

実務家のための形式手法

厳密な仕様記述を志すための形式手法入門

対象を如何にモデル化するか？

2013 年 3 月



独立行政法人 情報処理推進機構  
Information-technology Promotion Agency, Japan

掲載されている会社名・製品名などは、各社の登録商標または商標です。

Copyright© Information-Technology Promotion Agency, Japan. All Rights Reserved 2013

# 目次

まえがき	ix
はじめに	xi
0.1 対象者	xii
0.2 到達目標	xii
0.3 本書の構成	xii
第 I 部 対象を如何にモデル化するか？	1
第 1 章 なぜ形式手法か？	3
1.1 理論的根拠	4
1.2 経験的根拠	5
1.3 日本語仕様の曖昧さの例	6
図書館の本	6
特急券予約システム	6
応当日	8
第 2 章 形式手法 VDM 概要	11
2.1 形式手法 VDM とは？	12
2.2 VDM++ とは？	12
2.3 なぜ VDM を選択したか？	12
第 3 章 VDM 適用の成果	15
3.1 日本フィッツの事例	16
3.2 フェリカネットワークスの事例	16
3.3 オムロンの事例	17
第 4 章 VDM 導入・開発方法・体制の概要	19
4.1 VDM の教育	20
4.2 仕様記述	20
4.3 仕様テスト	24
4.4 プログラミング	25
4.5 プログラムのテスト	27
4.6 開発管理	27
4.7 開発体制	28

第 5 章 対象を如何にモデル化するか？	31
5.1 モデル化の一般的手順	32
5.2 特急券予約システムの例	34
問題点の要約	34
モデル化対象用語の選	34
モデル化する機能の範囲選択	35
仕様記述フレームワークの設定	35
VDM++ モデル例	36
特急券予約システムクラス	38
共通定義クラス	41
特急券予約 Domain クラス	42
特急券予約 DomainData クラス	46
特急券予約クラス	47
契約クラス	49
カードクラス	51
クレジットカードクラス	52
予約会員証	53
おサイフケータイクラス	55
回帰テストクラス	57
モデルの検証	66
静的検証	66
動的検証	66
特急券予約システムのまとめ	66
問題は何だったのか	66
本来どうすべきだったか	67
特急券予約システムモデル化の統計情報	67
第 6 章 まとめ	69
6.1 形式手法と VDM の有用性	70
6.2 構造化日本語仕様としての VDM	70
6.3 形式手法導入のコツ	71
第 7 章 演習問題	73
7.1 図書館システム	74
第 II 部 モデルの具体例	75
第 8 章 演習問題解答の図書館システムモデル	77
8.1 図書館システムへの要求	78
8.2 実行不可能な仕様	78
8.3 図書館 0 要求辞書	86
8.4 実行可能な仕様 (図書館 1)	90
8.5 図書館要求辞書	94

8.6	図書館 1 の回帰テスト	98
8.7	オブジェクト指向モデル	104
8.8	図書館 2	104
8.9	著者	106
8.10	本	107
8.11	書庫	109
8.12	分野	112
8.13	貸出情報	113
8.14	人	116
8.15	職員	117
8.16	利用者	118
8.17	図書館 2 回帰テスト	119
8.18	8 章の参考文献	125
第 9 章	運賃計算モデル	127
9.1	運賃を得る	128
9.2	運賃表辞書	129
9.3	路線網	132
9.4	路線検索	134
9.5	ダイクストラ算法による路線検索	136
9.6	ダイクストラ算法	137
9.7	鉄道ネットデータ	140
9.8	TestApp Class	142
9.9	回帰テストケース	144
第 10 章	特急券予約システム改善モデル	149
10.1	はじめに	150
10.2	特急券予約システムクラス	151
10.3	共通定義クラス	154
10.4	特急券予約 Domain クラス	155
10.5	特急券予約 DomainData クラス	159
10.6	特急券予約クラス	160
10.7	契約クラス	162
10.8	カードクラス	164
10.9	クレジットカードクラス	165
10.10	予約会員証	166
10.11	おサイフケータイクラス	167
10.12	TestApp クラス	168
10.13	TestCaseT0001 クラス	170
10.14	TestCaseT0002 クラス	171
10.15	TestCaseT0003 クラス	172
10.16	TestCaseT0004 クラス	174
10.17	まとめ	177

---

参考文献	179
索引	180

# 図目次

1.1	開発工程別の欠陥修正コスト	4
4.1	業務論理とシナリオとテストケースの対応関係	22
4.2	運賃計算モデルのクラス図	23
4.3	組合せテスト実行画面例	26
4.4	フェリカネットワークスの開発体制	28
5.1	クラス図 (特急券予約システム)	37
8.1	図書館 2 モデルのクラス図	104
10.1	修正後のクラス図	150



# 表目次

3.1	VDM による仕様作成の成果 . . . . .	16
4.1	VDM の教育 . . . . .	20
4.2	仕様記述フレームワーク (日本フィッツ) . . . . .	21
4.3	仕様記述フレームワーク (フェリカネットワークス) . . . . .	22
5.1	仕様の階層 (特急券予約システム) . . . . .	36
5.2	特急券予約システムの仕様の階層 . . . . .	37
6.1	日本語の仕様中の擬似コードと、VDM の比較 . . . . .	71



# まえがき

独立行政法人情報処理推進機構 (IPA) ソフトウェア・エンジニアリング・センター (SEC) において、ここ数年間にわたって高品質高信頼システムの開発技術に関して複数の作業部会 (WG) を通して活動を行ってきました。これらの WG の名称や委員の構成は、状況に応じて変遷してきましたけれども、いわゆるフォーマルメソッドに基づくソフトウェア開発に関する調査研究ならびに人材育成に関する活動は一貫して継続してきました。

2010 年度以降、フォーマルメソッドに関する入門セミナーないしフォーマルメソッド導入に関するガイダンスセミナーを国内の広島、札幌、熊本、尼崎、名古屋、東京、盛岡などで開催してきました。この間、セミナー実施経験や受講者からのアンケート結果などに基づいてセミナー教材の大幅な改訂を行い、IPA/SEC のホームページ上で「厳密な仕様記述を志すための形式手法入門」として公開しています。2012 年度からは、改訂版の教材を使用してセミナーを実施しておりますが、改定前の教材でセミナーを行った都市のなかには改訂版の教材を用いて再度セミナーを開催したところもあります。

このセミナーは、エンジニア向けと管理者向けに構成を変えて、また、半日コース、一日コース、二日間コースと所要時間も柔軟に組替えられるように工夫しました。しかしながら、このセミナーでは、受講者の方々にフォーマルメソッドの概要を伝えることはできても、フォーマルメソッド適用に関する具体的な手法を理解し習得して頂くには、あまりにも時間が短すぎます。このセミナーでフォーマルメソッドを代表する一つの手法として取り上げている VDM (Vienna Development Method) に基づく厳密なシステムの記述に関して、もっと具体的に知りたいという受講者の要望も数多く頂きました。

そこで、セミナーで VDM によるシステムの記述の部分を担当している佐原伸氏に、セミナーで紹介する具体例に関する詳細な副読本としてこの本を執筆して頂きました。

セミナー教材は、上述のように IPA/SEC のホームページ上で公開されておりますので、セミナー受講者でなくても、その教材とこの副読本によってフォーマルメソッド、特に、VDM による厳密なシステム記述に関する基本的な知識と、システムの厳密な記述の手法に関する理解をえることができます。実のところ、この副読本は、佐原氏のソフトウェア開発に関する豊富な経験ならびに高い見識に基づいて、フォーマルメソッドの意義や有用性について実践的な観点からの説得力ある記述と共に、具体事例を対象として詳細な VDM 記述を掲載しておりますので、フォーマルメソッドの入門書として単独で読んで頂いても十分に役立つ貴重な書物となっています。

また、IPA/SEC では、フォーマルメソッドのセミナー資料の他にも、「厳密な仕様記述における形式手法成功事例調査報告書」を公開しています。さらに、これの報告書に関連して、「厳密な仕様記述入門」という本も作成しました。これらは、本書の姉妹編とも言うべき有益な資料です。本書と併せてこれらもご覧頂くことによって、フォーマルメソッドに基づく高品質高信頼ソフトウェアの効率的な開発法に関する理解が深まるとともに、自分たちのところでも実際にフォーマルメソッドを導入してみようかという気持ちになって頂ければ幸いです。

荒木 啓二郎

上流品質技術部会 人材育成 WG 主査

# はじめに

形式手法で仕様を明確化する技術を教えている時に、常に問われる「対象を如何にモデル化するか？」を説明するために作成したのが、本ブックレットである。

ここで言うモデル化とは、システムの仕様に内部矛盾が無いかを検証（正当性検証）し、ユーザーの要求に合っているかを確認する（妥当性確認）モデルを作成することである。

## 0.1 対象者

以下の対象者を想定して作成した。

- 現場において、システムの品質を改善し生産性を高めたいと考えているプロジェクトのリーダー、開発者の方
- 仕様作成とプログラム開発について、ある程度の知見を持っている方

## 0.2 到達目標

到達目標は以下の通りである。

- 要求仕様の記述と分析の重要性が理解できていて、第三者に客観的に説明できる。  
特に、厳密な仕様記述が品質向上と生産性向上に寄与することを理解している。
- 要求仕様作成のために、「対象を如何にモデル化するか」の手順とその意味が理解できている。
- 形式仕様記述言語 VDM++ を言語マニュアル [4] 等を参照しながら活用すれば、  
小さな実システムの要求仕様を記述する事ができる。

上記の対象者と到達目標を考慮して、実際のシステム開発に流用することができ、読者にも知識がある問題について、小さな実例をベースに説明を行い、形式仕様記述言語 VDM++ のソースをできるだけ紹介することで、具体的イメージを持っていただくことにした。

ただし、VDM++ の言語仕様説明は必要最小限に留めたので、VDM++ ソース部分を完全に理解するためには、VDM++ 言語マニュアル [4] を参照していただきたい。

本書の VDM++ ソースは公開されている<sup>\*1</sup>ので、いつでも VDM のツールを用いて動かすことができる。これが UML などの、レビュー以外は検証できず動かないモデル化と大きく異なる点である。

## 0.3 本書の構成

本書は、Ⅱ部構成となっている。

第Ⅰ部では、対象を如何にモデル化するか？を説明する。

1章でなぜ形式手法か？を紹介し、2章で形式手法 VDM の概要を説明し、3章で VDM 適用の成果を紹介する。

そして、4章で VDM 導入・開発方法・体制の概要を示し、5章で対象を如何にモデル化するか？を説明する。

6章でまとめを行い、7章で演習問題を示す。

第Ⅱ部では、モデルの具体例を紹介する。第Ⅰ部の説明に関連する VDM++ モデルのソースとその説明を行なっている。

8章は、演習問題の解答例である図書館システムのモデルであり、実行不可能な陰仕様<sup>\*2</sup>と、それを回帰テストするた

<sup>\*1</sup> <http://sec.ipa.go.jp/reports/20121113.html>

<sup>\*2</sup> 関数・操作のインタフェースと、事前条件及び事後条件しか記述していない仕様を陰仕様と呼ぶ。陰仕様は、関数・操作の本体は記述していないが、「仕様」を記述していることになる。

めの実行可能な陽仕様<sup>\*3</sup>、さらに、やや設計仕様寄りのオブジェクト指向陽仕様としたモデルの例である。モデルの理解に必要な、最小限の VDM++ 文法も説明している。

9 章は本文中に出てくる運賃計算モデルであり、実システムの 10 分の 1 程度の行数でありながら、4.2.1 節で説明する仕様フレームワークと 4.3.2 節で説明する回帰テストを使った要求仕様の例である。

10 章は、5 章の特急券予約システムを改善したモデルの例であり、実用的なシステムを、ある目的のために抽象化して、小さな要求仕様モデルとして記述した例である。

---

<sup>\*3</sup> 関数・操作の本体が記述されていて実行可能な仕様を陽仕様と呼ぶ。



## 第Ⅰ部

# 対象を如何にモデル化するか？



## 第 1 章

# なぜ形式手法か？

形式手法を強く推奨する理由には、理論的根拠と経験的根拠がある。以下の節で、それぞれを説明する。

## 1.1 理論的根拠

形式手法を強く推奨する理論的根拠は、以下の 3 つである。

- プログラムは数学系だから

プログラムは元来、数学的に厳密な体系で定義された一つの数学系である。しかも、C++ や Java のプログラムは「検証が困難な数学系<sup>\*1</sup>」であることが分かっている。これに対して、形式手法は「検証が容易な数学系」をもとにしている。したがって、検証しやすい数学系で記述した方が品質を上げやすい。

- 形式手法では、上流工程で仕様を検証可能だから

要求分析工程<sup>\*2</sup>からプログラミング工程までのどこかで、数学系に変換しなければならない。いずれどこかの工程で数学系にするならば、なるべく早い工程、すなわち上流工程で数学系にしておいたほうが、下流工程から上流工程への手戻りが減り、プログラムの質だけでなく、開発の生産性も向上する。そして、形式手法は上流工程での検証を支援する。

要求分析工程での欠陥修正コストを 1 とした時の、その他の工程での欠陥修正コストを示したものが、図 1.1 である。<sup>\*3</sup>

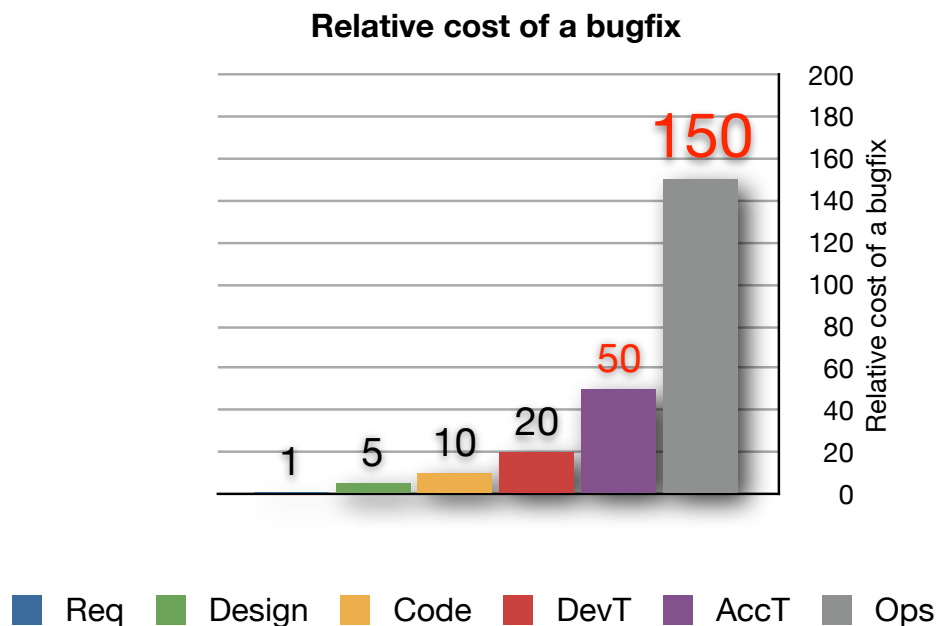


図 1.1 開発工程別の欠陥修正コスト

このグラフから分かるように、上流工程での欠陥修正コストは、下流工程のその 5 分の 1 から 150 分の 1 にもなる。

<sup>\*1</sup> 変数への値の代入などの破壊的代入を行うので、「同じ条件を与えれば必ず同じ結果が得られ、他のいかなる機能の結果にも影響を与えない」という参照透過性を保つことができない。

<sup>\*2</sup> ユースケース記述などを作成する要求収集 (Elicitation) 工程の出力を解析し、厳密な仕様を記述し、その妥当性確認 (validation) と正当性検証 (verification) を行う工程。本ブックレットでは、要求工学が対象とする工程、あるいは要件定義と称する工程は、要求収集工程と見なす。

<sup>\*3</sup> Source: Barry Boehm: "EQUITY Keynote Address", March 19th, 2007

ここで、横軸の Req, Design, Code, DevT, AccT, Ops は、それぞれ要求分析、設計、コーディング、開発テスト、受入テスト、運用の各工程を示している。開発テストはシステム開発者によるテスト、受入テストはシステム発注者によるテストである。

- 旧来の手法では、検証が困難だから  
旧来の手法では、レビュー以外の検証手段が無いため、中規模以上のシステムの検証は非常に困難であり、大規模システムの検証は不可能といっても良いほどの困難を伴う。

## 1.2 経験的根拠

形式手法による仕様記述を行った経験から分かったことは、以下の 2 点である。

- 日本語で「正しい」仕様を書くのは「不可能」である
  - － 曖昧さの排除が困難  
自然言語による仕様では、曖昧さの排除が困難である。
  - － ツールによるチェックが、ほとんど不可能  
自然言語で書かれた仕様を、ツールによってチェックすることはほとんどできない。これは、自然言語の文法定義と意味定義が曖昧なことによるもので、どのようなツールを作っても、曖昧さを排除することはできない。  
これに対し、形式手法で使用される形式仕様記述言語は、文法定義・意味定義ともに厳密であり、曖昧な仕様を書くことはできない。  
実際に、3 章で示す成功例では、ツールによる静的検証<sup>\*4</sup>によって、ほとんどの単純ミスを検出し、動的検証<sup>\*5</sup>によって、すべての欠陥を検出し、修正することに成功した。
  - － 仕様の修正に弱い  
自然言語仕様の検証は、主としてレビューに頼っているため機械化できない。これに対して形式手法は、形式仕様記述言語で書かれた仕様を、機械的に検証する手段を持っている。  
3 章で示す成功例では、「仕様修正に強い」形式手法の利点が、品質向上と検証工数削減に大いに役立った。
  - － その場で「文法」を考えなければならない  
日本語で仕様を書ききれなくなり、if 文などを使った擬似コード的な仕様を書くことが多く、擬似コードの文法を考えている時間が馬鹿にならない。すなわち、仕様を書く際に、仕様の意味ではなく、文法に注意が向いてしまい、肝心の「意味を考える」ことがおろそかになってしまう。
- 形式手法の方が技術移転や蓄積が容易である
  - － 日本語による意思疎通は難しい  
日本語仕様は、開発者間のコミュニケーションを経るたびに、一種の伝言ゲームになってしまい、当初の意味がずれていくことが多い。打ち合わせなどで「あれって、こういう意味じゃないの」というような議論が起こることが多ければ、それは、自然言語の曖昧さによって意思疎通がうまくいっていない事の証である。
  - － 仕様記述言語を読むのは容易である  
自然言語仕様であろうが、形式仕様であろうが、仕様を書くことは難しい。しかし、記述された仕様を読むことは、形式仕様記述言語の場合、容易である。それは、非形式的な図形言語 UML1.x より容易であり、UML2.0 より遙かに簡単である。
  - － フレームワークやライブラリの構築が容易である

<sup>\*4</sup> モデルを動かさずに検証すること

<sup>\*5</sup> モデルを動かして検証すること

形式仕様記述言語は、プログラミング言語などと同じように、フレームワークやライブラリの構築が容易であり、再利用性や保守性に優れている。

- － 業務知識の蓄積も容易である

Excel や Word による自然言語仕様と異なり、形式仕様記述言語の仕様は通常テキストファイルであり、プログラミング言語と同じように、版管理や構成管理ツールと連動して容易に管理できるため、業務知識をいったん形式仕様記述言語で記述できれば、それを、再利用・保守可能な業務知識として、蓄積していくことが容易である。

- － 仕様を「動かしてみる」ことができるので、理解しやすい形式手法は、形式仕様を動かす (実行する<sup>\*6</sup>、モデル検査する、証明する) が可能で、動かない旧来手法より理解することが容易である。

### 1.3 日本語仕様の曖昧さの例

この節では、著者が出会ったことのある、日本語仕様の曖昧さの例を示す。

#### 1.3.1 図書館の本

図書館のシステムを考えた場合、以下の 2 つの文に出てくる「本」は同じものを意味しているだろうか？

- 題名で本を検索する
- 本を図書館の蔵書として追加する

この 2 つの「本」は同じ場合もあるが、普通は、異なるものを指す。

「題名で本を検索する」場合の「本」は、多くの場合、ある著者が書いたある題名の「本」を指す。この「本」は抽象的な概念で、例えばこの「本」が 1000 部出版されているとしても、我々は「ひとつの本」として認識している。

しかし、図書館の蔵書に追加する「本」はそうではない。これは物理的な 1 冊の本であり、抽象的な「本」が 1000 部出版されていれば、こちらの「物理的な本」は 1000 冊存在する。

日本語による仕様記述では、このように区別すべき用語が同じものとして記述されたり、同じ意味の用語が別の言葉で記述されて曖昧さが混入し、開発者同士のコミュニケーションが混乱し、結果として、作業の手戻りの多発による生産性の低下と品質の低下を招く。

詳細は、8 章を参照していただきたい。

#### 1.3.2 特急券予約システム

ここでは、筆者が実際に遭遇した、鉄道会社 A の特急券予約システムの問題点を紹介する。

問題の発端は、クレジットカード会社の都合で、特急券予約システムに使用していたクレジットカードを変更せざるを得なくなり、特急券予約システムの設定を変更しようとしたところから始まった。

以下は、鉄道会社 A サポートセンターとの問答である。<sup>\*7</sup>

1. 客 (すなわち筆者) は、おサイフケータイ (鉄道会社 B) で特急券予約システム (鉄道会社 A) を使っていた。
2. 会社 C クレジットカードが廃止になり会社 D クレジットカードに変更して下さいとの連絡があった。
3. 鉄道会社 A サポートセンターに電話したところ、おサイフケータイの登録クレジットカードを変更すれば、2 日後には特急券予約システムが使えますとのことだった。

<sup>\*6</sup> 仕様アニメーションという

<sup>\*7</sup> 実際にあった事例をかなり省略している。

4. おサイフケータイの設定でクレジットカードを変更し、2 日後に使用しようとしたが、特急券が予約できなかった。
5. 予約できなかったので鉄道会社 A サポートセンターに電話した  
客 「クレジットカードを変更したら、予約できないんですが？」  
鉄道会社 A 「カードを変更したら、新規に特急券予約システムを契約して下さい。」  
客 「クレジットカード変更前に予約した特急券に引き換えようとしたらできないのですが？」  
鉄道会社 A 「カードで引き替えできるようになったので、それで引き換えて下さい。暗証番号はカードのものを  
使して下さい。」  
客 「カードの暗証番号って、何ですか？」  
鉄道会社 A 「クレジットカードの暗証番号です。」
6. 旅行当日の T 駅での問答  
客 「クレジットカードで特急券に引き換えできないんですが？」  
鉄道会社 B 「会員証でやってみて下さい。」  
客 「会員証でもできないんですが？」  
鉄道会社 B 「変ですねー、こちらでやってみましょう。駄目ですねー…」  
客 「ひょっとして、古いクレジットカードでは駄目ですか？」  
鉄道会社 B 「あ、できましたね。はい、切符です。」

上記のやり取りで、何が起こっていたかを正確に理解できる方は居るだろうか？理解できるはずはない。鉄道会社 A 側の用語の使い方が適切ではないので、鉄道会社 A サポートセンターの要員も、T 駅の鉄道会社 B の駅員も事態を正確に把握できなかったのだから。

その後、手元にある特急券予約システム関係のカード類と、インターネット上の情報と、鉄道会社 A サポートセンターへの何回かの電話によって、幾つかの用語の定義、すなわちカードの種類が判明した。

すなわち、

- 予約会員証 (変更前、変更後)、クレジットカード (変更前、変更後)
- おサイフケータイ<sup>\*8</sup>
- IC カード、予約カード

このうち、IC カードはおサイフケータイで予約する場合には関係ないカードであることが判明した。また、予約カードも鉄道会社 A のカードであり、おサイフケータイ (鉄道会社 B) から特急券予約システムする場合には関係ないことも判明した。

これらを反映して、最初の鉄道会社 A 側とのやり取りを、正確な用語で記述すると以下ようになる。

1. 予約できなかったので鉄道会社 A サポートセンターに電話した  
客 「クレジットカードを変更したら、予約できないんですが？」  
鉄道会社 A 「クレジットカードを変更したら、新規に特急券予約システムを契約して下さい。」  
客 「クレジットカード変更前に予約した特急券に引き替えようとしたらできないのですが？」  
鉄道会社 A 「予約会員証で引き替えできるようになったので、それで引き換えて下さい。暗証番号はクレジットカードのものを  
使して下さい。」
2. 旅行当日の T 駅での問答  
客 「変更後のクレジットカードで特急券に引き換えできないんですが？」  
鉄道会社 B 「予約会員証でやってみて下さい。」

<sup>\*8</sup> ここでは、スマートカードとして特急券を予約する機能だけに着目する。

客 「予約会員証でもできないんですが？」

鉄道会社 B 「変ですねー、こちらでやってみましょう。駄目ですねー…」

客 「ひょっとして、変更前のクレジットカードでは駄目ですか？」

鉄道会社 B 「あ、できましたね。はい、切符です。」

この記述は、VDM++ という形式仕様記述言語を使って、前記の問答に関わる仕様 (モデル) <sup>\*9</sup>を書き、最初の問題発生から 3 ヶ月かかって、問題を解決した後に記述したものである。

### 1.3.3 応当日

以下は、証券会社用のパッケージソフトウェアを作成していた際、「信用取引の決済日」の定義を聞いた時に回答として得た説明文章である。

- 信用取引の決済日 (期日) を得る。  
 弁済期限とは、信用建玉に対して当社がお客様に信用を供与する期限をいいます。弁済期限は、現在のところ 6 ヶ月のみを取扱っています。弁済期限が 6 ヶ月であるということは、信用建玉の建日 (信用建玉が約定した日) の 6 ヶ月目応当日が信用期日となり、この日を超えて建玉を保有することは法律で禁じられています。信用期日が休日の場合には、直近の前営業日が信用期日となります。

直ちに生じる疑問は、以下のようなものであろう。

- 弁済期限と決済日と信用期日との関係は？
- 応当日とは何か？

弁済期限と決済日と信用期日との関係を質した所、直ちに回答が得られた。すなわち、これらは同一である。

応当日については、何回かのやり取りの後、以下の回答が得られた。

普通の日本語で書くと分かり難くなるので、やや形式化した日本語で応当日を「曖昧に」定義すると、以下のようになる。

現在日を  $y$  年  $m$  月  $d$  日としたとき、 $n$  ヶ月後の応当日とは、 $y$  年  $m+n$  月  $d$  日のこと。

ここで、 $m+n \geq 13$  ならば、

年候補  $Y$  を  $y + (m+n - 1) \div 12$  とし、

月候補  $M$  を  $m+n \bmod 12$  として、

$M$  が 0 ならば、 $M$  は 12、

$M$  が 0 以外ならば、 $M$  をそのまま使って、

$Y$  年  $M$  月  $d$  日が応当日である。

ここで、 $\div$  は整数の除算、 $\bmod$  は整数除算の余りを得る演算子である。

この日本語仕様には、まだ、以下の問題が残っている。

1. たとえば信用建玉の建日が 8 月 31 日だと、応当日は翌年 2 月 31 日になってしまうのだが、そのとき 2 月 28 日あるいは 2 月 29 日にするのか 3 月 1 日にするのかの記述がない。
2. 応当日から月初日まですべて休日の場合、前月の営業日にしてよいのかどうかの記述がない。

---

<sup>\*9</sup> 詳細は、5 章と、10 章を参照のこと。

この疑問は、証券業務の専門家に質問したところ、半日ほどで回答を得た。

- 月末日を越えたら、月末日に最も近いその月の営業日。
- 応当日が休日で、前営業日が前の月の場合、その営業日が応当日になる。

以上のように、「信用取引の決済日を得る」という証券業務にとっては基本的な用語すら、同じ事を表す複数の言葉が存在し、定義が曖昧で、ある程度の仕様分析を行った後、証券業務の専門家に質問しても答えを得るのに半日以上を要してようやく正しい仕様を理解できた。

しかも、応当日の年や月を求める計算は、日本語で記述し理解するには複雑すぎる。

これでは、生産性と品質が悪くなるはずである。



## 第 2 章

# 形式手法 VDM 概要

本章では、形式手法の概要と本ブックレットで使用する VDM について説明する。

## 2.1 形式手法 VDM とは？

VDM は Vienna Development Method の略であり、1960 年代から 1970 年代に IBM ウィーン研究所で開発され、IBM の各種プログラミング言語の仕様記述に使用された。

その後、汎用の仕様記述言語として拡張され、1996 年に、その仕様記述言語 VDM-SL[3] が世界初の ISO 標準 (ISO/IEC 13817) 仕様記述言語になった 形式手法の元祖である。

特徴としては、以下がある。

- 厳密に定義された仕様記述言語 VDM-SL, VDM++, VDM-RT を持つ
- 制約条件 (不変条件、事後条件、事前条件) の記述が可能である
- 証明手法がある<sup>\*1</sup>
- 産業界の実用のために拡張された

## 2.2 VDM++ とは？

VDM++[4] は、1993 年に、欧州連合 ESPRIT 計画の AFRODITE プロジェクトで、VDM-SL[3] をオブジェクト指向拡張したオブジェクト指向形式仕様記述言語である。

オブジェクト指向構文以外に、関数型言語構文を持ち、C++ や Java などと同じく構造化言語の構文も持っている。VDMTools と Overture Tools という 2 つの VDM++ 支援ツールがあり、それらを用いて、以下の様な仕様の検証が可能になった。

- 構文チェック、型チェック、仕様アニメーション (仕様のシンボリック実行)
- 仕様アニメーションによる組合せテストおよび回帰テストと、テストのコードカバレッジ表示
- 証明課題<sup>\*2</sup>生成機能を使った、証明課題レビュー

VDMTools は、C++ と Java の生成機能がある。

## 2.3 なぜ VDM を選択したか？

### 2.3.1 形式手法の種類

形式手法には主なものとして、モデル規範型、性質規範型、モデル検査がある。この中で、現場の技術者に理解しやすくシステム全体を仕様化できるので、モデル規範型 VDM を採用した。実際に 3 章で紹介するシステムでは、3 ヶ月の学習と仕様記述の試行を経て、実用開発を行うことができた。

モデル規範型には、他にも B, event-B, Z, RAISE といった有用な手法があるが、いずれも、「証明を前提」としているため、開発現場で適用する「最初の形式手法」としては、不適切と判断したのである。

性質規範型の手法としては、CafeOBJ や Maude などがあるが、VDM より 1 段階抽象度が高く、これも「最初の形式手法」としては、不適切と判断した。

モデル検査手法には SPIN や SMV があるが、モデル検査用仕様記述言語は、検証効率を上げるため、システムの振舞を表す部分の一部だけを仕様記述するため、データ構造などを含むシステムの他の側面をほとんど記述することができ

<sup>\*1</sup> 証明手法は初心者には難しいため、本書では説明を省略した。

<sup>\*2</sup> 証明課題が全て証明できれば、モデルに内部矛盾は無いと主張できる。

ない。すなわち、システム全体の仕様を記述することはできず、ある側面からの検証にとどまり、かつ、開発現場に適用するには 3 年程度の試行期間が必要と思われる、「最初の形式手法」としては不適切と判断した。

### 2.3.2 形式手法の検証方法とツール

形式手法の検証方法としては、証明・モデル検査・仕様アニメーションの 3 つがある。

証明は RAISE を支援する RAISE Tool や、event-B を支援する Rodin があるが、最低でも 5 年程度の学習期間が必要と判断して、「最初の形式手法」としては採用しなかった。

モデル検査は、ツールを使用することにより、教科書的な簡単な問題は、簡単に検証することができる場合があるが、エラーが見つかった場合の、原因の特定などに難しいところがあり、「最初の形式手法」としては採用しなかった。

結局、VDM を使用する VDMTools が、仕様アニメーションを使用しているので、プログラムテストと同じ感覚で使うことができるため、開発現場の技術者の「最初の形式手法ツール」として最適だと判断した。

### 2.3.3 VDM の参考文献

VDM++<sup>[4]</sup> は、1970 年代中頃に IBM ウィーン研究所で開発された VDM-SL<sup>[3]</sup> を拡張し、さらにオブジェクト指向拡張した形式仕様記述言語である。

VDM++ の教科書としては <sup>[5]</sup> がある。

VDM++ を開発現場で実践的に使う場合の解説が <sup>[6]</sup> にある。



## 第 3 章

# VDM 適用の成果

VDM の海外での適用事例は、IBM での多くのプログラミング言語開発、旧ソ連の 100 万行以上の仕様で記述された衛星システム、VDM++ から C++ を生成したオランダの世界最大の花市場オークションシステムなどがある。

日本での適用事例には、表 3.1 がある。

表 3.1 VDM による仕様作成の成果

開発組織	対象システム	適用工程	仕様規模	欠陥数	生産性	開発期間
日本フィッツ 現 SCSK	証券業務	実システム UseCase 要求仕様	3+3 万行 仕様 + テストケース	0	2.5 倍	45%
フェリカネットワークス	おサイフケータイ ファームウェア	実システム API 外部仕様	7.4+6.6 万行 仕様 + テストケース	0	2 倍	85%
産業技術総合研究所 オムロン	鉄道 駅務システム	既存システム 厳密仕様 作成実験	0.75+80 万行 仕様 + テストケース	従来仕様書の 欠陥発見数 (29)		

表 3.1 では、生産性と開発期間の比較は、COCOMO81[1] による見積りと実績値の比較で行った。生産性のカラムは、見積もりに対して実績値が何倍になったかを示し、開発期間のカラムは、見積もりに対して実績値が何 % であったかを示している。

仕様規模のカラムは、 $m + n$  という形式で、注釈抜きの VDM++ ソース行数を表していて、 $m$  の部分は仕様本体、 $n$  の部分はテストケースの行数を表している。

以下に、各システムの概要を述べる。

### 3.1 日本フィッツの事例

証券会社バックオフィスシステム用パッケージソフトウェアの 2 つのサブシステムに VDM++ 仕様を適用した。最初に手がけたサブシステムは識別子に英語を使用したのが、2 つ目のサブシステムでは、VDMTools を修正して国際語化し、日本語識別子と日本語文字列を使用した VDM++ 仕様を作成した。

開発チームは証券業務の知識がなく、仕様が曖昧なまま開発するのは危険と考え、UseCase レベルの要求仕様記述を VDM++ を用いて行い、回帰テストのテストケースも VDM++ で記述し、仕様の検証を VDMTools の仕様アニメーション機能 (シンボリック仕様実行) で行った。回帰テストのコードカバレッジを VDMTools で計測し、リリース時には 95% 以上のカバレッジを達成した。

開発チームのうちの 2 人が VDM++ による記述を行ったが、2 人とも VDM++ の知識はあったものの、実際に VDM++ を業務として使うのは初めてだった。プログラマも Java と C++ の知識はあったものの、実業務に使うのは初めてであった。

表 3.1 には、VDM++ で開発した 3000 行あまりのライブラリの開発工数も含まれていて、ライブラリを除くアプリケーション開発自体の生産性はさらに高いが、ライブラリ開発とアプリケーション開発の工数を分離して計測する時間的余裕は無かったため、ライブラリ開発がどの程度の割合を占めたかの正確なデータは残っていない。

### 3.2 フェリカネットワークスの事例

以前のおサイフケータイ・ファームウェアの開発で、仕様の曖昧さのため非常に苦労したため、VDM++ で仕様を書き、検証することになった。

日本フィッツの開発者をコンサルタントとし、証券業務仕様の記述で用いた仕様記述フレームワーク<sup>\*1</sup>を流用して、予定通り開発を終了し、現在に至るまで欠陥が発生していない。

開発者に形式手法の知識はなかったため、中心となる技術者に VDM++ を教育し、仕様記述フレームワークとその使用例を開発し、それを元に、他の技術者が仕様を作成した。

API 外部仕様は、今までどおりの日本語仕様と、VDM++ 仕様の両方を記述し、矛盾する場合は VDM++ 仕様を優先することとした。

なお、本システムの対象はモバイル FeliCa チップであるが、本システムの後、Edy カードの FeliCa チップ開発にも VDM++ が使用され、本システムと同等以上の成果を上げた。このシステムでは、従来型の日本語仕様は作成せず、日本語識別子を使った VDM++ 仕様のみを作成した。

また、セキュリティ強化版の FeliCa チップ<sup>\*2</sup> 開発にも VDM++ が使用され、情報セキュリティ評価基準の国際標準であるコモンクライテリア (ISO/IEC 15408) において、組み込みソフトウェアを搭載した非接触 IC カードチップとして世界で初めて評価保証レベル EAL6+ 認証を取得した。

### 3.3 オムロンの事例

駅務システムは社会インフラとして高い信頼性が求められているが、従来の仕様書は、今や、理解することや保守あるいは再利用が非常に困難となってきたため、既存仕様を VDM++ で書き換える実験を行った。

結果として、従来型仕様書の不備を 29 件発見し、暗黙知<sup>\*3</sup>が VDM++ で 1400 行相当発見され、鉄道会社共通の仕様も VDM++ で 1400 行発見された。

開発者は「暗黙知は仕様の欠陥であると認識した」と述べている。

検証は、産業技術総合研究所の検証サーバー「さつき」を利用し、64 コア CPU のマシンで実テストデータ 80 万件で 5 日間実行した。このテストの実行速度は、実機の 10 分の 1 から 20 分の 1 であった<sup>\*4</sup>。

<sup>\*1</sup> 仕様記述フレームワークは、仕様のどの部分に何を書くかを定めた規則である。詳しくは、4.2.1 節に説明する。

<sup>\*2</sup> IC RC-SA00/1 Series と RC-SA00/2 Series

<sup>\*3</sup> ここで言う暗黙知は、担当者の一部だけが知っている業務知識で、明文化されていないものを指す。

<sup>\*4</sup> SEC 特別セミナーアーキテクチャ指向エンジニアリングと形式手法における以下の発表資料による。幡山五郎、大崎人士、相馬大輔、Nguyen Van Tang 著、上流工程大規模テストのための技術開発～組み込みシステム開発のフロントローディング化～、2011 年 7 月 5 日



## 第 4 章

# VDM 導入・開発方法・体制の概要

本章では、VDM を如何に導入し、どのような開発方法・体制を採用したかを述べる。

## 4.1 VDM の教育

日本で VDM の導入に成功した表 3.1 の事例で行われた教育の内容を表 4.1 に示す。

表 4.1 VDM の教育

組織	セミナー	教材	コンサルティング	受講者	備考
日本フィッツ  (現 SCSK)	4 日間	英語	形式手法知識を持った要員とオブジェクト指向分析・設計手法 (OOA/OOD) 専門家の要員が相談	平均年齢 50 歳弱  形式手法知識 4 人有 ソフトウェア知識有 業務知識無	VDM 記述予定者の 3 分の 1(2 人) は英語で脱落  C++, Java, 形式手法について、実践経験無
フェリカネットワークス	4 日間	日本語	3 ヶ月/週 1 回	平均年齢 20 歳代 形式手法知識無 ソフトウェア知識無 業務知識有	
産業技術総合研究所 オムロン	無し 自習	日本語 参考文献 [5] [6]	最初は無し その後、3 ヶ月/週 1 回	平均年齢 30 歳代 形式手法知識 1 人有  ソフトウェア知識無	周囲に形式手法経験者多数

海外の形式手法使用事例でも、形式手法の初期教育は約 1 週間であり、その後 3 ヶ月ほど形式手法の専門家によるコンサルティングを経て、成功している事例がほとんどであり [2]、日本での成功例の場合と教育方法・手順はほとんど同じである。

なお、日本フィッツの場合は開発チーム全員が VDM セミナーを受講したが、フェリカネットワークスは仕様記述フレームワーク作成予定者のみ VDM セミナーを受講した。

## 4.2 仕様記述

VDM++ を使って、仕様記述者がそれぞれ独自の仕様記述を行った場合、仕様の可読性や再利用性、保守性などが損なわれることが予想されたので、仕様記述のフレームワークを定め、仕様ライブラリを作成した。

### 4.2.1 仕様記述フレームワーク

仕様記述フレームワークは、仕様のどの部分に何を書くかを定めた規則である。どのように書くかを示すテンプレートとして提示したり、関連する仕様ライブラリを使用するよう指示する部分もあるが、多くは、日本語文で定めた規則で

ある。

仕様記述のためのフレームワークを作成した目的は以下の通りである。

- 仕様の可読性、再利用性、保守性の向上
- 仕様記述者の VDM 技術レベルと仕様の階層を対応させ、VDM 初級者でも仕様記述を行えるようにする
- DB インターフェースの隠蔽を行い、RDB による実装を行いやすくする (日本フィッツ事例 (3.1 節) の場合)
- 実装側フレームワークと対応させ、仕様とプログラムの対応関係が分かりやすいようにする (日本フィッツ事例の場合)

#### 4.2.1.1 仕様記述フレームワークの概要

##### 4.2.1.1.1 仕様記述の対象

日本フィッツ事例 (3.1 節) では、ユースケース・レベルの業務論理 (Business Logic) を仕様記述の対象とした。

フェリカネットワークス事例 (3.2 節) では、ファームウェアの API 仕様を仕様記述の対象とした。

##### 4.2.1.1.2 GUI の捨象

仕様記述フレームワーク作成では、GUI は捨象して記述しなかった。

これは、日本フィッツ事例の場合、サーバー側のユースケース・レベルの仕様を記述することが一番重要であり、クライアント側の GUI は、ユースケース・レベルの仕様記述とかなりの部分が重複するため、従来通りの画面仕様を中心とした仕様を使った。

GUI を除くクライアント側の仕様も、記述するための仕様記述フレームワークは試作したが、サーバー側のユースケース・レベル仕様との重複が多く、記述工数と比較して効果が薄いと判断し、記述しなかった。

フェリカネットワークス事例の場合、仕様記述対象がファームウェアの API 仕様であったため、そもそも、GUI 部分はなかった。

##### 4.2.1.1.3 仕様記述フレームワークの具体例

日本フィッツ事例 (3.1 節) の仕様記述フレームワークを、表 4.2 に示す。

表 4.2 仕様記述フレームワーク (日本フィッツ)

仕様記述の階層	充足する記述領域	必要な VDM 技量
テストケース	回帰テスト	初級
シナリオ	事後条件記述	初級
業務論理	業務アプリケーション	初級
要求辞書	業務知識兼用語辞書	中級
仮想 DB アクセス応用	DB 登録、抽出、削除、訂正	上級
仮想 DB アクセス基本	undo, old value アクセス, 差分	上級
基本レコード構造	必要データ	初級

この表で、シナリオ階層を作ったのは、業務論理で事後条件記述を試みた所、200 行～300 行の事後条件となってしまう、可読性や保守性に問題があると判断したためである。例えば、「口座を解説する」といった一つの業務論理に、「個

人顧客の口座を開設するシナリオ」「法人顧客の口座を開設するシナリオ」「口座開設が失敗するシナリオ」といったように、複数のシナリオを想定し、それぞれに事後条件を記述することで、各シナリオは 20 行～30 行程度の事後条件となり、読みやすくなった。したがって、シナリオ階層を設定した場合は、業務論理階層に事後条件は記述しない。業務論理とシナリオとテストケースの対応関係を図 4.1 に示す。

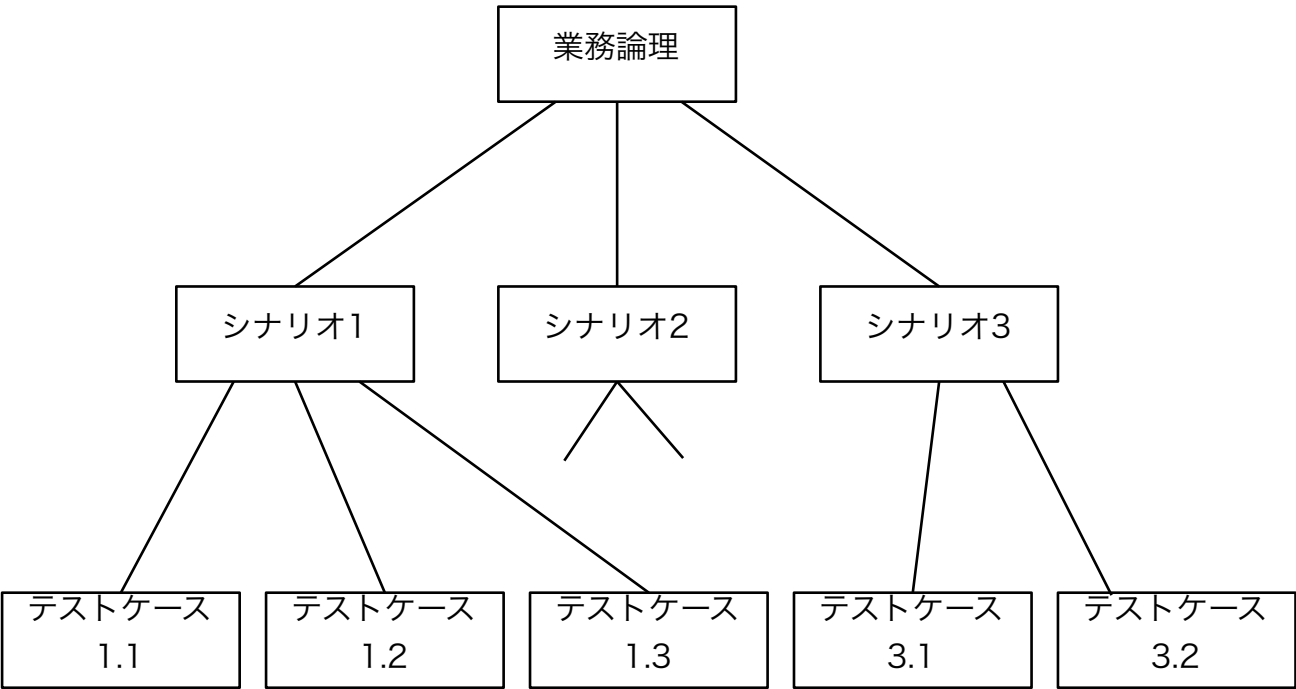


図 4.1 業務論理とシナリオとテストケースの対応関係

要求辞書階層は、業務論理階層で使う用語を定義している。この階層では、名詞に対応するクラス名や型名だけでなく、動詞に対応する関数 (function) や操作 (operation) も記述する。この階層に記述される仕様は、いわゆる暗黙知の明示的な仕様となるため、何を記述すべきかも含めて、記述は容易ではないが、この階層が記述できることにより、暗黙知を明示的な知識として共有することができるようになる。したがって、要求辞書階層の再利用性は、同じ問題領域であれば、かなり高い。要求辞書階層を記述する仕様記述者は、VDM++ の構文をある程度マスターしていなければならない。

9 章で紹介する運賃計算モデルの仕様記述フレームワークを、表 4.3 に示す。

表 4.3 仕様記述フレームワーク (運賃計算モデル)

仕様記述の階層	充足する記述領域	必要な VDM 技量
テストケース	回帰テスト	初級
業務論理	業務アプリケーション	初級
要求辞書	業務知識兼用語辞書	中級
ユーティリティ	共通機能	上級

この運賃モデルは、教育用に基本的な機能だけを持った仕様を記述しているため、仕様記述フレームワークの階層も深くない。

ユーティリティ階層は、鉄道ネットワークのグラフ構造から最短距離を計算するダイクストラ算法のために作った。当然ながら、ユーティリティ階層の再利用性は、要求辞書階層よりも更に高い。

図 4.2 に、運賃計算モデルのクラス図と、仕様記述フレームワーク階層の対応関係を示す。

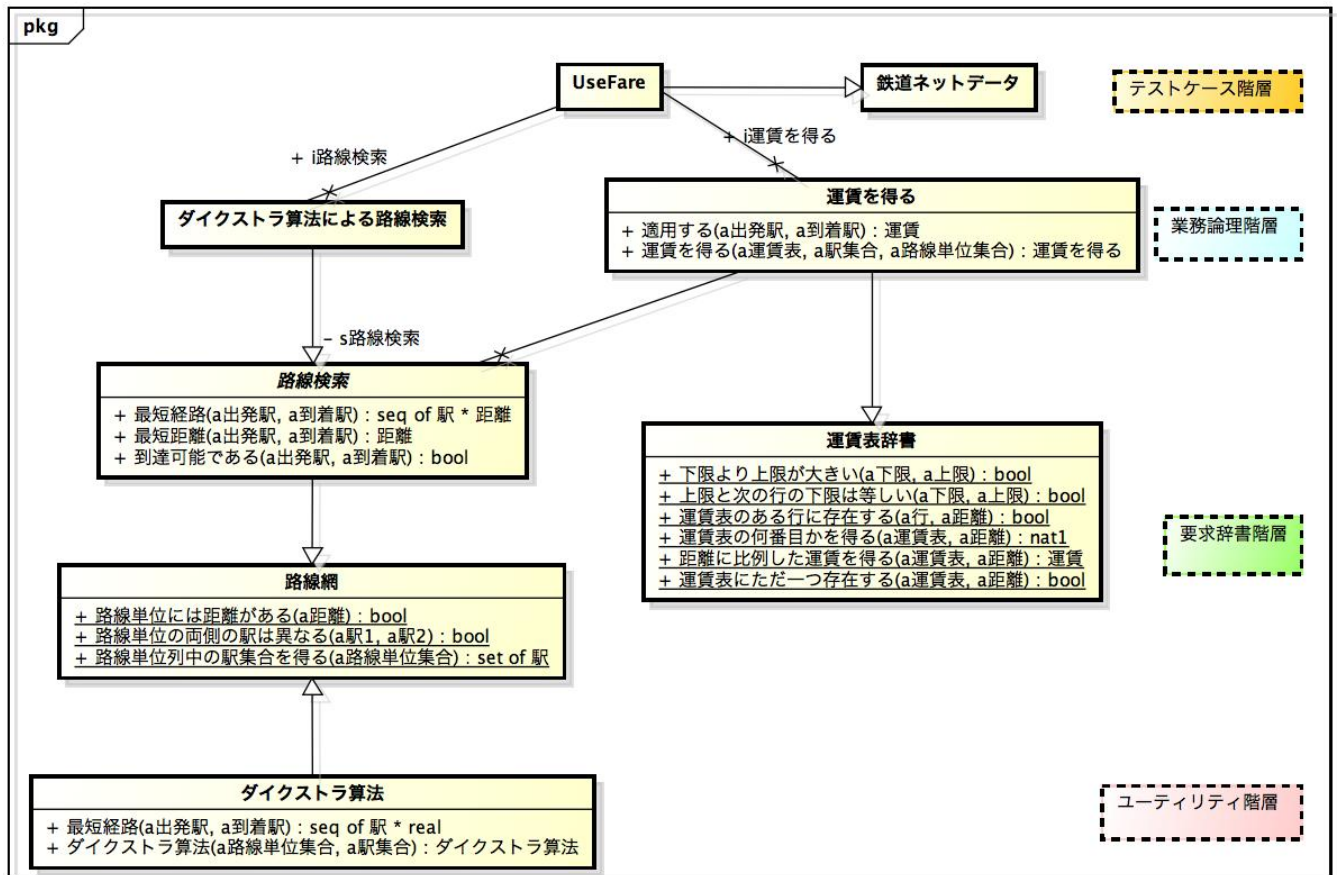


図 4.2 運賃計算モデルのクラス図

仕様記述フレームワークの各階層の具体例については、9 章を参照していただきたい。

#### 4.2.2 仕様ライブラリ

以下のライブラリ 3000 行ほどを、あらかじめ作成した。

- 文字、文字列ライブラリ

VDM++ は、特定の文字コードに依存しないよう、文字、文字列順という概念がないが、エンタープライズ系のシステムではソート順として多用されるため。

- 暦、日時、期間ライブラリ

エンタープライズ系のシステムでは、暦、日時、期間などに関する欠陥が多いため。

- 列ライブラリ

エンタープライズ系のシステムでは、合計、平均、ソートなどが多用されるため。

- 待ち行列、ハッシュ表ライブラリ

データ構造としてよく使われるため。

- 数値計算関連ライブラリ

実数の桁数、微分、ニュートン法などを使うことが予想されたため。

- その他、業務に関連したライブラリ

## 4.3 仕様テスト

仕様のテストは、以下のようにして行った<sup>\*1</sup>。

- VDMTools で、静的チェック
  - 構文チェック、型チェック  
型チェックは、当初予想していたより、はるかに多くの単純ミスを発見した。
  - 証明課題生成を行ってレビュー  
事前条件の記述忘れを、かなりの確率で発見できた。
- VDMTools で、動的チェック  
VDM++ インタープリタとデバッガで実行可能仕様<sup>\*2</sup>をテストした。
  - 動的に、不変条件・事前条件・事後条件をチェック
  - 組合せテスト機能を利用して、組合せテスト (4.3.1 参照) を実行
  - 回帰テスト支援ライブラリを改良して回帰テスト (4.3.2 参照) を実行  
回帰テストケースを作成するのはそれなりに変だが、仕様修正に対して非常に効果があった。
  - コードカバレッジ (網羅性検査) 機能で、テスト達成率を測定  
C0(命令) レベルで、85%(フェリカネットワークス事例 (3.2 節)) から 95%(日本フィッツ事例 (3.1 節)) を達成した。
  - 検出した矛盾点を、要求仕様に反映
- VDM 仕様に基づいて、ランダム評価ツールを作成し、数億件のテストケースを生成し、動的チェック (フェリカネットワークス事例) を行った。
  - 仕様エミュレータを内蔵
  - 実装に対し、ランダムにコマンドを送信し、応答が正しいか確認

### 4.3.1 組合せテスト

組合せテスト (Combinatorial Test) は、指定したテストパターンから、テストケースの組合せを生成し、テストする。テストでエラーになったものと同じパターンのテストケースは実行せず、テストの実行効率を上げる。

下記の例のように、trace 句の後に組合せテストケースを記述する。各組合せテストケースはラベル (今の場合 S1, S2 など) で始まり、VDM の構文の後ろに組合せの正規表現記述を行う。正規表現記述の説明は、VDM++ の注釈として示した。

```
class UseUniqueNumber
instance variables
sUN : 発番者 := new 発番者 ();

traces
```

<sup>\*1</sup> 詳細については、VDMTools ユーザマニュアル (VDM++) 2.0 <http://www.vdmttools.jp/modules/tinyd2/index.php?id=2> を参照のこと

<sup>\*2</sup> 陽仕様とも言う

```

S1 : let n in set {1,2,3,4} in sUN. 発番する (n){10,12}    -- 10 回から 12 回繰り返す

S2 : let n in set {2} in sUN. 発番する (n)+      -- 1 回から既定回数繰り返す

S3 : let n in set {3} in sUN. 発番する (n)*      -- 0 回から既定回数繰り返す

S4 : let n in set {4} in sUN. 発番する (n){10002}    --10002 回繰り返す

end UseUniqueNumber

```

図 4.3 に、組合せテスト実行後の Overture tool 画面例を示す<sup>\*3</sup>。

図 4.3 右上の CT Overview サブ・ウィンドウに組合せテストの実行結果が表示されている。

テスト S1 の最初のテスト Test 000001 は正常終了したので、緑色のチェックマークが付く。もちろん、正常終了したからといって、このテストケースが正しいとは限らない。不変条件、事前条件、事後条件が正しく記述されている場合は意味的にも正しいが、それ以外の場合は、単に実行が正常に終了したことを示すだけであって、意図した結果が得られているとは限らないので、注意が必要である。

次の、Test 000002 は実行時エラーがあったことを示して、赤色の×マークが付いている。このエラー原因は、右下の CT Test サブ・ウィンドウに示されていて、発番者クラス 17 行目第 3 カラムで ERROR 文が実行されたことが示されている。ERROR 文は、エラーがあったことを示して停止するだけなので、エラー原因は発番者クラスの VDM++ ソースを見て判断しなければならない<sup>\*4</sup>。

Test 000003 は、Test 000002 と同じ順序でテストケースを実行するため、必ず同じ所でエラーとなるので、それを検知した組合せテストケース機能が、実行せずにエラーを表示している。この印が漏斗のような灰色のアイコンである。組合せテストは、このようにして、数千件のテストケースを実行できるので、複雑な組合せ条件をテストするのに適している。

### 4.3.2 回帰テスト

回帰テスト (regression test) は、ソフトウェアが退化していないか確認するテストである。テストケースと期待する結果を一体にして記述し、実行して一致するか確認する。

VDM++ では、VDMUnit と呼ばれる回帰テスト用ライブラリを使用して回帰テストを行う。

回帰テスト実行用のクラスは 9.8.1 節の TestApp クラスを参照していただきたい。回帰テストケースは、9.9.1 節の回帰テストケースクラスと、そのサブクラス群を参照していただきたい。

回帰テストのノーマルケースの例は、9.9.2 節の TestCaseT0001 クラスを参照のこと。回帰テストのエラーケースの例は、9.9.3 節の TestCaseT0003 クラスを参照のこと。

## 4.4 プログラミング

VDM++ 仕様を見ながら、以下の作業を行った。

- 実装用フレームワークを作成
- VDM++ 仕様を見て、手作業でプログラム作成

<sup>\*3</sup> VDMTools は、組合せテスト実行結果をインタープリタ・ウィンドウにログとして表示する

<sup>\*4</sup> エラー原因を表示するように VDM++ 仕様を作成することは、例外処理文などを使えば可能である。ここでは、説明の単純化のために ERROR 文を使った

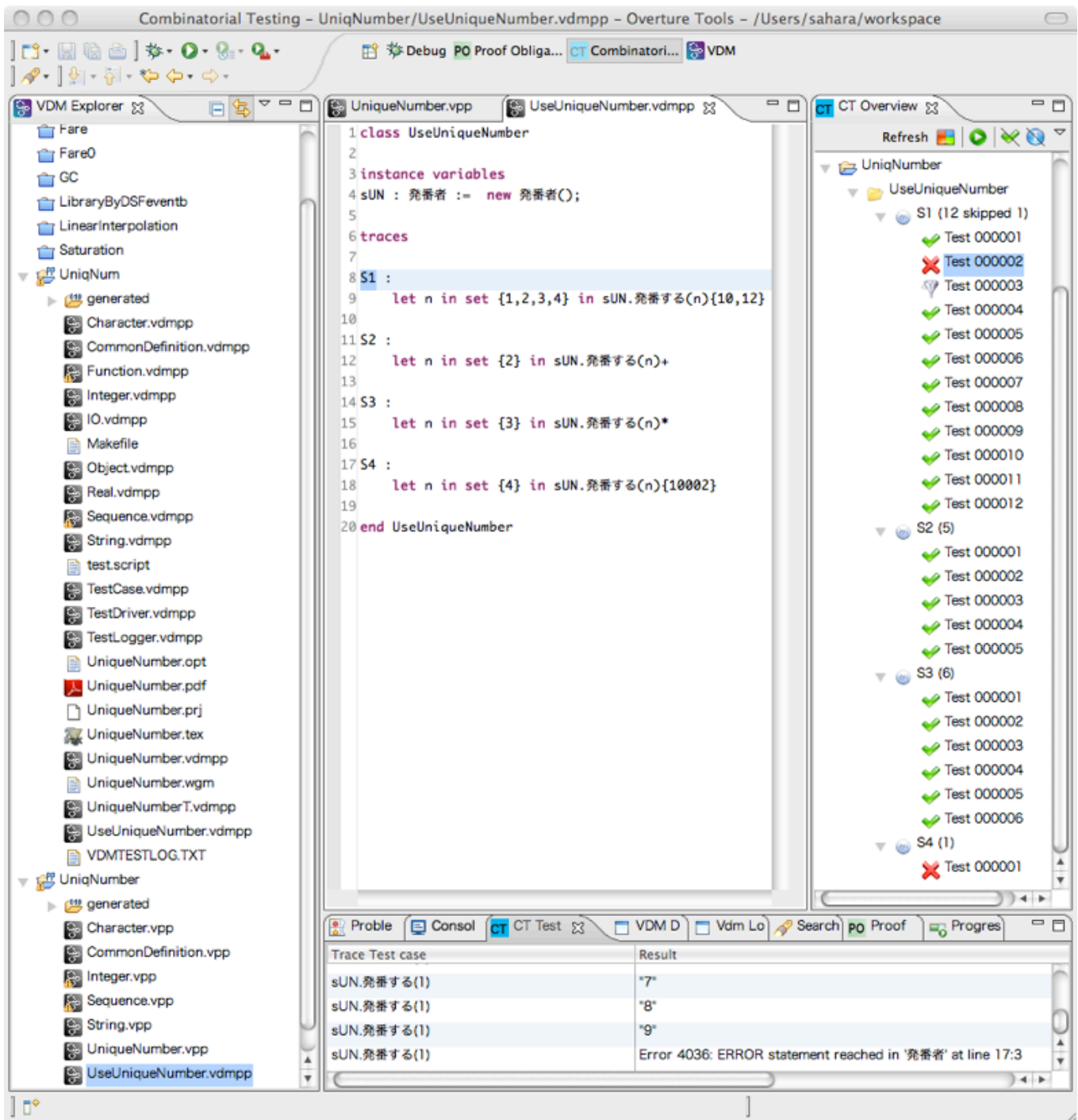


図 4.3 組合せテスト実行画面例

VDM++ から C++ 生成も可能だが

- 独自のミドルウェアに合わせて調整する時間的余裕が無かった (日本フィッツ事例)。  
生成されるプログラムの性能は、恐らく問題ないと予想したが\*5、プロジェクト工数の 5% 程度の作業に時間を取られたくなかったため、プログラム生成は使用しなかった。

\*5 SCSK では、現在、コード生成機能を大幅に改良し、エンタープライズシステム用 Java 生成フレームワークを試作中。

- 生成された C++ プログラムの性能が、十分でなかった (フェリカネットワークス事例)。
- 単純な画面生成プログラムを作成し、GUI 生成 (日本フィッツ事例)。
- オランダの花市場オークションシステム事例では 1999 年に C++ 生成を利用し、ソースの 90% を生成。残りは、手でコーディング。

## 4.5 プログラムのテスト

プログラムのテストは以下のように行った。

- 単体テスト
  - Purify <sup>\*6</sup> でソースの静的チェック  
メモリの破壊可能性などをチェックした。
  - C++ コードカバレッジ機能でテスト達成率測定
    - \* Excel データからテストケースを生成する簡単なプログラムを作成  
メモリの破壊可能性などをチェックした。
    - \* VDM++ テストケースを、C++ プログラムの Excel の単体テストケースに手作業で変換  
C0 レベルで 98% カバー
  - 仕様エミュレータを内蔵 (フェリカネットワークス事例)  
実装に対し、ランダムにコマンドを送信し、応答が正しいか確認 (フェリカネットワークス事例)
- 連結テスト  
要求定義担当者が、業務知識と Word で作成したユースケースから、テストケース生成 (日本フィッツ事例)
- 総合テスト
  - 評価環境を構築 (フェリカネットワークス事例)  
IC チップファームウェアと、実装と、形式仕様とが、同様の動作をすることを確認。
  - VDM 仕様に基づいて、ランダム評価ツールを作成し、数億件のテストケースを生成し、動的チェック (フェリカネットワークス事例)
  - 業務専門家全員で総合テストケース作成 (日本フィッツ事例)

## 4.6 開発管理

開発管理は、開発時点で実用性があると分かっていたソフトウェア工学技術とツールを使用して、以下のように行った。

- 版管理ツール cvs を使用  
VDM++ 仕様、ソースプログラム、Word などのドキュメント、テストケース、プログラム実行モジュールなど、全てのリソースを版管理した。
- Swiki(Squeak Wiki) サーバーを使って、共同作業を円滑にした (日本フィッツ事例)。
- ISO 9000 にしたがって、単体テスト以後の全欠陥を notes で記録し、追跡した (日本フィッツ事例)。

<sup>\*6</sup> 現在の製品名は、Rational PurifyPlus。 <http://www-06.ibm.com/software/jp/rational/products/purifyplus/highlights.html>

## 4.7 開発体制

### 4.7.1 日本フィッツの開発体制

仕様作成チーム、実装チーム、証券業務専門家チームの間で、相互チェックを行いながら開発を行った。

証券業務専門家チームは、ユースケースの最初の案を Word で作成し、VDM++ 仕様の回帰テスト結果を主としてチェックした。VDM++ ソースのチェックは、計算式部分など一部のみ行った。テスト工程では、連結テスト、総合テストのテストケース作成と結果の確認を行った。

仕様作成チームは、Word で記述されたユースケース相当の仕様をチェックしながら、VDM++ では、ほぼ完全にそれらを書き換えた。プログラムの単体テスト工程では、実装チームの作成したテストケースと VDM++ 仕様のテストケースを目で比較し、不足しているテストケースを追加した。

実装チームは、VDM++ 仕様を全て読み、手作業でプログラムに変換した。他チームの成果物のほとんどをレビューし、チェックした。

### 4.7.2 フェリカネットワークスの開発体制

仕様作成チーム、評価チーム、実装チームの間で、相互チェックを行いながら開発を行った。

各チームの相互チェックの様子を、図 4.4 に示す。

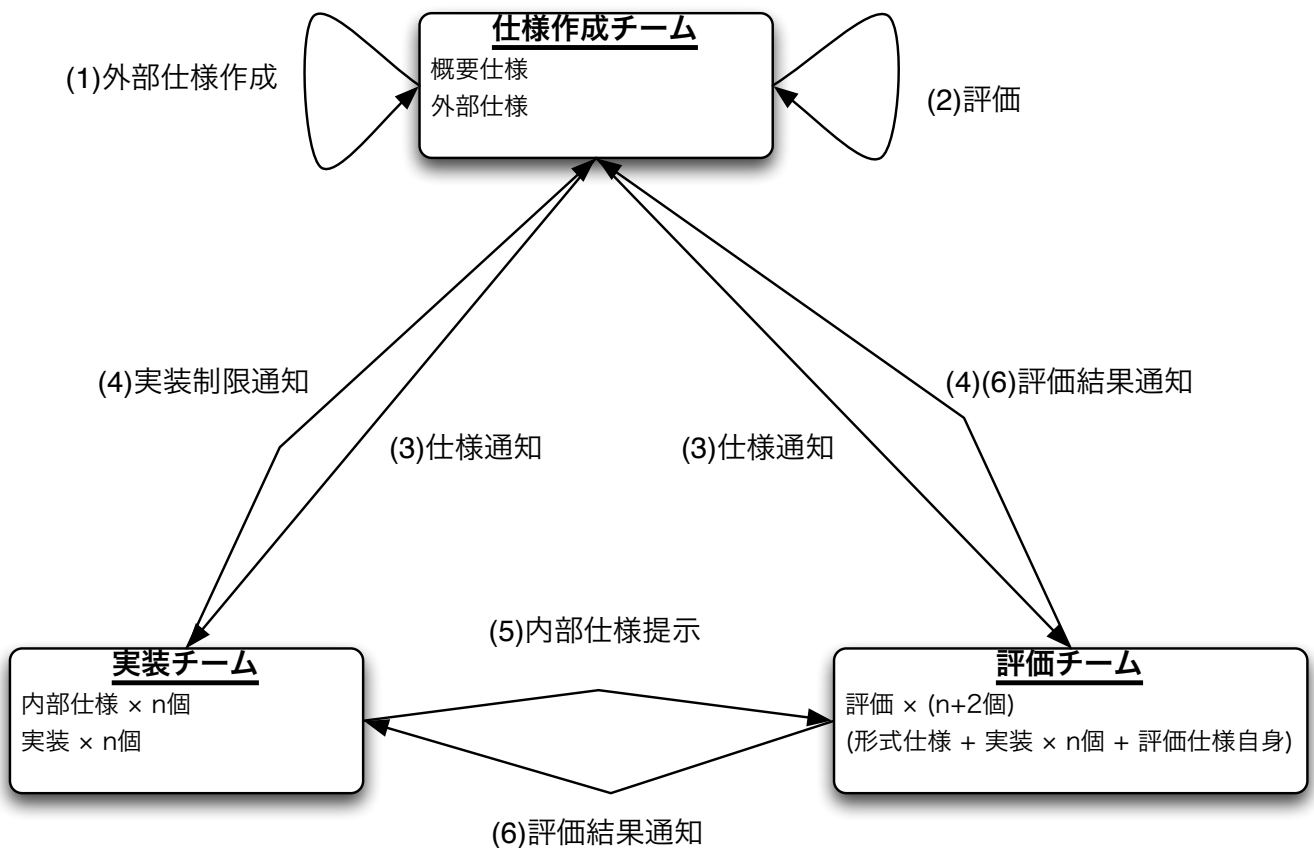


図 4.4 フェリカネットワークスの開発体制

1. 仕様作成チームは、日本語の概要仕様から VDM++ による外部仕様を作成し、
2. 回帰テストを行って、外部仕様の評価（正当性検証と仕様作成チームの観点からの妥当性確認）を行い、
3. 評価チームと実装チームに評価済みの仕様を通知する。
4. 評価チームは、評価チームの評価結果を仕様作成チームにフィードバックし、  
実装チームは、内部仕様と実装を通して発見した、実装上の制限を仕様作成チームにフィードバックする。
5. 実装チームは、作成した内部仕様を評価チームに提示し、
6. 評価チームは、VDM++ による形式仕様と、 $n$  個のチップの実装仕様と、評価仕様自身を総合して評価し、  
実装チームおよび仕様作成チームに評価結果をフィードバックする。



## 第 5 章

# 対象を如何にモデル化するか？

対象を如何にモデル化するかということは、こうやれば誰でも確実に良いモデルができるという一般的手順があるわけではない。しかし、数学やソフトウェア工学の歴史の中で培われた一般的原則と手順はある。本章では、このような一般的原則と手順を紹介する。何らかの仕様記述言語を前提としないと説明が難しいので、VDM++ 言語を前提とする。これらの原則と手順は、構造化分析・設計手法 (SA/SD) やオブジェクト指向分析・設計手法 (OOA/OOD) におけるモデル化手法と、より品質を高める形式手法の技術を除けば、さほど違いがあるわけではない。

## 5.1 モデル化の一般的手順

### 1. モデル化の範囲を決める。

例えば、「エラー処理の詳細を除くユースケースレベルの要求仕様を書く」とか、「アプリケーションレベルのエラー処理の詳細も含めて、外部仕様相当の設計仕様を記述する」といった方針を決める。過去に成功した形式手法使用プロジェクトの調査結果 [2] では、この範囲決めが重要で、かつ、この範囲内をすべて形式仕様で記述することが形式手法採用プロジェクト成功の鍵であることが分かっている。

### 2. 仕様を分割統治する。

階層化とモジュール化 (4.2.1 節と 5.2.4 節で紹介する仕様記述フレームワークを使う) を行い、ある階層やモジュールの変更余波が、他の階層やモジュールに及ばないようにモデルを構築する。これによって、モデルの保守性や再利用性が増し、可読性も増す。

### 3. 主に名詞から型を定義する。

属性の参照に留まらない機能を持つ型は、クラスとする。

### 4. システムが状態を持つ場合、その状態を定義する。

VDM++ の場合、状態はインスタンス変数として持つ。

### 5. 主に、述語 (動詞) から、関数・操作のインタフェース (シグネチャという) を記述する。

### 6. 関数・操作の事後条件・事前条件を記述する (陰仕様<sup>\*1</sup>作成)。

事後条件は、関数ないし操作が終わった状態を真 (true) を返す論理式で表す。したがって、関数ないし操作の「仕様」と言える。

事前条件は、引数の制約条件を論理式で記述し、仕様の責任を明確化するためのものである。すなわち、事前条件が偽となる引数に対して責任を負わないことを宣言している。操作の場合は、インスタンス変数の制約条件も記述することができる。

### 7. 静的検証を行う。

構文・型・証明課題チェックをツールを使用して行う。この段階で、多くの単純ミスが検出され、仕様作成者は「仕様の意味」に集中することができる。

### 8. 関数・操作の本体を記述する (陽仕様<sup>\*2</sup>にする)。

陰仕様は、静的検証以外の検証ができないため、事後条件を満たすか否かの動的検証を行うために、関数・操作の本体を記述する。

事後条件を満たす本体の記述は幾通りもあり得るし、本体の記述は一つの実装を表しているのも、本体は、仕様そのものではなく、事後条件で記述された「仕様」が正しいことを検証するためのものである、と言える。

### 9. モデルの正当性と妥当性を、動的に検証・確認する。

ツールのインタープリタ、デバッガを使用して、モデルの正当性 (verification) を検証し妥当性 (validation) を確認する。

<sup>\*1</sup> 関数・操作のインタフェースと、事前条件及び事後条件しか記述していない仕様を陰仕様と呼ぶ。陰仕様は、関数・操作の本体は記述していないが、「仕様」を記述していることになる。

<sup>\*2</sup> 関数・操作の本体が記述されていて実行可能な仕様を陽仕様と呼ぶ。

10. 要求仕様をレビューし、漏れがないか再検討する。

形式手法を使っている、仕様作成者やユーザーによるレビューは必要である。意味的な仕様の整合性は、人間しか気が付かないことが多いからである。

本章では、対象を如何にしてモデル化するかを、鉄道会社 A の特急券予約システムを例にして説明する。これは、乗客として特急券予約システムを使う際に起きたトラブルの原因を追求するために作成したモデルである。

## 5.2 特急券予約システムの例

### 5.2.1 問題点の要約

問題の発端は、クレジットカード会社の都合で、特急券予約システムに使用していたクレジットカードを変更せざるを得なくなり、特急券予約システムの設定を変更しようとしたところから始まった。

1. 客 (すなわち筆者) は、おサイフケータイ (鉄道会社 B) で特急券予約システム (鉄道会社 A) を使っていた。
2. 会社 C クレジットカードが廃止になり会社 D クレジットカードに変更して下さいとの連絡があった。
3. 鉄道会社 A サポートセンターに電話したところ、おサイフケータイの登録クレジットカードを変更すれば、2 日後には特急券予約システムが使えますとのことだった。
4. おサイフケータイの設定でクレジットカードを変更し、2 日後に使用しようとしたが、特急券の予約ができなかった。
5. 予約できなかったので 鉄道会社 A サポートセンターに電話  
客 「特急券を 2 枚予約しようとしたが、予約できなかったんですが？」  
鉄道会社 A 「カードを変更したら、新規に特急券予約システムを契約して下さい。」  
客 「新規にすると会費がかかるのでは？」  
鉄道会社 A 「はい。」  
客 「カード会社の都合で変更するのに、それはおかしいでしょう？」  
鉄道会社 A 「では、無料にします。」
6. 特急券に引き換えできなかったため 鉄道会社 A サポートセンターに電話  
客 「新しい予約はできたんですが、クレジットカード変更前に予約した特急券に引き換えようとしたらできないのですが？」  
鉄道会社 A 「予約会員証で引換できるようになったので、それで引き換えて下さい。暗証番号はクレジットカードの物を使って下さい。」
7. T 駅での問答  
客 「会員証で特急券に引き換えできないのですが？」  
鉄道会社 A 「会員証で引き換えできないですねー。おかしいな。クレジットカードでやってみましょう。駄目ですねー。」  
客 「古いクレジットカードでは駄目ですか？」  
鉄道会社 A 「あ、できましたね。はい、切符です。」

### 5.2.2 モデル化対象用語の選択

前節のやりとりは、電話でのやりとりをかなり (主として用語を中心として) 整理したものである。実際には、例えば、鉄道会社 A サポートセンターの担当者は「カード」という言い方をするのだが、それがクレジットカードの場合と、予約会員証の場合と、IC カードあるいは予約カードの場合があったので、個々の用語の正確な名前と意味を調べるだけでかなりの日数と時間を要した問答であった。

この問題で登場する用語は、インターネットなどで調べたところ以下に示すものがあつた。

- 鉄道会社 A、鉄道会社 B

- おサイフケータイ
- 特急券予約システム
- 予約会員証、IC カード、予約カード
- クレジットカード
  - 会社 C クレジットカード、会社 D クレジットカード
- 特急券

IC カードと予約カードは、結局、この問題とは直接は関係なかったので、モデルを単純化するため対象としないことにした。

「鉄道会社 B」をモデル化すると、さらに問題点が見つかると思ったが、今回の「特急券に引き換えできない問題」の解決には直接関係ないと思ったので、モデルを単純化するため対象としなかった。

このように、モデルを単純化し、一番問題となりそうなところからモデル化を始め、検証して問題がないことを確認し、必要があればモデルを拡張していくのがモデル化のコツである。

### 5.2.3 モデル化する機能の範囲選択

「鉄道会社 A」の「特急券予約システム」のうち、今回の問題に直接関係する以下の機能だけを VDM++<sup>[4]</sup> で要求仕様としてモデル化し、問題点を明確化することにした。<sup>\*3</sup>

- 予約する
- 特急券を得る
- クレジットカードを切り替える

したがって、モデル化する範囲に登場する主要な用語は以下だけである。

- おサイフケータイ
- 特急券予約システム
- 特急券予約
- 予約会員証
- クレジットカード
- 特急券

この用語と先程モデル化することにした機能を表す文（「予約する」、「特急券を得る」など）を併せて辞書とし、要求辞書と呼ぶこともある。

### 5.2.4 仕様記述フレームワークの設定

仕様を作成する際、仕様作成者各自が独自の形式で作成すると、保守性や再利用性に悪影響が出る。

そこで、仕様を階層分けしてカプセル化し、各階層で充足すべき記述領域を明確化して、保守性と再利用性の向上を図った。

このような仕様記述フレームワークを作っておくと、各階層毎の記述に必要な VDM++ 言語の知識も局所化でき、全員が VDM++ に精通していなくても、仕様記述が可能になることが、経験上分かっている。

本モデルの記述に際して使用する仕様記述フレームワークの階層は、表 5.1 のようになる。

表 5.1 で、業務論理階層の記述は、業務アプリケーションを記述する日本語識別子を使った構造化日本語仕様と見なす

<sup>\*3</sup> 仕様記述言語として VDM++ を採用したのは、開発現場に導入するのが最も容易な形式仕様記述言語と判断したからである。

表 5.1 仕様の階層 (特急券予約システム)

仕様記述の階層	充足する記述領域	備考	必要な VDM 技量
テスト	回帰テスト		初級
業務論理	業務アプリケーション	構造化日本語仕様	初級
要求辞書	業務知識兼用語辞書	ドメイン・オブジェクトも含まれる	中級
ユーティリティ	共通機能		上級～中級

こともできる。

日本語で擬似コードを使った仕様を書くことがあるが、この場合は逆に、厳密で明確な文法を持った VDM++ の仕様を日本語による擬似コードのように読めるよう工夫することで、読みやすさと厳密な記述による品質向上の両方を目指そうという訳である。

### 5.2.5 VDM++ モデル例

以下に、特急券予約システムを VDM++ で記述したモデルを示す。

このモデルは、要求仕様記述レベルであるが、アプリケーションレベルのエラー処理は記述せず、不変条件と事前条件および事後条件の記述を行う。すなわち、要求仕様記述・分析工程で最初に記述し検証する、最も抽象的なモデルである。

要求仕様記述・分析工程の最後では、アプリケーションレベルのエラー処理をすべて記述する、詳細な要求仕様を記述するが、通常、その規模は最初の抽象的な要求仕様より 10 倍ほどになるので、本ブックレットでは紹介できなかった。静的検証と動的検証の両方を使って正当性検証と妥当性確認を行う。動的検証は、仕様アニメーション、すなわち仕様実行によって行う。

特急券予約システムの、主要なクラスを記述したクラス図は図 5.1 のようになった。このクラス図は、最初からこのような形をしていたわけではなく、VDM++ でモデル化をしつつ、検証しながら、徐々に整理してこのような形となった。<sup>\*4</sup>

各クラスと、仕様階層の対応は、表 5.2 のようになる。

<sup>\*4</sup> このクラス図は、VDMTools から生成し、astah professional によって整形したものである。

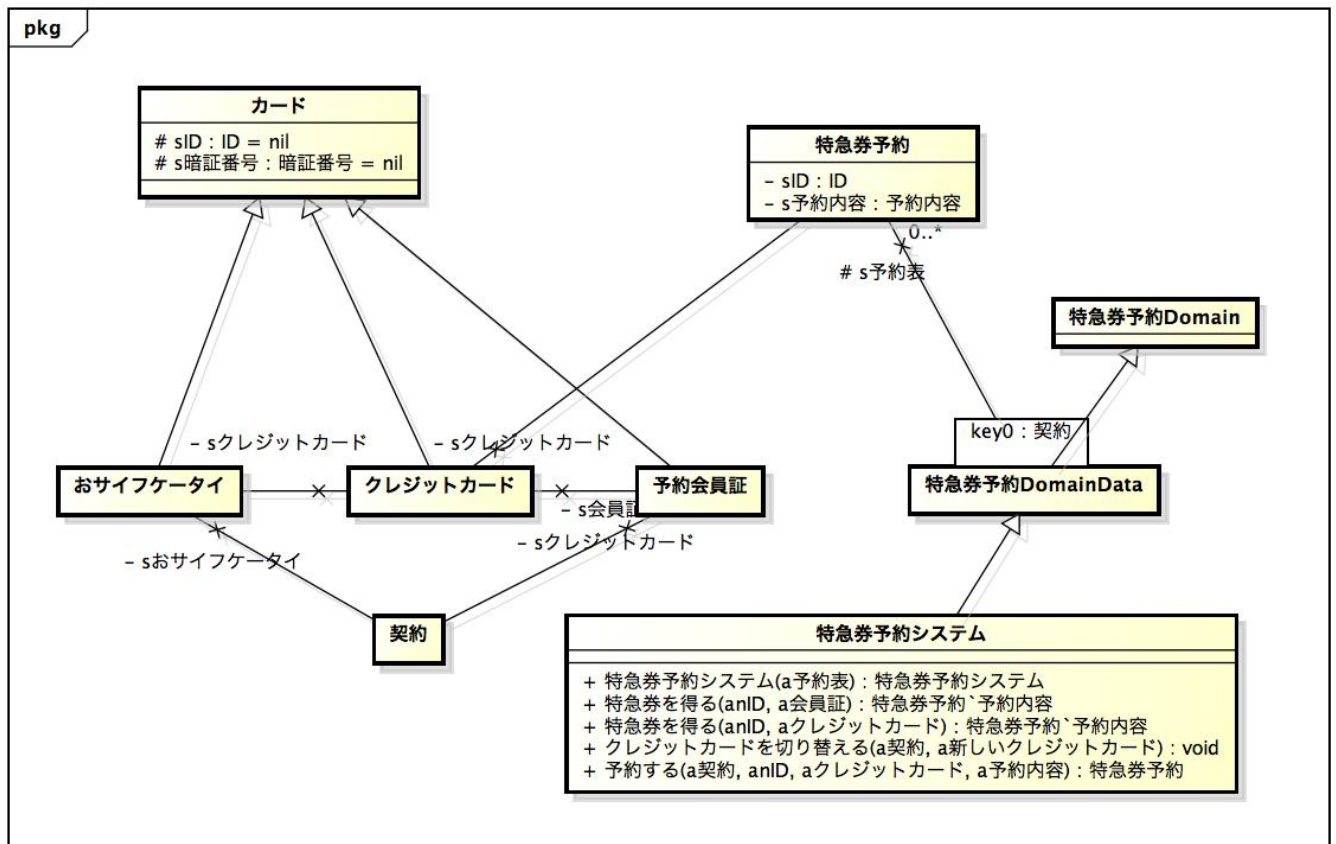


図 5.1 クラス図 (特急券予約システム)

表 5.2 特急券予約システムの仕様の階層

階層	クラス
テスト	TestApp
	回帰テストケースクラス (TestCaseT0001 など)
業務論理	特急券予約システム
要求辞書	特急券予約システム Domain
	特急券予約システム DomainData
	契約
	特急券予約システム
	クレジットカード
	予約会員証
	おサイフケータイ
ユーティリティ	回帰テストユーティリティなど

### 5.2.5.1 特急券予約システムクラス

最初に、業務論理階層の仕様である特急券予約システムクラスの仕様を記述するが、まず、クラスが果たすべき責任 (responsibility<sup>\*5</sup>) を記述することで、属性 (インスタンス変数) や、機能 (操作と関数) を決定する判断基準とする。このような基準で記述したクラスは、再利用性や保守性を満たす可能性が大きい。

#### 5.2.5.1.1 責任

特急券予約システムの主要機能を、要求辞書階層で定義されたクラスや型あるいは操作や関数を使用して、構造化日本語仕様と言える形の VDM++ で記述する。

#### 5.2.5.1.2 クラス定義と構成子定義

本クラスのスーパークラス「特急券予約 DomainData」は、特急券予約に関するドメインデータ (今の場合、インスタンス変数「s 予約表」) を定義している。

構成子「特急券予約システム」は、予約表をパラメータとして受け取り、特急券予約システムクラスのインスタンスを生成する。

```
.....
class
特急券予約システム is subclass of 特急券予約 DomainData
operations
public
  特急券予約システム : 予約表 ==> 特急券予約システム
  特急券予約システム (a 予約表) ==
    (   s 予約表 := a 予約表
      );
.....
```

#### 5.2.5.1.3 操作：予約する

「特急券予約 DomainData」のスーパークラスであり、本クラスのスーパークラスでもある、要求辞書階層の「特急券予約 Domain」クラスの関数「予約表を更新する」と「予約表に追加する」を呼び出して、予約を行う。

```
.....
public
  予約する : 契約 * ID * クレジットカード * 特急券予約 '予約内容 ==> 特急券予約
  予約する (a 契約, anID, a クレジットカード, a 予約内容) ==
    (   def w 特急券予約 = new 特急券予約 (anID, a クレジットカード, a 予約内容) in
      (   if 予約がある契約である (a 契約, s 予約表)
        then s 予約表 := 予約表を更新する (s 予約表, a 契約, w 特急券予約)
      );
.....
```

<sup>\*5</sup> 責務と訳すこともある。

```

        else s 予約表 := 予約表に追加する (s 予約表, a 契約, w 特急券予約);
        return w 特急券予約
    )
)
post if 予約がある契約である (a 契約, s 予約表~)
then 予約表が更新されている (s 予約表~, a 契約, RESULT, s 予約表)
else 予約表に追加されている (s 予約表~, a 契約, RESULT, s 予約表);
.....

```

#### 5.2.5.1.4 操作：特急券を得る

クレジットカードで特急券を得る。「特急券を得る」ことは、予約内容を返すということで抽象化した。

```

public
特急券を得る : ID * クレジットカード ==> 特急券予約 '予約内容
特急券を得る (anID, a クレジットカード) ==
(
    def w 予約 = 予約を得る (s 予約表, anID, a クレジットカード) in
        return w 予約.予約内容を得る ()
)
pre let w 予約 = 予約を得る (s 予約表, anID, a クレジットカード) in
    a クレジットカード = w 予約.クレジットカードを得る ();
.....

```

#### 5.2.5.1.5 操作：特急券を得る

予約会員証で特急券を得ることもできるので、前に記述したクレジットカードを使う「特急券を得る」操作のオーバーロード操作<sup>\*6</sup>を定義する。

```

public
特急券を得る : ID * 予約会員証 ==> 特急券予約 '予約内容
特急券を得る (anID, a 会員証) ==
(
    def w 予約 = 予約を得る (s 予約表, anID, a 会員証) in
        return w 予約.予約内容を得る ()
)
pre let w 予約 = 予約を得る (s 予約表, anID, a 会員証) in
    a 会員証.クレジットカードを得る () = w 予約.クレジットカードを得る ();
.....

```

<sup>\*6</sup> オーバーロード (overload) とは、同一の名前の関数や操作を複数定義し、使用時に文脈に応じて選択することで複数の動作を行わせる仕組みである。

## 5.2.5.1.6 操作：クレジットカードを切り替える

実システムでは、本操作を一般ユーザーが使うことはできず、鉄道会社 A サポートセンターの担当者だけが使えるが、本モデルでは、誰がある操作を行えるかのチェックは捨象して対象外としている。

.....

public

クレジットカードを切り替える：契約 \* クレジットカード ==> ()

クレジットカードを切り替える (a 契約, a 新しいクレジットカード) == a 契約.会員証を得る

( ) .クレジットカードを設定する(a 新しいクレジットカード)

end

特急券予約システム

.....

以下の表は、VDMTools が生成した VDM++ 仕様実行時の VDM++ の命令レベルで何 % 実行したかを表すカバレッジ情報である。

Test Suite : vdm.tc

Class : 特急券予約システム

Name	#Calls	Coverage
特急券予約システム ‘予約する	8	✓
特急券予約システム ‘特急券予約システム	4	✓
特急券予約システム ‘クレジットカードを切り替える	1	✓
特急券予約システム ‘特急券を得る	2	✓
特急券予約システム ‘特急券を得る	11	✓
Total Coverage		100%

命令レベルのコードカバレッジとは、各式や文を実行したか否かを表し、ループや分岐によって生じるすべての組合せが実行されたことは保証されないが、通常、95% 以上のカバレッジが達成されていれば、ほぼ信頼できる品質であることが分かっている。

Name カラムは関数ないし操作の名前、#Calls カラムは呼び出し回数、Coverage カラムは各関数あるいは操作内の命令の何 % を実行したかを表す。Coverage がチェックマークの場合は、100% 実行したことを表す。

## 5.2.5.2 共通定義クラス

## 5.2.5.2.1 責任

特急券予約モデルで共通に使用する定義を表す。

```
.....  
class  
共通定義  
types  
public ID = [token];  
public 暗証番号 = [token]  
operations  
public  
print : seq of char ==> ()  
print(a 文字列) ==  
  let - = new IO().echo(a 文字列) in  
  skip  
end  
共通定義  
.....
```

Test Suite : vdm.tc  
Class : 共通定義

Name	#Calls	Coverage
共通定義 'print	2	√
Total Coverage		100%

### 5.2.5.3 特急券予約 Domain クラス

#### 5.2.5.3.1 責任

特急券予約に関わる問題領域 (domain) の用語を定義する、要求辞書階層の仕様である。

#### 5.2.5.3.2 クラス定義

.....

```
class
```

特急券予約 Domain is subclass of 共通定義

.....

#### 5.2.5.3.3 型およびインスタンス変数定義：予約表

予約情報を持つ予約表は、契約から (個々の予約を表す) 特急券予約の集合への写像として定義した。

写像はキーとデータが対応した表と考えていただければよい。ただし、同じキーから異なるデータに対応してはいけないという制約がある

.....

```
types
```

```
public 予約表 = map 契約 to set of 特急券予約
```

.....

#### 5.2.5.3.4 関数：予約を得る

予約 ID を指定して、予約を得る。<sup>\*7</sup>

.....

```
functions
```

```
public
```

```
予約を得る : 予約表 * ID -> 特急券予約
```

```
予約を得る (a 予約表, anID) ==
```

```
  (let w 予約 in set dunion rng a 予約表 be st w 予約.ID を得る () = anID in
   w 予約)
```

```
pre exists w 予約 in set dunion rng a 予約表 &
  w 予約.ID を得る () = anID ;
```

.....

#### 5.2.5.3.5 関数：予約を得る

クレジットカードを指定して、予約を得る。

.....

---

<sup>\*7</sup> 以下の VDM++ ソースで赤く表示されることがあるのは、この部分が実行されていないことを示している。

```
public
```

```
予約を得る : 予約表 * ID * クレジットカード -> 特急券予約
```

```
予約を得る (a 予約表, anID, a クレジットカード) ==
```

```
  (let w 予約 in set dunion rng a 予約表 be st
    w 予約.クレジットカードを得る () = a クレジットカード and
    w 予約.ID を得る () = anID in
  w 予約)
```

```
pre exists w 予約 in set dunion rng a 予約表 &
  w 予約.クレジットカードを得る () = a クレジットカード and
  w 予約.ID を得る () = anID ;
```

.....

#### 5.2.5.3.6 関数：予約を得る

予約会員証を指定して、予約を得る。

.....

```
public
```

```
予約を得る : 予約表 * ID * 予約会員証 -> 特急券予約
```

```
予約を得る (a 予約表, anID, a 予約会員証) ==
```

```
  (let w 予約 in set dunion rng a 予約表 be st
    w 予約.クレジットカードを得る () = a 予約会員証.クレジットカードを得る () and
    w 予約.ID を得る () = anID in
  w 予約)
```

```
pre exists w 予約 in set dunion rng a 予約表 &
  w 予約.クレジットカードを得る () = a 予約会員証.クレジットカードを得る () and
  w 予約.ID を得る () = anID ;
```

.....

#### 5.2.5.3.7 関数：予約表を更新する

契約と特急券予約を指定して、予約表を更新する。

.....

```
public
```

```
予約表を更新する : 予約表 * 契約 * 特急券予約 -> 予約表
```

```
予約表を更新する (a 予約表, a 契約, a 特急券予約) ==
```

```
  a 予約表 ++ {a 契約 |-> 予約集合に追加する (a 予約表 (a 契約), {a 特急券予約 })}
```

```
pre a 契約 in set dom a 予約表
```

```
post 予約表が更新されている (a 予約表, a 契約, a 特急券予約, RESULT) ;
```

```
public
```

```
予約表が更新されている : 予約表 * 契約 * 特急券予約 * 予約表 +> bool
```

```
予約表が更新されている (a 予約表, a 契約, a 特急券予約, a 更新後予約表) ==
```

```
  a 更新後予約表 = a 予約表 ++ {a 契約 |-> 予約集合に追加する (a 予約表 (a 契約), {a 特急券予約 })};
```

## 5.2.5.3.8 関数：予約表に追加する

契約と特急券予約を指定して、予約表に追加する。

```

public
予約表に追加する：予約表 * 契約 * 特急券予約 -> 予約表
予約表に追加する (a 予約表, a 契約, a 特急券予約) ==
  a 予約表 munion {a 契約 |-> {a 特急券予約 }}
pre not 予約がある契約である (a 契約, a 予約表) and
  forall w 契約 1 in set dom a 予約表, w 契約 2 in set dom {a 契約 |-> {a 特急券予約 }} &
    w 契約 1 = w 契約 2 => a 予約表 (w 契約 1) = {a 契約 |-> {a 特急券予約 }} (w 契約 2)
post 予約表に追加されている (a 予約表, a 契約, a 特急券予約, RESULT) ;
public
予約表に追加されている：予約表 * 契約 * 特急券予約 * 予約表 +> bool
予約表に追加されている (a 予約表, a 契約, a 特急券予約, a 更新後予約表) ==
  a 更新後予約表 = a 予約表 munion {a 契約 |-> {a 特急券予約 }};

```

## 5.2.5.3.9 関数：予約がある契約である

予約がある契約が判定する。

```

public
予約がある契約である：契約 * 予約表 +> bool
予約がある契約である (a 契約, a 予約表) ==
  a 契約 in set dom a 予約表;

```

## 5.2.5.3.10 関数：予約集合に追加する

既存予約集合と新規予約集合を指定して、予約集合に追加する。

```

public
予約集合に追加する：set of 特急券予約 * set of 特急券予約 +> set of 特急券予約
予約集合に追加する (a 既存予約集合, a 新規予約集合) ==
  a 既存予約集合 union a 新規予約集合
end
特急券予約 Domain

```

Test Suite : vdm.tc

Class : 特急券予約 Domain

Name	#Calls	Coverage
特急券予約 Domain‘予約表に追加する	4	65%
特急券予約 Domain‘予約表を更新する	4	✓
特急券予約 Domain‘予約集合に追加する	12	✓
特急券予約 Domain‘予約がある契約である	20	✓
特急券予約 Domain‘予約表が更新されている	8	✓
特急券予約 Domain‘予約表に追加されている	8	✓
特急券予約 Domain‘予約を得る	0	0%
特急券予約 Domain‘予約を得る	3	✓
特急券予約 Domain‘予約を得る	21	✓
Total Coverage		80%

## 5.2.5.4 特急券予約 DomainData クラス

## 5.2.5.4.1 責任

特急券予約に関わる問題領域 (domain) の、状態を持つデータを表す。

## 5.2.5.4.2 クラス定義

```

.....
class
特急券予約 DomainData is subclass of 特急券予約 Domain
instance variables
protected s 予約表 : 予約表 := { |-> };
.....

```

## 5.2.5.4.3 操作：予約集合を得る

契約を指定して、予約集合を得る。

```

.....
operations
public
予約集合を得る : 契約 ==> set of 特急券予約
予約集合を得る (a 契約) ==
  if dom s 予約表 = {}
  then return {}
  else return s 予約表 (a 契約)
pre dom s 予約表 <> {} => a 契約 in set dom s 予約表
end
特急券予約 DomainData
.....

```

```

Test Suite :    vdm.tc
Class :        特急券予約 DomainData

```

Name	#Calls	Coverage
特急券予約 DomainData‘予約集合を得る	4	60%
Total Coverage		60%

## 5.2.5.5 特急券予約クラス

## 5.2.5.5.1 責任

特急券予約 1 件を表す、要求辞書階層の仕様である。

## 5.2.5.5.2 クラス定義

.....  
class

特急券予約 is subclass of 共通定義  
.....

## 5.2.5.5.3 型定義：予約内容

予約内容の型の詳細は、今回の問題に関わりがなく、仕様を詳細化していった時のみ必要なので、こういった場合の定石として token 型<sup>\*8</sup>で定義する。

.....  
types

public 予約内容 = token  
.....

## 5.2.5.5.4 インスタンス変数定義：ID, クレジットカード, 予約内容

.....  
instance variables

sID : ID;

s クレジットカード : クレジットカード;

s 予約内容 : 予約内容;  
.....

## 5.2.5.5.5 構成子

.....  
operations

public

---

<sup>\*8</sup> token 型は VDM++ に存在する、抽象化のために使用する型である。システムの本質と関係ない詳細な記述を回避し、短時間で主要な仕様を記述し検証するために使う。

特急券予約 : ID \* クレジットカード \* 特急券予約 ‘予約内容 ==> 特急券予約

特急券予約 (anID, a クレジットカード, a 予約内容) == `atomic`

```
( sID := anID;
  s クレジットカード := a クレジットカード;
  s 予約内容 := a 予約内容
);
```

#### 5.2.5.5.6 アクセッサー

```
public
```

ID を得る : () ==> ID

ID を得る () ==

```
  return sID;
```

```
public
```

クレジットカードを得る : () ==> クレジットカード

クレジットカードを得る () ==

```
  return s クレジットカード;
```

```
public
```

予約内容を得る : () ==> 予約内容

予約内容を得る () ==

```
  return s 予約内容
```

```
end
```

特急券予約

Test Suite : vdm.tc

Class : 特急券予約

Name	#Calls	Coverage
特急券予約 ‘ID を得る	46	✓
特急券予約 ‘特急券予約	8	✓
特急券予約 ‘予約内容を得る	11	✓
特急券予約 ‘クレジットカードを得る	67	✓
Total Coverage		100%

#### 5.2.5.6 契約クラス

##### 5.2.5.6.1 責任

特急券予約の客と鉄道会社 A の契約を表すドメイン・オブジェクトで、要求辞書階層の仕様である。

##### 5.2.5.6.2 クラス定義

```
.....  
class
```

```
契約 is subclass of 共通定義  
.....
```

##### 5.2.5.6.3 インスタンス変数定義：おサイフケータイ, 予約会員証

```
.....  
instance variables
```

```
s おサイフケータイ：おサイフケータイ；
```

```
s 会員証：予約会員証；  
.....
```

##### 5.2.5.6.4 構成子

実際には、契約が結ばれてから予約会員証が送られてくるまでの時差があるが、簡単化のため無視する。

```
.....  
operations
```

```
public
```

```
契約：おサイフケータイ * 予約会員証 ==> 契約
```

```
契約(a おサイフケータイ, a 会員証) == atomic
```

```
    ( s おサイフケータイ := a おサイフケータイ；
```

```
      s 会員証 := a 会員証
```

```
    )；  
.....
```

##### 5.2.5.6.5 アクセッサ

```
.....  
public
```

```
会員証を得る：() ==> 予約会員証
```

```
会員証を得る() ==
```

```
    return s 会員証
```

end

契約

.....  
Test Suite : vdm.tc

Class : 契約

Name	#Calls	Coverage
契約 ‘契約	4	✓
契約 ‘会員証を得る	5	✓
Total Coverage		100%

## 5.2.5.7 カードクラス

## 5.2.5.7.1 責任

カードの抽象クラスで、要求辞書階層の仕様である。

## 5.2.5.7.2 クラス定義

```
class
```

```
カード is subclass of 共通定義
```

## 5.2.5.7.3 インスタンス変数定義：ID, 暗証番号

```
instance variables
```

```
protected sID : ID := nil;
```

```
protected s 暗証番号 : 暗証番号 := nil;
```

```
end
```

カード

Test Suite : vdm.tc

Class : カード

Name	#Calls	Coverage
Total Coverage		undefined

## 5.2.5.8 クレジットカードクラス

## 5.2.5.8.1 責任

クレジットカードを表すドメイン・オブジェクトで、要求辞書階層の仕様である。

## 5.2.5.8.2 クラス定義

```
.....  
class
```

```
クレジットカード is subclass of カード  
.....
```

## 5.2.5.8.3 構成子

```
.....  
operations
```

```
public
```

```
クレジットカード : ID * 暗証番号 ==> クレジットカード
```

```
クレジットカード (anID, a 暗証番号) == atomic
```

```
  ( sID := anID;
```

```
    s 暗証番号 := a 暗証番号
```

```
  )
```

```
end
```

```
クレジットカード  
.....
```

```
Test Suite :    vdm.tc
```

```
Class :        クレジットカード
```

Name	#Calls	Coverage
クレジットカード ‘クレジットカード	7	√
Total Coverage		100%

## 5.2.5.9 予約会員証

## 5.2.5.9.1 責任

予約会員証を表すドメイン・オブジェクトで、要求辞書階層の仕様である。

## 5.2.5.9.2 クラス定義

```
.....
class
```

```
予約会員証 is subclass of カード
.....
```

## 5.2.5.9.3 インスタンス変数定義：クレジットカード

```
.....
instance variables
```

```
s クレジットカード : クレジットカード;
.....
```

## 5.2.5.9.4 構成子

```
.....
operations
```

```
public
```

```
予約会員証 : ID * 暗証番号 * クレジットカード ==> 予約会員証
```

```
予約会員証 (anID, a 暗証番号, a クレジットカード) == atomic
```

```
( sID := anID;
```

```
  s 暗証番号 := a 暗証番号;
```

```
  s クレジットカード := a クレジットカード
```

```
);
.....
```

## 5.2.5.9.5 アクセッサ

```
.....
public
```

```
クレジットカードを得る : () ==> クレジットカード
```

```
クレジットカードを得る () ==
```

```
  return s クレジットカード;
```

```
public
```

クレジットカードを設定する : クレジットカード ==> ()

クレジットカードを設定する (a クレジットカード) ==

s クレジットカード := a クレジットカード

end

予約会員証

.....  
Test Suite : vdm.tc

Class : 予約会員証

Name	#Calls	Coverage
予約会員証 ‘予約会員証	4	✓
予約会員証 ‘クレジットカードを得る	11	✓
予約会員証 ‘クレジットカードを設定する	1	✓
Total Coverage		100%

## 5.2.5.10 おサイフケータイクラス

## 5.2.5.10.1 責任

おサイフケータイに搭載されているスマートカードを表すドメイン・オブジェクトで、要求辞書階層の仕様である。

## 5.2.5.10.2 クラス定義

```
.....
class
```

```
おサイフケータイ is subclass of カード
.....
```

## 5.2.5.10.3 インスタンス変数定義：クレジットカード

```
.....
instance variables
```

```
s クレジットカード : クレジットカード;
.....
```

## 5.2.5.10.4 構成子

本モデルでは、単純化のため、おサイフケータイで改札を通る処理は対象としていないため、暗証番号設定は省略した。

```
.....
operations
```

```
public
```

```
おサイフケータイ : ID * クレジットカード ==> おサイフケータイ
```

```
おサイフケータイ (anID, a クレジットカード) == atomic
```

```
( sID := anID;
```

```
  s クレジットカード := a クレジットカード
```

```
);
.....
```

## 5.2.5.10.5 アクセッサ

```
.....
public
```

```
クレジットカードを得る : () ==> クレジットカード
```

```
クレジットカードを得る () ==
```

```
  return s クレジットカード
```

```
end
```

おサイフケータイ

Test Suite : vdm.tc

Class : おサイフケータイ

Name	#Calls	Coverage
おサイフケータイ 'おサイフケータイ	7	✓
おサイフケータイ 'クレジットカードを得る	7	✓
Total Coverage		100%

## 5.2.5.11 TestApp クラス

## 5.2.5.11.1 責任

特急券予約モデルの回帰テスト (5.2.6.2.1 クラス参照) を行う、テスト階層の仕様である。

## 5.2.5.11.2 クラス定義

```
.....
class
TestApp
.....
```

## 5.2.5.11.3 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
run : () ==> ()
run () ==
(   dcl ts : TestSuite := new TestSuite ("特急券予約モデルの回帰テスト \n"),
    tr : TestResult := new TestResult ();
    tr.addListener(new PrintTestListener());
    ts.addTest(new TestCaseT0001 ("TestCaseT0001 通常の予約成功 \n"));
    ts.addTest(new TestCaseT0002 ("TestCaseT0002 クレジットカードを切り替えたときの予約成
功 \n"));
    ts.addTest(new TestCaseT0003 ("TestCaseT0003 クレジットカードを切り替え、予約からクレジットカ
ードで特急券を得るが、予約と異なるクレジットカードでは失敗 \n"));
    ts.addTest(new TestCaseT0004 ("TestCaseT0004 クレジットカードを切り替え、予約から予約会員証で
特急券を得るが失敗 \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	85%
<b>Total Coverage</b>		<b>85%</b>

## 5.2.5.12 TestCaseT0001 クラス

## 5.2.5.12.1 責任

通常の予約をテストする、テスト階層の仕様である。

```

.....
class
TestCaseT0001 is subclass of TestCase, 共通定義
operations
public
TestCaseT0001 : seq of char ==> TestCaseT0001
TestCaseT0001 (name) ==
    setName(name);
public
test01 : () ==> ()
test01 () ==
    ( def w クレジットカード = new クレジットカード (mk_token (<クレジット ID01>), mk_token (<クレジット PW01>));
      w おサイフケータイ = new おサイフケータイ (mk_token (<おサイフケータイ ID01>), w クレジットカード);
      w 会員証 = new 予約会員証 (mk_token (<会員証 ID01>), mk_token (<会員証 PW01>), w クレジットカード);
      w 契約 = new 契約 (w おサイフケータイ, w 会員証);
      w システム = new 特急券予約システム ({ |-> }) in
        ( assertTrue("test01 契約に失敗",
            w システム.予約集合を得る (w 契約) = {} and
            w 契約.会員証を得る () = w 会員証);
          def w おサイフケータイクレジットカード = w おサイフケータイ.クレジットカードを得る () in
            ( def -- w システム.予約する (w 契約, mk_token (<予約 ID01>), w おサイフケータイクレジットカード, mk_token (<最初の予約内容>)) in
                assertTrue("test01 予約に失敗",
                    w システム.特急券を得る (mk_token (<予約 ID01>), w クレジットカード) = mk_token (<最初の予約内容>));
                def -- w システム.予約する (w 契約, mk_token (<予約 ID02>), w おサイフケータイクレジットカード, mk_token (<次の予約内容>)) in
                    assertTrue("test01 2 回目の予約に失敗",
                        w システム.特急券を得る (mk_token (<予約 ID02>), w クレジットカード) = mk_token (<次の予約内容>))
                )
            )
          )
        )
      )
    )
end

```

TestCaseT0001

Test Suite : vdm.tc

Class : TestCaseT0001

Name	#Calls	Coverage
TestCaseT0001'test01	1	✓
TestCaseT0001'TestCaseT0001	1	✓
Total Coverage		100%

### 5.2.5.13 TestCaseT0002 クラス

#### 5.2.5.13.1 責任

クレジットカードを切り替えたときの予約をテストする、テスト階層の仕様である。。

```
class
```

```
TestCaseT0002 is subclass of TestCase, 共通定義
```

```
operations
```

```
public
```

```
TestCaseT0002 : seq of char ==> TestCaseT0002
```

```
TestCaseT0002 (name) ==
```

```
    setName(name);
```

```
public
```

```
test01 : () ==> ()
```

```
test01 () ==
```

```
    ( def w クレジットカード = new クレジットカード (mk_token (<クレジット ID01>), mk_token (<クレジット PW01>));
```

```
        w おサイフケータイ = new おサイフケータイ (mk_token (<おサイフケータイ ID01>), w クレジットカード);
```

```
        w 会員証 = new 予約会員証 (mk_token (<会員証 ID01>), mk_token (<会員証 PW01>), w クレジットカード);
```

```
        w 契約 = new 契約 (w おサイフケータイ, w 会員証);
```

```

w システム = new 特急券予約システム ({ l-> }) in
(
  assertTrue("test01 契約に失敗",
    w システム.予約集合を得る(w 契約) = {} and
    w 契約.会員証を得る() = w 会員証);
  def w 古いクレジットカード = w おサイフケータイ.クレジットカードを得る() in
    (
      def - = w システム.予約する(w 契約, mk.token(<予約 ID01>), w 古いクレジットカード
, mk.token(<最初の予約内容>)) in
        assertTrue("test01 予約に失敗",
          w システム.特急券を得る(mk.token(<予約 ID01>), w クレジットカード) = mk.token(<最
初の予約内容>));
        def w2 クレジットカード = new クレジットカード(mk.token(<クレジットカード
ド ID02>), mk.token(<クレジットカード PW02>));
        w2 おサイフケータイ = new おサイフケータイ(mk.token(<おサイフケータイ ID01>), w2 ク
レジットカード);
        w 変更後のクレジットカード = w2 おサイフケータイ.クレジットカードを得る() in
          (
            def - = w システム.予約する(w 契約, mk.token(<予約 ID02>), w 変更後のクレジットカード
, mk.token(<次の予約内容>)) in
              assertTrue("test01 2 回目の予約に失敗",
                w システム.特急券を得る(mk.token(<予約 ID02>), w2 クレジットカード
) = mk.token(<次の予約内容>))
            )
          )
        )
      )
    )
  )
end

```

TestCaseT0002

Test Suite : vdm.tc

Class : TestCaseT0002

Name	#Calls	Coverage
TestCaseT0002'test01	1	✓
TestCaseT0002'TestCaseT0002	1	✓
<b>Total Coverage</b>		<b>100%</b>

#### 5.2.5.14 TestCaseT0003 クラス

##### 5.2.5.14.1 責任

クレジットカードを切り替え、予約した特急券を得る場合をテストする、テスト階層の仕様である。古いクレジットカードでは「Error 58: 事前条件の評価結果が false です」という実行時エラーが発生し、変更後のクレジットカードでは、特急券を得ることができる。

```

class
TestCaseT0003 is subclass of TestCase, 共通定義
operations
public
TestCaseT0003 : seq of char ==> TestCaseT0003
TestCaseT0003 (name) ==
    setName(name);
public
test01 : () ==> ()
test01 () ==
    ( def w クレジットカード = new クレジットカード (mk_token (<クレジット ID01>), mk_token (<クレジット PW01>));
      w おサイフケータイ = new おサイフケータイ (mk_token (<おサイフケータイ ID01>), w クレジットカード);
      w 会員証 = new 予約会員証 (mk_token (<会員証 ID01>), mk_token (<会員証 PW01>), w クレジットカード);
      w 契約 = new 契約 (w おサイフケータイ, w 会員証);
      w システム = new 特急券予約システム ({ |-> }) in
      ( assertTrue("test01 契約に失敗",
        w システム.予約集合を得る (w 契約) = {} and
        w 契約.会員証を得る () = w 会員証);
        def w 古いクレジットカード = w おサイフケータイ.クレジットカードを得る () in
        ( def -- w システム.予約する (w 契約, mk_token (<予約 ID01>), w 古いクレジットカード
, mk_token (<最初の予約内容>)) in
            assertTrue("test01 予約に失敗",
              w システム.特急券を得る (mk_token (<予約 ID01>), w 古いクレジットカード
) = mk_token (<最初の予約内容>));
              def w2 クレジットカード = new クレジットカード (mk_token (<クレジットカード ID02>), mk_token (<クレジットカード PW02>));
              w2 おサイフケータイ = new おサイフケータイ (mk_token (<おサイフケータイ ID01>), w2 クレジットカード);

```

```

w 変更後のクレジットカード = w2 おサイフケータイ.クレジットカードを得る() in
( def - = w システム.予約する(w 契約, mk_token(<予約 ID02>), w 変更後のクレジットカード
, mk_token(<次の予約内容>)) in
    assertTrue("test01 2 回目の予約に失敗",
        w システム.特急券を得る(mk_token(<予約 ID02>), w 変更後のクレジットカード
) = mk_token(<次の予約内容>));
    def w2 予約内容 = w システム.特急券を得る(mk_token(<予約 ID02>), w 変更後のクレジッ
トカード) in
        assertTrue("test01 変更後のクレジットカードで特急券を得ることに失敗",
            w2 予約内容 = mk_token(<次の予約内容>));
        def w2 予約内容 = w システム.特急券を得る(mk_token(<予約 ID01>), w 会員証) in
            assertTrue("test01 予約会員証で特急券を得ることに失敗",
                w2 予約内容 = mk_token(<最初の予約内容>));
        trap <RuntimeError>
        with print("\ttest01 テスト意図通り、古いクレジットカードで特急券を得ることに失
敗 \n") in
            def w3 予約内容 = w システム.特急券を得る(mk_token(<予約 ID02>), w 古いクレジッ
トカード) in
                assertTrue("\ttest01 テスト意図に反し、古いクレジットカードで特急券を得ることに成
功 \n",
                    w3 予約内容 = mk_token(<次の予約内容>))
            )
        )
    )
)
end
TestCaseT0003
.....
Test Suite :    vdm.tc
Class :        TestCaseT0003

```

Name	#Calls	Coverage
TestCaseT0003'test01	1	95%
TestCaseT0003'TestCaseT0003	1	✓
<b>Total Coverage</b>		<b>95%</b>

#### 5.2.5.15 TestCaseT0004 クラス

##### 5.2.5.15.1 責任

クレジットカードを切り替え、古いクレジットカードで予約した特急券を得る場合をテストする、テスト階層の仕様である。

古いクレジットカードで、特急券を得ることができる。

予約会員証および変更後のクレジットカードでは「Error 58: 事前条件の評価結果が false です」という実行時エラーが発生するが、ログに「test01 テストの意図通り、予約会員証で特急券を得ることに失敗」と表示され、回帰テスト自体は成功する。

```

.....
class
TestCaseT0004 is subclass of TestCase, 共通定義
operations
public
TestCaseT0004 : seq of char ==> TestCaseT0004
TestCaseT0004 (name) ==
    setName(name);
public
test01 : () ==> ()
test01 () ==
    ( def w クレジットカード = new クレジットカード (mk_token (<クレジット ID01>), mk_token (<クレジット PW01>));
      w おサイフケータイ = new おサイフケータイ (mk_token (<おサイフケータイ ID01>), w クレジットカード);
      w 会員証 = new 予約会員証 (mk_token (<会員証 ID01>), mk_token (<会員証 PW01>), w クレジットカード);
      w 契約 = new 契約 (w おサイフケータイ, w 会員証);
      w システム = new 特急券予約システム ({ |-> }) in
      ( assertTrue("test01 契約に失敗",
        w システム.予約集合を得る (w 契約) = {} and
        w 契約.会員証を得る () = w 会員証);
        def w 古いクレジットカード = w おサイフケータイ.クレジットカードを得る () in
        ( def - = w システム.予約する (w 契約, mk_token (<予約 ID01>), w 古いクレジットカード
, mk_token (<最初の予約内容>)) in
          assertTrue("test01 予約に失敗",
            w システム.特急券を得る (mk_token (<予約 ID01>), w 古いクレジットカード
) = mk_token (<最初の予約内容>));
            def w2 クレジットカード = new クレジットカード (mk_token (<クレジットカード ID02>), mk_token (<クレジットカード PW02>));
            w2 おサイフケータイ = new おサイフケータイ (mk_token (<おサイフケータイ ID01>), w2 クレジットカード);

```

```

w 変更後のクレジットカード = w2 おサイフケータイ.クレジットカードを得る() in
(
  def - = w システム.予約する(w 契約, mk_token(<予約 ID02>), w 変更後のクレジットカード
, mk_token(<次の予約内容>)) in
    assertTrue("test01 2 回目の予約に失敗",
      w システム.特急券を得る(mk_token(<予約 ID02>), w 変更後のクレジットカード
) = mk_token(<次の予約内容>));
    def w2 予約内容 = w システム.特急券を得る(mk_token(<予約 ID01>), w 古いクレジットカード) in
      assertTrue("test01 古いクレジットカード特急券を得ることに失敗",
        w2 予約内容 = mk_token(<最初の予約内容>));
      trap <RuntimeError>
      with print("\ttest01 テストの意図通り、予約会員証で特急券を得ることに失敗\n") in
        (
          w システム.クレジットカードを切り替える(w 契約, w 変更後のクレジットカード);
          def w3 予約内容 = w システム.特急券を得る(mk_token(<予約 ID01>), w 会員証) in
            assertTrue("\ttest01 テスト意図に反し、予約会員証で特急券を得ることに成功する
が...\n",
              w3 予約内容 = mk_token(<最初の予約内容>))
          )
        )
      )
    )
  )
end
TestCaseT0004

```

Test Suite : vdm.tc

Class : TestCaseT0004

Name	#Calls	Coverage
TestCaseT0004'test01	1	94%
TestCaseT0004'TestCaseT0004	1	✓
<b>Total Coverage</b>		<b>95%</b>

## 5.2.6 モデルの検証

モデルの検証は、VDM++ のツールである VDMTools を使った場合、以下を行う。

1. モデルを実行せずにツールで検証する静的検証
2. モデルを実行して検証する動的検証

### 5.2.6.1 静的検証

各仕様ファイル毎の構文チェックと、全仕様ファイルに静的な解析で分かるエラーがないかをチェックする型チェック、およびツールが生成する証明課題のレビューを行う。

このチェックで、かなりの単純ミスが検出される。

#### 5.2.6.1.1 証明課題レビュー

証明課題は、ツールから生成される VDM++ の条件式で、生成されるすべての条件式が true であることが証明できれば、モデルに内部矛盾が無いことを主張できる。

通常は、条件式が true であることを証明するのだが、開発現場では証明を行うことが難しいので、生成された証明課題をレビューすることで検証を行う。

証明課題レビューでは、不足する不変条件や事前条件あるいは事後条件が見つかることが多い。

### 5.2.6.2 動的検証

VDM++ のツール (VDMTools <sup>\*9</sup> と Overture Tools <sup>\*10</sup>) は、開発現場で証明を実際に行うのは難しいと考え、仕様アニメーション (仕様実行) によって正当性検証と妥当性確認を行うよう推奨している。

#### 5.2.6.2.1 回帰テスト

回帰テスト (regression test) は、プログラムを変更した際に、過去にテストした箇所に変更余波が波及し、品質が退化していないかを確認するテストである。

VDM++ の実行可能な仕様は、プログラムと同じ性質を持っているので、回帰テストをすることが可能である。

VDM++ の回帰テストには、通常、VDMUnit ライブラリ <sup>\*11</sup> を使用する。

## 5.2.7 特急券予約システムのまとめ

### 5.2.7.1 問題は何だったのか

結局、問題は何だったのかといえば、特急券予約システムのクレジットカードによる決済のモデルが考慮不足だったということになる。

まず、おサイフケータイのクレジットカードを変更すると、新規に特急券予約システムを契約せねばならず、新規に予約会員証も作成しなければならない。

---

<sup>\*9</sup> <http://www.vdmttools.jp>

<sup>\*10</sup> <http://www.overturetool.org>

<sup>\*11</sup> オランダの Marcel Verhoef 博士が開発した回帰テスト用の VDMUnit ライブラリ。 <http://www.vdmttools.jp/modules/tinyd2/index.php?id=9>

しかし、変更前の予約を行った古いクレジットカードは、その予約の特急券を受け取る時に必要になる。何故なら、図 5.1 のクラス図で見るように、個々の予約を表す特急券予約システムからクレジットカードにリンクが設定されているからである。

このような構造でも、変更後にすべての特急券予約システムからクレジットカードへのリンクを新しいクレジットカードに変更していれば問題はないはずだが、恐らく、変更に掛かる効率を理由に、古いクレジットカードとリンクしたままになっているのであろう。設計上の理由で、ユーザーに迷惑を掛ける仕様となっている訳である。

さらに、モデルに問題があった。予約会員証は新規に作るのが普通であるが、クレジットカード会社の都合による変更のため、新規に契約する費用は無料となった。そのため、予約会員証は古いものを使用することになった。契約は新規だが、予約会員証は古く、そして図 5.1 のクラス図から分かるように、予約会員証からリンクしているクレジットカードは、変更前の古いものであるという訳である。

これではまずいと思って、鉄道会社 A サポートセンターの係員がリンクされているクレジットカードを新しいものにしたようである。その結果、特急券予約システムの変更前の予約会員証で特急券を得ることができなかった。

#### 5.2.7.2 本来どうすべきだったか？

本来、特急券予約システムは図 10.1 のクラス図で見るように、契約<sup>\*12</sup>とリンクが設定されているべきであり、契約がクレジットカードとリンクしているべきである。

予約会員証も契約を介してクレジットカードを参照できるようにすべきである。

このようにしておけば、契約に変更があっても、古い特急券予約システムは新しい契約を介して、新しいクレジットカードにアクセスでき、同じく予約会員証も新しいクレジットカードにアクセスできる。

本来どうすべきだったのかを検証する VDM++ モデルは、1 日で記述および検証ができた。

詳細は、特急券予約システムの問題点を推測し、修正した 10 章の「特急券予約システム改善モデル」を参照のこと。

#### 5.2.8 特急券予約システムモデル化の統計情報

注釈抜きの VDM++ ソース行数は、501 行、モデル作成工数は約 1 日、発表用の資料作成と VDM++ モデルの読みやすさのための整形清書に約 2 日、モデルの修正工数は 1 日かかった。

なお、上記モデルの作成工数は、筆者が問題を解決するために消費した工数より少ない。

<sup>\*12</sup> 口座と言ってもよいが、本モデルでは契約という名前にした



## 第 6 章

## まとめ

ここまで、なぜ形式手法か？、VDM の概要・成果・導入方法、如何にして対象をモデル化するか？を説明してきたが、本章では、ここまで説明した内容を、形式手法と VDM の有用性、構造化日本語仕様としての VDM++ という観点からまとめる。

## 6.1 形式手法と VDM の有用性

- 理論的にも経験的にも形式手法が有効  
証明やモデル検査は、長期的にはやるべきだが、開発現場で導入するのはすぐには難しい。
- 証明やモデル検査と比べ、VDM 導入はさほど難しくない
  - － 成功プロジェクト<sup>\*1</sup>は、3 ヶ月程度の教育とコンサルティングで導入している
  - － 形式手法だけでなく、既存の役立つソフトウェア工学ツールと協調して、より効果が出る
- モデル化は、以下が重要である
  - － モデル化の範囲を決め、仕様を階層と分割によって、分割統治する
  - － 名詞から型、述語から関数または操作を導き出す
  - － 陰仕様を作成してから、静的検証を行う  
この段階で、間違いに気付くことが非常に多い。
  - － 陽仕様を作成してから、動的検証を行う  
回帰テストと組合せることで、仕様の変更が容易になる。
- VDM++ ソースは、識別子の日本語表現を適切に行うと、構造化日本語仕様として使うことができる

## 6.2 構造化日本語仕様としての VDM

5.2.5.1 節 5.2.5.1.3 段落の操作は、以下のようであり、若干の VDM++ 知識があれば、構造化された日本語仕様として読むことができる。

```
public 予約する : 契約 * ID * クレジットカード * 特急券予約‘予約内容 ==> 特急券予約
予約する(a 契約, anID, a クレジットカード, a 予約内容) == (
  def w 特急券予約 = new 特急券予約(anID, a クレジットカード, a 予約内容) in (
    if 予約がある契約である(a 契約, s 予約表) then
      s 予約表 := 予約表を更新する(s 予約表, a 契約, w 特急券予約)
    else
      s 予約表 := 予約表に追加する(s 予約表, a 契約, w 特急券予約);
    return w 特急券予約
  )
)
post
  if 予約がある契約である(a 契約, s 予約表~) then
    予約表が更新されている(s 予約表~, a 契約, RESULT, s 予約表)
  else
    予約表に追加されている(s 予約表~, a 契約, RESULT, s 予約表);
```

<sup>\*1</sup> 厳密な仕様記述における形式手法成功事例調査報告書 (<http://sec.ipa.go.jp/reports/20130125.html>) の TradeOne と FeliCa ファームウェアのプロジェクト参照のこと。

この VDM++ 仕様から、事後条件として、「予約がある契約である」ならば、「予約表が更新されている」状態になり、そうでなければ、「予約表に追加されている」状態になる事が分かる。「予約表が更新されている」と「予約表に追加されている」ことの違いは、それぞれの関数の内容を確認すれば理解できる。

VDM++ の知識が少しあれば、パラメータとして「予約表が更新されている」と「予約表に追加されている」ものに、インスタンス変数である s 予約表の旧値 (old value) と現在の値が渡されていて、恐らくそれぞれの関数の中で両者の比較が行われていることも分かるし、返り値を表す予約後の RESULT がパラメータとして渡されることから、返り値と何かを比較しているだろうことも分かる。

表 6.1 に、開発現場でよく使われる、擬似コードを交えた日本語仕様と適切な日本語を使って構造化された VDM++ 仕様の比較を示す。

表 6.1 日本語の仕様中の擬似コードと、VDM の比較

	構文を考える時間	構文チェック	関連チェック	実行テスト	証明課題生成
擬似コードによる仕様	かなりの時間が必要で、記法も統一できない	レビューのみ	レビューのみ	具体的データを想定したコードインスペクションに相当するチェックのみ (通常行われない)	不可能
VDM 仕様	言語マニュアルを参照すれば良いだけ	ツールでチェック	ツールの型チェック	ツールで実行	ツールで生成

表 6.1 を見れば、擬似コードを混じえた日本語仕様は「使えない」ことが分かる。

適切な日本語仕様を使って構造化された VDM++ 仕様、すなわち構造化日本語仕様としての VDM 仕様は、以下の特徴を持つ。

- 構文を考える時間が不要である  
逆に、擬似コードを使う日本語仕様では、仕様の意味を考えねばならないのに、構文を考えていることが非常に多い。
- 構文・型チェック、証明課題レビューで静的に検証できる  
非常に多くの単純ミスを簡単に発見できる。
- 証明課題で生成される条件式から、見落とししていた不変条件や事前条件が見つかる
- 組合せテストにより、動的に正当性検証ができる
- 回帰テストにより、動的に妥当性確認ができる
- VDM ソース自体が、要求辞書ともなる
- 日本語仕様より、記述と検証の工数が少ない  
特に、仕様修正に強い
- 擬似コード形式の日本語仕様に近い形で、かなりの部分を記述できる

### 6.3 形式手法導入のコツ

ここまで説明してきたことから、形式手法導入のコツは極めて簡単であることが分かる。

すなわち、日本で最大規模の形式仕様記述を使ってプロジェクトを成功に導いた、フェリカネットワークスの栗田太郎

氏が述べているように、「やれば良いだけです」。本書が、そのお役に立てることを願っている。

## 第 7 章

# 演習問題

本章では、演習問題として以下で説明する図書館システムの例題をあげる。

この図書館システムは、どなたでも基本的な機能を理解でき、規模も非常に小さいが、我々が開発するシステムの仕様としての基本的な特徴を持っているからである。

## 7.1 図書館システム

以下の機能を持つ図書館システムを、VDM++ でモデル化する。

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。

このシステムを、4 章と 5 章で説明した方法でモデル化してみよう。

最初にユースケースレベルの要求仕様を作成する。実行不可能だが静的検証のできる陰仕様を作成し、次に実行可能で動的検証のできる陽仕様を作成し、回帰テストによりテストせよ。

次に、設計仕様を、VDM++ を使ったオブジェクト指向モデルとして作成せよ。

解答例は、8 章に示す。

## 第Ⅱ部

# モデルの具体例



## 第 8 章

# 演習問題解答の図書館システムモデル

## 8.1 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。

### 8.1.1 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。

## 8.2 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  
.....
```

### 8.2.1 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 1 冊の蔵書を蔵書 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。

```
.....  
types  
public 蔵書 = map 蔵書 ID to 本;  
.....
```

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。<sup>\*1</sup>
- 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。

## 8.2.2 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。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。初期値は空の写像である。

<sup>\*1</sup> 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の意味で w にした。

```

.....
instance variables
private i 蔵書 : 蔵書 := { |-> };
inv card dom i 蔵書 <= v 最大蔵書数

```

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。

### 8.2.3 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、それは、上級の話なので、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

```

### 8.2.4 操作定義

#### 8.2.4.1 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。

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~);
.....

```

#### 8.2.4.2 蔵書を削除する

「本を図書館の蔵書として追加する」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。<sup>\*2</sup>

```

.....
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)
post 蔵書が削除されている (a 削除本, i 蔵書, i 蔵書~);
.....

```

#### 8.2.4.3 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)
.....

```

<sup>\*2</sup> 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。

## 8.2.4.4 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。

このように、VDM++ では、`types` や `instance variables` の定義は 1 箇所ですべて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 := { |-> };`

## 8.2.4.5 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~);`

## 8.2.4.6 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~)`

## 8.2.5 関数定義

### 8.2.5.1 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;

.....

### 8.2.5.2 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;

.....

### 8.2.5.3 蔵書に存在する

a 追加本の定義域集合が、a 図書館蔵書の定義域集合の部分集合であれば、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) ==

**forall** id in **set dom** a 追加本 & id in **set dom** a 図書館蔵書;

.....

### 8.2.5.4 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);

## 8.2.5.5 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};
```

## 8.2.5.6 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);
```

## 8.2.5.7 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
forall id in set dom a 貸出本 & id in set dom w 蔵書;
```

## 8.2.5.8 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);
```

## 8.2.5.9 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);

#### 8.2.5.10 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);

#### 8.2.5.11 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;

#### 8.2.5.12 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

## 8.3 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 RDO
.....
```

### 8.3.1 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
.....
```

### 8.3.2 関数定義

#### 8.3.2.1 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);

.....

### 8.3.2.2 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);

### 8.3.2.3 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;

### 8.3.2.4 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
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

Name	#Calls	Coverage
Total Coverage		0%

## 8.4 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
.....
```

### 8.4.1 インスタンス変数定義

```
.....
instance variables
private i 蔵書 : 蔵書 := { |-> };
inv card dom i 蔵書 <= v 最大蔵書数
private i 貸出 : 貸出 := { |-> };
.....
```

### 8.4.2 操作定義

#### 8.4.2.1 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~);
.....
```

## 8.4.2.2 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~);

```

## 8.4.2.3 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);

```

## 8.4.2.4 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~);

#### 8.4.2.5 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~);

#### 8.4.2.6 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	918	✓
図書館 1‘本を返す	306	✓
図書館 1‘蔵書を得る	408	✓
図書館 1‘貸出を得る	408	✓
図書館 1‘本を検索する	612	✓
図書館 1‘蔵書を削除する	204	✓
図書館 1‘蔵書を追加する	1224	✓
Total Coverage		100%

## 8.5 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
```

### 8.5.1 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
```

### 8.5.2 関数定義

#### 8.5.2.1 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) ==
  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);

```

#### 8.5.2.2 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
  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);

### 8.5.2.3 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;

### 8.5.2.4 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	253	✓
図書館 RD'貸出に存在する	220	✓
図書館 RD'貸出可能である	99	✓
図書館 RD'貸出可能でない	0	0%
図書館 RD'蔵書に存在しない	132	✓
図書館 RD'貸出に存在しない	154	✓
図書館 RD'検索キーが空でない	66	✓
図書館 RD'蔵書が削除されている	11	✓
図書館 RD'蔵書が追加されている	132	✓
図書館 RD'貸出に追加されている	88	70%
図書館 RD'貸出から削除されている	33	✓
図書館 RD'最大貸出数を超えていない	99	✓

Name	#Calls	Coverage
Total Coverage		87%

## 8.6 図書館 1 の回帰テスト

### 8.6.1 TestApp

#### 8.6.1.1 責任

回帰テストを行う。

#### 8.6.1.2 クラス定義

```
class
TestApp
```

#### 8.6.1.3 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
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 ("TestCaseUT0001 : \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	545	82%
<b>Total Coverage</b>		<b>82%</b>

## 8.6.2 TestCaseComm

### 8.6.2.1 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
.....
```

## 8.6.3 TestCaseUT0001

### 8.6.3.1 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
.....

```

## 8.6.4 TestCaseUT0002

## 8.6.4.1 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、最大貸出数を超えたエラー発生。\\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("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
TestCaseUT0002
.....

```

## 8.7 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 8.1 で見え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。

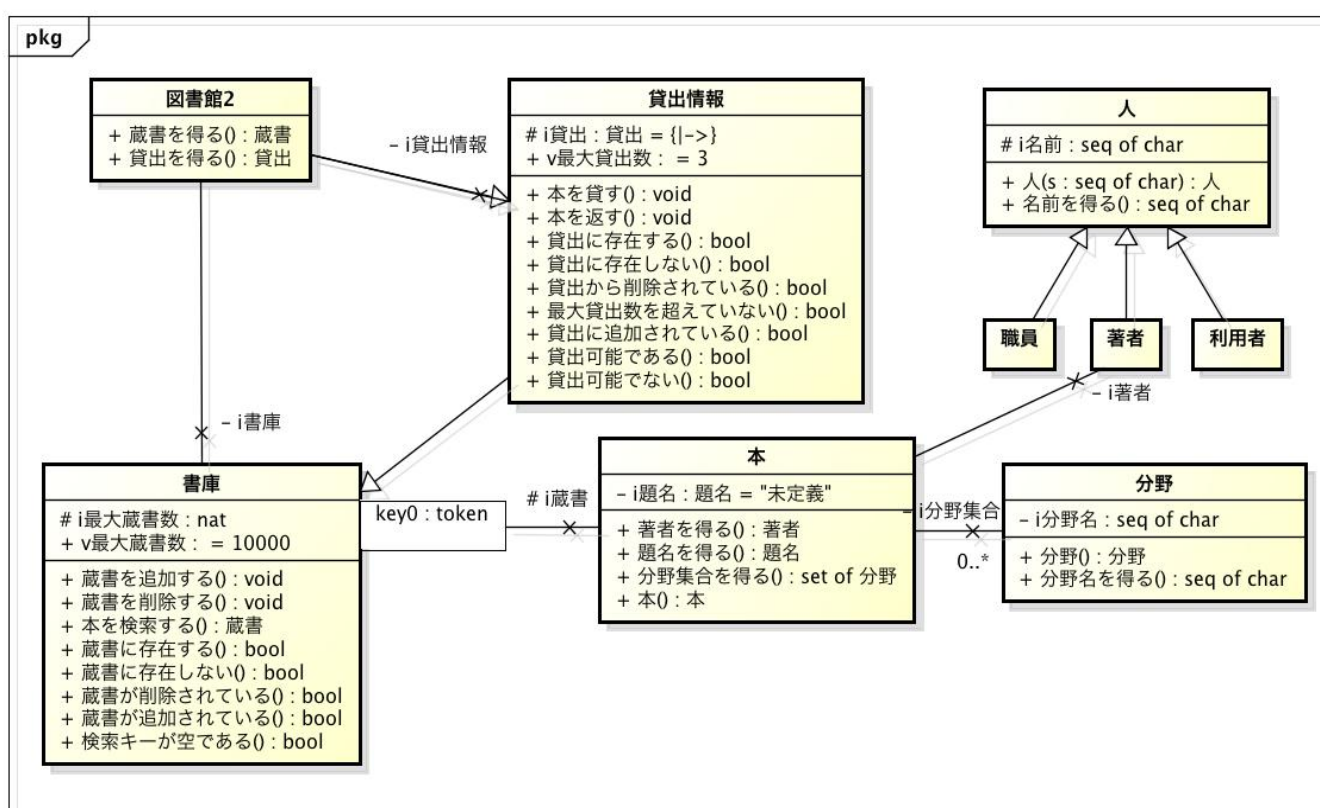


図 8.1 図書館 2 モデルのクラス図

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、インター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る。

## 8.8 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 ();

```

### 8.8.1 操作定義

#### 8.8.1.1 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

```

end

図書館 2

Test Suite : vdm.tc

Class : 図書館 2

Name	#Calls	Coverage
図書館 2'蔵書を得る	1635	✓
図書館 2'貸出を得る	1090	✓
Total Coverage		100%

## 8.9 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	2180	✓
著者 '著者名を得る	2180	✓
Total Coverage		100%

## 8.10 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<sup>\*3</sup>。したがって、一連の文の評価・実行に副作用が無い<sup>\*4</sup>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
著者を得る : () ==> 著者
著者を得る () ==
  return i 著者;

```

<sup>\*3</sup> 原子的であるという。

<sup>\*4</sup> 実行順序によって結果が変わることがない。

```
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
```

.....

## 8.11 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
instance variables
  protected i 最大蔵書数 : nat;
  protected i 蔵書 : 蔵書 := { |-> };
.....

```

### 8.11.1 操作定義

#### 8.11.1.1 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~);
.....

```

#### 8.11.1.2 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)
  post 蔵書が削除されている (a 削除本, i 蔵書, i 蔵書~);
.....

```

## 8.11.1.3 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 ()}}
```

.....

## 8.11.2 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);
```

.....

## 8.11.2.1 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 ‘本を検索する	36	√
書庫 ‘蔵書に存在する	135	√
書庫 ‘蔵書を削除する	9	√
書庫 ‘蔵書を追加する	72	√
書庫 ‘蔵書に存在しない	72	√
書庫 ‘検索キーが空でない	36	√
書庫 ‘蔵書が削除されている	9	√
書庫 ‘蔵書が追加されている	72	√
Total Coverage		100%

## 8.12 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	3270	✓
分野 '分野名を得る	14891	✓
Total Coverage		100%

## 8.13 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 := { |-> };

```

### 8.13.1 操作定義

#### 8.13.1.1 本を貸す

```

operations
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~) ;

```

#### 8.13.1.2 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~)
.....

```

### 8.13.2 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
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);
.....

```

## 8.13.2.1 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 '本を貸す'	54	✓
貸出情報 '本を返す'	18	✓
貸出情報 '貸出に存在する'	108	✓
貸出情報 '貸出可能である'	54	✓
貸出情報 '貸出可能でない'	0	0%
貸出情報 '貸出に存在しない'	72	✓
貸出情報 '貸出に追加されている'	45	70%
貸出情報 '貸出から削除されている'	18	✓
貸出情報 '最大貸出数を超えていない'	54	✓
Total Coverage		89%

## 8.14 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	11032	✓
Total Coverage		50%

## 8.15 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	1090	✓
Total Coverage		100%

8.16 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	1635	✓
Total Coverage		100%

## 8.17 図書館 2 回帰テスト

### 8.17.1 TestApp2

#### 8.17.1.1 責任

回帰テストを行う。

#### 8.17.1.2 クラス定義

```
class
```

```
TestApp2
```

#### 8.17.1.3 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
```

```
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	545	82%
<b>Total Coverage</b>		<b>82%</b>

## 8.17.2 TestCaseComm2

## 8.17.2.1 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
.....

```

## 8.17.3 TestCaseUT2001

## 8.17.3.1 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 (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
TestCaseUT2001
.....

```

## 8.17.4 TestCaseUT2002

### 8.17.4.1 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);
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 = { |-> })
)
)
end
TestCaseUT2002
.....
```

## 8.18 8 章の参考文献

VDM++<sup>[4]</sup> は、1970 年代中頃に IBM ウィーン研究所で開発された VDM-SL<sup>[3]</sup> を拡張し、さらにオブジェクト指向拡張した形式仕様記述言語である。

VDM++ の教科書としては <sup>[5]</sup> がある。

VDM++ を開発現場で実践的に使う場合の解説が <sup>[6]</sup> にある。



## 第 9 章

# 運賃計算モデル

## 9.1 運賃を得る

運賃を得るユースケース・レベルの要求仕様であり、業務論理階層の仕様である。

「適用する」操作を呼び出すことで、運賃を計算して返す。

```

.....
class
  運賃を得る is subclass of 運賃表辞書
  instance variables
  private s 運賃表 : 運賃表 := [];
  private s 駅集合 : 路線網 '駅集合;
  private s 路線単位集合 : 路線検索 '路線単位集合;
  private s 路線検索 : 路線検索;

  operations
  public
  運賃を得る : 運賃表 * 路線網 '駅集合 * 路線検索 '路線単位集合 ==> 運賃を得る
  運賃を得る (a 運賃表, a 駅集合, a 路線単位集合) ==
    ( s 運賃表 := a 運賃表;
      s 駅集合 := a 駅集合;
      s 路線単位集合 := a 路線単位集合;
      s 路線検索 := new ダイクストラ算法による路線検索 (s 駅集合, s 路線単位集合);
      return self
    );
  public
  適用する : 路線網 '駅 * 路線網 '駅 ==> 運賃
  適用する (a 出発駅, a 到着駅) ==
    let w 距離 = s 路線検索.最短距離 (a 出発駅, a 到着駅) in
    return 距離に応じた運賃を得る (s 運賃表, w 距離)
pre s 路線検索.到達可能である (a 出発駅, a 到着駅)
end
運賃を得る
.....
Test Suite :      vdm.tc
Class :          運賃を得る

```

Name	#Calls	Coverage
運賃を得る '適用する	2	✓
運賃を得る '運賃を得る	4	✓
Total Coverage		100%

## 9.2 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) 階層の仕様である。

運賃表に関する、問題領域の知識 (Domain Knowledge) を持つ。

```

.....
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;
.....

```

`:-` という記号は、レコードの欄である運賃型の `f 運賃` が、レコードのキーとはならないことを指示している。

### 9.2.1 運賃表 (型)

運賃表の各行の上限と下限は重複しないことを、不変条件として定義している。

```

.....
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)
.....

```

### 9.2.2 距離に応じた運賃を得る

```

.....
functions
public static
距離に応じた運賃を得る : 運賃表 * 路線網 ‘距離 -> 運賃
距離に応じた運賃を得る (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 ;
.....

```

### 9.2.3 運賃表のある行に存在する

.....

```
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;
```

.....

### 9.2.4 運賃表にただ一つ存在する

.....

```
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);
```

.....

### 9.2.5 運賃表の何番目かを得る

.....

```
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;
```

.....

### 9.2.6 下限より上限が大きい

.....

```
public static
```

下限より上限が大きい : 路線網 '距離 \* 路線網 '距離 -> bool

下限より上限が大きい (a 下限, a 上限) ==

```
  a 下限 < a 上限;
```

.....

### 9.2.7 上限と次の行の下限は等しい

.....

```
public static
```

上限と次の行の下限は等しい : 路線網 ' 距離 \* 路線網 ' 距離 -> 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	576	✓
運賃表辞書 ' 距離に応じた運賃を得る	2	✓
運賃表辞書 ' 運賃表の何番目かを得る	4	✓
運賃表辞書 ' 運賃表にただ一つ存在する	2	✓
運賃表辞書 ' 運賃表のある行に存在する	24	✓
運賃表辞書 ' 上限と次の行の下限は等しい	80	✓
Total Coverage		100%

## 9.3 路線網

運賃計算に限らず使用することのできる、路線網の明示的知識を記述した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;
.....

```

### 9.3.1 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 <> {}
.....

```

### 9.3.2 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	986	✓
路線網 ‘路線単位の両側の駅は異なる	986	✓
路線網 ‘路線単位列中の駅集合を得る	16	✓
Total Coverage		100%

## 9.4 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	6	✓
路線検索 到達可能である	8	✓
Total Coverage		94%

## 9.5 ダイクストラ算法による路線検索

ダイクストラ算法を呼び出すこと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
);

```

### 9.5.1 最短経路

ダイクストラ算法によって、最短経路を表す駅の列と、経路の距離を返す。

経路が繋がっていない場合は空列と距離 0 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) ==
(
  decl 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	12	✓
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	8	✓
Total Coverage		100%

## 9.6 ダイクストラ算法

路線単位 of 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 = 1000000000000000000

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) ==
  ( decl 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	72	✓
ダイクストラ算法 '最短経路	12	✓
ダイクストラ算法 '経路を作る	12	✓
ダイクストラ算法 'ダイクストラ算法	12	✓
Total Coverage		100%

## 9.7 鉄道ネットデータ

鉄道ネットワークのデータ (駅集合、路線単位、路線単位集合、路線集合) を持つ。

```

.....
class
鉄道ネットデータ
values
public
v 東京 : 路線網 '駅 = mk_token("東京");
public
v 新宿 : 路線網 '駅 = mk_token("新宿");
public
v 品川 : 路線網 '駅 = mk_token("品川");
public
v 四ツ谷 : 路線網 '駅 = mk_token("四ツ谷");
public
v 池袋 : 路線網 '駅 = mk_token("池袋");
public
v コペンハーゲン : 路線網 '駅 = mk_token("コペンハーゲン");
public
v 駅集合 : 路線網 '駅集合 = {v 東京, v 新宿, v 品川, v 四ツ谷, v 池袋, v コペンハーゲン };
public
v 東京から四ツ谷 : 路線網 '路線単位 = mk_路線網 '路線単位 (v 東京, v 四ツ谷, 4.2);
public
v 四ツ谷から東京 : 路線網 '路線単位 = mk_路線網 '路線単位 (v 四ツ谷, v 東京, 4.2);
public
v 四ツ谷から新宿 : 路線網 '路線単位 = mk_路線網 '路線単位 (v 四ツ谷, v 新宿, 3.5);
public
v 新宿から四ツ谷 : 路線網 '路線単位 = mk_路線網 '路線単位 (v 新宿, v 四ツ谷, 3.5);
public
v 東京から池袋 : 路線網 '路線単位 = mk_路線網 '路線単位 (v 東京, v 池袋, 10);
public
v 池袋から東京 : 路線網 '路線単位 = mk_路線網 '路線単位 (v 池袋, v 東京, 10);
public
v 池袋から新宿 : 路線網 '路線単位 = mk_路線網 '路線単位 (v 池袋, v 新宿, 5.1);
public
v 新宿から池袋 : 路線網 '路線単位 = mk_路線網 '路線単位 (v 新宿, v 池袋, 5.1);
public
v 新宿から品川 : 路線網 '路線単位 = mk_路線網 '路線単位 (v 新宿, v 品川, 9.8);
public
v 品川から新宿 : 路線網 '路線単位 = mk_路線網 '路線単位 (v 品川, v 新宿, 9.9);

```

```
public
```

```
v 品川から東京 : 路線網 '路線単位 = mk.路線網 '路線単位 (v 品川, v 東京, 5.1);
```

```
public
```

```
v 東京から品川 : 路線網 '路線単位 = mk.路線網 '路線単位 (v 東京, v 品川, 5.3);
```

```
public
```

```
v 路線単位集合 : 路線網 '路線単位集合 = {  
    v 東京から四ツ谷, v 四ツ谷から東京, v 四ツ谷から新宿, v 新宿から四ツ  
    谷,  
    v 東京から池袋, v 池袋から東京, v 池袋から新宿, v 新宿から池袋,  
    v 新宿から品川, v 品川から新宿, v 品川から東京, v 東京から品川 }
```

```
end
```

```
鉄道ネットデータ
```

```
.....  
Test Suite :    vdm.tc
```

```
Class :        鉄道ネットデータ
```

Name	#Calls	Coverage
Total Coverage		undefined

## 9.8 TestApp Class

### 9.8.1 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
```

### 9.8.2 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。

ここでは、TestSuite と TestResult のインスタンス ts と tr をそれぞれ作成し、tr にテスト結果の表示方法を持つ PrintTestListener を登録し、ts に 4 つの回帰テストケース TestCaseT0001, TestCaseT0002, TestCaseT0003, TestCaseT0004 を登録している。

次に、ts の run 操作で回帰テストを実行し、tr オブジェクトにすべての回帰テストが成功したかを尋ね、回帰テスト結果を表示している。

```
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%
<b>Total Coverage</b>		<b>85%</b>

## 9.9 回帰テストケース

### 9.9.1 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抽象クラスで、最短距離を求める算法のインスタンスと、「運賃を得る」クラスのインスタンスを持つ。

個々の回帰テストは、このクラスのサブクラスとして記述しなければならない。

print 操作は、回帰テストのログを表示する。

```

class
TestCaseT is subclass of TestCase, 鉄道ネットデータ
values
public
v 最大値 = 1000000000
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

```

### 9.9.2 回帰テストのノーマルケース例

assertTrue 文を使って、テスト結果が期待した値になっていることを確認している。

assertTrue 文の第 2 引数が、true になることを期待した式である。すなわち、距離が 7.7 で、運賃が 220 であれば、正しいということになる。そうでなければ、assertTrue 文の第 1 引数の文字列が、ログとして表示される。

```

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
.....

```

### 9.9.3 回帰テストのエラーケース例

trap 文を使って、実行時エラーの例外 (RuntimeError) が発生することを期待したテストケースである。

期待通り実行時エラー例外が発生したら、with 句以下の print 操作を呼び出して、期待した事前条件エラーが発生したことをログに表示する。

VDMTools で実行時エラーを補足するオプションをオンにしておけば、回帰テスト中に実行時エラーが発生した旨のエラー・ウィンドウが表示されるが、回帰テストの実行は最後まで行われ、他に予想外のエラーが無ければ、回帰テストは正常終了したとログに表示される<sup>\*1</sup>。

```

.....
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

```

<sup>\*1</sup> Overture tool は、実行時エラーを補足する機能がないので、最初のエラー発生時点で回帰テストが終了してしまう。

```
(  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
```

.....



## 第 10 章

# 特急券予約システム改善モデル

## 10.1 はじめに

鉄道会社 A の特急券予約システムに関する「特急券予約システムシステムの問題点を推測した仕様」(第 5.2.5 章参照)を、本来あるべき仕様として書き直した仕様である。

### 10.1.1 本来どうすべきだったのか？

修正後のクラス図 10.1 で見るように、本来、特急券予約システムは契約<sup>\*1</sup>とリンクが設定されているべきであり、契約がクレジットカードとリンクしているべきである。

予約会員証も契約を介してクレジットカードを参照できるようにすべきである。

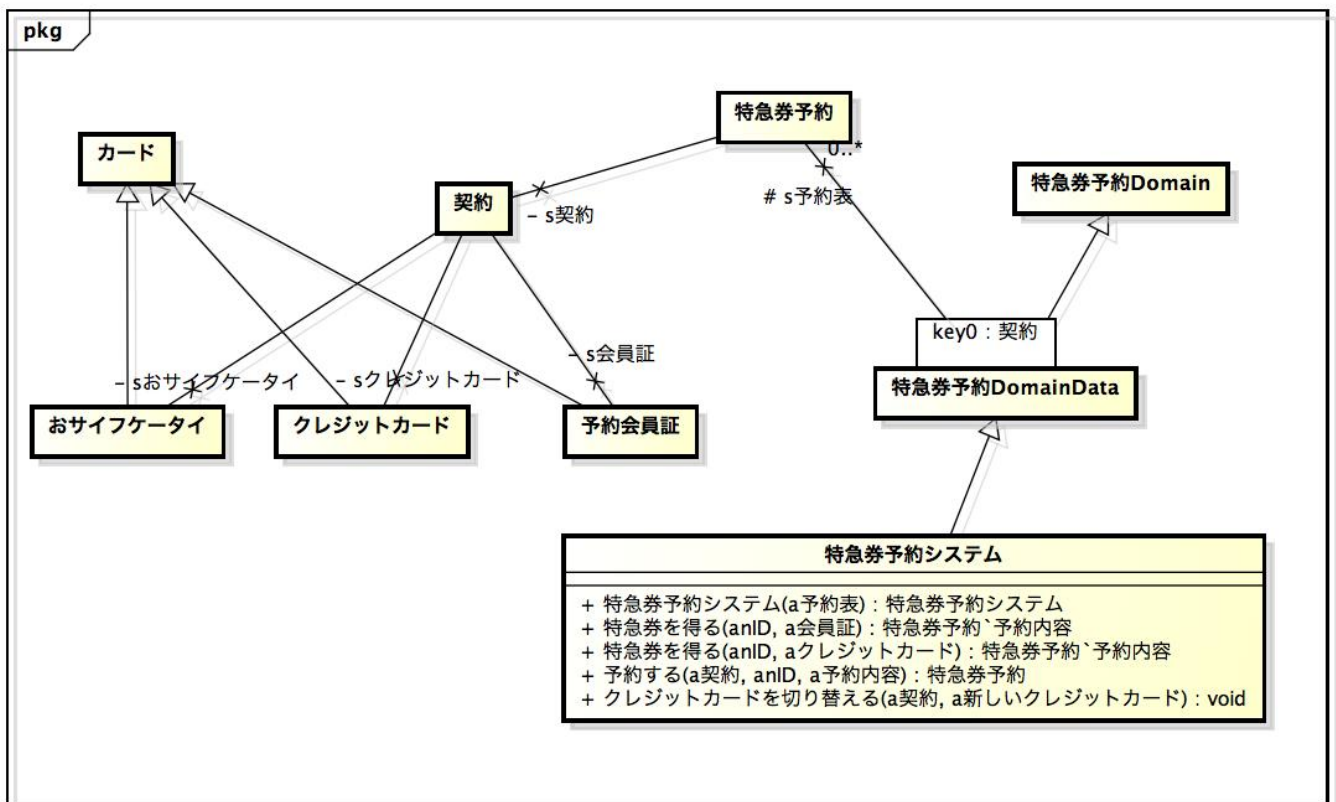


図 10.1 修正後のクラス図

このようにしておけば、契約に変更があっても、古い特急券予約は新しい契約を介して、新しいクレジットカードにアクセスでき、同じく予約会員証も新しいクレジットカードにアクセスできる。

修正前のクラス図 5.1 と比べれば、再利用性と保守性が増しているにもかかわらず、構造自体は特に複雑になっているわけではないことが分かる。

\*1 口座と言ってもよいが、本モデルでは契約という名前にした

## 10.2 特急券予約システムクラス

### 10.2.1 責任

改良した特急券予約システムを、構造化日本語仕様レベルで表す。陽仕様の実行に関わる部分は、1 階層下の仕様階層である、スーパークラスの「特急券予約 Domain」で記述した。

### 10.2.2 クラス定義と構成子定義

```
.....
class
特急券予約システム is subclass of 特急券予約 DomainData
operations
public
特急券予約システム : 予約表 ==> 特急券予約システム
特急券予約システム (a 予約表) ==
(   s 予約表 := a 予約表
);
.....
```

### 10.2.3 操作：予約する

予約を行う。

```
.....
public
予約する : 契約 * ID * 特急券予約 '予約内容 ==> 特急券予約
予約する (a 契約, anID, a 予約内容) ==
(   def w 特急券予約 = new 特急券予約 (anID, a 契約, a 予約内容) in
    (   if 予約がある契約である (a 契約, s 予約表)
        then s 予約表 := 予約表を更新する (s 予約表, a 契約, w 特急券予約)
        else s 予約表 := 予約表に追加する (s 予約表, a 契約, w 特急券予約);
        return w 特急券予約
    )
)
post if 予約がある契約である (a 契約, s 予約表~)
    then 予約表が更新されている (s 予約表~, a 契約, RESULT, s 予約表)
    else 予約表に追加されている (s 予約表~, a 契約, RESULT, s 予約表);
.....
```

### 10.2.4 操作：特急券を得る

クレジットカードで特急券を得る。

```
.....
```

```

public
特急券を得る : ID * クレジットカード ==> 特急券予約 ‘予約内容
特急券を得る (anID, a クレジットカード) ==
  (  def w 予約 = 予約を得る (s 予約表, anID, a クレジットカード) in
      return w 予約.予約内容を得る ()
  )
pre let w 予約 = 予約を得る (s 予約表, anID, a クレジットカード) in
  a クレジットカード = w 予約.クレジットカードを得る ()
post RESULT = 予約を得る (s 予約表, anID, a クレジットカード).予約内容を得る ();
.....

```

### 10.2.5 操作：特急券を得る

予約会員証で特急券を得る。

```

public
特急券を得る : ID * 予約会員証 ==> 特急券予約 ‘予約内容
特急券を得る (anID, a 会員証) ==
  (  def w 予約 = 予約を得る (s 予約表, anID, a 会員証) in
      return w 予約.予約内容を得る ()
  )
pre let w 予約 = 予約を得る (s 予約表, anID, a 会員証) in
  a 会員証 = w 予約.契約を得る ().会員証を得る ();
.....

```

### 10.2.6 操作：クレジットカードを切り替える

本操作は、一般ユーザーは使えず、鉄道会社 A の担当者だけが使える。

```

public
クレジットカードを切り替える : 契約 * クレジットカード ==> ()
クレジットカードを切り替える (a 契約, a 新しいクレジットカード) == a 契約.
  クレジットカードを設定する (a 新しいクレジットカード)
pre 契約が存在する (a 契約, s 予約表)
end
特急券予約システム
.....

```

Test Suite : vdm.tc

Class : 特急券予約システム

Name	#Calls	Coverage
特急券予約システム ‘予約する	8	✓
特急券予約システム ‘特急券予約システム	4	✓

Name	#Calls	Coverage
特急券予約システム ‘クレジットカードを切り替える	1	✓
特急券予約システム ‘特急券を得る	2	✓
特急券予約システム ‘特急券を得る	13	✓
Total Coverage		100%

### 10.3 共通定義クラス

#### 10.3.1 責任

特急券予約モデルで共通に使用する定義を表す。

```
.....
class
共通定義
types
public ID = [token];
public 暗証番号 = [token]
operations
public
print : seq of char ==> ()
print (a 文字列) ==
  let - = new IO().echo (a 文字列) in
  skip
end
共通定義
.....
Test Suite :      vdm.tc
Class :          共通定義
```

Name	#Calls	Coverage
共通定義 ‘print	2	✓
Total Coverage		100%

## 10.4 特急券予約 Domain クラス

### 10.4.1 責任

特急券予約 Domain を表す、上の階層の仕様で使える名詞と述語の用語辞書、すなわち要求辞書である。

### 10.4.2 クラス定義

.....  
`class`

特急券予約 Domain `is subclass of` 共通定義  
.....

### 10.4.3 型およびインスタンス変数定義：予約表

.....  
`types`

`public` 予約表 = `map` 契約 `to set of` 特急券予約  
.....

### 10.4.4 関数：予約を得る

ID を指定して、予約を得る。

.....  
`functions`

`public`

予約を得る : 予約表 \* ID -> 特急券予約

予約を得る (a 予約表, anID) ==

    (`let` w 予約 `in set dunion rng` a 予約表 `be st` w 予約.ID を得る () = anID `in`  
    w 予約)

`pre exists` w 予約 `in set dunion rng` a 予約表 &  
    w 予約.ID を得る () = anID ;  
.....

### 10.4.5 関数：予約を得る

クレジットカードを指定して、予約を得る。

.....  
`public`

予約を得る : 予約表 \* ID \* クレジットカード -> 特急券予約

予約を得る (a 予約表, anID, a クレジットカード) ==

```
(let w 予約 in set dunion rng a 予約表 be st
  w 予約.クレジットカードを得る () = a クレジットカード and
  w 予約.ID を得る () = anID in
w 予約)
pre exists w 予約 in set dunion rng a 予約表 &
  w 予約.クレジットカードを得る () = a クレジットカード and
  w 予約.ID を得る () = anID ;
```

.....

#### 10.4.6 関数：予約を得る

予約会員証を指定して、予約を得る。

```
public
予約を得る : 予約表 * ID * 予約会員証 -> 特急券予約
予約を得る (a 予約表, anID, a 予約会員証) ==
  (let w 予約 in set dunion rng a 予約表 be st
    w 予約.会員証を得る () = a 予約会員証 and
    w 予約.ID を得る () = anID in
  w 予約)
pre exists w 予約 in set dunion rng a 予約表 &
  w 予約.会員証を得る () = a 予約会員証 and
  w 予約.ID を得る () = anID ;
```

.....

#### 10.4.7 関数：予約表を更新する

契約と特急券予約を指定して、予約表を更新する。

```
public
予約表を更新する : 予約表 * 契約 * 特急券予約 -> 予約表
予約表を更新する (a 予約表, a 契約, a 特急券予約) ==
  a 予約表 ++ {a 契約 |-> 予約集合に追加する (a 予約表 (a 契約), {a 特急券予約 })}
pre a 契約 in set dom a 予約表
post 予約表が更新されている (a 予約表, a 契約, a 特急券予約, RESULT) ;
public
予約表が更新されている : 予約表 * 契約 * 特急券予約 * 予約表 +> bool
予約表が更新されている (a 予約表, a 契約, a 特急券予約, a 更新後予約表) ==
  a 更新後予約表 = a 予約表 ++ {a 契約 |-> 予約集合に追加する (a 予約表 (a 契約), {a 特急券予約 })};
```

.....

#### 10.4.8 関数：予約表に追加する

契約と特急券予約を指定して、予約表に追加する。

```

.....
public
予約表に追加する：予約表 * 契約 * 特急券予約 -> 予約表
予約表に追加する (a 予約表, a 契約, a 特急券予約) ==
  a 予約表 munion {a 契約 |-> {a 特急券予約 }}
pre not 予約がある契約である (a 契約, a 予約表) and
  forall w 契約 1 in set dom a 予約表, w 契約 2 in set dom {a 契約 |-> {a 特急券予約 }} &
    w 契約 1 = w 契約 2 => a 予約表 (w 契約 1) = {a 契約 |-> {a 特急券予約 }} (w 契約 2)
post 予約表に追加されている (a 予約表, a 契約, a 特急券予約, RESULT) ;
public
予約表に追加されている：予約表 * 契約 * 特急券予約 * 予約表 +> bool
予約表に追加されている (a 予約表, a 契約, a 特急券予約, a 更新後予約表) ==
  a 更新後予約表 = a 予約表 munion {a 契約 |-> {a 特急券予約 }};
.....

```

#### 10.4.9 関数：予約がある契約である

予約がある契約が判定する。

```

.....
public
予約がある契約である：契約 * 予約表 +> bool
予約がある契約である (a 契約, a 予約表) ==
  契約が存在する (a 契約, a 予約表) and
  a 予約表 (a 契約) <> {};
.....

```

#### 10.4.10 関数：契約が存在する

契約が存在するか判定する。

```

.....
public
契約が存在する：契約 * 予約表 +> bool
契約が存在する (a 契約, a 予約表) ==
  a 契約 in set dom a 予約表;
.....

```

#### 10.4.11 関数：予約集合に追加する

既存予約集合と新規予約集合を指定して、予約集合に追加する。

```

.....

```

```
public
予約集合に追加する : set of 特急券予約 * set of 特急券予約 +> set of 特急券予約
予約集合に追加する (a 既存予約集合, a 新規予約集合) ==
  a 既存予約集合 union a 新規予約集合
end

特急券予約 Domain
```

```
.....
Test Suite :    vdm.tc
Class :        特急券予約 Domain
```

Name	#Calls	Coverage
特急券予約 Domain‘契約が存在する	24	√
特急券予約 Domain‘予約表に追加する	7	75%
特急券予約 Domain‘予約表を更新する	1	√
特急券予約 Domain‘予約集合に追加する	3	√
特急券予約 Domain‘予約がある契約である	23	√
特急券予約 Domain‘予約表が更新されている	2	√
特急券予約 Domain‘予約表に追加されている	14	√
特急券予約 Domain‘予約を得る	0	0%
特急券予約 Domain‘予約を得る	4	√
特急券予約 Domain‘予約を得る	35	√
Total Coverage		83%

## 10.5 特急券予約 DomainData クラス

### 10.5.1 責任

特急券予約 Domain で扱うデータを表す。

### 10.5.2 クラス定義

.....

class

特急券予約 DomainData is subclass of 特急券予約 Domain

instance variables

protected s 予約表 : 予約表 := { |-> };

.....

### 10.5.3 操作：予約集合を得る

契約を指定して、予約集合を得る。

.....

operations

public

予約集合を得る : 契約 ==> set of 特急券予約

予約集合を得る (a 契約) ==

if dom s 予約表 = {}

then return {}

else return s 予約表 (a 契約)

pre dom s 予約表 <> {} => a 契約 in set dom s 予約表

end

特急券予約 DomainData

.....

Test Suite : vdm.tc

Class : 特急券予約 DomainData

Name	#Calls	Coverage
特急券予約 DomainData‘予約集合を得る	4	60%
Total Coverage		60%

## 10.6 特急券予約クラス

### 10.6.1 責任

特急券予約 1 件を表す。

### 10.6.2 クラス定義

.....  
`class`

特急券予約 `is subclass of` 共通定義  
.....

### 10.6.3 型定義 : 予約内容

.....  
`types`

`public` 予約内容 = `token`  
.....

### 10.6.4 インスタンス変数定義 : ID, 契約, 予約内容

.....  
`instance variables`

`sID` : ID;

`s 契約` : 契約;

`s 予約内容` : 予約内容;  
.....

### 10.6.5 構成子

.....  
`operations`

`public`

特急券予約 : ID \* 契約 \* 特急券予約 ‘予約内容 ==> 特急券予約

特急券予約 (`anID`, `a 契約`, `a 予約内容`) == `atomic`

( `sID` := `anID`;

`s 契約` := `a 契約`;

`s 予約内容` := `a 予約内容`

);  
.....

## 10.6.6 アクセッサ

```
.....
public
ID を得る : () ==> ID
ID を得る () ==
    return sID;
public
契約を得る : () ==> 契約
契約を得る () ==
    return s 契約;
public
予約内容を得る : () ==> 予約内容
予約内容を得る () ==
    return s 予約内容;
public
クレジットカードを得る : () ==> クレジットカード
クレジットカードを得る () ==
    return s 契約.クレジットカードを得る ();
public
会員証を得る : () ==> 予約会員証
会員証を得る () ==
    return s 契約.会員証を得る ()
end
特急券予約
.....
```

Test Suite : vdm.tc

Class : 特急券予約

Name	#Calls	Coverage
特急券予約 'ID を得る	81	✓
特急券予約 '契約を得る	2	✓
特急券予約 '特急券予約	8	✓
特急券予約 '会員証を得る	16	✓
特急券予約 '予約内容を得る	24	✓
特急券予約 'クレジットカードを得る	93	✓
Total Coverage		100%

## 10.7 契約クラス

### 10.7.1 責任

特急券予約の契約 1 件を表す。

### 10.7.2 クラス定義

```
.....  
class
```

```
契約 is subclass of 共通定義  
.....
```

### 10.7.3 インスタンス変数定義：おサイフケータイ, 予約会員証

```
.....  
instance variables
```

```
s おサイフケータイ : おサイフケータイ;
```

```
s クレジットカード : クレジットカード;
```

```
s 会員証 : 予約会員証;  
.....
```

### 10.7.4 構成子

実際には、契約が結ばれてから予約会員証が送られてくるまでの時差があるが、簡単化のため無視する。

```
.....  
operations
```

```
public
```

```
契約 : おサイフケータイ * クレジットカード * 予約会員証 ==> 契約
```

```
契約 (a おサイフケータイ, a クレジットカード, a 会員証) == atomic
```

```
  ( s おサイフケータイ := a おサイフケータイ;  
    s クレジットカード := a クレジットカード;  
    s 会員証 := a 会員証  
  );  
.....
```

### 10.7.5 アクセッサ

```
.....  
public
```

会員証を得る : () ==> 予約会員証

会員証を得る () ==

```
return s 会員証;
```

```
public
```

クレジットカードを得る : () ==> クレジットカード

クレジットカードを得る () ==

```
return s クレジットカード;
```

```
public
```

クレジットカードを設定する : クレジットカード ==> ()

クレジットカードを設定する (a クレジットカード) ==

```
s クレジットカード := a クレジットカード
```

```
end
```

契約

Test Suite : vdm.tc

Class : 契約

Name	#Calls	Coverage
契約 ‘契約	7	✓
契約 ‘会員証を得る	22	✓
契約 ‘クレジットカードを得る	97	✓
契約 ‘クレジットカードを設定する	1	✓
Total Coverage		100%

## 10.8 カードクラス

### 10.8.1 責任

カードの抽象クラス。

### 10.8.2 クラス定義

```
.....
class
カード is subclass of 共通定義
instance variables
protected sID : ID := nil;
protected s 暗証番号 : 暗証番号 := nil;

end
カード
.....
Test Suite :      vdm.tc
Class :          カード
```

Name	#Calls	Coverage
Total Coverage		undefined

## 10.9 クレジットカードクラス

### 10.9.1 責任

クレジットカード 1 件を表す。

### 10.9.2 クラス定義

.....  
`class`

クレジットカード `is subclass of` カード  
.....

### 10.9.3 構成子

.....  
`operations`

`public`

クレジットカード : ID \* 暗証番号 ==> クレジットカード

クレジットカード (anID, a 暗証番号) == `atomic`

( sID := anID;

  s 暗証番号 := a 暗証番号

)

`end`

クレジットカード  
.....

**Test Suite :** vdm.tc

**Class :** クレジットカード

Name	#Calls	Coverage
クレジットカード ‘クレジットカード	7	√
Total Coverage		100%

10.10 予約会員証

10.10.1 責任

予約会員証 1 件を表す。

10.10.2 クラス定義

```
.....
class
予約会員証 is subclass of カード
.....
```

10.10.3 構成子

```
.....
operations
public
予約会員証 : ID * 暗証番号 ==> 予約会員証
予約会員証 (anID, a 暗証番号) == atomic
(  sID := anID;
   s 暗証番号 := a 暗証番号
)
end
予約会員証
.....
Test Suite :      vdm.tc
Class :          予約会員証
```

Name	#Calls	Coverage
予約会員証 ‘予約会員証	7	✓
Total Coverage		100%

## 10.11 おサイフケータイクラス

### 10.11.1 責任

おサイフケータイに搭載されているスマートカードを表すドメイン・オブジェクトで、要求辞書階層の仕様である。

### 10.11.2 クラス定義

```
.....  
class
```

```
おサイフケータイ is subclass of カード  
.....
```

### 10.11.3 構成子

本モデルでは、単純化のため、おサイフケータイで改札を通る処理は対象としていないため、暗証番号設定は省略した。

```
.....  
operations
```

```
public
```

```
おサイフケータイ : ID ==> おサイフケータイ
```

```
おサイフケータイ (anID) ==
```

```
    (   sID := anID
```

```
    )
```

```
end
```

```
おサイフケータイ  
.....
```

```
Test Suite :    vdm.tc
```

```
Class :        おサイフケータイ
```

Name	#Calls	Coverage
おサイフケータイ ‘おサイフケータイ	4	√
Total Coverage		100%

## 10.12 TestApp クラス

### 10.12.1 責任

回帰テストを行う。

### 10.12.2 クラス定義

```
.....
class
```

```
TestApp
.....
```

### 10.12.3 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
```

```
run : () ==> ()
```

```
run () ==
```

```
(  dcl ts : TestSuite := new TestSuite ("特急券予約モデルの回帰テスト \n"),
```

```
    tr : TestResult := new TestResult ();
```

```
    tr.addListener(new PrintTestListener ());
```

```
    ts.addTest(new TestCaseT0001 ("TestCaseT0001 通常の予約成功 \n"));
```

```
    ts.addTest(new TestCaseT0002 ("TestCaseT0002 クレジットカードを切り替えたときの予約成
功 \n"));
```

```
    ts.addTest(new TestCaseT0003 ("TestCaseT0003 クレジットカードを切り替え、新しいクレジットカー
ドで新しい予約の特急券を得るが、古いクレジットカードでは失敗 \n"));
```

```
    ts.addTest(new TestCaseT0004 ("TestCaseT0004 クレジットカードを切り替え、新しいクレジットカー
ドで古い予約の特急券を得るが、古いクレジットカードでは失敗 \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	85%
<b>Total Coverage</b>		<b>85%</b>

## 10.13 TestCaseT0001 クラス

### 10.13.1 責任

通常の予約をテストする。

```

.....
class
TestCaseT0001 is subclass of TestCase, 共通定義
operations
public
TestCaseT0001 : seq of char ==> TestCaseT0001
TestCaseT0001 (name) ==
    setName(name);
public
test01 : () ==> ()
test01 () ==
    ( def w クレジットカード = new クレジットカード (mk_token (<クレジット ID01>), mk_token (<クレジット
    PW01>));
      w おサイフケータイ = new おサイフケータイ (mk_token (<おサイフケータイ ID01>));
      w 会員証 = new 予約会員証 (mk_token (<会員証 ID01>), mk_token (<会員証 PW01>));
      w 契約 = new 契約 (w おサイフケータイ, w クレジットカード, w 会員証);
      w システム = new 特急券予約システム ({ |-> }) in
    ( assertTrue("\ttest01 契約に失敗",
      w システム.予約集合を得る (w 契約) = {} and
      w 契約.会員証を得る () = w 会員証);
      def - = w システム.予約する (w 契約, mk_token (<予約 ID01>), mk_token (<最初の予約内容>)) in
      assertTrue("\ttest01 予約に失敗",
        w システム.特急券を得る (mk_token (<予約 ID01>), w クレジットカード) = mk_token (<最初の
        予約内容>));
      def - = w システム.予約する (w 契約, mk_token (<予約 ID02>), mk_token (<次の予約内容>)) in
      assertTrue("\ttest01 2 回目の予約に失敗",
        w システム.特急券を得る (mk_token (<予約 ID02>), w クレジットカード) = mk_token (<次の予
        約内容>))
    )
  )
end
TestCaseT0001
.....
Test Suite :      vdm.tc
Class :          TestCaseT0001

```

Name	#Calls	Coverage
TestCaseT0001'test01	1	✓
TestCaseT0001'TestCaseT0001	1	✓
<b>Total Coverage</b>		<b>100%</b>

## 10.14 TestCaseT0002 クラス

### 10.14.1 責任

クレジットカードを切り替えたときの予約をテストする。

```

class
TestCaseT0002 is subclass of TestCase, 共通定義
operations
public
TestCaseT0002 : seq of char ==> TestCaseT0002
TestCaseT0002 (name) ==
    setName(name);
public
test01 : () ==> ()
test01 () ==
    ( def w クレジットカード = new クレジットカード (mk_token (<クレジット ID01>), mk_token (<クレジット PW01>));
      w おサイフケータイ = new おサイフケータイ (mk_token (<おサイフケータイ ID01>));
      w 会員証 = new 予約会員証 (mk_token (<会員証 ID01>), mk_token (<会員証 PW01>));
      w 契約 = new 契約 (w おサイフケータイ, w クレジットカード, w 会員証);
      w システム = new 特急券予約システム ({ |-> }) in
      ( assertTrue("\ttest01 契約に失敗",
        w システム.予約集合を得る (w 契約) = {} and
        w 契約.会員証を得る () = w 会員証);
        def -- w システム.予約する (w 契約, mk_token (<予約 ID01>), mk_token (<最初の予約内容>)) in
        assertTrue("\ttest01 予約に失敗",
          w システム.特急券を得る (mk_token (<予約 ID01>), w クレジットカード) = mk_token (<最初の予約内容>));
          def w2 クレジットカード = new クレジットカード (mk_token (<クレジット ID02>), mk_token (<クレジット PW02>));
          w2 会員証 = new 予約会員証 (mk_token (<会員証 ID02>), mk_token (<会員証 PW02>));

```

```

w2 契約 = new 契約 (w おサイフケータイ, w2 クレジットカード, w2 会員証) in
(
  def -- w システム.予約する (w2 契約, mk_token (<予約 ID02>), mk_token (<次の予約内容>))
in
  assertTrue("\ttest01 2 回目の予約に失敗",
    w システム.特急券を得る (mk_token (<予約 ID02>), w2 クレジットカード) = mk_token (<
次の予約内容>))
  )
)
)
end
TestCaseT0002
.....
Test Suite :    vdm.tc
Class :        TestCaseT0002

```

Name	#Calls	Coverage
TestCaseT0002'test01	1	✓
TestCaseT0002'TestCaseT0002	1	✓
<b>Total Coverage</b>		<b>100%</b>

## 10.15 TestCaseT0003 クラス

### 10.15.1 責任

クレジットカードを切り替え、予約した特急券を得る場合をテストする。古いクレジットカードでは「Error 58: 事前条件の評価結果が false です」という実行時エラーが発生し、変更後のクレジットカードでは、特急券を得ることができる。

```

.....
class
TestCaseT0003 is subclass of TestCase, 共通定義
operations
public
TestCaseT0003 : seq of char ==> TestCaseT0003
TestCaseT0003 (name) ==
  setName(name);
public
test01 : () ==> ()
test01 () ==
  (
    def w クレジットカード = new クレジットカード (mk_token (<クレジット ID01>), mk_token (<クレジット PW01>));
    w おサイフケータイ = new おサイフケータイ (mk_token (<おサイフケータイ ID01>));
    w 会員証 = new 予約会員証 (mk_token (<会員証 ID01>), mk_token (<会員証 PW01>));

```

```

w 契約 = new 契約 (w おサイフケータイ, w クレジットカード, w 会員証);
w 古いクレジットカード = w 契約.クレジットカードを得る ();
w システム = new 特急券予約システム ({ |-> }) in
(
  assertTrue("\ttest01 契約に失敗",
    w システム.予約集合を得る (w 契約) = {} and
    w 契約.会員証を得る () = w 会員証);
  def -- w システム.予約する (w 契約, mk.token (<予約 ID01>), mk.token (<最初の予約内容>)) in
    assertTrue("\ttest01 予約に失敗",
      w システム.特急券を得る (mk.token (<予約 ID01>), w 古いクレジットカード) = mk.token (<最
初の予約内容>));
    def w2 クレジットカード = new クレジットカード (mk.token (<クレジット ID02>), mk.token (<クレ
ジット PW02>));
    w2 会員証 = new 予約会員証 (mk.token (<会員証 ID02>), mk.token (<会員証 PW02>));
    w2 契約 = new 契約 (w おサイフケータイ, w2 クレジットカード, w2 会員証);
    w 変更後のクレジットカード = w2 契約.クレジットカードを得る () in
    (
      def -- w システム.予約する (w2 契約, mk.token (<予約 ID02>), mk.token (<次の予約内容>))
in
      assertTrue("\ttest01 2 回目の予約に失敗",
        w システム.特急券を得る (mk.token (<予約 ID02>), w 変更後のクレジットカード
) = mk.token (<次の予約内容>));
      def w2 予約内容 = w システム.特急券を得る (mk.token (<予約 ID02>), w 変更後のクレジットカード
ド) in
      assertTrue("\ttest01 変更後のクレジットカードで特急券を得ることに失敗",
        w2 予約内容 = mk.token (<次の予約内容>));
      def w2 予約内容 = w システム.特急券を得る (mk.token (<予約 ID01>), w 会員証) in
      assertTrue("\ttest01 予約会員証で特急券を得ることに失敗",
        w2 予約内容 = mk.token (<最初の予約内容>));
      trap <RuntimeError>
      with print("\ttest01 テスト意図通り、古いクレジットカードで特急券を得ることに失
敗 \n") in
      def w3 予約内容 = w システム.特急券を得る (mk.token (<予約 ID02>), w 古いクレジットカード)
in
      assertTrue("\ttest01 テスト意図に反し、古いクレジットカードで特急券を得ることに成
功 \n",
        w3 予約内容 = mk.token (<次の予約内容>))
    )
  )
end
TestCaseT0003
.....

```

Test Suite : vdm.tc  
Class : TestCaseT0003

Name	#Calls	Coverage
TestCaseT0003'test01	1	95%
TestCaseT0003'TestCaseT0003	1	✓
<b>Total Coverage</b>		<b>95%</b>

## 10.16 TestCaseT0004 クラス

### 10.16.1 責任

クレジットカードを切り替え、古いクレジットカードで予約した特急券を得る場合をテストする。新しいクレジットカードでは特急券を得ることができるが、古いクレジットカードでは「Error 58: 事前条件の評価結果が false です」という実行時エラーが発生するが、「test01 テストの意図通り、古いクレジットカードで特急券を得ることに失敗」というメッセージをログに表示して、回帰テストとしては成功する。

```

.....
class
TestCaseT0004 is subclass of TestCase, 共通定義
operations
public
TestCaseT0004 : seq of char ==> TestCaseT0004
TestCaseT0004 (name) ==
    setName(name);
public
test01 : () ==> ()
test01 () ==
    ( def w クレジットカード = new クレジットカード (mk_token (<クレジット ID01>), mk_token (<クレジット PW01>));
      w おサイフケータイ = new おサイフケータイ (mk_token (<おサイフケータイ ID01>));
      w 会員証 = new 予約会員証 (mk_token (<会員証 ID01>), mk_token (<会員証 PW01>));
      w 契約 = new 契約 (w おサイフケータイ, w クレジットカード, w 会員証);
      w 古いクレジットカード = w 契約.クレジットカードを得る ();
    )

```

```

w システム = new 特急券予約システム ({ |-> }) in
(
  assertTrue("\ttest01 契約に失敗",
    w システム.予約集合を得る (w 契約) = {} and
    w 契約.会員証を得る () = w 会員証);
  def - = w システム.予約する (w 契約, mk_token (<予約 ID01>), mk_token (<最初の予約内容>)) in
  assertTrue("\ttest01 予約に失敗",
    w システム.特急券を得る (mk_token (<予約 ID01>), w 古いクレジットカード) = mk_token (<最
初の予約内容>));
  def w2 クレジットカード = new クレジットカード (mk_token (<クレジット ID02>), mk_token (<クレ
ジット PW02>));
  w2 会員証 = new 予約会員証 (mk_token (<会員証 ID02>), mk_token (<会員証 PW02>));
  w2 契約 = new 契約 (w おサイフケータイ, w2 クレジットカード, w2 会員証);
  w 変更後のクレジットカード = w2 契約.クレジットカードを得る () in
  (
    def - = w システム.予約する (w2 契約, mk_token (<予約 ID02>), mk_token (<次の予約内容>))
in
    assertTrue("\ttest01 2 回目の予約に失敗",
      w システム.特急券を得る (mk_token (<予約 ID02>), w 変更後のクレジットカード
) = mk_token (<次の予約内容>));
    def w2 予約内容 = w システム.特急券を得る (mk_token (<予約 ID01>), w 古いクレジットカード)
in
    assertTrue("\ttest01 古いクレジットカードで特急券を得ることに失敗",
      w2 予約内容 = mk_token (<最初の予約内容>));
    def w2 予約内容 = w システム.特急券を得る (mk_token (<予約 ID01>), w 会員証) in
    assertTrue("\ttest01 変更前の予約会員証で特急券を得ることに失敗",
      w2 予約内容 = mk_token (<最初の予約内容>));
    trap <RuntimeError>
    with print("\ttest01 テストの意図通り、古いクレジットカードで特急券を得ることに失
敗 \n") in
    (
      w システム.クレジットカードを切り替える (w 契約, w 変更後のクレジットカード);
      def w3 予約内容 = w システム.特急券を得る (mk_token (<予約 ID01>), w 変更後のクレジッ
トカード);

```

```
w4 予約内容 = w システム.特急券を得る (mk_token (<予約 ID01>), w 古いクレジットカ
ード) in
    (   assertTrue("\ttest01 テスト意図に反し、変更前の予約会員証で特急券を得ることに
失敗 \n",
        w3 予約内容 = mk_token (<最初の予約内容>);
        assertTrue("\ttest01 テスト意図に反し、予約会員証で特急券を得ることに成功した
が、予約内容が正しくない \n",
            w4 予約内容 = mk_token (<最初の予約内容>))
    )
)
)
)
)
end
TestCaseT0004
```

.....

**Test Suite :** vdm.tc

**Class :** TestCaseT0004

Name	#Calls	Coverage
TestCaseT0004'test01	1	90%
TestCaseT0004'TestCaseT0004	1	✓
<b>Total Coverage</b>		<b>90%</b>

## 10.17 まとめ

### 10.17.1 統計情報

注釈抜きの VDM++ ソース行数は、492 行、モデル作成工数は約 1 日、発表用の資料作成と VDM++ モデルの読みやすさのための整形清書に約 2 日かった。

なお、上記モデルの作成工数は、筆者が問題を解決するために消費した工数より少ない。



## 参考文献

- [1] Barry W. Boehm. *Software Engineering Economics*. Prentice Hall, 1981.
- [2] IPA Software Engineering Center. 厳密な仕様記述における形式手法成功事例調査報告書, 2013.
- [3] SCSK Corporation. VDM-SL 言語マニュアル. SCSK Corporation, 第 1.2 版, 2012. Revised for VDMTools V9.0.2.
- [4] SCSK Corporation. VDM++ 言語マニュアル. SCSK Corporation, 第 1.2 版, 2012. Revised for VDMTools V9.0.2.
- [5] 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・ブラット, 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.
- [6] 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.

# 索引

++, 91  
<-:, 113

atomic, 107

bool, 83

card, 80  
char, 79

dom, 80, 91

EAL6+, 17

forall, 81

GUI の捨象, 21

IC RC-SA00, 17  
in set, 81, 91  
inmap, 82  
is not yet specified, 80  
ISO/IEC 15408, 17  
ISO 標準 (ISO/IEC 13817), 12

map, 78  
munion, 91

new 式, 107

post, 80  
pre, 80

RESULT, 80

self 式, 107  
seq, 79  
seq of char, 79

token 型, 79

UML, 5

values, 80  
VDM++ とは?, 12  
VDM++ のモジュール, 78  
VDM++ モデル例, 36  
VDM による仕様作成の成果, 16  
VDM の海外での適用事例, 16  
VDM の教育, 20  
VDM の参考文献, 13  
VDM の日本での適用事例, 16

暗黙知, 22  
陰仕様, 78  
陰仕様を作成する, 32  
インスタンス変数定義, 79  
インスタンス変数のアクセス, 92  
運賃計算モデルのクラス図, 23

運賃表 (型), 129  
運賃表辞書, 129  
運賃表にただ一つ存在する, 130  
運賃表のある行に存在する, 130  
運賃表の何番目かを得る, 130  
運賃表の不変条件, 129  
運賃を得る, 128  
駅集合, 140  
駅務システム, 17  
特急券予約システムの仕様の階層, 37  
特急券予約システムのまとめ, 66  
特急券予約システムの例, 34  
応当日, 8  
おサイフケータイ・ファームウェアの開発, 16  
オブジェクト指向的モデルの行数, 104  
オブジェクト指向分析・設計手法 (OOA/OOD), 32  
オブジェクト指向モデル, 104  
オムロンの事例, 17  
オランダの花市場オークションシステム事例, 27  
回帰テスト, 24, 25, 66  
回帰テストのエラーケース例, 146  
回帰テストのノーマルケース例, 144  
開発管理, 27  
開発工程別の欠陥修正コスト, 4  
開発体制, 28  
下限より上限が大きい, 130  
貸出可能である, 84  
貸出可能でない, 85  
貸出から削除されている, 84  
貸出情報, 113  
貸出に存在しない, 84  
貸出に存在する, 84  
貸出に追加されている, 84  
貸出の状態に関する関数, 87, 95, 114  
型チェック, 24  
型定義, 78  
カプセル化, 35  
距離に応じた運賃を得る, 129  
擬似コードの文法, 5  
業務知識の蓄積, 6  
業務論理とシナリオとテストケースの対応関係, 22  
組合せテスト機能, 24  
組合せテスト, 24  
組合せテスト実行画面例, 26  
クラス図 (特急券予約システム), 37  
クレジットカードを切り替える, 40, 152  
グローバル変数, 83  
経験的根拠, 5  
形式手法 VDM とは?, 12  
形式手法の検証方法とツール, 13  
形式手法と VDM の有用性, 70  
形式手法導入のコツ, 71  
形式手法の種類, 12  
契約, 162  
検索キーが空でない, 85  
構成子, 107  
構造化分析・設計手法 (SA/SD), 32  
構文チェック, 24  
国際語化, 16

- 構造化日本語仕様としての VDM, 70
- コードカバレッジ, 24
- コードカバレッジ (命令レベル), 40
- 最短経路, 136
- 最大貸出数を超えていない, 85, 88, 96
- 再利用性, 35
- シナリオ階層, 21
- 修正後のクラス図, 150
- 証券会社バックオフィスシステム, 16
- 詳細な要求仕様, 36
- 証明課題, 12
- 証明課題生成, 24
- 証明課題レビュー, 66
- 職員, 117
- 書庫, 109
- 仕様記述, 20
- 仕様記述言語を読むのは容易, 5
- GUI の捨象, 21
- 仕様記述フレームワークの設定, 35
- 仕様テスト, 24
- 仕様の階層 (特急券予約システム), 36
- 仕様の修正に弱い, 5
- 仕様記述フレームワーク, 20
- 仕様記述フレームワーク (運賃計算モデル), 22
- 仕様記述フレームワーク (日本フィッツ), 21
- 仕様記述フレームワークの概要, 21
- 仕様記述フレームワークの具体例, 21
- 仕様ライブラリ, 23
- 仕様を動かす, 6
- 仕様を分割統治する, 32
- 事後条件, 24, 80
- 事前条件, 24, 80
- 実行可能仕様, 24
- 実行可能な仕様 (図書館 1), 90
- 実行不可能な仕様, 78
- 述語から、関数・操作のインタフェースを記述する, 32
- 上限と次の行の下限は等しい, 130
- 状態を定義する, 32
- 正規表現記述, 24
- 静的検証, 66
- 静的検証を行う, 32
- セキュリティ強化版の FeliCa チップ, 17
- 総合テスト, 27
- 蔵書を蔵書 ID から本への写像, 78
- 蔵書が削除されている, 83
- 蔵書が追加されている, 83
- 蔵書に存在しない, 83
- 蔵書に存在する, 83
- 蔵書の検索に関する関数, 88, 96, 110
- 蔵書の状態に関する関数, 86, 94, 110
- 蔵書を削除する, 81, 91, 109
- 蔵書を追加する, 80, 90, 109
- 大域変数, 83
- ダイクストラ算法, 137
- 対象者, x
- 多重代入文, 107
- 単体テスト, 27
- 題名と著者と分野で本を検索する, 91
- 抽象的なモデル, 36
- 著者, 106
- 定義域削減演算子, 92, 113
- 定数定義, 80
- 鉄道用ダイクストラ算法, 136
- 鉄道ネットデータ, 140
- 到達目標, x
- 図書館 0, 78
- 図書館システム, 74
- 図書館システムへの要求, 74, 78
- 図書館の本, 6
- 図書館要求辞書, 94
- 図書館 0 要求辞書, 86
- 図書館 2 モデルのクラス図, 104
- 特急券予約, 47
- 特急券予約システム, 6, 38, 151
- 特急券予約システムモデル化の統計情報, 67
- 特急券予約システムを VDM++ で記述したモデル, 36
- 特急券予約 Domain, 42, 155
- 特急券予約 DomainData, 46, 159
- 特急券予約, 160
- 特急券を得る, 39, 151, 152
- 動的検証, 66
- 動的チェック, 24
- 内包式 (写像), 91
- なぜ VDM を選択したか?, 12
- 名前付け規則, 79
- 日本語識別子, 16
- 日本語の仕様中の擬似コードと、VDM の比較, 71
- 日本語仕様の曖昧さの例, 6
- 日本語による意思疎通, 5
- 日本フィッツの開発体制, 28
- 日本フィッツの事例, 16
- 版管理ツール, 27
- 非接触 IC カードチップ, 17
- 人, 116
- フィールド, 81
- フェリカネットワークスの開発体制, 28
- フェリカネットワークスの事例, 16
- 副作用が無い, 107
- 不変条件, 24, 79
- フレームワークやライブラリの構築が容易, 5
- 分野, 112
- プログラミング, 25
- プログラムのテスト, 27
- プログラムは数学系, 4
- 保守性, 35
- 本, 107
- 本書の構成, x
- 本 (レコード型), 79
- 本を返す, 82, 92, 113
- 本を貸す, 82, 91, 113
- 本を検索する, 81, 110
- 名詞から型を定義する, 32
- 網羅性検査, 24
- モデル化する機能の範囲選択, 35
- モデル化対象用語の選択, 34
- モデル化の一般手順, 32
- モデル化の範囲を決める, 32
- モデルの検証, 66
- モデルの正当性と妥当性を動的に検証・確認する, 32
- 問題点の要約, 34
- 問題は何だったのか, 66
- ユースケースレベルの要求仕様, 74
- 要求仕様をレビュー, 33
- 要求辞書, 35
- 要求辞書階層, 22
- 要求分析工程, 4
- 陽仕様, 90
- 陽仕様を作成する, 32
- 予約がある契約である, 44
- 予約集合に追加する, 44
- 予約集合を得る, 46, 159
- 予約する, 38, 151
- 予約表に追加する, 44, 157
- 予約表を更新する, 43, 156
- 予約を得る, 42, 43, 155, 156
- 利用者, 118
- 理論的根拠, 4
- 路線検索, 134
- 路線集合, 140
- 路線単位, 140

路線单位集合, 140  
路線網, 132

実務家のための形式手法  
厳密な仕様記述を志すための形式手法入門

## 対象を如何にモデル化するか?

2013 年 3 月 初版発行

独立行政法人情報処理推進機構  
技術本部 ソフトウェア・エンジニアリング・センター  
統合系システム・ソフトウェア信頼性基盤整備推進委員会  
上流品質技術部会 人材育成WG委員

主査	荒木 啓二郎（国立大学法人九州大学）	はじめに執筆
副主査	山本 修一郎（国立大学法人名古屋大学）	
委員	鯨坂 恒夫（国立大学法人和歌山大学）	
	石黒 正揮（株式会社三菱総合研究所）	
	岡本 勝幸（有限会社イトワークス）	
	栗田 太郎（フェリカネットワークス株式会社）	
	佐原 伸（SCSK 株式会社、法政大学）	I 部、II 部執筆
	塚本 英昭（日本電信電話株式会社）	
	西原 秀明（独立行政法人産業技術総合研究所）	
	藤枝 純教（グローバル情報社会研究所株式会社）	
	結縁 祥治（国立大学法人名古屋大学）	

オブザーバ 日下部 茂（国立大学法人九州大学）

事務局	新谷 勝利
	神谷 慎吾
	中谷 浩康
	藤原 由起子