B-022

MDA ： 構想からプログラミングまでの継目のないアプローチの実際

中村　正治
Masaharu NAKAMURA

1． まえがき

MDA, MDD というと、いささか手垢のついた、という感じを持たれる方もいると思われる。それは、そもそも完璧なものを望みえない CASE ツールという文脈の中で、モデル駆動とは裏返しに、ツールが主となりモデルが従となるような本末転倒の末の故だけではない。前例のないシステム構築に挑む必要のある先端的企業が、モデリングを研究するコンピューターメーカーの深奥部との協業を行っても、様々な制約から、その知識は、広く出版されることも発表されることもないままになることがあるからだと考える。

しかしながら、現実の世界を、関心の領域において理解し・記述したうえで、あるべき世界を現実のものとするためにまずは記述する、という、本来のモデリングの重要性は減じるはずもない。実務家が日々格闘しているのは、まさにこの、構想して作り上げるという過程を通じて発展性と整合性を保ち続けることのできるモデルの作成だといっても過言ではない。

2．本論文の概要

本論文では、いったんツールは離れ、構想から構築までのモデルによるシームレスな記述の方法を紹介する。そのうえで、局面化プロジェクトの局面とモデルの位置関係を解説する。最後に、モデルに基づくプロジェクト遂行の上での課題を記述し、今後の普及の便としたい。

神学的なあるいは宗派的な論争を避けるため、付録としてモデルの定義を与えておく。また、本稿においては、特に断らない限り、「システム」は、人のタスクとコンピューターシステムのタスクとの複合したものを指す。

3. モデルの論理構成

3.1　歴史的背景

1970 年代後半から 1980 年代にかけて、プログラム設計ではなくシステム設計になって行く過程で、設計・実装されたプロセスの「足の速さ」と、「データ」の本質的な意味合いが明らかになって行った。1980 年代半ば境にして、それまでのプロセス中心設計からデータモデルを出発点とするデータ中心設計に世の中は大きく舵を切った。データ中心設計は、アプリケーションの大域構造に透明性を与え、環境の変化に対しての耐性の高いシステムの設計を可能にした。しかしながらデータモデルに基づくプロセスの設計、という課題は、幾つかの方法論が提案されはしたが、決定打を欠いていた。

その後、オブジェクト指向の研究と実用化が進み、プロセスとデータは同時に設計すればよい、と言われるようになったが、多様で恣意性の高い関係が定義されるオブジェクトクラスモデルは、それを構成し正規化するうえでの実用的な方法論を作り出すことができないでいた。

その状況は、1990 年代の後半から、UML として、いろいろなモデルが記述面から整理されていったことと、タスクの駆動をコリオグラフィーとして外出しする SOA 構造への設計アプローチをきっかけにして大きく動いた。コンピューターメーカーは、会社間での共通性の高いいくつかのアプリケーション分野に対して、カストマイズのテンプレートとして使える完成版のモデルを提供し始めた。このメーカーのモデルは、そもそも有料のソルーションでメーカーのビジネス施策と連動したものであり、また、特定のビジネス分野に特化したものであった。このため、汎用の方法論として世に紹介される機会は大変少なかった。

筆者は、これを特定の業務分野とメーカーの意図から切離して、汎用の方法論として整理し、自身の企画・実施した滝型開発プロジェクトに適用した。この結果、そのシームレス性・プロジェクト管理との親和性において、大変優れているとの確信を得た。

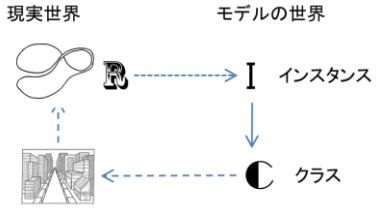
本論の最初に述べたように、長きにわたる、「プロジェクトの進展と連携したシームレスなモデルによるシステムの記述」という問題は、高いレベルで解決されていると考えるが、このことは残念ことにあまり知られていない。本論文執筆の機会をお借りしてご報告するものである。

3.2　メタモデル

二つの考え方からなる。それはまず、現実世界の理解には様々な次元からの抽象が必要とされることである。図 1-a

さらにモデルの世界へ、厳密な定義の困難な写像をしなければならないことである。自己撞着的であるが、この写像を共通に合意するためのよすがが、モデルの世界でクラスとして定義されたものが現実世界与えるパースペクティヴである。

図 1-b



3.3　使用するモデルと作成手順

(1) プロセスモデル　(図 2)

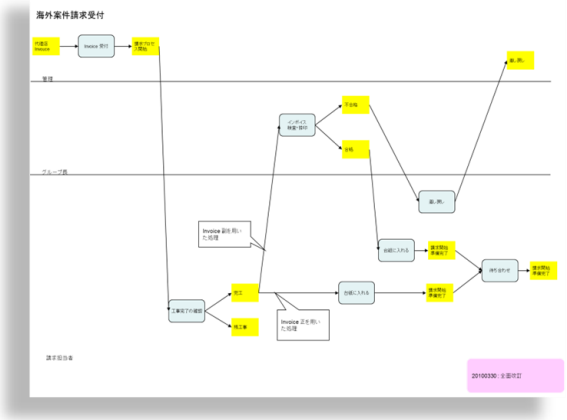
プロセスモデルでは、プロセスを、タスクとタスク間の遷移関係・遷移条件で記述する。タスクは内部構造と

して、更なるプロセスを含むような入れ子構造をとることができる。それ以上の入れ子構造を定義することが無意味と判断できるタスクを素タスクと呼ぶ。ある素タスクからもう一つの素タスクに選択の余地なく遷移する場合でも、タスク遂行の主体・時刻・場所・意味などが異なる場合は、これらを融合させてはいけない。

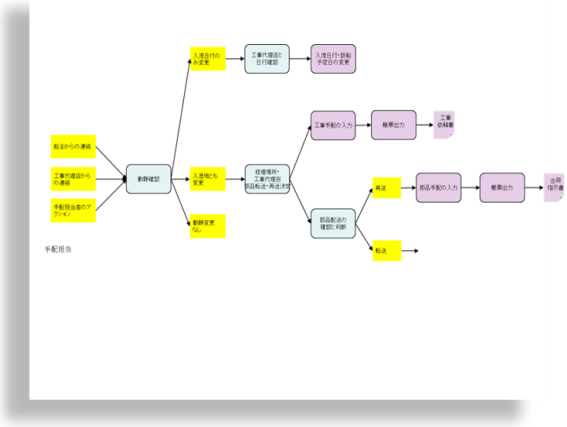
同じタスクが複数のプロセスに存在すること自体は許容される。ただし、このことも含めて、プロセスが最適なものかどうかは、モデル作成者の判断に依存する。

プロセスモデルは、セッションを通じて、IT 担当者の支援のもとで、ビジネス担当者が作成する。ファシリテーターは利害関係の外にいて、迅速かつ知的に結論を導いていく。

図 2
プロセスモデル(組織・担当を合わせて記述したもの)



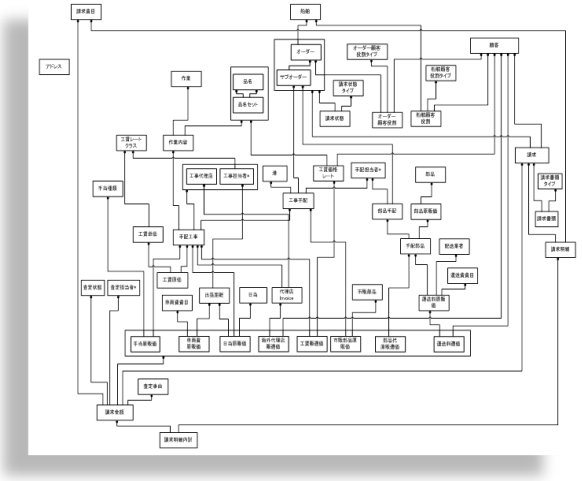
プロセスモデル(コリオグラフィーを記述したもの)



(2) 正規化データモデル(図 3)

正規化されたエンティティーとリレーションシップによって記述する。アトリビュート・リストは、最初は完全である必要はない。議論になりやすい個所では、サンプルのインスタンスの記述が必要である。リレーションシップの記述は、かなり後になるまで詳細である必要はない。よって、簡便な記号の仕様が望ましい。

図 3 正規化データモデル



企業ごとに共通性の高いビジネスに関しては、そのビジネス分野のデータモデルが、幾つかの会社から販売されている。

データモデルは、セッションを通じて、IT 担当者の支援で、ビジネス担当者が作成する。ただし、IT 担当者には現実のデータベースへの拘泥から、データモデルの発想の自由度の低いものがいる場合があり、人選に注意が必要である。ビジネス担当者の出席は本来必須だが、強い抵抗感を示しセッションの崩壊を惹起すことがある。実施に関しては十分な注意が求められる。

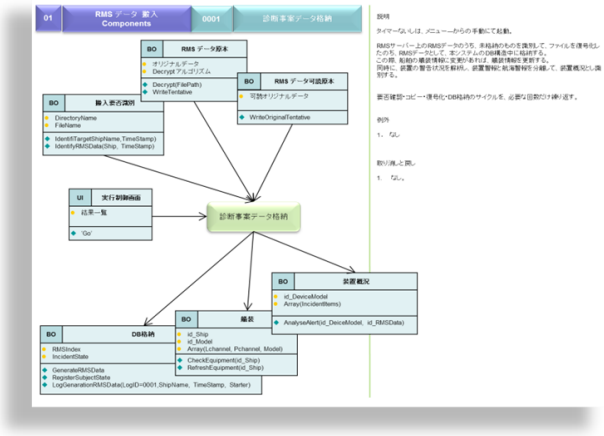
(3) ユースケース　図 4

素タスクを、コアとオブジェクトクラスとの相互作用として記述する。IT 担当者の作業として作成され、セッションでレビューされる。

画面等の UI も、ユースケース定義において、オブジェクトクラスの一つとして定義される。

機械化タスクのユースケースの記述には、ビジネス担当者の参加は必要ない。人力タスクの場合、IT のみに関心があるのであれば、ユースケースの記述は不要である。

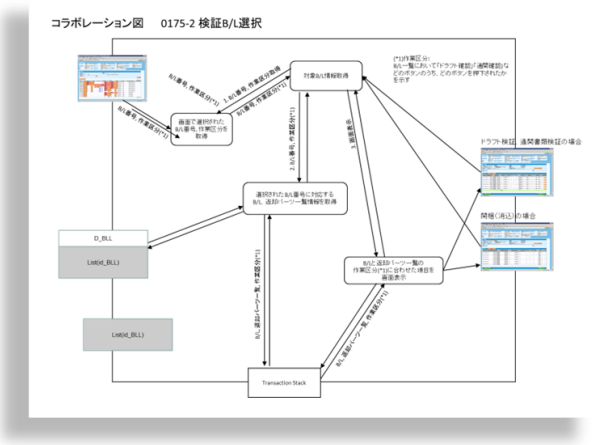
図 4 ユースケース



(4) コラボレーションモデル 図 5

これにより、ユースケース・コアの記述をする。(3)で定義されたオブジェクトクラスは、コアの外部にグローバルなクラスとして存在するが、これらと相互作用する内部モジュールを定義する。このモジュールは、手続き・函数・オブシェクトとして実装されることになる。

図 5 コラボレーションモデル



作成は IT 担当者が実施する。

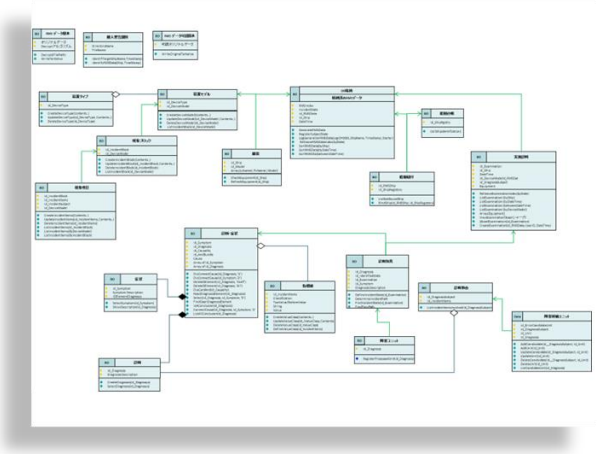
(4) オブジェクトクラスモデル 図 6

ステレオタイプを定義したうえで、(3)で識別したオブジェクトクラスを洗練してその関係を記述する。その結果は、コラボレーションモデルの改訂を惹起することもある。この作業で、データモデルのアトリビュートとオブジェクトモデルのプロパティの整合性を担保する。

ステレオタイプは、多くを定義する必要はないと考える。User Interface, Business Object, Data Access, Enumeration などを定義しておくと便利である。

作成は IT 担当者が実施する。

図 6 オブジェクトクラスモデル



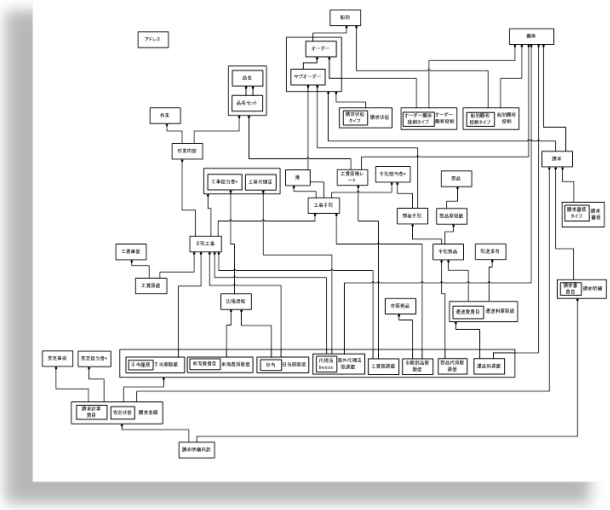
(5) コリオグラフィー

本質的にはプロセスモデルそのもので、特にこのための新たな表現は不要としてもよい。たたし、プログラミングとの相性を考えれば、シークェンス図の使用が便利な場合がある。

(6) 非正規化データモデル　図 7

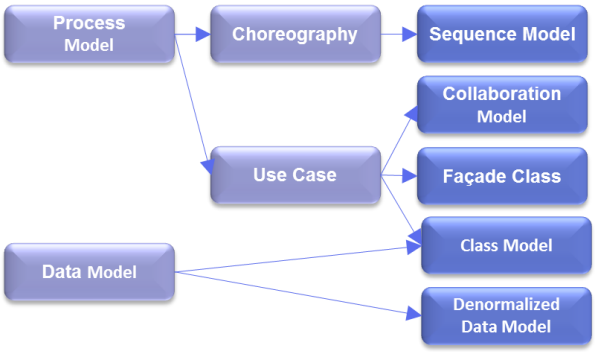
手順に従って、(2)の正規化データモデルを非正規化する。非正規化手順を、どこまで手順厳密に遂行するかは、プロジェクト管理側との協議と合意が必要である。実施は IT 担当者の責任である。

図 7 非正規化データモデル



これらのモデルとその作業に従った時間的発展による相互の関係を図 8 に示す。

図 8 モデル時間発展



4. 滝(Waterfall)型プロジェクトモデルへの適用

4.1 滝型プロジェクトモデルの構造

多くの議論があるにせよ、一定の規模のシステムの開発においては、画期を設定したタクティングによるプロジェクト管理が必須である。同時に企業システムにおいては、これに投入される資源に関しての経営判断のポイントもプロジェクト進捗の大きな要素となる。この大きな構造を図 9 に示す。なお、用語の混乱をけるため、大きな構造での画期を本論では「段階(Stage)」、その下位構造を「局面(Phase)」と呼んでおく。

図 9 段階と局面

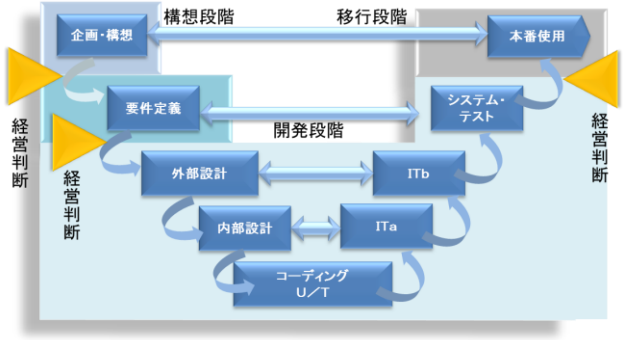


図 9 に関しては、

(1) 企画段階の後、開発段階は複数の必ずしも同期しない

プロジェクトが定義されることがある。また、全くプロジェクトが定義されないことがある。
(2) 開発段階の最初に実施される要件定義局面では、構想段階のアーティファクトは変更されうる。これは、経営・責任者・担当者の利害・視点・立場が違うからであり、各段階内部での滝の原則は適用されない。
(3) 開発段階の後の移行段階の実施に関しては、その実施の応否もふくめて、品質・ビジネス環境などに基づく判断となり、モデリングとは別のイシューである。

開発段階は、9 図のような内部構造を有しており、前半の各作成局面とそれに対応する「試験」局面に大別される。通常、要件定義局面と外部設計局面の間には経営判断ポイントが設定される。この段階で、後続工程実施の応否、プロジェクトの分割などの意思決定が行われる。

要件定義局面の本質は、構築対象の定義である。システムを、IT とビジネスプロセスの複合体ととらえ、ここでその本質を表現する。

外部設計局面は、設計単位をブラックボックスとして、祖の入出力を詳細に定義する、というのが本来の命名の意図であり、従来の概要設計・詳細設計が境界と作業定義に問題があった点を改良して定義・命名された局面である。

内部設計局面は、このブラックボックスの内部構造を設計する局面である。

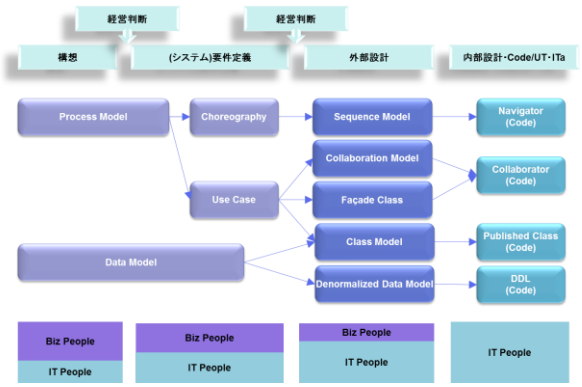
残念ながら、構築物の階層が増し、複数のプラットフォーム上の構築物の相互作用が前提となる現代のシステムでは、このシンプルな内部設計・外部設計という概念では間に合わないことは明らかである。

本論文では、局面の名称としての外部設計・内部設計は使用するが、内容はモデリング・タスクにより再定義する。その中で、外部設計局面は、作業・レビューにビジネス側からの参加が必要な局面、内部設計局面は IT 側のみで進める局面、ということになる。

内部設計は、プラットフォーム・プロクラム処理系・設計者の洗練度に依存する部分が大きく、MDA の関心が終了する領域である。本論文ではここについては概略を述べるにとどめる。
それ以降の局面については、方法論としての MDA の範囲外である。ツールの観点からはプログラムや DB 構造の自動生成が当然求められるが、これらは、メーカー・実装・プラットフォームに依存することになる。

4.2 滝型プロジェクトと設計モデル

図 10



(1) 構想段階

プロセスモデルと正規化データモデルを作成する。プロセスモデルは、新たな業務プロセスについての視覚的で理解容易な知見を提供し、データモデルは、ビジネスの本質に対して透明で不易な知見を提供する。これらのアーティファクトには、経営の理解を支援する解説書が付されなければならない。また、構想段階でのアーティファクトは「設計書」ではない。実務的な観点からの変更が開発段階で行われることが共通認識として了解されている必要がある。プロジェクトとしては、これらのアーティファクトを変更管理対象とする必要はない。

プロセスは、モデル作成者の方法論・現ビジネスに対する理解の促進と、現新比較のために、As-Is, To-Be の両モデルを作成するべきである。

データモデルは、As-Is は一旦棚上げし、ビジネスの概念からトップダウンで大構造を定義するのが良い。かつてのデータ中心設計で議論が尽くされたように、ビジネス定義の本質はこのデータモデルにあるといっても過言ではないからである。

(2) 開発段階.要件定義局面

ここで、プロセスモデルと正規化データモデルを完成させ、変更管理対象とする。
プロセスモデルからは、ナビゲーション・プログラム生成のために必要な形式で、コリオグラフィーを定義する。プロセスモデル、データモデルからはユースケースを作成する。データモデル、プロセスモデルの完成にはビジネス担当者の参加が必要であるが、機械化タスクのユースケースの作成には不要である。
要件定義局面のアーティファクトに基いて、構築対象がヒューマンタスクの事務プロセスなのか、マシン・タスクの IT プログラムなのかが定義される。
これ以降の議論においては、ヒューマンタスクは必要がない限り割愛する。

マシン・タスクのユースケースでは、コアとオブジェクトクラスを定義する。ここで定義されるコアは、内部ロジックを持つプログラムモジュールで、後工程で内部ロジックをコラボレーション図により定義する。コアによって駆動されるオブジェクトクラスは、機能の識別が可能な単位に分割され命名された複数のメソッドと、先に定義したデータモデルの複数のエンティティ・アトリビュートの組である。この段階では、メソッドの内部仕様、オブジェクトクラスの正規性の担保およびクラス図は不要である。

プロセスモデルは、必要に応じて記述図を替えたうえで、コリオグラフィーとして再定義する。コリオグラフィーは、コアとなるコリオグラファーと、そこから呼び出される上記のユースケースによって定義される。

重要な点として、これらのアーティファクトによって、後続工程の作業が明確に定義されることと、経験値をベースとした負荷見積が高い精度で行えることである。

(3) 開発段階.外部設計局面

外部設計では、コリオグラフィーを確定する。プロセスモデルには、プロセスのドライバーと、例外プロセスが明示的には記述されない。シークェンスモデルにり、これらが記述されるようにする。プロセスモデルに記述

された素タスクは全てコリオグラファーからの呼出しによるものと定義することが適当である。

ユースケースからは、コアの振舞いとグローバルなオブジェクトクラスとの相互作用の両方をコラボレーションモデルにより記述する。ここで呼びたされるオブジェクトクラスの定義と、ステレオタイプへ分類したうえでのクラス図で表現するオブジェクトクラスモデルを作成する。オブジェクトクラスの定義は、データモデルに定義されたエンティティとアトリビュートとの整合性が担保されている必要がある。

ユースケース・コアの内部構造は定義されたところで、以降をプログラマーに委ねるか、更に明示的に定義するかの判断が必要となる。コアの作成を分割する場合、内部の設計単位をファサードとして外形的に定義するのが便利である。

同時に、正規化データモデルから、手続き的に非正規化データモデルを作成する。繰込み・包含による非正規化の手続きの詳細は割愛する。

(4) 開発段階.内部設計局面

内部設計局面は、いうなればプログラムの外部設計である。ここでのアーティファクトは、コリオグラファーの定義としてのナビゲーター設計書、コラボレーションモデルとファサード定義からコラボレーター設計書、オブジェクトクラスモデルからはパブリッシュト・クラスの設計書である。

ここでの設計書は、プログラム仕様そのものであり、もはや MDA のモデルではない。従って、これらは本論文の範囲の外であるが。付け加えるならば、これらの仕様書はプロジェクト管理者と熟達のプログラム実務家により、そのプロジェクトに最適なものを定義すべきである。

非正規化データモデルからは、DDL が作られる。モデルと DDL の間には、DB パラリズムなど、非機能要件に起因する相違が発生する。

4.3 UI と反復試作

外部設計のアーティファクトで、ビジネス側にとって重要なものが UI である。モデル上は、定義からわかるように、UI のコンテナはオブジェクトクラスであるが、これを可視化し、UI のみの外形的なフローを作成することにより、構築するシステムのビジネス側から見た「品質」の担保ができる。UI は、処理系が定義されていれば、ユースケース・コアや他のオブジェクトクラスからは独立して、表現部分のみの厳密な定義ができる。個々の反復試作(Iterative Prototyping)により、要求品質の大きな部分の担保ができる。

4.4 CTP(Comprehensive Test Plan)

ここでは、テスト・テスト計画については論じないが、補足としては、要件定義局面における CTP 作成においては、それぞれの局面でのアーティファクトの記述内容が IT システムとして実装されていることの検証を、アーティファクトに基づいて実施する試験の計画をつくるための、方法論の選択と試験データ作成が必要である。

5. 課題と考慮点

5.1 ツール

(1)構想段階

構想段階ではモデルの無矛盾性は大きな問題ではない。それよりはビジネス側から参加する担当者の高度な専門業務知識を反映し記述できるツールが求められる。筆者の経験では、プロセスモデル・データモデル共に、満足できるものはない。その一番大きな要因はディスプレイである。部屋中の壁がディスプレイであるようなセッションルームがれば、どれほど効率が良いことか、と思う。付随して、その入力手段である。ディスカッションとともに、モデルの図は変更されていくが、入力担当者がキーボードとマウスを使って発言者の意見を入力していくのは如何にも迂遠である。思考とディスカッションとのスピードが違い過ぎて、大きな効率低下の要因となってしまう。今後の技術的進歩に期待したい。

(2)要件定義以降

要件定義以降では、テスト終了までのモデルは全て変更管理対象となる。したがって、変更の影響を反映し全体の無矛盾性を維持することのできるツールは是非とも欲しいところである。

一方、プログラミングのためのオブジェクトクラス内部のメソッドの実装設計やユースケース・コアの実装などの厳密な設計は、外部設計の段階では不要である。ケースツールに求められるような、最終的にプログラムの自動生成を行えるような設計を早い時期から行うのは、オーバーヘッドを生むだけである。

モデルの整合性は、記述されたモデル内部では絶対に必要だが、プログラム自動生成までを含む必要はない場合が多いと筆者は考える。

繰り返しになるが、ツールはあくまで手段にすぎない。ツールがあれば生産性は上がるだろうが、そのツールが MDA の本質的な価値を損なってしまうようでは本末転倒である。

5.2 グループワーク

データモデル、クラス図などが記述している対象はグローバルである。さらに、オブジェクト・クラスは自身はローカルであっても、多くの相互参照(Cross Reference)を前提としている。

グローバルなモデルでは、かなり大きなプロジェクトでも、最終的には一人の管理者が全体の責任を負わざるを得ないし、ローカルなモデルで、チェックイン・チェックアウトのライブラリー管理・メカニズムがあったとしても、チェックイン時に全モデルとの整合性を担保するのが一人の管理者、という状態にならざるを得ない。

この問題も、エレガントな解決策を見出すのは、今後の経験の蓄積を待つほかないと考える。

5.3 配役(Casting)

システムの設計には、ビジネス側からの専門家の参加と IT 側からアーキテクトというスキルエリアの担当者の参加が不可欠なのは無論であるが、アーキテクトのうちでも、特に「モデラー」と呼ばれる担当者が明示的に参加しているケースが欧米ではほとんどである。モデラーはビジネスコンサルティングの技量とモデルのプロジェット進捗に伴う発展の細部を追跡し見通すことのできる能力とファシリテーションの力が求められる。このようなスキルエリアを開発するのも今後の課題である。

6 結び

筆者は得られた機会により、数度の中規模のプロジェクトを、この方法論を用いて実施した。モデルとは、現実の世界から、関心の領域のみを抽象し、定めた表現により記述するものである。当然、抽象の過程で、重要な

要素が切り捨てられたり、あるいは実装という、抽象されたモデルから現実世界への写像の中で、夾雑物が付加されたりしてしまう。完璧なものはない。ましてや、方法論を支援するツールはそうである。

しかし、われわれは半世紀をかけて、プログラムとファイルの記述、システムの記述・システム・アーキテクチャーの記述の方法を考え続けてきた。MDA と言われる道具立てによって、われわれはほぼその方法論を手中にできたと考える。あとはその知識を多くの人が利用し、洗練度を深めていくことができれば、と願う。

7 付録

(1)プロセスモデルの定義

T：タスククラスの集合。**C：**条件クラスの集合。
任意の **t1, t2** ∈ **T,** において **t1 ≠ t2** のとき、
T をタスククラスの正規化集合と呼ぶ。
t1≠t2, c∈C のときに、
s=[t1, t2, c] の組を順次クラスとして定義する。
s1=[t1,t2,c1], s2=[t3,t4,c2], ・・・
s1, s2 ∈ **S,** かつ **(t1≠t3 ∨ t2≠t4 ∨ c1≠c2)**のとき
S を順次クラスの正規化集合と呼ぶ。
ss=[φ, tx, φ], se=[ty, φ, φ]として、
p ={s1,s2,s3,・・・} かつ
∃sx∈p|sx=ss, ∃sy∈p|sy=se のとき、
p をプロセスクラスと呼ぶ。
任意の **p1, p2** ∈ **P** ならば
P をプロセスクラスの正規化集合と呼ぶ。
単射 **n：∃p → t** かつ **n(-1)**が定義できて
n(-1) : t → p が単射の時、
n をネスティング、**t** を **p** のネストプロセスと呼ぶ。
ある **t** において ネスティング**(**とその逆**)**が存在しない時、
t を素タスクという。
T = {t1,t2,・・・},
この時、∀t∈T∃**t**∈**T** に単射 m : t → **t** を定義できるとき、
t は **t** のインスタンスと呼ぶ。
C = {c1,c2,・・・},
この時、∀c∈C∃**c**∈**C** に単射 n : c → **c** を定義できるとき、
c は **c** のインスタンスと呼ぶ。
t1, t2∈ T、かつ t1≠t2, c∈C のときに、s=[t1, t2, c] の組を定義し、
∀s∃**s**∈**S** に単射 m : s → **s** を定義できるとき、
s は **s** のインスタンスと呼ぶ。
p = {s1,s2,s3,・・・} かつ ∃sx∈p|sx=ss, ∃sy∈p|sy=se のとき、
∀p ∃**p**∈**P** に単射 o : p → **p** を定義できるとき、
p は **p** のインスタンスと呼ぶ。
現実世界の実体と対応させて、
タスク t1,t2,・・・, 条件 c1,c2, ・・・, 順次 s1,s2, プロセス
p1,p2,・・・ が定義できて、
これらが、クラス、インスタンスの定義と矛盾しない場合、
この対応をプロセスモデルと呼ぶ。

(2)データモデルの定義

A：アトリビュート・クラスの集合。
E：エンティティ・クラスの集合。
任意の **a1, a2** ∈ **A,** において **a1 ≠ a2** のとき、**A** をアトリビュート・クラスの正規化集合と呼ぶ。
a1, a2, ・・・∈A, のときに、**e=[e1,e2,・・・]** の組をエンティティ・クラスとして定義する。
任意の **e1, e2** ∈ **E,** において **e1 ≠ e2** のとき、**E** をエンティティ・クラスの正規化集合と呼ぶ。
A = {a1,a2,・・・},
この時、∀a∈A∃**a**∈**A** に単射 m : a → **a** を定義できるとき、a は **a** のインスタンスと呼ぶ。
E = {e1,e2,・・・},
この時、∀e∈E∃**e**∈**E** に単射 n : e → **e** を定義できるとき、e は **e** のインスタンスと呼ぶ。
ex,ey∈ **E** において∀ey←**ey** | ∃単射 r : ey→ex, ex←**ex** のとき、**ey** は **ex** に従属的という。
任意の **ey,ex**∈**E** で **E** が正規化集合の場合、
ey が **ex** に従属的で、かつ **ex** は **ey** に従属的でなければ、**E** は第三正規化集合であるという。
E={e1,e2,・・・en}のとき、定義により、可附番なので順番に自然数を与えることができる。
1 から N を与えた場合、
全単射 i を定義して i: n→xn が定義されるとき、
この xn をエンティティ・インスタンス en のインデックスと呼ぶ。
定義から明らかなように、en≠em ならば n≠m。
逆は成り立たない。
現実世界の実体と対応させて、
アトリビュート a1,a2,・・・, 第三正規化エンティティ e1,e2, ・・・,が定義できて、
これらが、クラス、インスタンスの定義と矛盾しない場合、
この対応をデータモデルと呼ぶ。

(3)注意

記法は必ずしも厳密ではない。また、一部に「アトリビュート」のように無定義語がある。

以上