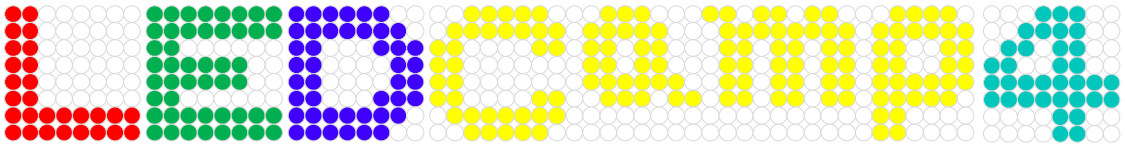

モデル駆動開発
Model Driven Development

細合 晋太郎

もくじ

.....0
1　モデル駆動開発とは.....2
1.1　いろいろなモデル駆動開発.....2
1.2　モデルとは.....3
1.3　抽象化と自動化.....4
1.4　MDD のトレードオフ.....5
2　モデル駆動開発の仕組み.....7
2.1　モデル駆動開発の工程.....8
3　モデル駆動開発の導入方法.....9
3.1　既存製品への部分適用.....9
3.2　既存製品の差分開発.....9
3.3　新規開発.....9
4　今回の MDD の仕組み.....10
5　おわりに.....13
理解度チェック.....13

1　モデル駆動開発とは

モデル駆動開発とは，従来のコードを主体とは異なり，モデルを主体とした開発手法です．ソフトウェア開発技術は抽象化の歴史と言っても過言ではありません．機械語からアセンブリ言語，C 等の高級言語，オブジェクト指向言語という風に，機械の言葉からヒトの言葉に近づけてきました．

ソフトウェア開発とは，ヒトの求める要求をコンピュータ上で実行できるプログラムに変換する作業であり，上記のような抽象化はソフトウェア開発工程における要求と実装の間のギャップを埋めより人の思考に近い概念で開発を行うためのものです．

モデル駆動開発は詳細設計から実装までの工程を自動化することで，プログラミング言語よりも抽象度が高い開発を行うことができます．

もちろん何もかも自動で行ってくれる訳ではなく，自動化のための仕組みを作り，ドメインに応じたモデルを定義する必要があります．モデル駆動開発は銀の弾丸ではありませんが，ドメインによっては非常に強力な武器になりえます．使いどころを見極めて導入を検討してみてください．

本稿では，モデル駆動開発の仕組みや導入方法などについて解説します．

1.1　いろいろなモデル駆動開発

モデルからコードを生成するという広義のモデル駆動開発には，多くの手法が含まれます．

- MDA（Model Driven Architecture）：OMG の提唱するモデル駆動開発のアーキテクチャです．主要な考え方として，MOF（Meta Object Facility）四層構造や，PIM（Platform Independent Model）/PSM（Platform Specific Model）などがあげられます．UML 仕様も MDA の枠組みで定義されています．
- MDE（Model Driven Engineering）：MDA をもう少し広義に捉え，モデルを用いた開発技術全般を指します．
- MBD（Model Based Development）：意味的には MDD と変わらないですが，一般に Matlab/Symlink といった連続系モデルを用いたコード生成を行う開発を指すこと

Shintaro Hosoai

が多いです．制御系の開発に非常に有効でエンジン制御など車載システムの開発に多く導入されています．

- MDD（Model Driven Development）：狭義には MDA に沿うモデル駆動開発を指しますが，MBD と対比して，離散モデルに基づくコード生成を用いるものを MDD と呼ぶことも多いようです．
- DSL/DSM（Domain Specific Language/Model）：ドメインに特化した独自の言語やモデルを作成し，それらからコード生成を行う開発手法です．DSL/DSM と並ぶ場合は，DSM は多義を持ちますが，DSL はテキストベースの言語，DSM は UML のようなグラフィカルなモデルを入力モデルとするものを指すことが多いです．両者を textual DSL/graphical DSL のように表記することもあります．
- DDD（Domain Driven Design）：モデル駆動とは異なりますが，ドメインモデルを設計の中心に据えるドメイン駆動設計という設計手法が近年活気づいています．モデル駆動開発の枠組みを設計するうえでもとても参考になる考え方です．

業界によっても MDD と MBD は入り混じっていることも多いため，その人の指すモデルが何かを確かめながら話すといいように思います．本書では UML を用いた MDD を中心に解説します．

1.2 モデルとは

対象をある観点と抽象度に基づいて情報を取捨し纏めたものをモデル，モデルを図面に落としたものをモデル図，モデルを作成することをモデリングといいます．例えば UML のクラス図は，ソフトウェアを構造の観点でクラスという抽象度で抽出したものです．同様にステートマシン図は状態という観点で，クラス内部の抽象度で抽出したものとなります．このように情報の取捨を行うことで，必要な情報のみに注力して考えることができます．また予め分析する観点を決めておくことで，設計の抜け漏れを防ぐことができます．

Shintaro Hosoai

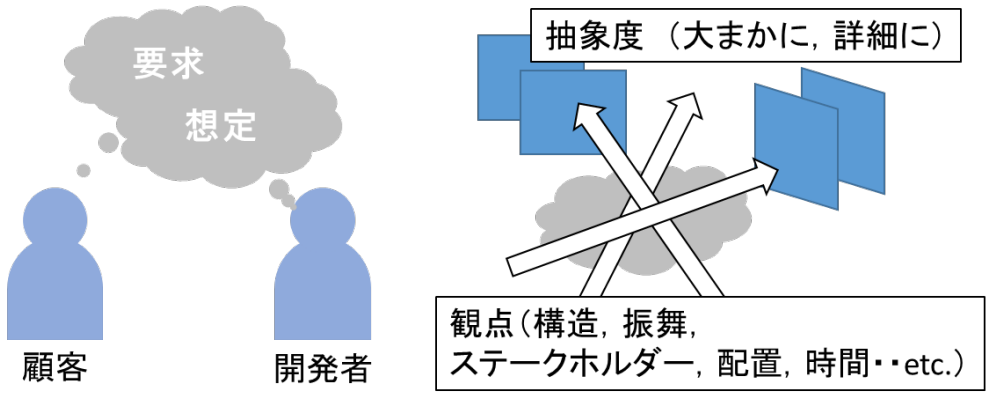

図 1 要求・想定のずれ，観点と抽象度

ソフトウェア開発では多くの場合，様々なステークホルダーが関わってきます．各々のステークホルダーが求めるシステムのイメージは必ずしも一致しているとは限りません．このためモデルの段階でシステムの全体構成や機能について合意を形成しておくことで，システムが完成してからの齟齬を軽減することができます．

1.3 抽象化と自動化

近年のソフトウェアの規模は，人が捉えるにはあまりにも膨大であるため，このような情報の取捨のためモデル化が必要となってきます．現在のスコープに不要な情報を排除することで，見たいものにだけフォーカスすることができます．

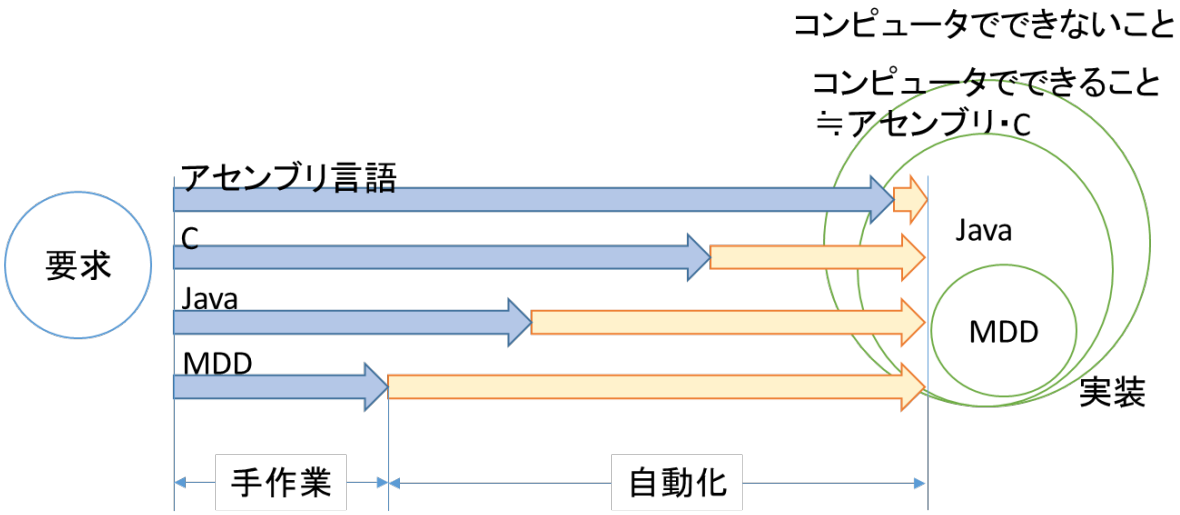

図 2 各言語における抽象化と自動化

ソフトウェア開発とは要求を満たすソフトウェアを作成することです．図 2 に要求から実装までの自動化の度合いを示します（感覚的に描いたものなので，厳密には異なります）．当然コンピュータでできないことは，実装のしようがありません．要求分析の段階で，実現不可能なものは弾いておきましょう．アセンブリ言語はコンピュータでできうる事はおおよそ実装することができますが，実装には多くの工数が掛かってしまいます．C 言語はアセンブリ言語よりも抽象度が高く，ほぼアセンブリ言語と同等の範囲を実装することができます（できないこともありますが・・）．Java 言語では，ダイレクトメモリアクセスができないなどいくつかの制約がありますが，C 言語よりも抽象度が高く，より少ない工数で実装を行うことができます．MDD ではさらにできる範囲は狭まりますがさらに少ない工数で実装を行うことができます．

1.4 MDD のトレードオフ

MDD は対象のドメインを限定することにより，自動化を行います．簡単に言ってしまうとそのドメインでよくあるパターンをテンプレートとして，そのテンプレートにモデルをはめ込むことで，コードを生成します．例えば Web システムでデータベースにアクセスするコードや OR マップのコードは，どのシステムでも似通っていますが，データベース構造に合わせて毎回作成する必要があります．このようなコードをクラス図で作ったデータベース構造から自動生成するイメージです．

当然，Web アプリを生成するための MDD の枠組みを組込みシステムに適用することはできないため，ドメインに応じた MDD の枠組みを新たに作成する必要があります．このように自由度と自動化はトレードオフの関係にあります．対象としているソフトウェアが今後どのようにスケールするかを考えながら，MDD の枠組みを作成する必要があります．

変換ルールの作成には，モデルからコードへ手動で実装するよりも多くの工数が掛かります．一度きりの開発しか行わないシステムや一度しか使わないモデルでは，モデル駆動開発を採用するメリットは少ないです．モデル駆動開発を導入する際には，この初期投資と何サイクルで投資が回収できるかのトレードオフをよく考えることが重要です．

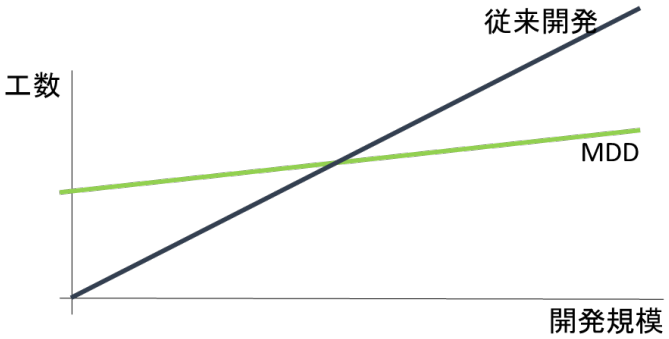

図 3 モデル駆動開発のトレードオフ

加えて，可変性のトレードオフも重要です．なんでも生成できる変換ルールを作るには無限の工数が掛かります．将来必要となる変更を見据えて，可変性の範囲を検討してください．

2 モデル駆動開発の仕組み

モデル駆動開発は，従来開発の工程の一部を自動化することにより，高い生産性を実現することができます．ここでは，従来開発との比較とモデル駆動開発の仕組みをみていきましょう．

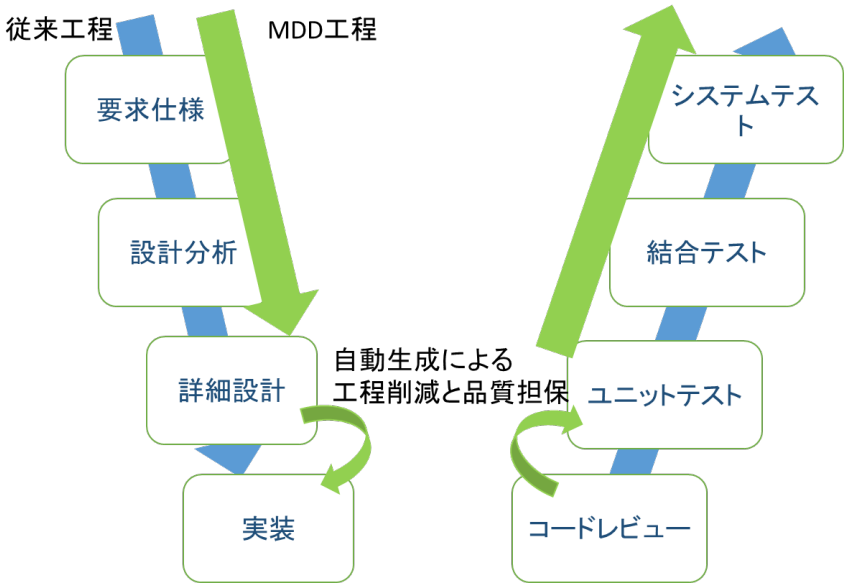

図 4 従来開発と MDD の開発工程

図 4 に従来開発の開発工程と，MDD による自動化される工程を示します．
従来開発では詳細設計を元に手動にてコードの実装を行いますが，MDD ではこの詳細設計のモデルからコードへの変換を定義しておくことで，自動化することができます．
コードレビューは主に詳細設計通りにコードが適切に実装されているか確認する工程ですが，自動生成では変換が適切に定義されていれば，モデルからコードへの変換では抜け漏れは起こりません．またコードテンプレートに従って生成されるため，コードフォーマットが崩れることもありません．また繰り返し利用され十分に実績のある変換定義であれば，ある程度テストも削減することができるでしょう．
このように人の手で行っていた実装という工程を変換として定義することで，人によるミスやブレを排除することができます．当然，変換の定義にバグが混入することもありますが，繰り返し利用される変換定義を洗練することでコード品質を高めることができます．

2.1 モデル駆動開発の工程

モデル駆動開発は開発のラインを構築する準備のフェーズと運用のフェーズに分けられます．準備フェーズでは，既存の開発工程を分析し，メタモデルの作成やモデルからコードへの変換を定義します．運用フェーズは図 4 の MDD 工程となります．

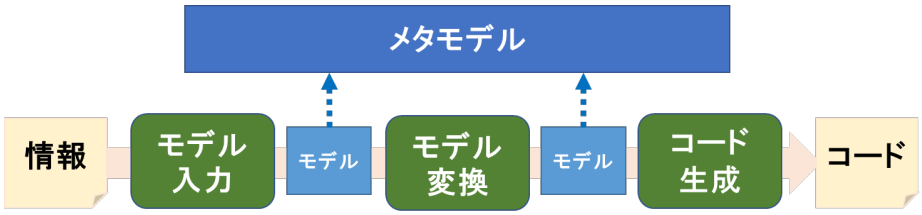

図 5 汎用的な *MDD* の主要なプロセス

図 5 に MDD の主要なプロセスを示します．準備のフェーズでは，このうちメタモデル・モデル入力・モデル変換・コード生成の部分を準備し，何かしらの情報からコードまでの変換の流れを作ります．運用のフェーズでは実際に情報を入力し，モデルに変換し，コードを生成します．

一方，UML を用いた MDD ではもう少しシンプルです．図 6 に UML による MDD のプロセスを示します．

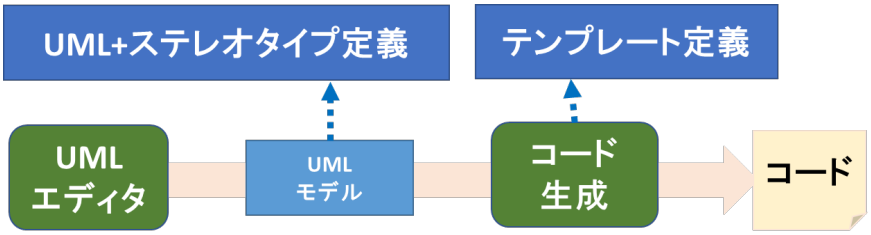

図 6 *UML* による *MDD* プロセス

準備フェーズでは，どのような UML とステレオタイプを用いるか検討し，コードを生成するためのテンプレートを定義します．運用フェーズではエディタで UML モデルを書き，そのモデルをテンプレートに当てはめることでコード生成を行います．

テンプレートを記述するには，予め生成したいコードのイメージと UML のどの要素を使ってコードを生成するか検討する必要があります．詳細なテンプレートの作りかたは 4 章を参照してください．

3 モデル駆動開発の導入方法

2 章で述べた通り，モデル駆動開発の導入（準備）には，多くの工数が掛かります．一気に導入するよりも，導入しやすい箇所から部分的に導入し徐々に慣れていくのがよいでしょう．本章では導入についていくつかの方法を見ていきます．

3.1 既存製品への部分適用

同じような構造のクラスが大量にある場合や，特定のルールに従って繰り返し記述するようなコードがある場合は，その部分から適用してみるのがいいでしょう．

似通ったコードの Diff をとってみて，共通部分と可変部分を探してみてください．可変部分を整理して，メタモデルとコードテンプレートを作成できれば，以後の繰り返し作業は自動化することができます．すべて自動化することが難しい場合は，できるだけ可変部分は小さくして，生成が難しい部分のみ手動で実装してもいいでしょう．場合によっては，DSL の方が小さい規模でさく導入しやすいでしょう．

3.2 既存製品の差分開発

基本的に 3.1 の作業とやることは変わりません．既存のコードや構造を分析し，共通部分・可変部分を抽出し，メタモデルとコードテンプレートを作成します．この際メタモデルをドメインモデルのように業務全体を整理したモデルまで分析・改善できれば，より高度な MDD を実現できるでしょう．

ここでもすべて自動化するのではなく，自動化しやすいパッケージやコンポーネントに限って導入していくとよいでしょう．

3.3 新規開発

新規開発で導入する場合，全く既存資産がない状態や，そのドメインの開発を行った経験のない状態での導入は難しいです．既存製品の延長上や新バージョンの立ち上げ，先行研究開発などで既存資産を再整理して導入するのが現実的であるといえます．

新規開発で導入する際は，既存のアーキテクチャに影響されず，一度製品全体のアーキテクチャを見直すチャンスでもあります．ドメインモデルをしっかりと作り，モデル駆動開発や今後の製品展開を見据えて，変更や保守に強いアーキテクチャを構成してください．

4 今回の MDD の仕組み

astah*は機能拡張のためにプラグイン機能を提供しています．プラグインを作成することで，astah*で編集中のモデル図へのアクセスや astah*にメニューを追加することができます．今回作成した MDD ツールもプラグインとして作成し，編集中のモデル図からモデル要素を抽出し，コードテンプレートに適用することでコード生成を行っています．（https://github.com/s-hosoai/astahm2t）

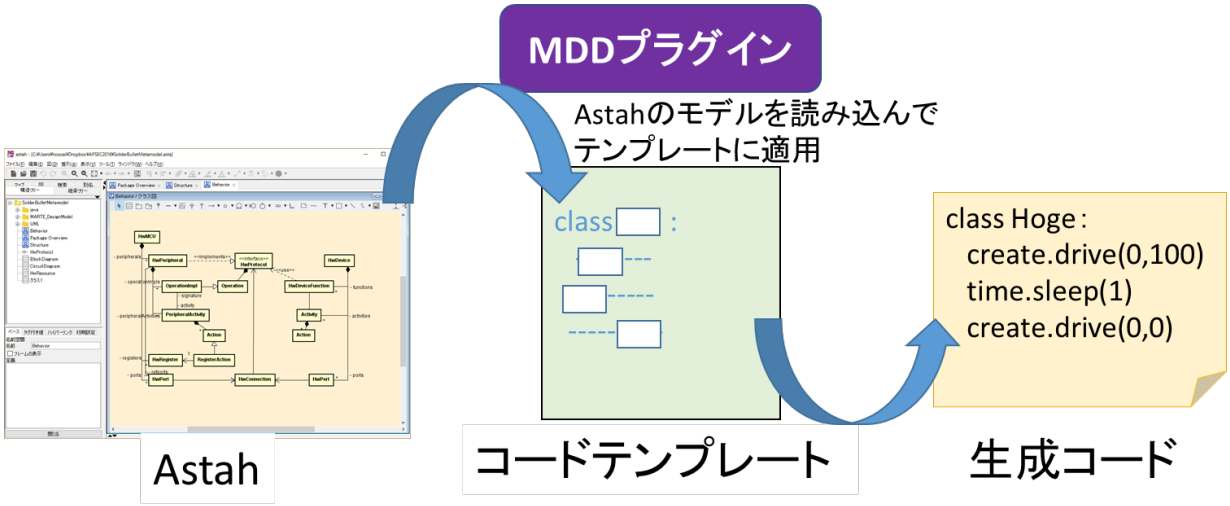

図 *7 Astah MDD プラグインのプロセス*

プラグインは通常 Java で作成するのですが，今回は Java のコードを生成できる軽量 VM 言語である Xtend を用いて実装しています．またテンプレート部分はスクリプト的にあとから編集可能にするため Groovy のコードテンプレート機能を利用しています．

MDD プラグインで取得したモデルは一旦ユーティリティクラスのインスタンスに保持しています．今回のテンプレートでは，クラス毎にファイルを生成するため，1 クラス

ごとにユーティリティクラスにモデル要素が渡されます．u.class でそのクラスモデルの実体を，u.name でクラス名という風にアクセスできます．

例えば，Python でクラスを生成するのであれば，

```
class #{u.name} :
    pass
```


といったテンプレートを作成すれば，#{u.name}の部分が astah*のクラス図で定義したクラスごとにクラス名.py というファイルが作成され，下記の内容が生成されます．

```
class  クラス名 :
    pass
```

もう少し複雑な例を見てみましょう．下記はステートマシン図からステートマシンのコードを生成する例です．少しコードが複雑なため，先に入力モデルと生成したいコードの構造を示します．

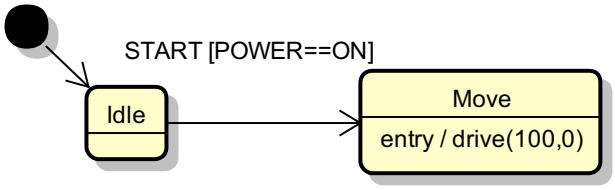

図 *8* 入力モデル

#生成コード

```
def doTransition(self, event):
    if “Idle” == self.currentState:
        if “START” == event
            if Power == ON :
                self.currentState = “Move”
                drive(100, 0)
    if “Move” == self.currentState:
        ・・・
```

doTransition メソッドは何かしらのイベントが発生したときに呼ばれるメソッドです．上から順に確認すると，現在状態が Idle 状態である時に，START というイベントを受け取り，Power==ON である時に，Move 状態に遷移し，entry アクションである

drive(100,0)を実行する．となります．（今回簡略化のため，状態とイベントは文字列として定義しています．

次にこのようなコードを生成するためのテンプレートを見てみましょう．テンプレートに入る前に UML のステートマシンのモデル構造を確認します．

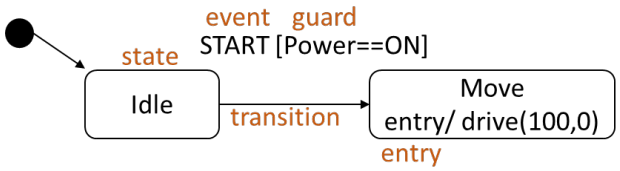

図 9 ステートマシンのモデル要素

赤字のところがモデルの要素名だと思ってください．テンプレートではこのような要素名を使ってモデルにアクセスしています．

#テンプレート

```
def doTransition(self, event):
<% for(state in u.states){%>
    if ${state}==self.currentState:
    <% for(transition in state.transitions){%>
        if ${transition.event} == event
            if ${transition.guard} :
                self.currentState = ${transition.nextState}
                ${transition.nextState.entry}
    <% } %>
<% } %>
```

<%%>で囲われた部分は，制御コードです．プログラムのように実行される箇所だと思ってください．ここではクラスの持つ状態をループで一つずつ取り出しています．

あとは，上記のステートマシンのモデル要素で示した通り，各要素名で要素を取り出し，コードに展開しています．

今回は解説のため，簡略化したテンプレートで解説しましたが，実際のテンプレートではもう少し複雑な処理を行っています．興味の沸いた方は，今回配布する astah*m2t プラグインのテンプレートフォルダを確認してみてください．ユーザディレクトリの.astah/plugins/m2t/template 以下にテンプレートのコードがあります．

5 おわりに

モデル駆動開発といっても堅苦しいものではなく，あくまで楽をする手段の一つです．身近なところから少しずつ使ってみて，面倒なことは全部コンピュータに任せる快適な開発者ライフをお過ごしください．モデル駆動開発について導入を検討されている方は，細合までご連絡頂ければご相談させて頂きます．

理解度チェック

□モデル駆動開発は何を開発の主体とする開発手法でしょう？

□モデル駆動開発には多くの手法がありますが，連続系モデルを扱う開発手法はなんでしょう？

□モデルは□□□と□□に基づいて情報の取捨を行います．空白の単語はなんでしょう？

□モデル駆動開発を実現するにあたり，いろいろなトレードオフを考える必要があります．どのようなトレードオフが必要か考えてみてください．

□モデル駆動開発は従来開発のどの工程を自動化するものでしょう？

Shintaro Hosoai