ユニケージ手法－高生産性で安価な顧客重視の開発手法

松岡泰史 [1]　　當仲寛哲 [2]
タオベアーズ合同会社 [1]　有限会社ユニバーサル・シェル・プログラミング研究所 [2]

シェルスクリプトを主力言語として流通小売分野向け業務アプリケーション開発で成功している企業がある。この企業は通常の開発手法に囚われず、様々な取捨選択を行っている。アーキテクチャ面では、Intel 系サーバとオープンソース・ソフトウェアを採用し、Linux のファイルシステムを永続性メカニズムにするシンプルなアーキテクチャを選択し低コストを実現している。プロセス面では、顧客指向を実践して中間成果物を削減するためにトレードオフを行い、開発速度を早める工夫がされている。ソフトウェア成果物の大半はスクリプト言語で作成されるので、運用での変化に柔軟に対応する事ができる。本稿では、この本手法の特徴とアジャイル的な側面を紹介する。

Unicage method - A productive,low-cost, and user-oriented development method

Taiji Matsuoka[1]　Nobuaki Tounaka[2]
TAO Bears LLC.[1]　Universal Shell Programming Lab. [2]

There is a successful company that uses shell scripts as its main programming language in developing business application software for retail distribution industry. It breaks all the conventional rules of software development methods to sort out its activities in diverse ways. Through the position of the architecture, it selects Intel-based servers and open source software, and realizes low-cost by using the Linux file system as the persistence mechanism. Through the position of the process, it investigates trade-off between user-oriented activities and intermediate artifacts, and schemes to speed up the software development. Most of its software artifacts are written by script languages, and so will be adapted flexibly to changes in the operation phase. This paper introduces features and agile aspects of its method.

1.　はじめに

本手法は、USP 研究所が流通小売業向けのシステム開発に適用している開発手法である。XP に似た価値観を有するアジャイル的な開発手法であるが、顧客指向の実践、実質的にプロトタイプ作成のフェーズを持つ事、中間成果物の削減、顧客参加のテスト、シンプルなアーキテクチャの採用、シェルスクリプトの重視などの特徴を持つ。

2.　概論

本手法は反復型開発手法であり、流通小売業向けのソフトウェアを素早く作成することを目標にしている。この手法では、シンプルなアーキテクチャ上にスクリプト主体のソフトウェア成果物を作成する事と、業務知識と開発スキルを駆使して少人数開発を遂行する事で開発コストを低減する。また、変化に柔軟に対応できる運用基盤を提供することにより、 顧客（エンドユーザ・情シス）に高い満足度を提供できる。

主な開発者（実態は USP 研究所の担当者）による顧客の仕事場への常駐から作業は始まる。数ヶ月に渡って、主な開発者は顧客とコミュニケーションをとりながら、自分達の持っている技術力や手法をプロトタイプ作成やデモを通して、顧客の信頼を得るように努める。主な開発者はプロトタイピングを行い、常駐する開発者を増員して本開発に備え体制を整えてゆく。プロジェクトが成功すれば情シスと一体化した状態になることができる。

顧客の仕事場（情報システム部門）に常駐している開発者は、情シス（実態は情報システム部員）とコミュニケーションをとり、情シスの頭の中にある業務ワークフローを理解するように努める。 この事により素早いコード作成と少人数での開発が可能になる。

他アジャイル手法と比較して、次の部分が特徴的と考えられる。実現すべき業務ワークフローが既に情シスの頭の中に存在すると考える事、開発スキルを持つ業務コンサルタント的な開発者が、顧客の仕事場への常駐を含めてプロジェクトの最初から最後までプロジェクトに係ってコミュニケーションをとる事、できるだけシンプルなアーキテクチャを選択する事である。

開発でのイテレーション期間は約 2 週間である。本手法は可能な限り中間成果物（設計書、管理ドキュメント）をなくそうとしている。要求仕様書ですら、情シス提供の画面遷移図にミーティングでのメモを加える程度で済ませている。その代わり、実装する機能について疑問点が出た時は、開発者が顧客の仕事場に常駐しているので、顧客と話をして機能詳細を補完することができ、素早くプログラムを作成して仕様を顧客と確認することができる。

この方法は決して杜撰な開発方法ではなく、開発速度とコミュニケーションを重視して、中間成果物の作成をトレードオフ対象にしているだけである。流通

小売業に従事する仕事場に常駐する事による副作用として、顧客本来の業務の周期である1週間という期間に影響を受ける。この周期はプロジェクトに規律を与えている。

本手法の価値は、コミュニケーション、シンプルさ、フィードバック、勇気という四つの本質的な方法でソフトウェアプロジェクトを改善するという XP に近いものがある。しかしプラクティスでは、XP とあまり一致していない。XP の 12 個の基本プラクティスとの比較は後述する。

本手法のプラクティスは、XP の様に相乗的に働くかどうか検証はされていない。しかし、例えば、開発者が文書に依存するのではなく、 顧客とのコミュニケーションに頼るのだと考えれば、後出の開発速度とコミュニケーションを重視して、中間成果物の作成をトレードオフ対象とするプラクティス:「(16)可能な限り中間成果物を排除する」をプラクティス:「(2)少数派として客先へ常駐する」が補完しているように見える。

2.1. ライフサイクル

XP のライフサイクルフェーズを修正した本手法のライフサイクルフェーズを以下に示す。

【S:探索・P:計画】

数ヶ月の間、主な開発者が情シスの仕事場に常駐して、顧客の頭の中にある業務ワークフローを理解するように努める。作成したサンプルをデモすることより、技術力を示し顧客の信頼を確保しながらプロトタイプを作成してシステム全体の形を作る。この間、運用段階以降で、スクリプトを修正できるようにするため、情シスに対してプログラミング教育も行なう。計画は探索の末尾に配置され、リリースの日程は顧客の受け入れ都合により、情シスが決める。

【I:リリースへのイテレーション】

リリースは業務毎に行なわれ、エンドユーザ向けにリリースされる。その後、エンドユーザからのフィードバック(修正または機能追加の要望)が他の業務のイテレーション中に開発者へ届き、対応するという過程が繰り返される。全ての業務を構築し、顧客からのフィードバックが無くなった時、開発が終了する。

業務毎にイテレーションの最初に情シスが提供する資料を元に、顧客を含めて要求のためのミーティングを行なう。必要のない機能は作成しない方針であるが、約 2 週間という短期イテレーションを続けるためには、開発者のハードワークを必要とする。実績によると、プロジェクト全体では最初のイテレーションの開始後、数ヶ月～6 ヶ月程度で終了する。

開発者は約 2 週間というタイムボックスの中で、ひとつの業務の要求、設計、実装、テストを行なう。開発中も情シスの仕事場で作業しながら、顧客とコミュニケーションをとり、開発者間でも随時のミーティングを行なう。テストは顧客と一緒に行なう。

【T:稼動への移行】

ひとつのイテレーションが終了した後、成果物(1 個の業務アプリケーション)はエンドユーザへリリースされる。実際に実務で使用した後、エンドユーザは、修正または機能追加の要望を、情シス経由で開発者へと伝える。開発者のタイムボックスと違い、期間としては 1～2 ヶ月かかる。

【M:メンテナンス】

別の業務アプリケーションを作成している開発者は、現在のイテレーションの中で、情シス経由で伝わるフィードバックへの対応を行なう。

	目的	作業
S	・実現可能性を確認する ・技術力を示す ・運用への布石 ・顧客とのコミュニケーション	・プロトタイプ作成 ・デモ ・プログラミング教育
P	・リリースの日程に合意する	・顧客の受け入れ都合による開発スケジュールの作成
I	・1 個の業務アプリケーションを実装・テストしリリースする	・要求、設計、実装、テスト
T	・実際の運用で業務アプリケーションをテストする	・エンドユーザにより実業務で業務アプリケーションを評価する
M	・修正または機能追加をおこないリリースする	・要求、設計、実装、テスト

図1 ユニケージ手法のライフサイクル

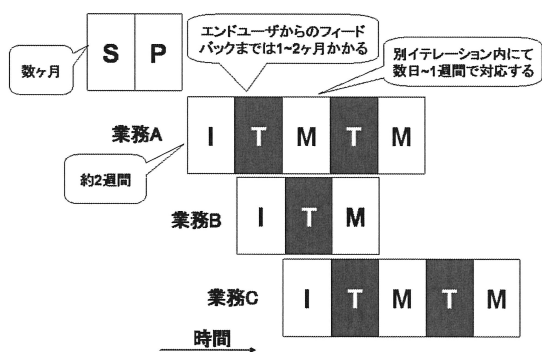

図 2 ユニケージ手法のライフサイクル例

2.2. 作業成果物、ロール、プラクティス

分野	作業成果物
要求	要求仕様書 (情シスから提示された画面遷移表や合意時のメモ)
設計	−
実装	−
テストおよび検証	−
プロジェクトマネジメント	課題解決表 (課題のチェックシート)
構成管理および変更管理の環境	−

図 3 作業成果物(ソフトウェア以外)

開発者は、情シスから提示された画面遷移図や、各イテレーションを開始する際の要求時に決まったメモを加えて、要求定義書を作成する。管理者は他で並行に業務アプリケーションを開発するチームがある場合、課題のチェックシート(実態はバグ管理表)を作成する。この他に開発者が最終成果物の一部として、全体ワークフロー図を作成する。

分類	ロール	作業
顧客	エンドユーザ	・リリースされたソフトウェアを評価する ・テストを行なう
	情シス	・業務ワークフローを伝える ・資料を作成する ・テストを行なう ・リリースされたソフトウェアの評価を伝える ・最終的な受け入れテストを行なう
開発	主な開発者	・プロトタイプを作成する ・最初に顧客と信頼関係を築く ・要求、設計、実装、テストを行なう
	開発者	・要求、設計、実装、テストを行なう
マネジメント	管理者	・開発スケジュールを作成する ・課題解決表のチェックを行なう
その他	講師	・プログラミング教育を行なう

図4 ロール

実際には、顧客の情報システム部門員が「情シス」ロールと「管理者」ロールを兼任している。また、USP研究所の担当者が、「主な開発者」ロールと「講師」ロールを兼任している。

分野	プラクティス
要求	■(2)少数派として客先へ常駐する □(3)顧客が同席するミーティング
設計	■(2)少数派として客先へ常駐する ■(4)随時に顧客からヒアリングを行なう □(9)シンプルな設計
実装	■(2)少数派として客先へ常駐する ■(4)随時に顧客からヒアリングを行なう □(10)チームでのコードの所有 □(11)スクリプト主体のプログラミング □(12)コーディング規約
テストおよび検証	■(2)少数派として客先へ常駐する ■(4)随時に顧客からヒアリングを行なう □(6)顧客が参加するイテレーション毎のテスト
プロジェクトマネジメント	□(5)短期のイテレーション □(7)イテレーション終了後顧客に対してリリースする
構成管理および変更管理の環境	−

図5 プラクティス

「(1)顧客の考え方を学ぶ」プラクティスは共通。複数の分野に重なる場合は前に■が付いている。

2.3. 基本プラクティス

(1)顧客の考え方を学ぶ

コンピュータが普及してから随分時間が経過しているので、現在では流通小売分野の大半の顧客は、ロジスティクス的なシステム構想を持っていて、業務データフロー案も既に出来ている。この認識から、開発者は分析せずに顧客の考え方を理解して、実際の作業をトレースすればシステムは構築できるという発想が生まれる。これが正しければ、開発者は技術指向で活動できる。

このプラクティスはライフサイクルの【S:探索・P:計画】フェーズの活動から開始される。このフェーズでは、主な開発者が情シスの仕事場に常駐して、顧客

から考え方を学ぶ事から始める。ヒアリングと並行して、業務アプリケーションのプロトタイプを作成して、顧客へのデモを続ける。こうして小さな実績を積み重ね、主な開発者の技術に対する顧客の信頼を得るように努める。

プロトタイプ作成を通じ業務システムの全体の形が見えてきた時点で、開発者チームを作って、本格的な開発に入ってゆく。顧客の考え方を理解して、実作業をトレースする必要があるので、開発者は業務とコミュニケーションに長けている。また、開発者は流通小売分野の経験がある方が有利である。

(2)少数派として客先へ常駐する

普通の SIer による開発だと、開発者の仕事場に少数の顧客が常駐するとか、限られた期間で顧客からヒアリングするという形になりがちである。情シスの仕事場に比較的少数の開発者が常駐するという逆転の発想で、顧客が開発に参加する。顧客と開発者が頻繁にコミュニケーションをとれる状況を作り出している。

(3)顧客と開発者が同席するミーティング

各イテレーションの開始時に 2 時間程度の要求ミーティングの時間をとる。このミーティングには顧客が参加する。イテレーション中は随時ミーティングにより、要求や設計の問題点の解決を行なう。

(4)随時に顧客からヒアリングを行なう

情シスの仕事場の中で開発者が作業する。本格的な開発が始まると開発者チームで常駐する。情シスは流通小売業の専門家であり、必ずしも自分で実装はできないが、 構築すべきシステムの業務ワークフロー案を頭の中に持ち、要求と要求の優先順位を決める権限を持っている。情シスはエンドユーザの要望を把握している。

情シスの役割は、自分達の業務ワークフロー案から要求の資料を書いて開発者に説明すること、要求ミーティングに参加すること、実装の優先順位や不明点を明確にすること、随時に発生する開発者とのミーティングに参加すること、業務毎のリリースのためテストに協力すること、リリースされた業務アプリケーションを使用するエンドユーザからのフィードバックを開発者へ伝えること、最終的な受け入れテストを行うことなどである。

エンドユーザ(構築されたシステムの利用者)は、流通小売業の従事者である。役割は業務毎のリリースのためのテストに協力すること、実際の業務でリリースされたシステムを使用して、修正または機能追加の要望を情シスに伝えることである。

(5)短期のイテレーション

本手法は、顧客の活動に合わせた約 2 週間という短期のイテレーションに主眼を置いている。このイテレーションに合わせるため、開発者は顧客の仕事場への常駐や時間外労働を厭わない。結果的にハードワークになる。USP 研究所では、開発者のハードワークが報われるように、報酬・自由時間・やりがい等について試行錯誤している。

(6)顧客が参加するイテレーション毎のテスト

イテレーション毎に開発者と顧客が協力して行なわれるテストが、成果物に対する受け入れテストになる。このテストとは別にプロジェクトの終了時、情シスが独自に最終的な受け入れテストを行う。

(7)イテレーション終了後顧客に対してリリースする

約 2 週間の開発イテレーションでのリリースを繰り返す。1 回のイテレーションで1つの業務システムをリリースする。リリースされた後は、エンドユーザが実際の業務で使用して評価する。

(8)顧客からのフィードバックを別のイテレーション中で反映する

エンドユーザからの評価は、情シスから開発者へフィードバックされる。修正または機能追加の要望は、正式なフィードバックとして情シス経由で開発者に伝えられ、別のイテレーション中でプログラムが修正されてリリースされる。この過程は、エンドユーザの修正または機能追加の要望がなくなるまで繰り返される。開発者は常駐しているので、顧客からの非公式なフィードバックは、昼食時等でも随時受け付ける。

(9)シンプルな設計

各イテレーションでは、エンドユーザが必要とする最低限の機能を要求で定義する。将来必要となる機能を予想して設計しない、今すぐ必要で無い要素を作成しない。エンドユーザへリリースされて変更要望や不満点がフィードバックされてから開発者は必要な機能を追加する。

アーキテクチャ面でもシンプルで安価な選択をしている。特徴は、今や日用品となっている Intel 系のコンピュータと Linux 系オープンソース・ソフトウェアを活用している事である。主な構成例を以下に挙げる。

- Intel 系サーバ
- Cent OS (Red Hat Enterprise Linux 互換)
- Apache (Web サーバ)
- bash (コマンドインタープリタ)
- awk (文書処理ツール)
- GNU コマンド群 (cp,find,uniq,sort 等)

・USP オリジナルコマンド群(40~50 個)*[1]
・プロジェクトで開発する専用コマンド群(20 個)*[1]

永続性メカニズムはリレーショナル・データベースではなく、Linux ファイルシステムを使用している。各種コンピュータ・リソースの進化の為、Intel 系サーバ、Cent OS、bash、GNU コマンド群、USP オリジナルコマンド群という低コストでシンプルなリソース選択を行い、高速な処理を実現している。

各イテレーションでの設計は、データベース用途で使用するファイルのデータ構造を決める事が主な作業となる。開発者が定義するファイルのデータ構造は、顧客の業務に直結している内容なので顧客には理解できる。開発中、コマンドから出力されたファイルを見せて顧客に確認することができる。また、開発者は業務アプリケーションで必要となる専用コマンドの仕様をまとめ、主な開発者へリクエストする。リクエストを受けて、主な開発者が、主に C 言語で書かれた専用コマンドを追加する。

(10)チームでのコードの所有

本手法では「全有」と呼び、集中管理せずにチーム全員で全てのコードを所有する。

(11)スクリプト主体のプログラミング

シンプルなアーキテクチャを選択しているので、本手法でのソフトウェア成果物の大半はシェルスクリプトと HTML ファイルである。残りは、主に C 言語で書かれたプロジェクト専用コマンドである。

少人数で開発作業を行ない、実装されたプログラムのテストは手動で行なう。コードは開発者全員が個別に全コードを保有している。バージョン管理は行なわず、開発者間でファイルのコンフリクトがある場合は、手動で対応する。スクリプト主体の開発であるため、大規模なビルド作業は発生しない。開発成果物の多くがプログラムの実行以前にコンパイルされたバイナリ・コードではない。にもかかわらず、業務アプリケーションが稼動する。

業務ロジックを記述するシェルスクリプトは 1 個のファイルが 10 行から 100 行程度で入出力はファイルで行なう。プログラミングでは、bash の機能を使用したパイプ処理やリダイレクトが多用される。 ユーザインターフェースを記述する HTML ファイルでは、HTML の他に JavaScript、CSS、Ajax 等のスクリプト言語や技術を使う。

(12)コーディング規約

主な開発言語であるシェルスクリプトでのプログラム作成を促進する為の規約がある。特徴的な規約としては、1 コマンド 1 行の原則、入出力でファイル渡しの原則、考えて悩むより良いプログラムのコピーの奨励、バックアップは取るがバージョンを作成しない事、修正パッチを充てずに作り直す事などがある。規約というよりも、原則や作法に近い。

2.4. その他のプラクティス

(13)顧客に対してプログラミング教育を行なう

本格的な開発に入る前の【S:探索・P:計画】フェーズにおいて、将来、業務アプリケーションを修正できるように、講師が情シスに対してシェルスクリプトのプログラミング教育を行なう。この中で Unix 系のエディタである vi や、テキスト処理コマンド awk に関する教育も行なう。本来、運用段階での効果を期待するものであるが、副次的な効果として、受講以降の顧客参加ミーティングやデモでの反応が良好になる。

主な開発言語はシェルスクリプトであり、オブジェクト指向言語が採用されていない理由は、色々な概念や技術の理解が必要となり、顧客への教育が難しくなるのが一因である。

(14)余計な機能を排して基本機能を実装する

顧客は基本機能だけを求めている。業務単位で実装・テストの後、業務アプリケーションをリリースする。その後、開発者は別の業務のイテレーションを始める。顧客から修正または機能追加の要望のフィードバックが来てから対応する。

(15)コピー元のリファクタリング

本手法では開発速度を優先にしているため、原則、リファクタリングは行なわない。但し、プログラミングで迷った時、10 分間考えて分らない場合は他コードからのコピーを奨励している。このため、コピーされる元になる模範コードに対してはリファクタリングを行なう。

(16)可能な限り中間成果物を減らす

本手法では顧客の要望に応じて早く作成するという目標がある。このため、プログラミングが優先され、必要でない中間成果物は作成しない。しかし、複雑な処理のための仕様書や課題解決表のような開発者自身が必要となる文書と、顧客が要求する文書は作成する。

(17)測定基準を持つ

プログラムを実行する際にはレスポンス時間を 1 秒以下に収めるという目標がある。これ以上かかると何らかのチューニング作業を行なう。

(18)作業環境を整える

常駐先が情シスの仕事場といえ、開発のためには最低限の作業環境を整える必要がある。例えば、以

[1] ＊印はコードが公開されていない

下のような工夫がある。シェルスクリプトの開発で守っているコーディング規約を壁に張り出す。 ミーティングでは絵や表を多用して、解り易すい説明を心がける。ホワイトボードの利用率を高くして、付属プリンタやデジカメで内容を記録する。

(19)進捗を管理する

少人数で開発を進めているので、進捗状況は毎日の随時ミーティングで共有されている。本格的に開発が行なわれる場合、他で並行に業務アプリケーションを開発するチームもあるので、管理者は課題のチェックシートを作成して管理する。

(20)業務単位での見積もり

業務単位で見積もりを立てている。本手法での見積もりは短く、約 2 週間で見積もられている。イテレーション期間に直結している。

(21)10 分考えて分らなかったらコピーする

実装中、10 分考えて良いアイデアが浮かばなかったら他コードからコピーする。コピーによりプログラミング作業での平準化生産性を図り、開発者の技術レベルが同じになるようにする。

(22)確認の手順

利害関係者の間で、仕様などの確認の際に使うコミュニケーションのプラクティスである。仕様などを伝える人が説明を行なった後、聴いた人が逆に伝える人へ説明する。その後、伝える人が聴いた人へ説明が正しいかどうかを知らせる。

(23)廃棄性を重視する

基本的に再利用しない、すぐに作れるので捨てることに躊躇しないというプラクティスである。例えば、業務アプリケーションの専用データに対しては、間違いを蓄積しかねないので毎日消してしまう。廃棄することで、過去に引きずられる事を回避している。

2.5. 価値

本手法の価値は XP に近く、以下の四つの本質的な方法でソフトウェアプロジェクトを改善する。

2.5.1. コミュニケーション

本手法では開発期間中の顧客の職場への常駐を基本にしている。常駐は少人数で行われるため、顧客数と比べると少数派になることが特徴である。日本社会特有かもしれないが、この形態により顧客との随時のコミュニケーションが期待できる。常駐する開発者が要求も行うので顧客からのヒアリング機会は多くなる。開発者間のコミュニケーションは、二人一組のプログラミングや毎日の随意ミーティングにより促進される。イテレーション毎のテストに顧客が参加することでもコミュニケーションが活発になる。

2.5.2. シンプルさ

本手法はできるだけシンプルな要求を素早く実装する開発手法である。エンドユーザにできるだけ早くリリースして、フィードバックを貰い修正または機能追加する事を繰り返す。実装しても使わないような不要な機能は排除される。シンプルさの追求は色々なものに適用されている。例えば、開発組織を小さなものにする。毎日の随意ミーティングによりプロジェクト管理を簡単にする。アーキテクチャを複雑なものにしない。開発期間を短くする為、中間成果物もできるだけ作成しない。実装でもコードの1行単位で処理を明確にする事やパイプ処理の多用や 10 分考えて分らなかったらコピーするというような工夫で作業を簡単にしている。

2.5.3. フィードバック

本手法では、【M：メンテナンス】フェーズで情シスがフィードバックを反映した要求を出す。開発者は情シスの仕事場に常駐しているため、情シスは開発者の力量を理解できる。開発期間は流通小売業の業務の周期である 1 週間に納まるボリュームの要求に落ち着く。イテレーション毎の成果物はエンドユーザへ直ちにリリースされるので、開発者は以降のイテレーション中にフィードバックが得られる。顧客は部分的にシステムが構築されてゆく様を観察・体感でき、頭の中にある業務ワークフローをより明確に要求に反映させたり、修正したりすることができる。

2.5.4. 勇気

本手法の場合、「勇気」以前の 3 つの価値は持っているので XP 的に考えると、早く開発し、早く変更するという勇気は実践可能である。「仕様は顧客の頭の中にある」、「顧客は基本機能しか欲していない」と考える謙虚な開発手法なので、一般的な開発者が頭を切り替えるには勇気が必要である。また、プラクティス「(14)余計な機能を排して基本機能を実装する」は一種の楽観論にも思えるので、実行には勇気が必要である。

3. ウォーターフォール型開発と本手法の比較

以下、ウォーターフォール型開発と本手法の比較を2つの観点から行う。

3.1. 要求にかける人数や時間について

ウォーターフォール型開発の開発スタイルを踏襲する一般的な SIer では、限られた期間で、顧客の代表者から複数のコンサルタントがヒアリングを行い、ビジネスモデルや業務ワークフローを抽出・分析し、分析結果の成果物をまとめる。その後、結果を基にシス

テムの要求に入り、上級システムエンジニアがシステム開発するための要求の成果物をまとめ、顧客の代表者から承認を貰う。残りの開発期間でシステムの分析・設計・実装・テストを行なう。リスクを軽減するために、本番の開発前にプロトタイプ開発の機会が与えられる場合もある。

普通は限られた期間でのヒアリングになるため、要求への入力が顧客の表現力やコンサルタントの能力に依存してしまう。顧客の職業は各業務であり業務コンサルティングやシステム開発ではないので、上手く自分達の希望や必要な情報を表現できない場合も考えられる。顧客の表現力を補完するのもコンサルタントの仕事である。また複数のコンサルタント間での意見の相違点を調整する必要もある。コンサルタントは高単価なので要求の期間を過ぎるとプロジェクトから居なくなってしまう。

顧客発の要求は伝言ゲームのようになる。例えば、以下のようになる。

顧客⇒分析者⇒設計者⇒実装者⇒テスト担当者⇒顧客

大規模になるほど分業され、各役割が専門の担当者に任される傾向にある。担当者は自分の観点から要求を選択し実装してゆくが、各担当者間の情報伝達は口頭での説明と文書の成果物及びプログラムになる。直前の担当者の成果物が重視される傾向がある。開発規模が大きければ開発組織が階層化されるので、ヒアリングされた顧客と面識のある開発側担当者はコンサルタントか上級システムエンジニア(上記の例では分析者)のみという可能性もある。開発に参加する顧客側の利害関係者の数は少ない。

本手法では、ウォーターフォール型開発とは発想が違う。現在対象としている流通小売業では、日用品レベルまでコンピュータ・システムが普及した結果、必ずしもすぐに開発者(前出の「主な開発者」ロールや「開発者」ロール)が理解できる形式では無いが、ビジネスモデリングや要求が既に顧客内で終わっていて、どの業務データを組み合わせて、何処へ出力するなどの業務ワークフローに落とせるレベルまで分析が終了している顧客が大半であるという認識から出発している。スピードを重視して、顧客の状況が変わらない内に、顧客側からヒアリングし作成したプログラムを業務システムへ組み込めば良い。

本手法の開発スタイルでは、開発者チームを開発期間の大部分で顧客の現場での常駐にあてる。期間中、開発者は自分でシェルスクリプトによる実装を繰返し、現場の顧客から実装に対する評価を貰い、実装へのフィードバックを繰り返す。開発者は必要以上に中間成果物を作成せず、顧客が考えている要求をシェルスクリプトへ変換する。

開発者は、活動の主眼を顧客の業務や社風の「理解」に置いている。既に業務データフローなどの分析結果は顧客内に存在しているので「分析」ではなく「理解」である。開発者は、顧客側が既に持っている分析結果を踏襲して実装できるまで、常駐を継続し開発を行う。顧客内でも新システムに対する理解度は一枚岩ではなく、認識不足や理解不足になっている人も居る。組織の階層に沿って、順にヒアリングしながら構想を解説する場面もある。主な開発者は後続の開発にも加わり開発の最後まで常駐する。

開発者の数が少ない反面、比較的長期に渡り常駐する事で、結果的に業務や社風を「理解」するという目的に足りる顧客数からのヒアリングが可能になっている。

3.2. リスク

ウォーターフォール型開発だが、本番の開発前にプロトタイプ開発の機会が与えられる場合について書く。普通はプロトタイプ開発を行うことにより、アーキテクチャ的な問題点の洗い出しと解決ができるのでアーキテクチャ的なリスクは軽減される。プログラム面では「今回、この機能が動かないと意味が無い」という機能をプロトタイプ開発で解決すれば、リスクがかなり軽減される。ウォーターフォール型では本番開発の結合テスト時にならないとリスクが減らないが、何事も起こらなければ上手く回る筈である。規模が大きく諸事情でプロトタイプ開発ができない場合、かなり危険なプロジェクトになる。混乱が起こるのは本番の開発の際に、SIer 側の事情など考慮しない顧客の一部からアーキテクチャ的な追加・変更や設計的に大きな変更が必要な機能が追加される場合である。プロトタイプ開発で解消すべき事柄が、未解決な状態で新たに追加される為、プロトタイプ開発の効果が減ぜられる。ウォーターフォール型開発では、開発期間が比較的長期に渡り、開発体制は軍隊モデルなので、あらゆる要員の交換が有り得る。このためか各要員への依存度を低レベルで保とうとする。リスクが問題に変わる最中、プロジェクト初期に関わった主要な利害関係者(PM、コンサルタント、アーキテクト)が途中で居なくなる場合もある。要員への依存度を低くする為に中間成果物を作成するとも考えられる。募集時期や報酬が原因で、SIer 側の希望するスキル・レベルの要員が集まらないというリスクもある。

本手法では、アーキテクチャ的には Intel 系サーバ、Cent OS、Apache、bash を使用するだけなので、かなり安定していて問題が発生する可能性は少ないと考えられる。開発方法論的には反復型開発に分類され、1 回のイテレーション毎にプログラムの一部が完成し、実際に運用されるのでリスクは減ってゆく。何が起ころうとも、開発期間は数ヶ月～6 ヶ月程度と短い

し、開発での担当範囲は大きいが、基本的に少人数開発なので、もし人月コストの観点で考えるならリスクは小さい。しかし、顧客の作業場に常駐する開発者への依存度が高いので、病気など何らかの理由でシステム開発から離脱すると開発自体が頓挫してしまう事が考えられる。また、開発者に要求される事柄が多いのでビギナーではスキル不足に陥ることもある。顧客とのヒアリングに参加する予備の開発者をアサインすると、このリスクは軽減できる。USP 研究所の現状では、上位者とビギナーによる組み合わせでペアとして常駐させ、教育的な側面も含め、リスク回避としている。本手法は、現在の流通小売分野の大半の顧客は、ロジスティクス的なシステム構想を持っていて、業務データフロー案も既に出来ているという性善説に立っている。顧客の発言が始終一貫している場合は問題ないが、そうでないなら責任範囲が明確でなくなる可能性がある。将来、適応業種が広がる際には、仕様を詰める必要があるかどうかで業種を選ぶ必要がある。また、本手法が、顧客参加が必要であり、負荷のかかる開発手法である事は、予め顧客側に認識されるべき事柄である。

4. XP と本手法の比較

以下、XP の基本プラクティスと本手法の比較を行う。

(a)計画ゲーム

行なう。【S：探索・P：計画】フェーズでは、リリースについて計画し、この中でイテレーション内にて行なう作業の計画もおこなう。

(b)小規模で頻繁なリリース

行なう。前出「(5)短期のイテレーション」を参照。

(c)システムのメタファ

本手法では行なわない。

(d)シンプルな設計

行なう。情報系、業務系、整理系という形でシステムを分けている。前出「(9)シンプルな設計」を参照。

(e)テスティング

本手法ではテスト駆動型開発は行なわない。前出「(6)顧客が参加するイテレーション毎のテスト」を参照。

(f)頻繁なリファクタリング

行なう。【S：探索・P：計画】フェーズで、リファクタリングを行い優れたサンプルコードを作成する。前出「(15)コピー元のリファクタリング」を参照。

(g)ペアプログラミング

本手法ではペアプログラミングは行なわない。しかし、二人一組とし、上位者がビギナーを指導しながらプログラミングをしている。

(h)チームでのコードの所有

行なう。前出「(10)チームでのコードの所有」を参照。

(i)継続した結合

本手法では大規模なビルド作業は行なわない。

(j)持続可能なペース

本手法では行なわない。顧客は素早い開発を求めているので、コミュニケーションをとりながら素早く作り上げる。

(k)チーム全体が一緒に

行なう。前出「(2)少数派として客先へ常駐する」を参照。

(l)コーディング規約

行なう。前出「(12)コーディング規約」を参照。

5. 結論

本手法のプロセス的な側面では、顧客が業務ワークフローを持ち、それを開発者が実装する形式なので顧客の満足が期待できる。少数の開発者が始終、顧客へ常駐するアジャイル的な反復型開発を行なうので効率が良い。また、対象業務を理解できる開発者がシステム構築すれば中間成果物を減らすことができる。アーキテクチャ的な側面では、低コストでシンプルなアーキテクチャによるシステム構築が期待できる。加えて、永続化メカニズムに Linux のファイルシステムを使用して、システムを作ることができる。主な開発言語であるシェルスクリプトは顧客にも理解でき、開発成果物の大半がスクリプトであってもシステムは稼動する。

参考文献

1)當仲寛哲「シェルスクリプトによるプロセス指向型開発手法」,『SEA Forum April 2007』

2)グレーグ・ラーマン『初めてのアジャイル開発』東京、日経 BP 社、2004 年

3)スコット・W・アンブラー『アジャイルモデリング−XP と統一プロセスを補完するプラクティス』東京、翔泳社、2003 年