2009/01/02

開発方法論

1. 開発方法論

システム開発の一連の手順を開発方法論として、色々な考え方が示されています。

このシステム開発方法論を一言で言うと、いかに伝言ゲームに勝利するかと言う事だと思います。

開発方法論における作業局面としては、一般的に以下の4つになります。

① 業務分析・業務設計

② システム設計

③ プログラム開発

④ テスト・移行

「業務分析・業務設計」は、先ず、システム開発における Why と What を明らかにします。

例えば、新しい事業や新商品に対応する場合は、新たな対応自体が Why と言えますが、現行業務の改善を図る場合などには、現行業務自体の分析から、問題提起の作業が Why への対応となります。

また、新しいマネジメントシステムや管理手法の導入が中心となる場合には、現行業務分析に加えて、新しい手法の導入によって、何が変わるのかを見極める事が Why への対応となります。

前者が発生型の問題への対応、後者は設定型の問題対応と言う事が出来ます。

プロジェクトを進める内に、手段が目的化してしまい、開発コスト削減を目的としたプロダクツの適用に、開発コストを遙かに超えるカスタマイズ費用を投入したり、作業の合理化・省力化を目的にした情報システムの開発に、省力化される金額ではシステムの存続期間での回収は無理な費用を投入する様なケースがあります。

結果的にではあるにせよ、その様なケースが少なからず存在するなら、工程や作業の節目では必ず Why に立ち戻り、冷静になって成果物を評価し、次工程の継続を費用対効果から再評価する必要があります。

その場合、Why を明確にした上で、効果の数量化を行っておくなど、判断基準を整理し、判断を容易にする工夫があると、責任者への過度の判断責任の集中も緩和されるので、判断の客観性が保たれ、組織にとっては不利益な判断も比較的容易に行う事が可能になります。

先ずは、Why を明らかにした後に、「では、その問題の解決策として業務は何をすれば良いのか」が What となります。

つまり、現行業務を分析して問題・課題を明らかにした上で、その解決策として業務設計で次に作るべき業務の形を明らかにするのがこの工程です。

なお、この工程では、あくまでも業務の設計なので、必ずしもシステム開発が前提となる訳ではありません。

しかしながら、この工程を SI ベンダー・IT 系コンサルタント・コンピュータメーカに委託した場合には、既にシステム開発前提での選択肢しか有りません。

特に、外部のベンダーに直接エンドユーザの要求をまとめさせた場合には、要求された機能の必要性を公平に判断することが難しいので、何でも対応する前提で話しが進んでしまいます。

少なくとも業務を如何に運営するかについては、ものの軽重が分かる内部の人間が検討する必要があるので、日頃からの要員育成が不可欠となります。

この工程の成果物としては、業務設計書と言う事になりますが、これには業務に関する設計事項が全て盛り込まれている必要があるのは言うまでもありません。

つまり、後にシステム対応する事になるかどうかを問わず、業務に関する入出力、処理プロセス(受注・請求などの業務プロセスの事で、先ず担当者が～画面で伝票のデータを入力し、などと言う作業や処理の手順は次工程で考えれば良い事です)、データ構造、業務組織、決裁権限などについては、全てを明らかに

する必要があります。
　また、この工程で確認しておかなければならない重要なポイントの一つに実現性の確認があります。
　夢を膨らませる事は無制限に出来ますが、資源が無尽蔵にある訳ではありません。
　その手順を実行する能力があるか、そのデータを収集する事ができるか、その時間で出来るか、その知識はあるかなどの実現可能性の確認を行ない、必要なら教育・訓練・採用・外注を計画に加えるなどの対応を考える必要があります。

　次のシステム設計工程は、前工程で明らかとなった業務の What について、いかに実現するか、つまり How を検討します。
　その限りに於いては、要求の全てに対して、事務手続きの改善や体制・権限の再構築のみで対応する方法も考えられるのですが、何しろ、システム開発方法論として、SI ベンダーやコンピュータメーカが紹介していることなので、そのような選択が出来ることについては、あまり触れられない様で、当然システム対応を行なう事を前提とした、事の運びになっている事はしかたないようです。
　How の設計に関しては、ベンダーなどに任せるケースも増えていますが、後々の運用・保守をどうするのか、又、開発を行なったベンダーとの関係をどう続けるのかについても戦略的な決定を行なう必要があるので、ユーザも無関係と言う訳には行きません。
　ここでの成果物はシステム設計書と言う事になります。
　システム設計書には、前工程の業務設計書を受けて、それを実現するための全情報を網羅することになります。
　プログラムの構成、処理の手順、データの構成、運用の手順、管理の手順、機器の構成、ネットワークの構成、運用体制などは全て明らかにする必要があります。
　ここでのポイントは関連です。
　データ間・プログラム間・データとプログラム間に関連があると、影響の調査や調整が発生し、開発のスピードを遅くし、保守の生産性を低下させます。
　関連の代表的なものは、重複です。
　データも機能も、重複があると同期や反映の負荷が増える事はご理解頂けると思います。
　いかに相互に関連のない仕組みを作るかが、その後の開発やシステム完成後の負荷（主に変更対応の負荷）を大きく左右します。
　実際問題として、上記の検討要素の全てをこの工程で全て網羅することは困難だと思います。
　担当者の能力の問題もありますが、一般的に開発されるシステム規模は、既にその様な次元を遥かに超越した規模と複雑さを持っていると考えて良いでしょう。
　開発方法論に何らか新しい要素が求められる所以です。

　次のプログラム開発は、システム設計で示されたプログラムの実装単位と仕様に従ってプログラムを作り、テスト（単体テストから結合テスト程度までか→方法論の考え方や実装方法の違いで範囲は変わる事になります）を行ないます。
　ここで、プログラムのパターン化やモジュール化などの工夫や、オブジェクト指向開発などの技法、または特殊なツール、自動生成ツールなどを適用する場合には、あらかじめシステム設計工程でその旨を定め、必要な環境設定、作業マニュアルの整備、要員教育などを事前に行なう事になります。
　この工程の作業は全てシステム設計工程での設計内容に基づいて実施されるので、新たに考える作業はありませんが、実装の方法と担当者のスキルが適合していないと思った生産性が実現しない場合があります。

ここでも、個別プログラムの運用・保守を考えて対応する必要があるので、単純に見かけの生産性や、その時々の流行に惑わされること無く、運用・保守部隊の訓練・習熟を考慮した選定を行なう必要があります。

言語に習熟し自由に使える様になるのは、相当の訓練が必要です。

ツールや自動生成を採用するとしても、それらに習熟し使いこなす必要があります。

そして、扱う言語やツールが何であれ、習熟しているかいないかで著しく、生産性や品質に差が出る事は明らかです。

よって、他者に委託する場合でも、その習熟度を測る手立てを持ち、それを適用して費用を決める位の事はやって見せないと、他者に委託する資格を問われかねません。

ここでの成果物は、プログラムやデータベース定義などのソフトウェアなので、その更新・修正に当たっては、世代管理・履歴管理の仕組みを運用する必要が発生するので、システム管理とか開発管理と言われる管理作業が発生し、その体制が必要となります。

新規にそれら管理機能を導入する場合は、それらについてもシステム設計で規定し、ソフトウェアの受け皿として事前の対応が必要となるので、段取り良く対応する必要があります。

テスト・移行の局面は、ソフトウェア同士、ソフトウェアとデータベース、データベース同士の連携接続についてシステム設計を元に確認します。

当然、移行結果としてのデータ群もテストの対象です。

第一、システムだけが完成してもデータが無ければシステムは機能しないので、移行設計・移行開発は、むしろ先行して完了している必要があります。

移行を含んだ段取りの良いプロジェクト計画を立案し、移行に配慮した資源配分の下にプロジェクトを運営する必要がある事は言うまでもありません。

例えば、プログラム開発時には全てのデータが移行を完了していたとすると、単体テストからの作業が如何に楽になるか容易に想像して頂けるかと思います。

テストの種類としては、連携接続の範囲を段階的に広げて行くと同時に、受け入れ・運用などと関係者の範囲を広げ段階的に本番運用に近い環境でのテストへと進めて行きます。

テストの段階では多数の問題が抽出されますが、全てはシステム設計に沿っているか否かを基準に判断することになります。

逆に、システム設計の内容で判断が出来ず、一々システム設計や業務設計の内容について再検討・再確認をしなければならないような事態が発生した場合は、明らかに業務設計・システム設計の不備なので、徒にテストを繰り返すことに関しては難しい判断が必要になります。

なお、開発方法論は多数存在し、開発方法論自体についても、その工程や作業についても、オリジナリティを強調しようと、細かく分けたり大きくまとめたり、独自の命名でアピールしたりしますが、基本的に、工程ごとに作成する成果物、つまり、まとめなければならない情報に変わりはありません。

変わりはありませんが、表現や管理の方法には大きな違いがあります。

特に留意する必要があるのは、規模への配慮です。

十で出来たら百が出来るか、百が出来れば千はどうか、千が良くても万はどうかについては、実際試して見る事で正しい判断が可能になるので、それぞれの状況に合わせて試行する必要があるでしょう。

表現や管理の方法が稚拙で要領が悪ければ、大量の情報の中に、不必要な情報や重複した情報、また、更新が追いつかない情報や更新されない情報が増え、全体的に情報の品質を低下させてしまいます。

表現や管理の方法が優れていれば、無駄や重複無く体系的に情報が管理され、開発時に限らず、運用・保守の局面まで、検索性・更新性を良好に維持することが可能です。

2.　伝言ゲーム

以上、方法論を作業工程から見ましたが、システム開発とは、これら工程・作業を通して大量の情報を変換・創生・付加しながら最終的にプログラムとデータを実装する作業と言う事になります。

良く、システム開発は伝言ゲームに例えられます。

伝言ゲームは、最初に提示される課題のメーッセージが、人から人へと次々に伝わる間に、思いがけなく変化して行く事で笑いを取るゲームなので、いい加減に聞いていい加減に伝えた方がメーッセージが変わってゲームが盛り上がります。

しかし、システム開発で伝言メーッセージの内容がどんどん変わって行ったのでは、笑い事では済みません。

この多段階に亘る大伝言ゲーム大会で勝利するには、前章の最後に述べた、伝言すべき情報の表現や管理の方法がものを言います。

初期のシステム開発には、いわゆるユーザと言う業務の生き字引が存在し、そのニーズに答える形で、今まさに行なわれている手作業での事務をシステムに置き換えれば良く、疑問や確認事項があれば、その何でも答えてくれるユーザに聞けば教えてくれるので、文書化や図表表現がずさんで形式的で単に概要をぼんやり伝える事しか出来ない物であったとしても、大した問題にはなりません。

当然、作っただけで役に立たない設計文書は更新されることもなく放置され、棚に威容をさらす事になります。

この段階では、ユーザの頭の中に業務設計が存在し、システムの規模が小さかったこともあって、システム設計以下は実装を以って代替すると言うところでしょうか。

つまり、ユーザのブレイン（脳）、もしくは記憶がビジネスを表現し、その情報を持ってシステムをコントロールしていたと言えます。

ここで言うコントロールとは、ビジネスの要件としてシステムがどうあるべきかと言うことを示す情報で、何か問題が発生した時に、どうすれば良いかの判断をするための拠り所と言う意味です。

すなわち、システムの開発やテスト・運用保守の期間を通して、（システムの作りの問題ではなく、ビジネスの要件として）「あれ、ここってどうなってなければいけないんだっけ？」となった所で参照し従うべき情報と言う事なので、いわゆるビジネスルールを網羅した情報または情報群と言う所です。

システム化が進み、規模が拡大してくるとシステム化対象の業務全体を全体的に把握しているユーザと言う便利な情報主体は存在しなくなります。

いや、むしろユーザは、長い間システムの端末操作者と化していたため、近視眼的で自分勝手な部分的で断片的な要求に終始するやっかいな存在になり果てたと言っても過言ではないでしょう。

そのユーザーの要求を真に受けていたのでは大変な事になってしまいます。

つまり、聞けば答えてくれる便利な情報源はいなくなるので、代わって部分的・断片的な要求を集めて、文書として情報を記録することで情報源とする工夫が行なわれてきます。

大量の様式の固まりが、方法論と言う事で盛んに紹介されましたが、既に、そのような対応で乗り切れる規模では無くなって来ており、教育の不十分さもあって訳も分からず大量の成果物を作っただけで、何かあると結局、その都度人が集って個別に考えるので、成果物が省みられることはありません。

当然、省みられない成果物は保守される訳もなく、運用・保守の局面で役立つはずも無く、棚に威容をさらすことになります。（今の表現では、グループウェアーに一覧をさらすと言う所でしょうか）

OA化が言われ、成果物自体は電子化される傾向にありますが、媒体の如何によらず、文書として作成されたものの更新・保守は大変です。

最終的に人が見つけて判断しなければならず、抜け漏れの発生を防ぐのは無理な相談です。

そのため、管理スパンと称する記憶の限界が言われ、ある一定規模のサブシステムごとに担当者を置くこと

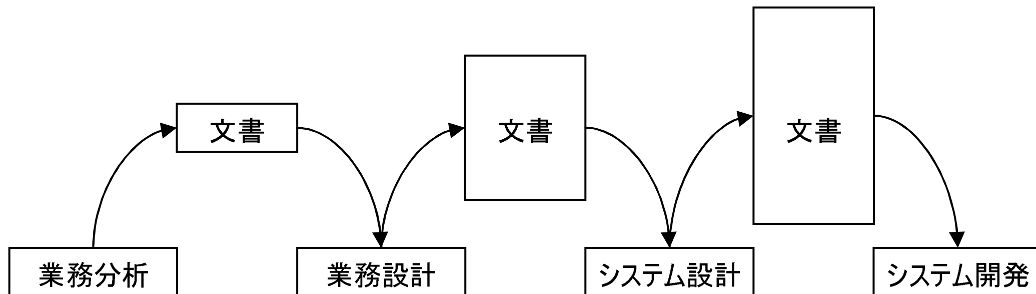
になります。

担当者とは、要するに業務システムの語り部、文字通りの生き字引に他なりません。

つまり、記憶に頼っていてはいけないとして、文書化を進めた結果、大量の文書を管理する必要が生じ、結果的に情報の索引機能を担当者の記憶に置いたと言うことでしょうか。

この、人の記憶と言う便利な機能から、なかなか抜けられない。

図に、大量の文書による巨大伝言ゲームと化した、システム開発のイメージを説明します。



工程が進むに連れて、最終的なプログラムとデータの仕様定義に向けて、同じことが詳細化されながら、多段階に伝言されて行きます。

もちろん、この情報管理にも工夫の余地はたくさんあります。

また、伝言の内容も、参加する技術者の能力や経験によって、大きな相違があります。

一般に、上流ではビジネスに関する記述、すなわちWhyとWhatが中心であったものが、下流に行くに従って、システムやプログラムの作りに関する記述量が増加し、How中心となります。

上流から引き継ぐ業務情報に関しては、段階を経るに従って、必然的に同じ事を複数箇所に記述する事になるので、変更が発生した場合の同期制御が大変困難になります。

文書の管理に当たっては、検索性・関連性・更新性の維持が必要となりますが、何しろ文章と言うやつは便利なもので、書くとなったら何でも書けてしまうので、あることについての記述を調べようとしても（検索性）有効な索引の作成・維持は非常に難しい事になります。

勿論、文書間の関連（関連性）を維持する事も、関連を伝って行なう事になる変更対応（更新性）などに関しても絶望的です。

そこで、様式を定め、記述内容を制約する（記入欄を分けることで、記述内容が名称や年月日である事を明確にするなど）と同時に、システム→サブシステム→ジョブ→タスク→アクティビティなどのプロセス系の情報、データ項目・エンティティ（ファイル・テーブルなどを問わず、データ項目の集合単位）などのデータ系情報を並べて、索引とすることで、検索性・関連性・更新性を確保すべく工夫がされます。

もっともエンドユーザに最も馴染みが深いのは、ユーザインターフェースである事は間違いないので、画面や帳票などのユーザインターフェース一覧は必須ですが、これだけを考えて見ても、情報を何順に並べておけば検索し易いのか工夫が必要です。

ユーザインターフェースに関連した情報としては、システム・サブシステム名称・プログラム名称・ジョブ名称・ユーザインターフェース自身（画面・帳票）の名称及びコード（画面コード・帳票コードなど）又は番号（画面番号・帳票番号など）、更に、ユーザインターフェース上で使用されるデータ項目、及びその提供元としてのデータベース（物理的情報提供元）やエンティティ（論理的情報提供元）との関係などがあります。

実際の利用場面を考えると、ユーザインターフェースの特定から、それを実現するプログラム、そのプログ

ラムで使用するデータベースから、データ項目へと、多段階の検索と関連付けが必要となります。

また、要求や問題の発生は場所を選ばないので、要求事項はシステム・サブシステムの機能から発生したり、法規制によるデータ項目導出ルールの変更から発生したりと、最終的にシステム全体の中でどの部分を対象に何を行なえば良いのかを特定するためには、一覧表形式の管理では限界があります。

結局、毎日朝会を開いては、担当者が集って自分が担当しているサブシステムが関係するかどうかを判断しながら、作業を割り振ることになり、人の記憶が頼りの管理レベルから抜け出せない状況に陥ります。

3. 情報整理技術としてのデータ中心アプローチ

データ中心アプローチは、データモデルをビジネスのモデルとして詳細に描き、モデルにビジネスルールを描き込むことで、記憶による情報管理から、記録による管理に脱皮しようと言う画期的な技術です。

世に大半のシステムエンジニアは、データモデルをデータベースの設計情報だと思い込んでおり、データベースを設計しようとして、データモデルを描こうとしている様ですが、データモデルはビジネスのモデルなので、恣意的にデザイン出来る要素はありません。

ビジネスの中で、「どの様な情報（アトリビュート）」が「何の単位（エンティティ）」で「どの様な順序（関連-リレーションシップ）」で記録されるのか（されなければならないのか）を明確にし、一定の表記法に従って作図したものがデータモデルです。

このモデルの、エンティティをテーブルに、アトリビュートをカラムに変換したものが、データベースの設計として適切なものになるのは、エンティティがビジネスの中で記録する単位として独立性・関連性・順序性を十分に吟味・分析された結果であれば当然であると言えます。

何度も言いますが、ビジネスシステムのデータベースに設計の要素はありません。

良いデータベースを作るポイントは如何にビジネスの実体に合致しているか否かの一点のみです。

そのため、同じ情報を同じ表記法で作図すれば、誰が描いても同じ図面になります。

この点、世間には二つ誤解があります。

第一の誤解は、データモデルの名人の存在です。

業務を分析する場合の見極めの方法は、全アトリビュートがエンティティ収まった事を確認することです。

そのビジネスに登場する全てのデータ項目が、全て何れかのエンティティに収まれば、それ以上整理する必要は無くなります。

後は、ビジネスルールに従って、割り出されたエンティティの関連を設定すればデータモデルは完成ですが、ここで多くの技術者が間違いを犯します。

それは、ビジネスに関係の無いデータ項目やエンティティに収まらないデータ項目まで、無理やりデータモデル上に表現しようとすることです。

例えば、登録日時・登録担当者/承認者・更新フラグなど、データベースの運用管理に関するデータ項目は、最終的に実装する場合には、勿論カラムとして実装されますが、それは、データベースの運用管理モデルとして決めることであり、ビジネスのモデルを表現しようとする場合には無用なことです。

同様に、契約金額を月別・支店別に集計した「月別支店別契約合計金額」なるデータ項目は、データモデルの中に属するべきエンティティがありません。

これを無理に描き込もうとすれば、月別支店別の集計データを収容する仮想のエンティティを設定しなければならなくなります。

その類のデータ項目は実際のビジネスではコンピュータのパワーを頼んで様々な切り口で多数存在するので、一々収容するためのエンティティを作っていたのでは、エンティティ数が膨大な数になってしまい、元々のビジネスの中で存在するエンティティの構造を分かりにくくしてしまいます。

勿論、これらのデータ項目もシステムで使用するからには、データベースに実装するので、名称やルール

の管理対象にはなりますが、エンティティとしてデータモデル上に表現する必要性はありません。

以上のことに留意しただけで、データモデルはモデルとしての精度を格段に向上させることができます。

つまり、データモデルの名人などは居ません。

ここでの上手い下手の違いは、データ整理の基本をビジネスに置くかシステムに置くかと言うことの違いでしかありません。

ビジネスのモデルを描くと言う姿勢で一貫すれば、たちまちあなたも名人級と言うところです。

ただし、分析の巧拙や表現の粗密の差は出る可能性があるので、データ項目が全部エンティティに収まったからと安心しないで、初期値設定が必要にならないか、区分・フラグの差異を把握しているかなどの詳細な分析と表現を行なわなければならないことは言うまでもありません。

第二の誤解は、表現は表記法に優ると言うことです。

つまり、「ここはこう描かなければならない」と言う事は表記法の制約やツールの機能などに構わず描かなければならないと言うことです。

分析法・表記法は、何れも長い経験と多数の事例を踏まえて作られており、基本的にその制約の下で分析と作図を進めて行けば良いのですが、業務の実体が技法の制約で表現出来ないケースが出ることがあります。

その場合は分析法・表記法のルールに従わなければならない義理はありません。

自分が考えた通りに描く(「考えを描く」これぞ概念モデル)ことが重要であり、ルールは補助的な手段でしかありません。

ちなみに、データモデルに表現出来るのは以下の内容です。

○　エンティティ

- リソースエンティティ →物理的・概念的な存在を表現する
- イベントエンティティ→出来事を表現する
- 分類構造→区分・フラグによる相違点を構造的に表現する
- 従属エンティティ→繰り返しなどの解消のためにエンティティに付加する表現
- エンティティ名→全てのエンティティはビジネスの中で相当する存在・事象の名称で表現する
- 識別子→全てのエンティティは、そのインスタンスの識別をするデータ項目(または項目群)を識別子として表現する

○　リレーションシップ

- イベントエンティティ間の関連→ビジネスの骨格構造を表現する(先行エンティティが後続エンティティに対してn:1の関連を持つ場合は後続エンティティに従属エンティティを設定し、先行エンティティと従属エンティティの関係を1:1、従属エンティティと後続エンティティの関係をn:1として、後続エンティティ内に先行エンティティの識別子が参照キーとして不定の繰り返し構造となることを防ぐ)
- リソース・イベント間の関連→イベントエンティティ発生のWhen・Where・Who・Whomなどを表現する
- リソースエンティティ間の関連→組合せによる存在またはイベントと認識されていない出来事を表現する(関連エンティティの設定)
- 再帰の関連→同じエンティティ内でのインスタンスの関連を表現する(再帰エンティティの設定)
- 明細の関連→イベントエンティティがリソースエンティティに対して1or　n:nの関連を持つとき、明細エンティティを設定してイベントエンティティと明細エンティティの関連を1:1or　n、明細エンティティとリソースエンティティの関連をn:1とし、イベントエンティティ内部にリソースエンティティの識別子が参照キーとして不定の繰り返し構造となることを防ぐ
- 参照キー→全ての関連は、時系列に従って、識別子を挿入し、関連の存在を参照キーで表現する

- 多重度→全ての関連は、インスタンス同士の対応関係を多重度で表現する

○　アトリビュート

- データ項目→全てのエンティティに帰属する識別子・参照キー以外のデータ項目は、アトリビュートとして、そのエンティティの特徴・性質を表現する
- データ項目名→アトリビュートは基本的にビジネスの中に実在するデータ項目なので、その名前で表現する
- チェック/導出ルール→入力されるアトリビュートはチェックルールで、内部で導出されるアトリビュートは、導出ルールで名称とともに表現する（名称が同じでルールが違えば、同音異義語、名称が違うがルールが同じなら異音同義語）

また、これを使って表現するデータモデルをエンティティリレーションシップダイアグラムと言い、ERDと略記しますが、細かい表現や分析アプローチの違いによって、多くの方法が存在しています。

そのため、データ中心アプローチの導入初期にはツールや技法の選定に力を入れることになりますが、基本的に大きな相違は有りません。

如何なるツールや技法を採用しようと、ビジネスの実態を把握し、どう表現すべきかの判断が（技法のルールやツールの機能に頼ることなく）自分で出来なければ使い物にはなりません。

4.　伝言から集中へ

いわゆる文章で表記されたものは言うに及ばず、一覧表にせよダイアグラムにせよ情報から情報に細分化し詳細化する過程で、情報の重複は避けられません。

そのため、変更が発生するとそれら一連の重複情報を関連付けて変更しなければなりません。

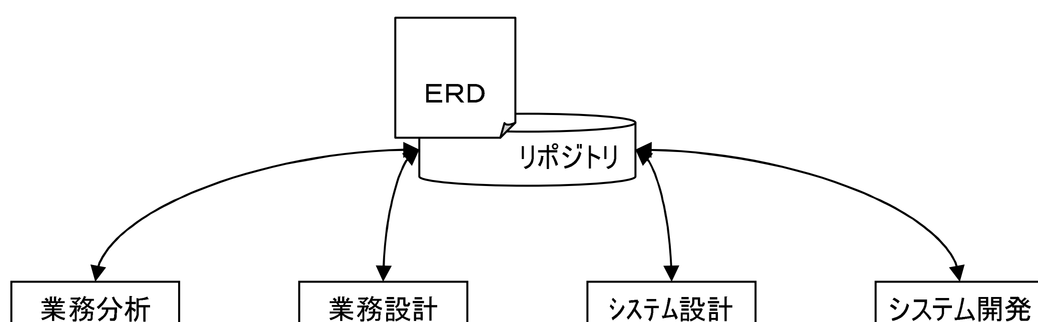
そこで、管理の効率を上げるためには、漫然と存在する情報を対象とするのではなく、分析・検討の過程で作成し、結論を得ることで不要となる類のものと、最終的な結論が記録してあるものに分けることで、継続的管理対象とする情報は、極小化した上で、冗長性を排除する必要があります。

冗長性が排除されれば、変更に対応した「そこ」だけに対応すれば良いので、関連を気にする必要はなくなります。

データ中心アプローチと言うと「One Fact in one place」・「One Fact One Data One Place」などの有名なキャッチフィレーズが、データ中心アプローチの特徴として、データの重複排除の必要性を言いますが、データ中心アプローチをデータ管理術として考えれば、当然のことと理解出来ると思います。

データベースがビジネスに関するデータの交換場として機能する事を目的としている様に、ERD（＋リポジトリ）を、システム開発に関するデータ交換場として機能させる事を目的にしているのだと考えれば理解がより容易になるかも知れません。

次に、システム開発に於ける伝言ゲームを、ERD（＋リポジトリ）が、上流工程での情報の集中・蓄積と、下流工程で、その情報を検索・利用している状況を図で説明します。



ビジネスシステムはビジネスの実体に対応して存在します。

業務分析から業務設計にかけてERDとリポジトリにビジネスルールが集中・蓄積されて行きます。

データ項目に導出ルールなど、ERDに描ききれない情報は、リポジトリに記録されます。

システム設計から、システム開発にかけては、ERDとリポジトリに描かれた情報に従って、データベースとユーザインターフェースを実装して行きます。

ERDとリポジトリは、ビジネスルールを記述し管理する物なので、プログラム言語やDBMSなどの実装方法に影響される事はありません。

そのため、機器・環境・言語・担当が変わっても、ビジネス自体に変化が無ければ変更する必要は有りません。

良く、データ中心アプローチの特長を言う場合に「データはプログラム（プロセス）より安定している」との表現がされますが、あれは、この事を指しています。

即ち、情報システムの扱うデータとは、ビジネスの記録なので、どのデータを何の単位で記録しなければならないと言う事は、どうやって記録（したり利用）するかには影響されないと言うことです。

5. まとめ

データ中心アプローチの特徴を、システム開発方法論の伝言ゲームとの対比で説明して来ました。

現在のシステム開発は、ERPとプロジェクトマネジメントに席巻されています。

10 年前、20年前に比べ、ハードウェアが格段に安価になったにも関わらずその恩恵を直接的に受けている企業が幾つあるでしょうか。

ソフトの自動生成や再利用が言われて久しいにも関わらず、システム開発プロジェクトの期間や費用が大幅に削減されたニュースを聞いたことがありますか。

ダウンサイジング・ライトサイジングでシステム運用費用の大幅削減が実現する予定では無かったですか。

質はともかく、量的には、この間の爆発的なシステムの肥大化が、全てのメリットを帳消しにした上で、更なる費用の浪費を強いている事に間違いは無いでしょう。

この稿は引き続きシステム肥大化への唯一の対抗策としてのデータ中心アプローチについて、その考え方と実践手段について述べて行きたいと思います。

本稿に関するご意見・ご感想・確認事項などがありましたらUBIKあてにメールをお願いいたします。

出来るだけ、反映したいと考えています。

以上

小見山昌雄