# 1 モデル駆動開発とその周辺

## Introduction to Model-driven Development

山田　正樹　（有）メタボリックス
masaki@metabolics.co.jp

### モデル駆動開発以前

モデル駆動開発（Model-Driven Development，MDD）とは対象に関する知識を「モデル」に変換することによって行われるソフトウェア開発手法の総称である．「モデル」とは「対象のある側面を抽象化して表現したもの」である．したがってモデルは対象のすべての側面を忠実に表現しているわけではない（もしそうだとしたらそれはすでにモデルではなく対象そのものである）．また同じ対象に対してどの側面を取り上げるか，どのように表現するかによって複数の種類のモデルが存在し得る．たとえば地図は地形を対象としたモデルであるが，地形そのものではない．また測量のための地図か，ドライブのための地図か，人口分布を表すための地図かなどによって，同じ場所を表す多くの種類の地図が存在する．

従来のソフトウェア開発も問題領域の現実や欲求に関する知識を CPU の上で動作するある計算モデルに変換することと考えれば，実はモデル駆動開発であったということができるだろう（図 -1）．

しかし問題領域に関する知識を CPU の計算モデルに直接変換するようなソフトウェア開発は非常に困難であった．なぜならば問題領域と計算モデルの距離が非常に遠く，問題領域の特徴をよく保存したまま計算モデルに変換する方法がまれだったからである．現在では当たり前となっている高水準言語を用いた「プログラミング」によるソフトウェア開発手法は，それが始められた時点（1950 年代）では問題領域を CPU で実行可能な計算モデルに「自動」変換する「自動プログラミング」と見なされていたのである（図 -2）．

ここで変換されたモデルが妥当なものであるかどうかは実際にソフトウェアを実行することによって確認される．

しかしその後ソフトウェア開発が産業化し，より複雑／大規模で現実的な問題領域を扱う必要性が高まってきた時点（1970 年代）では，古典的なプログラミング言語ではそれらの必要性に対応しきれなくなっていた．そのための 1 つの解決方法が要求／分析／設計などの上流工程を充実させ，その工程の成果物をドキュメントとして表現することである（図 -3）．このドキュメントの表現方法はほとんどが自然言語によるものではあったが，確かにプログラミング言語よりは問題領域に近いモデルであった．

知識には暗黙的な知識（暗黙知）と形式的な知識（形式知）がある．暗黙知とは何らかの言語によっては表現することのできない知識（たとえば気持ちとか，行間，体験など）のことである．一方形式知とは何らかの言語によって表現することのできる知識である．

自然言語による問題領域のモデルには，問題領域の持つ暗黙知を自然言語の持つ暗黙知に対応づけることができるために情報の量と質を大きく落としてしまうことがないという特徴がある．しかしプログラムとして動作させようとするならば，いずれ暗黙知は形式知に変換されなければならない．

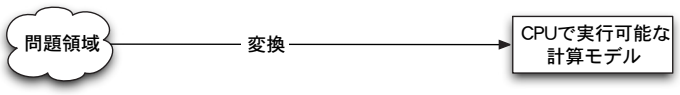


図 -1　モデル変換としてのソフトウェア開発

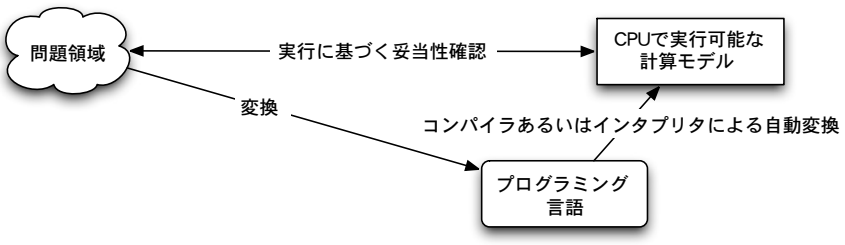


図 -2　プログラミング言語を介したモデル変換

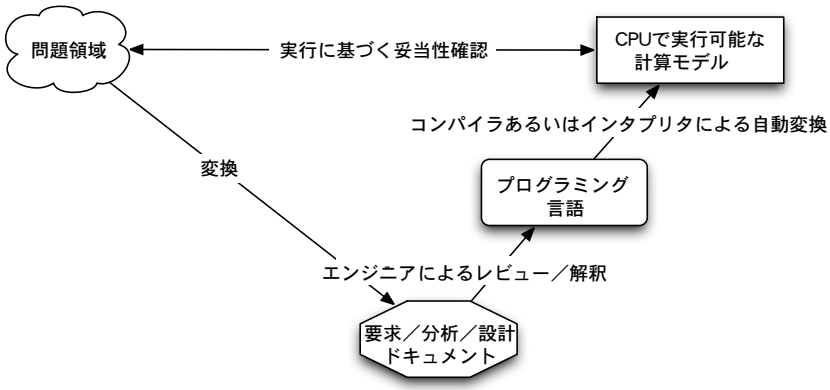


図 -3　ドキュメントを介したモデル変換

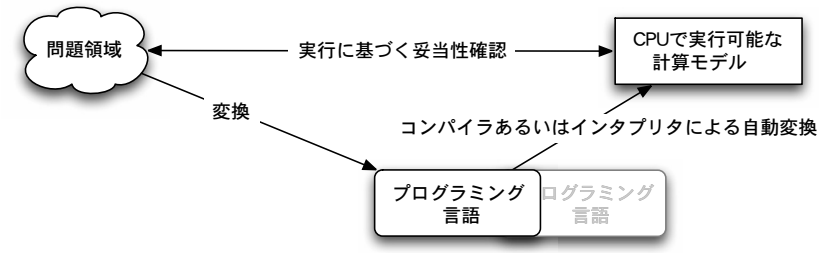


図 -4　より問題領域に近いプログラミング言語

自然言語によるモデルからプログラミング言語によるモデルへの変換の過程には大きな曖昧さと誤りが介在する可能性がある．

もう 1 つの解決方法は，プログラミング言語をより問題領域に近づけようというものである（**図 -4**）．エキスパート・システム，オブジェクト指向プログラミング言語，論理型言語などがその試みに相当する．

この方法が前者に比べて優れているのは，モデル（プログラム）が実行可能である，という点である．したがって検証や妥当性確認のしにくい要求／分析／設計ドキュメントの作成に膨大な工数を費やす必要がない．ただしプログラミング言語が相当高水準になったとしても，問題領域の複雑化／大規模化に追いつくことは難しかったし，プログラミング言語が高水準になればなるほどプログラム作成に高度なスキルが必要になっていった．

さて，1990 年代以降ソフトウェアは社会全体のインフラストラクチャとなり，ソフトウェア開発も新たな質的な展開を求められるようになった．その背景にあるのは

- 開発期間の短縮化
- ソフトウェアの複雑／大規模化
- ソフトウェアのオープン化（相互運用性，多くの準拠すべき標準，多様なユーザビリティ）

などである．

これらの諸問題に対する回答の 1 つが，開発プロセスの側面からはアジャイル開発プロセス，開発テクノロジの側面からはモデル駆動開発であり，両者の間には強い関連があると筆者は考えている．

本稿ではモデル駆動開発の概要と，アジャイル開発プロセス，ドメイン・エンジニアリングなどその周辺のトピックスとの関連について述べる．

## モデル駆動開発とは

ここで今までの歴史的な議論をふまえて，現在話題となっているモデル駆動開発を仮に次のように定義しておこう．

- 対象領域に関する知識を変換することによって最終的に CPU 上で実行可能なモデルとするソフトウェア開発手法である
- 対象の領域ごとに，異なる側面を抽象化した，異なる表現方法の複数のモデルを用いる
- モデルを一挙に直接変換するのではなく，段階的に複数回の変換を経る
- モデルはできる限り実行可能／検証可能である

特に中の 2 点が従来のソフトウェア開発手法では見られなかった，あるいはあまり重視されなかった点である（**図 -5**）．モデルをテキストとして表現するか，グラフとして表現するかはさしあたり重要ではない．

### モデル駆動開発と UML

最近モデル駆動開発が注目されている理由の 1 つは，もちろん Unified Modeling Language（UML）の普及によって「モデル」というものに対する理解が進み，余計な先入観がなくなっているからである．しかし，モデル駆動開発で用いられるモデルの記法や種類は UML に限られるわけではない．たとえばハードウェアを含めたシステム開発では本来の UML では定義されていないアー

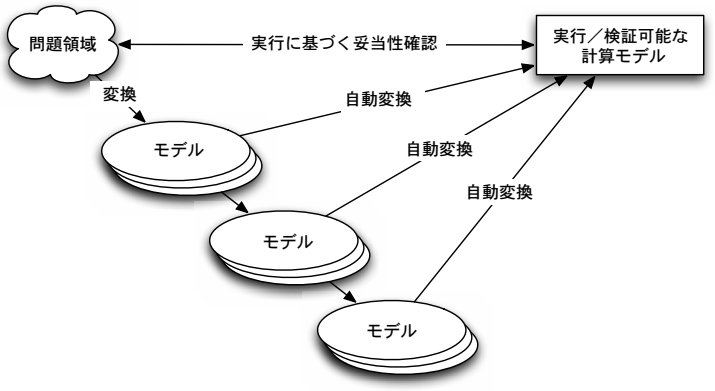


図 -5　モデル駆動開発

キテクチャ図が用いられることもあるだろう．

UML は Meta-Object Facility（MOF）をベースとした比較的強力な拡張性を持っているので，そのような場合でも UML の拡張として各種のモデルを追加していくことができる．また MOF をベースとしているので，追加したモデル同士あるいは追加したモデルと UML のモデルとの間の整合性を正しく定義してやることもできる．この場合 UML は単なる何種類かの図の表記法を定めたものというよりも，モデリング・プラットフォームであると考えたほうがよい．

現在 Object Management Group（OMG）によって標準化作業の最終段階にある UML2.0 では，現在使われている UML（1.x）に比べてモデル駆動開発を意識した方向づけが行われている．たとえば

- より厳密でコンパクトなメタ・モデル定義
- 計算（アクション）概念の完全な統合
- 拡張可能性の強化

などである．これが実現し普及すれば，現在アドホックにツールごとに実装されているモデル駆動開発手法が大きく進展するものと考えられる．

## 複数のモデル

モデル駆動開発においては，1 つの対象についてさまざまな側面から複数のモデリングを行う．たとえば古典的なオブジェクト指向モデリング（OMT，Shlaer-Mellor など）ではシステムを次の 3 つの側面からモデリングしている（手法によって名称は異なる）．

- 情報モデル－ 静的な構造を表す
- 状態モデル－オブジェクトのライフサイクルを表す
- 機能モデル－メソッドが行う処理を表す

同じように Unified Process では 4+1 ビュー（**図 -6**）を用いている．

また UML2 では次の種類のモデル表記法を提供している．

- Structrue Diagrams
  - Composite Structure Diagrams
- Activity Diagrams
- Interaction Diagrams
  - Sequence Diagrams
  - Communication Diagrams
  - Interaction Overview Diagrams
- Statemachine Diagrams
- Usecase Diagrams

これらの図の種類それぞれが対象をとらえる側面に対応していると考えてよい．

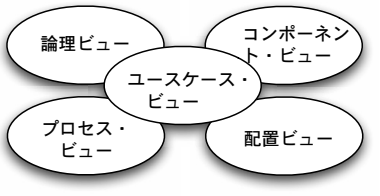


図 -6　4+1 ビュー

また実際のモデリング過程を詳細に観察すると，

- 動的モデリング／静的モデリング
- 具体的（インスタンス・レベル）モデリング／抽象的（クラス・レベル）モデリング
- 外側（インタフェース）のモデリング／内側（実装）のモデリング

を繰り返して，モデルを推敲しながら進化させていることが分かる．

このようにモデリングの過程ではさまざまな種類のモデルを利用する．モデリングで重要なのは単にいろいろな側面を表すモデルを表記できるということだけではなく，それらのモデルの間の一貫性／ 無矛盾性を何らかのかたちでチェックしたり，他のモデルの情報を元に別のモデルのモデリングを支援できたりすることである．

UML では UML 自体が MOF という小さくて厳密に定義されたコア部分によって定義されている．したがって UML の各図や各モデルは MOF のレベルで統一的に扱うことができ，モデルの間の関係や制約を MOF の言語を使って定義しておくことができる．

## モデルの変換

いわゆる「ハッキング」的ソフトウェア開発プロセスが，問題領域の知識を一気にプログラム言語で記述された計算モデルに変換したり，「ウォータフォール」的ソフトウェア開発プロセスが，問題領域の知識を要件／分析／基本設計／詳細設計といった（ほとんどの場合）自然言語による詳細化／洗練の過程を経るのに対して，モデル駆動開発では，モデリング言語を用いた段階的な詳細化／洗練の過程を繰り返す．

どのような段階でどのようなモデリングを行うかは，開発プロセスによってさまざまであるが，**図 -7** は典型的なモデリングの詳細化の過程を表している．ここでドメイン・モデルとは問題領域をモデル化したものであり，我々が開発するシステムの目的は現状（as-is）をあ

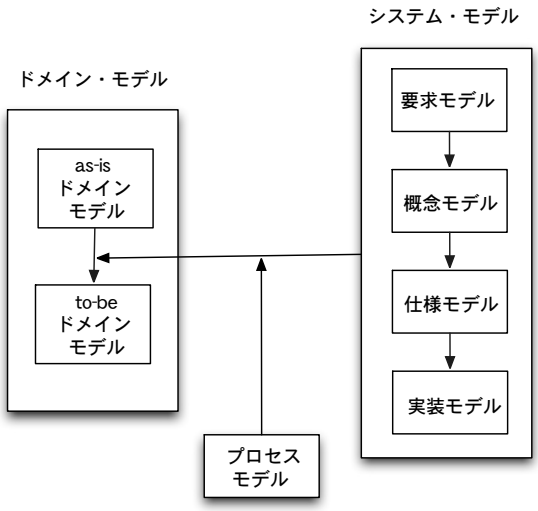


図 -7　典型的なモデリング・プロセス

るべき姿（to-be）に変えることである．この変換過程に作用するシステム・モデルとは解決領域をモデル化したものである．システム・モデルもその視点や段階に応じて要求モデル -> 概念モデル -> 仕様モデル -> 実装モデルのように段階的に詳細化／洗練される．またプロセス・モデルはドメイン・モデルからシステム・モデルへの変換過程（つまりモデル駆動開発のプロセス）そのものをモデル化したものである．

もっとも段階的といっても実際には一方向に順次整然とモデリングが進められるのでなく，さまざまなモデリングが並行に進められ，繰り返されるのが普通である．

OMG では，UML に基づいた Model Driven Architecture（MDA）と名付けたモデル駆動開発手法を提案している．MDA では Plathome Independent Model（PIM）と Plathome Specific Model（PSM）という 2 種類のモデルを想定する．PIM はプラットフォームに依存しない抽象的なモデルで，PSM はプラットフォーム上での実装を考慮した具体的なモデルである．ただしプラットフォームにもミドルウェア，オペレーティング・システム，CPU などさまざまなレベルがあり，あるレベルの PSM がより下位のレベルでは PIM と見なされる場合がある．このような場合には PIM から PSM への変換は複数段階に渡るモデル変換の連鎖になる．MDA では開発プロセスを PIM から PSM への変換を繰り返す過程であるととらえる（**図 -8**）．

モデル駆動開発においては，モデルからモデルへの変換ができる限り人手を介さず，機械的に行われることが従来のソフトウェア開発手法との大きな違いである．モデルからモデルへの変換を自動的に行うためにはいくつかの条件を満たしている必要がある．

(1) 個々のモデルの文法と意味論が厳密に定義されている
(2) モデルからモデルへの変換ルールが厳密に定義されている

UML あるいは UML を拡張したモデリング言語を

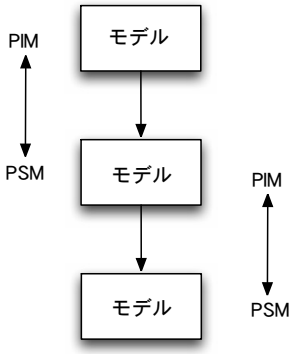


図 -8　MDA のモデリング・プロセス

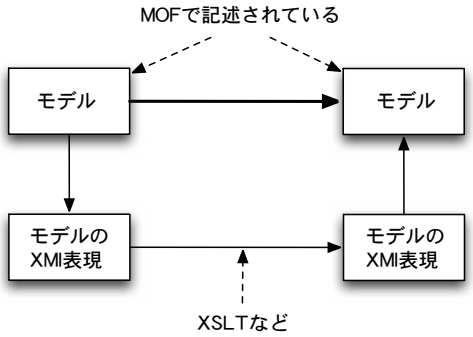


図 -9　XMI レベルでの変換

利用する場合には，モデルの文法や意味論の定義には MOF が使われる．つまり UML を使って記述されたモデルは最終的には MOF のモデルに翻訳される．MOF はモデルの記述と変換を行うのに十分厳密であると考えてよい．一方モデルからモデルへの変換ルールを定義する方法は現在のところ標準的な方法はなく，ツールによってさまざまなやり方が行われているが，基本的には MOF レベルでのマッピングを定義することになる．

最も単純な方法は定義されたテンプレートに従ってマクロ展開を行う方法である．単純な変換には単純なテンプレートが少数あればよいが，より複雑な変換や効率のよい変換を行おうとすると，さまざまな文脈や制約に応じて多数のテンプレートを用意し，適切なテンプレートを選択することが必要になる．たとえば Eclipse の Eclipse Modeling Framework（EMF）では，JSP（Java Server Pages）のサブセットであるマクロ・プロセッサを利用してモデル変換（主に MOF で記述されたモデルからソース・コードへの変換）を行うことができる．

あるいは MOF で記述されたモデルの XML 表現として XML Metadata Interchange（XMI）が OMG によって定義されているので，XSLT（XSL Translator）などを利用して XMI から XMI への変換を行うことも考えられる（**図 -9**）．このようなツールの例としては UMLX などがある．

今のところツールごとにこれらのアドホックな（MOF の枠の外の）モデル変換方法を採用している．したがってモデル駆動開発を行う開発者はツールごとに異なるモデル変換ルールの記述法を身につけなければな

らない．現在 OMG では MOF/QVT（Query/Views/Transformations）という仕様の策定が検討されている．MOF/QVT はその名前の通り，MOF で記述されたモデルに対する問合せや変換を行うためのフレームワークである．MOF/QVT が標準として制定されれば多くの準拠ツールが現れるものと期待される．

また実際のモデル変換では，単に 1 つのモデルと変換ルールが与えられただけでは十分実用になるモデル変換を行うことができない場合もある．そのような場合にはモデル中にヒントとなるような情報や制約（タグやマークなどと呼ばれる）を埋め込むことによって，変換の手助けとすることが多い．

## モデルの検証と実行

モデル駆動開発の大きな長所の 1 つは，ソフトウェア開発の比較的初期の段階で実行あるいは検証可能な成果物が得られる点である．ソフトウェアは他のエンジニアリング（メカ，エレクトロニクス，建築など）に比べて目に見えにくく，物理的な制約が少ないという特徴がある．そのためにどのようなソフトウェアを開発しようとしているのか，どんなに厳密に仕様を指定したとしても，実際に実行してみなければ顧客やユーザの意図をくみ取れなかったり，システムが予期しなかったような振舞いをする場合が多いのである．

現在のモデル駆動開発ではない通常のソフトウェア開発においてはモデルの記述はある意味でドキュメントの代替物あるいは補完物でしかない．モデル駆動開発においては，開発のできる限り初期段階からモデルを実行したり，検証したりするためにアクションや制約をモデル中に記述する必要がある．

UML では制約を記述するための言語として Object Constraint Language（OCL）という一階述語論理をベースにした言語が定義されている．OCL は計算メカニズムを含まない単なる論理式でしかないので，これだけではモデルを実行させることはできない．しかし必要なポイントに制約を記述することによって，モデルの正当性を検証することができる．主にモデル中に制約を記述するポイントは，静的構造の詳細の指定，振舞いの事前条件／事後条件などである．

〈code.1 OCL の記述例〉

```
context Person::getCurrentSpouse() : Person
# Person クラスの Person を返すメソッド
getCurrentSpouse()( 現在の配偶者 ) について
pre: self.isMarried = true
# 事前条件は対象オブジェクトが既婚であること
body: self.mariages -> select(m | m.ended = false).spouse
# 返値は対象オブジェクトの婚姻歴のうちでまだ続いて
いるものの相手と等しい（返値に関する事後条件）
```

モデル駆動開発において制約を記述することは，従来の形式的仕様記述を思い出させる．しかし大きく異なる点はモデル駆動開発においては対象となるシステム全体を形式的に記述するのではなく，全体はモデルとして半形式的に記述しておき，要所要所にのみ形式的な制約を加える点である．これは従来の形式的仕様記述による開発が非常に困難であったためにその有効性にもかかわらずあまり普及しなかったことに対する，プラグマティズムからの回答であると考えてもよい．

もう 1 つの形式的仕様記述の問題点は，記述された形式的仕様と現実に解決すべき問題領域との対応関係が多くの場合必ずしも明確にできないという点であった．つまり多くのソフトウェアの要求に対しては，どんな立派な形式的仕様が書けたとしてもそれが実際に問題解決になっているとは限らないのである．モデル駆動開発ではそれに対して，モデルを実際に実行させることによって現実の問題領域に対するモデルの妥当性を初期から繰り返し確認することができる．そのためには多かれ少なかれモデル中にアクション（実行指令）を記述する必要がある．

UML ではアクションを記述するための言語として Action Semantics が定義されている．Action Semantics はその名前の通り，意味論のみで文法は定義されていない．これはすべてのモデラやツール，問題領域を十分満足させるアクション言語の文法を定義するのは困難であろうという思惑からきている．しかしアクション言語の意味論とメタ・モデル，XML 表現などが規定されているので，Action Semactics を実装しているツール間の相互運用性はある程度保たれている．Action Semantics はある意味でアクション言語の仮想機械（VM）の定義を提供していると考えればいいだろう．

〈code.2 アクションの記述例〉[3)]

```
create object instance newPublisher of Publisher;
#Publisher クラスのインスタンス newPublisher を 1 つ作る
newPublisher.name = “Addison-Wesley”;
#newPublisher の属性 name に値を代入する
select many newBooks from instances of Book where
selected.copyright == 2002;
#Book クラスのインスタンスから copyright が 2002 年
のものを集めて newBooks とする
relate newBook to newPublisher across R1;
# 関連 R1 で newBook と newPublisher をつなぐ
```

モデル駆動開発においてアクションを書き始めると，通常のプログラミングとどこが違うのかという疑問が当然わき上がってくるだろう．基本的にはどちらもアクションを記述しているという点は変わらない．しかしモデル駆動開発におけるアクション言語は通常のプロ

グラミング言語に比べて非常に抽象度が高く，実装プラットフォームを意識する必要がない．たとえばUMLのAction Semanticsはデータフロー言語を暗黙の前提としている．したがって直接的な変数領域という概念はなく，メモリ管理について気にする必要はないし，並列性は自然に備わっているものと考えることができる．

その代わりにこのような高レベルの言語で記述されたアクションを，実際に使われているプログラミング言語や計算機アーキテクチャに直接変換するのは難しい．そのためにモデルに変換のためのヒントを「印付け」する場合がある．たとえばある集合を表す要素（属性や関連）の大きさが一定以上にならないという前提が許されるのならば，モデルから実装コードへの変換において，より効率的なコードを生成できるかもしれない．ただしこれらの印付けと元のモデルは明確に分離する必要がある．

またシステムのすべてをアクションで記述するのではなく，振舞いの記述が必要な部分についてのみアクションを追加するのもモデリング言語とプログラミング言語との違いである．プログラミング言語ではアクションも構造もプログラムの中にたたみ込まれているが，モデリング言語ではアクションと構造は記述上かなり明確に分離されている．UMLでアクションを主に記述するのは主にステートチャート図などの振舞い図が中心である．クラス図に詳細なアクションを書き込んでいくとすると高レベルなプログラミング言語と大して変わらないことになってしまう．

### メタ・モデリング

このように見てくると，モデル駆動開発では単なるモデリングだけではなく，どうやって自分たちにとって最適なモデルを記述できるモデリング言語を用意するか，どのようにしてモデルからモデルへの変換ルールを記述するかということが重要になってくることが分かる．モデリングをしながらモデリング言語を定義し，モデル間の変換ルールを定義していくのである．これらはみなモデルそのものではなく，モデルを取り扱うための技術である．

ここでモデリング言語の定義もモデル変換ルールの定義もモデルで行えばよい，というのがソフトウェア・エンジニアリングに携わっているものの自然なアイディアである．このようにモデルに関するモデルのことを「メタ・モデル」と呼ぶ．たとえばUML自身もMOFの上のメタ・モデルとして表現されている．それによってモデル駆動開発が技術的に大きな恩恵を受けているといっていいだろう．

メタ・モデルはもちろんOMGのような組織によって標準化されている必要がある．しかしすべての面にわたってメタ・モデルが整備されていることを期待するのは困難なので，実際のモデル駆動開発においては開発者がモデリングだけではなく，メタ・モデリングの能力を持っていることが（少なくともしばらくの間は）必要となってくるだろう．これは従来のソフトウェア開発でいえばソフトウェアを作るのにコンパイラを作る能力が必要であるというのに近い．しかしたとえばかつてUnixに習熟したソフトウェア技術者がsed, m4, yacc/lexなどの「メタ・ソフトウェア」を駆使することによって当時としてはきわめて高い生産性を達成していたことを考えれば，十分理にかなった話であるともいえよう．

## モデル駆動開発の周辺技術

ここではモデル駆動開発と関連するいくつかの技術的トピックについて簡単に述べる．

### アジャイル開発プロセスとの関連

アジャイル開発プロセスとは，高品質なソフトウェアを手早く無駄なく開発することを目的としたソフトウェア開発プロセスである．明確な定義はなく，かなり多様なプロセスを含んでいるが，主に繰り返し的でインクリメンタルなプロセスが多い．大規模でドキュメント中心のウォータフォール的開発プロセスの行き詰まりから，近年大きく注目を浴びている．本来の出自はオブジェクトのコミュニティで，1990年代末からさまざまな分野で広く知られるようになってきた．その中でもExtreme Programmingがよく知られており，テスト駆動開発など技術的な貢献も多い．

モデル駆動開発はアジャイル開発プロセスの1つであると考えられている（実際代表的なモデル駆動開発手法の1つであるxUMLの提唱者S. Mellorはアジャイル開発プロセス提唱者の集まりであるAgile Allianceに参加して，Agile Manifestに署名している）．その理由は以下の通りである．

- 要件や技術上の変更に対して強く，対処しやすい
  モデル駆動開発ではモデルという単一の対象を扱い，任意の時点で実行と検証が可能である．したがって要件の変更を成果物に直接的に反映させやすい．また実装に依存しない抽象的なモデルと実装を考慮した具体的なモデルがあるので，技術的な変更に対して可搬性が高い．

- 無駄な（＝最終成果物に直接寄与しない）工程を含まない
  モデル駆動開発では成果物はすべてモデルであり，最終成果物である実行モデルに段階的に変換される．その過程ではすべて実行と検証が可能である．実行や検証が不可能であったり，最終成果物に直接変換できないような中間成果物は生成しない．

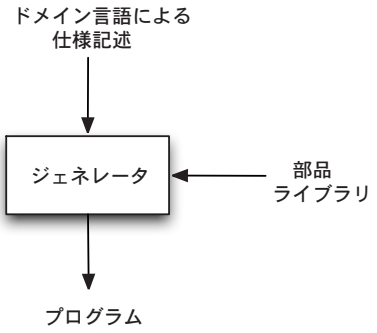


図 -10　ドメイン・モデリング

• 正しいことだけを効率よく効果的に行う
モデル駆動開発では成果物はかなり初期の段階から常に検証と妥当性確認がそのレベルに応じて行われている．したがって誤った成果物を後期にまで持ち込んでしまうことは少ない．またそれによって手戻りによる無駄な作業を減少させることができる．

以上のような意味で，モデル駆動開発は他のアジャイル開発プロセスと共通する性質を持っているといってよいだろう．しかし現在のところモデル駆動開発には他のアジャイル開発プロセスが大きな課題としているプロジェクト管理や，人間的な側面はあまり含まれていない．モデルを中心とした開発において，ソフトウェア開発組織が組織としてどのようなアジリティ（敏捷さ）を獲得していけるのか，これから検討が必要になると思われる．

### ドメイン・モデリングとの関連

ドメイン・モデリングとは「対象システム自身が本来持つ各種の性質や開発上のさまざまな知識を十分に分析し認識して組織化し，システムの開発に有効な，共通の対象領域に属する，用語，問題のとらえ方，システムの構造，システムの作り方などの，固有な概念構造を得るプロセスである」[2]．1990 年代にさまざまなドメイン・モデリングの考え方が提唱された（**図 -10**）．

おおざっぱにはモデル駆動開発は，ドメイン・モデリングの考え方を UML という標準的で共通した土台の上に移し，発展させたものと考えることもできる．ドメイン・モデリングにおけるドメインという考え方と，モデル駆動開発における複数のモデルという考え方は対応している．ドメイン・モデリングにおけるドメインごとのドメイン言語はアド・ホックにドメインに適切なものを定義するやり方であったが，モデル駆動開発では UML における MOF のような統一的で共通なベースを持つことができる．したがってモデル駆動開発においては，ドメイン間の関係はドメイン・モデリングにおけるよりも明確であり，ドメイン・モデル間の変換もよく定義することができる．

ドメイン・モデリングにおいてシステムをどのようにドメインに分割すればよいかは，非常に難しい問題であったが，その点はモデル駆動開発においても同様であり，あまり進展があるとはいえない．この点についてはモデル駆動開発においても今後研究が必要であろう．

一方モデル駆動開発では今のところ主に J2EE，.NET などのフレームワークを用いたビジネス・システムや状態遷移制御を中心とした一部の制御系システムを対象としており，多様なドメインに対して，モデリング・プロセスまで含めた十分な経験が蓄積されているとはいえない．この点に関してはすでに 10 年以上に及ぶ経験を持つドメイン・モデリングから学ぶべき点は多いものと考えられる．

## まとめ

モデル駆動開発は近年の UML などによるモデリング技術の普及と発達，標準化によって注目されるようになってきたソフトウェア開発手法であるが，その背景にはソフトウェア開発プロセスに対する新しい考え方や従来のドメイン・モデリングの考え方がある．モデル駆動開発が一般的な技術として広く用いられるようになれば，ソフトウェア開発の生産性や品質に大きな影響を及ぼすようになるものと考えられる．しかし，モデル駆動開発にはそれを支援するツールの存在が必要不可欠であると同時に，ソフトウェア開発に携わるメンバに対してモデリング能力，メタ・モデリング能力のような新たな能力が要求されるようになると考えられる．

今後はモデル駆動開発が適用可能なソフトウェア開発の範囲を広げていくのと同時に，モデル駆動開発にかかわる各種標準の整備，技術的な成熟，モデル駆動開発に適した開発プロセスの定義，支援ツールの大衆化と進化などが望まれる．

**参考文献**
1）UML Specifications , http://www.omg.org/uml/
2）伊藤他 : ドメイン分析・モデリング , 共立出版社（1996）.
3）Mellor, S. 他 :Executable UML, Addison-Wesley（2002）.
4）Frankel, D.: Model Driven Architecture, Addison-Wesley（2003）.
5）Warmer, J. and Kleppe, A. : The Object Constraint Language（2nd Ed.）, Addison-Wesley（2003）.

（平成 15 年 11 月 26 日受付）