

VDMTools

VDM++ to Java コードジェネ
レータ
ver.1.0



How to contact:

<http://fmvdm.org/>

VDM information web site(in Japanese)

<http://fmvdm.org/tools/vdmttools>

VDMTools web site(in Japanese)

inq@fmvdm.org

Mail

VDM++ to Java コードジェネレータ 1.0

— Revised for VDMTools v9.0.6

© COPYRIGHT 2016 by Kyushu University

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement.

This document is subject to change without notice

目 次

1	導入	1
2	コードジェネレータ - はじめに	2
2.1	VDM++ Toolbox を用いたコード生成	2
2.2	生成コードとのインターフェイス	4
2.3	Java コードのコンパイルと実行	7
3	コードジェネレータ - 発展事項	9
3.1	VDM++ to Java コードジェネレータのオプション	9
3.2	暗黙のかつ予備的な関数 / 操作の実装	12
3.3	抽象クラスの生成	14
3.4	生成された Java コード部分の置き換え	16
3.4.1	実体	19
3.4.2	キータグのための規則	19
3.5	インターフェイスの生成	20
3.6	制限	24
3.6.1	言語の違いに起因する VDM++ 仕様の要件	24
3.6.2	サポートされていない構成要素	27
4	VDM++ 仕様のコード生成	30
4.1	VDM Java ライブラリ	30
4.2	クラスを生成するコード	32
4.3	生成された Java クラスの継承構造	34
4.4	コード生成型	37
4.4.1	匿名 VDM++ 型から Java への写像	37
4.4.2	VDM++ 型定義から Java への写像	40
4.4.3	不変数	42
4.5	コード生成値	43
4.6	インスタンス変数のコード生成	45
4.7	関数と操作のコード生成	46
4.7.1	明示的な関数および操作定義	46
4.7.2	予備的な関数操作定義	47
4.7.3	暗黙の関数と操作の定義	47
4.7.4	事前事後条件	47
4.8	式と文のコード生成	48
4.9	名称変換	48



5 並列 VDM++ 仕様のコード生成	49
5.1 導入	49
5.2 概論	49
5.2.1 コード生成	49
5.3 翻訳手引き	50
5.3.1 コア翻訳	50
5.3.2 手続きスレッド	51
5.3.3 定期スレッド	52
5.4 例題	52
5.5 制限	56
A コードジェネレータのインストール	58
B VDM Java ライブラリ	59
C DoSort 例題	60
C.1 クラス DoSort(Sort.rtf) の VDM+++ 仕様	60
C.2 クラス DoSort (DoSort.java) の Java コード	61
C.3 手書きの Java 主プログラム (MainSort.java)	63

1 導入

VDM++ to Java コードジェネレータ は、VDM++ 仕様からの自動的な Java コード生成を支援するものである。このコードジェネレータ では、VDM++ 仕様に基づいた Java アプリケーションを早期に実装する方法を提供する。

またコードジェネレータ は VDM++ Toolbox に対してのアドオン形式をとる。本書は *User Manual for the VDM++ Toolbox* [3] の拡張版であり、VDM++ to Java コードジェネレータ への導入を行う。

本書の構成は、以下の通りである:

第 2 章で VDM++ to Java コードジェネレータへの導入を行う。VDM++ Toolbox からコードジェネレータを起動する方法を述べ、生成される Java コードに対する処理を行う上での指針を与える。さらに、その Java コードをコンパイルし実行させる方法を説明していく。

第 3 章では、さらに 4 つの発展事項を提示している。VDM++ 仕様から Java コードを生成するときに、選択可能なオプションをまとめた。さらに、暗黙のあるいは予備的な関数操作定義をどのように取り扱うかを述べ、生成された Java コードを手書きコードと置き換えることが可能か検討する。最後に、コンパイル可能となるように翻訳され正しい Java コードとするため、VDM++ 仕様が満たさなければならない要件を並べていく。

第 4 章では、生成された Java コードの構造を詳細に示す。加えて、VDM++ と Java のデータ型間関連を説明し、VDM++ to Java コードジェネレータにより開発を行うとき、用いる名称変換を含めて構造上でなされる決まりについていくつか述べている。専門的にコードジェネレータを用いる場合には、この章を集中して学習するべきであろう。

最後に第 5 章では、並列仕様に対しどのようなコード生成を行うかについて説明がなされている。このような仕様に対しては、多重スレッド Java コードが生成される。用いる命令と同様、翻訳手法の概観が与えられる。

2 コードジェネレータ - はじめに

コードジェネレータの使用を始めるには、VDM++ 仕様が1つ以上のファイルに書かれている必要がある。

以下で Java コードジェネレータを説明するために、DoSort クラスの VDM++ 仕様を用いる。仕様は付録 C.1 に載せているが、配布で提供される Sort.rtf ファイル中にある。第 2.1 章では、VDM++ Toolbox を用いて VDM++ DoSort クラスに対して Java コードを生成する方法を説明する。第 2.2 章では、生成済み Java コードの先頭にアプリケーションを書きこむ方法を説明する。第 2.3 章では、アプリケーションをコンパイルし実行させる方法を示す。

読者には、第 2.1 章から第 2.3 章で記述されたステップを自身のコンピュータ上で実際に行ってみることを推奨する。

2.1 VDM++ Toolbox を用いたコード生成

ここでは、VDM++ Toolbox のユーザーインターフェイス画面で VDM++ to Java コードジェネレータを用いる方法を記述する。

VDM++ Toolbox をスタートすると、新規プロジェクトが Sort.rtf ファイルを生成するはずである。Java コード生成を行う前に、VDM++ 仕様は必要な要件を満たしていることが確認されていなければならない:

- 任意に選択されたクラスに対して正しいコード生成を行うためには、プロジェクト内 VDM++ 仕様のすべてのファイルで、構文チェックの成功していることが必要である。
- さらに、コードジェネレータは正しい型のクラスに対してのみコード生成が可能となる。¹まだ型チェックのなされていないクラスに対してコード生成しようとする、Toolbox によって型チェックは自動的に行われる。

User Manual for the VDM++ Toolbox [3] で述べているように、DoSort クラスを構文チェックおよび型チェックしよう。結果は図 1 で示される。

¹[2] で説明しているが、2つの適格性を有するクラスが存在する。本書においては正しい型とは可能な限りの適格性を有することを意味している。



図 1: 構文および型チェック後の管理画面


ここで、 (Generate Java) ボタンをクリックすることで、DoSort クラスに対するコード生成を行うことが出来る。一般的に複数ファイル/クラスの選択が可能で、その場合はそれらすべてが Java に翻訳される。

図 2 では、DoSort クラスに対してどのように Java コードを生成するかを示している。見て分かるように、DoSort.java と呼ばれる Java ファイルがコード生成されている。これは DoSort の Java クラス定義を含むものだ。DoSort.java ファイルが書き込まれるディレクトリは、プロジェクトファイルが置かれている場所となる。プロジェクトファイルが存在しない場合は、VDM++ Toolbox の始動ディレクトリにファイルは書き込まれる。

図 3 は、DoSort クラスに対する VDM++ 仕様のスケルトンおよび相当する Java 生成コードを表す。生成コードの各部分は、続く章で説明していく。Appendix C.2 には、ファイル DoSort.java の全体が載せられている。



図 2: DoSort クラスをコード生成中

VDM++ Toolbox のコマンドライン版からもまた、Java コード生成は可能である。VDM++ Toolbox は、コマンド `vppde` を用いることでコマンドラインから始動される。Java コード生成には、`-j` オプションを用いる。クラス `DoSort` のコード生成ならば以下のコマンド実行を行う:

```
vppde -j Sort.rtf
```

最初に仕様が構文解析される。構文エラーが検出されない場合は、仕様に対して可能な限りの適格性のための型チェックがなされることになる。もし型エラーが検出されなかったならば、最後に、仕様はたくさんの Java ファイルに翻訳される。記述された仕様に対して、`DoSort` クラス定義を含む `DoSort.java` ファイルが生成される。

注意: もし、仕様がいくつかのクラスを含むか コードジェネレータ でコマンドライン版を用いるならば、すべてのクラスは同時にコード生成されている必要がある。

2.2 生成コードとのインターフェイス

ここまでで、VDM++ 仕様から Java コードが生成される地点まで到達した。これからそのアプリケーションをコンパイルし実行するために、生成された `DoSort` クラスとのインターフェイスをどう記述するかを示す。


```

class DoSort

operations
  public Sort: seq of int ==> seq of int
  Sort(l) ==
    ...

functions

  protected DoSorting: seq of int -> seq of int
  DoSorting(l) ==
    ...

  private InsertSorted: int * seq of int -> seq of int
  InsertSorted(i,l) ==
    ...

end DoSort

```

VDM++

```

public class DoSort {

// ***** VDMTOOLS START Name=vdmComp KEEP=NO
  static UTIL.VDMCompare vdmComp = new UTIL.VDMCompare();
// ***** VDMTOOLS END Name=vdmComp

// ***** VDMTOOLS START Name=Sort KEEP=NO
  public Vector Sort (final Vector l) throws CGException{
    ...
  }
// ***** VDMTOOLS END Name=Sort

// ***** VDMTOOLS START Name=DoSorting KEEP=NO
  protected Vector DoSorting (final Vector l) throws CGException{
    ...
  }
// ***** VDMTOOLS END Name=DoSorting

// ***** VDMTOOLS START Name=InsertSorted KEEP=NO
  private Vector InsertSorted (final Long i, final Vector l)
                                throws CGException {
    ...
  }
// ***** VDMTOOLS END Name=InsertSorted
}

```

図 3: VDM++ と生成された Java DoSort クラス

最初は VDM++において、主プログラムを指定することから始める。

```
01  Main() ==  
02      let arr = [23,1,42,31] in  
03      ( dcl res : seq of int = [],  
04          dos : DoSort := new DoSort();  
05          res = dos.Sort(arr);  
06      )
```

ここで、上記の VDM++ 仕様と同様の機能性をもった Java の主プログラムを実装する。Java ファイルは主プログラムを含んでいるが、VDM Java ライブラリ package `jp.co.csk.vdm.toolbox.VDM` の全クラスをインポートすることから始めるべきである:

```
import jp.co.csk.vdm.toolbox.VDM.*;
```

これらのクラスに対して、完全に修飾された名称を記述する必要性が依然のこされる。VDM Java ライブラリ については、第 4.1 章にさらなる詳細を記述する。ここでは 1 つ 1 つ、上記に挙げた VDM 仕様を Java に翻訳していこう。

行 02 では整数の並びを指定する。Java に翻訳されることで、以下に続くコードを得るであろう:

```
Vector arr = new Vector();  
arr.add(new Integer(23));  
arr.add(new Integer(1));  
arr.add(new Integer(42));  
arr.add(new Integer(31));
```

Vector クラスは `java.util` パッケージに含まれる。DoSort クラスの Sort メソッドでは、Vector 型のオブジェクトが入力として要求される。

行 03 では `seq of int` 型の変数 `res` を宣言し、後でソートされた整数列を含めるために用いられる。これに対応する Java コードは次の通り:

```
Vector res = new Vector();
```

DoSort クラスの Sort メソッドを呼び出す方法を示そう。行 04 では、DoSort クラスのインスタンスに対してオブジェクト参照 `dos` を宣言し、行 05 では、引数としての整数

列 `arr` と共に `DoSort` クラスの `Sort` メソッドを呼び出している。結果は `res` に代入される。Java に翻訳されることで、以下のコードを得る:

```
System.out.println("Evaluating Sort("+UTIL.toString(arr)+"):");
DoSort dos = new DoSort();
res = dos.Sort(arr);
System.out.println(UTIL.toString(res));
```

VDM Java ライブラリ の一部である `UTIL.toString` メソッドを用いて、VDM 値の ASCII 表現を含む列を得る。このメソッドはここでは、実行中の標準出力に対する関連ログメッセージの印刷に用いられている。

上記に挙げた Java コードは、生成された Java コード内メソッドで起こされた例外を取り扱うために、`try` ブロック内に書かれていなければならない。`try` ブロックは `catch` clause に続き、これらの例外を捉えて処理を行う。生成された Java コードによって起こされる例外はすべて `CGException` クラスのサブクラスであり、これはまた VDM Java ライブラリの一部である。このように次の `catch` 文が可能である:

```
try {
    ...
}
catch (CGException e){
    System.out.println(e.getMessage());
}
```

前述の主プログラムは、`MainSort.java` という名のファイルに実装され、これは付録の [C.3](#) に全体が載せてある。

2.3 Java コードのコンパイルと実行

主プログラムを手書きすることで、Java コードのコンパイルと実行が可能である。

VDM++ to Java コードジェネレータ のこの版で生成された Java コードは、Java 開発キット 1.3 版と相性がよい。

主プログラムは次でコンパイル可能である:

```
javac MainSort.java
```

CLASSPATH 環境変数が VDM Java ライブラリすなわち VDM.jar ファイルを含めていることを確認しよう。もし Unix Bourne シェル または 互換性のあるシェルを用いているのであれば、これは以下のコマンドで行うことができる:

```
CLASSPATH=VDM_Java_Library/VDM.jar:$CLASSPATH
export CLASSPATH
```

VDM Java ライブラリ がインストールされたディレクトリ名称を VDM_Java_Library と置き換えよう。

もし Windows ベースのシステムを使用しているのであれば、CLASSPATH 環境変数は autoexec.bat において、あるいはコントロールパネルの System アイコンから、更新できる。Windows に対しては、区切り文字は “:” ではなく “;” を用いなければならないことに注意しよう。

主プログラム MainSort は、これで実行可能である。この出力を以下に載せる。

```
$ java MainSort
Evaluating Sort([23, 1, 42, 31]):
[1, 23, 31, 42]
$
```

この章では、どのように コードジェネレータを使用するかについて、簡単な導入を示した。以下の章ではもう少し詳細に コードジェネレータ の様々な局面を記述する。以降は、生成コード部分の提示は常に、論点に関連する部分のみのテキスト提示とするので注意しよう。

3 コードジェネレータ - 発展事項

第 2 章では コードジェネレータ への簡単な導入を示した。この章は、以下に続く質問への答えを与えることになる:

- VDM++仕様から Java コードを生成するときに、どのようなオプションを使用できるか? (第 3.1 章)
- 仕様が暗黙または予備的な関数 / 操作を含める場合には、何が行えるか? (第 3.2 章)
- 生成された Java コードを手書きコードと置き換える可能性は? (第 3.4 章)
- VDM++ 仕様がコンパイル可能な正しい Java コードに翻訳されるために満たすべき要件は何か? (第 3.6 章)

3.1 VDM++ to Java コードジェネレータのオプション

VDM++ 仕様から Java コードを生成するとき、生成コードに影響を与える 1 つ以上の次のオプションを選択することができる。利用可能なオプションを見るために、図 4 に示すように、オプションメニューから *Java Code Generator* エントリーを選択しよう。

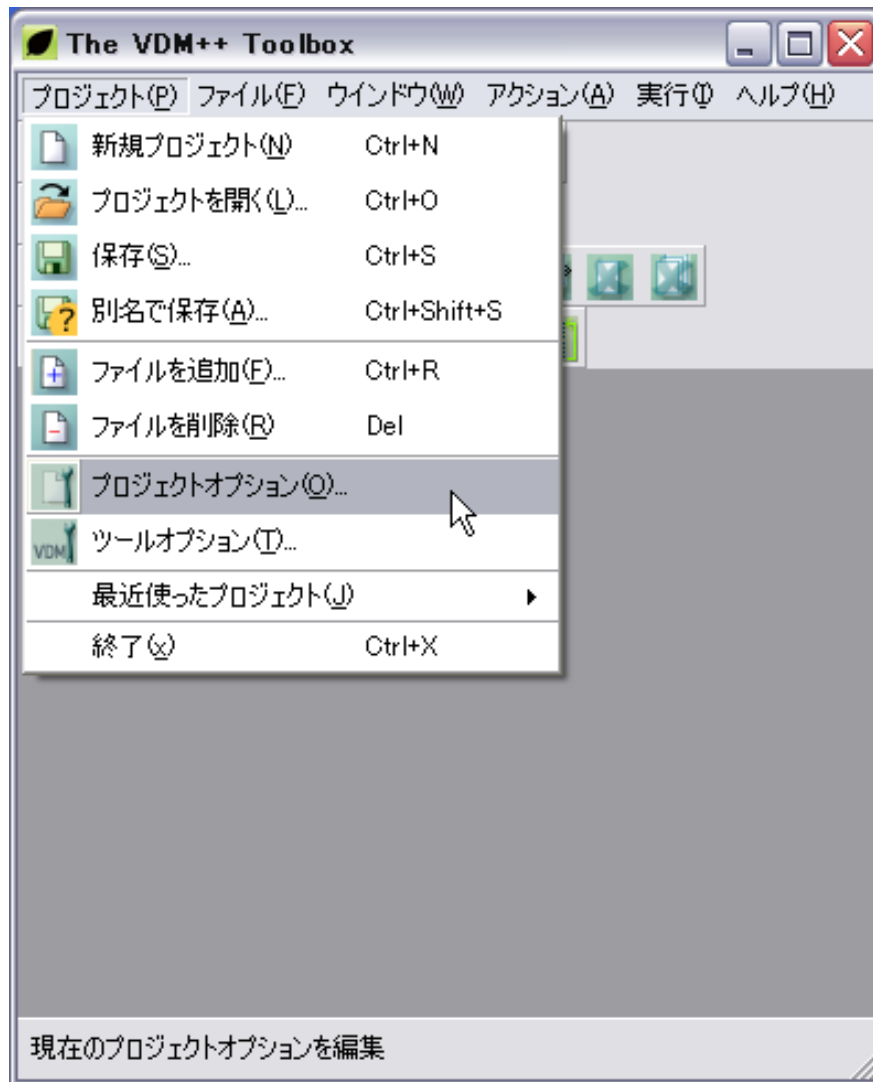


図 4: Java コードジェネレータのオプションの選択

図 5 に示されるように、コードジェネレータにおいて様々なオプションが利用可能である。これら各オプションは以下に記述している。これらオプションすべてが、コードジェネレータのコマンドライン版でも利用可能であることに注意しよう。以下の各オプション名称の後の括弧内に、適切なフラグを提示している。既定の動作もまた、既定で与えられたオプションで指定された動作が用いられないという意味の “off” や、動作が用いられるという意味の “on” に記載されている。

(-s) 型以外に対してスケルトンのみのコード生成 スケルトンクラスを生成するために、

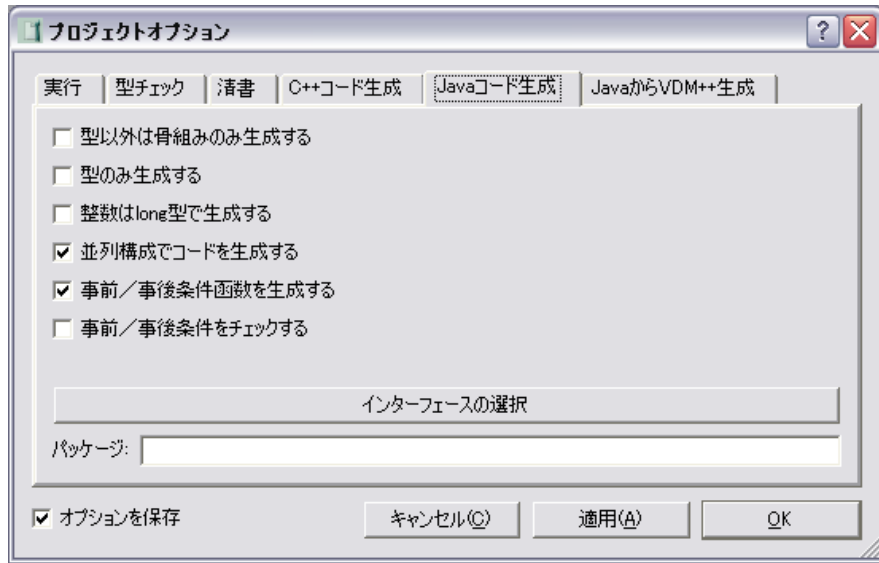


図 5: Java コード生成のためのオプション

このオプションを指定する。スケルトンクラスはすべての型、値、インスタンス変数定義を含むが、空の関数や操作の定義は含まないクラスである。既定: off

Code generate only types (-u) 型のみをコード生成 (-u) VDM++ 型定義に対して Java コードを生成したいというだけのとき、このオプションを指定する (つまり 関数、操作、インスタンス変数、値、は生成されない)。既定: off.

Longs (-L) として整数をコード生成 このオプションを使用することで、VDM++ 整数値と変数を Java の Integer ではなく Long として生成が可能である。既定: off

並列構成要素に伴うコードのコード生成 (-e) このオプションは、コードジェネレータが並列支援を含めるコード生成を行うことを強要するために用いられる。この詳細は第 5 章を参照のこと。既定: on.

事前事後の関数/操作のコード生成 (-k) 事前事後条件に対する Java メソッドやそれらの不変数をコード生成するために、このオプションを指定する。既定: on

事前および事後条件のチェック (-P) このオプションを、関数の事前事後条件のインラインチェック生成のために指定する。チェックが失敗した場合に例外が発生する。この意味は、事前事後条件としての前述のオプションが、コンパイル可能なコードのために生成されるべきであることを含めている。既定: off.

パッケージ (-z) *packagename* コードジェネレータの既定の動作は、ディレクトリ内に生成された Java ファイルを書き出すことであり、このディレクトリとはプロジェ

クトファイルが置かれている場所か、あるいはプロジェクトファイルが存在しない場合は VDM++ Toolbox がスタートした場所である。ファイルは名称のない既定パッケージの一部である。生成された Java クラスを含むはずの指定パッケージを生成するために、このオプションを指定する。コードジェネレータは、生成ファイルを含めるため与えられたパッケージ名称を使った新しいディレクトリを作成し、生成ファイルには適切な package 文が含まれることになる。

インターフェイスの選択 (-U) Java インターフェイスとして生成されるべきクラスを選択する。詳細は第 3.5 章を参照。

コマンドラインから VDM++ Toolbox をスタートさせる場合、以下のコマンドが用いられる必要がある:

```
vppde -j [options] specfile(s)
```

3.2 暗黙のかつ予備的な関数 / 操作の実装

暗黙の関数 / 操作および予備的な関数 / 操作 (“is not yet specified” で指定されたもの) が、コードジェネレータによっても同じ方法で取り扱える。予備的な操作定義を含む以下の VDM++ クラス定義を見よう。

```
class A
operations
op:() ==> int
op() == is not yet specified;
end A
```

このクラスは以下のように生成されることになる:

```
public class A {
    protected external_A child = new external_A(this);
    private Integer op () throws CGException{
        return child.impl_op();
    }
};
```


上述のコードから見てとれるように、クラス AA は型 external_A の保護されたインスタンス変数 child を含める。これは、暗黙の関数 / 操作または予備的な (“is not yet specified” で指定された) 関数 / 操作を含めるすべてのクラスに当てはまる。クラスがこれらの定義をいくつか含めるとしても、その外部クラスは1つのインスタンスしか存在しないはずである。

op メソッドは、このインスタンスの impl_op という名のメソッドを呼び出すことになる。² impl_op メソッドの結果は op メソッドの結果として返される。

したがってクラス external_A にメソッド impl_op を実装することは、ユーザーの責任である。メソッド impl_op の入出力パラメータは、メソッド op のそれと同じでなければならない。

もし VDM++ クラスが1つ以上の暗黙の関数 / 操作または予備的な関数 / 操作 (“is not yet specified” で指定されたもの) を含むとすれば、すべてのメソッドが external_<CLASSNAME> に実装されていなければならない。

ユーザーの外部クラスファイルをユーザーが実装するのを容易とするために、コードジェネレータ はこれに対する external_A.java ファイルを生成する。このファイルの支援で、生成された Java コードはコンパイル可能となる。しかしながら予備的な関数が呼ばれた場合は、実行時エラーが起きることになる。external_A クラスを含む external_A.java ファイルは、以下におかれている。

```
public class external_A {
    A parent = null;
    public external_A (A parentA) {
        parent = parentA;
    }
    public Integer impl_op () throws CGException{
        UTIL.RunTime("Preliminary Operation op has been called");
        return new Integer(0);
    }
};
```

external_A クラスを実装するための最も簡単な方法は、テンプレートクラスを修正すること、つまりユーザーは次のコードに対して、ユーザー定義コードに生成コードを置き換えることが可能な通常の方法で、置き換えを行う必要がある。

```
UTIL.RunTime("Preliminary Operation op has been called");
```

²impl は “Java で実装されるべき” ことを表す。

```
return new Integer(0);
```

(詳細は第 3.4 章を参照)

注意したいのは、外部クラスに対して生成されたコンストラクタが、入力パラメータとしてクラス A のインスタンスを取り入れ変数 `parent` に代入することである。この方法で、予備的な操作定義の実装はクラス A のパブリック状態にアクセス可能である。クラスの内部状態に作用することが許されないため、予備的な関数に対応する Java メソッドがこのコンストラクタを用いることになる。しかしそれらはある操作を呼び出すことで、間接的に内部状態に作用が可能となる。

暗黙的に定義された関数と操作は、“is not yet specified” 節を含む予備的な関数や操作の仕様と同じ方法で取り扱われる。

注意したいのは、外部クラスが暗黙および予備的な操作と関数を含めることが可能であることだ。生成された定型書式では、生成された次のような実行時エラーメッセージで区別可能である：

```
UTIL.RunTime("Preliminary Operation op has been called");
```

これは `op` という名の予備的な操作定義に対するものであり

```
UTIL.RunTime("Implicit Function f has been called");
```

これは `f` という名の暗黙の関数定義に対するものがある。

3.3 抽象クラスの生成

VDM++ クラスは、予備的な関数あるいは操作の定義を含むかあるいは抽象クラスのサブクラスであるならば抽象なものであり、継承されている抽象の関数や操作に対しての実装は提供しないものとなる。このように抽象であることは、VDM++ クラスの間接特性である。

一方で、Java は抽象クラスの基本概念を提供している。したがって、Java コードを生成するときに抽象と識別される VDM++ クラスは、抽象 Java クラスとして生成されることになる。たとえば、以下の VDM++ クラス A、B、C を考察してみよう：

```
class A

instance variables
  protected m : nat := 1

operations
  public op : nat ==> nat
  op(n) == is subclass responsibility;

functions
  public f : int -> int
  f(i) == is subclass responsibility

end A

class B is subclass of A

operations
  public op : nat ==> nat
  op(n) ==
    return m + n

end B

class C is subclass of B

functions
  public f : int -> int
  f(i) == i + 1

end C
```

クラス A は予備的関数や操作を含み、そのため抽象である。したがって次のようなコード生成がなされる:

```
public abstract class A {

  protected Integer m = null;
  public abstract Integer op (final Integer n) throws CGException;
```

```
public abstract Integer f (final Integer i) throws CGException;

}
```

クラス B は抽象クラス A を継承し、関数 f の実装を提供していない。したがって抽象でもある:

```
public abstract class B extends A {

    public Integer op (final Integer n) throws CGException {
        return new Integer(m.intValue() + n.intValue());
    }
}
```

最後は、クラス C は B から継承し f の実装を提供するためにある。したがって通常クラスである:

```
public class C extends B {

    public Integer f (final Integer i) throws CGException{
        return new Integer(i.intValue() + new Integer(1).intValue());
    }
}
```

3.4 生成された Java コード部分の置き換え

標準的な応用として、生成コードに対し、例えば外部ライブラリや手書きコードといった他コードと相互作用を行う必要が生じてくるだろう。このような相互作用を手助けするため、生成コードの修正が可能で、しかも コードジェネレータ が再実行されてもそれらの修正が上書きされない方法で、可能である。

これは、キータグの使用を通して達成される。これらは生成された Java コードにおいてはコメントであるが、そのコードの比率が上書されるべきか否かを コードジェネレータ が決定するときに用いるものである。

たとえば以下の例題を考えよう:

```
class Date
```

```
types
  public Day = <Mon> | <Tue> | <Wed> | <Thu> | <Fri> | <Sat> | <Sun>;
  public Month = <Jan> | <Feb> | <Mar> | <Apr> | <May> | <Jun>
    | <Jul> | <Aug> | <Sep> | <Oct> | <Nov> | <Dec>;
  public Year = nat

instance variables
  d : Day;
  m : Month;
  y : Year

operations

  public SetDate : Day * Month * Year ==> ()
  SetDate(nd,nm,ny) ==
  ( d := nd;
    m := nm;
    y := ny );

  public today : () ==> Date
  today() ==
    return new Date()
end Date
```

VDM++ も VDM++ Toolbox も時間の基本概念を持たないため、today に完璧な仕様を与えることはできない。生成コードで today は次のようになる:

```
// ***** VDMTOOLS START Name=today KEEP=NO
  public Date today () throws CGException{
    return (Date) new Date();
  }
// ***** VDMTOOLS END Name=today
```

関数定義の上下のコメントは、この関数に対するキータグに相当する。キータグでは以下の情報が得られる:

- タグが適用される実体の名称 (実体を構成するものは次で述べられる)。テキスト Name=の直後に置かれる。

- この実体が保存されるべきか上書きされるべきかを示すフラグ。KEEP=の後にテキストで与えられる。もし NO ならば実体は上書きされる; YES ならば保存される。ファイル生成時の既定は NO。

現在の日付を実際通りに戻すように、この関数を修正したいと仮定する。これは、Java 開発キットの一部として提供されている Calendar クラスを用いて可能である。

```
// ***** VDMTOOLS START Name=today KEEP=YES
public Date today () throws CGException{
    Calendar c = Calendar.getInstance();
    Date result = new Date();
    Object td = new Object(), tm = new Object();
    switch (c.get(Calendar.DAY_OF_WEEK)){
    case Calendar.MONDAY:
        td = new quotes.Mon();
        break;
    ...
    }
    switch (c.get(Calendar.MONTH)){
    case Calendar.JANUARY:
        tm = new quotes.Jan();
        break;
    ...
    }
    result.SetDate(td, tm, new Integer(c.get(Calendar.YEAR)));
    return result;
}
// ***** VDMTOOLS END Name=today
```

最初にキーフラグが YES に変化してしまっていることに注意しよう。これは変更が保存されたことを保証している。関数本体はしたがって通常の Java コードであり、任意の外部クラスで用いることができる。

現存の実体の変更に加えて、Java ファイルに新しい実体の追加ができる。既定の toString メソッド (java.lang.Object から継承) を日付に適用したものに置き換えたいと仮定しよう。以下をクラス定義に追加できるだろう。

```
// ***** VDMTOOLS START Name=toString KEEP=YES
public String toString(){
```

```
        return d.toString() + m.toString() + y.toString();
    }
    // ***** VDMTOOLS END Name=toString
```

3.4.1 実体

キーブタグを用いて保存できた生成 Java ファイルにおいて、実体は1つの領域である。これは以下の1つとなる可能性がある:

- トップレベルのクラス要素変数。
- トップレベルのクラスメソッド (コンストラクタを含める)。
- 内部クラス。
- インポート宣言の集まり。
- パッケージ宣言。
- ヘッダーコメント つまりファイルの先頭領域には、たとえばバージョン管理情報といったコメントを置くことができる。

キーブタグは、インターフェイスとして生成されたクラスと共に用いられることもあることは注意しよう (第 3.5 章参照); この場合も同じ規則が適用され、クラスの代わりにインターフェイスが読み込まれる。

事前に定義された3つのタグ名称: ヘッダーコメントのための `HeaderComment`、パッケージ宣言のための `package`、インポート宣言のための `imports`、は生成されたファイル中でよく現れる。

3.4.2 キーブタグのための規則

キーブタグを使用するときは以下の規則に従わなくてはならない。

- 各タグ名称は唯一のものでなければならない。
- タグは同一レベルでなければならない、つまりタグのネストは不可能である。
- クラス定義の外部のみで現れる可能性のあるタグは、`HeaderComment`、`package`、`imports` である。

- 付加された実体はクラス定義の 内部に、それも先頭部分に現れなければならない。このように、たとえばある関数がクラス内部に追加された場合、クラス内部全体が YES とタグづけされなければならない。
- タグ保持の構文では、case 文と空白は細心の注意を払うべきである。正確に後ろを続けていく必要がある。

これらの規則に従わないとき、コードは上書きされてしまう可能性がある。しかしながら元のファイルは常にバックアップされるため、これが必ずしも致命的となることはない

3.5 インターフェイスの生成

コードジェネレータ は Java インターフェイス [4] の生成を可能としている。VDM++ クラスは以下の条件に適應する場合は、インターフェイスとして生成される可能性がある:

- このクラスで定義された関数と操作のすべてが、本体 is subclass responsibility を含む。
- インスタンス変数を含まないクラスが、このクラス中に定義されている。
- このクラスで定義されたすべての型が、パブリックである。
- このクラスで定義されたすべての値は、直接の定義が可能である (直接定義される値の意味の説明は第 4.5 章を参照)。
- このクラスのすべてのスーパークラスはインターフェイスとして生成可能である。

たとえば、図 6 の例題を考えてみよう。クラス A は明らかにインターフェイスとして生成されるための要件を満たすが、なぜなら直接定義された値をもち、その関数と操作すべてが is subclass responsibility だからである。クラス B もまたインターフェイスとして生成可能だが、なぜなら抽象関数 1 つだけが与えられ、インターフェイスとして生成が可能なクラスから継承しているからである。しかしクラス C はインターフェイスとして生成はできない、なぜなら抽象でない関数の宣言を行っているからである。

どのクラスをインターフェイスとして生成すべきかの選択には、オプションダイアログボックス (第 3.1 章に記述されている) の インターフェイス選択 ボタンをクリックしよう。図 7 で示すように、新しいダイアログボックスが開く。

最初は、インターフェイス -A としては唯 1 つクラスが生成される可能性がある。これが選択されると (追加ボタンをクリックすることにより)、図 8 に示されるようにダイアログは更新される。


```
class A

values
  public v : nat = 1

operations
  public op : nat ==> nat
  op(n) == is subclass responsibility

functions
  public f : nat -> nat
  f(n) == is subclass responsibility

end A

class B is subclass of A

functions
  public g : nat -> nat
  g(n) == is subclass responsibility

end B

class C is subclass of A

functions
  public g : nat -> nat
  g(n) == n + 1

end C
```

図 6: インターフェイス例題

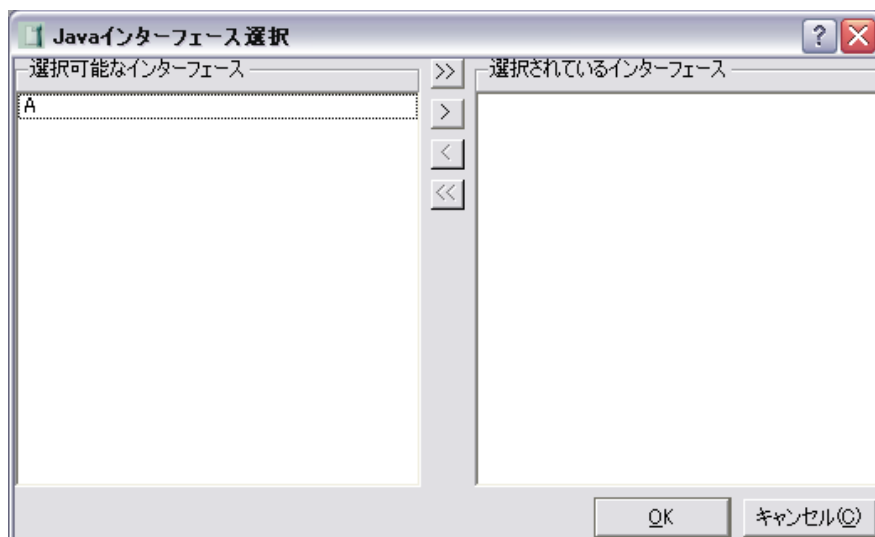


図 7: 初期インターフェース選択ダイアログ

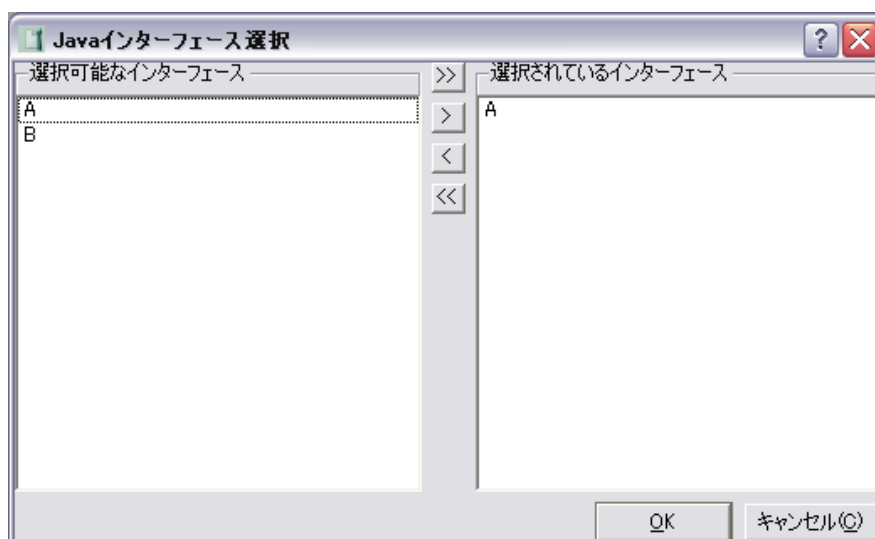


図 8: 更新されたインターフェース選択ダイアログ

ここで可能なインターフェイスの一覧に B が現れていることに注目しよう。もしこのスーパークラス - A - がインターフェイスであるならば、これはインターフェイスとして生成されることのみ可能となるためである。選択インターフェイスの一覧から A が今取り除かれたら、B は自動的に一覧から取り除かれるが、その時点でもはやインターフェイスの基準を満たさないからである。

インターフェイスとして生成されるべきクラスが選択されると、コード生成は通常通りに開始する。A に対して以下のコードが生成されるはずである：

```
public interface A {

    // ***** VDMTOOLS START Name=v KEEP=NO
    private static final Integer v = new Integer(1);
    // ***** VDMTOOLS END Name=v

    // ***** VDMTOOLS START Name=op KEEP=NO
    public abstract Integer op (final Integer n) throws CGException;
    // ***** VDMTOOLS END Name=op

    // ***** VDMTOOLS START Name=f KEEP=NO
    public abstract Integer f (final Integer n) throws CGException;
    // ***** VDMTOOLS END Name=f

}
```

Toolbox のコマンドライン版を用いてもまたインターフェイスの選択は可能であり、この場合 -U オプションを用いる：

```
vppde -j -U class{,class} specfiles
```

もしクラスが上記のインターフェイス基準を満足させないとする場合、以下のエラーメッセージが生成されるだろう：

```
Can not generate class class as an interface - ignored
```

3.6 制限

すべての VDM++ 仕様で Java コード生成が可能というわけではない。コンパイル可能で正しい Java コードに翻訳されるために、VDM++ 仕様はある要件を満たす必要がある。これらの制限は主に 2 つの理由が原因で引き起こされる:

- 用いられる翻訳アルゴリズムの制限: VDM++ および Java は 2 つの異なる言語であること。わずかな例として、VDM++ 構成要素の何らかの翻訳が正しくない Java コードを導く可能性がある。このことから引き起こされる制限は、第 3.6.1 章に並べている。並べた要件を満たさない VDM++ 仕様は、コンパイル可能でない誤った Java コードを導く可能性がある。Java へ翻訳できない VDM++ 特性にぶつかると、コードジェネレータは警告/エラーのメッセージを生成する。
- 翻訳の範囲制限: コードジェネレータはすべての構成要素をサポートしているわけではない。第 3.6.2 章では、コードジェネレータでサポートされない VDM++ 構成要素をまとめている。これらの構成要素はコンパイルできない Java コードではないが、これらに対して生成されたコードを実行すると実行時エラーとなる。コードジェネレータはサポートされていない構成要素に出会えば必ず、警告を与えることになる。

Java と VDM++ の意味論においては、プライベートメソッドが動的配置に伴ってどう扱われるかが異なることに注意しよう。以下の例題を考える:

<pre>class C operations public op1 : () ==> seq of char op1() == op2(); private op2 : () ==> seq of char op2() == return "C'op2" end C</pre>	<pre>class D is subclass of C operations public op3 : () ==> seq of char op3() == op1(); private op2 : () ==> seq of char op2() == return "D'op2" end D</pre>
---	--

Java において、式 `new D().op3()` は結果 `C'op2` をもたらす。VDM++ においては、同じ式が `"D'op2"` をもたらす。

3.6.1 言語の違いに起因する VDM++ 仕様の要件

VDM++ 仕様は、コンパイル可能で正しい Java コードを生成するために、以下の要件を満たさなければならない:

- “型情報不明”という型チェッカー警告は取り除かれるべきであるが、生成コードでエラーに導くことが可能であるからである。もし VDM 構成要素に型情報が失われていたら、コードジェネレータ は正しい Java 型の生成ができない。

節、インスタンス変数、型、値、関数、操作は同じ名称をもつことはない。さらには、名称の再宣言は避けるべきである。この意味は、たとえば以下の VDM++ 仕様は変数名称 `a` が再宣言されているため、コンパイル不可能な結果となるだろう、ということである:

```
f : int | (int * int) ==> bool
f(a) ==
  cases a:
    2 -> return true,
    mk_(a,b) -> return false,
    others -> let a = 1 in return true
end;
```

抽象の操作/関数は、それらを実装する操作 / 関数と同じ型をもつ必要がある。以下の例題を考えよう:

```
class A
operations
  m: nat ==> nat
  m(n) == is not yet specified;
end A

class B is subclass of A
operations
  m: nat ==> nat
  m(n) = return n+n;
end B
```

もし $B.m$ の型が $A.m$ の型とぴったりとは一致しない場合、 $A.m$ はまだ B では抽象であり、したがって B は抽象クラスということになる。

- 多重継承のため制限形式が用いられる可能性がある。しかし、これに含まれるクラスは第 3.5 章に記載された条件を満たしている必要がある。

case 文のすべての場合わけが return 文を含めるならば、case 文は others 岐をもたなくてはならない。そうでない場合には、生成された Java コードをコンパイルしながら Java コンパイラは “*Return required*” エラーを生成する。

- 不要なコードは避けるべきである。以下の例題を考えよう:

```
operations
  m : nat ==> nat
  m(n) ==
    (return n;
```

```
    a:= 4;  
  );
```

文 `a:= 4;` は決して実行されることはなく、生成された Java コードをコンパイルしながら “*Statement not reached*” エラーに導く。

- スーパークラスにおける操作呼出しが名称で修飾された場合、生成コードは誤ったものとなる可能性がある。以下の例題を見よう:

```
class A  
  
operations  
  
public SetVal : nat ==> ()  
SetVal(n) == ...;  
  
end A  
  
class B is subclass of A  
  
operations  
  
public SetVal : nat ==> ()  
SetVal(n) == ...  
  
end B  
  
class C is subclass of B  
  
operations  
  
public Test : () ==> ()  
Test() ==  
  ( self.SetVal(1);  
    self.B' SetVal(1);  
    self.A' SetVal(2)  
  )  
  
end C  
  
class D  
  
instance variables  
  b : B := new B()  
  
operations  
  
public Test: () ==> ()  
Test() ==
```

```
(b.SetVal(1);  
  b.B'SetVal(5);  
  b.A'SetVal(2)  
)  
  
end D
```

まずはクラス C から見よう: 文 `self.SetVal(1)` はクラス C における `SetVal` 操作を呼び出し、Java において `this.SetVal(1)` としてコード生成されることになる。文 `self.B'SetVal(1)` はクラス B における `SetVal` 操作を呼び出し、Java において `super.SetVal(1)` としてコード生成されることになる。Java では、クラス A の `SetVal()` メソッド呼び出しは不可能である。文 `self.A'SetVal(2)` は `super.SetVal(2)` としてコード生成されるだろう。クラス B に `SetVal` 操作がなかったなら、これは正しくなるはずである。しかし上記の場合、これは VDM++ 仕様に適合しない。2つの操作呼び出し `self.B'SetVal(1)` と `self.A'SetVal(2)` はコードジェネレータに警告 “*Quoted method call is code generated as a call to super*” を与える原因となる。ユーザーはしたがって正しいメソッドが呼び出されたかの確認ができる。

次にクラス D を見よう: 文 `b.SetVal(1)` はクラス B の `SetVal` 操作を呼び出し、Java で `b.SetVal(1)` としてコード生成されることになる。Java ではオーバーライドしているクラスの外部からオーバーライドされたメソッドを起動することはできない。したがってクラス A で `SetVal` メソッドを呼び出す方法はない。クラス D で引用された操作呼び出しは、したがってすべて `b.SetVal(1)` としてコード生成される。しかしながらコード生成は、ユーザーに知らせるために警告 “*Quoted method call is removed*” を発するはずである。

- Java における整数型およびダブル型に対する最大 (最小) 値は、VDM++ のそれぞれの値より小さい (大きい)。Java では有効でない値が、生成された Java コードの実行時にエラーに導く。

3.6.2 サポートされていない構成要素

コードジェネレータ のこの版では、以下の VDM++ 構成要素はサポートされていない:

- 式:
 - ラムダ式
 - 関数に対する合成、繰り返し、同等
 - 型判定式

- 高次関数。
- ローカル関数定義。
- 関数型インスタンス化式。しかし、以下の例題の中にあるように、コードジェネレータが適用式と組み合わせることで関数型インスタンス化式をサポートする:

```
Test:() -> set of int
Test() ==
  ElemToSet[int](-1);

ElemToSet[@elem]: @elem +> set of @elem
ElemToSet(e) ==
  {e}
```

- 文:

- 仕様記述文
- startlist 文

- 型束縛 ([2] 参照) ただし次の中:

- Let-be-st 式/文
- 列、集合、写像の包括式
- Iota 式 と修飾式

例題として、以下の式がコードジェネレータによりサポートされている:

```
let x in set numbers in x
```

一方で以下はサポートされていない (型束縛 $n: \text{nat}$ に起因する):

```
let x: nat in x
```

- パターン:

- 集合結合パターン。
- 列連結パターン。

コードジェネレータ はこれらの構成要素を含む仕様に対してコンパイル可能なコードを生成できるが、サポートされていない構成要素を含む分岐が実行された場合は、コードの実行は実行時エラーという結果となる。以下の関数定義を考えよう:


```
f: nat -> nat
f(x) ==
  if x <> 2 then
    x
  else
    iota x : nat & x ** 2 = 4
```

f に対して生成されたコードは、コンパイルされることになる。しかし f に相当するコンパイルされた Java コードは、 f に値 2 が適用されれば、iota 式の型束縛はサポートされず実行時エラーという結果に終わるだろう。

注意したいのは、コードジェネレータ はサポートされていない構成要素に出会った場合は必ず警告を与えるはずだということである。上記の関数 f に対するコード生成が、図 9 で示される *Error* ウィンドウに導く。

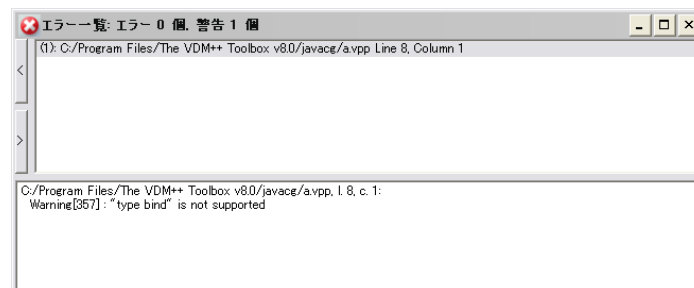


図 9: コードジェネレータにより生成された警告

4 VDM++ 仕様のコード生成

この章では、VDM++ 構成要素がコード生成される方法を詳細に記述していく。この記述は、コードジェネレータを専門的に利用しようとする場合には集中して学習すべき内容である。始めは VDM Java ライブラリへの導入を行うが、これが VDM++ 仕様コード生成の基礎を形成する。その後、上記で述べた VDM++ 構成要素 1 つ 1 つに対して生成されるコードを述べていく。

4.1 VDM Java ライブラリ

生成コードのデータ改編は、VDM Java ライブラリに基づき、パッケージ `jp.co.csk.vdm.toolbox.VDM` に実装されている。ここでは、このライブラリに短い導入を与えるだけとする。さらには、*javadoc* プログラムにより生成された HTML 文書で説明されている。このライブラリの全体理解のためには、説明書を読むべきである。*javadoc* プログラムを使用した HTML 文書生成方法についての記述は、付録 A を参照のこと。

VDM Java ライブラリ は以下の VDM++ データ型の固定実装を提供する:

- Product/Tuple Type
- 積型/組型
- Record Type
- レコード型

これらの型のそれぞれに対して、1つのクラスが、相当する VDM++ 型と同じパブリックメソッドを提供しながら実装されてくる。これらクラスは Java 言語により提供されたクラスの先頭に実装される。

VDM++ データ型で上記で並べていないもの (基本 VDM++ データ型、集合型、列型、写像型、Optional 型、ObjectReference 型) は、Java 言語自身の一部であるクラス / 構成要素 あるいは標準 Java 開発キット (JDK) の一部配布により表現される。

上記に挙げた VDM++ データ型の実装の提供に加えて、VDM Java ライブラリ はさらに 2 つのクラスを提供する:

- このクラスは補助メソッドを含むが、これは生成されたコードに用いられ、またユーザーが生成コードとインターフェイスをとるときに使用できる。これらの補助的メソッドのうちもっとも重要なものを以下に並べる:

- clone: VDM 値を (綿密に) 模倣する。しかしながら、VDM++ クラスと基本 VDM++ データ型は模倣できない。
- equals: 2 つの VDM 値を比較する。
- toString: VDM 値の ASCII 表現を含む列を返す。
- 実行時エラーが起きるときに呼び出される。VDM Java ライブラリで定義されている `VDMRunTimeException` を起こす。
- NotSupported: サポートされていない構成要素が実行されたときに呼び出される。NotSupportedConstructException を起こすが、これは VDM Java ライブラリで定義されているものだ。

注意: 常に UTIL クラスの clone、toString、equals のメソッドを用いて - VDM++ データ型に相当する Java クラスで定義されたメソッドは用いない。

- CGException クラスとそのサブクラス。

VDM Java ライブラリのエラー操作は、Java 例外取扱い機構に基づいている。生成された Java コードかあるいはライブラリメソッドの 1 つによりエラーが検出されたとき、適当な例外は起こされる。すべての実装済みエラーはクラス `CGException` のサブクラスであるが、さらに `java.lang.Exception` クラスの 1 サブクラスである。例外クラスの継承構造は、図 10 で示されている。

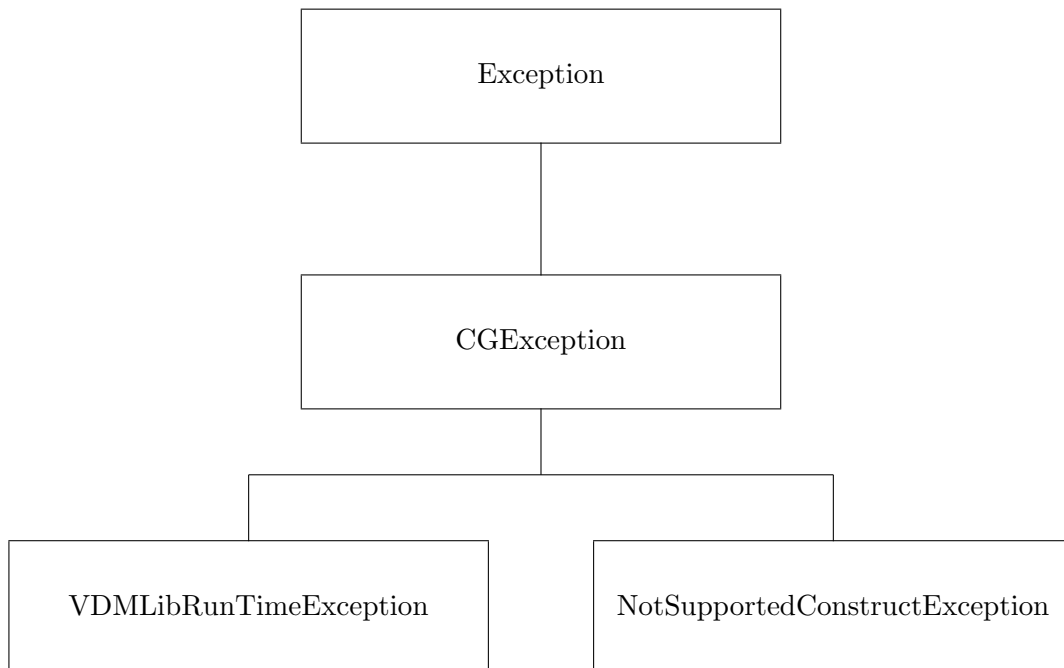


図 10: コードジェネレータの例外を取り扱う Java クラスの継承構造

例外の様々な種類は2つの型にグループ分けされる。

- `VDMRunTimeException` クラスのインスタンス: それらは Java 生成コードの中で起動される。VDM++ 仕様の実行中に起きる実行時エラーに相当する。
- `NotSupportedConstructException` クラスのインスタンス: コードジェネレータでサポートされていない構成要素が実行されたときに、それらが起こされる。

4.2 クラスを生成するコード

各 VDM++ クラスに対して、相当する Java クラスが生成される。各 VDM++ クラス要素に対して、Java クラスで相当する項目が VDM++ 要素と同じアクセス修飾子をもつことになる。

VDM++ 仕様のクラスに対して生成された Java クラスの構造を、より詳しく見てみよう。

生成された Java クラスは次を含む:

- 静的な比較器で、Java 開発キットのインターフェイス `java.util.Comparator` を実装しているもの。これはツリーベースのデータ構造で用いられ、同等の VDM 概念を実装する。
- VDM++ データ型を実装する Java コード。(第 4.4 章を参照)
- VDM++ 値を実装する Java コード。(第 4.5 章を参照)
- VDM++ インスタンス変数を実装する Java コード。(第 4.6 章を参照)
- 静的初期化 (値が初期化されなければならない場合)。
- コンストラクタ (クラスがインスタンス変数定義を含む場合)。
- VDM++ 関数を実装する Java メソッド。(第 4.7 章を参照)
- VDM++ 操作を実装する Java メソッド。(第 4.7 章を参照)
- 並列性 (同期、スレッドその他) に対するコードで、オプションが選択されている場合; 第 5 章を参照。

VDM++ クラス定義に対し生成される、生成 Java クラスの結果としてのスケルトンを考え、これを A としよう:

```
public class A {

// ***** VDMTOOLS START Name=vdmComp KEEP=NO
    static UTIL.VDMCompare vdmComp = new UTIL.VDMCompare();
// ***** VDMTOOLS END Name=vdmComp

    ...Implementation of VDM++ types...
    ...Implementation of VDM++ values...
    ...Implementation of VDM++ instance variables...
    ...VDM++ 型の実装...
    ...VDM++ 値の実装...
    ...VDM++ インスタンス変数の実装...

// ***** VDMTOOLS START Name=static KEEP=NO
    static {
        ...Initialization of VDM++ values...
        ...VDM++ 値の初期化...
    }
// ***** VDMTOOLS END Name=static

// ***** VDMTOOLS START Name=A KEEP=NO
    public A () {
        try { ...
            Initialization of VDM++ instance variables...
            VDM++ インスタンス変数の初期化...
            ...
        }
        catch (Throwable e) { ...
        }
    }
// ***** VDMTOOLS END Name=A

    ...Implementation of VDM++ functions...
    ...Implementation of VDM++ operations...
    ...VDM++ 関数の実装...
    ...VDM++ 操作の実装...

};
```

If a VDM++ class is abstract, the generated Java class will also be declared as such.
VDM++ クラスが抽象ならば、生成された Java クラスもまた同様に宣言されることになる。

4.3 生成された Java クラスの継承構造

生成された Java クラスの継承構造は、VDM++ クラスの継承構造にぴったり相当する。

VDM++ クラスの継承構造とソート例題に対して生成された Java クラスを、図 11 で見れる。

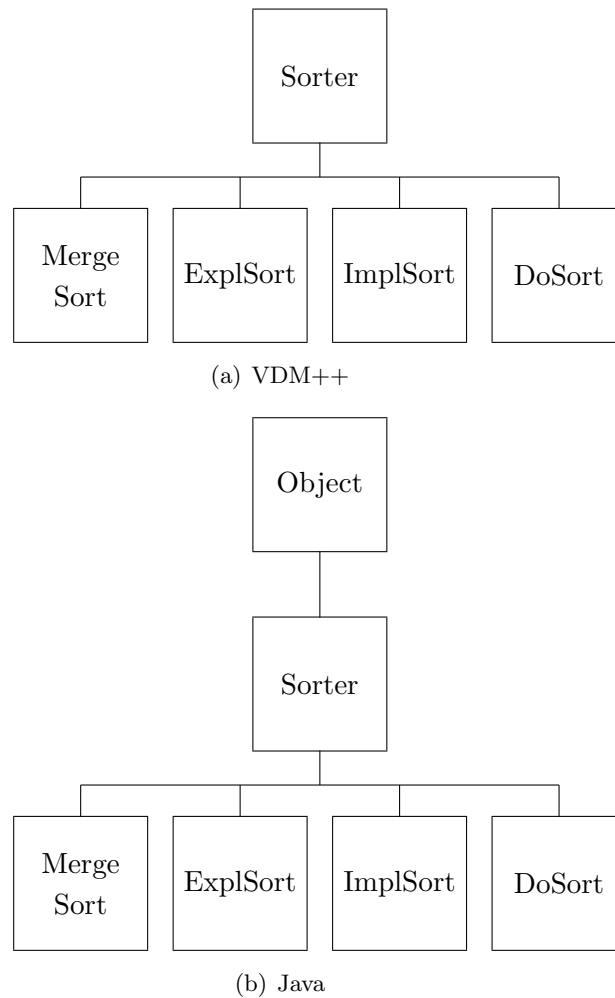


図 11: VDM++ クラスと生成された Java クラスの継承構造

VDM++ では、多重継承を用いて 1 つ以上のスーパークラスをもつことが許されている。Java は多重継承をサポートしない。代わりに、Java ではインターフェイス [4] との多重継承でその代わりとする。Java におけるクラスは、随意に 1 つのスーパークラスに拡張し、随意に 1 つ以上のインターフェイスを実装する。インターフェイスを実装するために、クラスは最初に implements 節にインターフェイスを宣言しなければならないし、そのためインターフェイスのすべての抽象メソッドに対する実装を提供しなければならない。これは実際は、VDM++ における多重継承と Java インターフェイスの間の本当の相違である。Java において、クラスは 1 つのスーパークラスからのみ実際の実装を継承することができる。インターフェイスからは追加の abstract メソッドの継承が可能だが、これらのメソッド自体の実装は提供しなければならない。

VDM++ レベルの多重継承を解決するために、ユーザーはインターフェイスとしてどのクラスがコード生成されるべきかを選択しなければならない (どのようにこれがなされるかの詳細は第 3.5 章を参照)。Java のインターフェイスモデルは VDM++ 多重継承モデルよりもさらに簡略なので、VDM++ におけるたくさんの多重継承のすべての場合ではないが、適切に解決され得る。このような環境では、完璧なコード生成が求められているのであれば VDM++ モデルは修正される必要がある。

VDM++ における多重継承のための Java コードを生成するためには、VDM++ においてスーパークラスが以下の条件を満たす必要がある:

- 1 つのスーパークラスだけが関数と操作を定義している可能性があり、このスーパークラスだけがインスタンス変数を提供している可能性がある。このクラスは Java においては単一のスーパークラスとしてコード生成されるであろう。
- インターフェイスとしての他のすべてのスーパークラスを生成することができなければならない (第 3.5 章を参照)。

もしサブクラスがすべての抽象関数や操作に対する実装を提供していないならば、抽象クラスとして生成されることになるということには注意しよう。

コード生成が可能な VDM++ 仕様の以下の例題を考えてみよう:

```
class E

instance variables
  protected i : nat

end E

class F
```

```
values
  public n : nat = 3

operations
  public getx : () ==> nat
  getx() == is subclass responsibility;

end F

class G is subclass of E, F

operations
  public getx : () ==> nat
  getx() == if true then return n else return i;

end G
```

並べられた VDM++ 仕様は必要な条件を満たす: クラス G は 2 つのクラス E と F のサブクラスである。クラス E は 1 つのインスタンス変数を定義する。したがって Java における単一のスーパークラスとしてコード生成されることになる。クラス F はインターフェイスとして生成される可能性があり、したがって第 3.5 章で与えられた範囲を満たす。G に対して生成された Java コードを以下におく。

```
public class G extends E implements F {

    static UTIL.VDMCompare vdmComp = new UTIL.VDMCompare();

    public Integer getx () throws CGException{
        if (new Boolean(true).booleanValue())
            return n;
        else
            return i;
    }
};
```

もし VDM++ 仕様が上記に並べた要求を満たさないならば、VDM++ to Java コードジェネレータ は誤ったコードを生成することになるだろう。

VDM++ レベルの多重継承が解決されない場合、コードジェネレータ は以下のエラーとなる:

```
Error : "Multiple inheritance in this form" is not supported and is not
        code generated
```


注意: VDM++ 式である 基本クラス、クラス、同一基本クラス、同一クラス といった構成要素は、多重継承の存在の下で、原型となる VDM++ 仕様と比べて生成された Java コードに対しての様々な意味あいをもつことになるだろう。

4.4 コード生成型

この章では、VDM++ 型が Java コードに写像変化する方法が述べてある。加えて、型に対する名前付け変換がまとめられている。

4.4.1 匿名 VDM++ 型から Java への写像

匿名型は、VDM++ 仕様において名称を与えられていない型である。これがコード生成される方法についてが以下の節で述べられている。

ブール型、数値型、文字型 Java 言語パッケージでは、基本データ型である倍精度型、整数型、ブール型、文字型、に対して以下の“ラッパー”クラスを提供している: 各々にダブル、整数、ブール、文字。これらのクラスは以下の VDM++ データ型を表わすために用いられる: `real`、`rat`、`int`、`nat`、`nat1`、`bool`、`char`。VDM `real` と `rat` 型は Java クラス `ダブル` に写像される。VDM `nat`、`nat1`、`int` 型は Java クラス `整数` に写像される。VDM `bool` 型は Java クラス `ブール` に写像される。VDM `char` 型は Java クラス `文字` に写像される。

ここで VDM++ と Java の間には、意味において相違があることに注意しよう。VDM++ において、`int`、`nat`、`nat1` 型は `real` および `rat` 型のサブタイプである。この意味は、もし値が整数値であるのなら、ある整数は `real` 型の変数に代入することが可能であるし、またある実数は `int` 型の変数に代入することが可能であるということである。

Java において、`Double` 型オブジェクトと `Integer` は互いを同じようにキャストを行うことはできない。したがって、2つの補助的 Java メソッド: `UTIL.NumberToDouble` と `UTIL.NumberToInteger` が提供されてきた。`Number` クラスは、`Double` と `Integer` クラス両方に対してのスーパークラスである。

引用型 コードジェネレータは、VDM++ 仕様で用いられたすべての引用に対して、クラス定義を生成する。全引用は `quotes` パッケージに収集されている。引用 `<HELLO>` は、パッケージ `quotes` 中の `HELLO.java` ファイルで `HELLO` クラス定義を導入することになる:

```
package quotes;

public class HELLO {

    static private int hc = 0;

    public HELLO () {
        if (hc == 0)
            hc = super.hashCode();
    }

    public int hashCode () {
        return hc;
    }

    public boolean equals (Object obj) {
        return obj instanceof HELLO;
    }

    public String toString () {
        return "<HELLO>";
    }
};
```

引用 <HELLO> はその後、以下のようにコード生成が可能である:

```
new quotes.HELLO()
```

hashCode メソッドが、引用定数の全インスタンスが同じハッシュコードをもつことを保証していることに、注意しよう。

トークン型 VDM++ 仕様がトークン型を含む場合、コードジェネレータは Token.java ファイルに Token クラスを生成する:

```
import jp.co.csk.vdm.toolbox.VDM.*;

public class Token {

    Object vdmValue;

    public Token (Object obj) {
        vdmValue = obj;
    }

    public Object GetValue () {
        return vdmValue;
    }
}
```

```
}  
public boolean equals (Object obj) {  
    if (!(obj instanceof Token))  
        return false;  
    else  
        return UTIL.equals(this.vdmValue, ((Token) obj).vdmValue);  
}  
public String toString () {  
    return "mk_token(" + UTIL.toString(vdmValue) + ")";  
}  
};
```

トークン値である `mk_token(<HELLO>)` は、たとえば以下のようにコード生成される:

```
new Token(new quotes.HELLO());
```

列型、集合型、写像型 VDM 列型 (seq of char 型以外)、集合型、写像型は、java.util パッケージの Vector クラス、TreeSet クラス、HashMap クラスに写像される。TreeSet に対しては、比較は、提供されている UTIL クラスで定義された比較器に基づいて行われている。これらのクラスは各々が、これもまた java.util パッケージで定義されているインターフェイス List、Set、Map を実装している。seq of char 型は、Java 言語クラス String に写像される。たとえば“(seq of char | seq of nat)”や“seq of (char | nat)”型は Vector として生成されることに注意しよう。

組型/積型 積型の値は組と呼ばれる。VDM 組型を表したクラスは Tuple と呼ばれ、VDM Java ライブラリで見つけることができる。注意したいのは、VDM++ 型、つまり `int * real` と `seq of nat * nat` は簡単に Tuple としてコード生成されている、ということである。

ユニオン型 匿名 VDM++ ユニオン型は、Java オブジェクト クラスで対応している。

選択型 VDM 選択型は、オブジェクト参照は Java では “null” となる可能性があることで示される。

オブジェクト参照型 第 4.2 章では、各 VDM++ クラスに対してどのように Java クラスが生成されているかが述べられている。VDM++ においては、オブジェクト参照型はクラス名によって示される。オブジェクト参照型のクラス名称は、仕様で定義されたク

ラスの名称でなければならない。さらに VDM++ においては、オブジェクト参照型の値は1つのオブジェクトへの参照と見なされ得る。

オブジェクト参照型は、Javaのクラス/インスタンス基本構造に相当する。JavaはVDM++の場合と同じに、オブジェクトを“by reference”で操作する。

関数型 VDM++ 関数型はコードジェネレータではサポートされていない。

4.4.2 VDM++ 型定義から Java への写像

VDM++ レコード型、それと完全に複数レコードだけからなるユニオン型に対して、VDM++ to Java コードジェネレータは型を表す内部クラスを生成する。他の種類の型定義に対して、Java 記述生成の必要はないというのは、他の型定義はすべて表面的定義だからである。つまり、現存の型に対して単純に新しい名称を示している。このような場合、VDM++ to Java コードジェネレータは現存の型を代わりに用いて、新しい名称が使用されることはない。これを説明するために、以下の例題を考えよう：

```
types
  A = nat
  B = seq of char
  C = A | B
```

新しい型は常に右側の型と等しくなる。このように、それらは右側の現存の型に対しての新しい名称である。したがって、生成されるコードはこれら右側の Java 実装を用いることになる型 A、B、C が VDM++ 仕様で用いられる場合、それらは Java クラスである Integer、Vector、Object にそれぞれ写像されることになるだろう。

しかしながら、複数のレコード型で構成されるレコード型とユニオン型は、深い型定義を表す。つまり、モデルに新しい型を導入する。したがって以下に述べる方法で、コード生成がなされる：

- 合成型/レコード型

VDM++ 仕様で定義されたすべてのレコード型はクラス定義に写像され、VDM Java ライブラリで見つかる Record インターフェイスを実装する。レコード項目は新しいクラスの変数となる。

たとえば、次の合成型³は

³Note: 合成型の定義に 構文 から成るものを用いてはならない。

```
public A::      real
              k : int
```

以下のように次のようにである。

```
public static class A implements Record {

    public Double f1;
    public Integer k;

    public A () {}
    public A (Double p1, Integer p2){
        f1 = p1;
        k = p2;
    }
    public Object clone () {
        return new A(f1,k);
    }
    public String toString () {
        "mk_G'A(" + UTIL.toString(f1) + "," + UTIL.toString(k) + ")";
    }
    public boolean equals (Object obj) {
        if (!(obj instanceof A))
            return false;
        else {
            A temp = (A) obj;
            return UTIL.equals(f1, temp.f1) && UTIL.equals(k, temp.k);
        }
    }
    public int hashCode () {
        return (f1 == null ? 0 : f1.hashCode()) +
            (k == null ? 0 : k.hashCode());
    }
};
```

レコード各項目に対して、パブリックインスタンス変数が生成されたクラス定義に追加されてきている。これら変数の名称は、相当する VDM レコード項目選択枝の名称と一致する。もし項目選択枝がなければ、たとえば上記例題中の `f1` というような、レコード中の要素の位置が代わりに用いられるだろう。もし `f1` がすでに項目選択枝として用いられているのであれば、その後文字 “f” が繰り返し、単一の項目選択枝が得られるまで付け加えられることになる。

- 合成型から構成されるユニオン型

ユニオン型は合成型から構成されるものだが、Java インターフェイスを用いてコード生成される。以下の VDM++ 型を見よう:

```
Item = MenuItem | RemoveItem;
```

```
MenuItem = Seperator | Action;  
Action:: text: String;  
Separator::;  
RemoveItem::;
```

生成された Java コードは次のようになる:

```
public static interface Item {  
};  
  
public static interface MenuItem extends Item {  
};  
  
private static class Action implements MenuItem , Record {  
    ...  
};  
  
private static class Separator implements MenuItem , Record {  
    ...  
};  
  
private static class RemoveItem implements Item , Record {  
    ...  
};
```

見ての通り、レコード型に対して生成されたクラスはユニオン型に対して生成されたインターフェイスを実装する。

4.4.3 不変数

不変数が仕様において VDM++ 型定義を制限することに用いられた場合、不変 VDM++ 関数がまた役に立つ。この不変関数が、関連する型定義と同じ範囲内に呼び込まれることは可能であるのか ([2] 参照)。生成している事前事後の関数 / 操作に対するオプションが選択されたとき、VDM++ to Java コードジェネレータ はこのような不変関数に相当する Java メソッド定義を生成する。1つの例題として、以下の VDM++ 型定義を考えよう:

```
public S = set of int  
inv s == s <> {}
```

VDM++ 関数 `inv_S` に相当するメソッド宣言を、以下に並べる。

```
public Boolean inv_S(final TreeSet s) throws CGException {  
    ...  
};
```

注意したいのは、VDM++ to Java コードジェネレータ は不変数の動的チェックをサポートしていないこと、しかし不変関数は明示的に呼び出されることが可能だということである。

4.5 コード生成値

VDM++ 値定義は、生成された Java クラスの静的最終変数へと翻訳される。静的キーワードは、特定変数がインスタンス変数であるよりはむしろクラス変数であるということを示している。さらには最終キーワードが、変数が1つの定数であることを示す。

以下の例題を考えてみよう:

```
class A  
values  
    public mk_(a,b) = mk_(3,6);  
    private c : char = 'a';  
    protected d      = a + 1;  
    e                = 2 + 1;  
end A
```

Java クラス A において生成されたクラス変数は次のようになる:

```
public class A {  
    public static final Integer a;  
    public static final Integer b;  
    private static final Character c = new Character('a');  
    protected static final Integer d;  
    private static final Integer e = new Integer(new Integer(2).intValue() +  
                                                new Integer(1).intValue());  
}
```

VDM++ 値が単純式で初期化されるならば、つまり追加として加えて他の VDM++ 値を含まず、相当する Java 変数が“直接に”初期化される。例題が示す通り、変数 c と e は直接に初期化される。他の変数はクラスの静的初期化子で初期化される。静的初期化

子はクラス変数のための初期化メソッドである。クラスが読み込まれると、システムにより自動的に起動される。インスタンス変数 a、b、d はこのように、生成された Java クラス A の静的初期化子の中で初期化される。クラス A のための静的初期化子を以下に並べる:

```
static {
    Integer atemp = null;
    Integer btemp = null;
    Integer dtemp = null;

    /** Initialization of class variables a & b */
    boolean succ_2 = true;
    {
        try{
            Tuple tmpVal_1 = new Tuple(2);
            tmpVal_1 = new Tuple(2);
            tmpVal_1.SetField(1, new Integer(3));
            tmpVal_1.SetField(2, new Integer(6));
            succ_2 = true;
            {
                Vector e_l_7 = new Vector();
                for (
                    int i_8 = 1; i_8 <= tmpVal_1.Length(); i_8++)
                    e_l_7.add(tmpVal_1.GetField(i_8));
                if (succ_2 = 2 == e_l_7.size()) {
                    atemp = UTIL.NumberToInt(e_l_7.get(0));
                    btemp = UTIL.NumberToInt(e_l_7.get(2 - 1));
                }
            }
            if (!succ_2)
                UTIL.RunTime("Pattern match did not succeed in value definition");
        }
        catch (Throwable e) {
            System.out.println(e.getMessage());
        }
    }
    a = atemp;
    b = btemp;

    /** Initialization of class variable d */
    {
        try{
            Integer tmpVal_11 = null;
            tmpVal_11 = new Integer(a.intValue() + new Integer(1).intValue());
            dtemp = tmpVal_11;
        }
        catch (Throwable e) {
            System.out.println(e.getMessage());
        }
    }
}
```



```
    }  
    d = dtemp;  
}
```

4.6 インスタンス変数のコード生成

インスタンス変数のコード生成は、大変直接的である。インスタンス変数は、相当する Java クラスの要素変数に翻訳される。

VDM++ における以下のインスタンス変数宣言を考えよう:

```
class A  
instance variables  
  public i : nat;  
  private k : int := 4;  
  protected message : seq of char := [];  
  inv len message <= 30;  
  j : real := 1;  
  ...  
end A
```

A.java ファイルの コードジェネレータ により生成された相当する Java コードは、次のようになるだろう:

```
public class A {  
  static UTIL.VDMCompare vdmComp = new UTIL.VDMCompare();  
  public Integer i = null;  
  private Integer k = null;  
  protected String message = null;  
  private Double j = null;  
  ...  
}
```

オブジェクトが生成されるときインスタンス変数は初期化される。Java においては、クラスの実体が生成されたときに実行されるコンストラクターメソッドで、インスタンス変数が初期化される。

このように、クラス A のためのコンストラクタメソッドの実装は、以下に示すようにインスタンス変数を初期化する:

```
public class A {  
    public A () {  
        try{  
            k = new Integer(4);  
            message = UTIL.ConvertToString(new String());  
            j = UTIL.NumberToReal(new Double(1));  
        }  
        catch (Throwable e) {  
            System.out.println(e.getMessage());  
        }  
    }  
    ...  
}
```

注意: インスタンス変数ブロックにおいて指定された不変数定義は、コードジェネレータによって無視される。

4.7 関数と操作のコード生成

VDM++ においては、関数および操作は明示的かまたは暗黙にかと両定義が可能である。VDM++ to Java コードジェネレータ は暗黙のと明示的など両方の、関数および操作定義に対する Java メソッドを生成する。VDM++ と Java 両方において、すべての関数と操作は仮想のものであり意味における違いはない。生成されたメソッドに与えられたアクセス修飾子は、相当する VDM++ 関数あるいは操作と同様となるだろう。VDM++ 仕様における関数や操作の名称が、相当する Java 実装における同じ名称として与えられることになるだろう。

すべての生成されたメソッドは `CGException` 例外を起こす。これは、生成された Java コードで起きた例外を取り扱うためである。

4.7.1 明示的な関数および操作定義

明示的な VDM++ 関数と操作定義をコード生成するための例題を見てみよう。

VDM++ クラス `DoSort` における操作定義 `Sort` は明示的であり、ファイル `DoSort.java` 中のクラス `DoSort` における以下の Java メソッドを導入する:

```
public Vector Sort (final Vector l) throws CGException{  
    ...  
}
```

4.7.2 予備的な関数操作定義

明示的な関数と操作本体の定義の本体は、“is subclass responsibility” 節と “is not yet specified” 節を用いて予備的なやり方で指定可能である。“is subclass responsibility” 節は、この本体の実装は任意のサブクラスでなされなければならないことを示す。“is subclass responsibility” 節を含む予備的な関数/操作の仕様が、Java の抽象メソッドに翻訳されている。抽象メソッドを含む Java クラスは抽象クラスである。生じた節はすべて、全抽象メソッドが実装されるまで抽象であり続けることになる。正しい Java コードを生成するために、VDM++ 仕様の抽象操作/関数がそれらを実装する操作 / 関数と同じ入出力パラメータをもたなければならない。抽象メソッドを実装しないサブクラスは、抽象クラスとして生成されることになるだろう。“is not yet specified” 節はこの本体の実装はユーザーによってなされなければならないことを示す。第 3.2 章では、どのようにこれがなされるかを記述した。

4.7.3 暗黙の関数と操作の定義

暗黙の関数と操作定義の実装は、ユーザーによってなされなければならない。さらなる情報は第 3.2 章を参照のこと。

4.7.4 事前事後条件

関数に対して事前事後条件が指定された場合、VDM++ to Java コードジェネレータによって相当の事前事後メソッドの生成が可能である。さらに、メソッドは操作の事前条件に対して生成可能である。しかし、操作仕様における事後条件は、VDM++ to Java コードジェネレータによって無視される。“Code generate pre and post functions/operations” オプションが、事前事後条件に相当する Java メソッド定義を生成するために選択されなければならない。

生成された事前事後メソッドは、相当する関数や操作のものと同一アクセス修飾子を取る。それらの名称は各々 post や pre を前に置き、戻り値の型は常に Boolean となる。“Check pre and post conditions” オプションが、事前事後条件をチェックするコードを生成するために用いられる可能性がある (操作事後条件を含まない)。

4.8 式と文のコード生成

VDM++ 式と文はコード生成されるが、生成コードが仕様により意図されたように動くようにである。

未定義式とエラー文は、VDM Java ライブラリ で見つけれ `VDMRunTimeException` を起こす関数 `UTIL.RunTime` の呼び出しに、翻訳される。

4.9 名称変換

VDM++ to Java コードジェネレータ により使用される名称付けの戦略としては、VDM++ 仕様で使用されていると同じ名称を保持するということだ。この戦略は VDM++ 仕様で用いられるすべての識別子に適用される。しかしながら、識別子の中に現れるアンダースコア (‘_’) やシングルクォーツ (‘’) はそれぞれアンダースコア - u (‘_u’) やアンダースコア - q (‘_q’) に、生成された Java コード中では交換されているはずである。さらには、予約語、予約メソッド名称、`java.lang` パッケージにおけるクラスの名称に ‘vdm_’ が接頭辞として付けられる。変数名称の再宣言の結果に起きる問題は、変数名称に `_number` で接尾辞を付けることで解決する。最後に、補助的/一時的な変数名称は `name_number` として名づけられる。

5 並列 VDM++ 仕様のコード生成

5.1 導入

VDM++ は、並列にスレッドを実行するシステムを指定するために、たくさんの機能を提供する。これらは、個々のスレッド機能の仕様、それにスレッド内で共有するオブジェクトに対する同期の仕様、を許している。

Java では、スレッドに対するサポートを Thread クラスを通して提供し、モニターを用いて共有オブジェクトの同期を許す。しかしながら、VDM++ はアクセス同期に対してさらに洗練された機構を提供し、したがって VDM++ 仕様から Java への翻訳は期待されるものよりさらにいくらか微小となる。

5.2 概論

前述の章で述べられたコード生成に加えて、並列 VDM++ to Java コードジェネレータは以下の構成要素の生成を許す:

- 手続きスレッド
- 定期スレッド
- *start* 文
- 許可述語
- 相互排除同期
- 履歴式

5.2.1 コード生成

VDM++ Toolbox のグラフィカルなユーザーインターフェイスから、“Generate code with concurrency constructs” オプションが選択されるべきである。コマンドラインからは `-e` フラグが、並列構成の生成指定に用いられるべきである:

```
vppde -j -e [other options] specfile(s)
```

5.3 翻訳手引き

並列 VDM++ 仕様のコード生成は、連続仕様のコード生成よりは直接的でないが、大雑把に言えば、同期のためのメカニズムが実装される必要があるからだ。特に翻訳手引きは、操作呼び出しが任意の同期制御順守を保証するという必要がある。これは翻訳手引きが、許可述語を評価するために必要とされる情報と、特に特定の操作のための履歴カウンタ、とを記録する手段を提供する必要があることを、含有している。

以下では、各クラスに対して行われるコア翻訳を説明する。もし手続きのまたは定期的なスレッドが指定されたならば、そこでこれに対する拡張の記述を行う。

これらの章の知識は 並列 VDM++ to Java コードジェネレータの使用が必要とされていない、したがってこれらの章は最初に読む場合に飛ばし読みして大丈夫かもしれない。手引きのさらに詳細な記述は [5] にある。

5.3.1 コア翻訳

この章では、とられる基本手引きの概観を与え、どのように同期が実装されるかを述べる。

翻訳されるすべての VDM++ クラスは、以下のことがその Java 翻訳に含まれる:

- evaluatePP メソッド
- 内的 Sentinel クラスと sentinel という名称のセンチネル要素変数

もちろん、これらはすべて現存のインスタンス変数への追加であり、前の章で記述されたやり方で翻訳された VDM++ クラスの関数である。操作は大体は前と同様に翻訳されるが、以下で述べられた少しの調整を含めている。ここでは短略して上記に並べた Java 構成要素の各々を記述する。evaluatePP メソッドは各翻訳されたクラスが実装する 並列 VDM Java ライブラリ の EvaluatePP インターフェイスによって指定される。その VDM++ クラスからの操作の 1 つの名称を表す整数を引数とし、その操作に対する許可述語の評価に相当する true や false を返す (その操作に対して許可述語が存在しないならば完全に true となる)。

Sentinel クラスは履歴カウンタ情報を記録するために使用される。VDM++ クラスの操作 Op は以下のスキーマを用いて翻訳されるであろう

```
sentinel.entering(((Op Sentinel) sentinel).Op);
try {
```

```

    Translation of body of op
  }
  finally { sentinel.leaving(((OpSentinel) sentinel).Op);}

```

sentinel.entering の呼び出しが #req 履歴カウンタを更新し、その後 evaluatePP メソッドを使用した操作のための許諾述語を評価する。許諾述語が true の場合、sentinel.entering の呼び出しは終了し本体が実行する; そうでない場合、履歴カウンタに関するアクティビティの通知を待ちながら、呼び出しが遮断している。たとえ他の許諾述語に使用されていたとしても、インスタンス変数に相当する要素変数値には変更通知がされないことに注意しよう。しかしながらこれは、インスタンス変数が変わってしまったとき許諾述語の再評価を要求しない VDM++ の意味論をちょうど映している。

同様に、操作の終わりで適切な履歴カウンタを更新する sentinel.leaving の呼出しがある。これは、本体が正常かまたは異常に終了するかどうか実行されたことを確認し、保証する finally 文内に、取り込まれている。

これらの追加事項と同様に、翻訳戦略に対しては修正もまた少しなされている:

- VDM++ インスタンス変数は Java *volatile* 要素変数に翻訳されるが、複数のスレッド間で共有されるかもしれないからである。
- クラスコンストラクタは sentinel を初期化するように拡張される。

5.3.2 手続きスレッド

翻訳されるはずの VDM++ クラスが 手続きスレッドを含めるのであれば、コア翻訳は 4 つの方法で拡張される:

- 翻訳されたクラスは、java.lang パッケージから Runnable インターフェイスを実装する。これは EvaluatePP インターフェイスの実装に追加するものである。
- VDMThread 要素変数が加えられる; VDMThread は 並列 VDM Java ライブラリの一部として定義されている。
- *Runnable* インターフェイスで指定されたように、run メソッドがクラス本体に実装されている。このメソッドの本体は VDM++ クラスの thread 節の翻訳に相当する。
- start メソッドが加えられる。スレッドが初期化され、その後スレッド独自の start メソッドを用いてスタートする。

5.3.3 定期スレッド

翻訳されるべき VDM++ クラスが定期スレッドを含むならば、コア翻訳は 3 つの方法に拡張される:

- `perThread` と呼ばれる `PeriodicThread` 要素変数が加えられる。並列 VDM Java ライブラリで定義された `PeriodicThread` が定義される。
- コンストラクタで `perThread` が初期化され、その `threadDef` メソッドが、VDM++ クラスでどの操作が定期的に行われるように指定されているべきかが定義されている。
- `start` メソッドが加えられる。その `invoke` メソッドを用いて `perThread` をスタートする。

5.4 例題

タイマーの例題とともに 並列 VDM++ to Java コードジェネレータ を説明する。タイマーは現時刻を記録するインスタンス変数を保守していて、2 つの操作を行う: 1 つは時間の設定で 1 つは時間の増加である。後の操作は、クラスの定期スレッドにより毎 1000 ミリ秒で実行される。(注 that the jitter、delay と offset パラメータは Java コード生成から無視される。)

```
class Timer
```

```
  instance variables
```

```
    hour: nat := 0;  
    min: nat := 0;  
    sec: nat := 0
```

```
  operations
```

```
    IncrementTime: () ==> ()  
    IncrementTime() == (  
      sec := sec + 1;  
      if sec = 60 then (sec := 0; min := min + 1);  
      if min = 60 then (min := 0; hour := hour + 1);  
      if hour = 24 then hour := 0;  
    );
```



```
-- This is for use by threads other than the periodic thread
public SetClock: nat * nat * nat ==> ()
SetClock(h,m,s) == (
    hour := h;
    min  := m;
    sec  := s
);

thread
    periodic (1000,jitter,delay,offset) (IncrementTime)

sync
    mutex(IncrementTime, SetClock);

end Timer
```

IncrementTime と *SetClock* は相互に排他的で、3つのインスタンス変数に書き込むからであることに注意しよう。これはクラスの *sync* 節で表明されている。

相当する Java コードが以下に置かれている。灰色にハイライトされた部分は、並列構成の翻訳に特有である。

```
public class Timer implements EvaluatePP {

    static UTIL.VDMCompare vdmComp = new UTIL.VDMCompare();
    private volatile Integer hour = null;
    private volatile Integer min = null;
    private volatile Integer sec = null;
    volatile Sentinel sentinel;
    PeriodicThread perThread;

    class TimerSentinel extends Sentinel {

        public final int IncrementTime = 0;
        public final int SetClock = 1;
        public final int nr_functions = 2;

        public TimerSentinel () throws CGException{}

        public TimerSentinel (EvaluatePP instance) throws CGException{
            init(nr_functions, instance);
        }
    };

    public Boolean evaluatePP (int fnr) throws CGException{
        Boolean temp;

        switch(fnr) {
        case 0: {
            temp = new Boolean(UTIL.equals(
                new Integer(sentinel.active[((TimerSentinel) sentinel).IncrementTime]
                    + sentinel.active[((TimerSentinel) sentinel).SetClock]),
                new Integer(0)));
            return temp;
        }
        case 1: {
            temp = new Boolean(UTIL.equals(
                new Integer(sentinel.active[((TimerSentinel) sentinel).IncrementTime]
                    + sentinel.active[((TimerSentinel) sentinel).SetClock]),
                new Integer(0)));
            return temp;
        }
        }
        return new Boolean(true);
    }

    public void setSentinel () {
        try{
            sentinel = new TimerSentinel(this);
        }
        catch (CGException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

```

public void start () throws CGException{
    perThread.invoke();
}

public Timer () {
    try{
        perThread = new PeriodicThread(new Integer(1000),perThread){
            public void threadDef () throws CGException{
                IncrementTime();
            }
        };
        setSentinel();
        hour = new Integer(0);
        min = new Integer(0);
        sec = new Integer(0);
    }
    catch (Throwable e) {
        System.out.println(e.getMessage());
    }
}

private void IncrementTime () throws CGException{
    sentinel.entering(((TimerSentinel) sentinel).IncrementTime);
    try{
        sec = UTIL.NumberToInt(UTIL.clone(new Integer(sec.intValue() +
                                                    new Integer(1).intValue())));
        if (new Boolean(sec.intValue() == new Integer(60).intValue()).booleanValue()) {
            sec = UTIL.NumberToInt(UTIL.clone(new Integer(0)));
            min = UTIL.NumberToInt(UTIL.clone(new Integer(min.intValue() +
                                                    new Integer(1).intValue())));
        }
        if (new Boolean(min.intValue() == new Integer(60).intValue()).booleanValue()) {
            min = UTIL.NumberToInt(UTIL.clone(new Integer(0)));
            hour = UTIL.NumberToInt(UTIL.clone(new Integer(hour.intValue() +
                                                    new Integer(1).intValue())));
        }
        if (new Boolean(hour.intValue() == new Integer(24).intValue()).booleanValue())
            hour = UTIL.NumberToInt(UTIL.clone(new Integer(0)));
    }
    finally {
        sentinel.leaving(((TimerSentinel) sentinel).IncrementTime);
    }
}

public void SetClock (final Integer h, final Integer m, final Integer s) throws CGException{
    sentinel.entering(((TimerSentinel) sentinel).SetClock);
    try{
        hour = UTIL.NumberToInt(UTIL.clone(h));
        min = UTIL.NumberToInt(UTIL.clone(m));
        sec = UTIL.NumberToInt(UTIL.clone(s));
    }
    finally {
        sentinel.leaving(((TimerSentinel) sentinel).SetClock);
    }
}
};

```

5.5 制限

並列 VDM++ to Java コードジェネレータ を使用している場合、以下のことを考慮すべきである:

- 一般的に、順次コードジェネレータにより生成された Java クラスは、並列システム内でのみ非同期方式で使用される可能性があり、同期の仕組みは付属であるというよりは翻訳クラスの全体をなすものである。もし同期が要求されているのであれば、並列オプションと共に コードジェネレータ を用いてコードの再生成がなされるべきである。
- 定期スレッドに対して、定期的に行われるべき操作の実行時間は、一定期間より少ない。そうした失敗は捕捉されない例外という結果になる可能性がある。
- `startlist` 文は現在は 並列 VDM++ to Java コードジェネレータ ではサポートされていない。

参考文献

- [1] CSK. *VDM++ Installation Guide*. CSK.
- [2] CSK. *The VDM++ Language*. CSK.
- [3] CSK. *VDM++ Toolbox User Manual*. CSK.
- [4] JAMES GOSLING, BILL JOY, G. S., AND BRACHA, G. *The Java Language Specification, Second Edition*. The Java Series. Addison Wesley, 2000.
- [5] OPPITZ, O. Concurrency extensions for the vdm++ to java code generator of the vdm++ toolbox. Master's thesis.

A コードジェネレータのインストール

VDM++ to Java コードジェネレータ は VDM++ Toolbox へのアドオン機能である。インストールについては [\[1\]](#) に記述されている。配布中に次の名称のディレクトリが見つかるだろう

javacg

このディレクトリは、 1 つの ファイルとその他 2 つのディレクトリを含める:

- VDM.jar: VDM Java ライブラリ (付録 [B](#) 参照)。
- libdoc: VDM Java ライブラリ の HTML 文書を含める (付録 [B](#) 参照)。
- example: 本書でコードジェネレータを説明するために使用した DoSort 例題を含める (付録 [C](#) 参照)。

B VDM Java ライブラリ

第 2.2 章で述べた通り、生成コードの実行を可能とするためには、CLASSPATH 環境変数が VDM Java ライブラリ、たとえば、VDM.jar ファイル、を含めることを、絶対保証しなければならない。このファイルは次のディレクトリで見つけることができる

```
javacg/
```

さらに、次のディレクトリ

```
javacg/libdoc
```

においては、このライブラリの javadoc により生成された HTML 文書を含める。

C DoSort 例題

本書でコードジェネレータを説明するのに用いられる DoSort 例題は、javacg/example という名称のディレクトリで見つけることができる。このディレクトリは 以下のファイルを含める:

- Sort.rtf
- sort.vpp
- DoSort.java
- MainSort.java

java ファイルは javacg/example ディレクトリで以下のコマンドを実行することによりコンパイルされることが可能である:

```
javac -classpath ../VDM.jar DoSort.java MainSort.java
```

Unix Bourne shell または互換のシェルを使用しているのであれば、主プログラムの実行は以下のコマンドの実行で可能である:

```
java -classpath ../VDM.jar MainSort
```

Windows-基本システムで動作している場合は、区切り文字として “:” でなく “;” を使用しなければならない:

```
java -classpath .;../VDM.jar MainSort
```

C.1 クラス DoSort(Sort.rtf) の VDM+++ 仕様

```
class DoSort
```

```
operations
```

```
public Sort: seq of int ==> seq of int
```

```
Sort(l) ==
```

```
return DoSorting(l)
```


functions

```
protected DoSorting: seq of int -> seq of int
DoSorting(l) ==
  if l = [] then
    []
  else
    let sorted = DoSorting (tl l) in
      InsertSorted (hd l, sorted);

private InsertSorted: int * seq of int -> seq of int
InsertSorted(i,l) ==
  cases true :
    (l = [])    -> [i],
    (i <= hd l) -> [i] ^ l,
    others      -> [hd l] ^ InsertSorted(i,tl l)
  end

end DoSort
```

C.2 クラス DoSort (DoSort.java) の Java コード

```
//
// THIS FILE IS AUTOMATICALLY GENERATED!!
//
// Generated at 2012-12-13 by the VDM++ to JAVA Code Generator
// (v8.3.2 - Wed 12-Dec-2012 16:31:57 +0900)
//
// Supported compilers: jdk 1.4/1.5/1.6
//

// ***** VDMTOOLS START Name=HeaderComment KEEP=NO
// ***** VDMTOOLS END Name=HeaderComment

// This file was generated from "../Sort.vpp".

// ***** VDMTOOLS START Name=package KEEP=NO
// ***** VDMTOOLS END Name=package

// ***** VDMTOOLS START Name=imports KEEP=NO
import java.util.List;
```

```
import java.util.ArrayList;
import jp.vdmtools.VDM.UTIL;
import jp.vdmtools.VDM.CGException;
// ***** VDMTOOLS END Name=imports

public class DoSort {

// ***** VDMTOOLS START Name=vdm_init_DoSort KEEP=NO
    private void vdm_init_DoSort () {}
// ***** VDMTOOLS END Name=vdm_init_DoSort

// ***** VDMTOOLS START Name=DoSort KEEP=NO
    public DoSort () throws CGException {
        vdm_init_DoSort();
    }
// ***** VDMTOOLS END Name=DoSort

// ***** VDMTOOLS START Name=Sort#1|List KEEP=NO
    public List Sort (final List l) throws CGException {
        return DoSorting(l);
    }
// ***** VDMTOOLS END Name=Sort#1|List

// ***** VDMTOOLS START Name=DoSorting#1|List KEEP=NO
    protected List DoSorting (final List l) throws CGException {
        List varRes_2 = null;
        if (UTIL.equals(l, new ArrayList()))
            varRes_2 = new ArrayList();
        else {
            final List sorted = DoSorting(new ArrayList(l.subList(1, l.size())));
            varRes_2 = InsertSorted(UTIL.NumberToInt(l.get(0)), sorted);
        }
        return varRes_2;
    }
// ***** VDMTOOLS END Name=DoSorting#1|List

// ***** VDMTOOLS START Name=InsertSorted#2|Number|List KEEP=NO
    private List InsertSorted (final Number i, final List l) throws CGException {
        List varRes_3 = null;
        boolean succ_4 = false;
        {
            /* (l = []) -> [i] */
            /* (l = []) */
            succ_4 = (UTIL.equals(Boolean.TRUE, Boolean.valueOf(UTIL.equals(l, new ArrayList()))));
```

```

        if (succ_4) {
            /* [i] */
            List tmpSeq_10 = new ArrayList();
            tmpSeq_10.add(i);
            varRes_3 = tmpSeq_10;
        }
    }
    if (!succ_4) {
        /* (i <= hd l) -> [i] ^ 1 */
        /* (i <= hd l) */
        succ_4 = (UTIL.equals(Boolean.TRUE, Boolean.valueOf(i.intValue() <= UTIL.NumberToInt(l.get
        if (succ_4) {
            /* [i] ^ 1 */
            List tmpSeq_17 = new ArrayList();
            tmpSeq_17.add(i);
            varRes_3 = new ArrayList(tmpSeq_17);
            varRes_3.addAll(l);
        }
    }
    /* others */
    if (!succ_4) {
        List tmpSeq_21 = new ArrayList();
        tmpSeq_21.add(UTIL.NumberToInt(l.get(0)));
        varRes_3 = new ArrayList(tmpSeq_21);
        varRes_3.addAll(InsertSorted(i, new ArrayList(l.subList(1, l.size()))));
    }
    return varRes_3;
}
// ***** VDMTOOLS END Name=InsertSorted#2|Number|List

}
;

```

C.3 手書きの Java 主プログラム (MainSort.java)

```

import jp.vdmtools.VDM.*;
import java.util.List;
import java.util.ArrayList;

public class MainSort {

    @SuppressWarnings("unchecked")
    public static void main(String[] args){
        try{
            List arr = new ArrayList();

```

```
        arr.add(new Integer(23));
        arr.add(new Integer(1));
        arr.add(new Integer(42));
        arr.add(new Integer(31));
        DoSort dos = new DoSort();
        System.out.println("Evaluating Sort("+UTIL.toString(arr)+"):");
        List res = dos.Sort(arr);
        System.out.println(UTIL.toString(res));
    }
    catch (CGException e){
        System.out.println(e.getMessage());
    }
}
}
```