システム開発の３つの Tips

二神　明治：株式会社クロスフィールド

1. はじめに

システム開発はエンジニアリングである。システム開発の技術者をシステムエンジニアと呼称するように、エンジニアリングでなくてはならない。エンジニアリングである以上、要求仕様と合致したものが逆算の生産工程を経て当たり前のように生産されるべきだろう。だが、エンジニアリングとしてのシステム開発の成熟度はとても低い。建築・建造や精密機械などの他領域とは比べるべくもない。アジャイル開発だ、ローコード開発だ、ＲＰＡだなどと言っても、未だに前近代的な工場制手工業の域に留まっているのが実態と思える。手工業であるがゆえに、品質や生産性に及ぼす属人性の影響は無視できない。システム開発に携わる一人一人の振る舞いや創意工夫の積み重ねが、開発全体の結果を左右する。とりわけプロジェクトを統括する立場にある者の影響はとても大きい。

私はコンサルタントに転身するまでは、システムエンジニアとして日々システムの開発・保守運用に没頭していた。自身のシステム設計やプログラミングの腕前は悪くなかったと自負するが、開発リーダーとして管理に専念したプロジェクトであっても、担当したいずれの開発も標準値を大きく上回る生産性と高品質（この場合は初期不具合の発生率の低さ）を達成できた。計数管理上は異常値に分類されてしまい、お客様の品質管理部門から監査を受けたこともあったが、当時は自分でも理由がわからず、説明に窮してしまったこともある。今でも十分な説明ができるかわからないが、当時を振り返って特に留意、工夫していた事項を３つの Tips としてお伝えしたい。現役のシステム開発者だけでなく、システム開発の経験なしにシステム導入プロジェクトを推進しているユーザー企業の方や若手コンサルタントにとっても参考になれば幸いだ。

2. システム開発の３つの Tips

（１） 其処彼処のコミュニケーションギャップこそ最大の敵

システム開発は伝言ゲームの如しである。

システム導入を一度でも経験した方ならば、ユーザー企業とシステム会社（開発を内製している場合は業務部門、システム部門と読み替えて頂きたい）との間にコミュニケーションギャップが生じやすいことは実体験をもってイメージしやすいだろう。いわゆる要件定義の不備である。ユーザー企業の業務担当者とシステム会社の技術者はお互い異なる領域の専門家同士なので、前提知識も先入観も異なって当然である。コミュニケーションギャップが生じやすいのは必然に思えるが、誰しも往々にして、知らず知らずに相手に「1 を聞いて 10 を知る」ことを期待しがちのようだ。とくに発注側であるユーザー企業側、とりわけシステム導入経験がない方が初めてプロジェクトをリードするときに顕著に表れるように思える。20 年前にコンサルタントに転身してから頓挫プロジェクトの再生を何度か担当したが、その経験ではプロジェクトの頓挫は、途中で失敗したというよりは要件定義でのボタンの掛け違えがテスト工程以降で露呈するケースが典型に思える。要件定義の際にユーザー企業側がシステム会社の担当者を業務の門外漢か、せいぜい半可通くらいに認識して優しく丁寧に説明していれば、もう少し円滑なプロジェクトになったこと

だろう。もちろん受託者である以上、道義的にはシステム会社側の責任のほうが重い。仮にユーザー企業側から要件定義らしきものが提示されても、エンジニアリングの観点で言えば後工程である製造工程が意識されていない以上、要件定義っぽいものでしかない。要件定義のインプットとなる要求分析といったところだろう。システムエンジニアはエンジニアリングの観点、すなわち後工程に齟齬なく漏れなく伝達できるよう要件定義を行い、要求分析の行間にスキマがあればきちんと確認および補完を行う責務があるはずだ。しかるにコスト削減のつもりかわからないが、要求分析を受領後、要件定義フェーズが省略されて、いきなり設計フェーズに突入したとしか思えないプロジェクトを見聞きすることは少なくない。

実は深刻なコミュニケーションギャップは、設計者とプログラマー間というシステム技術者内でも生じやすい。多くのエンジニアリング業界では設計工程から製造工程への伝達は主に設計図面で行われる。一方、システム開発の場合、設計者からプログラマーへの仕様の伝達は主に自然言語で書かれた文章で行われる。これを仕様書という。自然言語であるがゆえに図面と比較して、圧倒的に読み手の解釈に幅を持たせる。しかも、（少なくとも私がシステムエンジニアであった当時の日本においては）総じてシステム技術者は文章力、読解力はあまり高くなかった。平均的なプログラマーに１０を説明してようやく７くらいを理解してもらえるイメージだ。なので仕様書の記載も歩留まり７割を前提にわかりやすくせねばならない。１０を理解してもらうためには１４の説明が必要なのだ。

私は各開発フェーズのアウトプットをレビューする際は、受け手である後工程の担当者のつもりで通読するよう留意した。基本設計書の場合は詳細設計者として悩むことなく詳細設計できるか、詳細設計書の場合はプログラマーとして悩むことなくプログラミングできるかという風に。あいまいな表現やわかりづらい表現があれば当然修正させる。理解が難しい複雑な機能があれば、イメージ図などで分かりやすい例示を追加させる。また詳細設計者の意図をプログラマーに齟齬なく伝えるには、詳細設計者が単体テスト仕様書を作成して詳細設計書とセットでプログラマーに提示する方法はかなり有効であった。一般には単体テスト仕様書はプログラミングの後にプログラマー自身が作成することが多いように思うが、プログラマーがそもそもの仕様を誤解している場合は意味をなさない。その点、詳細設計者が単体テスト仕様書を作成すれば明確に自分の意図をプログラマーに伝達できる。自然言語で書かれた詳細設計書に解釈の幅が生じえても、テストケースと対応するテスト結果を記載したテスト仕様書に解釈の幅は生じえない。

（２） 品質は前工程で作りこむ

覆水盆に返らず。テスト工程以降で低品質を挽回するのは至難の業である。

開発管理にとって仕様の手戻りが一番恐ろしい。後工程に行くほどダメージが大きい。システムは目に見えないので一般の人にはイメージしづらいのかもしれないが、タワーマンションなどの大掛かりな建築物を想像してほしい。基本的な設計の段階では、いくら見直しが入ろうとも消費される時間と労力は限定的である。少人数の作業であるし、しょせん机上である。いくらでもやり直しはきく。これが後工程、例えば建築予定３０階建て中の２０階までくみ上げた段階で、強度設計に致命的な不備があることが判明したらどうか。当然、工事のやり直しであり、今までの作業はチャラとなる。もし仮に手戻りを嫌って、パッチ当てのような補強作業を行いつ

つ30 階まで建築が強行されたなら、そんなマンションに安心して住めるだろうか。そんなバカげたことがシステム開発では往々にして起きている。開発管理者としては、人員数も納期も見直されず、機能要求だけが修正されるならば低品質に帰結しても是非もなしだ。

システム会社によって呼称などは異なるが、ウォータフォール型開発は、要求分析（構想立案ともいう）→要件定義→基本設計→詳細設計→プログラミング→結合テスト→総合テスト（運用テストともいう）→受入テストの工程で構成される。当然、仕様の不備なりミスなりの低品質は、後工程で発見されるほどやり直しの範囲が広くなり、時間×労力=コストも大きくなる。逆に言えば低品質は前工程で摘出するほど低コストで済む。この鉄則を忘れたかのように、要件定義や設計フェーズに十分な人員と時間を投入せず、早め早めにプログラミングに突入したがっているようなプロジェクトが少なくないように感じる。システム開発は「急がば回れ」が肝要だ。

余談であるが、急がば回れと言えば、私の経験上、総合テスト段階で露呈した低品質をバグ FIX 程度のパッチ当てで挽回するのはまず不可能である。あてどないデスマーチが始まるだけだ。速やかに開発工程をさかのぼって精査し、致命的な問題が生じた工程を特定し、そこから改修範囲を定めるべきだ。またシステムの不具合には偏在性がある。チーム全員が低品質を生み出しているケースよりも特定のメンバーがやらかしている可能性も十分ある。総合テストで低品質に直面した場合は不具合の偏在性をモジュール別や設計者別、プログラマー別などで分析してみる価値はあるだろう。特定の分類に偏っていた場合は、そこを集中的に再レビューすれば効率的だ。

（3） ピープルウエアを重視する

ＩＴ技術者もしょせん人の子である。

システムの構成要素としてソフトウエア（プログラム等）とハードウエア（コンピュータ等の機器）という単語は有名だろう。これに対比させて、システムを使う側（つまりはユーザー）の人間的側面を差してヒューマンウエア、システムを開発する側の人間的側面を差してピープルウエアというそうだ。

システム開発は手工業、つまりは人力である。また個々の技術者の能力には個人差が大きく、監督者の采配次第で成果が左右される余地が大きいことはスポーツチームの様かもしれない。超優秀なプログラマーの生産性は 10 倍とも 100 倍とも言われるが、私の経験でもプログラマーを１０ 人も集めれば、上位者と下位者で生産性が数倍違うことは珍しくなかった。プログラミング工程の品質と生産性を追求すると、開発メンバーの個人差をはじめとするピープルウエアを考慮することに帰結する。

難易度の高いモジュールは能力の高い技術者をアサインし、能力の低い技術者には簡単なモジュールを割り当てるのは当然の戦術だ。肝心なのは個々人の能力を早期に的確に把握することだ。能力が把握できていない当初のうちは、難易度中か低のモジュールを全員にアサインし、生産性とか成果物の出来栄えなどを観察する。もしメンバーの中にエース級が見つかれば、かなり戦いやすくなる。エース級には、余裕のあるスケジュールで高難易度のモジュールを割り当てるのに加えて、開発チーム全体の品質の底上げするようインスペクションを主業務として担当させる。インスペクションとは、他のプログラマーが作成したプログラムコードを机上レビュ

ーして不具合や問題点を指摘したり、可読性・保守性の観点で修正・改善をアドバイスしたりすることだ。私の経験では、単体テストを行う前にインスペクションを行うことは、結果的に品質と生産性の両方を大きく向上させた。往々にして初版のプログラムを書くよりも単体テストを行い、不具合があったらプログラムを修正して、再度単体テストを行うということを繰り返している時間のほうが長くなりやすい。特に未熟なプログラマーほどプログラム作成→単体テスト→不具合修正→再テスト→不具合修正→・・・の長いループに陥りやすい。適切なインスペクションはこうした無駄を大いに削減する。やはりシステム開発は「急がば回れ」である。

能力に応じたアサインメント以外にも、コピー率を意識した開発計画など工夫したことは他にもあるが、私の管理スタイルとして最も特徴的だったのは「残業させない」ことだったろう。このことは私の中では高い生産性と高い品質を達成するには重要事項である。システムの設計やプログラミングは集中力を要する高度に知的な活動である。朝から全力で頭を動かしていれば、8 時間後には脳が疲弊しきっているはずだ。脳が疲弊した後の 3 時間のスループットなど朝の 1 時間のスループットにも満たない。また脳が疲弊しているのだから、ケアレスミスも起こしやすい。ケアレスミスも品質管理の大敵である。3 時間残業するくらいなら翌日 1 時間早出するべきだ。開発技術者は疲れ知らずの機械ではない。

3. おわりに

上記３つの Tips はウォータフォール型開発の全盛期に培ったものなので、最後に昨今主流になりつつあると言われるアジャイル開発でも当てはまるのか持論を述べて終わりたい。

アジャイル開発は少人数のチームで行われる。これを単に少数精鋭主義と捉えても、うまく機能しないだろう。代表的なアジャイルの方法論であるスクラムでは業務側とシステム側を合わせても３人から 10 人が適切なチームの人数とされている。この人数は全てのメンバーが当事者意識をもって参画し、相互に密なコミュニケーションがとれるチームサイズの経験値らしい。アジャイル開発においても「コミュニケーションギャップこそ最大の敵」であるという点は変わらない。むしろコミュニケーションギャップを最小化するための手法とも思える。

アジャイル開発でも「品質管理は前工程で作りこむ意識」を忘れてほしくない。アジャイル開発では開発全体を小さいスコープに分割し、優先度の高いスコープから順に開発→リリースを繰り返す。開発を小分けにしているので手戻りの影響は最小化されてはいるが、小分けされた開発内にはやはり要求分析→要件定義→設計→製造→テストの工程が存在している以上、いつでも仕様変更し放題では収拾がつきそうにない。

「ピープルウエアを重視」すべき点も共通だろう。ただしアジャイル開発は少人数で行われるので、メンバーの能力差を考慮した采配の余地は少ない。代わりにチームワークの醸成とか、多様性の尊重とか別の観点のピープルウエアを重視する必要があるだろう。