VDM++ 言語入門セミナー　演習問題

(Revision : 0.2 – 2017 年 4 月 11 日)

佐原　伸

概要

「対象を如何にモデル化するか？」のモデル作成例。

目次

1　図書館システムへの要求

以下の機能を持つ図書館システムを、ユースケースレベルの要求仕様としてモデル化することを考えよう。

1. 本を図書館の蔵書として追加、削除する。
2. 題名と著者と分野のいずれかで本を検索する。
3. 利用者が、蔵書を借り、返す。
4. 蔵書の追加・削除や、利用者への貸出・返却は職員が行う。
5. 利用者や職員の権限などは考慮しなくてよい。
6. 最大蔵書数は 10000、利用者一人への最大貸出冊数は 3 冊とする。

1.1　VDM++ のモジュール

VDM++ は、オブジェクト指向仕様記述言語であると同時に、関数型言語の機能を持った仕様記述言語でもあるので、モデルを作る際「オブジェクト指向」であることにこだわることはない。一般に、関数型指向でモデルを作成した方が抽象度が高く、行数も少なくモデルを作成できることが多い。オブジェクト指向モデルは、より詳細な、設計よりのモデル化に向いている。

そこで、まず、オブジェクト指向にこだわらないモデル化を考える。しかし、VDM++ のモジュール記述単位はクラスなので、図書館クラスを VDM++ のモジュールとして考える。

2　実行不可能な仕様

図書館システムとしては、最初のモデル化なので、実行可能な陽仕様とはせず、事後条件・事前条件・不変条件などを考えた実行不可能な形で陰仕様を記述する。

陰仕様は、実行はできないものの「仕様である」と言ってよい。

```
class
図書館 0
```

2.1　型定義

日本語仕様では、抽象的な意味の本と、図書館の蔵書として管理すべき個々の本が明確に区別されていない。図書館では、同じ本を何冊か蔵書として持つことがあるので、以下のモデルでは、著者や題名という属性を持つ本に対し、管理対象となる個々の蔵書を蔵書 ID から本への写 (maplet) として定義し、蔵書を蔵書 ID から本への写像 (map) とする。蔵書型は、濃度 (集合の要素数) が 10000 以下であるという不変条件を持つ。ただし、10000 という数は、ここで直接書くのではなく、仕様を読みやすくするため values 句で定数定義を行うこととする。

```
types
```

```
public 蔵書 = map 蔵書 ID to 本
inv w 蔵書 == card dom w 蔵書 <= v 最大蔵書数;
```

型名の前の予約語 public は、オブジェクト外部からアクセスできるかどうかを示すもので、public の場合、名前がオブジェクト外に公開されていることを示す。protected の場合は、サブクラスのオブジェクトにのみ公開されていて、private の場合だとクラス外には未公開であることを示す。

今後のモデル化で、項目が追加される可能性が強いので、本はレコード型として定義した。

著者や題名や分野は文字型 (char) の列 (seq) である文字列 (seq of char) とした。 著者は、モデルを詳細化するに伴いオブジェクトとしてモデル化する可能性があるが、ここでは単に文字列として管理することにしたわけである。

```
public
  本::f 題名 : 題名
        f 著者 : 著者
        f 分野集合 : set of 分野;
  public 題名 = seq of char;
  public 著者 = seq of char;
  public 分野 = seq of char;
```

蔵書を一意に識別するため、型として蔵書 ID を定義する。蔵書 ID は、token 型とする。token 型は、他と一意に識別できる型であるという性質を持つだけの「抽象的な定義」を行うのに便利な型である。より詳細な仕様にしていく場合、token 型を他のより具体的な型に詳細化していく。

```
public 蔵書 ID = token;
```

職員と利用者も token 型として定義する。職員も利用者も共に、今着目している機能では「存在している」こと以外に積極的な役割がないからである。権限などの機能が導入された場合、より詳細に定義していく必要があるだろう。

```
public 職員 = token;
public 利用者 = token
```

上記で、レコード型である本の欄 (field) 名の頭に f が付いているのは、VDM++ の文法ではなく弊社における名前付け規則である。VDM++ では、クラス名や型名あるいは後述する関数名などの、すべての識別子がユニークな名前を持たなければならないので、重複が起こらないよう名前付け規則を定めた。

具体的には、以下のように決めた。

- レコード型の欄 (フィールド) 名は、先頭に field を表す f を付ける。
- インスタンス変数名は、先頭に instance を表す i を付ける。
- 定数名は、先頭に values を表す v を付ける。

- 局所的に使う識別子名は、先頭に work area を表す w を付ける。[*1]
- 局所的に使う識別子名であっても、仮引数としてのパラメータ名は、先頭に「一つの」を表す a を付ける。

2.2 インスタンス変数定義

蔵書 ID から本への写像である蔵書をインスタンス変数として持つようにする。初期値は空の写像である。

```
instance variables
  private i 蔵書 : 蔵書 := { |-> };
```

上記で、dom は写像型にあらかじめ定義された演算子で、写像の定義域の集合を返す。今の場合、蔵書 ID の集合を返す。

card は集合型の演算子で、濃度を返す。

2.3 定数定義

最大蔵書数 10000 冊と利用者への最大貸出数 3 冊を定義している。通常のプログラミング言語とは異なり、VDM++ は通常の定数定義以外に、関数やオブジェクトを定数として定義することができるが、上級の話題なので、これ以上触れない。

```
values
public
  v 最大蔵書数 = 10000;
public
  v 最大貸出数 = 3
```

2.4 操作定義

2.4.1 蔵書を追加する

「本を図書館の蔵書として追加する」機能は、「蔵書を追加する」操作でモデル化する。

引数は追加する蔵書を表す蔵書型で、事後条件 (post) で、確かに蔵書が追加されたことを、関数「蔵書が追加されている」を呼び出して確認している。関数「蔵書が追加されている」については後述する。

事後条件が true となれば事後条件が満たされ、false であれば事後条件は満たされない。事後条件は、どうなるかではなく、どうなっていて欲しいかを記述するものなので、要求仕様あるいは設計仕様の核心であると言ってよい。

事後条件では、引数、インスタンス変数、結果を表す予約後 RESULT を使うことができる。

[*1] local を表す l は、数字の 1 と間違いやすいので、作業用に局所的に使う名前といういう意味で w にした。

操作の本体は is not yet specified という予約語で、まだ記述していないことを示している。この部分を具体的に書けば実行可能な仕様となる。

事前条件 (pre) は、操作が受け入れ可能な引数やインスタンス変数の条件を、true を返す論理式として記述する。 ここでは、追加する本を表す a 追加本 が、すでに蔵書中に存在しないことを、関数「蔵書に存在しない」を使って確認している。関数「蔵書に存在しない」についても後述する。

モデルをより詳細化していく場合、この事前条件を削除して、操作本体内でエラー処理仕様として記述していく必要がある。

事後条件は、関数「蔵書が追加されている」にインスタンス変数「i 蔵書」やその旧値 (識別子の後ろに'~'を付けると旧値を意味する) などを渡して判定している。旧値 (old value) は、操作が動く前の (すなわち、事前条件判定時の) インスタンス変数の値を示す。今の場合、「i 蔵書~」は、インスタンス変数「i 蔵書」の旧値であり、事前条件に出てくる「i 蔵書」と同じ値を持つ。事後条件に出てくる「i 蔵書」は、蔵書が追加されたあとの「i 蔵書」の値を示す。

```
operations
public
  蔵書を追加する : 蔵書 ==> ()
  蔵書を追加する (a 追加本) ==
    is not yet specified
  pre  蔵書に存在しない (a 追加本, i 蔵書)
  post 蔵書が追加されている (a 追加本, i 蔵書, i 蔵書~) ;
```

2.4.2 蔵書を削除する

「本を図書館の蔵書として削除する」機能は、「蔵書を削除する」操作でモデル化する。「蔵書を追加する」操作の場合と同様に、関数「蔵書に存在する」といった、日本語仕様風の関数を呼ぶことで、仕様の読みやすさを向上させる。[*2]

貸出している本を蔵書から削除するわけにはいかないので、「貸出に存在しない」ことを事前条件で確認しておく。

```
public
  蔵書を削除する : 蔵書 ==> ()
  蔵書を削除する (a 削除本) ==
    is not yet specified
  pre  蔵書に存在する (a 削除本, i 蔵書) and
       貸出に存在しない (a 削除本, i 貸出)
  post 蔵書が削除されている (a 削除本, i 蔵書, i 蔵書~) ;
```

[*2] 仕様の行数はやや多くなるが、VDM++ にあまり詳しくない人でも一応理解できることが重要である。

2.4.3　題名と著者と分野で本を検索する

「題名と著者と分野で本を検索する」機能は、「本を検索する」操作でモデル化する。

「本を検索する」操作の事後条件は、予約語 RESULT で表されている返り値の蔵書中で、引数「a 検索キー」で指定された文字列が、題名、著者、分野のいずれかの文字列と一致することである。

forall 以下は、全称限量式と呼ばれ、in set の後の式で表される集合 (今の場合、返り値を表す RESULT の値域、すなわち本の集合) の要素 (いまの場合本を表す book) 全てについて、&以下の条件が成り立てば true、そうでなければ false を返す式である。この全称限量式が true であれば事後条件が満たされ、 false であれば事後条件違反になる。

book.f 題名といった記法は、本の集合の要素である book レコードの欄 (フィールド) を示す。

```
public
  本を検索する : ( 題名 | 著者 | 分野) ==> 蔵書
  本を検索する (a 検索キー) ==
    is not yet specified
  pre  検索キーが空でない (a 検索キー)
  post  forall book in set rng RESULT &
           (book.f 題名 = a 検索キー or
            book.f 著者 = a 検索キー or
            a 検索キー in set book.f 分野集合)
```

2.4.4　本を貸す

「本を貸す」を記述するには、型定義とインスタンス変数の定義が必要になる。

まず、貸出という型を、利用者から蔵書への 1 対 1 写像として定義する。蔵書は蔵書 ID から本への写像であったから、貸出は inmap 利用者 to (蔵書 ID to 本)　ということになる。そして、その型のインスタンス変数「i 貸出」を定義する。「i 貸出」に含まれる蔵書は、図書館の蔵書でなければならないので、インスタンス変数の不変条件として「蔵書に存在する」ことを確認しておく。ここでは、「蔵書に存在する」関数を呼び出すことで確認する。

「i 貸出」写像に、写像の値域の集合を取り出す写像型演算子 rng を適用することで、「i 貸出」写像の値域の蔵書の集合（蔵書は写像型なので写像の集合になる）を返すので、その写像の集合に、写像の分配的併合演算子 merge を適用することで、写像の集合に含まれる、すべての写（写像の個々の要素を写と呼ぶ。）を持つ写像を得る。この写像が「i 蔵書」の一部であれば蔵書に存在することになり、「蔵書に存在する」関数がそれを判定する。

このように、VDM++ では、types や instance variables の定義は 1 箇所で定義する必要はなく、必要なときに何回でも定義してよい。他にも、values や operations あるいは functions の定義も、何回行ってもよい。

```
types
  public 貸出 = inmap 利用者 to 蔵書
instance variables
```

```
i 貸出 : 貸出 := { |-> };
```

2.4.5 本を貸す

「本を貸す」操作は職員型の引数を持つが、仕様内部では使わないため、仮引数に'-'(don't-care 記号) というパターンを記述し、仮引数が存在することだけを強調する。職員を使わないのは、現在の仕様が、まだそこまで詳細化する必要がないと判断したためである。

```
operations
public
  本を貸す : 蔵書 * 利用者 * 職員 ==> ()
  本を貸す(a 貸出本, a 利用者, -) ==
    is not yet specified
  pre  貸出可能である(a 利用者, a 貸出本, i 貸出, i 蔵書, v 最大貸出数)
  post 貸出に追加されている(a 貸出本, a 利用者, i 貸出, i 貸出~) ;
```

2.4.6 本を返す

```
public
  本を返す : 蔵書 * 利用者 * 職員 ==> ()
  本を返す(a 返却本, a 利用者, -) ==
    is not yet specified
  pre  貸出に存在する(a 返却本, i 貸出)
  post 貸出から削除されている(a 返却本, i 貸出, i 貸出~)
```

2.5 関数定義

2.5.1 蔵書が追加されている

蔵書が追加されているかを bool 型、すなわち true か false を返すことで判定する関数である。

関数は、操作と異なりインスタンス変数にアクセス出来ない。引数を受け取り、計算結果を返り値として返すだけである。インスタンス変数という大域変数 (グローバル変数) にアクセスしないため、副作用を避ける事ができ、安全性が高い。

引数で渡された図書館蔵書旧値と追加する本「a 追加本」の 2 つの写像を併合したものが、図書館蔵書の蔵書の写像と等しいと true を返す。旧値は、この関数を呼び出した操作が実行する前の値であることを示す。VDM++ では、名前の後ろに'~' を付けると旧値を意味するが、ここでの「a 図書館蔵書旧値」は、呼び出し側の操作の旧値を表し、この関数の旧値を表すわけではないので、'~' 記号は使えない。

```
functions
  蔵書が追加されている : 蔵書 * 蔵書 * 蔵書 +> bool
  蔵書が追加されている (a 追加本, a 図書館蔵書, a 図書館蔵書旧値) ==
    a 図書館蔵書 = a 図書館蔵書旧値 munion a 追加本;
```

2.5.2 蔵書が削除されている

ここで、”<-:”は、定義域削減演算子である。これは、左辺の集合 (定義域) をもつ右辺の写像の要素 (写という) を削除した写像を返す。今の場合、「dom a 削除本」が定義域の蔵書 ID の集合を返すので、要するに「削除する蔵書 ID の集合に対応する写像を削除する」ことになる。

```
  蔵書が削除されている : 蔵書 * 蔵書 * 蔵書 +> bool
  蔵書が削除されている (a 削除本, a 図書館蔵書, a 図書館蔵書旧値) ==
    a 図書館蔵書 = dom a 削除本 <-: a 図書館蔵書旧値;
```

2.5.3 蔵書に存在する

「a 追加本」の定義域集合の要素である蔵書 ID が、a 図書館蔵書の定義域集合の要素であれば、true を返す。

```
  蔵書に存在する : 蔵書 * 蔵書 +> bool
  蔵書に存在する (a 追加本, a 図書館蔵書) ==
     exists id in set dom a 追加本 & id in set dom a 図書館蔵書;
```

2.5.4 蔵書に存在しない

```
  蔵書に存在しない : 蔵書 * 蔵書 +> bool
  蔵書に存在しない (a 追加本, a 図書館蔵書) ==
    not 蔵書に存在する (a 追加本, a 図書館蔵書);
```

2.5.5 貸出に追加されている

```
  貸出に追加されている : 蔵書 * 利用者 * 貸出 * 貸出 +> bool
  貸出に追加されている (a 貸出本, a 利用者, a 貸出, a 貸出旧値) ==
    if a 利用者 in set dom a 貸出
    then a 貸出 = a 貸出旧値 ++ {a 利用者 |-> (a 貸出 (a 利用者) munion a 貸出本)}
    else a 貸出 = a 貸出旧値 munion {a 利用者 |-> a 貸出本 };
```

2.5.6　貸出から削除されている

```
貸出から削除されている : 蔵書 * 貸出 * 貸出 +> bool
貸出から削除されている (a 削除本, a 貸出, a 貸出旧値) ==
  貸出に存在する (a 削除本, a 貸出旧値) and
  貸出に存在しない (a 削除本, a 貸出);
```

2.5.7　貸出に存在する

```
貸出に存在する : 蔵書 * 貸出 +> bool
貸出に存在する (a 貸出本, a 貸出) ==
  let w 蔵書 = merge rng a 貸出 in
   exists id in set dom a 貸出本 & id in set dom w 蔵書;
```

2.5.8　貸出に存在しない

```
貸出に存在しない : 蔵書 * 貸出 +> bool
貸出に存在しない (a 貸出本, a 貸出) ==
  not 貸出に存在する (a 貸出本, a 貸出);
```

2.5.9　貸出可能である

貸出可能であるとは、貸し出し予定の本が蔵書に存在し、貸し出されておらず、利用者の最大貸出数を超えて貸し出されていない、ということを確認している。

```
public
貸出可能である : 利用者 * 蔵書 * 貸出 * 蔵書 * nat1 +> bool
貸出可能である (a 利用者, a 貸出本, a 貸出, a 図書館蔵書, a 最大貸出数) ==
  蔵書に存在する (a 貸出本, a 図書館蔵書) and
  貸出に存在しない (a 貸出本, a 貸出) and
  最大貸出数を超えていない (a 利用者, a 貸出本, a 貸出, a 最大貸出数);
```

2.5.10　貸出可能でない

```
public
```

```
  貸出可能でない : 利用者 * 蔵書 * 貸出 * 蔵書 * nat1 +> bool
  貸出可能でない (a 利用者, a 蔵書, a 貸出, a 図書館蔵書, a 最大貸出数) ==
    not 貸出可能である (a 利用者, a 蔵書, a 貸出, a 図書館蔵書, a 最大貸出数);
```

2.5.11 最大貸出数を超えていない

```
public
  最大貸出数を超えていない : 利用者 * 蔵書 * 貸出 * nat1 +> bool
  最大貸出数を超えていない (a 利用者, a 貸出本, a 貸出, a 最大貸出数) ==
    if a 利用者 not in set dom a 貸出
    then card dom a 貸出本 <= a 最大貸出数
    else card dom a 貸出 (a 利用者) + card dom a 貸出本 <= a 最大貸出数;
```

2.5.12 検索キーが空でない

```
public
  検索キーが空でない : 題名 | 著者 | 分野 +> bool
  検索キーが空でない (a 検索キー) ==
    a 検索キー <> ""
```

```
end
図書館 0
```

3　図書館 0 要求辞書

図書館のドメイン知識を持つ。

図書館に関する要求辞書の役割も持つ。クラス名に付いている RD は、Requirement Dictionary の略である。

```
.....................................................................................
class
図書館 RD0
.....................................................................................
```

3.1　型定義

```
.....................................................................................
types
  public 蔵書 = map 蔵書 ID to 本;
public
  本 :: f 題名 : 題名
       f 著者 : 著者
       f 分野集合 : set of 分野;
  public 題名 = seq of char;
  public 著者 = seq of char;
  public 分野 = seq of char;
  public 蔵書 ID = token;
  public 職員 = token;
  public 利用者 = token;
  public 貸出 = inmap 利用者 to 蔵書
.....................................................................................
```

3.2　関数定義

3.2.1　蔵書の状態に関する関数

```
.....................................................................................
functions
public
  蔵書が追加されている : 蔵書 * 蔵書 * 蔵書 +> bool
  蔵書が追加されている (a 追加本, a 図書館蔵書, a 図書館蔵書旧値) ==
    a 図書館蔵書 = a 図書館蔵書旧値 munion a 追加本;
public
```

```
  蔵書が削除されている : 蔵書 * 蔵書 * 蔵書 +> bool
  蔵書が削除されている (a 削除本, a 図書館蔵書, a 図書館蔵書旧値) ==
    a 図書館蔵書 = dom a 削除本 <-: a 図書館蔵書旧値;
public
  蔵書に存在する : 蔵書 * 蔵書 +> bool
  蔵書に存在する (a 蔵書, a 図書館蔵書) ==
    forall id in set dom a 蔵書 & id in set dom a 図書館蔵書;
public
  蔵書に存在しない : 蔵書 * 蔵書 +> bool
  蔵書に存在しない (a 蔵書, a 図書館蔵書) ==
    not 蔵書に存在する (a 蔵書, a 図書館蔵書);
```

3.2.2 貸出の状態に関する関数

```
public
  貸出に追加されている : 蔵書 * 利用者 * 貸出 * 貸出 +> bool
  貸出に追加されている (a 貸出本, a 利用者, a 貸出, a 貸出旧値) ==
    if a 利用者 in set dom a 貸出
    then a 貸出 = a 貸出旧値 ++ {a 利用者 |-> (a 貸出 (a 利用者) munion a 貸出本)}
    else a 貸出 = a 貸出旧値 munion {a 利用者 |-> a 貸出本 };
public
  貸出から削除されている : 蔵書 * 貸出 * 貸出 +> bool
  貸出から削除されている (a 削除本, a 貸出, a 貸出旧値) ==
    貸出に存在する (a 削除本, a 貸出旧値) and
    貸出に存在しない (a 削除本, a 貸出);
public
  貸出に存在する : 蔵書 * 貸出 +> bool
  貸出に存在する (a 貸出本, a 貸出) ==
    let w 蔵書 = merge rng a 貸出 in
    forall id in set dom a 貸出本 & id in set dom w 蔵書;
public
  貸出に存在しない : 蔵書 * 貸出 +> bool
  貸出に存在しない (a 貸出本, a 貸出) ==
    not 貸出に存在する (a 貸出本, a 貸出);
```

```
public
```

```
  貸出可能である : 利用者 * 蔵書 * 貸出 * 蔵書 * nat1 +> bool
  貸出可能である (a 利用者, a 貸出本, a 貸出, a 図書館蔵書, a 最大貸出数) ==
    蔵書に存在する (a 貸出本, a 図書館蔵書) and
    貸出に存在しない (a 貸出本, a 貸出) and
    最大貸出数を超えていない (a 利用者, a 貸出本, a 貸出, a 最大貸出数);
public
  貸出可能でない : 利用者 * 蔵書 * 貸出 * 蔵書 * nat1 +> bool
  貸出可能でない (a 利用者, a 蔵書, a 貸出, a 図書館蔵書, a 最大貸出数) ==
    not 貸出可能である (a 利用者, a 蔵書, a 貸出, a 図書館蔵書, a 最大貸出数);
```

3.2.3 最大貸出数を超えていない

```
public
  最大貸出数を超えていない : 利用者 * 蔵書 * 貸出 * nat1 +> bool
  最大貸出数を超えていない (a 利用者, a 貸出本, a 貸出, a 最大貸出数) ==
    if a 利用者 not in set dom a 貸出
    then card rng a 貸出本 <= a 最大貸出数
    else card rng a 貸出 (a 利用者) + card rng a 貸出本 <= a 最大貸出数;
```

3.2.4 蔵書の検索に関する関数

```
public
  検索キーが空である : 題名 | 著者 | 分野 +> bool
  検索キーが空である (a 検索キー) ==
    a 検索キー <> ""
```

```
end
```

図書館 RD0

Test Suite : vdm.tc

Class : 図書館 RD0

Name	#Calls	Coverage
図書館 RD0‘蔵書に存在する	**undefined**	**undefined**
図書館 RD0‘貸出に存在する	**undefined**	**undefined**
図書館 RD0‘貸出可能である	**undefined**	**undefined**
図書館 RD0‘貸出可能でない	**undefined**	**undefined**

Name	#Calls	Coverage
図書館 RD0‘蔵書に存在しない	**undefined**	**undefined**
図書館 RD0‘貸出に存在しない	**undefined**	**undefined**
図書館 RD0‘検索キーが空である	**undefined**	**undefined**
図書館 RD0‘蔵書が削除されている	**undefined**	**undefined**
図書館 RD0‘蔵書が追加されている	**undefined**	**undefined**
図書館 RD0‘貸出に追加されている	**undefined**	**undefined**
図書館 RD0‘貸出から削除されている	**undefined**	**undefined**
図書館 RD0‘最大貸出数を超えていない	**undefined**	**undefined**
Total Coverage		**0%**

4　実行可能な仕様 (図書館 1)

実行不可能な仕様で、事後条件・事前条件・不変条件などを整理した時点で、構文チェック・型チェックが可能なので、静的な検証は行うことができる。

実際のシステム開発では、日本語による仕様の欠陥より少ないとはいえ、VDM++ の実行不可能な陰仕様には多数の欠陥が残る。そこで、実行可能な仕様を作成することにする。

図書館システムの例では、下記のように、関数や操作の本体に、集合と写像を扱う演算子や、内包表現を使うことにより、容易に実行可能な陽仕様とすることができる。

関数や操作の本体の VDM++ による記述は、陰仕様で定義された「仕様」を検証するための記述である。したがって、設計時や実装時にこの記述をそのまま使う必要はない。「仕様」は、あくまでも陰仕様で記述された部分である。

ここで、スーパークラスの図書館 RD は、図書館のドメイン知識を持つクラスであり、後述する。

```
class
図書館 1 is subclass of 図書館 RD
values
public
  v 最大蔵書数 = 10000;
public
  v 最大貸出数 = 3
```

4.1　型定義

```
types
  public 蔵書 = map 蔵書 ID to 本
  inv w 蔵書 == card dom w 蔵書 <= v 最大蔵書数
```

4.2　インスタンス変数定義

```
instance variables
  private i 蔵書 : 蔵書 := { |-> };
  private i 貸出 : 貸出 := { |-> };
```

4.3 操作定義

4.3.1 蔵書を追加する

```
operations
public
  蔵書を追加する : 蔵書 ==> ()
  蔵書を追加する (a 追加本) ==
    i 蔵書 := i 蔵書 munion a 追加本
  pre  蔵書に存在しない (a 追加本, i 蔵書)
  post 蔵書が追加されている (a 追加本, i 蔵書, i 蔵書~) ;
```

4.3.2 蔵書を削除する

```
public
  蔵書を削除する : 蔵書 ==> ()
  蔵書を削除する (a 削除本) ==
    i 蔵書 := dom a 削除本 <-: i 蔵書
  pre  蔵書に存在する (a 削除本, i 蔵書) and
       貸出に存在しない (a 削除本, i 貸出)
  post 蔵書が削除されている (a 削除本, i 蔵書, i 蔵書~) ;
```

4.3.3 題名と著者と分野で本を検索する

ここでは、写像の内包式 で題名と著者と分野のいずれかに一致する蔵書を検索結果として返している。内包式は集合や列に対しても形式は少し異なるが存在し、ある条件を満たした写像、集合、列のデータを得るのによく使う。

```
public
  本を検索する : ( 題名 | 著者 | 分野) ==> 蔵書
  本を検索する (a 検索キー) ==
    return {id |-> i 蔵書 (id) | id in set dom i 蔵書 &
                  (i 蔵書 (id).f 題名 = a 検索キー or
                   i 蔵書 (id).f 著者 = a 検索キー or
                   a 検索キー in set i 蔵書 (id).f 分野集合)}
  pre  検索キーが空でない (a 検索キー)
```

```
  post  forall book in set rng RESULT &
          (book.f 題名 = a 検索キー or
           book.f 著者 = a 検索キー or
           a 検索キー in set book.f 分野集合) ;
```

4.3.4 本を貸す

利用者がすでに本を借りている場合と、初めて借りる場合で、処理が異なる。

すでに本を借りているか否かは、「a 利用者 in set dom i 貸出」で判定できる。 インスタンス変数「i 貸出」の定義域集合を得る演算子が dom なので、得られた定義域の利用者集合に含まれているかを in set という集合の演算子を使って判定する。

すでに借りたことがある場合は、すでに借りている本と、新たな貸出本の併合を行って、貸出の写像のうち、利用者の貸出部分を上書き演算子"++" で書き換える。

初めて借りる場合は、単純に利用者と貸出本の写を、従来の貸出写像に munion 演算子を使って併合する。

```
public
  本を貸す : 蔵書 * 利用者 * 職員 ==> ()
  本を貸す (a 貸出本, a 利用者, -) ==
    if a 利用者 in set dom i 貸出
    then i 貸出 := i 貸出 ++ {a 利用者 |-> (i 貸出 (a 利用者) munion a 貸出本)}
    else i 貸出 := i 貸出 munion {a 利用者 |-> a 貸出本 }
  pre  貸出可能である (a 利用者, a 貸出本, i 貸出, i 蔵書, v 最大貸出数)
  post 貸出に追加されている (a 貸出本, a 利用者, i 貸出, i 貸出~) ;
```

4.3.5 本を返す

利用者への貸出本から、定義域削減演算子 <-: を使って返却本に関するデータを削除する。<-: 演算子は、貸出がなくなった利用者の情報を削除するためにも使っている。

```
public
  本を返す : 蔵書 * 利用者 * 職員 ==> ()
  本を返す (a 返却本, a 利用者, -) ==
    let w 利用者への貸出本 = i 貸出 (a 利用者),
        w 利用者への貸出本 new = dom a 返却本 <-: w 利用者への貸出本 in
    if w 利用者への貸出本 new = { |-> }
    then i 貸出 := {a 利用者 } <-: i 貸出
    else i 貸出 := i 貸出 ++ {a 利用者 |-> w 利用者への貸出本 new}
  pre  貸出に存在する (a 返却本, i 貸出)
  post 貸出から削除されている (a 返却本, i 貸出, i 貸出~) ;
```

4.3.6　インスタンス変数のアクセッサ

以下の操作は、主に回帰テストケースを記述するのに便利になるよう作成した。

```
public
  蔵書を得る : () ==> 蔵書
  蔵書を得る () ==
    return i 蔵書;
public
  貸出を得る : () ==> 貸出
  貸出を得る () ==
    return i 貸出
end
```

図書館 1

Test Suite : vdm.tc

Class : 図書館 1

Name	#Calls	Coverage
図書館 1‘本を貸す	13	√
図書館 1‘本を返す	5	√
図書館 1‘蔵書を得る	6	√
図書館 1‘貸出を得る	6	√
図書館 1‘本を検索する	8	√
図書館 1‘蔵書を削除する	2	√
図書館 1‘蔵書を追加する	20	√
Total Coverage		**100%**

5　図書館要求辞書

図書館のドメイン知識を持つ。

図書館に関する要求辞書の役割も持つ。クラス名に付いている RD は、Requirement Dictionary の略である。

実行不可能な陰仕様を記述した図書館 0 クラスで定義されていた、関数群に対応した関数群を持つ。

```
class
図書館 RD
```

5.1　型定義

```
types
  public 蔵書 = map 蔵書 ID to 本;
public
  本 :: f 題名 : 題名
       f 著者 : 著者
       f 分野集合 : set of 分野;
  public 題名 = seq of char;
  public 著者 = seq of char;
  public 分野 = seq of char;
  public 蔵書 ID = token;
  public 職員 = token;
  public 利用者 = token;
  public 貸出 = inmap 利用者 to 蔵書
```

5.2　関数定義

5.2.1　蔵書の状態に関する関数

```
functions
public
  蔵書が追加されている : 蔵書 * 蔵書 * 蔵書 +> bool
  蔵書が追加されている (a 追加本, a 図書館蔵書, a 図書館蔵書旧値) ==
    a 図書館蔵書 = a 図書館蔵書旧値 munion a 追加本;
public
```

```
  蔵書が削除されている : 蔵書 * 蔵書 * 蔵書 +> bool
  蔵書が削除されている (a 削除本, a 図書館蔵書, a 図書館蔵書旧値) ==
    a 図書館蔵書 = dom a 削除本 <-: a 図書館蔵書旧値;
public
  蔵書に存在する : 蔵書 * 蔵書 +> bool
  蔵書に存在する (a 蔵書, a 図書館蔵書) ==
    exists id in set dom a 蔵書 & id in set dom a 図書館蔵書;
public
  蔵書に存在しない : 蔵書 * 蔵書 +> bool
  蔵書に存在しない (a 蔵書, a 図書館蔵書) ==
    not 蔵書に存在する (a 蔵書, a 図書館蔵書);
```

5.2.2 貸出の状態に関する関数

```
public
  貸出に追加されている : 蔵書 * 利用者 * 貸出 * 貸出 +> bool
  貸出に追加されている (a 貸出本, a 利用者, a 貸出, a 貸出旧値) ==
    if a 利用者 in set dom a 貸出
    then a 貸出 = a 貸出旧値 ++ {a 利用者 |-> (a 貸出 (a 利用者) munion a 貸出本)}
    else a 貸出 = a 貸出旧値 munion {a 利用者 |-> a 貸出本 };
public
  貸出から削除されている : 蔵書 * 貸出 * 貸出 +> bool
  貸出から削除されている (a 削除本, a 貸出, a 貸出旧値) ==
    貸出に存在する (a 削除本, a 貸出旧値) and
    貸出に存在しない (a 削除本, a 貸出);
public
  貸出に存在する : 蔵書 * 貸出 +> bool
  貸出に存在する (a 貸出本, a 貸出) ==
    let w 蔵書 = merge rng a 貸出 in
    exists id in set dom a 貸出本 & id in set dom w 蔵書;
public
  貸出に存在しない : 蔵書 * 貸出 +> bool
  貸出に存在しない (a 貸出本, a 貸出) ==
    not 貸出に存在する (a 貸出本, a 貸出);
public
```

```
  貸出可能である : 利用者 * 蔵書 * 貸出 * 蔵書 * nat1 +> bool
  貸出可能である (a 利用者, a 貸出本, a 貸出, a 図書館蔵書, a 最大貸出数) ==
    蔵書に存在する (a 貸出本, a 図書館蔵書) and
    貸出に存在しない (a 貸出本, a 貸出) and
    最大貸出数を超えていない (a 利用者, a 貸出本, a 貸出, a 最大貸出数);
public
  貸出可能でない : 利用者 * 蔵書 * 貸出 * 蔵書 * nat1 +> bool
  貸出可能でない (a 利用者, a 蔵書, a 貸出, a 図書館蔵書, a 最大貸出数) ==
    not 貸出可能である (a 利用者, a 蔵書, a 貸出, a 図書館蔵書, a 最大貸出数);
```

5.2.3 最大貸出数を超えていない

```
public
  最大貸出数を超えていない : 利用者 * 蔵書 * 貸出 * nat1 +> bool
  最大貸出数を超えていない (a 利用者, a 貸出本, a 貸出, a 最大貸出数) ==
    if a 利用者 not in set dom a 貸出
    then card dom a 貸出本 <= a 最大貸出数
    else card dom a 貸出 (a 利用者) + card dom a 貸出本 <= a 最大貸出数;
```

5.2.4 蔵書の検索に関する関数

```
public
  検索キーが空でない : 題名 | 著者 | 分野 +> bool
  検索キーが空でない (a 検索キー) ==
    a 検索キー <> ""
```

```
end
```

図書館 RD

Test Suite : vdm.tc

Class : 図書館 RD

Name	#Calls	Coverage
図書館 RD‘蔵書に存在する	36	√
図書館 RD‘貸出に存在する	29	√
図書館 RD‘貸出可能である	14	√
図書館 RD‘貸出可能でない	1	√

Name	**#Calls**	**Coverage**
図書館 RD‘蔵書に存在しない	20	√
図書館 RD‘貸出に存在しない	20	√
図書館 RD‘検索キーが空でない	8	√
図書館 RD‘蔵書が削除されている	1	√
図書館 RD‘蔵書が追加されている	19	√
図書館 RD‘貸出に追加されている	12	70%
図書館 RD‘貸出から削除されている	4	√
図書館 RD‘最大貸出数を超えていない	13	√
Total Coverage		**94%**

6　図書館 1 の回帰テスト

6.1　TestApp

6.1.1　責任

回帰テストを行う。

6.1.2　クラス定義

```
class
TestApp
```

6.1.3　操作：run

回帰テストケースを TestSuite に追加し、実行し、成功したか判定する。

```
operations
public static
 run : () ==> ()
 run () ==
   (   dcl ts : TestSuite := new TestSuite ("図書館 1 : 図書館貸し出しモデルの回帰テスト \n"),
           tr : TestResult := new TestResult ();
       tr.addListener(new PrintTestListener ());
       ts.addTest(new TestCaseUT0001 ("TestCaseUT0001 : \t ノーマルケースの単体テスト \n"));
       ts.addTest(new TestCaseUT0002 ("TestCaseUT0002 : \t エラーケースの単体テスト \n"));
       ts.run(tr);
       if tr.wasSuccessful () = true
       then def - = new IO ().echo ("*** 全回帰テストケース成功。 ***") in
            skip
       else def - = new IO ().echo ("*** 失敗したテストケースあり!! ***") in
            skip
   )
end
TestApp
```

Test Suite :　vdm.tc

Class :　TestApp

Name	**#Calls**	**Coverage**
TestApp‘run	1	82%
Total Coverage		**82%**

6.2 TestCaseComm

6.2.1 責任

回帰テスト共通機能を記述する。

```
class
TestCaseComm is subclass of TestCase
operations
public
  print : seq of char ==> ()
  print(a 文字列) ==
    let - = new IO().echo(a 文字列) in
    skip
end
TestCaseComm
```

6.3 TestCaseUT0001

6.3.1 責任

図書館のノーマルケースの単体テストを行う。

```
class
TestCaseUT0001 is subclass of TestCaseComm
operations
public
  TestCaseUT0001 : seq of char ==> TestCaseUT0001
  TestCaseUT0001(name) ==
    setName(name);
public
  test01 : () ==> ()
  test01() ==
    (   let w図書館 = new 図書館1(),
            分野1 = "ソフトウェア",
            分野2 = "工学",
            分野3 = "小説",
            著者1 = "佐原伸",
            著者2 = "井上ひさし",
            本1 = mk_図書館1`本("デザインパターン", 著者1, {分野1, 分野2}),
```

```
        本2 = mk_図書館1`本("紙屋町桜ホテル", 著者2, {分野3}),
        蔵書1 = {mk_token(101) |-> 本1},
        蔵書2 = {mk_token(102) |-> 本1},
        蔵書3 = {mk_token(103) |-> 本2},
        蔵書4 = {mk_token(104) |-> 本2},
        利用者1 = mk_token("利用者1"),
        利用者2 = mk_token("利用者2"),
        職員1 = mk_token("職員1") in
    (   w図書館.蔵書を追加する(蔵書1);
        w図書館.蔵書を追加する(蔵書2);
        w図書館.蔵書を追加する(蔵書3);
        w図書館.蔵書を追加する(蔵書4);
        assertTrue("test01 蔵書の追加がおかしい。",
            w図書館.蔵書を得る() = {mk_token(101) |-> 本1,mk_token(102) |->
本1,mk_token(103) |-> 本2,mk_token(104) |-> 本2});
        let w見つかった本1 = w図書館.本を検索する("井上ひさし"),
            w見つかった本2 = w図書館.本を検索する("工学") in
        assertTrue("test01 本の検索がおかしい。",
            w見つかった本1 = {mk_token(103) |-> 本2,mk_token(104) |-> 本2} and
            w見つかった本2 = {mk_token(101) |-> 本1,mk_token(102) |-> 本1});
        w図書館.本を貸す({mk_token(101) |-> 本1},利用者1,職員1);
        w図書館.本を貸す({mk_token(103) |-> 本2},利用者1,職員1);
        w図書館.本を貸す({mk_token(104) |-> 本2},利用者2,職員1);
        assertTrue("test01 本の貸出がおかしい。",
            let w貸出 = w図書館.貸出を得る() in
            w貸出 = {利用者1 |-> {mk_token(101) |-> 本1,mk_token(103) |->
本2},利用者2 |-> {mk_token(104) |-> 本2}});
        w図書館.本を返す({mk_token(103) |-> 本2},利用者1,職員1);
        w図書館.本を返す({mk_token(104) |-> 本2},利用者2,職員1);
        assertTrue("test01 本の返却がおかしい。",
            let w貸出 = w図書館.貸出を得る() in
            w貸出 = {利用者1 |-> {mk_token(101) |-> 本1}});
        w図書館.蔵書を削除する({mk_token(104) |-> 本2});
        assertTrue("test01 蔵書の削除がおかしい。",
            w図書館.蔵書を得る() = {mk_token(101) |-> 本1,mk_token(102) |->
本1,mk_token(103) |-> 本2})
    )
)
end
TestCaseUT0001
```

6.4 TestCaseUT0002

6.4.1 責任

図書館のエラーケースの単体テストを行う。

```
class
TestCaseUT0002 is subclass of TestCaseComm
operations
public
  TestCaseUT0002 : seq of char ==> TestCaseUT0002
  TestCaseUT0002(name) ==
    setName(name);
public
  test01 : () ==> ()
  test01() ==
    (   trap <RuntimeError>
        with  print("\ttest01 意図通り、最大貸出数を超えて貸し出そうとするエラー発生。\n") in
        let w図書館 = new 図書館1(),
            分野1 = "ソフトウェア",
            分野2 = "工学",
            分野3 = "小説",
            著者1 = "佐原伸",
            著者2 = "井上ひさし",
            本1 = mk_図書館1`本("デザインパターン", 著者1, {分野1, 分野2}),
            本2 = mk_図書館1`本("紙屋町桜ホテル", 著者2, {分野3}),
            蔵書1 = {mk_token(101) |-> 本1},
            蔵書2 = {mk_token(102) |-> 本1},
            蔵書3 = {mk_token(103) |-> 本2},
            蔵書4 = {mk_token(104) |-> 本2},
            利用者1 = mk_token("利用者1"),
```

```
            職員 1 = mk_token("職員 1") in
        (   w 図書館.蔵書を追加する(蔵書 1);
            w 図書館.蔵書を追加する(蔵書 2);
            w 図書館.蔵書を追加する(蔵書 3);
            w 図書館.蔵書を追加する(蔵書 4);
            assertTrue("test01 蔵書の追加がおかしい。",
                 w 図書館.蔵書を得る() = {mk_token(101) |-> 本 1,mk_token(102) |->
本 1,mk_token(103) |-> 本 2,mk_token(104) |-> 本 2});
            let w 見つかった本 1 = w 図書館.本を検索する("井上ひさし"),
                w 見つかった本 2 = w 図書館.本を検索する("工学") in
            assertTrue("test01 本の検索がおかしい。",
                 w 見つかった本 1 = {mk_token(103) |-> 本 2,mk_token(104) |-> 本 2} and
                 w 見つかった本 2 = {mk_token(101) |-> 本 1,mk_token(102) |-> 本 1});
            w 図書館.本を貸す({mk_token(101) |-> 本 1},利用者 1,職員 1);
            w 図書館.本を貸す({mk_token(102) |-> 本 1},利用者 1,職員 1);
            w 図書館.本を貸す({mk_token(103) |-> 本 2},利用者 1,職員 1);
            w 図書館.本を貸す({mk_token(104) |-> 本 2},利用者 1,職員 1);
            assertTrue("test01 本の貸出がおかしい。",
                 let w 貸出 = w 図書館.貸出を得る() in
                 w 貸出 = { |-> })
        )
    );
public
  test02 : () ==> ()
  test02() ==
    (   trap <RuntimeError>
        with  print("\ttest02 意図通り、貸出本を削除しようとするエラー発生。\n") in
        let w 図書館 = new 図書館 1(),
            分野 1 = "ソフトウェア",
            分野 2 = "工学",
            分野 3 = "小説",
            著者 1 = "佐原伸",
            著者 2 = "井上ひさし",
            本 1 = mk_図書館 1`本("デザインパターン",著者 1,{分野 1,分野 2}),
            本 2 = mk_図書館 1`本("紙屋町桜ホテル",著者 2,{分野 3}),
            蔵書 1 = {mk_token(101) |-> 本 1},
            蔵書 2 = {mk_token(102) |-> 本 1},
            蔵書 3 = {mk_token(103) |-> 本 2},
            蔵書 4 = {mk_token(104) |-> 本 2},
            利用者 1 = mk_token("利用者 1"),
```

```
            利用者 2 = mk_token ("利用者 2"),
            職員 1 = mk_token ("職員 1") in
        (   w 図書館.蔵書を追加する ( 蔵書 1) ;
            w 図書館.蔵書を追加する ( 蔵書 2) ;
            w 図書館.蔵書を追加する ( 蔵書 3) ;
            w 図書館.蔵書を追加する ( 蔵書 4) ;
            assertTrue("test02 蔵書の追加がおかしい。",
                w 図書館.蔵書を得る () = {mk_token (101) |-> 本 1,mk_token (102) |->
本 1,mk_token (103) |-> 本 2,mk_token (104) |-> 本 2}) ;
            let w 見つかった本 1 = w 図書館.本を検索する ("井上ひさし"),
                w 見つかった本 2 = w 図書館.本を検索する ("工学") in
            assertTrue("test02 本の検索がおかしい。",
                w 見つかった本 1 = {mk_token (103) |-> 本 2,mk_token (104) |-> 本 2} and
                w 見つかった本 2 = {mk_token (101) |-> 本 1,mk_token (102) |-> 本 1}) ;
            w 図書館.本を貸す ({mk_token (101) |-> 本 1}, 利用者 1, 職員 1) ;
            w 図書館.本を貸す ({mk_token (103) |-> 本 2}, 利用者 1, 職員 1) ;
            w 図書館.本を貸す ({mk_token (104) |-> 本 2}, 利用者 2, 職員 1) ;
            assertTrue("test02 本の貸出がおかしい。",
                let w 貸出 = w 図書館.貸出を得る () in
                w 貸出 = { 利用者 1 |-> {mk_token (101) |-> 本 1,mk_token (103) |->
本 2}, 利用者 2 |-> {mk_token (104) |-> 本 2}}) ;
            w 図書館.本を返す ({mk_token (103) |-> 本 2}, 利用者 1, 職員 1) ;
            assertTrue("test02 本の返却がおかしい。",
                let w 貸出 = w 図書館.貸出を得る () in
                w 貸出 = { 利用者 1 |-> {mk_token (101) |-> 本 1}, 利用者 2 |-> {mk_token (104) |->
本 2}}) ;
            w 図書館.蔵書を削除する ({mk_token (104) |-> 本 2}) ;
            assertTrue("test02 蔵書の削除がおかしい。",
                w 図書館.蔵書を得る () = {mk_token (101) |-> 本 1,mk_token (102) |->
本 1,mk_token (103) |-> 本 2})
        )
    ) ;
public
  test03 : () ==> ()
  test03 () ==
    (   trap <RuntimeError>
        with  print("\ttest03 意図通り、貸出に存在しない本を返そうとするエラー発生。\n") in
        let w 図書館 = new 図書館 1 (),
            分野 1 = "ソフトウェア",
            分野 2 = "工学",
```

```
    分野 3 = "小説",
    著者 1 = "佐原伸",
    著者 2 = "井上ひさし",
    本 1 = mk_図書館 1‘本 ("デザインパターン", 著者 1, { 分野 1, 分野 2}),
    本 2 = mk_図書館 1‘本 ("紙屋町桜ホテル", 著者 2, { 分野 3}),
    蔵書 1 = {mk_token (101) |-> 本 1},
    蔵書 2 = {mk_token (102) |-> 本 1},
    蔵書 3 = {mk_token (103) |-> 本 2},
    蔵書 4 = {mk_token (104) |-> 本 2},
    利用者 1 = mk_token ("利用者 1"),
    利用者 2 = mk_token ("利用者 2"),
    利用者 3 = mk_token ("利用者 3"),
    職員 1 = mk_token ("職員 1") in
(   dcl w 貸出 : 図書館 1‘貸出 := { |-> };
    w 図書館.蔵書を追加する ( 蔵書 1) ;
    w 図書館.蔵書を追加する ( 蔵書 2) ;
    w 図書館.蔵書を追加する ( 蔵書 3) ;
    w 図書館.蔵書を追加する ( 蔵書 4) ;
    assertTrue("test03 蔵書の追加がおかしい。",
          w 図書館.蔵書を得る () = {mk_token (101) |-> 本 1, mk_token (102) |-> 本 1,
           mk_token (103) |-> 本 2, mk_token (104) |-> 本 2}) ;
    let w 見つかった本 1 = w 図書館.本を検索する ("井上ひさし"),
```

```
        w 見つかった本 2 = w 図書館.本を検索する ("工学") in
    assertTrue("test03 本の検索がおかしい。",
        w 見つかった本 1 = {mk_token (103) |-> 本 2, mk_token (104) |-> 本 2} and
        w 見つかった本 2 = {mk_token (101) |-> 本 1, mk_token (102) |-> 本 1}) ;
    w 図書館.本を貸す ({mk_token (101) |-> 本 1}, 利用者 1, 職員 1) ;
    w 図書館.本を貸す ({mk_token (103) |-> 本 2}, 利用者 1, 職員 1) ;
    w 図書館.本を貸す ({mk_token (104) |-> 本 2}, 利用者 2, 職員 1) ;
    w 貸出 := w 図書館.貸出を得る ();
    assertTrue("test03 本の貸出がおかしい。",
        w 貸出 = { 利用者 1 |-> {mk_token (101) |-> 本 1, mk_token (103) |-> 本 2},
         利用者 2 |-> {mk_token (104) |-> 本 2}}) ;
    assertTrue("test03 意図に反して、貸出が可能になっている。",
        let w 貸出 = w 図書館.貸出を得る (),
            w 蔵書 = w 図書館.蔵書を得る () in
        w 図書館.貸出可能でない ( 利用者 1, 蔵書 1, w 貸出, w 蔵書, w 図書館.v 最大蔵書数)) ;
    w 図書館.本を返す ({mk_token (103) |-> 本 2}, 利用者 1, 職員 1) ;
    w 図書館.本を返す ({mk_token (105) |-> 本 2}, 利用者 2, 職員 1) ;
    assertTrue("test03 本の返却がおかしい。",
        let w 貸出 = w 図書館.貸出を得る () in
        w 貸出 = { 利用者 1 |-> {mk_token (101) |-> 本 1}}) ;
    w 図書館.蔵書を削除する ({mk_token (104) |-> 本 2}) ;
    assertTrue("test03 蔵書の削除がおかしい。",
        w 図書館.蔵書を得る () = {mk_token (101) |-> 本 1, mk_token (102) |->
本 1, mk_token (103) |-> 本 2}) ;
    assertFalse("test03 意図に反して、利用者が存在している。",
        let w 貸出 new = w 図書館.貸出を得る () in
        w 図書館.貸出に追加されている ({mk_token (104) |-> 本 2}, 利用者 3, w 貸
出 new, w 貸出))
    )
  ) ;
public
  test04 : () ==> ()
  test04 () ==
    ( trap <RuntimeError>
      with print("\ttest04 意図通り、すでに存在する蔵書に追加しようとするエラー発生。\n") in
      let w 図書館 = new 図書館 1 (),
          分野 1 = "ソフトウェア",
          分野 2 = "工学",
          分野 3 = "小説",
          著者 1 = "佐原伸",
```

```
        著者2 = "井上ひさし",
        本1 = mk_図書館1`本("デザインパターン", 著者1, {分野1, 分野2}),
        本2 = mk_図書館1`本("紙屋町桜ホテル", 著者2, {分野3}),
        蔵書1 = {mk_token(101) |-> 本1},
        蔵書2 = {mk_token(102) |-> 本1},
        蔵書3 = {mk_token(103) |-> 本2},
        蔵書4 = {mk_token(101) |-> 本1, mk_token(104) |-> 本2} in
    (   w図書館.蔵書を追加する(蔵書1);
        w図書館.蔵書を追加する(蔵書2);
        w図書館.蔵書を追加する(蔵書3);
        w図書館.蔵書を追加する(蔵書4);
        assertTrue("test04 蔵書の追加がおかしい。",
            w図書館.蔵書を得る() = {mk_token(101) |-> 本1, mk_token(102) |-> 本1,
             mk_token(103) |-> 本2, mk_token(104) |-> 本2})
    )
)
end
TestCaseUT0002
.....................................................................................
```

7　オブジェクト指向モデル

前節までの VDM++ によるモデルは、class を定義しているとはいえ、オブジェクト指向モデルとは言い難かった。

以下では、同じ例題をオブジェクト指向モデル化していく。

本や書庫や貸出は、クラス図 1 で見るようにクラスとなり、図書館 2 クラスの外で定義する。題名を得る ()、著者を得る ()、分野集合を得る () といった操作を本クラスで定義し、蔵書を追加する () や蔵書を削除する () といった操作は書庫クラスで定義し、本を貸す () や本を返す () 操作は貸出クラスで定義する。図書館 2 クラスは、書庫と貸出クラスをスーパークラスとしてその操作を継承し、蔵書を追加する () や本を貸す () といった操作を使用する。

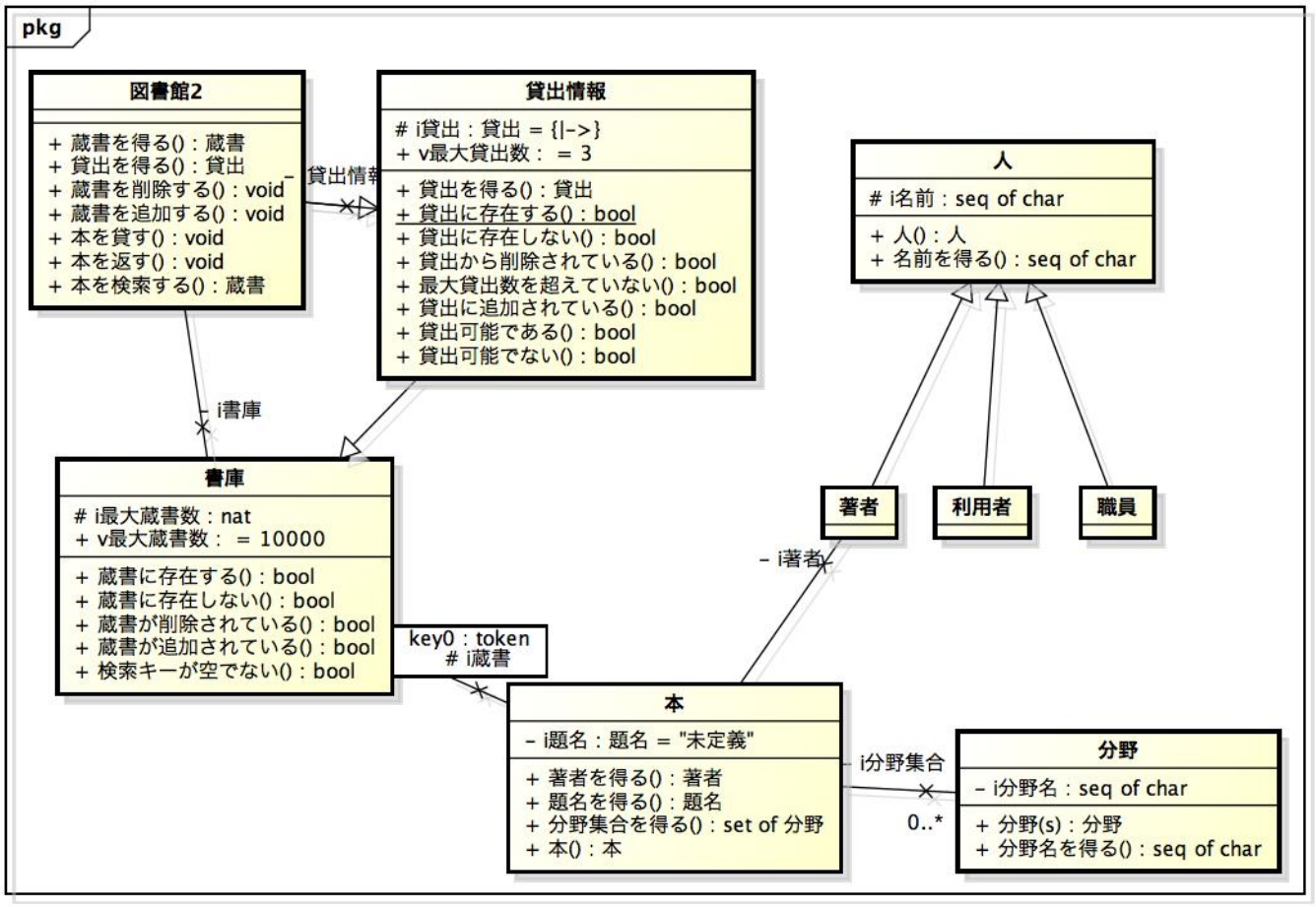

図 1　図書館 2 モデルのクラス図

今までの図書館クラス仕様は、本の内部構造を知らなければ書けない部分があったが、今回の仕様は本クラスの操作名と機能だけを知っていれば書くことができる。本クラスが変更されても、インタフェースさえ変わらなければ、図書館 2 クラスを変更する必要が無くなった。オブジェクト指向により保守性が向上したことになる。

ただし、オブジェクト指向的でない VDM++ モデルに比べ、オブジェクト指向的モデルは 3 倍ほど行数が増える傾向がある。また、どう実現するかを含んだ設計仕様に近い内容を持つことになる。

モデルの全体を見通すには非オブジェクト指向的モデルの方が優れているし、再利用性や保守性の必要な本格的モデルはオブジェクト指向的モデルの方が優れているということになろう。

8 図書館 2

```
class
図書館２ is subclass of 貸出情報
instance variables
  private i 書庫： 書庫 := new 書庫();
  private i 貸出情報： 貸出情報 := new 貸出情報();
```

8.1 操作定義

8.1.1 インスタンス変数のアクセッサ

```
operations
public
  蔵書を得る：() ==> 蔵書
  蔵書を得る() ==
    return i 蔵書;
public
  貸出を得る：() ==> 貸出
  貸出を得る() ==
    return i 貸出;
```

8.1.2 蔵書を追加する

```
public
  蔵書を追加する：蔵書 ==> ()
  蔵書を追加する(a 追加本) ==
    i 蔵書 := i 蔵書 munion a 追加本
  pre 蔵書に存在しない(a 追加本, i 蔵書)
  post 蔵書が追加されている(a 追加本, i 蔵書, i 蔵書~);
```

8.1.3 蔵書を削除する

```
public
  蔵書を削除する : 蔵書 ==> ()
  蔵書を削除する (a 削除本) ==
    i 蔵書 := dom a 削除本 <-: i 蔵書
  pre 蔵書に存在する (a 削除本, i 蔵書) and
      貸出に存在しない (a 削除本, i 貸出)
  post 蔵書が削除されている (a 削除本, i 蔵書, i 蔵書~) ;
```

8.1.4 本を検索する

```
public
  本を検索する : ( 本 `題名 | 著者 | 分野) ==> 蔵書
  本を検索する (a 検索キー) ==
    return {id |-> i 蔵書 (id) | id in set dom i 蔵書 &
                    (i 蔵書 (id).題名を得る () = a 検索キー or
                     i 蔵書 (id).著者を得る ().名前を得る () = a 検索キー or
                     a 検索キー in set {f.分野名を得る () | f in set i 蔵書 (id).分野集合を得る ()})}

  pre 検索キーが空でない (a 検索キー)
  post RESULT =
       {id |-> i 蔵書 (id) | id in set dom i 蔵書 &
               (i 蔵書 (id).題名を得る () = a 検索キー or
                i 蔵書 (id).著者を得る ().名前を得る () = a 検索キー or
                a 検索キー in set {f.分野名を得る () | f in set i 蔵書 (id).分野集合を得る ()})}
;
```

8.1.5 本を貸す

```
public
  本を貸す : 書庫 `蔵書 * 利用者 * 職員 ==> ()
  本を貸す (a 貸出本, a 利用者, -) ==
    if a 利用者 in set dom i 貸出
    then i 貸出 := i 貸出 ++ {a 利用者 |-> (i 貸出 (a 利用者) munion a 貸出本)}
    else i 貸出 := i 貸出 munion {a 利用者 |-> a 貸出本 }
  pre 貸出可能である (a 利用者, a 貸出本, i 貸出, i 蔵書, v 最大貸出数)
```

```
  post 貸出に追加されている(a 貸出本, a 利用者, i 貸出, i 貸出~);
```

8.1.6 本を返す

利用者への貸出本から、定義域削減演算子 <-: を使って返却本に関するデータを削除する。<-: 演算子は、貸出がなくなった利用者の情報を削除するためにも使っている。

```
public
  本を返す : 書庫 '蔵書 * 利用者 * 職員 ==> ()
  本を返す(a 返却本, a 利用者, -) ==
    let w 利用者への貸出本 = i 貸出(a 利用者),
        w 利用者への貸出本 new = dom a 返却本 <-: w 利用者への貸出本 in
    if w 利用者への貸出本 new = { |-> }
    then i 貸出 := {a 利用者} <-: i 貸出
    else i 貸出 := i 貸出 ++ {a 利用者 |-> w 利用者への貸出本 new}
  pre 貸出に存在する(a 返却本, i 貸出)
  post 貸出から削除されている(a 返却本, i 貸出, i 貸出~)
end
```

図書館 2

Test Suite : vdm.tc

Class : 図書館 2

Name	#Calls	Coverage
図書館 2'本を貸す	10	√
図書館 2'本を返す	2	√
図書館 2'蔵書を得る	5	√
図書館 2'貸出を得る	6	√
図書館 2'本を検索する	6	√
図書館 2'蔵書を削除する	2	√
図書館 2'蔵書を追加する	12	√
Total Coverage		**100%**

9 著者

著者を表す。

```
class
著者 is subclass of 人
operations
public
  著者 : seq of char ==> 著者
  著者(s) ==
    i 名前 := s;
public
  著者名を得る : () ==> seq of char
  著者名を得る() ==
    return i 名前
end
著者
```

Test Suite : vdm.tc

Class : 著者

Name	#Calls	Coverage
著者 ‘著者	6	√
著者 ‘著者名を得る	4	√
Total Coverage		**100%**

10　本

今までレコード型で定義していた本は、「本」クラスのインスタンス変数である著者と題名として定義した。

クラス名と同名の操作「本」は、構成子と呼ばれる特殊な操作で、new 式で呼び出すことで本のインスタンスを定義する。返値はクラス名でなければならないので、self 式を使って自分自身 (このクラスのインスタンス) を返すようにしている。

構成子の中で、多重代入文 atomic　を使用している。多重代入文は、指定されている文が、一つの文であるかのように評価・実行される[*3]。したがって、一連の文の評価・実行に副作用が無い[*4]ことを期待する場合は、多重代入文を用いる。インスタンス変数に不変条件が指定されている場合、2 つ以上のインスタンス変数に関わる不変条件があると、片方のインスタンス変数に値を設定した瞬間、不変条件を満たさなくなることがある。このような、事態を避けるため、構成子の定義では通常多重代入文を用いる。

インスタンス変数の値を参照したり、値を設定する操作を用意するのが、オブジェクト指向の定石であるので、ここでは、本の題名を得る () 操作を定義した。

```
class
本
types
 public 題名 = seq of char
instance variables
 private i 題名 : 題名 := "未定義";
 private i 著者 : 著者 := new 著者();
 private i 分野集合 : set of 分野 := {};

operations
public
 本 : 著者 * 題名 * set of 分野 ==> 本
 本(a 著者, a 題名, a 分野集合) ==
   (  i 著者 := a 著者;
      i 題名 := a 題名;
      i 分野集合 := a 分野集合
   );
public
 題名を得る : () ==> 題名
 題名を得る() ==
   return i 題名;
public
```

[*3] 原子的であるという。

[*4] 実行順序によって結果が変わることがない。

```
  著者を得る : () ==> 著者
  著者を得る () ==
    return i 著者;
public
  分野集合を得る : () ==> set of 分野
  分野集合を得る () ==
    return i 分野集合
end
```
本

11　書庫

書庫を表す。

```
class
書庫
values
public
  v 最大蔵書数 = 10000
types
  public 蔵書 ID = token;
  public 蔵書 = map 蔵書 ID to 本
  inv w 蔵書 == card dom w 蔵書 <= v 最大蔵書数
instance variables
  protected i 最大蔵書数 : nat;
  protected i 蔵書 : 蔵書 := { |-> };
```

11.1　蔵書の状態に関する関数

```
functions
public
  蔵書が追加されている : 蔵書 * 蔵書 * 蔵書 +> bool
  蔵書が追加されている (a 追加本, a 図書館蔵書, a 図書館蔵書旧値) ==
    a 図書館蔵書 = a 図書館蔵書旧値 munion a 追加本;
public
  蔵書が削除されている : 蔵書 * 蔵書 * 蔵書 +> bool
  蔵書が削除されている (a 削除本, a 図書館蔵書, a 図書館蔵書旧値) ==
    a 図書館蔵書 = dom a 削除本 <-: a 図書館蔵書旧値;
public
  蔵書に存在する : 蔵書 * 蔵書 +> bool
  蔵書に存在する (a 蔵書, a 図書館蔵書) ==
    forall id in set dom a 蔵書 & id in set dom a 図書館蔵書;
public
  蔵書に存在しない : 蔵書 * 蔵書 +> bool
  蔵書に存在しない (a 蔵書, a 図書館蔵書) ==
    not 蔵書に存在する (a 蔵書, a 図書館蔵書);
```

11.1.1 蔵書の検索に関する関数

```
public
  検索キーが空でない : 本 ‘題名 | 著者 | 分野 +> bool
  検索キーが空でない (a 検索キー) ==
    a 検索キー <> ""
end
```

書庫

Test Suite : vdm.tc

Class : 書庫

Name	#Calls	Coverage
書庫 ‘蔵書に存在する	55	√
書庫 ‘蔵書に存在しない	12	√
書庫 ‘検索キーが空でない	6	√
書庫 ‘蔵書が削除されている	1	√
書庫 ‘蔵書が追加されている	12	√
Total Coverage		**100%**

12　分野

分野を表す。

```
class
分野
instance variables
  private i 分野名 : seq of char;

operations
public
  分野 : seq of char ==> 分野
  分野(s) ==
    i 分野名 := s;
public
  分野名を得る : () ==> seq of char
  分野名を得る() ==
    return i 分野名
end
分野
```

Test Suite : vdm.tc

Class : 分野

Name	#Calls	Coverage
分野`分野	9	√
分野`分野名を得る	63	√
Total Coverage		**100%**

13　貸出情報

貸出情報を持つ。

```
class
貸出情報 is subclass of 書庫
values
public
  v 最大貸出数 = 3
types
  public 貸出 = map 利用者 to 書庫‘蔵書
instance variables
  protected i 貸出 : 貸出 := { |-> };
  inv 蔵書に存在する (merge rng i 貸出, i 蔵書)

operations
public
  貸出を得る : () ==> 貸出
  貸出を得る () ==
    return i 貸出
```

13.1　貸出の状態に関する関数

```
functions
public
  貸出に追加されている : 書庫‘蔵書 * 利用者 * 貸出 * 貸出 +> bool
  貸出に追加されている (a 貸出本, a 利用者, a 貸出, a 貸出旧値) ==
    if a 利用者 in set dom a 貸出
    then a 貸出 = a 貸出旧値 ++ {a 利用者 |-> (a 貸出 (a 利用者) munion a 貸出本)}
    else a 貸出 = a 貸出旧値 munion {a 利用者 |-> a 貸出本 };
public
  貸出から削除されている : 書庫‘蔵書 * 貸出 * 貸出 +> bool
  貸出から削除されている (a 削除本, a 貸出, a 貸出旧値) ==
    貸出に存在する (a 削除本, a 貸出旧値) and
    貸出に存在しない (a 削除本, a 貸出);
public static
```

```
  貸出に存在する : 書庫`蔵書 * 貸出 +> bool
  貸出に存在する(a 貸出本, a 貸出) ==
    let w 蔵書 = merge rng a 貸出 in
     forall id in set dom a 貸出本 & id in set dom w 蔵書;
public
  貸出に存在しない : 書庫`蔵書 * 貸出 +> bool
  貸出に存在しない(a 貸出本, a 貸出) ==
    not 貸出に存在する(a 貸出本, a 貸出);
.....
.....
public
  貸出可能である : 利用者 * 書庫`蔵書 * 貸出 * 書庫`蔵書 * nat1 +> bool
  貸出可能である(a 利用者, a 貸出本, a 貸出, a 図書館蔵書, a 最大貸出数) ==
    蔵書に存在する(a 貸出本, a 図書館蔵書) and
    貸出に存在しない(a 貸出本, a 貸出) and
    最大貸出数を超えていない(a 利用者, a 貸出本, a 貸出, a 最大貸出数);
public
  貸出可能でない : 利用者 * 書庫`蔵書 * 貸出 * 書庫`蔵書 * nat1 +> bool
  貸出可能でない(a 利用者, a 蔵書, a 貸出, a 図書館蔵書, a 最大貸出数) ==
    not 貸出可能である(a 利用者, a 蔵書, a 貸出, a 図書館蔵書, a 最大貸出数);
.....
```

13.1.1 最大貸出数を超えていない

```
.....
public
  最大貸出数を超えていない : 利用者 * 書庫`蔵書 * 貸出 * nat1 +> bool
  最大貸出数を超えていない(a 利用者, a 貸出本, a 貸出, a 最大貸出数) ==
    if a 利用者 not in set dom a 貸出
    then card dom a 貸出本 <= a 最大貸出数
    else card dom a 貸出(a 利用者) + card dom a 貸出本 <= a 最大貸出数
.....
.....
end
```

貸出情報

```
.....
```

Test Suite : vdm.tc

Class : 貸出情報

Name	#Calls	Coverage
貸出情報`貸出を得る	0	0%

Name	#Calls	Coverage
貸出情報 ‘貸出に存在する	19	√
貸出情報 ‘貸出可能である	11	√
貸出情報 ‘貸出可能でない	1	√
貸出情報 ‘貸出に存在しない	15	√
貸出情報 ‘貸出に追加されている	10	√
貸出情報 ‘貸出から削除されている	2	√
貸出情報 ‘最大貸出数を超えていない	10	√
Total Coverage		**97%**

14 人

人を表す。

```
class
人
instance variables
  protected i 名前 : seq of char;

operations
public
  人 : seq of char ==> 人
  人(s) ==
    i 名前 := s;
public
  名前を得る : () ==> seq of char
  名前を得る() ==
    return i 名前
end
人
```

Test Suite : vdm.tc

Class : 人

Name	#Calls	Coverage
人`人	0	0%
人`名前を得る	48	√
Total Coverage		**50%**

15 職員

職員を表す。

```
class
職員 is subclass of 人
operations
public
  職員 : seq of char ==> 職員
  職員(s) ==
    i 名前 := s
end
職員
```

Test Suite : vdm.tc

Class : 職員

Name	#Calls	Coverage
職員 ‘職員	3	√
Total Coverage		**100%**

16　利用者

利用者を表す。

```
class
利用者 is subclass of 人
operations
public
  利用者 : seq of char ==> 利用者
  利用者(s) ==
    i 名前 := s
end
利用者
```

Test Suite : vdm.tc

Class : 利用者

Name	#Calls	Coverage
利用者 ‘利用者	5	√
Total Coverage		**100%**

17　図書館 2 回帰テスト

17.1　TestApp2

17.1.1　責任

回帰テストを行う。

17.1.2　クラス定義

```
class
TestApp2
```

17.1.3　操作：run

回帰テストケースを TestSuite に追加し、実行し、成功したか判定する。

```
operations
public static
 run : () ==> ()
 run () ==
   (   dcl ts : TestSuite := new TestSuite ("図書館 2 : オブジェクト指向図書館貸し出しモデルの回帰テスト \n"),
           tr : TestResult := new TestResult ();
       tr.addListener(new PrintTestListener ());
       ts.addTest(new TestCaseUT2001 ("TestCaseUT2001  :  \t ノーマルケースの単体テスト \n"));
       ts.addTest(new TestCaseUT2002 ("TestCaseST2002 : \t エラーケースの単体テスト \n"));
       ts.run(tr);
       if tr.wasSuccessful () = true
       then def - = new IO ().echo ("***  全回帰テストケース成功。 ***") in
            skip
       else def - = new IO ().echo ("***  失敗したテストケースあり!!  ***") in
            skip
   )
end
TestApp2
```

Test Suite :　vdm.tc

Class :　TestApp2

Name	**#Calls**	**Coverage**
TestApp2‘run	1	82%
Total Coverage		**82%**

17.2 TestCaseComm2

17.2.1 責任

回帰テスト共通機能を記述する。

```
class
TestCaseComm2 is subclass of TestCase
operations
public
  print : seq of char ==> ()
  print(a 文字列) ==
    let - = new IO().echo(a 文字列) in
    skip
end
TestCaseComm2
```

17.3 TestCaseUT2001

17.3.1 責任

図書館のノーマルケースの単体テストを行う。

```
class
TestCaseUT2001 is subclass of TestCaseComm2
operations
public
  TestCaseUT2001 : seq of char ==> TestCaseUT2001
  TestCaseUT2001(name) ==
    setName(name);
public
  test01 : () ==> ()
  test01() ==
    (   let w図書館 = new 図書館2(),
            分野1 = new 分野("ソフトウェア"),
            分野2 = new 分野("工学"),
            分野3 = new 分野("小説"),
            著者1 = new 著者("佐原伸"),
            著者2 = new 著者("井上ひさし"),
            本1 = new 本(著者1,"デザインパターン",{分野1,分野2}),
```

```
        本2 = new 本(著者2, "紙屋町桜ホテル", {分野3}),
        蔵書1 = {mk_token(101) |-> 本1},
        蔵書2 = {mk_token(102) |-> 本1},
        蔵書3 = {mk_token(103) |-> 本2},
        蔵書4 = {mk_token(104) |-> 本2},
        利用者1 = new 利用者("利用者1"),
        利用者2 = new 利用者("利用者2"),
        職員1 = new 職員("職員1") in
    (   w図書館.蔵書を追加する(蔵書1);
        w図書館.蔵書を追加する(蔵書2);
        w図書館.蔵書を追加する(蔵書3);
        w図書館.蔵書を追加する(蔵書4);
        assertTrue("test01 蔵書の追加がおかしい。",
            w図書館.蔵書を得る() = {mk_token(101) |-> 本1, mk_token(102) |-> 本1, mk_token(103) |-> 本2, mk_token(104) |-> 本2});
        let w見つかった本1 = w図書館.本を検索する("井上ひさし"),
```

```
                w 見つかった本 2 = w 図書館.本を検索する ("工学") in
            assertTrue("test01 本の検索がおかしい。",
                w 見つかった本 1 (mk_token (103)).題名を得る () = "紙屋町桜ホテル" and
                w 見つかった本 1 (mk_token (103)).著者を得る ().著者名を得る () = "井 上 ひ さ し
" and
                {f.分野名を得る () | f in set w 見つかった本 1 (mk_token (103)).分野集合を得る ()} = {"
小説"} and
                w 見つかった本 1 (mk_token (104)).題名を得る () = "紙屋町桜ホテル" and
                w 見つかった本 1 (mk_token (104)).著者を得る ().著者名を得る () = "井 上 ひ さ し
" and
                w 見つかった本 2 (mk_token (101)).題名を得る () = "デザインパターン" and
                w 見つかった本 2 (mk_token (101)).著者を得る ().著者名を得る () = "佐原伸" and
                {f.分野名を得る () | f in set w 見つかった本 2 (mk_token (101)).分野集合を得る ()} = {"
ソフトウェア", "工学"} and
                w 見つかった本 2 (mk_token (102)).題名を得る () = "デザインパターン" and
                w 見つかった本 2 (mk_token (102)).著者を得る ().著者名を得る () = "佐原伸") ;
            w 図書館.本を貸す ({mk_token (101) |-> 本 1}, 利用者 1, 職員 1) ;
            w 図書館.本を貸す ({mk_token (103) |-> 本 2}, 利用者 1, 職員 1) ;
            w 図書館.本を貸す ({mk_token (104) |-> 本 2}, 利用者 2, 職員 1) ;
            assertTrue("test01 本の貸出がおかしい。",
                let w 貸出 = w 図書館.貸出を得る () in
                w 貸出 = { 利用者 1 |-> {mk_token (101) |-> 本 1, mk_token (103) |->
本 2}, 利用者 2 |-> {mk_token (104) |-> 本 2}}) ;
            w 図書館.本を返す ({mk_token (103) |-> 本 2}, 利用者 1, 職員 1) ;
            w 図書館.本を返す ({mk_token (104) |-> 本 2}, 利用者 2, 職員 1) ;
            assertTrue("test01 本の返却がおかしい。",
                let w 貸出 = w 図書館.貸出を得る () in
                w 貸出 = { 利用者 1 |-> {mk_token (101) |-> 本 1}}) ;
            w 図書館.蔵書を削除する ({mk_token (103) |-> 本 2}) ;
            assertTrue("test01 本の返却がおかしい。",
                w 図書館.蔵書を得る () = {mk_token (101) |-> 本 1, mk_token (102) |->
本 1, mk_token (104) |-> 本 2})
        )
    )
end
TestCaseUT2001
.....................................................................................
```

17.4 TestCaseUT2002

17.4.1 責任

図書館のエラーケースの単体テストを行う。

```
.....
class
TestCaseUT2002 is subclass of TestCaseComm2
operations
public
  TestCaseUT2002 : seq of char ==> TestCaseUT2002
  TestCaseUT2002(name) ==
    setName(name);
public
  test01 : () ==> ()
  test01() ==
    (   trap <RuntimeError>
        with  print("\ttest01 意図通り、最大貸出数を超えたエラー発生。\n") in
        let w図書館 = new 図書館2(),
            分野1 = new 分野("ソフトウェア"),
            分野2 = new 分野("工学"),
            分野3 = new 分野("小説"),
            著者1 = new 著者("佐原伸"),
            著者2 = new 著者("井上ひさし"),
            本1 = new 本(著者1,"デザインパターン",{分野1,分野2}),
            本2 = new 本(著者2,"紙屋町桜ホテル",{分野3}),
            蔵書1 = {mk_token(101) |-> 本1},
            蔵書2 = {mk_token(102) |-> 本1},
            蔵書3 = {mk_token(103) |-> 本2},
            蔵書4 = {mk_token(104) |-> 本2},
            利用者1 = new 利用者("利用者1"),
            職員1 = new 職員("職員1") in
        (   w図書館.蔵書を追加する(蔵書1);
            w図書館.蔵書を追加する(蔵書2);
            w図書館.蔵書を追加する(蔵書3);
            w図書館.蔵書を追加する(蔵書4);
            assertTrue("test01 蔵書の追加がおかしい。",
                  w図書館.蔵書を得る() = {mk_token(101) |-> 本1,mk_token(102) |->
本1,mk_token(103) |-> 本2,mk_token(104) |-> 本2});
            let w見つかった本1 = w図書館.本を検索する("井上ひさし"),
```

```
            w 見つかった本 2 = w 図書館.本を検索する ("工学") in
        assertTrue("test01 本の検索がおかしい。",
            w 見つかった本 1 = {mk_token (103) |-> 本 2, mk_token (104) |-> 本 2} and
            w 見つかった本 2 = {mk_token (101) |-> 本 1, mk_token (102) |-> 本 1}) ;
        w 図書館.本を貸す ({mk_token (101) |-> 本 1}, 利用者 1, 職員 1) ;
        w 図書館.本を貸す ({mk_token (102) |-> 本 1}, 利用者 1, 職員 1) ;
        w 図書館.本を貸す ({mk_token (103) |-> 本 2}, 利用者 1, 職員 1) ;
        w 図書館.本を貸す ({mk_token (104) |-> 本 2}, 利用者 1, 職員 1)
    )
  ) ;
public
 test02 : () ==> ()
 test02 () ==
  (  let w 図書館 = new 図書館 2 (),
        分野 1 = new 分野 ("ソフトウェア"),
        分野 2 = new 分野 ("工学"),
        分野 3 = new 分野 ("小説"),
        著者 1 = new 著者 ("佐原伸"),
        著者 2 = new 著者 ("井上ひさし"),
        本 1 = new 本 ( 著者 1, "デザインパターン", { 分野 1, 分野 2}),
        本 2 = new 本 ( 著者 2, "紙屋町桜ホテル", { 分野 3}),
        蔵書 1 = {mk_token (101) |-> 本 1},
        蔵書 2 = {mk_token (102) |-> 本 1},
        蔵書 3 = {mk_token (103) |-> 本 2},
        蔵書 4 = {mk_token (104) |-> 本 2},
        利用者 1 = new 利用者 ("利用者 1"),
        利用者 3 = new 利用者 ("利用者 3"),
        職員 1 = new 職員 ("職員 1") in
    (  dcl w 貸出 : 図書館 2`貸出 := { |-> };
        w 図書館.蔵書を追加する ( 蔵書 1) ;
        w 図書館.蔵書を追加する ( 蔵書 2) ;
        w 図書館.蔵書を追加する ( 蔵書 3) ;
        w 図書館.蔵書を追加する ( 蔵書 4) ;
        assertTrue("test02 蔵書の追加がおかしい。",
            w 図書館.蔵書を得る () = {mk_token (101) |-> 本 1, mk_token (102) |->
本 1, mk_token (103) |-> 本 2, mk_token (104) |-> 本 2}) ;
        let w 見つかった本 1 = w 図書館.本を検索する ("井上ひさし"),
```

```
            w 見つかった本 2 = w 図書館.本を検索する ("工学") in
        assertTrue("test02 本の検索がおかしい。",
            w 見つかった本 1 = {mk_token (103) |-> 本 2, mk_token (104) |-> 本 2} and
            w 見つかった本 2 = {mk_token (101) |-> 本 1, mk_token (102) |-> 本 1}) ;
        w 図書館.本を貸す ({mk_token (101) |-> 本 1}, 利用者 1, 職員 1) ;
        w 図書館.本を貸す ({mk_token (102) |-> 本 1}, 利用者 1, 職員 1) ;
        w 図書館.本を貸す ({mk_token (104) |-> 本 2}, 利用者 1, 職員 1) ;
        w 貸出 := w 図書館.貸出を得る ();
        assertTrue("test02 本の貸出がおかしい。",
            let w 貸出 = w 図書館.貸出を得る () in
            w 貸出 = { 利用者 1 |-> {mk_token (101) |-> 本 1, mk_token (102) |->
本 1, mk_token (104) |-> 本 2}}) ;
        assertTrue("test03 意図に反して、貸出が可能になっている。",
            let w 貸出 = w 図書館.貸出を得る (),
                w 蔵書 = w 図書館.蔵書を得る () in
            w 図書館.貸出可能でない ( 利用者 1, 蔵書 1, w 貸出, w 蔵書, w 図書館.v 最大蔵書数)) ;
        trap <RuntimeError>
        with print("\ttest01 意図通り、貸出本を削除するエラー発生。\n") in
        w 図書館.蔵書を削除する ({mk_token (104) |-> 本 2}) ;
        assertFalse("test03 意図に反して、利用者が存在している。",
            let w 貸出 new = w 図書館.貸出を得る (),
                m : 書庫 ‘蔵書 = {mk_token (104) |-> 本 2} in
            w 図書館.貸出に追加されている (m, 利用者 3, w 貸出 new, w 貸出))
    )
  )
end
TestCaseUT2002
.....
```

18　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[*5]の形式仕様記述言語である。

VDM++ の教科書としては [3] がある。

VDM++ を開発現場で実践的に使う場合の解説が [4] にある。

参考文献

[1] SCSK Corporation. VDM-SL 言語マニュアル. SCSK Corporation, 第 1.2 版, 2012. Revised for VDMTools V9.0.2.

[2] SCSK Corporation. VDM++ 言語マニュアル. SCSK Corporation, 第 1.2 版, 2012. Revised for VDMTools V9.0.2.

[3] ジョン・フィッツジェラルド, ピーター・ゴルム・ラーセン, ポール・マッカージー, ニコ・プラット, マーセル・バーホフ (著), , 酒匂寛 (訳). VDM++ によるオブジェクト指向システムの高品質設計と検証. IT architects' archive. 翔泳社, 2010.

[4] 佐原伸. 形式手法の技術講座—ソフトウェアトラブルを予防する. ソフトリサーチセンター, 2008.

[*5] 使用に際しては、SCSK 株式会社との契約締結が必要になる。

索引