

VDMTools

Java コード生成マニュアル
(VDM++)

How to contact SCSK:

http://www.vdmtools.jp/	VDM information web site(in Japanese)
http://www.vdmtools.jp/en/	VDM information web site(in English)
http://www.scsk.jp/	SCSK Corporation web site(in Japanese)
http://www.scsk.jp/index_en.html	SCSK Corporation web site(in English)
vdm.sp@scsk.jp	Mail

Java コード生成マニユアル (VDM++) 2.0

— Revised for VDMTools v9.0.2

© COPYRIGHT 2013 by SCSK CORPORATION

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	まえがき	1
2	コードジェネレータ – 入門編	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ース	6
2.3	Java コードのコンパイルと実行	8
3	コードジェネレータ – 詳細編	10
3.1	Java コードジェネレータのオプション	10
3.2	陰関数／操作および仮関数／操作の実装	13
3.3	抽象クラスの生成	16
3.4	生成された Java コード部分の置き換え	18
3.4.1	実体	22
3.4.2	KEEP タグのための規則	22
3.5	interface の生成	23
3.6	制限	27
3.6.1	言語の違いに起因する VDM++ 仕様の要件	28
3.6.2	サポートされていない構成要素	30
4	VDM++ 仕様のコード生成	33
4.1	VDM Java ライブラリ	33
4.2	クラスを生成するコード	35
4.3	生成された Java クラスの継承構造	37
4.4	コード生成型	40
4.4.1	匿名 VDM++ 型から Java への写像	41
4.4.2	VDM++ 型定義から Java への写像	44
4.4.3	不変条件	47
4.5	値のコード生成	47
4.6	インスタンス変数のコード生成	49
4.7	関数と操作のコード生成	51
4.7.1	陽仕様な関数および操作定義	52
4.7.2	予備的な関数操作定義	52
4.7.3	陽仕様の関数と操作の定義	52
4.7.4	事前・事後条件	52
4.8	式と文のコード生成	53
4.9	名称変換	53

5 並列 VDM++ 仕様のコード生成	54
5.1 導入	54
5.2 概論	54
5.2.1 コード生成	54
5.3 変換手引き	55
5.3.1 コア翻訳	55
5.3.2 手続きスレッド	56
5.3.3 定期スレッド	57
5.4 例題	57
5.5 制限	61
A コードジェネレータのインストール	63
B VDM Java ライブラリ	64
C DoSort 例題	65
C.1 クラス DoSort(Sort.rtf) の VDM+++ 仕様	65
C.2 クラス DoSort (DoSort.java) の Java コード	66
C.3 手書きの Java メインプログラム (MainSort.java)	68

1 まえがき

Java コードジェネレータ は、VDM++ 仕様から自動的に Java コードを生成する機能を提供する。コードジェネレータ は、VDM++ 仕様に基づいた Java アプリケーションの実装を手早く行う手段を提供する。

本マニュアルは、*User Manual for the VDM++ Toolbox* [SCSc] を拡張するものであり、Java コードジェネレータ を解説する。

本マニュアルの構成を以下に示す:

第 2 章で 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るとともに、生成された Java コードを使用する方法の指針を与える。さらに、生成された Java コードのコンパイルと実行の方法を解説する。

第 3 章では、4 つの項目についてより詳しく説明する。まず、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示す。次に、陰関数／操作、仮関数／操作の取り扱いを解説し、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できることを解説する。最後に、変換された Java コードが、コンパイルでき、正しく動作するために、VDM++ 仕様が満たさなければならない要件を列挙する。

第 4 章では、生成された Java コードの構造を詳細に示す。次に、VDM++ と Java のデータ型の関係を説明し、Java コードジェネレータにより開発を行うとき、命名規則を含む、設計上考慮されるべきことがらについて解説する。業務等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、並列動作を記述した仕様に対し、どのようなコード生成が行われるかについて解説する。このような仕様に対しては、多重スレッドの Java コードが生成される。コード生成の方法と、変換実現方法の概観を示す。

2 コードジェネレータ – 入門編

コードジェネレータ の使用を始めるには、1 つ以上のファイルの VDM++ 仕様が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、配布パッケージの `javacg/javasort` ディレクトリ内の `Sort.vpp` あるいは `Sort.rtf` ファイルに記載されている。第 2.1 章では、VDM++ Toolbox を用いて VDM++ の DoSort クラスから 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ース画面で、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、`Sort.vpp` ファイルを含む新しいプロジェクトを作成しなければならない。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、構文チェックが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。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、型チェックが Toolbox により自動的に行われる。

User Manual for the VDM++ Toolbox [SCSb] 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を行う。結果を図 1 に示す。

¹[SCSb] 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。

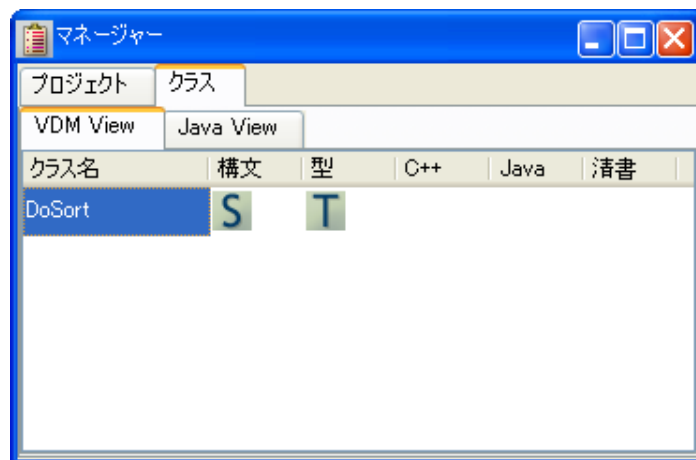


図 1: 構文および型チェック後の管理画面


ここで、 (Java コード生成) 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。この操作は複数ファイル／クラスを選択が可能で、その場合はそれらすべてが Java に変換される。

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。DoSort.java と呼ばれる Java ファイルがコード生成されていることが示されている。そのファイルには DoSort の Java クラス定義が含まれている。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 の起動ディレクトリにファイルは書き込まれる。

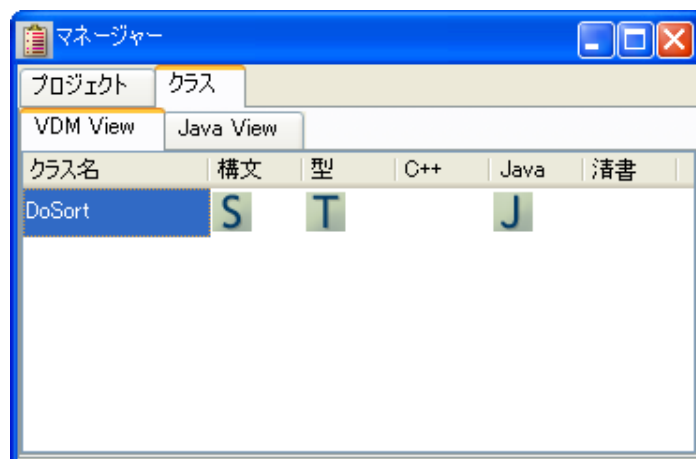


図 2: DoSort クラスをコード生成

図 3 と図 4 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、ファイル DoSort.java の全体が記載されている。

```
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
```

図 3: VDM++ による DoSort クラス


```
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 {
        ...
    }
// ***** VDMTOOLS END Name=DoSort

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

// ***** VDMTOOLS START Name=DoSorting#1|List KEEP=NO
    protected List DoSorting (final List l) throws CGException {
        ...
    }
// ***** 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 {
        ...
    }
// ***** VDMTOOLS END Name=InsertSorted#2|Number|List

}
```

図 4: 生成された Java DoSort クラス

VDM++ Toolbox のコマンドライン版からも、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` のコード生成を行う場合、以下のコマンドを実行する:

```
vppde -j Sort.vpp
```

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 POS 型の適格性型の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、仕様は複数の Java ファイルに変換される。例に挙げている仕様に対しては、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、仕様が複数のクラスを含んでいる場合、コードジェネレータ のコマンドライン版を用いると、すべてのクラスが同時にコード生成されることになる (-K オプションでクラス指定が可能)。

2.2 生成コードとのインターフェース

ここまでで、VDM++ 仕様から Java コードを生成ができている。次に、1 個のアプリケーションとしてコンパイルおよび実行するための、生成された DoSort クラスとのインターフェースの記述方法を示す。

最初に VDM++ を使ったメインプログラムを示す。

```
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:   )
```

上記の VDM++ 仕様と同機能の Java のメインプログラムを実装する。メインプログラムを含む Java ファイルは、VDM Java ライブラリパッケージ `jp.vdmtools.VDM` の全クラスをインポートすることから始める。

```
import jp.vdmtools.VDM.*;
```

これにより、完全に修飾された名称を記述することを省略することができる。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。ここでは順を追って、上記の VDM 仕様を Java に変換する。

行 02 では整数の列を指定する。Java に変換すると、以下のコードになる。

```
List arr = new ArrayList();
arr.add(23);
arr.add(1);
arr.add(42);
arr.add(31);
```

`ArrayList` クラスと `List` インターフェースは `java.util` パッケージに含まれている。`DoSort` クラスの `Sort` メソッドでは、`List` 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。

```
List res = new ArrayList();
```

`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 に変換すると、以下のコードとなる。

```
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 値の文字表現を含む文字列を得る。このメソッドは、ここでは実行中に結果をログメッセージとして標準出力に出力するために使用されている。

上記の Java コードは、生成された Java コード内のメソッドで発生する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` ブロックには `catch` 節が続き、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、これらもまた VDM Java ライブラリの一部である。次のような `catch` 文が想定される:

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

前述のメインプログラムは、`MainSort.java` という名のファイルに実装され、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

メイン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。

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 **1.5** 版と相性がよい。

メイン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システムアイコンから、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。

メインプログラム `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 – 詳細編

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)
- 仕様が陰関数／操作または仮関数／操作を含む場合には、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 コードに変換されるために満たすべき要件は何か? (第 3.6 章)

3.1 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 5 に示すように、プロジェクトオプションメニューから *Java* コード生成 タブを選択しよう。

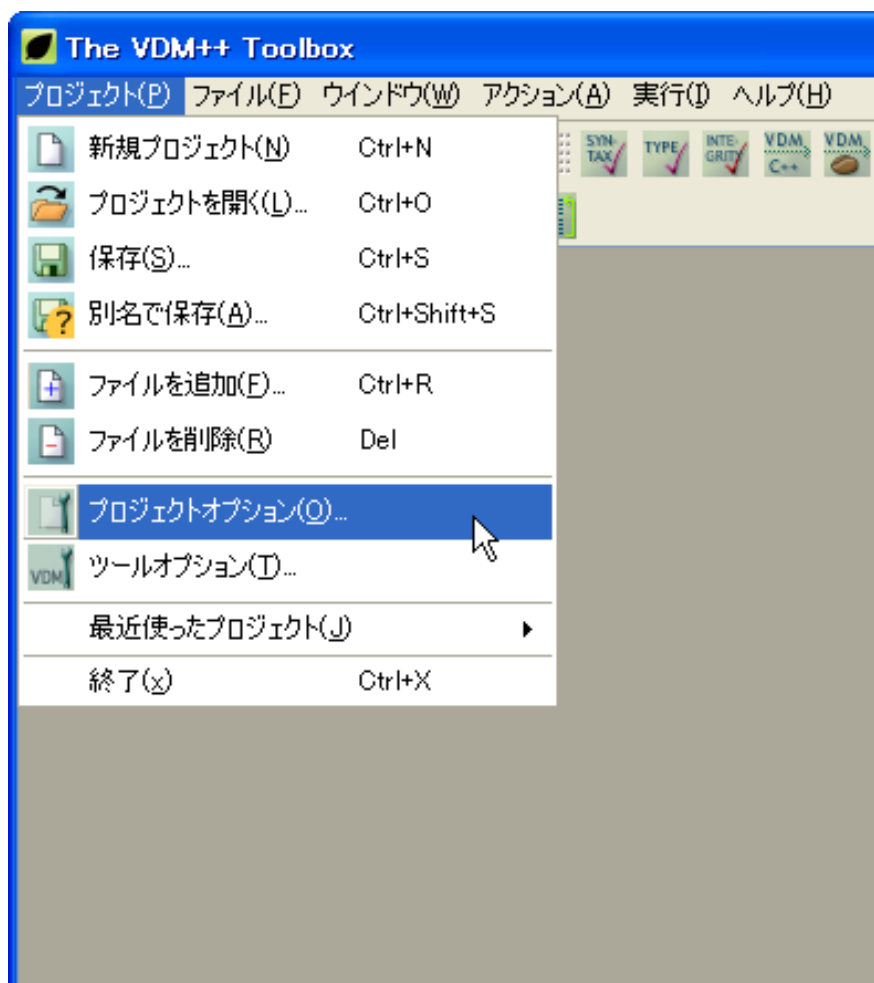


図 5: Java コードジェネレータのオプションの選択

図 6 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の説明は以下に記述している。

これらオプションのいくつかは、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、対応するフラグを提示している。デフォルトの動作もまた、デフォルト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。

型以外は骨組みのみ生成する (-s) スケルトンクラスを生成するために、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

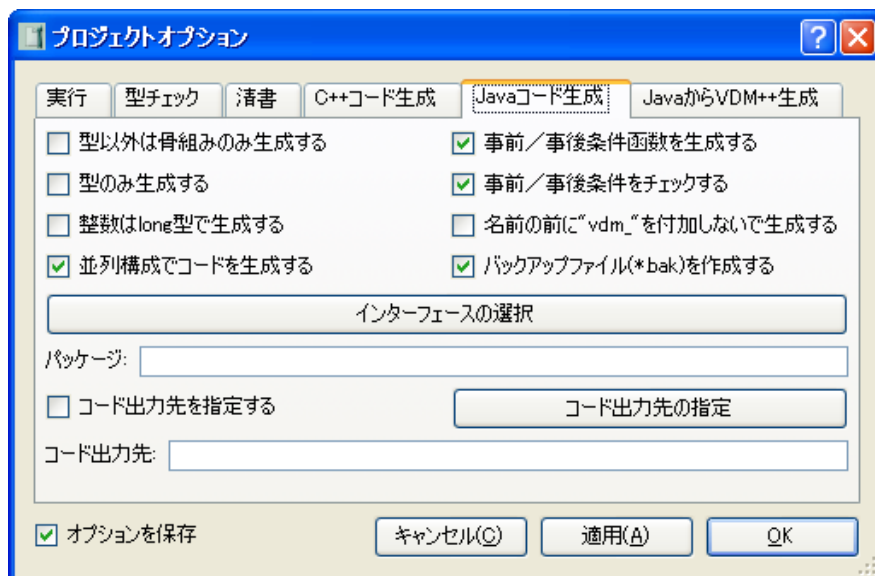


図 6: Java コード生成のためのオプション

が、空の関数や操作の定義は含まないクラスである。

デフォルト: **off**

型のみ生成する (-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)。

デフォルト: **off**.

整数は long 型でコードを生成する (-L) 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。

デフォルト: **off**

並列構成でコードを生成する (-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ために用いられる。この詳細は第 5 章を参照のこと。

デフォルト: **on**

事前／事後条件関数を生成する (-k) 事前事後条件に対する Java メソッドやそれらの不変条件をコード生成するために、このオプションを指定する。

デフォルト: **on**

事前／事後条件をチェックする (-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。

デフォルト: **off**

名前の前に “vdm_” を付加しないで生成する このオプションは、VDM++のクラス名が JDK のクラス名と同じ場合に、“vdm_” を付加せずに生成するために用いられる。

デフォルト: **off**

バックアップファイル (*.bak) を作成する このオプションは、以前に生成した java ファイルを保存するために用いられる。

デフォルト: **on**

インターフェースの選択 (-U classname[,classname]) Java インターフェー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。

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。

コード出力先の指定 生成される Java ファイルは、プロジェクト名が指定されている場合、プロジェクトファイルと同じディレクトリに生成される。他の場所に生成したい場合にこのオプションを使用する。

コマンドラインから VDM++ Toolbox をスタートさせる場合、以下のコマンドを用いる。

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

3.2 陰関数／操作および仮関数／操作の実装

陰関数／操作および仮関数／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。仮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 A は型 `external_A` の `protected` なインスタンス変数 `child` を保有している。これは、陰関数／操作または仮 (“`is not yet specified`” で指定された) 関数／操作が記述されてい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んでいるとしても、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つ以上の陰関数／操作または仮関数／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。

ユーザーが外部クラスファイルを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、仮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` ファイルを、以下に示す。

²`impl` は “Java で実装されるべき” ことを表す。

```
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` クラスを実装する最も簡単な方法は、テンプレートクラスを修正することだ。つまり、ユーザーは次のコード部分を、生成コードをユーザ独自のコードで置き換えるというありふれた方法で、置き換えを行う必要がある。

```
UTIL.RunTime("Preliminary Operation op has been called");  
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、仮操作定義の実装はクラス `A` の `public` な属性や操作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、仮関数に対応する 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、間接的に内部状態への作用が可能となる。

陰関数／操作は、“`is not yet specified`” 節を含む仮関数／操作の仕様と同じ方法で取り扱われる。

注意したいのは、外部クラスが陰関数／操作と仮関数／操作の両方を含むことが可能であることだ。生成されたテンプレートにおいて、生成される実行時エラーメッセージの違いで区別可能である。

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

これは `op` という名の仮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` という名の陰関数定義に対するものある。

3.3 抽象クラスの生成

VDM++ クラスは、仮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り、継承している抽象関数・操作の実装を提供しない場合に、抽象クラスとなる。

一方で、Java においても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、抽象クラスと識別される VDM++ クラスは、抽象 Java クラスとして生成されることになる。例えば、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 は仮関数・操作を含むので、抽象クラスである。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。したがって通常のクラスである。

```
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がある。

これは、*KEEP* タグの使用を通して実現される。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。

例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 KEEP タグに相当する。KEEP 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、**YES** ならば保存される。ファイル生成時のデフォルトは **NO** である。

現在の日付を返すように、この関数を修正したいと仮定する。これは、Java 開発キット (JDK) 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, c.get(Calendar.YEAR));
    return result;
}
// ***** VDMTOOLS END Name=today
```

最初に KEEP タグが YES に変化していることに注意しよう。これは変更が保存される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

KEEP 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メンバー変数
- トップレベルのクラスメソッド（コンストラクタを含む）
- インナークラス
- 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（ファイルの先頭領域に、例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）

KEEP タグは、`interface` 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 `interface` 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 KEEP タグのための規則

KEEP 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（例えばある関数がインナークラスに追加された場合、インナークラス全体が **YES** とタグづけされなければならない）

- KEEP タグの構文では、case 文と空白は細心の注意を払うべきであり、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、オプション設定で、元のファイルをバックアップするよう設定しておけば、必ずしもこれが致命的となることはない。

3.5 interface の生成

コードジェネレータは Java の interface [JGB00] の生成を可能としている。VDM++ クラスは以下の条件に適応する場合は、interface 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** である
- 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、public 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 interface として生成可能である

例えば、図 7 の例題を考えてみよう。クラス A は明らかに interface として生成されるための要件を満たす。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 interface として生成可能だ。なぜなら抽象関数 1 つだけが与えられ、interface として生成が可能なクラスを継承しているからである。しかしクラス C は interface 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 interface として生成すべきかの選択には、プロジェクトオプションダイアログ (第 3.1 章に記述されている) の インターフェースの選択 ボタンをクリックしよう。図 8 で示すように、新しいダイアログボックスが開く。

最初は、interface -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 9 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 7: インターフェース例題

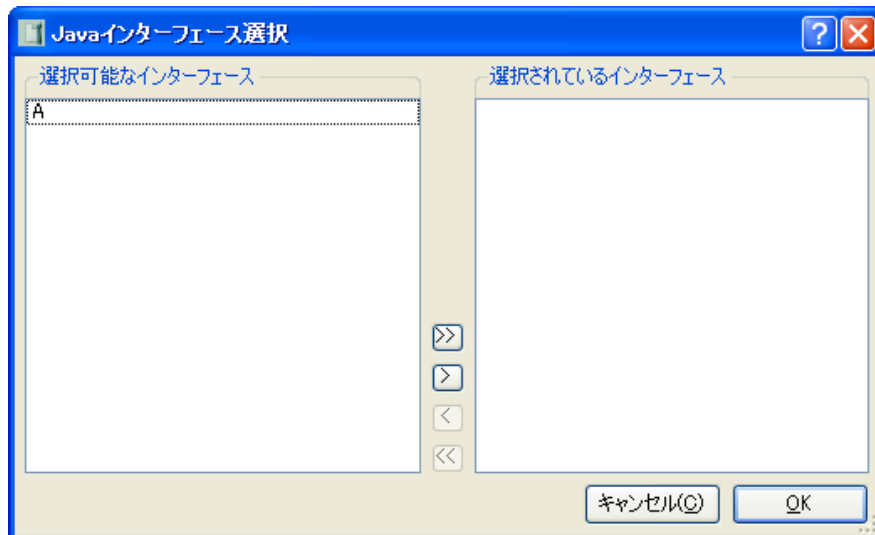


図 8: 最初のインターフェース選択ダイアログ

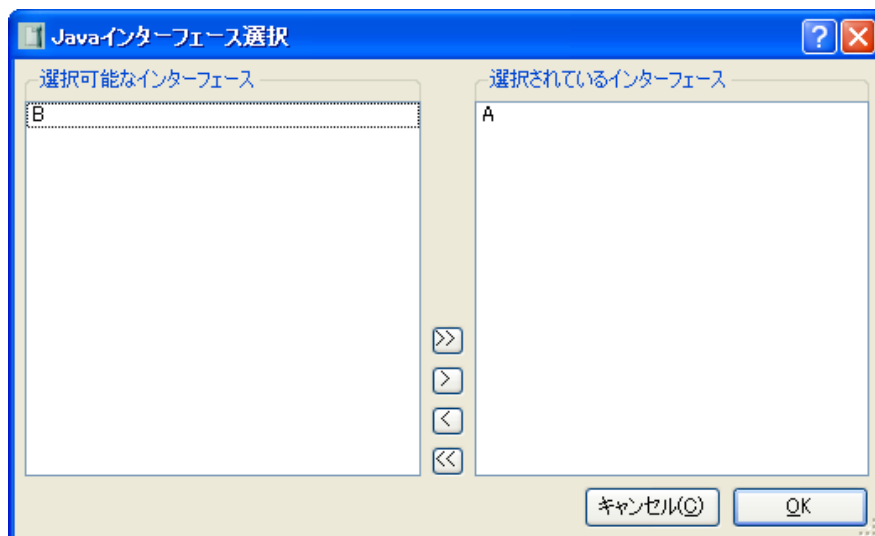


図 9: 更新されたインターフェース選択ダイアログ

ここで可能な interface 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 interface であるならば、これは interface として生成されることのみ可能となるためである。選択 interface の一覧から A が取り除かれたら、B は自動的に一覧から取り除かれる。その時点でもはや interface の基準を満たさないからである。

インターフェー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
    public static final Number v = new Integer(1);
    // ***** VDMTOOLS END Name=v

    // ***** VDMTOOLS START Name=op#1|Number KEEP=NO
    abstract public Number op (final Number n) throws CGException ;
    // ***** VDMTOOLS END Name=op#1|Number

    // ***** VDMTOOLS START Name=f#1|Number KEEP=NO
    abstract public Number f (final Number n) throws CGException ;
    // ***** VDMTOOLS END Name=f#1|Number

}
```

Toolbox のコマンドライン版を用いてもまた interface 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ース基準を満たさせないとする場合、以下のエラーメッセージが表示されるだろう。

```
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 コードに変換されるために、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:

用いられる変換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。数少ない例として、VDM++ 構成要素の何らかの変換が正しくない 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。これらの要件を満たさない VDM++ 仕様は、コンパイル可能でない誤った Java コードに変換される可能性がある。Java へ変換できない VDM++ の機能にぶつかると、コードジェネレータは警告/エラーメッセージを表示する。

変換の範囲制限 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、動的ディスパッチに関し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。

```
class C
operations
public op1 : () ==> seq of char
op1() == op2();

private op2 : () ==> seq of char
op2() == return "C'op2"
end C
```

```
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
```

Java において、式 `new D().op3()` は `C'op2` という結果となる。一方、VDM++ においては、同じ式が `"D'op2"` となる。

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。これは、生成コードにエラーが含まれる可能性があるからである。もし VDM 構成要素の型情報がない場合、コードジェネレータ は正しい Java 型の生成ができない。
- クラス、インスタンス変数、型、値、関数、操作に同じ名称を付けてはならない。
- 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 の型と正確に一致しない場合、A ‘m は B においても抽象操作であり、したがって B は抽象クラスということになる。

- 多重継承には一定の制限がある。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 文のすべての場合分けが return 文が含まれる場合、case 文は **others** をもたなくてはならない。そうでない場合には、Java コンパイラが生成された Java コードをコンパイルすると、“*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すると、“*Statement not reached*” エラーが生じる。

- スーパークラスにおける操作呼出しを、クラス名で修飾した場合、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)` に対して、コードジェネレータは、警告 “*Quoted method call はスーパークラスの呼び出しとしてコード生成されます*” を生成する。これにより、正しいメソッドが呼び出されるかどうかの確認を行わねばならないことがわかる。

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、すべて `b.SetVal(1)` としてコード生成される。この場合、コード生成は、警告 “*Quoted method call が削除されています*” 生成し、ユーザに通知する。

- Java における整数型および倍精度浮動小数点型の最大 (最小) 値が、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あると、生成された Java コードの実行時にエラーが発生する。

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
 - － 型変数を含む型判定式
 - － 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
- 型束縛 ([SCSB] 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 `x : nat` に起因する)。

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

- パターン (簡単なものはコード生成可能)
 - 集合合併パターン。
 - 列連結パターン。
 - 写像併合パターン。

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では、図 10 で示される、*Error* ウィンドウが表示される。

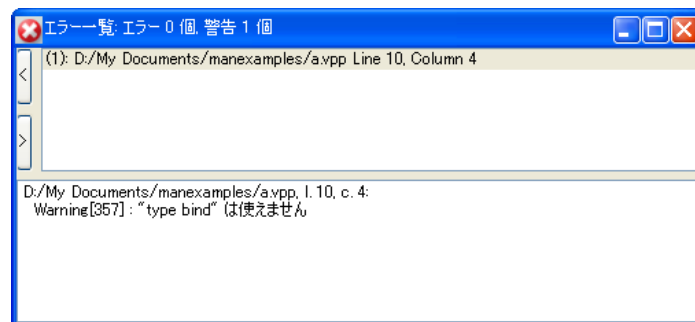


図 10: コードジェネレータにより生成された警告

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.vdmtools.VDM` に実装されている。ここでは、このライブラリについて簡単に紹介する。詳細は、*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。このライブラリについて詳しく理解したければ、その文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++ データ型の実装を提供する。

- 積型/組型
- 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（基本データ型、集合型、列型、写像型、オプション型、オブジェクト参照型）は、Java 言語自身の一部であるクラス／構成要素かあるいは 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。

Util クラス 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ー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）複製する。しかしながら、VDM++ クラスと基本 VDM++ データ型は複製できない。

equals 2つの VDM 値を比較する。

toString VDM 値の文字表現を含む列を返す。

RunTime 実行時エラーが起きた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` を発生させる。

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` を発生させるが、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` のメソッドを使用しなければならない。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 の Exception ハンドリングに基づいている。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例外が発生する。実装済みエラーは、すべて `CGException` のサブクラスであり、さらに `java.lang.Exception` クラスのサブクラスである。例外クラスの継承構造を、図 11 に示す。

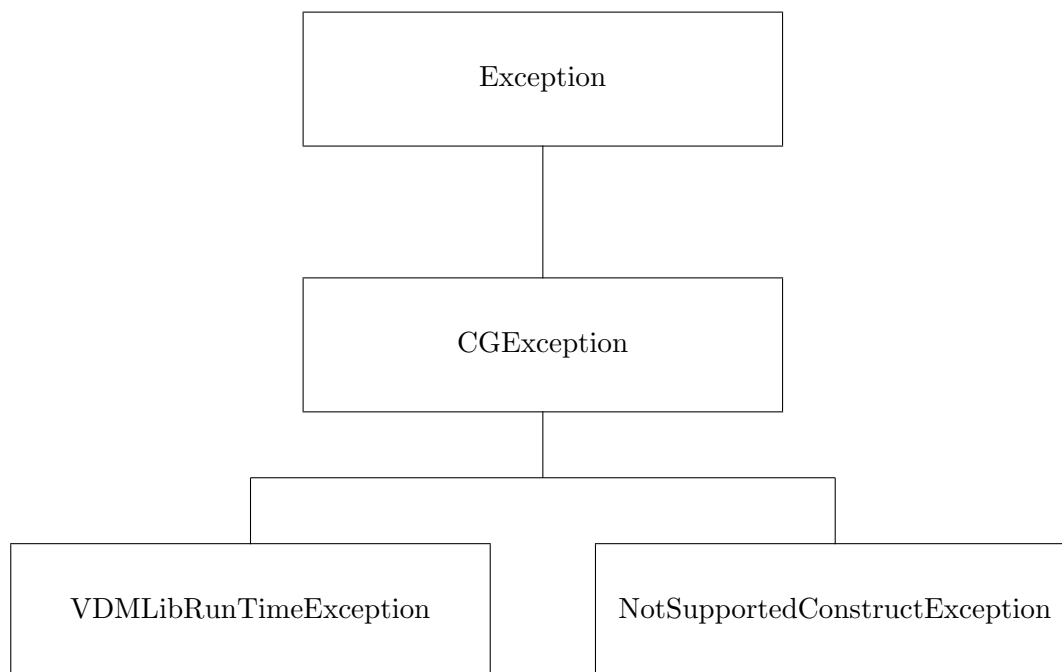


図 11: コードジェネレータの例外を取り扱う Java クラスの継承構造

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:

- 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)
- インスタンス変数初期化 (コンストラクタから呼ばれる)
- 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...

};
```


VDM++ クラスが抽象クラスならば、生成された Java クラスもまた同様に宣言される。

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正確に一致する。

ソート例題についての、VDM++ クラスと生成された Java クラスの継承構造を、図 12 に示す。

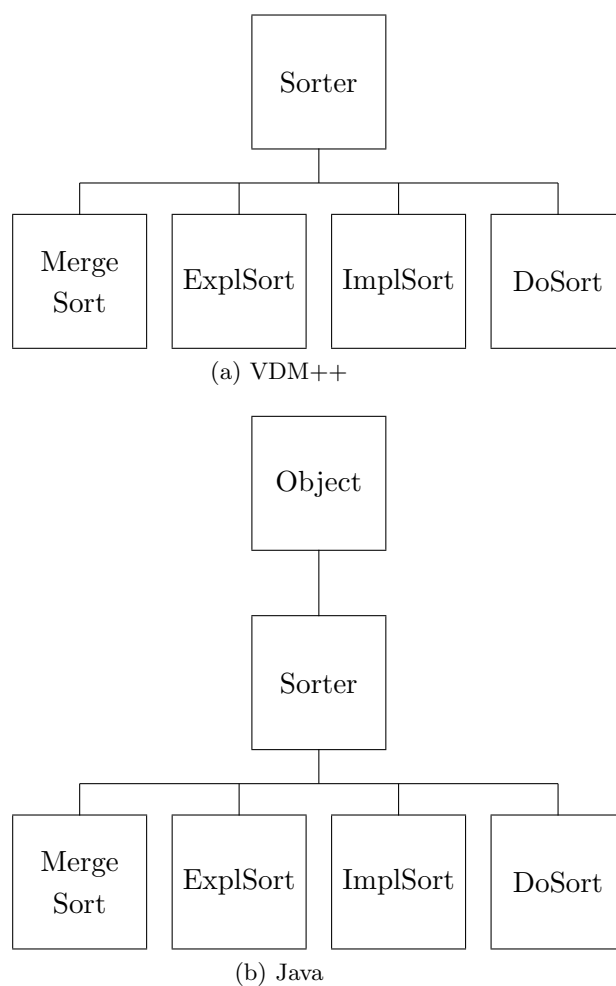


図 12: VDM++ クラスと生成された Java クラスの継承構造

VDM++ では、多重継承を用いて複数のスーパークラスをもつことが許されているが、Java は多重継承をサポートしていない。Java では、多重継承の代わりにインターフェー

ス [JGB00]で行っている。Java のクラスは、1つのスーパークラスを **extends** ことができ、1つ以上のインターフェースを **implements** することができる。インターフェースを実装するためには、クラスは最初に **implements** 節にインターフェースの宣言行い、インターフェー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。これが、VDM++ における多重継承と Java インターフェー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。インターフェースからは追加の **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ー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 のインターフェースモデルは 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++ の多重継承に対して、Java コードを生成するには、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や操作を定義することができ、そのスーパークラスだけがインスタンス変数を持つことができる。このクラスは、Java においては1個のスーパークラスとしてコード生成される。
- 他のすべてのスーパークラスは、インターフェースとして生成されなければならない (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を実装していなければ、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
```

列挙された VDM++ 仕様は、必要な条件を満たしている: 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している。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。クラス **F** はインターフェースとして生成できる。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 コードを以下に示す。

```
public class G extends E implements F {  
  
    // ***** VDMTOOLS START Name=vdm_init_G KEEP=NO  
    private void vdm_init_G () {}  
    // ***** VDMTOOLS END Name=vdm_init_G  
  
    // ***** VDMTOOLS START Name=G KEEP=NO  
    public G () throws CGException {  
        vdm_init_G();  
    }  
    // ***** VDMTOOLS END Name=G  
  
    // ***** VDMTOOLS START Name=getx KEEP=NO  
    public Number getx () throws CGException {  
        if (true)  
            return n;  
        else  
            return i;  
    }  
    // ***** VDMTOOLS END Name=getx  
  
}  
;
```

もし VDM++ 仕様が上記に並べた要求を満たさないならば、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。

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を表示する。

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

注意 VDM++ の *Base Class*、*Class*、*Same Base Class*、*Same Class* 式においては、多重継承の下では、元の VDM++ 仕様と生成された Java コードで、異なる意味を持つことになる。

4.4 コード生成型

この章では、VDM++ 型が Java コードにどのように割り当てられるかを述べる。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、プリミティブ型である `double`、`int`、`boolean`、`char`、に対して“ラッパー”クラスを提供している（それぞれ `Double`、`Integer`、`Boolean`、`Character`）。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 の `Double` クラスに写像される。VDM `nat`、`nat1`、`int` 型は Java の `Integer` クラスに写像される。VDM `bool` 型は Java の `Boolean` クラスに写像される。VDM `char` 型は Java の `Character` クラスに写像される。

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`）が提供されている。`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型に対して、クラス定義を生成する。全引用型は `quotes` パッケージに収集されている。例えば、引用型 `<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>";
    }
};
```

引用型 <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。

token 型 VDM++ 仕様が token 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、例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++の列型 (`seq of char` 型以外)、集合型、写像型は、`java.util` パッケージの `ArrayList` クラス、`HashSet` クラス、`HashMap` クラスに写像される。`HashSet` 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 `comparator` 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ース `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。例えば “(`seq of char` | `seq of nat`)” や “`seq of (char | nat)`” 型は `ArrayList` 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と示される。

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“参照”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、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、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`、`String`、`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** インターフェースを実装する。レコード項目は新しいクラスの変数となる。

例えば、次の合成型³は、

```
public A :: f1 : real
           k : int;
```

以下のように生成される。

```
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());
    }
};
```

³合成型の定義に **compose of** 構文を用いてはならない

レコード各項目に対して、public なメンバー変数が生成されたクラス定義に追加される。これら変数の名称は、相当する VDM レコード項目選択肢の名称と一致する。もし項目選択肢がなければ、例えば上記例題中の **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** がすでに項目選択肢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、単一の項目選択肢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 の interface 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;
Seperator :: ...;
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 Seperator 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ースを実装する。

4.4.3 不変条件

不変条件が VDM++ の型定義をに用いられた場合、不変条件 VDM++ 関数が役に立つ。この不変条件関数が、関連する型定義と同じ範囲内に呼び出されることは可能である ([SCSB] 参照)。生成している事前事後条件の関数／操作に対するオプションが選択されたとき Java コードジェネレータ はこのような不変条件関数に相当する Java メソッドを生成する。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、Java コードジェネレータ は不変条件の動的チェックをサポートしていないこと、しかし不変条件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 値のコード生成

VDM++ 値定義は、生成された Java クラスの `static final` 変数へと変換される。`static` 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 `final` 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 Number a;
  public static final Number b;
  private static final Character c = new Character('a');
  protected static final Number d;
  private static final Number e = new Integer(2 + 1);
}
```

VDM++ 値が単純な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”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 static 初期化子を以下に並べる。

```
static {
    Number atemp = null;
    Number btemp = null;
    Number dtemp = null;
    boolean succ_1 = false;
    try {
        Tuple tmpVar_12 = Tuple.mk_(new Integer(3), new Integer(6));
        /* mk_(atemp, btemp) */
        if (2 == tmpVar_12.Length()) {
            succ_1 = true;
            /* atemp */
            atemp = UTIL.NumberToInt(tmpVar_12.GetField(1));
            /* btemp */
            btemp = UTIL.NumberToInt(tmpVar_12.GetField(2));
        }
        else
            succ_1 = false;
        if (!succ_1)
            UTIL.RunTime("Run-Time Error:Pattern match did not succeed in value definition");
    }
    catch (Throwable e) {
        System.out.println(e.getMessage());
    }
    a = atemp;
    b = btemp;
    try {
        /* dtemp */
        dtemp = new Integer(a.intValue() + 1);
    }
    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変換される。

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 {
  public Number i = null;
  private Number k = null;
  protected String message = null;
  private Number 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。

クラス A のためのコンストラクタメソッドの実装は、以下に示すようにコンストラクタから `vdm_init_A` メソッドを呼び出し、当該メソッド内にてインスタンス変数が初期化される。

```
public class A {
    public A () throws CGException {
        vdm_init_A();
    }

    private void vdm_init_A () {
        try {
            k = new Integer(4);
            message = "";
            j = new Double(1);
        }
        catch (Exception e) {
            e.printStackTrace(System.out);
            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条件定義は、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、関数および操作は陰仕様か陽仕様かの両定義が可能である。Java コードジェネレータは陰仕様と陽仕様の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。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。

すべての生成されたメソッドは **CGException** 例外を throws する。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 陽仕様な関数および操作定義

陽仕様である 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 は陽仕様であり、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 List Sort (final List l) throws CGException{  
    ...  
}
```

4.7.2 予備的な関数操作定義

陽仕様である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 の抽象メソッドに変換されている。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 陽仕様の関数と操作の定義

陽仕様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、Java コードジェネレータによって相当の事前・事後条件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も生成可能である。しかし、操作における事後条件は、Java コードジェネレータ

によって無視される。“事前／事後条件関数を生成する” 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・事後条件メソッドは、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。“事前／事後条件をチェックする” 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 (操作の事後条件を含まない)。

4.8 式と文のコード生成

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

undefined 式と error 文は、VDM Java ライブラリ で見つけられ `VDMRunTimeException` を起こす関数 `UTIL.RunTime` の呼び出しに変換される。

4.9 名称変換

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ー u (‘_u’) やアンダースコアー 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に許している。

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 への変換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 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ースから、“並列構成でコードを生成する” オプションを選択する。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ることができる。

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

5.3 変換手引き

並列 VDM++ 仕様のコード生成は、シーケンシャル仕様のコード生成よりは直接的でない。なぜならば、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。特に変換手引きは、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。これは変換手引きが、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となる翻訳方法を説明する。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 Java コードジェネレータの使用が必要とされる場合にのみ参照すれば良い。手引きのさらに詳細な記述は [Opp] にある。

5.3.1 コア翻訳

この章では、基本となる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 に含まれる。

- `evaluatePP` メソッド
- インナー `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` インターフェー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` インターフェースを実装する。これは `EvaluatePP` インターフェー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` インターフェー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実行さ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 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 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、定期的に実行さ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 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

- [JGB00] Guy Steele James Gosling, Bill Joy and Gilad Bracha. *The Java Language Specification, Second Edition*. The Java Series. Addison Wesley, 2000.
- [Opp] Oliver Oppitz. Concurrency extensions for the vdm++ to java code generator of the vdm++ toolbox. Master's thesis.
- [SCSa] SCSK. *VDM++ Installation Guide*. SCSK.
- [SCSb] SCSK. *The VDM++ Language*. SCSK.
- [SCSc] SCSK. *VDM++ Toolbox User Manual*. SCSK.

A コードジェネレータのインストール

Java コードジェネレータ は VDM++ Toolbox に含まれる機能である。インストールについては [SCSa] に記述されている。配布中に次の名称のディレクトリがある。

javacg

このディレクトリには、4つのファイルとその他3つのディレクトリが含まれる:

VDM.jar VDM Java ライブラリ (付録 B 参照)

IO.java 標準ライブラリクラス IO の java 実装ファイル

MATH.java 標準ライブラリクラス MATH の java 実装ファイル

VDMUtil.java 標準ライブラリクラス VDMUtil の java 実装ファイル

libdoc VDM Java ライブラリ の HTML 文書を含める (付録 B 参照)

javasort 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（`VDM.jar` ファイル）を含めなければならない。このファイルは次のディレクトリに置かれている。

```
javacg/
```

さらに、次のディレクトリ

```
javacg/libdoc
```

には、`javadoc` により生成された このライブラリの HTML 文書が含まれている。

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ている DoSort 例題は、javacg/javasort という名称のディレクトリ内に置かれている。このディレクトリには以下のファイルが含まれる。

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

java ファイルは javacg/javasort 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することができる。

```
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、メインプログラムを以下のコマンドで実行する。

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

Windows OS 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(1) ==
```

```
    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] ^ l */
    /* (i <= hd l) */
    succ_4 = (UTIL.equals(Boolean.TRUE, Boolean.valueOf(i.intValue() <= UTIL.NumberToInt(l.get
if (succ_4) {
    /* [i] ^ l */
    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 メインプログラム (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(23);
arr.add(1);
arr.add(42);
arr.add(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());
}
}
}
```