# 運賃計算システム VDM++ 仕様

佐原　伸

SCSK Corporation
VDM 推進担当

**概要**

レコードと列を使った要求仕様の例。
距離から運賃を求めるようにしている。
Makefile を使って、make コマンドで、モデルの検証とドキュメントの PDF ファイル生成を一挙に行うようにしている。

## 目次

# 1 クラス図

VDM++ 仕様のクラス図は以下の通りである。

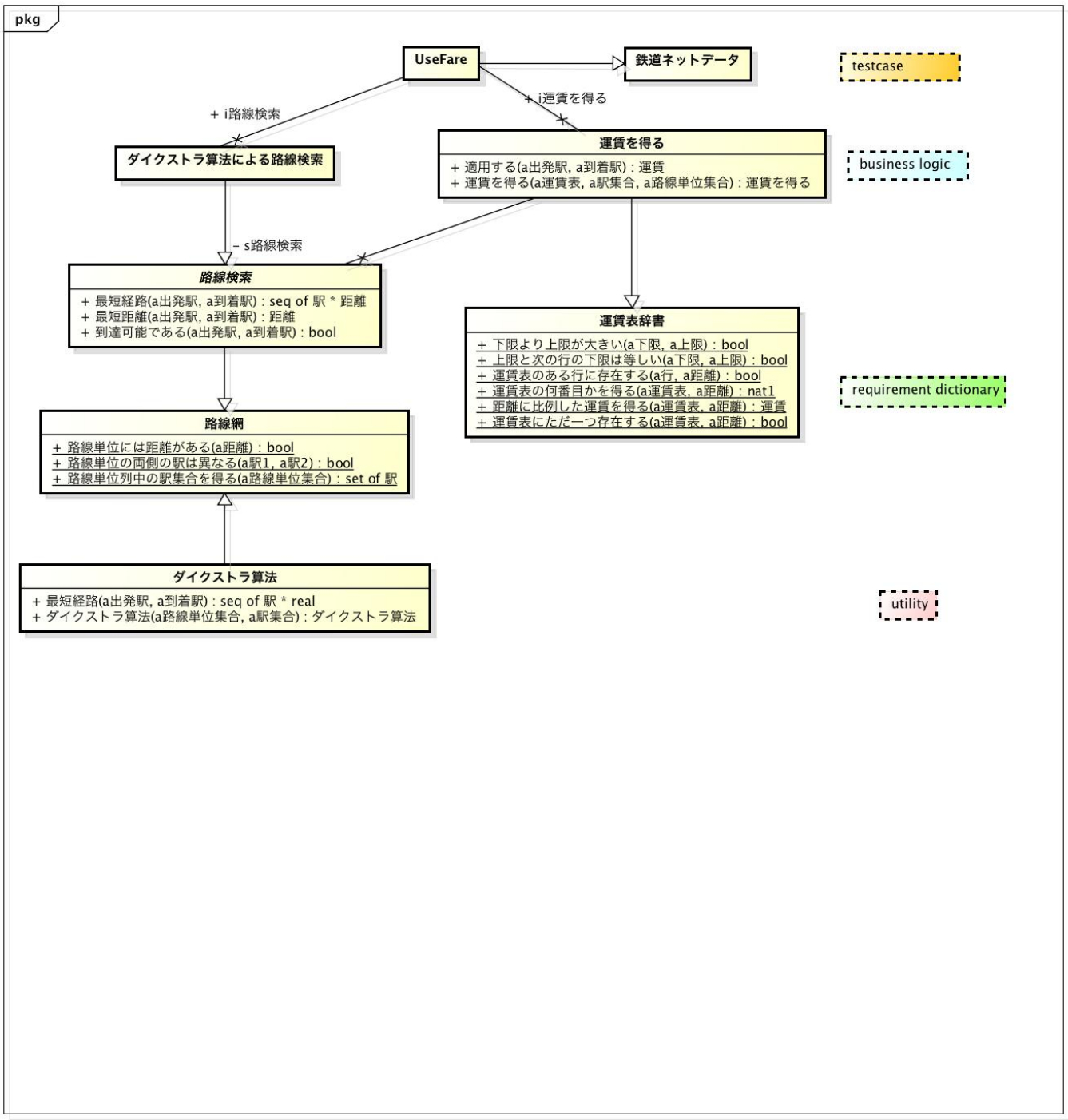


図 1　Class Diagram

# 2 運賃表辞書

運賃表の用語 (名詞と述語) を定義する要求辞書 (Requirement Dictionary) である。

```
class
運賃表辞書
types
  public 運賃 = nat;
public
  行 :: f 下限 : 路線網 ‘距離
       f 上限 : 路線網 ‘距離
       f 運賃 : – 運賃;
```

## 2.1 運賃表 (型)

運賃表の各行の上限と下限は重複しないこと。

```
  public 運賃表 = seq of 行
  inv w 運賃表 ==
       forall i, j in set inds w 運賃表 &
           下限より上限が大きい (w 運賃表 (i).f 下限, w 運賃表 (i).f 上限) and
           j = i + 1 =>
           上限と次の行の下限は等しい (w 運賃表 (i).f 上限, w 運賃表 (j).f 下限)
functions
public static
  距離に比例した運賃を得る : 運賃表 ∗ 路線網 ‘距離 +> 運賃
  距離に比例した運賃を得る (a 運賃表, a 距離) ==
    let n = 運賃表の何番目かを得る (a 運賃表, a 距離) in
    a 運賃表 (n).f 運賃
  pre  運賃表にただ一つ存在する (a 運賃表, a 距離)
  post let n = 運賃表の何番目かを得る (a 運賃表, a 距離) in
       RESULT = a 運賃表 (n).f 運賃 ;
public static
  運賃表のある行に存在する : 行 ∗ 路線網 ‘距離 -> bool
  運賃表のある行に存在する (a 行, a 距離) ==
    a 行.f 下限 <= a 距離 and a 距離 < a 行.f 上限;
public static
  運賃表にただ一つ存在する : 運賃表 ∗ 路線網 ‘距離 +> bool
  運賃表にただ一つ存在する (a 運賃表, a 距離) ==
    exists1 i in set inds a 運賃表 & 運賃表のある行に存在する (a 運賃表 (i), a 距離);
public static
```

```
  運賃表の何番目かを得る : 運賃表 * 路線網 * 距離 +> nat1
  運賃表の何番目かを得る (a 運賃表, a 距離) ==
    let  i in set inds a 運賃表be st  運賃表のある行に存在する (a 運賃表 (i), a 距離) in
    i;
public static
  下限より上限が大きい : 運賃 * 運賃 +> bool
  下限より上限が大きい (a 下限, a 上限) ==
    a 下限 < a 上限;
public static
  上限と次の行の下限は等しい : 運賃 * 運賃 +> bool
  上限と次の行の下限は等しい (a 下限, a 上限) ==
    a 下限 = a 上限
end
```

運賃表辞書

**Test Suite :** vdm.tc

**Class :** 運賃表辞書

| Name | #Calls | Coverage |
|---|---|---|
| 運賃表辞書 `下限より上限が大きい | 1080 | √ |
| 運賃表辞書 `運賃表の何番目かを得る | 10 | √ |
| 運賃表辞書 `距離に比例した運賃を得る | 5 | √ |
| 運賃表辞書 `運賃表にただ一つ存在する | 5 | √ |
| 運賃表辞書 `運賃表のある行に存在する | 60 | √ |
| 運賃表辞書 `上限と次の行の下限は等しい | 150 | √ |
| **Total Coverage** | | **100%** |

# 3 路線網

運賃計算に限らず使用することのできる、路線網のドメインモデルである。

```
class
路線網
types
  public 駅 = token;
  public 距離 = real;
  public 駅集合 = set of 駅
  inv w 駅集合 == card w 駅集合 >= 2;
```

## 3.1 路線単位レコード

路線単位レコードは、両端の駅が同一にも関わらず複数の距離があることは考慮していない。

```
public
  路線単位 :: f 駅 1 : 駅
              f 駅 2 : 駅
              f 距離 : 距離
  inv w 路線単位 == 路線単位の両側の駅は異なる (w 路線単位.f 駅 1, w 路線単位.f 駅 2) and 路線単位には距離がある (w 路線単位.f 距離);
  public 路線単位集合 = set of 路線単位
  inv w 路線単位集合 == w 路線単位集合 <> {}
```

## 3.2 関数

```
functions
public static
  路線単位の両側の駅は異なる : 駅 * 駅 +> bool
  路線単位の両側の駅は異なる (a 駅 1, a 駅 2) ==
    a 駅 1 <> a 駅 2;
public static
  路線単位には距離がある : 距離 +> bool
  路線単位には距離がある (a 距離) ==
    a 距離 > 0;
public static
  路線単位列中の駅集合を得る : 路線単位集合 +> set of 駅
  路線単位列中の駅集合を得る (a 路線単位集合) ==
    dunion {{w 路線単位.f 駅 1, w 路線単位.f 駅 2}|w 路線単位 in set a 路線単位集合 }
end
路線網
```

**Test Suite :** vdm.tc
**Class :** 路線網

| **Name** | **#Calls** | **Coverage** |
|---|---|---|
| 路線網 ‘路線単位には距離がある | 1894 | √ |
| 路線網 ‘路線単位の両側の駅は異なる | 1894 | √ |
| 路線網 ‘路線単位列中の駅集合を得る | 20 | √ |
| **Total Coverage** | | **100%** |

# 4 路線検索

路線検索の抽象クラスである。

```
class
路線検索 is subclass of 路線網
types
public
  経路条件 :: 発駅 : 駅
              着駅 : 駅
  inv w 経路条件 ==
      w 経路条件.発駅 <> w 経路条件.着駅
instance variables
  protected s 駅集合 : 駅集合;
  protected s 路線単位集合 : 路線単位集合;
  inv let w 路線単位集合中の駅集合 = 路線単位列中の駅集合を得る (s 路線単位集合) in
      w 路線単位集合中の駅集合 subset s 駅集合

operations
public
  最短経路 : 駅 * 駅 ==> seq of 駅 * 距離
  最短経路 (a 出発駅, a 到着駅) ==
    is subclass responsibility;
public
  最短距離 : 駅 * 駅 ==> 距離
  最短距離 (a 出発駅, a 到着駅) ==
    (   def mk_ (-, w 最短距離) = 最短経路 (a 出発駅, a 到着駅) in
        return w 最短距離
    )
  pre  到達可能である (a 出発駅, a 到着駅) ;
public
  到達可能である : 駅 * 駅 ==> bool
  到達可能である (a 出発駅, a 到着駅) ==
    let mk_ (-, d) = 最短経路 (a 出発駅, a 到着駅) in
    return d > 0
end
```

路線検索

**Test Suite :** vdm.tc

**Class :** 路線検索

| Name | #Calls | Coverage |
|---|---|---|
| 路線検索 ‘最短経路 | 0 | 0% |
| 路線検索 ‘最短距離 | 12 | √ |
| 路線検索 ‘到達可能である | 18 | √ |
| **Total Coverage** | | **94%** |

# 5 ダイクストラ算法による路線検索

ダイクストラ算法にによって最短経路を求める。

```
class
ダイクストラ算法による路線検索 is subclass of 路線検索
operations
public
  ダイクストラ算法による路線検索 : 駅集合 * 路線単位集合 ==> ダイクストラ算法による路線検索
  ダイクストラ算法による路線検索 (a 駅集合, a 路線単位集合) == atomic
    (  s 駅集合 := a 駅集合;
       s 路線単位集合 := a 路線単位集合
    );
```

## 5.1 最短経路

ダイクストラ算法にによって、最短経路の路線単位列を求め、経路に変換する。

経路が繋がっていない場合は空列を返す。

```
public
  最短経路 : 駅 * 駅 ==> seq of 駅 * 距離
  最短経路 (a 出発駅, a 到着駅) ==
    (  dcl w 最短距離アルゴリズム : ダイクストラ算法 := new ダイクストラ算法 (s 路線単位集合, s 駅
集合);
       return w 最短距離アルゴリズム.最短経路 (a 出発駅, a 到着駅)
    )
end
ダイクストラ算法による路線検索
```

**Test Suite :** vdm.tc

**Class :** ダイクストラ算法による路線検索

| Name | #Calls | Coverage |
|---|---|---|
| ダイクストラ算法による路線検索 ‘最短経路 | 32 | √ |
| ダイクストラ算法による路線検索 ‘ダイクストラ算法による路線検索 | 10 | √ |
| **Total Coverage** | | **100%** |

# 6 ダイクストラ算法

路線単位の長さが非負である鉄道グラフ（ネットワーク）の上で、ある駅から他の任意の駅への最短経路・最短距離を求める。

「アルゴリズム辞典」のアルゴリズム (p.455) をそのまま使用。ただし、配列は写像を使って実装した。

```
class
ダイクストラ算法 is subclass of 路線網
types
  public 確定 = <未確定> | <既確定> ;
  public X 確定 = map 駅 to 確定;
  public V 最短距離 = map 駅 to real;
  public P 直前の駅 = map 駅 to 駅
instance variables
  public s 路線単位集合 : 路線単位集合;
  public s 駅集合 : 駅集合;
  public x : X 確定 := { |-> };
  public v : V 最短距離 := { |-> };
  public p : P 直前の駅 := { |-> };

values
  v 無限大 = 10000000000000000
operations
public
  ダイクストラ算法 : set of 路線単位 * set of 駅 ==> ダイクストラ算法
  ダイクストラ算法 (a 路線単位集合, a 駅集合) ==
    (   s 路線単位集合 := a 路線単位集合;
        s 駅集合 := a 駅集合
    ) ;
public
  最短経路 : 駅 * 駅 ==> seq of 駅 * real
  最短経路 (a 出発駅, a 到着駅) ==
    (   dcl i : 駅 := a 出発駅;
        for all w 駅 in set s 駅集合
        do if w 駅 = a 出発駅
           then (   x(a 出発駅) := <既確定> ;
                    v(a 出発駅) := 0
                )
```

```
            else (   x(w 駅) := <未確定> ;
                     v(w 駅) := v 無限大
                 ) ;
        for all w 駅 in set s 駅集合
        do (   def Ni = {u.f 駅 2|u in set s 路線単位集合 & u.f 駅 1 = i};
                   Nu = {u|u in set s 路線単位集合 & u.f 駅 1 = i} in
               (   for all j in set Ni
                   do (   if x(j) = <未確定>
                          then def w = v(i) + d(Nu, i, j) in
                               if w < v(j)
                               then (   v(j) := w;
                                        p(j) := i
                                    )
                      ) ;
                   def Ni 未確定 = {e|e in set Ni & x(e) = <未確定> } in
                   if Ni 未確定 <> {}
                   then let   s in set Ni 未確定be st   forall s1 in set Ni 未確定 & v(s) <= v(s1) in
                        (   i := s;
                            x(i) := <既確定>
                        )
               )
           ) ;
        def w 最短経路 = 経路を作る(p, a 出発駅, a 到着駅);
            w 最短距離 = v(a 到着駅) in
        return mk_(w 最短経路, w 最短距離)
    )
functions
  d : 路線単位集合 * 駅 * 駅 +> real
  d(a 路線単位集合, a 駅 1, a 駅 2) ==
    let   di in set a 路線単位集合be st   di.f 駅 1 = a 駅 1 and di.f 駅 2 = a 駅 2 in
    di.f 距離
operations
  経路を作る : P 直前の駅 * 駅 * 駅 ==> seq of 駅
  経路を作る(aP 直前の駅, a 出発駅, a 到着駅) ==
    (   dcl w 最短経路 : seq of 駅 := [],
            w 駅 : 駅 := a 到着駅;
        while w 駅 <> a 出発駅
```

```
        do (   w 最短経路 := [w 駅]^w 最短経路;
               w 駅 := aP 直前の駅(w 駅)
           );
        return [a 出発駅]^w 最短経路
    )
  pre  a 到着駅 in set dom aP 直前の駅
end
```

ダイクストラ算法

**Test Suite :** vdm.tc

**Class :** ダイクストラ算法

| Name | #Calls | Coverage |
|---|---|---|
| ダイクストラ算法‘d | 192 | √ |
| ダイクストラ算法‘最短経路 | 32 | √ |
| ダイクストラ算法‘経路を作る | 32 | √ |
| ダイクストラ算法‘ダイクストラ算法 | 32 | √ |
| **Total Coverage** | | **100%** |

# 7 TestSimple

鉄道の運賃計算仕様の回帰テストケース（単純版）である。

```
class
TestSimple is subclass of 鉄道ネットデータ
values
public
  v最大値 = 100000000
instance variables
  public s路線検索：ダイクストラ算法による路線検索 := new ダイクストラ算法による路線検索(v駅集合,v路線単位集合);
  public s運賃を得る：運賃を得る := new 運賃を得る([
                                    mk_運賃表辞書‘行(0,1,150),
                                    mk_運賃表辞書‘行(1,3,160),
                                    mk_運賃表辞書‘行(3,6,190),
                                    mk_運賃表辞書‘行(6,10,220),
                                    mk_運賃表辞書‘行(10,15,250),
                                    mk_運賃表辞書‘行(15,v最大値,300)],
                                   v駅集合,
                                   v路線単位集合);

operations
public
  run : () ==> seq of char * bool * map nat to bool
  run() ==
    let testcases = [
                    t1(),t2(),t3(),t4(),t5(),
                    t6(),t7(),t8()],
        testResults = makeOrderMap(testcases) in
    return mk_("回帰テスト結果 = ", forall i in set inds testcases & testcases(i),testResults)
functions
public static
  makeOrderMap : seq of bool -> map nat to bool
  makeOrderMap(s) ==
    {i |-> s(i)|i in set inds s}
operations
public
```

```
  print : seq of char ==> ()
  print (s) ==
    let - = new IO().echo (s) in
    skip;
public
  t1 : () ==> bool
  t1 () ==
    let mk_(-, d) = s 路線検索.最短経路 (v 四ツ谷, v 品川) in
    return d = 9.5 and
           s 路線検索.最短距離 (v 四ツ谷, v 品川) = 9.5;
public
  t2 : () ==> bool
  t2 () ==
    trap <RuntimeError> with (  print("\t t2 意図した事前条件エラーが発生した。\n");
           return true
         ) in
    return s 路線検索.到達可能である (v 東京, v コペンハーゲン) = false;
public
  t3 : () ==> bool
  t3 () ==
    let mk_(r, -) = s 路線検索.最短経路 (v 東京, v 新宿) in
    return r = [v 東京, v 四ツ谷, v 新宿] and
           s 路線検索.最短距離 (v 東京, v 新宿) = 7.7;
public
  t4 : () ==> bool
  t4 () ==
    let mk_(r, -) = s 路線検索.最短経路 (v 池袋, v 四ツ谷) in
    return r = [v 池袋, v 新宿, v 四ツ谷] and
           s 路線検索.最短距離 (v 池袋, v 四ツ谷) = 8.6;
public
  t5 : () ==> bool
  t5 () ==
    let mk_(r, d) = s 路線検索.最短経路 (v 品川, v 四ツ谷) in
    return r = [v 品川, v 東京, v 四ツ谷] and
           d = 9.3;
public
  t6 : () ==> bool
  t6 () ==
    return s 運賃を得る.適用する (v 池袋, v 東京) = 250;
```

```
public
  t7 : () ==> bool
  t7() ==
    return s運賃を得る.適用する(v池袋, v新宿) = 190;
public
  t8 : () ==> bool
  t8() ==
    return s運賃を得る.適用する(v四ツ谷, v品川) = 220
end
TestSimple
```

**Test Suite :** vdm.tc

**Class :** TestSimple

| Name | #Calls | Coverage |
|---|---|---|
| TestSimple‘t1 | 1 | √ |
| TestSimple‘t2 | 1 | 93% |
| TestSimple‘t3 | 1 | √ |
| TestSimple‘t4 | 1 | √ |
| TestSimple‘t5 | 1 | √ |
| TestSimple‘t6 | 1 | √ |
| TestSimple‘t7 | 1 | √ |
| TestSimple‘t8 | 1 | √ |
| TestSimple‘run | 1 | √ |
| TestSimple‘print | 1 | √ |
| TestSimple‘makeOrderMap | 1 | √ |
| **Total Coverage** | | **99%** |

# 8 TestApp Class

## 8.1 責任

「運賃を得る」クラス、及び関連するクラスの回帰テストを行う。

```
class
TestApp
```

## 8.2 操作 : run

回帰テストケースを Testsuite に追加し、回帰テストを行い、結果を判定する。

```
operations
public
  run : () ==> ()
  run () ==
    (   dcl ts : TestSuite := new TestSuite ("鉄道運賃計算の回帰テスト。\n"),
            tr : TestResult := new TestResult ();
        tr.addListener(new PrintTestListener ()) ;
        ts.addTest(new TestCaseT0001 ("TestCaseT0001 計算に成功するケース。\n")) ;
        ts.addTest(new TestCaseT0002 ("TestCaseT0002 計算に成功するケース。\n")) ;
        ts.addTest(new TestCaseT0003 ("TestCaseT0003 事前条件エラーを検出するケース。\n")) ;
        ts.addTest(new TestCaseT0004 ("TestCaseT0004 事前条件エラーを検出するケース。\n")) ;
        ts.run(tr) ;
        if tr.wasSuccessful () = true
        then def - = new IO ().echo ("***  すべての回帰テストが成功した。  ***\n") in
             skip
        else def - = new IO ().echo ("***  失敗した回帰テストケースがある。  ***\n") in
             skip
    )
end
TestApp
```

**Test Suite :** vdm.tc

**Class :** TestApp

| Name | #Calls | Coverage |
|---|---|---|
| TestApp`run | 1 | 85% |
| **Total Coverage** | | **85%** |

# 9 回帰テストケース

## 9.1 責任

「運賃を得る」クラス、及び関連するクラスをテストする。

```
class
TestCaseT is subclass of TestCase, 鉄道ネットデータ
values
public
  v 最大値 = 100000000
instance variables
  public s 路線検索 : 路線検索 := new ダイクストラ算法による路線検索 (v 駅集合, v 路線単位集合);
  public s 運賃を得る : 運賃を得る := new 運賃を得る ([
                                        mk_運賃表辞書 `行 (0, 1, 150),
                                        mk_運賃表辞書 `行 (1, 3, 160),
                                        mk_運賃表辞書 `行 (3, 6, 190),
                                        mk_運賃表辞書 `行 (6, 10, 220),
                                        mk_運賃表辞書 `行 (10, 15, 250),
                                        mk_運賃表辞書 `行 (15, v 最大値, 300)],
                                       v 駅集合,
                                       v 路線単位集合);

operations
public
  print : seq of char ==> ()
  print (s) ==
    let - = new IO ().echo (s) in
    skip
end
TestCaseT
class
TestCaseT0001 is subclass of TestCaseT
operations
public
  TestCaseT0001 : seq of char ==> TestCaseT0001
  TestCaseT0001 (name) ==
    setName(name) ;
public
```

```
  test01 : () ==> ()
  test01() ==
    (   def wDistance =
                s 路線検索.最短距離(v 東京, v 新宿) in
        assertTrue("\t test01 計算結果が間違っている \n",
             wDistance = 7.7 and
             s 運賃を得る.適用する(v 東京, v 新宿) = 220)
    )
end
TestCaseT0001
class
TestCaseT0002 is subclass of TestCaseT
operations
public
  TestCaseT0002 : seq of char ==> TestCaseT0002
  TestCaseT0002(name) ==
    setName(name);
public
  test01 : () ==> ()
  test01() ==
    (   def wDistance =
                s 路線検索.最短距離(v 四ツ谷, v 品川) in
        assertTrue("\t test01 計算結果が間違っている。\n",
             wDistance = 9.5 and
             s 運賃を得る.適用する(v 四ツ谷, v 品川) = 220)
    )
end
TestCaseT0002
class
TestCaseT0003 is subclass of TestCaseT
operations
public
  TestCaseT0003 : seq of char ==> TestCaseT0003
  TestCaseT0003(name) ==
    setName(name);
public
```

```
  test01 : () ==> ()
  test01() ==
    (   trap  <RuntimeError>
        with  print("\t test01 期待した事前条件エラーを検出した。\n") in
        (   def – =
                    s 路線検索.最短距離(v 東京, v 東京) in
            print("\t test01 予想外のエラーに遭遇。\n")
        )
    )
end
TestCaseT0003
class
TestCaseT0004 is subclass of TestCaseT
operations
public
  TestCaseT0004 : seq of char ==> TestCaseT0004
  TestCaseT0004(name) ==
    setName(name);
public
  test01 : () ==> ()
  test01() ==
    (   trap  <RuntimeError>
        with  print("\t test01 期待した事前条件エラーを検出した。\n") in
        (   def – =
                    s 路線検索.最短距離(v 東京, v コペンハーゲン) in
            print("\t test01 予想外のエラーに遭遇。\n")
        )
    )
end
TestCaseT0004
```

# 10 参考文献、索引

VDM++[2] は、1970 年代中頃に IBM ウィーン研究所で開発された VDM-SL[1] を拡張し、さらにオブジェクト指向拡張した[*1]の形式仕様記述言語である。

## 参考文献

[1] CSK システムズ. VDM-SL 言語マニュアル. CSK システムズ, 第 1.1 版, 2007. Revised for VDMTools V7.1.

[2] CSK システムズ. VDM++ 言語マニュアル. CSK システムズ, 第 1.1 版, 2007. Revised for VDMTools V7.1.

[*1] 使用に際しては、（株）CSK システムズとの契約締結が必要になる。

# 索引