

MetaEdit+ ジェネレータの動作について

MetaEdit+のソースコード（ドキュメント）ジェネレータは基本的に、独自のスクリプトを使ってモデル内を巡回し、見つかったメタモデルのデータ（プロパティ）に応じて、プログラム言語を構成するための文字列を出力する事で実現しています。これが、出力するプログラム言語を選ばず、要求仕様やデザイン仕様すら出力できる理由です。決して、メタモデルに適切なプログラムソースが書き込まれていて、メタモデルの順番に合わせてそれをつなげている訳ではありません。

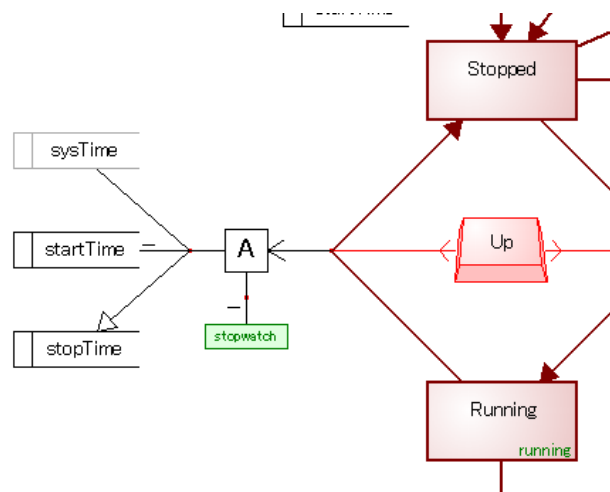
また、巡回方法は、“全てのメタモデルを順に見つける”、“特定のメタモデルに注目しそのメタモデルに接続されている全てのメタモデルに順に注目してゆく（これを繰り返す）”という方法があります。

IFを使ってメタモデルのタイプやプロパティの値に応じて処理を変更する（出力するソースを変更する）事も可能です。また、サブ関数のようなものを使用して、それを呼び出すことも可能です。

文字列の出力先は、任意の名前のファイル（.c、.h、.cpp、.java等）やMetaEdit+の標準出力が選択可能です。以上の内容をMetaEdit+のコード生成の基本動作として、以下フレームワークの呼び出し方法と#IFDEFに相当する部分の記述方法について説明します。

尚、この資料ではフレームワークの詳細な内容に関しては詳しく述べていません。別途用意した資料“ドメインフレームワークについて”に詳しく解説されています。併せてお読み頂く事でより深い理解が得られます。

フレームワークの呼び出しを記述するには、あるメタモデルがフレームワークに結びついていた場合に、そのプロパティを引数にして、フレームワークへの呼び出し関数を記述してゆくという方法が一般的です。



このモデルは、Up ボタンが押されたときに、Running ステートから Stopped ステートに移動し、同時に stopTime 変数に、sysTime 変数から startTime 変数を引き算した値を代入し、stopwatch アイコンを非表示にするという動作を表現したものです。（デジタル時計のストップウォッチ機能をモデル化しています）

このモデルに対応するジェネレータの定義は次ページの様になっています。

スクリプト中の青い文字が実際に出力される文字列（ソースコード）で、スクリプトの動作の説明は緑文字で挿入しています。オブジェクト、ロール、リレーションシップの名前に関しては、スクリプトの次のページに挿入した図に書き込んでいます

```
Report '_Actions'
```

```
foreach .Action
```

```
{  
    case a'; oid; ':'; newline;
```

```
    do ~ActionBody      全ての ActionBody ロールを探します
```

```
    { do >0 { '          '; subreport; '_' type; run; };
```

見つかった Actionbody に設定されているリレーションシップの名称と同じサブレポート（サブ関数）を呼び出します。（_icon と _set が呼ばれます）

```
    newline;
```

```
};
```

```
'          return null;'; newline;
```

```
}
```

```
endreport
```

```
Report '_Icon'
```

```
do ~Boolean; { 'icon'; id; do .Icon; {('"; id; "");}};
```

Boolean ロールに繋がった icon オブジェクトの名前を引数として、iconoff という関数を呼び出すようにソースを出力しています。（前の id はここでは off になります）

```
endreport
```

```
Report '_Set'
```

```
'set'; do ~Set.() {id;}; '(';
```

set ロールに繋がったオブジェクトの名前に set をつけた関数呼び出しを行うソースを出力しています。実際には _calcValue サブレポートを呼び出し、その中で、セットする値を引数として追加するようなソースを出力させています。

```
subreport; '_calcValue'; run;
```

```
);';
```

上で作成した関数呼び出しを終了する為の処理を出力させています。

```
endreport
```

```
Report '_calcValue'
```

```
do ~Get.()
```

```
{ 'get' id '0'
```

```
}
```

```
do ~(Minus | Plus)
```

```
{ '.me' type
```

```
do .()
```

```
{ '(get' id '0)'
```

```
}
```

```
}
```

```
endreport
```


newline; newline;

```
'      public Object perform(int methodId)
      {
          switch (methodId) {
';
subreport; '_Actions'; run;
subreport; '_DisplayFns'; run;
      }
      return null;
}; newline;

};
```

実際の動作に対する処理は
この部分で出力させている
メタモデルの接続応じて処
理を行わせている
実際には
subreport '_Actions' run で
上のソースの処理に移行

ちなみに、#IFDEF に関しては、#IFDEF を使ったバリエーションの影響を受けないようなメタモデル（全てのケースを包括するようなメタモデル環境）をつくり、#IFDEF のバリエーションを、モデル環境そのものに設定し、出力するソースを変更する方法で対応可能です。

具体的には、例えば#IFDEF を使って、複数のグレードや仕向地に合わせたプログラムを作っているような場合は、先ず全てのグレードと仕向地に対して適用できる様なメタモデルを作成します。モデル（図）そのものにプロパティを設定することができるので、そこにグレードや仕向地等のプロパティを設定しておき、そのプロパティによって、出力するソースを変える（IF 文を使って出力する文字列を変更する）という方法で対処できます。

全てのグレードや仕向地に合わせたメタモデルが作成できず、バリエーションによって完全にモデルの書き方が変わるような場合は、そのバリエーションに応じたモデルを作成することになるでしょう。（なるべく、そうならない様にメタモデルを作成することが最良です）