使う方が、精神的にはよいなぁ。数式とかも扱いやすいし。そうそう、EPUB の出力機能が InDesign にあるんですね？ EPUB はどうでした？

〔N 氏〕いわゆる電子本は世の中では騒がれていますが、まだまだ決定打はないです。今回の知働誌電子版は、皆さん提出の Word ファイルに基づいて Sigil というツールで作成しました。

〔O 氏〕ちゃんとフリーツールを使いこなすことろが偉いなあ。とりあえず iPad とかで見れる選択肢が増えるのは良いことです。でも、電子的な読み物は、もっと別な方向へ行きそうですね。

〔N 氏〕小論で書きましたが、「拡散への調整」といったものです。いずれにせよ、ウォータフォール型に執筆して、編集して、印刷してといった流れではなくなり、もっとオープンな相互作用の中にとけ込んでいく方向だと思います。

〔O 氏〕なるほど。ソフトも出版もアジャイル化、知働化しておるということですね。深い。

〔N 氏〕年 1 回、研究会誌を発行するっていうのは、とても良いことです。その時々の皆さんの状況をお伝えできますし、半ば強制的に書くことで、自由研究のペースメークにもなっていると思います。

〔O 氏〕これは、コンセプトリーダの山田さんが、アジャイルプロセス協議会の設立趣意書を作った時に、２ヶ月に 1 回の会合と、年 1 回の研究誌って書いちゃったのがいけないんです。今になって見れば、慧眼ですけど。

〔N 氏〕さすがコンセプトリーダ。

〔O 氏〕そういう N さんも、隠れコンセプトリーダでしょ。もともと、知働化研究会を敷居の高いコミュニティにした方がよいとかアドバイスいただいたし。

〔N 氏〕世の中コミュニティ活動や勉強会が流行していたので、差別化が必要だと考えて、ああいう助言をしたのは確かです。

〔O 氏〕それが功を奏しているように思いますよ。

〔N 氏〕研究誌作品執筆も、敷居高くしてますね。ということで、次の知働誌はどういった方向にしましょう？

〔O 氏〕何度か口にしてるんだけど、新ソフトウェア宣言特集でしょう。

〔N 氏〕横浜開港記念館でやったソフトウェアシンポジウムのワークショップの成果の「呪縛」ですね。

〔O 氏〕そうです。あれはあの時点（2010 年 6 月）のまとめですし、それから皆さん進展もあったと思うので、あたらしい「新ソフトウェア宣言」を中心にして、いろいろ書いてもらおうかと思っています。

〔N 氏〕それに、今回の積み残しの方々もいらっしゃるので、結構な分量になりますね。

〔O 氏〕概ね、Vol.1 の時でメンバの半分の方々が書いた感じになってるので、まだまだ埋もれた作品いっぱいあると思います。それに、今回書いた方々も進展があるでしょうし。○×△さんなんか、作品の最後に「続く」って書いてあるから驚きです。

〔N 氏〕直感で 200 頁超えは間違いないですね。

〔O 氏〕そのうち、投稿査読なんて仕組みが必要になるかも。冗談です。

〔N 氏〕なんか進化し続ける知働誌って感じですね。

〔O 氏〕そうだね、知働誌そのものが、知働化の研究対象の題材とも言えるね。

〔N 氏〕こうして一つの「形」ができると、いろいろ広がりがでてくるような気がします。

〔O 氏〕「知働化」って、ふわっとしたコンセプトだし、あるんだかないんだかわからないけど、何かあるねっていう領域こそ、書くこと、形にすることがとても大切だと思うんだ。

〔N 氏〕編集の仕方、作業の進め方、そして、作品のあり方も進化していくと思いますが、まず第 1 歩を踏み出せた感じがします。

〔O 氏〕そう、最終的には知識情報の世界でのインパクトなり、社会貢献というところに繋がっていく、ちょっとした実験場かもしれないね。

・・・

っと、懇談は延々と続いていました。ではこのへんで、現世に戻ることにしましょう。

結

〔知働化研究会誌創刊第１号〕
知働化研究会
Executable Knowledge and Texture Laboratory

知働化研究会メンバ募集

■概要

本研究会の設立趣旨は「ソフトウェアとは実行可能な知識であり、ソフトウェアが置かれる世界や様相を主題としなくてはならない。不確実性に対応するアジャイルプロセスを発展させ、利用や社会的な相互作用をデザインする手法を探求していく。」としておりますように、IT、システム、ソフトウェア、サービスのビジネスや業務世界でのあり方、利活用、相互作用と、それ等を支えるビジネスやエンジニアリングについて検討していくコミュニティ活動を目指しています。

2009 年７月に有志による準備会合に続き、２ヶ月に１回の割合で会合を開催しています。参加費用は無料です。ただし、アジャイルプロセス協議会への参加を強くお願いしております。

■研究会メンバ募集について

研究会メンバは随時募集しております。勉強会は雨後のタケノコのように有りますが、本研究会は、一風変わったコミュニティです。参加メンバの役割は、

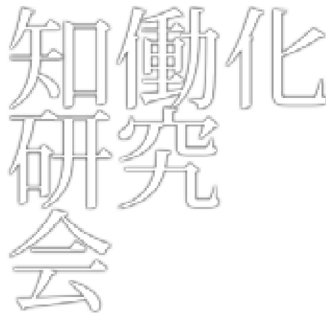
* 自由研究員（自らテーマ設定をして研究や検討を進める人)、
* サポータ、
* サイエンスコミュニケータ、
* エンジニアリングコミュニケータ、
* テクノロジコミュニケータ、
* 啓蒙家

といった方々です。心あるエンジニア、ユーザ企業の CIO、IT を活用したいと思っている農業、商業、流通、金融、保険分野等の広範な方々を対象としています。

本研究会ではさまざまな検討、議論が繰り広げられていくと思いますが、先が見えてない不安感を払拭、IT 技術や新技術を適確に評価、取組み方や考え方についての刺激を受け、人脈開拓といった効用も期待できると思います。また、自社（ドメイン）のコアな知識を IT を使って展開したいとか､開発会社が自らの真の技術を活かしてくれるユーザ（クライアント）に出会いたいといった具体的な課題をお持ちの方々も大歓迎です。

■参考資料について

いくつかの主要な資料は、サイトにアップロードしてありますので、ご覧下さい。



http://www.exekt-lab.org/

ワーキンググループ紹介 .pdf 1093KB - 2009/07/01

準備会合 _20090723.pdf 145KB - 2009/07/19

知働化 FAQ_20090719.pdf 153KB - 2009/07/20

知働化ネーミング .pdf 72KB - 2009/07/30

知働化対談 _20090715.pdf 658KB - 2009/07/25

設立趣意書 .pdf 97KB - 2009/07/01

■申込方法

参加される方は、下記アドレスまでメールでお申込みください。同一会社で複数名参加される場合お一人づつお申込下さい。ご連絡いただき次第、メーリングリストに登録させていただきます。また、参加 / 退会は随時可能です。ご不明のことなどありましたら、申込みメール宛先にお問い合わせください。

申込みメール宛先：info@exekt-lab.org
ご所属：
ご氏名：
協議会会員の有無：（企業会員、個人会員、入会手続き予定、非会員）
メールアドレス（WG メーリングリスト登録用）：
ご要望：

■アジャイルプロセス協議会への参加

知働化研究会は、アジャイルプロセス協議会の一つの WG という位置づけです。協議会への参加を検討される場合には、以下のサイトをご覧下さい。

http://www.agileprocess.jp/

初版 第 3 刷 2010.11.11