4. デザインパターンによる OOA/OOD 開発技法

この章では、デザインパターンを使ったオブジェクト指向分析・設計の開発技法について述べる。事務系の図書館貸出管理システムを例に具体的に説明する。制御系のシステムについては 5 章で記述する。

モデルの記法は UML[1] を採用し、仕様記述言語は UML の仕様記述言語に採用された OCL[2] を使用する。

4.1 デザインパターンを使ったシステム開発 の概要

オブジェクト指向システムは、以下の４つの工程を短期間[3]に繰り返していくことで開発する。

(1) **ドメイン分析**

システムの対象となる問題領域 (ドメイン) は何をしているかをモデル化し、ドメインモデルを作成する。

(2) **システム分析**

作成するシステムが「何をすべきか（What)」をモデル化し、分析モデルを作成する。

---

1. 『UML Notation Guide version 1.1, Rational Software, 1 September 1997』および付録 7.1 参照。
2. 『Object Constraint, Language Specification,Rational Software, 1 September 1997』および付録 7.2 参照。
3. プロジェクトで採用する開発環境やオブジェクト指向言語により異なるが、通常、1 週間から 3ヶ月くらいで 1 サイクルを完了させる。ソフトウェア工学的開発技法は、このように短期間で開発することにより、ユーザー要求との不一致をできるだけ早期に検出しようとする。

(3) 設計

作成するシステムを「どう構築するか（How)」をモデル化し、設計モデルを作成する。

(4) 実装

設計結果を、Smalltalk や Java といったオブジェクト指向言語や C++ のようなオブジェクト指向風言語で実装する。

従来の開発技法では、このような各工程ごとに異なる種類のモデルやドキュメントを作成していたのだが、オブジェクト指向分析・設計法は記法的にすべて同じものを使う。すなわち、ドメインモデルと分析モデルと設計モデルは、記法が同じなのである。もちろん、各工程ごとに視点が異なるので、モデルの意味や内容は異なってくる。

短期間に実装までを行うのだから、当然、当初実装する機能は少なくしなければならない。すなわち、最重要機能のみ 1 サイクル目で実装するのがよい。

デザインパターンは各工程で使用できるか検討するが、ドメイン分析やシステム分析工程では「分析パターン」の方が使用できる可能性が高い。設計工程ではデザインパターンとアーキテクチャパターンの適用を検討する。また、実装時にはイディオム[4] の適用を検討する。

各工程共通の開発プロセスがあり、以下の通りである。詳細は後述する。

- **モデルの作成と修正**

  クラスとオブジェクトの認識を行い、デザインパターンや分析パターンなどの内で適当なパターンがあればそのパターンを適用する。

- **勘所（Hotspot）の発見**

  システムのライフサイクルを考え、その間に不変の箇所と、変更が多発する箇所を認識し、別モジュールにすることにより、変更余波 (変更したときの影響度合) を最小にする。

  変更多発箇所を勘所（Hotspot）と呼ぶ。

4. Smalltalk で実装する場合は、『青木淳　著、「Smalltalk イディオム」、SRC、1997 年』参照

- **仕様記述**

  できるだけ自然言語を使わず、仕様記述言語で制約や操作[5] 仕様を記述する。特に、ドメイン分析とシステム分析では、従来の開発で行われていたように、あたかもプログラミングをしているような「手続き的な仕様」を書くのではなく、操作が終わるときにどのような「状態」になっていればよいかを記述する「宣言的な」仕様を書く[6]。

- **プロトタイプ作成**

  各工程のモデル作成後、必要ならプロトタイプ (システムの模型) を作成する。例えば、ドメイン分析やシステム分析工程では、ユーザーの要求にあったシステムかどうかをプロトタイプで確認する[7]。設計工程では、システムが構築可能かどうか確認したり、効率を測定するためにプロトタイプを作成する。設計工程の場合は、プロトタイプだけでなくシミュレーション・モデル[8] を構築してシミュレーションを行うことも必要な場合がある。

- **モデルの検証**

  各工程のモデル構築が終わった段階で、モデルの検証を行う。具体的なテストケースを想定し、机上で「モデルを実行する」ことによって検証を行う。

4.2 ドメイン分析

システムの対象となる問題領域 (ドメイン) は何をしているかをモデル化し、ドメインモデルを作成する。ドメイン分析で行うべき作業は、以下の通りである。最初の２つ（ビジネスゴールの作成と用語集の作成）は、オブジェクト指向開発技法以前から必要な作業で、重要性はオブジェクト指向開発になっても変わらない。

---

5. Smallltalk 風に言えばメソッド、C++ 風に言えばメンバー関数。
6. 宣言的仕様については後述する。
7. この際、GUI の詳細はモデル化にとって重要ではないので、凝ったGUI 構築は避ける。
8. 「(株) スリースカンパニー、意志決定支援のための多目的シミュレーションツール EX・TD　マニュアル、Imagine That, Inc、1994」参照。

- **ビジネスゴールの作成**
- **用語集の作成**
- **UseCase 作成および制約の発見**
- **ドメインオブジェクトの発見**
- **オブジェクトの状態変化の記述**
- **要求仕様記述**
- **ドメインモデルの検証**

以下に、各作業の内容を述べる。

4.2.1 ビジネスゴールの作成

システムを作成する際、通常、ユーザーは自分たちの真の要求を理解していない。また、ユーザーの意見が統一されていることも少ないのであって、ユーザーごとあるいはユーザーの組織ごとにシステムの最終目標であるビジネスゴールが異なることすらある。

そこで、ビジネスゴールとユーザーのマトリクスを作成し、今回作成するシステムがどこまでやるかを整理しておかなければならない。

例えば、図書館の貸出管理システムでは以下の図のようになる。

	管理者	司書	システム管理者
操作が容易な貸出返却機能		○	
セキュリティ保持	○		
各種の検索機能		○	
運用の容易性	○		○
3-Tier構成			○

図 36 ビジネスゴール (図書館) マトリクス

もちろん、上の図のマトリクスは、本書で使用する例題を使っているので小さなものだが、実際のシステム開発ではかなり大きなものになる。

ビジネスゴールには、機能要求や設計方法や性能要求など種々雑多なものが含まれるが、それらをすべて記述しておく。ただし、現在の工程がドメイン分析工程であることを忘れてはいけない。ドメインモデルを作成する際には、ビジネスゴールのうち最も重要な機能を対象にする。設計方法や性能要求などはシステム設計工程の楽しみとして残しておく。システムが「何をすべきか」を定義する分析工程で、決して「どう作るか」を議論してはならないというのが、ソフトウェア工学の鉄則である。

4.2.2 用語集の作成

システム開発の最中に使用する用語を定義するのが用語集である。用語集の作成は、大規模システムでは最も重要な作業となる。用語集が正確にできているかどうかで、そのプロジェクトの状態が分かると言っても過言ではないほどである。オブジェクト指向開発で「用語」はオブジェクト名や操作名や関連名に直結することが多いので、特に重要となる。

ドメイン分析工程では、ドメインに固有な用語やその他主要な用語を定義する。曖昧になりがちな用語としては、「ユーザー」「組織」「～情報」「～種別」といったものが挙げられる。

例えば、図書館の貸出管理システムを例に取ってみると、システムのユーザーは職員だけなのか？ 職員と言っても司書だけで管理職は除くのかどうか？ 利用者はユーザーになり得ないのかどうか？ といった点が、用語集がないと曖昧になる。これらの用語はクラスに直結するので、用語の意味が変わるとクラス図が変わってしまい大きな影響を及ぼす。従って、以下のような用語集を作っておかなければならない。

- **利用者**

  貸出管理システムの利用者。図書館の本を借りることのできる人と、貸出を行う職員からなる。

- **職員**

  司書の資格を持つ職員。本の貸出を行うことができる。

- **蔵書**

  図書館で管理しなければならない個々の本。従って、同じ題名の蔵書が複数存在する。

- **本**

  題名・著者が同じ本を 1 つと考える場合の、抽象概念としての「本」。

- **本の実体**

  物理的な個々の本。このうち、図書館で管理しているものが「蔵書」。

4.2.3UseCase 作成および制約の発見

システムのユーザーにインタビューしたり、対象とする問題領域の本を見たり、旧システムのマニュアルを読んだりして UseCase を作成する。UseCase[9] はユーザーの要求事項を機能単位でまとめたものである。ただし、オブジェクト指向なので、能動的なオブジェクトを見つけ、その周りに機能をまとめる方向で記述していく。

機能の記述であるから、UseCase は下図のような階層（入れ子）構造になる。

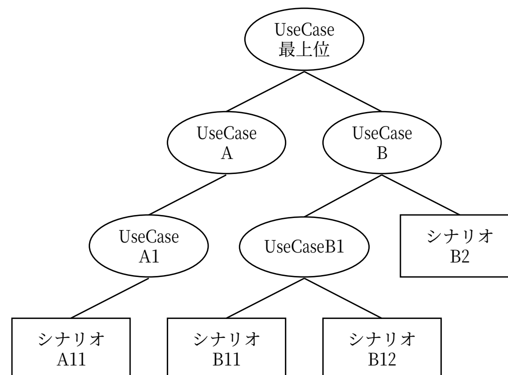

図 37 UseCase の階層

9. 『I. **ヤコブソン**，**他著、西岡・渡邊・梶原監訳、オジェクト指向ソフトウェア工学** OOSE use-case **によるアプローチ、アジソン　ウェスレイ・トッパン、**1995』 参照

Jacobson が提唱した UseCase では、あとで紹介する UseCase 図と自然言語による UseCase の記述を、共に UseCase と呼んでいる。本書では、このような呼び方は紛らわしいので、自然言語で記述する UseCase の詳細を「シナリオ」と呼ぶことにする。さらに、シナリオを、自然言語でなく仕様記述言語である OCL で書くことにする。また、シナリオを実行したあとの状態を記述する後件（post-condition）を重視する。シナリオ実行前の状態を書く前件（pre-condition）とシナリオを使用する文脈も記述する。

では、図書館の貸出管理システムの主な UseCase を見てみよう。UseCase の目次は下図のようになる。四角の部分が「シナリオ」で、それ以外の楕円部分が図で表した UseCase である。

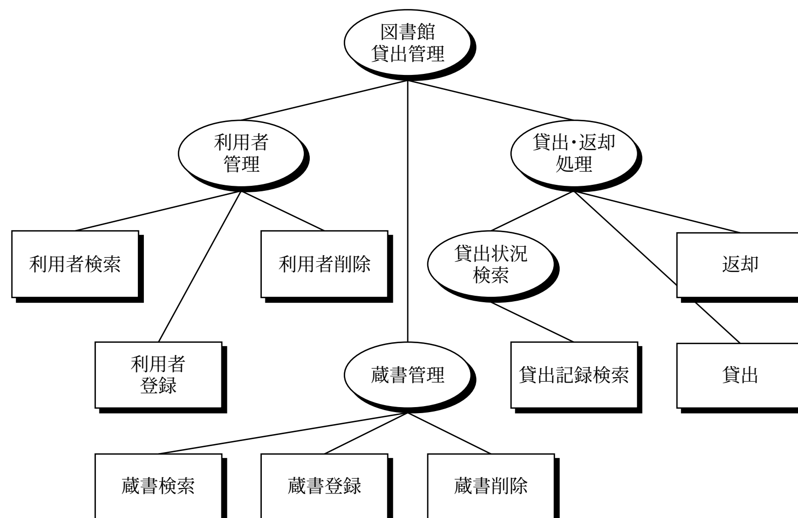

図 38 図書館貸出管理システムの UseCase 目次

図書館貸出管理のトップレベルの UseCase は下図のようになる。

トップレベルの UseCase 図は、システムをその内部と外部に分け、境界を明確にする働きもある。この UseCase の場合、司書と利用者がシステムの境界にいること

を示す。人間としての司書や利用者はこのシステムの外にいて、司書と利用者がどのような話をするかシステムは関知しないことになる。利用者が「この本を借りたい」と司書に言ってもシステムは関知せず、司書が貸出処理を行ってはじめてそれに気が付くわけである。

もちろん、司書と利用者の機能の一部を代行するオブジェクトはこのシステム内に作ることになるだろう。だからといって、司書と利用者のクラスが、人間の司書と利用者のすべての振る舞いを表すわけではない。司書と利用者のクラスを作るという「抽象化」は、実は分析者の「偏見」によって実物(人間) としての司書や利用者の振る舞いを削ぎ落としているのである。

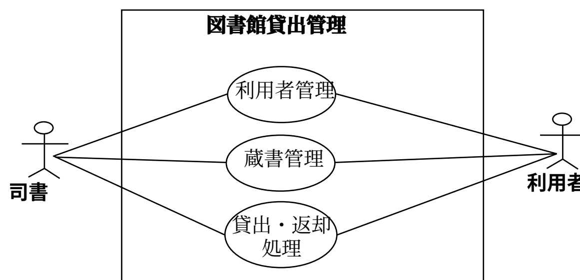

図 39 図書館貸出管理トップレベルの UseCase 図

上の UseCase 図では、図書館貸出管理システムが「利用者管理」「蔵書管理」「貸出・返却処理」の3 つの UseCase から成り、それぞれの UseCase は能動的なオブジェクトとして「司書」と「利用者」と関係があることを示している。

貸出・返却処理の UseCase は下図のようになる。破線の矢印は、依存関係を示す。例えば、貸出は利用者検索と蔵書検索に依存している。

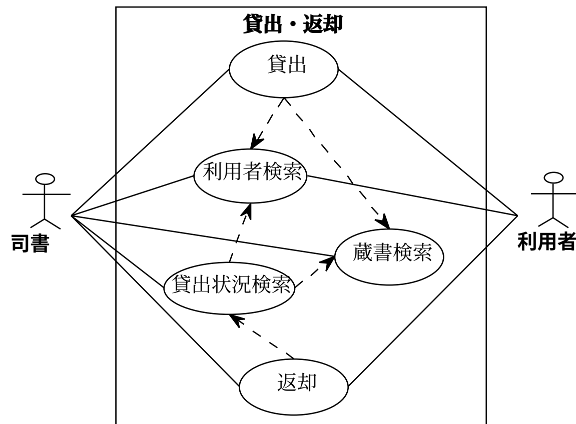

図 40 貸出の UseCase 図

貸出のシナリオは以下のようになる。ここで、前件（pre-condition）にはシナリオ実行前の状態を記述し、後件（post-condition）は、シナリオ実行後の状態を記述する。従来型の手続き的仕様記述に対して、このような仕様記述法を宣言的仕様記述と呼ぶ。手続き的仕様だとプログラミングと同じ手間がかかるのに対し、宣言的仕様だと、どのようになって欲しいかだけを書くので、若干の慣れは必要だが、記述は簡単になる。[10]

なお、仕様記述言語 OCL の記法については、脚注で説明していく。内容は自然言語による注釈でも補っていくので、当面はこちらだけを読んでいっても構わない。OCL 全体の要約は付録を参照していただきたい。

仕様 2 貸出のシナリオ

- **シナリオ名**

  貸出

10. ただし、実行不可能な仕様も容易に記述できるので注意が必要である。

- **概要**

  蔵書の貸出を行う。

- **文脈**

  登録された利用者から貸出を請求されたときに、図書館の司書が行う。

  利用者が借りることのできる蔵書は合計 5 冊までである。

- **基本パス [貸出可能]**[11]

- **前件**

  -- [12] 今までに利用者が借りた蔵書の冊数は 5 未満で、

  -- かつ今回貸し出す蔵書は貸出可能でなければならない。

  Set(利用者が借りた蔵書)[13] ->size < 5 and 貸し出す蔵書 . 貸出可能

- **後件**

  -- 貸し出す前の蔵書の集合から貸し出す蔵書を削除したものが、

  -- 貸出後の蔵書の集合である。

  Set(蔵書) = Set(蔵書)@pre[14] − Set{ 貸し出す蔵書 }[15]

- **代替パス [貸出不能]**[16]

- **前件**

  -- 今までに利用者が借りた蔵書の冊数が 5 冊以上か、

  -- または、今回貸し出す蔵書に貸出不能のものがある。

  Set{ 利用者が借りた蔵書 }->size >= 5 or 貸し出す蔵書 . 貸出可能でない

---

11. 基本パスは、シナリオの主な流れを記述する。[ ] 内は注釈。
12. -- は OCL の注釈を表す記法。
13. Set() の記法は、() 内のクラスのインスタンスの集合を表す。
    -> は集合の操作を示す記号であり、size は既定義の操作で、集合の要素数を返す。
    従って、Set(利用者が借りた蔵書)->size で集合の要素数を示す。
14. @pre は後件の中で使用できる記号で、シナリオや操作の動く前の「値」を示す。−は−の左側の集合から−の右側の集合または要素を削除することを示す。
15. Set{ 要素リスト } は、要素リストに記述されたインスタンスからなる集合を示す。
16. 代替パスは、シナリオの副次的な流れを記述する。エラー処理の記述になることが多い。2 つ以上の代替パスを記述してもよい。[ ] 内は注釈。

- **後件**

  -- 貸し出す前と後の蔵書の集合は等しく、貸出不能状態になっている。

  Set(蔵書) = Set(蔵書)@pre and 貸出不能状態[17]

返却のシナリオは以下のようになる。

仕様 3 返却のシナリオ

- **シナリオ名**

  返却

- **概要**

  蔵書の返却を行う。

- **文脈**

  登録された利用者が返却した時に、図書館の司書が行う。

- **基本パス [返却]**

- **前件**

  -- 返却された蔵書は利用者が借りた蔵書の要素であり、

  -- いずれも空でない。

  Set{ 利用者が借りた蔵書 }->notEmpty[18] and

  Set{ 返却された蔵書 }->notEmpty and

  Set{ 利用者が借りた蔵書 }->includes(返却された蔵書)[19]

- **後件**

  -- シナリオが動く前の蔵書の集合と返却された蔵書からなる集合の和集合が、

  -- 新たな蔵書の集合となっている。

17. 貸出不能状態の詳細は、ドメイン分析工程では定義しない。
18. notEmpty は既定義の操作で「空でなければ真」の意味。
19. includes も既定義の操作で、( ) 内のオブジェクトが「-> の左辺の集合」の要素であれば真を返す。

Set(蔵書) = Set(蔵書)@pre union Set{ 返却された蔵書 }[20]

貸出と返却のシナリオの後件を見てみると、いずれも、集合演算がその本体である。要するに、貸出は図書館の蔵書から貸し出した本の集合を削除し、返却はそれらを元に戻す。実は、事務処理の多くは集合演算で後件を書くことで、簡潔に「やりたいこと」を記述することができる。

OCL の集合演算の記法は数学のそれよりやや分かりにくいが、記号が ASCII 記号のみでできているので、ワープロや CASE ツールなどで仕様を書くのに特別なことをしなくてよいという利点がある。

「集合はどうも苦手で」と言う人もいるが、今や中学校でも教えている概念なので、ソフトウェアの分析や設計に必須の概念として勉強していただきたい[21]。

だいぶ OCL の -> という記法と既定義の操作が出てきたので、少し説明しよう。OCL では、ものの集まりを表す Collection という型があって、Set（集合）や Bag（重複のあるものの集まり）や Sequence（順序のあるものの集まりで、要素の重複が許される）という型のスーパー型[22] になっている。OCL は型とクラスを同一視している。そして、クラスに定義された特性(Property)[23] を指定するためにピリオド（.）を使い、Collection 型に定義されている特性を指定するために矢印記号（->）を使っている。

例えば、下図のように「会社」と「人」というクラスの間に関連があり、「従業員」というロール名が付いているとする。人クラスは属性として名前と誕生日を持ち、操作として収入を持つ。操作収入のパラメータは Date クラスのインスタンスを持つ。一方、会社クラスは属性として名前を持ち、操作として従業員数と株価を持つ。

20.union も既定義の操作で、2 つの集合の和集合を返す。

21.C.L リュー著、成嶋弘・秋山仁訳、コンピュータサイエンスのための離散数学入門、マグロウヒル、1990 年、参照のこと。

22. 型はスーパー型の操作を継承する。例えば、Set は Collection の操作を継承する。

23. クラスや型の操作あるいは属性あるいはロール名などのこと。

人
名前
誕生日
収入(Date)

管理者
0..1
従業員
＊
雇用者
1

会社
名前
従業員数()
株価()

図 41 会社クラスと人クラスの関連の例

「会社」の「従業員」（すなわち「人」クラスのインスタンスのうち「従業員」である人たちの集合）の数を求めたいとする。そのためには、

会社
self. 従業員 ->size

と記述すればよい。ここで、会社はクラス名（ 文脈上自明の時は省略してよい)、self はそのクラスの着目しているインスタンス自身（すなわち会社のインスタンス）を示し、ピリオドはインスタンスの要素の特性 (この場合、ロール名「従業員」) を指定するのに使い、-> はインスタンスの集合の特性を指定するのに使う。同じ特性指定でも、対象が個別のインスタンスなり要素なのか、集合やCollection なのかで指定方法が異なるのである。

今の場合、self. 従業員で「人」クラスのインスタンスの集合を指定したことになる。そして、その従業員の集合に ->size 操作を適用し、結果としてその集合の要素数（つまり従業員数）を求めることになる。

会社クラスの従業員数操作を使って、

会社
self. 従業員数

としても、もちろん従業員数は求まるが、従業員数操作の中身は

self. 従業員 ->size

そのものである。

ピリオド（.）で特性を指定する例を、下記の蔵書削除のシナリオで見てみよう。

仕様 4 蔵書削除のシナリオ

- **シナリオ名**

  蔵書削除

- **概要**

  蔵書の削除を行う。

- **文脈**

  特定の本の実体を削除したいとき。

- **基本パス [蔵書削除]**

- **前件**

  -- 蔵書の集合が空でなく、削除する蔵書と同じ本の蔵書がある。

  Set(蔵書)->notEmpty and

  削除する蔵書 . 本 . 蔵書 ->size > 2[24]

- **後件**

  -- 削除する前の蔵書の集合と削除する蔵書からなる集合の差が

  -- 削除後の蔵書の集合と等しい。

  Set(蔵書) = Set(蔵書)@pre − Set{ 削除する蔵書 }

- **代替パス [図書館の本削除]**

- **前件**

  -- 蔵書の集合が空でなく、削除する蔵書と同じ本の蔵書は他にない。

  Set(蔵書)->notEmpty and

  削除する蔵書 . 本 . 蔵書 ->size = 1[25]

---

24. 削除する蔵書を含めて同じ「本」の蔵書が 2 冊以上あれば、削除しても蔵書は 1 冊は残るので、「本」を削除する必要がないが、その「本」の蔵書が 1 冊もなくなれば、その「本」も削除しなければならない。
25. 削除する蔵書と同じ「本」の蔵書は他にない。「削除する蔵書」は集合でないのでピリオド（.）記法を使って特性を指定する。

- **後件**

  -- 削除前の蔵書集合から削除する蔵書を除いた集合が、

  -- 削除後の蔵書の集合と等しく、

  -- 削除前の図書館の本の集合から削除する蔵書の本を除いた集合が、

  -- 削除後の図書館の本の集合と等しい。

  Set(蔵書) = Set(蔵書)@pre − Set{ 削除する蔵書 } and

  Set(図書館の本) = Set(図書館の本) @pre − Set{ 削除する蔵書 . 本 }[26]

その他の UseCase とシナリオを以下に示す。

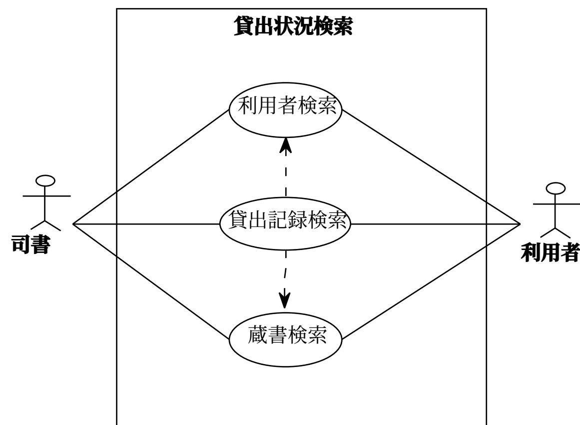

図 42 貸出状況検索の UseCase 図

仕様 5 貸出記録検索のシナリオ

- **シナリオ名**

  貸出記録検索

26. 要するに蔵書と本の情報の両方を削除する。

- **概要**

  ある利用者が借りている蔵書の検索と、ある蔵書を借りている利用者の検索を行う。

- **文脈**

  図書館の司書が「利用者が借りている蔵書を検索したいとき」と「蔵書を借りている利用者の検索をしたいとき」に行う。

  利用者が自分の借りている本を検索したい時に行う。

- **基本パス [利用者による検索]**
- **前件**

  -- 利用者集合は空でなく、操作者は利用者の集合の要素である。

  Set(利用者)->notEmpty and Set(利用者)->includes(操作者)[27]

- **後件**

  -- 操作者が司書でないときは (一般利用者なので)

  -- 操作者の借りている蔵書の集合が結果となる。

  -- そうでなければ、利用者集合の借りている蔵書の集合が結果となる。

  if[28] Set{ 操作者 }->intersection[29] (Set(司書))->isEmpty then

  result[30] = 操作者 . 借りている蔵書

  else

  result = Set{ 利用者 }-> 借りている蔵書

  endif

- **代替パス [蔵書による検索]**
- **前件**

  -- 蔵書が空でない。

---

27. 操作者が利用者集合の要素であれば includes 操作は真を返す。
28. if 条件式 then (A) else (B) endif は、Boolean 型の操作で（つまり if 文ではない！）、条件式が真なら式 A を評価した値を返し、偽なら式 B を評価した値を返す。
29. intersection 操作は２つの集合の共通集合を返す。
30. result は操作の結果を返すオブジェクトの名前である。

Set(蔵書)->notEmpty

- **後件**

  -- 蔵書の借用者の集合が結果となる。

  result = Set(蔵書)-> 借用者

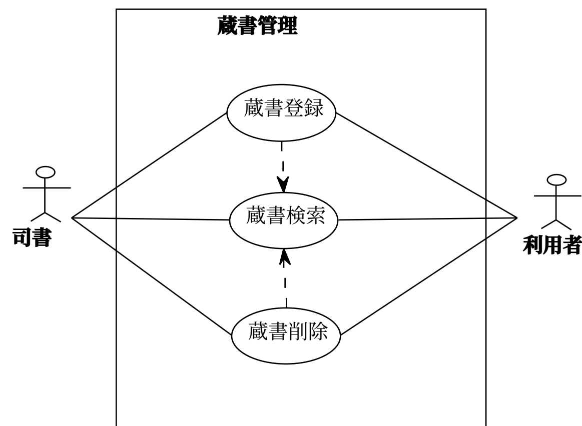

図 43 蔵書管理の UseCase 図

仕様 6 蔵書登録のシナリオ

- **シナリオ名**

  蔵書登録

- **概要**

  蔵書の登録を行う。

- **文脈**

  特定の本の実体を登録したいとき。

- **基本パス [蔵書登録]**
- **後件**

  -- 登録する前の蔵書の集合に、登録する本の実体を追加した集合と、

  -- 登録後の蔵書の集合が等しく、かつ

  -- 登録する前の図書館の本の集合に、登録する本を追加した集合が、

  -- 登録後の図書館の本の集合と等しい。

  Set(蔵書) = Set(蔵書)@pre union Set{ 登録する本の実体 } and

  Set(図書館の本) = Set(図書館の本)@pre union Set{ 登録する本 }

仕様 7 蔵書検索のシナリオ

- **シナリオ名**

  蔵書検索
- **概要**

  蔵書の検索を行う。
- **文脈**

  特定の蔵書を検索したいとき。

  特定の作者の蔵書を検索したいとき。

  特定の分野の蔵書を検索したいとき。

  操作と条件の指定は図書館の司書が行う。
- **基本パス [蔵書検索]**
- **前件**

  -- 指定した本の集合が空でないこと。

  Set(図書館の本)->select[31] (本の指定条件)->notEmpty

31.select も既定義の操作で、( ) 内の条件を満たす要素の集合を返す。

- **後件**

  -- 指定した本に対応する蔵書の集合を結果とする。

  result = Set(図書館の本)->select(本の指定条件)-> 蔵書

- **代替パス [作者による検索]**

- **前件**

  -- 指定した作者の集合が空でなく、その作者の著書の集合も空でないこと。

  Set(作者)->select(作者の指定条件)->notEmpty and

  Set(作者)->select(作者の指定条件)-> 著書 ->notEmpty

- **後件**

  -- 指定した作者の著書の集合に対応する蔵書の集合を結果とする。

  result = Set(作者)->select(作者の指定条件)-> 著書 -> 蔵書

- **代替パス [分野による検索]**

- **前件**

  -- 指定した分野の集合が空でなく、その分野の本の集合も空でないこと。

  Set(分野)->select(分野の指定条件)->notEmpty and

  Set(分野)->select(分野の指定条件)-> 本 ->notEmpty

- **後件**

  --- 指定した分野の本の集合に対応する蔵書の集合を結果とする。

  result = Set(分野)->select(分野の指定条件)-> 本 -> 蔵書

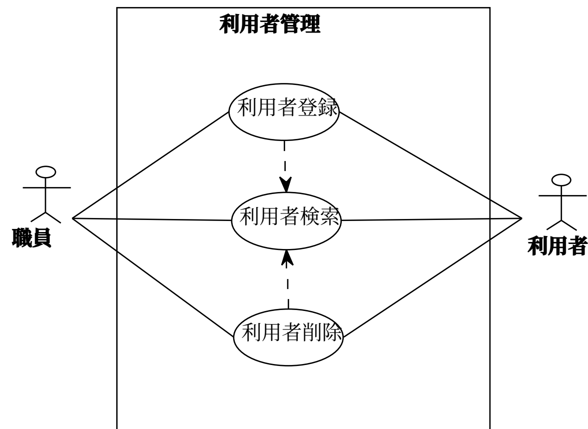

図 44 利用者管理の UseCase 図

仕様 8 利用者登録のシナリオ

- **シナリオ名**

  利用者登録

- **概要**

  利用者の登録を行う。

- **文脈**

  特定の利用者を登録したいときに司書が行う。

- **基本パス [利用者登録]**

- **後件**

  -- 登録する前の利用者の集合に登録する利用者を追加した集合と、

  -- 登録後の利用者の集合が等しい。

  Set(利用者) = Set(利用者)@pre union Set{ 登録する利用者 }

仕様 9 利用者削除のシナリオ

- **シナリオ名**

  利用者削除

- **概要**

  利用者の削除を行う。

- **文脈**

  特定の利用者を削除したいとき。

- **基本パス [利用者削除]**

- **前件**

  -- 利用者の集合が空でなく、削除する利用者が本を借りていないこと。

  Set(利用者)->notEmpty and 利用者 . 借りている本 ->isEmpty

- **後件**

  -- 削除する前の利用者の集合から削除する利用者を除いた集合が、削除後の利用者の集合と等しい。

  Set(利用者) = Set(利用者)@pre − Set{ 削除する利用者 }

- **代替パス [利用者削除不能]**

- **前件**

  -- 利用者の集合が空でなく、削除する利用者が本を借りている。

  Set(利用者)->notEmpty and

  利用者 . 借りている本 ->notEmpty

- **後件**

  -- 削除前と後の利用者の集合は等しく、利用者削除不能状態になる。

  Set(利用者) = Set(利用者)@pre and 利用者削除不能状態[32]

32. 利用者削除不能状態の詳細をドメイン分析で詰める必要はない。

仕様 10 利用者検索のシナリオ

- **シナリオ名**

  利用者検索

- **概要**

  利用者の検索を行う。

- **文脈**

  特定の利用者を検索したいとき。

  特定の蔵書を借りている利用者を検索したいとき。

  操作と条件の指定は図書館の司書が行う。

- **基本パス [利用者検索]**
- **前件**

  -- 利用者の集合が空でないこと。

  Set(利用者)->notEmpty

- **後件**

  -- 指定した条件を満たす利用者の集合を結果とする。

  result = Set(利用者)->select(利用者の指定条件)

- **代替パス [借りている蔵書による検索]**
- **前件**

  -- 蔵書の集合が空でなく、指定した蔵書の集合も空でないこと。

  Set(蔵書)->notEmpty and

  Set(蔵書)->select(蔵書の指定条件)->notEmpty

- **後件**

  -- 指定した蔵書の集合の借用者の集合を結果とする。

  result = Set(蔵書)->select(蔵書の指定条件)-> 借用者

さて、すべての UseCase とシナリオを紹介したので、次に、この問題の制約条件を見つけてみよう。

すでに「利用者が借りることのできる蔵書は合計 5 冊までである」という制約条件は貸出のシナリオで見つけている。もちろん、このような制約条件は分析者が頭の中で考えていても見つからない。ドメインのエキスパートや専門書から見つけるしかない。ここでは、ユーザーインタビューから、以下の制約条件を見つけたことにしよう。最大貸出数の制約も、以下のように汎用性を持たせて書き換える。

仕様 11 図書館貸出管理の制約条件

- **蔵書は貸出可能かすでに貸し出されているかのどちらかである。**[33]
- **蔵書が貸出可能でかつ存在しないということはない。**[34]
- **利用者への最大貸出数は上限がある。**

実は、UseCase やシナリオを書きながら、以下に述べるドメインオブジェクトやクラス図のたたき台は頭に思い浮かべている。例えば、

削除する蔵書 . 本 . 蔵書 ->size = 1

などという記述をしたが、これは以下のようなクラス図の一部を思い浮かべてこそ記述できる。クラス「本」とクラス「蔵書」の間を、本のインスタンスとその本の蔵書のインスタンスを関係付ける関連が結んでいる。本クラスにある蔵書 ()[35] 操作は、その本の蔵書のインスタンス集合を返す。蔵書クラスにある本 () 操作は、その蔵書の属する本（のインスタンス）を返す。

---

33. もちろん、実際のシステムでは、これ以外に「修理中」といった状態も考えられるが、ここでは問題を簡単にするため制約条件を単純化した。
34. これも、実際の図書館では「盗難」などで起こり得るが、単純化した。
35. まだ、操作のパラメータの詳細は興味がないので、とりあえずパラメータなしの記法 ( ) を書いている。

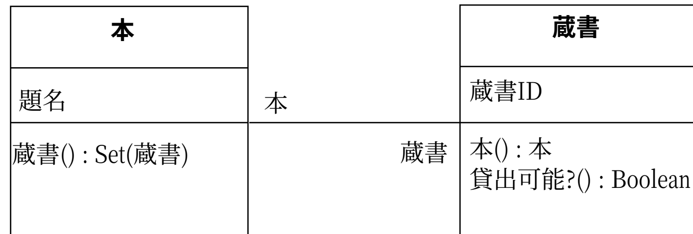

図 45 本と蔵書のクラス図の一部 (叩き台)

4.2.4 **ドメインオブジェクトの発見**

さて、いよいよドメイン分析の中核であるドメインオブジェクトの発見方法を記述する。ドメインオブジェクトの発見は以下のような手順を踏むが、実際にはこの３つの仕事をほぼ同時に何度も繰り返す。

- **オブジェクトの名前・責任・役割を見つける**
- **オブジェクトを分類し、関連・包含関係を見つける**

まず、ドメインオブジェクトの名前・責任・役割を見つけるところから説明しよう。

(1) **オブジェクトの名前・責任・役割を見つける**

ドメインオブジェクトは主に UsaCase とシナリオから探す。ドメインオブジェクトは、対象問題領域 (ドメイン) で主要な役割を果たすオブジェクトである。ドメインオブジェクトは、他と明確に区別でき[36]、システムのライフサイクル内で「長生きする」可能性の高いオブジェクト[37]を選ぶ必要がある。

36. 例えば「ユーザー」と「利用者」というオブジェクトを両方持つと、あるオブジェクトがどちらに所属するのか曖昧になる可能性が強い。こうなると、正しいプログラムが作成できなくなる。
37. オブジェクトを定義するクラス構造は、システム実行時に気軽には変えられないので、「短い寿命」のドメインオブジェクトを作ると、プログラムの修正が頻繁に発生する可能性がある。

図書館の貸出管理システムでは、トップレベルの UseCase からは、「司書」と「利用者」がドメインオブジェクトの候補として見つかる。貸出のシナリオから、「貸出」「蔵書」「図書館」を見つけ、蔵書検索シナリオから「作者」「分野」を見つける。

各ドメインオブジェクト候補の責任と役割を見つけるのが次の仕事である。以下のようになるだろう。

仕様 12 図書館貸出管理のドメインオブジェクトの責任と役割

- **利用者**

  蔵書を借りる主体であり、セキュリティ上の理由で、システムに登録されていなければならない。

  利用者クラスは利用者のインスタンスを生成・管理する役割とする。

- **司書**

  貸出・返却を行う主体である。

  司書クラスは司書のインスタンスを生成・管理する役割とする。

- **図書館**

  システムに関するリソース (人と本）の管理を行う主体であり、運用ルールも持つ。

  図書館クラスは、図書館のインスタンスを生成する役割とする。[38]

- **本**

  題名・著者が同じ本を 1 つと考える場合の、抽象的論理的概念としての「本」。

  題名・著者を答え、「本の実体」オブジェクトを把握している主体。

  本クラスは、本のインスタンスを生成・管理し、検索する役割とする。

- **本実体**

  個々の本の情報を持つ物理的な概念としての「本」。蔵書管理の対象。[39]

---

38. 複数の図書館の貸出管理をするなら、図書館クラスはインスタンスの管理も行わなければならないが、ここでは簡単化のため、図書館のインスタンスを 1 つとする。
39. UseCase とシナリオでは「蔵書」と呼んでいたが、「本の実体」を図書館が所有しているとき「蔵書」と呼ぶので、「蔵書」はクラス名というよりロール名であるのが適当だと判断した。

本実体クラスは、本実体インスタンスの生成・管理を行う。

- **貸出**

  貸出行為を処理・管理する主体。

  貸出クラスは、貸出インスタンスの生成・管理を行う。

- **著者**

  本の著者の情報を持つ。

  著者クラスは、著者インスタンスの生成・管理・検索を行う。

- **分野**

  本の分野の情報を持つ。

  分野クラスは、分野インスタンスの生成・管理・検索を行う。

実際のプロジェクトでは、当初ドメインオブジェクトと思ったものが、責任と役割を考えていくうちに大した役割を果たさないことが分かり、ドメインオブジェクトでなく、別のドメインオブジェクトの属性になってしまうということがよくある[40]。今回の例題は小さなものなので、幸いこのようなことは起こらなかった。

(2) オブジェクトを分類し、関連・包含関係を見つける

さて、いよいよドメインオブジェクトの関係をクラス図として書いていくことにより、ドメインオブジェクトを分類し、ドメインオブジェクト同士の関連・包含関係を見つけていくことになる。

モデリングに慣れてくると、直接クラス図を書き始めることができるが、普通はシナリオごとに順序図を作っていって、オブジェクト間でどのようなメッセージをどのような順番でやり取りすればよいかを決めていく。

以下に、貸出の順序図を示す。

順序図はオブジェクト間にどのような順番でメッセージが流れるかを記述するための図である。四角の箱の中にクラス名が記述されているが、下線を付けて、その

40. 比喩としては「小渕首相ドメインオブジェクトは竹下元首相ドメインオブジェクトの属性になってしまう」という感じだろうか。もっとも、ある個人やその役割をドメインオブジェクトにするのは、普通は、よいことではない。

クラスのあるインスタンスであることを示す[41]。メッセージは横方向の矢印で示し、メッセージ名 (パラメータ)：返値という形式をとる。時間は上から下に流れ、オブジェクトの下の縦棒（ 生命線）がオブジェクトを表す。破線の矢印は、制御の戻りを示す[42]。

例えば、一番下の「貸出」から「司書」へ向かう破線の矢印は、「生成（ 利用者 , 本実体) : 貸出」メッセージに対応していて、この時点で制御が司書オブジェクトへ戻り、「貸出」オブジェクトが生成されて司書オブジェクトがそれへの参照を手に入れたことを示す。中ほどにある「利用者」から「貸出」へ向かう破線の矢印は、その 2 本上の「貸出」から「利用者」へ向かう「貸出可？ () : Boolean」メッセージの制御の戻りを表す。この貸出可？のメッセージを扱っている最中に、「利用者」オブジェクトが「図書館」オブジェクトへ「最大貸出数 () : 冊数」メッセージを送り、最大貸出数を得ていることが読みとれる。

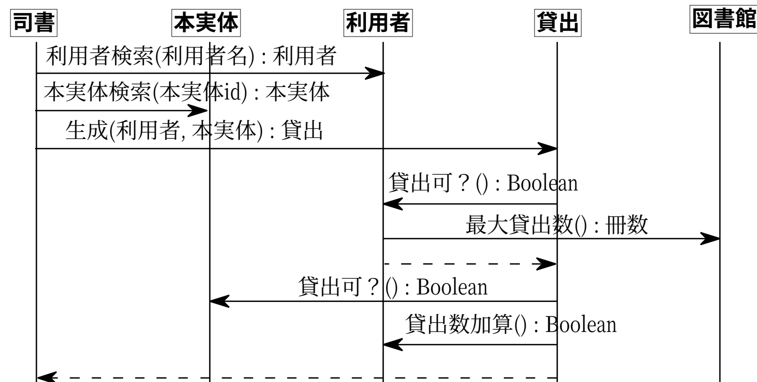

図 46 貸出の順序図

この順序図で、貸出可能かどうかを「貸出」オブジェクトが「利用者」オブジェクトと「本実体」オブジェクトに問い合わせているのは、貸出可能かどうかを判定

41. 使用ツールの関係で、下の例では下線が付いていない。
42. 制御の戻りを強調する場合に記述する。

するのは「貸出」オブジェクトの責任ではないという考えからである。本実体が貸出可能かどうかは、それ自身が判断すべきだし、利用者が借りる資格があるかどうかは、やはりそれ自身が判断すべきだからだ[43]。

また、「利用者オブジェクト」は最大貸出数を自分で持たずに、「図書館」オブジェクトにメッセージを投げて聞いている。これも、図書館のルールは利用者でなく「図書館」オブジェクトが持つべきだとの考えによる[44]。

次に、返却の順序図を示す。

ここでは、「貸出」オブジェクトが自分自身に「本実体借用者？ () : 利用者」メッセージを投げ、得られた利用者オブジェクトへ「貸出数減算 () : Boolean」メッセージを送っている。

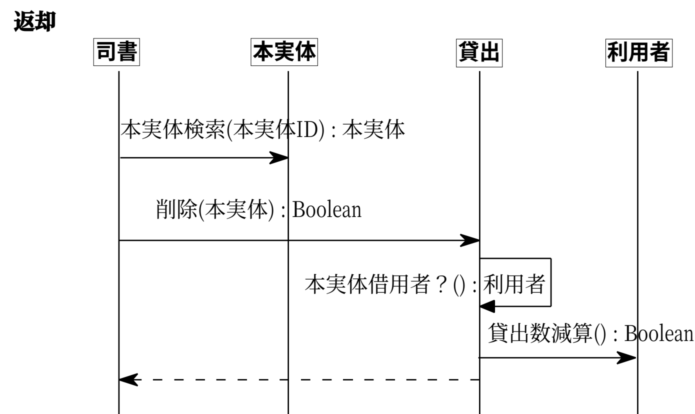

図 47 返却の順序図

次は、蔵書検索の順序図である。著者をキーとしたものと分野をキーとしたものの 2 種類ある。

43. もちろん、利用者オブジェクトは「実際の利用者」ではないのでセキュリティ上の問題は発生せず、こういうことが可能になる。ここで考えている利用者オブジェクトは、あくまでもシステム内の抽象データなのだ。
44. もちろん、利用者オブジェクトが最大貸出数を持つモデルも間違いとは言えない。モデル内でどこか 1 カ所に情報が隠蔽されていれば、この場合、大きな問題が発生することはない。

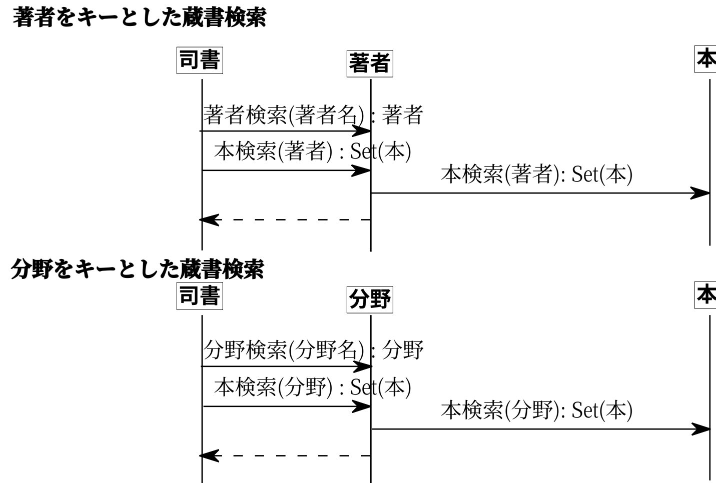

図 48 蔵書検索の順序図

いずれも、まず著者あるいは分野を検索するメッセージを送り、次に得られた著者または分野をパラメータとして、本検索を行っている。

このように、シナリオに対応して順序図を作っていき、オブジェクトの役割と、オブジェクト間のメッセージのやり取りを考える。実プロジェクトでは、クラス図と順序図と「クラスの責任と役割」は、何回もフィードバックしながら徐々によいものにしていく。

ここでは、そのようなフィードバックが何回かあったものとして、クラス図のたたき台を示そう。

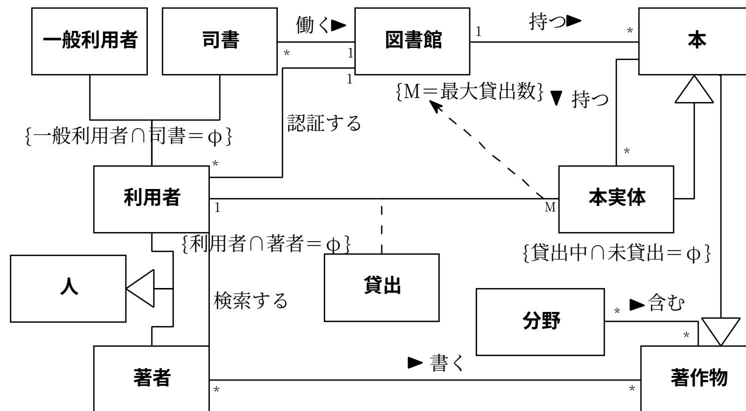

図 49 図書館貸出管理のクラス図 (たたき台)

上のクラス図では、クラスの分類もある程度した。その思考過程をこれから示そう。

まず、貸出管理の対象となる本実体クラスであるが、本の性質も持っているので本のサブクラスとする。また、本は CD やビデオなどと同じ著作物であるので、抽象クラス[45]「著作物」を作り、本はそのサブクラスとした。こうしておけば、図書館で CD やビデオを貸し出すようにしたときもクラス図の修正はわずかで済む。

また、図書館は本を持ち、本は本実体を持つという関連を設定した。この２つの関連で「蔵書」という概念を表す。

例えば、図書館に本が 10000 タイトルあり、ある本の本実体が 10 冊有るとする。図書館と本の間の関連は前者のリンク[46] を持ち、本と本実体の関連は後者のリンクを持つ。本クラスは主に「検索機能」で使用し、本実体クラスは主に「貸出機能」で使用するという役割分担をする。本実体は貸出中か未貸出[47] のどちらかである。

45. インスタンスのない、再利用可能な操作を持つクラス。サブクラスでいくつかの操作を実装すれば、他の操作 (機能) が使えるようになる。
46. インスタンス同士の結合関係をリンクという。リンクの集合体が関連である。
47. 未貸出の時は、常に貸出可能であることは前に述べた。

貸出の情報は、利用者クラスと本実体クラスの間の関連クラス「貸出」が持つ。ただ「貸出」を記録するだけなら、利用者クラスと本実体クラスの間に「貸出」関係が有ればよいのだが、「貸出」の記録を参照したり「貸出」を取り消したりといった機能が予想されるので、貸出クラスを利用者クラスと本実体クラスの間の関連クラスとしておいた方が、再利用性も保守性も向上する。

利用者は本実体 M 冊を借りることができ、その最大貸出数の情報は図書館クラスが持っている[48]。利用者クラスは、一般利用者クラスと司書クラスをサブクラスに持つ抽象クラスで、一般利用者と司書に共通する機能を持つ。一般利用者クラスと司書クラスのインスタンスに重複はない[49]。

図書館クラスは利用者クラスと「認証する」関連[50] を持ち、司書クラスは図書館クラスと「働く」関連[51] を持つ。利用者クラスは抽象クラス「人」のサブクラスであり、著者クラスも人クラスのサブクラスである[52]。単純化のために、著者と利用者の和集合は空であるとした。著者が偶然図書館の利用者であったとしたら、データを 2 重に登録することになるが、運用の手間が大して増えるわけではない。

著者クラスと分野クラスは、それぞれ著作物クラスと関連で結ばれている。これによって、著者と分野による本および本実体の検索を行える。

4.2.5 オブジェクトの状態変化の記述

さて、ドメインオブジェクトの候補は大分明確になってきて、クラス図のたたき台もできた。このまま、クラス図の各クラスに必要な操作(機能) を追加していってもよいのだが、その前に各クラスのオブジェクトの振る舞いを検討してみよう。

今までは、ドメインの静的な構造を中心に考えていたので、このあたりでドメインオブジェクトの振る舞いを考えることにより、「思考」の偏りを是正しておこうというわけである。こうした方が、ドメインモデルの誤りを早く発見できることが多い。

---

48. 図書館のルールは図書館クラスが持っているべきだろう。
49. すなわち、一般利用者と司書の和集合は空である。
50. 利用者一覧 () のメッセージを送るのに使うことが予想される。
51. 職員一覧 () のメッセージを送るのに使うことが予想される。
52. 著者クラスは具象クラスであり、インスタンスを持つ。

ドメインオブジェクトの振る舞いは、各オブジェクトの状態変化を記述することで分析できる。このための道具が「状態遷移図」である。

まず、本実体の状態遷移図を見てみよう。これは本実体は貸出中か未貸出のどちらかであるという制約条件がそのまま反映されている。黒丸から出た矢印記号は初期遷移を表し、最初にどの状態に移るかを示す[53]。今の場合、未貸出状態になる。状態は四角の記号で表し、上のフィールドに状態名を記す。未貸出状態から 2 重丸記号に向かう矢印は終了遷移を表し、現時点での状態遷移図を抜けることを示す[54]。

未貸出状態で貸出イベントがくると、状態は貸出中に遷移する。貸出中状態で返却イベントがくると、状態は未貸出に遷移する。

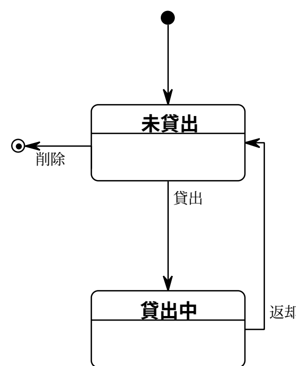

図 50 本実体の状態遷移図

次に、利用者オブジェクトの状態遷移図を見てみよう。ここで、「借用中」状態の中にある H を○で囲った記号は、履歴状態指標（history state indicator）を意味し、状態が「前の状態」を覚えていることを示す。今の場合は、借用数を覚えていることを表している。

状態遷移を表す矢印に付いている文字列は、以下のような形式である。

イベント名 引数リスト[55] '[' ガード条件 ']' '/' 動作式

53. 状態遷移図がオブジェクトに対応しているときは、そのオブジェクトが生成されたときの遷移を表していることになる。
54. 状態遷移図がオブジェクトに対応しているときは、そのオブジェクトがなくなるときの遷移を表していることになる。

イベント名は状態を遷移させるイベントを示し、ガード条件は「ある条件が満たされたとき」遷移が起こることを示す。動作式は、遷移の結果起こる 1 つまたは複数の動作を表す。動作は、最終的には操作すなわちプログラムとなる。

結局、この状態遷移図では、「借用中」状態で借用しようとするときに最大貸出数≦借用数 @pre であれば「貸出限度超過」の処理を行い、最大貸出数＞借用数 @pre であれば「借用数加算」を行う。返却するときに借用数 @pre ＞ 1 であれば「借用数減算」を行い、借用数 @pre = 0 であれば「未借用」状態へ戻る。

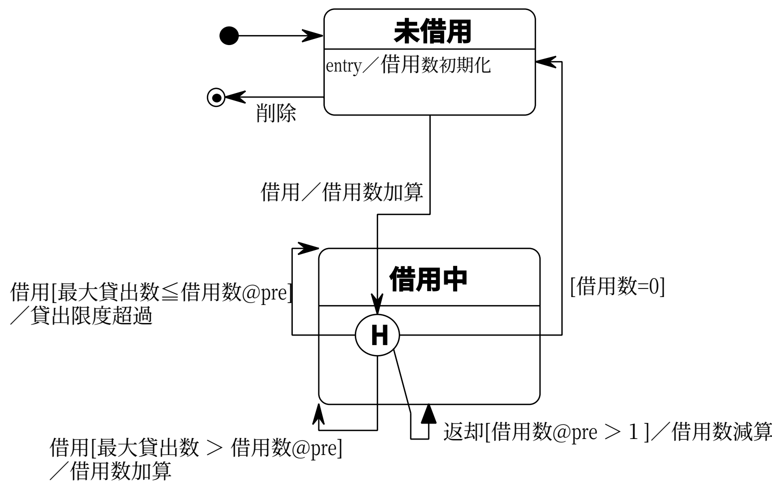

図 51 利用者オブジェクトの状態遷移図

図書館など他のオブジェクトは、あまり興味ある「振る舞い」をしないので、状態遷移図を書く必要はない。

4.2.6 要求仕様記述

この工程の主な仕事は各クラスごとの操作名の洗い出しと、操作の仕様として前件・後件を記述することである。主要なクラスとその主な操作を記述していく。

---

55. 引数リストは、この例ではまだ出てこない。

まず、人クラスとそのサブクラスから見ていく。

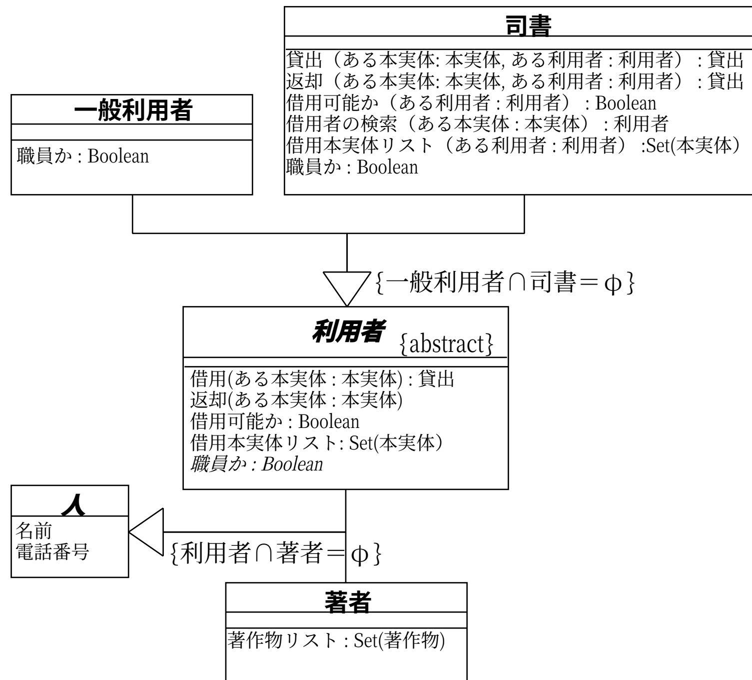

図 52 人関係のクラス図

ここで人クラスと利用者クラスは抽象クラス[56]であり、利用者クラスの「職員か」操作は抽象操作[57]である。

抽象クラス「利用者」は、図書館の利用者を表す。仕様は以下の通りである。

56. インスタンスのないクラスで、サブクラスで定義すべき抽象操作がある。
57. サブクラスで定義される操作で、クラス図中ではイタリック体で示される。

仕様 13 利用者クラスの仕様

(1) **借用 (ある本実体 : 本実体) : 貸出**

私 (利用者)[58] がある本実体 (1 冊) を借用し、ある貸出のインスタンスを返す。

- **前件１ [貸出可能]**

-- 今までに利用者が借りた本実体の冊数は最大貸出数未満で、かつ

-- 今回借用する本実体は貸出可能でなければならない。

self. 借用本[59] ->size ＜図書館 . 最大貸出数 and

ある本実体 . 貸出可能か = true

- **後件１ [貸出可能]**

-- 借用前の本実体の集合からある本実体を削除したものが、

-- 貸出後の本実体の集合である。

-- 貸出の集合は 1 要素分増え、増えた貸出の利用者は私であり、

-- その貸出の本実体は「ある本実体」である。

Set(本実体) = Set(本実体)@pre － Set{ ある本実体 } and

result[60] ->size = self. 貸出 @pre->size + 1 and

result. 利用者 = self and

result. 本実体 = ある本実体

- **前件２ [貸出不能]**

-- 今までに利用者が借りた本実体の冊数が最大貸出数以上か、

-- または、今回貸し出す本実体が貸出不能である。

self. 借用本 ->size >= 図書館 . 最大貸出数 or

ある本実体 . 貸出可能か = false

---

58. ここで言う「私」は、本を書いている私ではなく、利用者クラスのあるインスタンス・オブジェクト（OCL では self で表す）を示している。
59.「利用者」クラスと「本実体」クラスの「貸出」関係で、「本実体」のロール名であるとする。「利用者」のロール名は「借用者」とする。
60. 今、result は「ある貸出」を表している。

- **後件２ [貸出不能]**

  -- 貸し出す前と後の本実体の集合は等しく、

  -- 貸出の集合は変化せず、返値は空である。

  Set(本実体) = Set(本実体)@pre and

  Set(貸出) = Set(貸出)@pre and result ->isEmpty

(2) **返却 (ある本実体 : 本実体)**

私が借用しているある本実体 (1 冊) を返す。

- **前件**

  -- 返却された本実体が、利用者が借りた本実体に含まれている。

  self. 借用本 ->notEmpty and Set{ ある本実体 }->notEmpty and

  self. 借用本 ->includes(ある本実体)

- **後件**

  -- シナリオが動く前の本実体の集合と返却された本実体の集合の和集合が、

  -- 新たな本実体の集合となっている。

  Set(本実体) = Set(本実体)@pre union Set{ ある本実体 } and

  self. 貸出 ->size = self. 貸出 @pre->size － 1 and

  ある本実体 . 貸出可能か = true

(3) **借用可能か** : Boolean

図書館の規則によって、私が借用可能かどうかを返す。

- **前件 [貸出あり]**

  self. 借用本 ->notEmpty

- **後件 [貸出あり]**

  -- 私の借用数が、図書館の最大貸出数以下であれば真を返す。

  result = self. 図書館 . 最大貸出数 >= self. 借用本 ->size

- **前件 [貸出なし]**

  self. 借用本 ->isEmpty

- **後件 [貸出なし]**

  result = true

(4) **借用本実体リスト** : Set(**本実体）**

私が借用している本実体の集合を返す。

- **後件**

  -- 私の借用本の集合が結果となる。

  result = self. 借用本

司書クラスは、本の貸出を行うことができる司書の資格を持つ図書館職員を表す。

仕様 14 司書クラスの仕様

(1) **貸出（ある本実体** : **本実体** , **ある利用者** : **利用者）** : **貸出**

ある利用者にある本実体を貸し出す。

- **前件**

  -- ある利用者が存在し、図書館の認証利用者であること。

  Set{ ある利用者} ->notEmpty and self. 図書館 . 利用者 ->includes(ある利用者）

- **後件**

  -- ある利用者の借用 (ある本実体) 結果を返す。

  result = ある利用者 . 借用 (ある本実体)

(2) **返却（ある本実体** : **本実体** , **ある利用者** : **利用者）** : **貸出**

ある利用者に貸していたある本実体を返却する。

- **前件**

  -- ある利用者とある本実体が存在すること。

  -- ある利用者が図書館の認証利用者であること。

  Set{ ある利用者 }->notEmpty and Set{ ある本実体 }->notEmpty and

  self. 図書館 . 利用者 ->includes(ある利用者）

- **後件**

  -- ある利用者の返却 (ある本実体) 結果を返す。

  result = ある利用者 . 返却 (ある本実体)

(3) **借用可能か（ある利用者** : **利用者）**: Boolean

図書館の規則によって、ある利用者が借用可能かどうかを返す。

- **前件**

  -- ある利用者が存在し、図書館の認証利用者であること。

  Set{ ある利用者 }->notEmpty　and　self. 図書館 . 利用者 ->includes(ある利用者）

- **後件**

  -- ある利用者が借用可能か返す。

  result = ある利用者 . 借用可能か

(4) **借用者の検索（ある本実体** : **本実体）**: **利用者**

ある本実体の借用者を返す。

- **前件**

  -- ある本実体が空でないこと。

  Set{ ある本実体 }->notEmpty

- **後件**

  result = ある本実体 . 貸出 . 利用者

(5) **借用本実体リスト（ある利用者** : **利用者）**:Set(**本実体）**

ある利用者の借用している本実体の集合を返す。

- **前件**

  -- ある利用者が存在し、図書館の認証利用者であること。

  Set{ ある利用者 }->notEmpty　and　self. 図書館 . 利用者 ->includes(ある利用者）

- **後件**

  -- ある利用者の借りている本実体の集合が結果となる。

result = ある利用者 . 借用本

著作物クラスは、本や CD やビデオなどを表す抽象クラスである。ここでは、そのサブクラスである本クラスと本実体クラスと共に示す。

著作物クラスは、クラス操作として題名検索・著者検索・分野検索を持つ。いずれも、文字列をパラメータとして受け取り、題名あるいは著者名あるいは分野名が一致する著作物の集合を返す。

本クラスは、題名と著者が同じ本を 1 つと考える場合の、抽象概念としての「本」を表す。

本のインスタンスは、個々の実際の本を表す本実体クラスのインスタンスを「持っている」。例えば「Z」という題名の本が図書館に 5 冊あるとすると、本クラスの「Z」のインスタンスが 1 つあり、そのインスタンスが本実体クラスの 5 個のインスタンス（5 冊の「Z」）を「持っている」[61]。

61. この本クラスと本実体クラスを結ぶ「持っている」関連を保持している本クラスの属性「本実体」は、UML のクラス図では省略する約束になっている、また、クラスの属性に値を設定したり取り出したりする「操作」もクラス図では省略することになっている。これらの操作の仕様は自明なので、仕様記述も省略する。

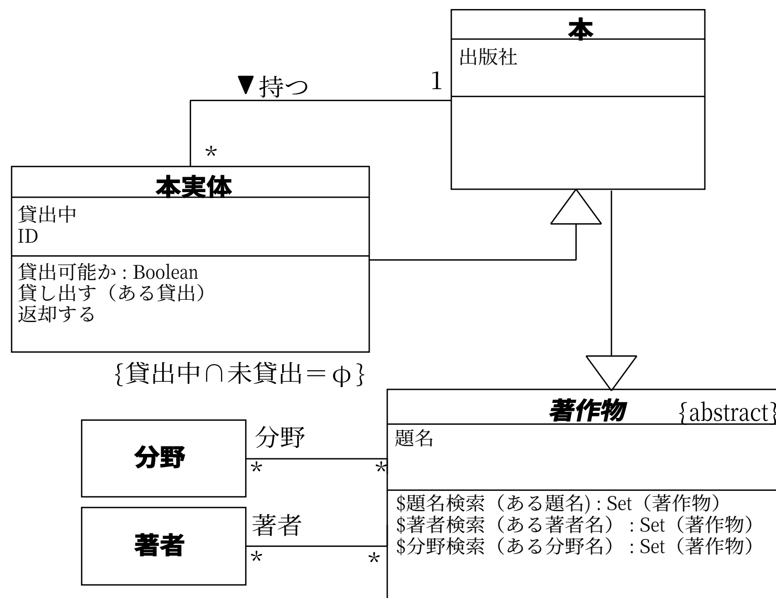

図 53 著作物関係のクラス図

仕様 15 **著作物クラスの仕様**

(1) **題名検索（ある題名**) : Set **（著作物）**[62]

題名が「ある題名[63]」と一致する著作物の集合を返す。

- **前件**

  -- 私のインスタンスが空でないこと。

  self.allInstances->notEmpty

62. 下線の引いてある操作あるいは $ が名前の先頭に付いている操作は、クラス操作であることを示す。
63. ある題名の型はまだ決めていない。多分文字列になるのだが、そのような決定は後の工程でやるべきである。

- **後件**

 -- 求める著作物の題名は、ある題名と等しい。

 result->forAll[64]（literary : 著作物 | literary. 題名 = ある題名）= true

(2) **著者検索（ある著者名）**: Set（**著作物**）

 著者名が「ある著者名」と一致する著作物の集合を返す。

- **前件**

 -- 私のインスタンスが空でないこと。

 self.allInstances->notEmpty

- **後件**

 -- 求める著作物の著者名は、ある著者名と等しい。

 result->forAll（author : 著者 | author. 名前 = ある著者名）= true

(3) **分野検索（ある分野名）**: Set（**著作物**）

 分野名が「ある分野名」と一致する著作物の集合を返す。

- **前件**

 -- 私のインスタンスが空でないこと。

 self.allInstances->notEmpty

- **後件**

 -- 求める著作物の分野名は、ある分野名と等しい。

 result->forAll（field : 分野 | field. 分野名 = ある分野名）= true

本実体クラスは、図書館で管理している物理的な概念としての個々の本を表す。従って、同じ題名の本実体が複数存在する。

64.forAll は物の集まり (Collection 型) の既定義操作で、引数で指定された条件がすべての要素について成り立つと真を返す。

仕様 16 本実体クラスの仕様

(1) **貸出可能か** : Boolean

私が貸出可能かどうかを返す。

- **後件**

result = not (self. 貸出中)

(2) **貸し出す（ある貸出 : 貸出）**

私とある貸出をリンクさせ、貸出中とする。

- **前件**

-- 私は貸出可能で、私とリンクされた貸出オブジェクトはない。

self. 貸出中 = false and Set{ self. 貸出 }->isEmpty

- **後件**

-- 私は貸出中で、私とリンクされた貸出オブジェクトが「ある貸出」になる。

self. 貸出中 = true and self. 貸出 = ある貸出

(3) **返却する**

私と貸出オブジェクトのリンクを解消し、私を貸出可能とする。

- **前件**

-- 貸出中である。

self. 貸出中 = true and Set{ self. 貸出 }->notEmpty

- **後件**

-- 私は貸出可能で、私とリンクされた貸出オブジェクトはない。

self. 貸出中 = false and Set{ self. 貸出 }->isEmpty

次に図書館クラスと貸出クラスを示す。

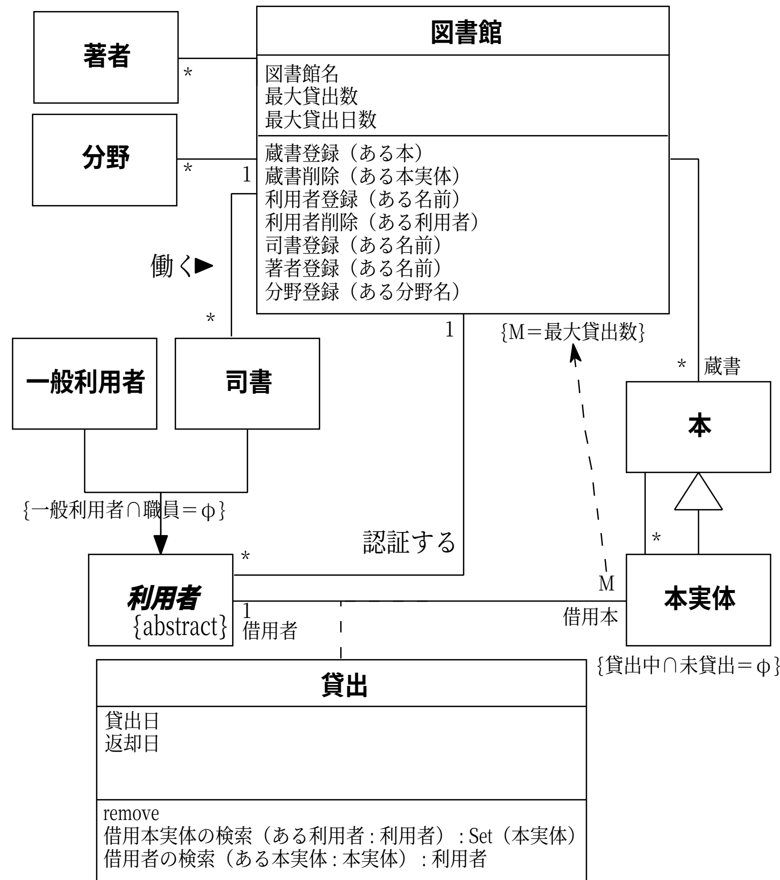

図 54 図書館クラスと貸出クラス

貸出クラスは貸出行為を処理・管理する主体である。

仕様 17 **貸出クラスの仕様**

(1) **借用者の検索（ある本実体** : **本実体）**: **利用者**

ある本実体の借用者を返す。

- **前件**

  ある本実体 . 貸出中 = true

- **後件**

  result = self. 利用者

(2) 借用本実体の検索（ある利用者 : 利用者）: Set（本実体）

ある利用者の借用している本実体の集合を返す。

- **前件**

  -- ある利用者が存在し、図書館の認証利用者であること。

  Set{ ある利用者 }->notEmpty and

  self. 利用者 . 図書館 . 利用者 ->includes(ある利用者）

- **後件**

  result = self. 利用者 . 借用本

(3) remove[65]

私を貸出クラスのインスタンスから削除する。関連する本実体と利用者のオブジェクトの情報の整合性を維持する。

- **前件**

  -- 私とリンクされた本実体とリンクされた貸出オブジェクトは私であり、

  -- 私とリンクされた利用者の貸出オブジェクトの集合には私が含まれている。

  self. 本実体 . 貸出 = self and self. 利用者 . 貸出 ->includes(self)

- **後件**

  -- 貸出の集合から私は削除され、

  -- 私とリンクされた本実体とリンクされた貸出オブジェクトは空になり、

  -- 私とリンクされた利用者の貸出オブジェクトの集合に私は含まれない。

  Set（貸出）= Set（貸出）@pre − Set{self} and self. 本実体 . 貸出 ->isEmpty and

---

65. 多くのクラスライブラリーで remove は「削除する」意味で使われている。「削除する」を操作名とすると、同じ意味の操作に２つの名前を付けてしまうことになる。そこで、メッセージの多相性を保つために、あえて英字名を使った。

not (self. 利用者 . 貸出 ->includes(self))

図書館クラスは、システムに関するリソース (人と本）の管理を行う主体であり、運用ルールも持つ。

仕様 18 図書館クラスの仕様

(1) **司書登録（ある名前）**
ある名前の司書を登録する。

- **後件**

-- 司書の集合の要素が 1 つ増え、増えた司書の名前は「ある名前」である。

Set(司書)->size = Set(司書)@pre->size + 1 and

(Set(司書) ー Set(司書)@pre)->exists[66](librarian : 司書 | ある名前 = librarian. 名前)

(2) **利用者登録（ある名前）**
ある名前の利用者を登録する。

- **後件**

-- 利用者の集合の要素が 1 つ増え、増えた利用者の名前は「ある名前」である。

Set(利用者)->size = Set(利用者)@pre->size + 1 and

(Set(利用者) ー Set(利用者)@pre)->forAll(user | ある名前 = user. 名前)

(3) **利用者削除（ある利用者）**

- **前件**

-- ある利用者が存在し、図書館の認証利用者であり、本を借りていないこと。

Set{ ある利用者 }->notEmpty and self. 利用者 ->includes(ある利用者） and

self. 利用者 . 借用本 ->isEmpty

66.exists も既定義操作の 1 つで、() 内の条件を満たす要素が 1 つでもあれば真を返す。

- **後件**

  -- 削除する前の利用者の集合から削除する利用者を除いた集合が、

  -- 削除後の利用者の集合と等しい。

  Set(利用者) = Set(利用者)@pre − Set{ ある利用者 }

(4) 著者登録（ある名前）

ある名前の著者を登録する。

- **後件**

  -- 著者の集合の要素が 1 つ増え、増えた著者の名前は「ある名前」である。

  Set(著者)->size = Set(著者)@pre->size + 1 and

  (Set(著者) − Set(著者)@pre)->forAll(author | ある名前 = author. 名前)

(5) 分野登録（ある分野名）

ある名前の分野を登録する。

- **後件**

  -- 分野の集合の要素が 1 つ増え、増えた分野の名前は「ある分野名」である。

  Set(分野)->size = Set(分野)@pre->size + 1 and

  (Set(分野) − Set(分野)@pre)->forAll(field | ある名前 = field. 分野名)

(6) 蔵書登録（ある本）

ある本を登録する。

- **後件**

  -- 本の集合の要素が 1 つ増え、増えた本は「ある本」である。

  Set(本) = Set(本)@pre->including[67] (ある本)

(7) 蔵書削除（ある本実体）

ある本実体を削除する。ある本実体がその本の最後の本実体であるとき、その本も削除する。

67.() 内の要素を追加した集合を返す Set クラスの既定義操作。

- **前件 [本実体削除]**

  -- 本実体の集合が空でなく、削除する本実体と同じ本の他の本実体がある。

  Set(本実体)->notEmpty and ある本実体 . 本 . 本実体 ->size > 2

- **後件 [本実体削除]**

  -- 削除する前の本実体の集合とある本実体の集合の差が

  -- 削除後の本実体の集合と等しい。

  Set(本実体) = Set(本実体)@pre － Set{ ある本実体 }

- **前件 [本削除]**

  -- 本実体の集合が空でなく、ある本実体と同じ本の他の本実体は他にない。

  Set(本実体)->notEmpty and ある本実体 . 本 . 本実体 ->size = 1

- **後件 [本削除]**

  -- 削除前の本実体集合とある本実体の差が、削除後の本実体の集合と等しく、

  -- 削除前の本の集合からある本実体の本を除いた集合が、

  -- 削除後の本の集合と等しい。

  Set(本実体) = Set(本実体)@pre － Set{ ある本実体 } and

  Set(本) = Set(本) @pre － Set{ ある本実体 . 本 }

4.2.7 ドメインモデルの検証

ここまでで、一応ドメインモデルは作成できたので、そのモデルを検証する作業に移る。かなりの時間を費やしてモデルを作ってきたのは、この検証工程でより多くの欠陥を発見しようとしているからである。人間が一番強力な欠陥発見ツールであることが、ソフトウェア工学の研究の結果分かっている。

検証は以下の 4 種類の作業からなる。

- **有効性確認（Validation）**

  ユーザーの要求と合っているかを確認する作業である。エンドユーザーを交えたレビューやプロトタイプの作成で確認することが多い。

- **検証（Verification）**

  仕様を満たしているかを確認する作業である。プロトタイプを作成・配布して多くの人に検証してもらうのが一般的である。

- **正当性チェック（Justification）**

  仕様の正当性を証明する作業である。正当性チェックは検証の一部であるが、より厳密な証明が要求される。そのため、形式仕様記述言語で仕様を記述し、証明規則を適用して使用を証明する。通常、生命に関わるシステム（Safety Critical System）の核となる部分に適用する[68]。

  オブジェクト指向技術では、OCL のような形式仕様記述言語はあるが、その証明規則はまだ整備されていない。そのため、仕様の構文チェックを除くと人間の目による机上の正当性チェックしか行えない。それでも、ソフトウェア・システムの致命的な欠陥を発見するのに有効なことが分かっている。

- **欠陥の管理**

  欠陥を管理し、欠陥の修正によって新たな欠陥が発生しないことを保証する作業である。大きなシステムの開発では、元々の欠陥の数より、それを修正することで発生する欠陥の方が多くなる傾向がある。

  欠陥の発生から、それをいつ・誰が修正したかまでの履歴を作成し、保守する必要がある。この作業のためには、仕様やプログラムの版管理システムが有効である。

さて、本書では、上記の作業の内、机上で行えるレビューによる有効性確認と検証のやり方を説明しよう。

机上チェックのテストケースは、UseCase とシナリオから作る。テストケースは、システムの検証に必要なオブジェクトを順番に生成するように構成していく。こうしていけば、あるテストケースをテストするとき「必要なデータがない」という事態を避けられる。

---

68. 実用的な大きさのシステムで、システム全体の正当性を証明することは事実上不可能であることが分かっているため。

例えば「利用者登録」のシナリオの次に「蔵書登録」シナリオを実行してから、「貸出」と「返却」のシナリオを実行すれば、このシステムの主要機能が検証できる。

ここでは、「利用者登録」と「蔵書登録」の検証は終わったものとして、「貸出」のシナリオを検証してみよう。

まず検証するのは、司書クラスの操作「貸出」である。この操作のパラメータは「ある本実体」と「ある利用者」であるから、それぞれ「蔵書登録」と「利用者登録」で登録したはずの「Ｚ という本の実体」と「佐原という利用者」を具体的なデータとして貸出操作を机上実行してみよう。

前件は

Set{ 佐原という利用者｝->notEmpty and

self. 図書館 . 利用者 ->includes(佐原という利用者）

となるので、真である。

後件は

result = 佐原という利用者 -> 借用 (Ｚ という本の実体)

となり、利用者クラスの操作「借用」を呼び出している。この操作の前件[69] は、

佐原という利用者 . 借用本 ->size ＜ 図書館 . 最大貸出数 and

Ｚ という本の実体 . 貸出可能か = true

となり、

1 < 5 and true

に帰着するので、真となる。従って、後件は

Set(本実体) = Set(本実体)@pre － Set{ Ｚ という本の実体 } and

佐原への貸出 ->size = 佐原という利用者 . 貸出 @pre->size + 1 and

佐原への貸出 . 利用者 = 佐原という利用者 and

佐原への貸出 . 本実体 = Ｚ という本の実体

となり、これも真である。

一見、「貸出」シナリオはうまくいっているように見える。しかし、利用者オブジェクトの状態遷移図を検証してみると、忘れ物があったことに気付く。「貸出」により、利用者オブジェクトの状態遷移図では、「未借用」状態から「借用中」状態へ

---

69. まだ一冊も借りていないはずなので、貸出可能な場合の前件 1 を使う。

遷移するが、この時「借用数加算」動作を行うことになっている。この操作をどこにも定義していなかった。定義すべき場所は利用者クラスであろう。

仕様 19 利用者クラスの借用数加算操作の仕様[70]

- **前件**

  self. 借用可能か = true

- **後件**

  self. 借用数 = self. 借用数 @pre + 1

同様にして検証を進めていくと、利用者クラスの返却操作で貸出のインスタンスを返していないため、司書クラスの返却操作で貸出のインスタンスを返せないことが検出できる。

さらに、図書館の蔵書登録によって本は登録できるが、その本の本実体は登録できないことに気が付く。

以上のように、操作仕様と状態遷移図を対象に、具体的なテストデータを想定しながら、クラスの責任や操作仕様の実行可能性を考慮しながら、机上検証を進めていく。このような作業は面倒で工数の掛かる仕事であるが、同じ作業をプログラミング工程やテスト工程で行うと 10 倍から 100 倍のコストが掛かることがソフトウェア工学の研究の結果分かっている。検証作業を手抜きすれば、後の工程でさらに面倒で工数の掛かる仕事が待っているのである。

なお、レビューの進め方の詳細は、本書の対象範囲を越えるので、参考文献[71] を参照して欲しい。

---

70. 借用数減算操作の仕様は省略する。
71. D.P.　フリードマン、G.M.　ワインバーグ共著、岡田正志監訳、「ソフトウェア技術レビューハンドブック」、産学社、1987 年

4.3 システム分析

システム分析は、ドメイン分析の結果であるドメインモデルを入力として、これから作るシステムが何をやるべきかを記述した分析モデルを作成する。ドメイン分析では「このドメインはどうなっているか？」「本来どうあるべきか」という視点から分析していったが、システム分析では「ユーザーは何を望んでいるか？」「使える分析パターンあるいはデザインパターンはないか？」という視点でより精密な分析モデルを作っていく。システム分析の出力である分析モデルには「システムが何をやるか？」がすべて記述されていなければならない。システム分析で行う作業は以下の通りである。

- **クラス図の作成**
- **操作仕様の記述**
- **状態遷移図の作成**
- **その他の要求仕様記述**
- **分析モデルの検証**

以下に各作業の内容を述べる。

4.3.1 クラス図の作成

ドメイン分析で作成したクラス図を、ユーザーの要求に合わせてさらに精密化していく。また、使える分析パターンあるいはデザインパターンはないか、という視点でクラス図を改良していく。さらに、システムの変更が多発するであろう勘所（Hotspot）と変更のほとんど考えられない部分を認識し、勘所を分離して、そこを修正してもモデルの他の箇所に影響が及ばないようにする。

今回の例題は小さなものなので、ユーザーの要求に合わせて精密化していくところはあまりない。従って、分析パターンとデザインパターンで適用できるものがないかという検討と勘所の発見が主になる。

分析パターンで適用できるものを探していくと、下図のような当事者パターンと責任パターンが使えそうであることが分かる。ここで、ロール名の「委任者」は「責

任」を依頼あるいは委任するオブジェクトを示す。「責任者」は、「責任」を果たすあるいは実行するオブジェクトを示す。

図書館のスーパークラスとして組織を考え、利用者クラスと図書館クラスの関連（認証利用者）および司書クラスと図書館クラスの関連（働く）を、それぞれ責任タイプクラスのインスタンスが「認証利用者」と「雇用」として、当事者クラスと責任クラスの関連で表せばよさそうである。

責任 (Accountability) パターンを適用すると、「認証利用者」や「雇用」の時期も表すことができて、元のモデルよりも有効期間の確認などがやりやすい。

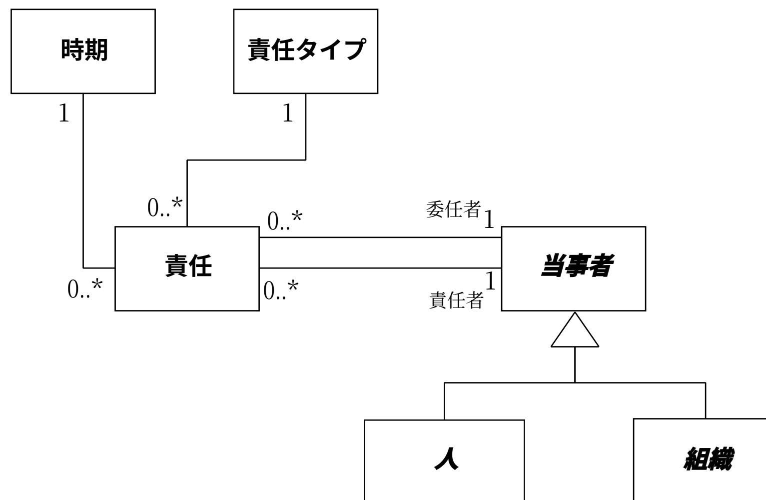

図 55 責任 (Accountability) パターンのクラス図

責任タイプが「認証利用者」である場合のインスタンス・オブジェクト同士の関係の例は以下の図ようになる。委任者は図書館クラスのインスタンスである「国会図書館」でそこを利用する責任者が「佐原伸」である。責任者と言っても、この場合「利用する」責任を果たすという意味であり、一般の意味の「責任者」とは意味が異なる。

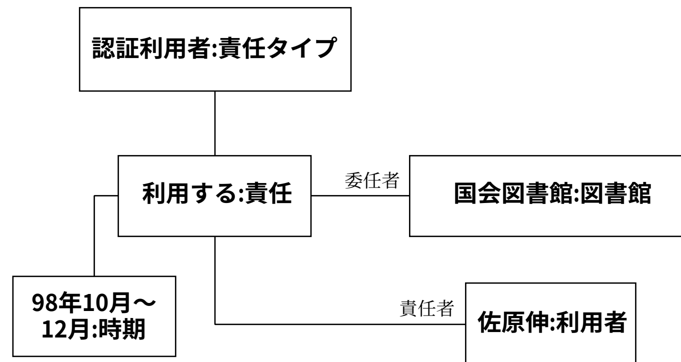

図 56 認証利用者の場合の責任 (Accountability) パターン例

責任タイプが「雇用」である場合のインスタンス・オブジェクト同士の関係の例は以下の図のようになる。

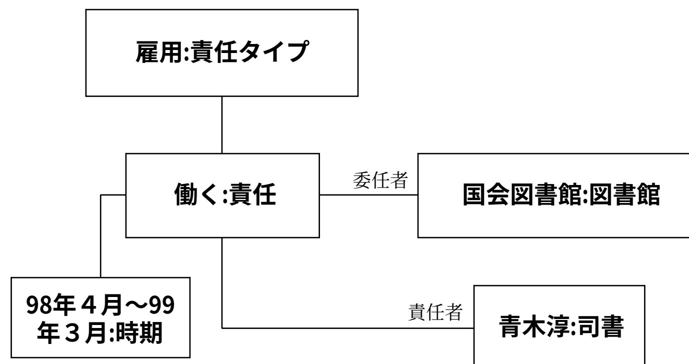

図 57 司書が働く場合の責任 (Accountability) パターン例

ドメインモデルのクラス図にあった利用者クラスと著者クラスの間の関連「検索する」は、図書館が著者を検索する方が自然だろうということで、図書館クラスと著者クラスの間の関連と考えられる。ただし、関連名は「持つ」の方がよい。図書館が著者の情報を持っていて、その情報を使って「検索する」という解釈である。こ

うしておけば、「検索する」以外のメッセージも送ることができる。実際に、「著者を登録する」といったメッセージを、この関連を通して送る必要が出てくる。

そうすると、「持つ」も責任クラスのインスタンスと考えることができる。この場合「責任タイプ」が「保持」、委任者が著者のインスタンス、責任者が図書館のインスタンスということになる。

以上をまとめると、責任 (Accountability) パターン部分を除いた全体のクラス図は以下のようになる。

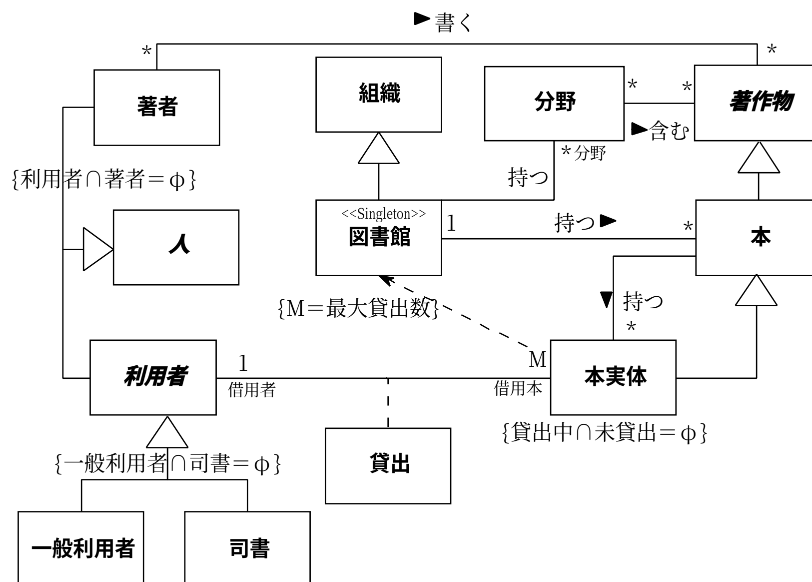

図 58 図書館の貸出管理の全体クラス図

詳細なクラス図は後述するとして、勘所はどこか検討してみよう。

このシステムで一番変更が予想されるのは貸出の規則であろう。この規則は図書館ごとに異なるはずなので、「貸出」ではなく、図書館が規則を持つのがよい[72]。図

72. 大域（グローバル）変数で規則を持つのは、保守性や再利用性を極端に低下させる最悪の選択である。

書館クラスが規則を持つ案と、一枚札 (Singleton) パターンをあてはめて図書館のインスタンスを 1 つとし、それが規則を持つ案が考えられる。後者の案の方が変更余波が少なく、将来このシステムが複数の図書館を対象としたシステムに成長したとしても耐えられるので、ここでは Singleton パターンを使った案を採用する。

次に変更が予想されるのは、図書館の組織や利用者種別の変更であろう。例えば、図書館に分館ができたり、司書以外の職員がシステムを使うようになることが考えられる。図書館に分館ができるといったような組織の変更は、責任パターンで対応できる。組織の上下関係などを責任クラスと責任タイプクラスのインスタンスとして追加すればよいのである[73]。利用者の種類の追加は、人クラスのサブクラス追加で対応できるし、人同士の関係が新たに必要になれば、やはり責任クラスと責任タイプクラスのインスタンスを追加すればよい。

分野クラスは、多彩な検索を行うためにいくつかのクラスに分けなければならなくなることはあり得る。しかし、このような変更のほとんどは設計モデル上の話だろうし、仮に分析モデルを変更しなければならないとしても、著作物クラスや図書館クラスとの関連は変える必要がなさそうなので、変更余波は小さい。

詳細なクラス図は以下のようになる。

---

73. この場合、図書館は Singleton にならないが、修正箇所は少ない。

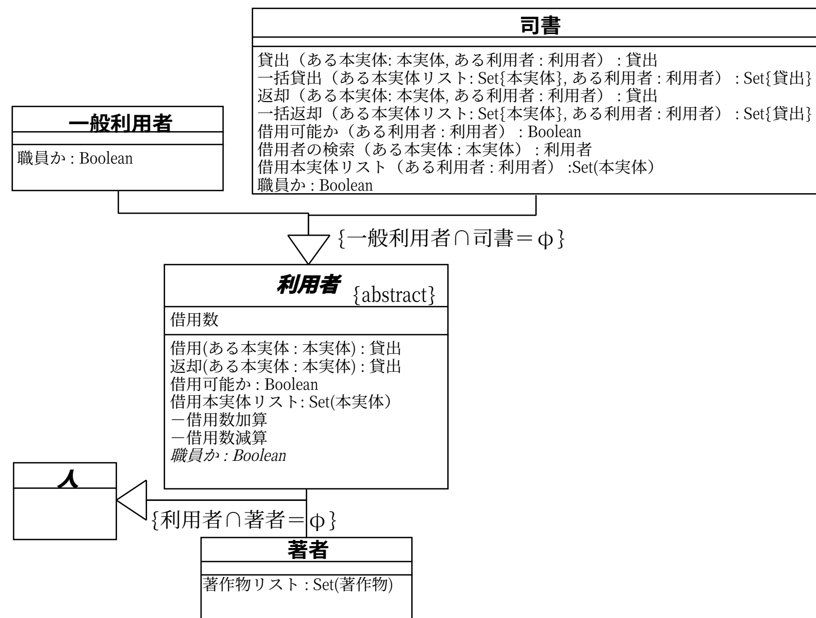

図 59 利用者関係のクラス図

ここではドメインモデルに比べて、便利さのために司書クラスの一括貸出と一括返却操作が追加した。また、ドメインモデルの検証で発見した、忘れていた借用数加算と減算の操作も、利用者クラスの private 操作[74] として追加した。同じく返却操作が貸出のインスタンスを返すようにして、司書クラスの返却操作の定義と矛盾しないようにした。

74. 他のクラスから参照できず、そのクラスの他の操作が使う私的な操作のこと。操作名の前にハイフン「ー」を付けて表す。

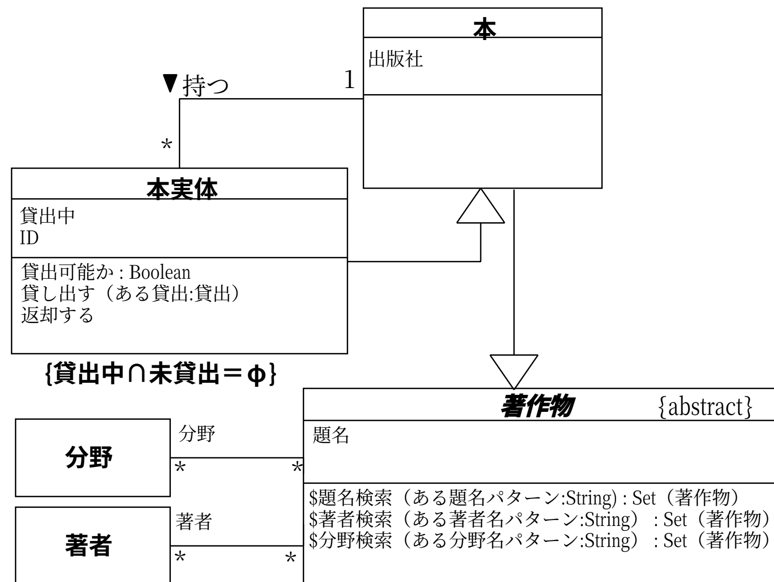

図 60 本関係のクラス図

ここでは、便利さのために、著作物クラスのクラスレベル操作である題名検索・著者検索・分野検索のパラメータとしてすべて文字列のパターン[75]を指定することにした。例えば「＊ソフトウェア工学＊」という文字列パターンをパラメータとして分野検索を行うと、分野名中に「ソフトウェア工学」という文字列を含む著作物の集合を返す。あるいは、「R[a-z] ＊」で著者名を検索すると、R で始まり小文字のアルファベットが任意の個数続く著者名を持つ著作物の集合を返す[76]。

75. 厳密に言えば、正規表現を指定する。正規表現については、『A.V. エイホ他著、原田賢一訳、コンパイラ I　原理・技法・ツール、サイエンス社、1990 年』を参照のこと。
76. 例えば、Ralph Johnson を著者として持つ本の集合が返る。

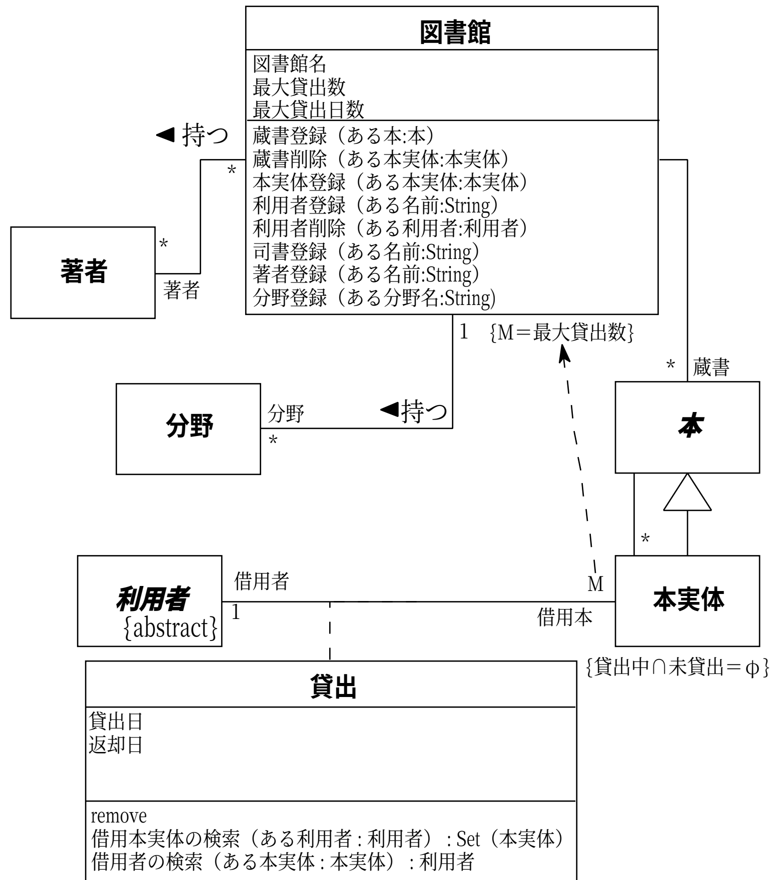

図 61 貸出関係のクラス図

ここでは、ドメインモデルの検証で発見した図書館クラスの本実体登録操作を追加し、図書館クラスの操作のパラメータの型を定義した。

4.3.2 操作仕様の記述

システム分析の操作仕様記述は、システムが何をするかを各操作ごとに完全に記述する。記述すべきものは操作のインタフェース[77]・前件・後件であり、どうやってその操作を実現するかの記述は一切書かない点は、ドメインモデルの操作仕様の場合と同じである。

以下では、ドメインモデルと異なる主要な操作のみ記述する。

仕様 20 司書クラスの仕様

司書クラスに追加した一括登録と一括返却の操作仕様を以下に示す。

(1) **一括貸出（ある本実体の集合** : Set(**本実体**), **ある利用者** : **利用者）**: Set(**貸出**)

ある利用者に、本実体の集合を貸し出す。

- **前件**

```
-- ある利用者が存在し、図書館の認証利用者であること。
Set{ ある利用者 }->notEmpty   and
責任 ->exists(s : 責任 |
        s. 責任タイプ = # 認証利用者  and  s. 責任者 = ある利用者 )
```

- **後件**

```
-- ある利用者の借用操作の結果が空でない本実体に関わる貸出が
-- この操作の結果である。すなわち、貸し出せる本実体だけ貸し出す。
result  = 貸出 ->select ( k : 貸出 |
        ( ある本実体の集合 ->forAll ( each : 本実体 |
                ある利用者 . 借用 (each)->notEmpty)))
```

(2) **一括返却（ある本実体の集合** : Set(**本実体**), **ある利用者** : **利用者）**: Set(**貸出**)

ある利用者の借用している、本実体の集合の返却処理を行う。

77. 操作名・パラメータ・返値の組を表す。

- **前件**

  -- ある利用者が存在し、図書館の認証利用者であること。

  Set{ ある利用者 }->notEmpty and

  責任 ->exists(s : 責任 | s. 責任タイプ = # 認証利用者 and s. 責任者 = ある利用者)

- **後件**

  -- ある利用者の返却操作の結果が空でない本実体に関わる貸出が

  -- この操作の結果である。すなわち、返却できる本実体だけ返却する。

  result = 貸出 ->select (k : 貸出 |

  (ある本実体の集合 ->forAll (each : 本実体 |

  ある利用者 . 返却 (each) ->notEmpty)))

仕様 21 著作物クラスの仕様

著作物クラスのクラスレベル操作である題名検索・著者検索・分野検索は、ほぼ同じ仕様なので、以下には著者検索操作の仕様のみ示す。

(1) **著者検索（ある著者名パターン** :String**）**: Set**（著作物）**

- **前件**

  -- 私のインスタンスが空でないこと。

  self.allInstances->notEmpty

- **後件**

  -- 結果の著者の名前は、ある著者名パターンで指定された正規表現と一致する。

result->forAll(author : 著者 | 正規表現一致[78] (author. 名前 , ある著者名パターン))

仕様 22 図書館クラスの仕様

図書館クラスに追加した本実体登録操作の仕様を示す。

(1) **本実体登録（ある本実体 : 本実体）**

- **前件**

  -- 対応する本の集合が空でない。

  ある本実体 . 本 ->notEmpty

- **後件**

  -- ある本実体に対応する本の本実体の集合に、ある本実体を追加したものが、

  -- 追加後の本実体の集合になる。

  ある本実体 . 本 . 本実体 =

  (ある本実体 . 本 . 本実体 @pre)->including(ある本実体)

4.3.3 状態遷移図の作成

状態遷移図は、ドメイン分析で示した本実体オブジェクトと利用者オブジェクトの 2 つに加えて、以下のオブジェクトの状態遷移図を示せば十分であろう。その他のオブジェクトは、興味ある振る舞いをしない。

本オブジェクトは、対応する本実体が有る場合とない場合を区別しなければならないので、以下のような状態遷移図になる。

78. 正規表現の一致を判定する操作の仕様はこの本の対象範囲を超えるので、『A.V. エイホ他著、原田賢一訳、コンパイラ I　原理・技法・ツール、サイエンス社、1990 年』を参照して欲しい。通常、自分で作成しなくても UNIX や GNU の C 言語ライブラリーとして存在するので、それらを変換器 (Adapter) パターンを使用してラップして使うとよい。

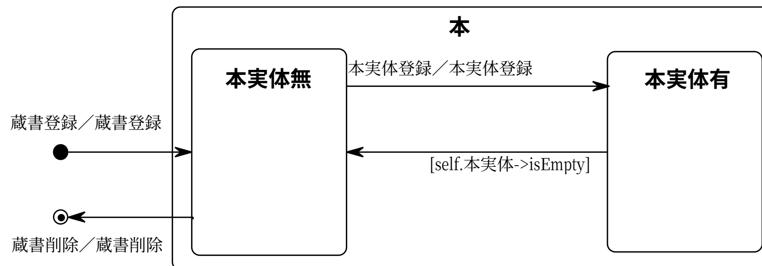

図 62 本オブジェクトの状態遷移図

貸出オブジェクトは期限超過かどうかを区別する必要があるので、以下のような状態遷移図になる。ここで、N 日というのは借用者に催促をする間隔（催促間隔日数）を示す。

「返却日が過ぎたら期限超過になる」という要求はUseCaseやシナリオには出てこなかったが、現実のシステムでは、このように「ユーザーが明示的に要求しない要求事項」はたくさんある。

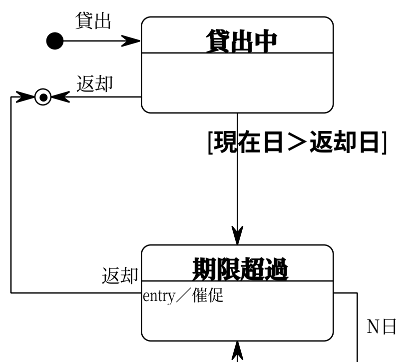

図 63 貸出オブジェクトの状態遷移図

このシステムでは、以上の状態遷移図でシステムの振る舞いを考えれば十分だろう。より複雑な例は、次の章で紹介する。

4.3.4 分析モデルの検証

分析モデルの検証は、ドメインモデルの検証と同じ方法で行う。ここでは、説明の重複を避け、検証で見つかった結果のみ紹介する。

まず、貸出オブジェクトの状態遷移図のところで見つかった「期限超過」に関係して、利用者クラスの「貸出可能か」操作の定義を変更する必要がある。

仕様 23 利用者クラスの仕様

(1) **借用可能か** : Boolean

図書館の規則によって、私が借用可能かどうかを返す。

- **前件 [貸出あり]**

self. 借用本 ->notEmpty

- **後件 [貸出あり]**

-- 私の借用数が、図書館の最大貸出数以下で、かつ

-- 私の借用で期限超過したものがなければ真を返す。

result = self. 図書館 . 最大貸出数 >= self. 借用本 ->size and

self. 貸出 ->forAll(返却日 >= Date. 現在日[79])

- **前件 [貸出なし]**

self. 借用本 ->isEmpty

- **後件 [貸出なし]**

result = true

次に、図書館の属性として催促間隔日数を追加し、操作として「催促」を追加する。

また、図書館クラスの蔵書削除操作は本実体を削除していたが、本と本実体の削除は別々な操作で行った方がよさそうである。そこで、今までの「蔵書削除」は「本実体削除」に名前を変え[80]、「蔵書削除」は本を削除することにする。

79. 日付を処理する Date クラスがあり、現在日を求める操作があるものとする。

仕様 24 図書館クラスの仕様

(1) **催促（Set(貸出)）**:Set(**貸出**)

催促すべき貸出の集合を返す[81]。

- **前件 [貸出あり]**

Set(貸出)->notEmpy

- **後件 [貸出あり]**

result->forAll(each : 貸出 |

Date.addDays[82] (each. 前回催促日 , self. 催促間隔日数) > Date. 現在日)

- **前件 [貸出なし]**

Set(貸出)->isEmpy

- **後件 [貸出なし]**

result = { }

(2) **蔵書削除（ある本** : **本）**

ある本を削除する。ある本に対応する本実体があるときは削除できない。

- **前件**

-- ある本に対応する本実体が存在しない

ある本 . 本実体 ->isEmpty

- **後件**

-- 実行前の本の集合からある本を削除したものが、実行後の本の集合である。

Set(本) = Set(本)@pre->excluding[83] (ある本)

80. 操作名が変わっただけなので、仕様は省略する。
81. 具体的な催促方法はここでは決めないことにする。
82.Date クラスに、ある日付に日数を足した日付を求める addDays 操作があるものとする。
83.() 内の要素を削除した集合を返す Set クラスの既定義操作。

ここまでで一応検証済みの分析モデルができたことになる。クラス図とそこに出てきた操作の仕様および状態遷移図という、どのシステムを定義するのにも必要な分析モデルの主要な要素が定義された[84]。

システムが何をやりたいかということは、かなり細かいレベルまで定義したが、「どうやって作るか」は一切記述しなかったつもりである。それを行うのが、次の設計工程である。

4.4 設計

設計工程では、システムが何をすべきかを記述した分析工程と異なり、システムをどのように作成していくかを記述する。設計で行うべき作業は以下の通りである。

- **アーキテクチャの決定**
- **クラス図の修正**
- **状態遷移図の修正**
- **デザインパターンの適用**
- **操作仕様の変換**

  前件・後件などで宣言的に記述された操作仕様を、手続き的仕様に変換する。
- **状態遷移の実装方法決定**
- **関連の実装方法決定**
- **効率の検討**
- **再利用性・保守性分析**
- **設計モデルの検証**

以下に、各作業の説明を行う。

84.UML 記法には、大規模システムの記述で必要になることもある他の図があるが、いつも使うわけではないし、このシステムでは必要ないので省略した。

4.4.1 アーキテクチャの決定

アーキテクチャの決定は、本来、設計工程の最初に行うのは難しい。決定が誤りであった場合の影響が大きすぎるからだ。かといって、決まった手順を踏めばアーキテクチャを導出することができるほどソフトウェアの開発技術が進んでいるわけではない。そこで、同種のシステム構築の経験あるいはアーキテクチャ・パターンが有効となる。

さて、図書館の貸出管理システムの場合、よほど小さな図書館でない限り、クライアント / サーバー・パターンを適用するのが普通であろう。「実は、うちの図書館にはすでに Macintosh と PC 互換機があって、それらをできるだけ使いたい。それから、うちの図書館には分館があって、そちらに運用担当者は置けない。」などという、運用や実装に関係する要求に対応しやすいためである。[85]

ここでは設計を簡単にするために分散 OODB である GemStone を採用することにする。GemStone は 3-tier 構成のクライアント / サーバー・パターンを包含し、分散処理機能も提供する。外部スキーマとしてのプロセスは GemBuilder と呼ばれ、Smalltalk あるいは Java から呼び出すことができる。概念スキーマとしてのプロセスは Gem と呼ばれ、論理的に 1 つのプロセスが、物理的に複数のサーバー上にある複数個の Gem で実現できる[86]。内部スキーマは Stone と呼ばれるプロセスで、これも論理的には 1 つだが、物理的に複数個のサーバー上にある複数個の Stone で実現できる。Gem や Stone の構成は、実行時にシステムを止めずに変更できる。また、2 相コミットを実現したトランザクション処理・2 相ロックを実現した並行制御・オブジェクト単位のロック・複製（replication）などの機能を提供する。

---

85. これらの要求は分析モデルでは記述できないので、自然言語による文書として分析工程でまとめておく必要がある。
86. クライアント側に置くことすらできる。

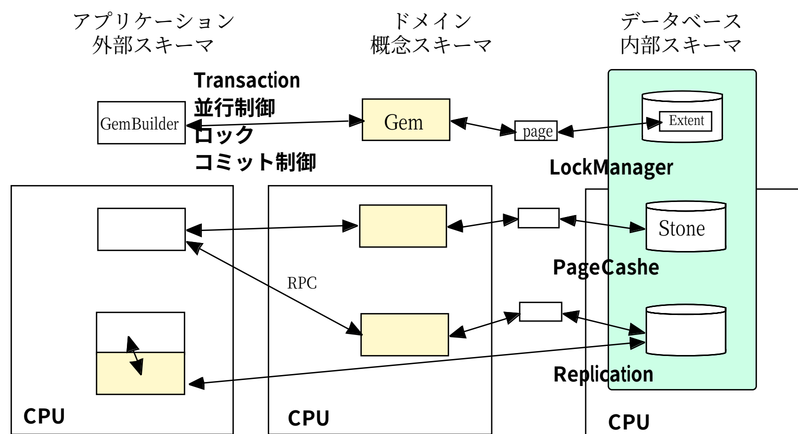

図 64GemStone の構成

ただし本書では、外部スキーマに相当する GUI アプリケーションの記述は省略し、概念スキーマに当たる図書館貸出管理システムのアプリケーション階層 (Layer) に絞って話を進めるので、読者は GemStone のことをあまり意識しなくてよい。

C++ を開発に使っている読者は、CORBA/ORB[87] を採用することを想定していただけばよい。GemStone より機能・性能は劣るが[88]、今回のシステムくらいでは、ほぼ同等の機能が提供される。また、開発言語に依存しないシステム作りが可能になる利点もある。

4.4.2 デザインパターンを考慮したクラス図の修正

分析モデルのクラス図に対して、設計モデルでは (1) デザインパターンを考慮した修正、(2) 実装のための属性と操作の追加、(3) 属性と操作のパラメータと返値の型の確定、などを行う。

87.『T.J. トーマス , R.C. マルボー著、大谷真監訳、CORBA Design Pattern、IDG コミュニケーションズ、1998 年』参照。
88.DBMS として何を使うかに依存するが、GemStone より優位になることはほぼあり得ない。

責任 (Accountability) パターンと当事者 (Party) パターンに関連するクラス図は以下のようになる。人クラスにあった名前や電話番号といった属性は当事者クラスに移動した。時期クラスの属性の型は、このシステムでは日付に関する機能を持った Date 型あるいはクラスで十分であるが、日時を持った型あるいはクラスの方が汎用性はある。責任クラスは、このシステムでは 4 つの関連以外の属性は持たない。

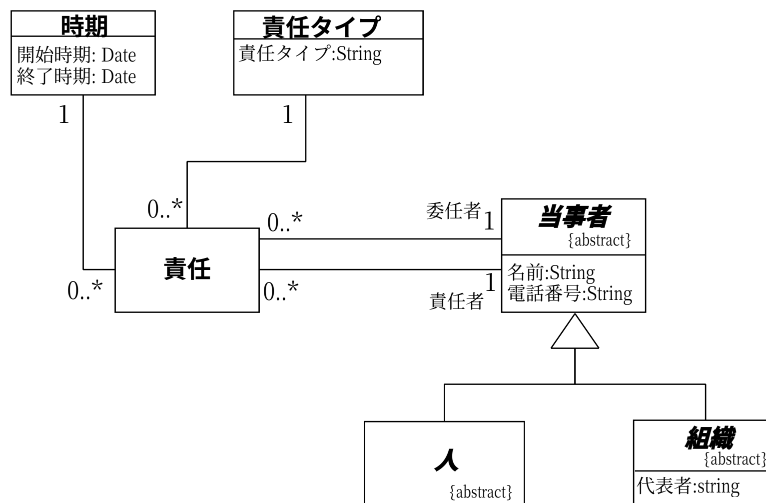

図 65 責任 (Accountability) パターン関係のクラス図

その他のクラスの全体図は以下のようになる。図書館は一枚札 (Singleton) パターンで、本と本実体は混成 (Composite) パターンになる。本クラスは混成 (Composite) パターンの構成要素 (Component) と混成 (Composite) の両方の役割を果たす。通常の混成 (Composite) パターンと異なるのは、本クラスのオブジェクトが葉 (Leaf) としての本実体は持つが、入れ子になった本オブジェクトは持てない点である。

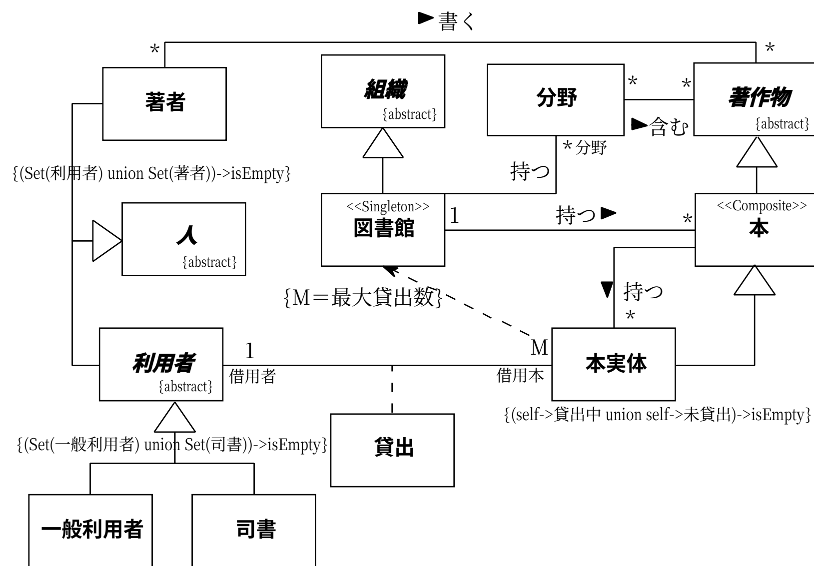

図 66 図書館貸出管理システムの全体クラス図

人関係のクラス図は以下のようになる。利用者クラスに private 操作である貸出限度超過を追加した。

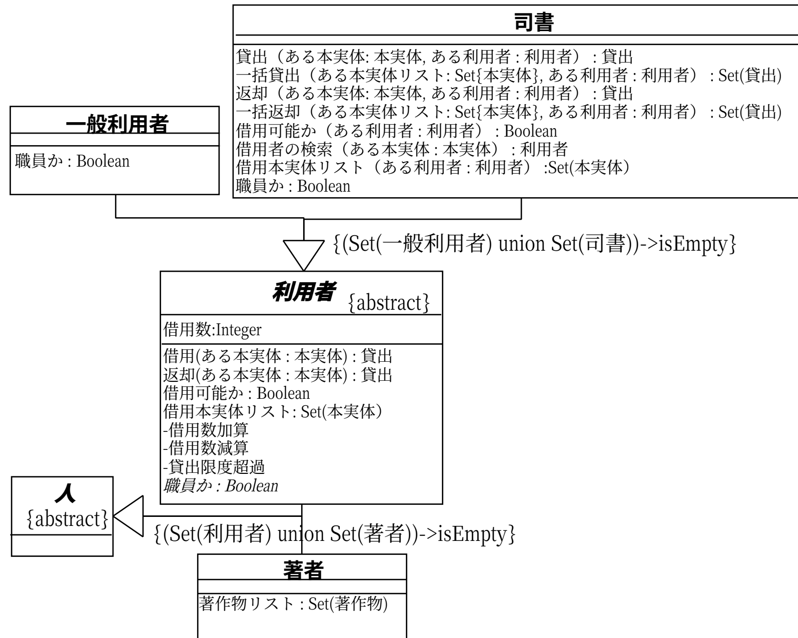

図 67 人関係のクラス図

本関係のクラス図は以下のようになる。

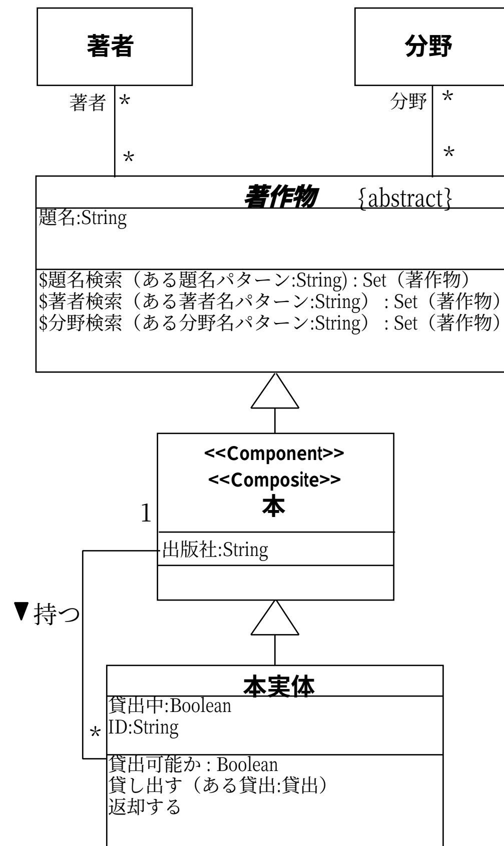

図 68 本関係のクラス図

貸出関係のクラス図は以下のようになる。図書館クラスに「催促間隔日数」属性と「催促」操作を追加し、貸出クラスに前回催促日を追加した。

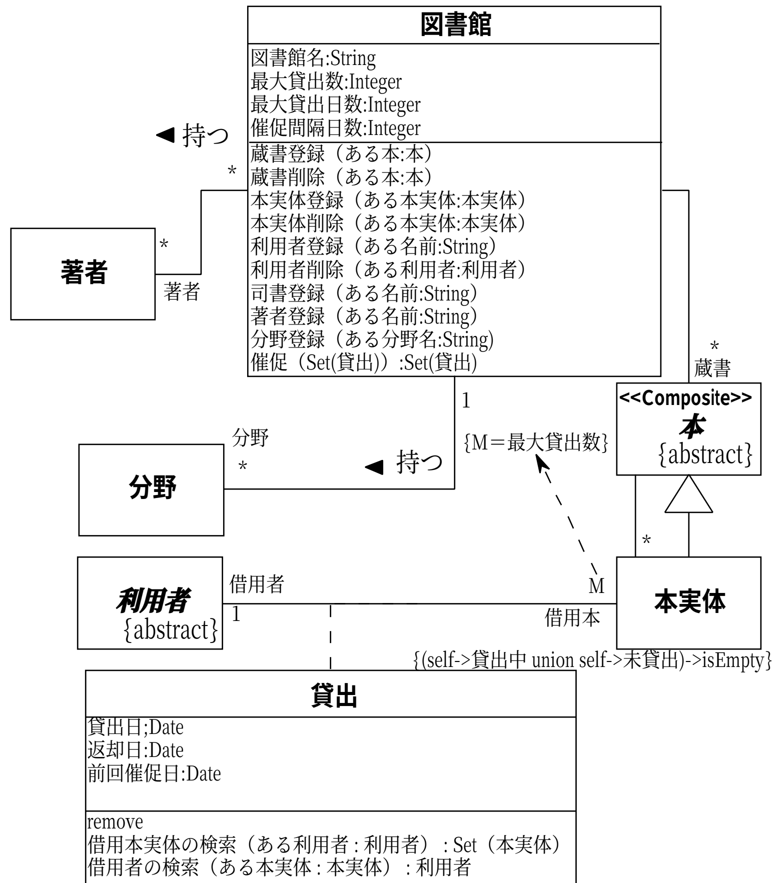

図 69 貸出関係のクラス図

4.4.3 **効率の検討**

クラス図の修正が一応終わった段階で、効率の検討を行う。効率の検討は以下のように行う。

- **インスタンスの数 (最大・最小・平均・分散）を調べる。**

  オブジェクトの数やリンク[89]の数を調べ、効率上問題あるときには、クラス構成を変更したり、オブジェクトへアクセスするパスの最適化を考える。

- **実行効率・空間効率など各種の効率に考慮しながら、算法 (Algorithm) を選択する。**

  操作のコストを最小にするクラス構造と算法 (Algorithm) を選択する。

- **プロトタイプを作成し、測定ツールを使用する。**

  実用的プログラムの効率の推測は非常に難しいことが分かっているので、効率的にネックになりそうな部分のプロトタイプを作成し、測定ツールを使用して計測する。

- **シミュレーションツールで、モデルを評価する。**

  プロトタイプを作成するのが困難なとき、シミュレーション用モデルを構築して、効率をシミュレートする。もちろん、シミュレーションで効率の悪い部分の当たりをつけておき、その部分のプロトタイプを作成してもよい。

本システムの場合、オブジェクト数が多いものを考えると、本実体オブジェクトか貸出オブジェクトということになるだろう。また分野クラスと著作物クラスの間の関連のリンク数も多そうである。

例えば、本の数が 80 万冊、本実体の数が 100 万冊、利用者数が 1000 人で平均貸出数が履歴も含めて 100、分野数が 1000 分野で 1 つの本当たり平均 3 分野に対応するとすると、貸出オブジェクトは 10 万個、分野と著作物の間のリンクすなわち分野オブジェクトと本オブジェクトの間のリンク数は 240 万個にもなる。貸出オブジェクトは時間と共に増大することが明らかである。

最大の問題は、題名で本を検索するところだろう。題名検索操作は正規表現パターンで題名を検索するが、正規表現による検索は全数の順次検索になり、80 万冊分検索しなければならない。しかも 1 冊分の題名と正規表現パターンが一致するかを検

89. リンクは関連のインスタンス。

索するのに O(題名の長さ×正規表現パターンの長さ)[90] の計算時間が必要となることが分かっている[91] ので、対話的プログラムとしては到底成り立たないことになる。

このような場合は、要求定義の変更をユーザーにお願いしなければならない[92]。題名検索のパターンとして多いのは、題名の頭の何文字かあるいは終わりの何文字かを指定するケースであろう。従って、例えば、頭の何文字かあるいは最後の何文字かの指定は正規表現を使わないことにすれば、効率をかなり改善できる可能性がある。何文字で許容範囲に入るかは、データを調査しなければ分からないが、ここでは、仮に頭の 2 文字までで共通の題名を持っている本は高々 1000 冊ほどになるとしよう。このような仕様変更ならば、図書館の司書も許容できる範囲であろう。

さて、このように仕様変更した場合、どのように効率改善を行うかが問題になる。ここでは、算法 (Algorithm) パターンのうちのハッシュ法を適用し、題名の頭 2 文字をパラメータとしたハッシュ関数を持つハッシュ表を考え、その要素数を 64K 個、ハッシュ表の各要素を 2 バイト、同じハッシュキーを持つ本が高々 1000 冊とする。こうすると、6400 万冊ほどの本をメモリー上に置いた 128K のハッシュ表で検索できる。

もちろん、題名の部分は、平均40 バイトで 100 万冊とすれば 40MB で、リスト構造のためのポインターを 4 バイトとすればその部分が 4MB になるので、運用環境によってはメモリー上に置くには大きすぎるかもしれない。この場合は、題名のリストはディスク上に置くことになるが、同一ハッシュアドレスの題名リストは高々 1000 冊× 40 バイト＝ 40KB なので、キャッシュに入れることができ、ほぼ 1 回のディスクアクセスで検索が完了する。従って、性能上の問題はディスク上に題名を置く場合でもクリアーできた。

また、ここでは詳細を説明しないが、題名リストのかわりに、同一ハッシュアドレスの題名を木構造に格納する手があり、同一ハッシュアドレスのデータが多い場合、劇的に効率が改善されることが分かっている。

90.O(n) は計算時間を表す記法で、n に比例した計算時間が掛かることを示す。

91.『島内剛一他編、アルゴリズム辞典、共立出版、1994 年』参照。

92. 本来ならば、要求仕様を分析する段階でこのことに気づくべきだという議論はあろうが、往々にして現実のプロジェクトではそうはならない。

ハッシュ関数の仕様は以下のようにすれば、ある程度の性能が得られることが分かっている。

仕様 25 題名検索ハッシュ関数の仕様

(1) hash(**題名** : String) : Integer

Horner 法を用いて、文字列をハッシュ表のサイズ内の整数に変換する。

- **手続き的仕様**[93]

```
hash( 題名 : String) ≡
maxSize := 65536;    -- 64K= ハッシュ表のサイズ。
hashKey := 0;
 i := 1;
while  題名 ->size  + 1  >  i do
    hashKey := (hashKey * 10 + 題名 (i)->codeValue).mod[94] (maxSize);
    -- 表の範囲内に納めるため、.mod(maxSize) で「余り」を求める。
    -- codeValue は文字コードを整数として返す関数とする。
    i := i + 1
end;
result := hashKey
```

ハッシュ表を題名検索に使うことによって、著作物クラスに属性として持っていた題名を、題名ハッシュ表クラスおよび題名 Link として独立させることになるので、クラス図は以下のように変更する。

93.OCL は手続き的仕様を書けないため、ここでは手続き的仕様に関する構文のみ Raise Specification Language (RSL) 準拠の仕様で記述した。RSL については、『The RAISE Language Group, The RAISE Specification Language, Prentice Hall, 1992』参照のこと。

94.mod は余りを求める操作。

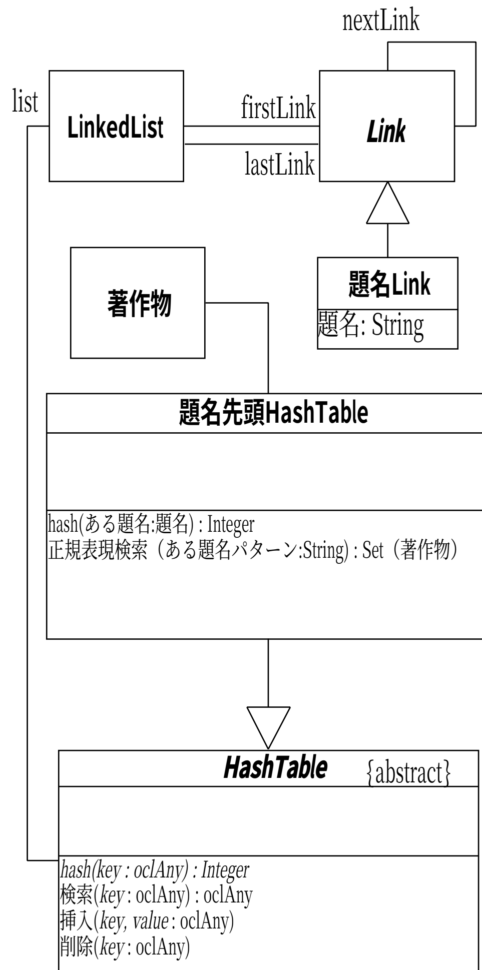

図 70 題名リストとハッシュ表関係のクラス図

プロトタイプとして、著作物クラスと本クラスおよびハッシュ表関連のクラスを実装すれば、性能上一番問題になりそうな題名による検索の効率が実用に耐えるかを早い段階で確認できる。上の説明で使用したハッシュ関数の仕様は一番単純なものであり、ここを改良することがプロトタイプの主な目的になる。

シミュレーションによる効率検討は、このシステムの場合は必要ないだろう。

次に効率上問題になりそうな、分野による検索はリンク数こそ 240 万個ほどと多いものの、1 分野の本はたかだか数万冊であろうし、それをリンクとして直接持っているので、検索による効率上の問題は避けられそうである。唯一問題になりそうなのは、分野分けの基準の変更であろう。今までの1 つの分野が複数に分かれたり、複数の分野が 1 つに統合される場合である。しかし、これは対話的アプリケーションの機能ではなく、何らかの切換プログラムを作成して一括変更するはずであり、今現在は考えなくてもよいだろう[95]。

次に問題になりそうな貸出オブジェクトの増加であるが、現在 10 万件で 1 日 1000 貸出発生したとしても、3 年ほどで 10 万件増える程度であり、この問題への対応も先延ばしできそうである[96]。

効率上の問題は、実装言語やデータベースシステムといった実装環境に依存する場合もあるが、通常それらはミクロの効率化の問題であり、上に述べたようなマクロ的な効率改善に比べると、1 桁か 2 桁改善率が落ちる。実装環境に依存する改善では 10 倍の効率改善が精一杯であるが、算法 (Algorithm) の改良などは数十倍・数百倍そして時には悪い算法 (Algorithm) では実現できないことが達成できるといった効果を生むことが多い。

4.4.4 操作仕様の変換

さて、効率の改善を考慮したクラス図の改良を終えた段階で、次は、操作仕様を分析モデルでの宣言的仕様から手続き的仕様に変換していく。もちろん、手続き的仕様を書くというのは一種のプログラミングであるから、この工程は実装工程の一部であるとも言える[97]。

---

95. もちろん、何も考えないのではなく、切換プログラム作成可能なクラス構成であることや、切換は休日や夜間の一括更新で対応できそうな見込みくらいは付けておく。
96. これも、古い貸出オブジェクトは待避することで対応できそうである。
97. 筆者の属するオブジェクト指向グループは、設計と実装は不可分と考えている。設計モデルをプログラマーに渡してコーディングしてもらうというのは、アセンブラーや機械語によるプログラミングをしていた時代の風習であり、現在のオブジェクト指向言語のような高水準言語による開発には合わないと思っているのである。

(1) 宣言的仕様から手続き的仕様への変換

宣言的仕様から手続き的仕様への変換は、行き当たりばったりに行うと欠陥が多数発生する[98]。そこで、手続き的仕様への変換は「段階的詳細化[99]」と呼ばれる目的指向の手法で行う。

(2) 仕様の段階的詳細化

段階的詳細化は、宣言的仕様の前件の条件を徐々に後件の条件に近づけていく。しかし、このような作業はかなり難しいので、実際には、下図のように後件の条件を緩めて前件（P_0）から後件（P_3）に至る途中の条件（P_2, P_1 など）を導出する。

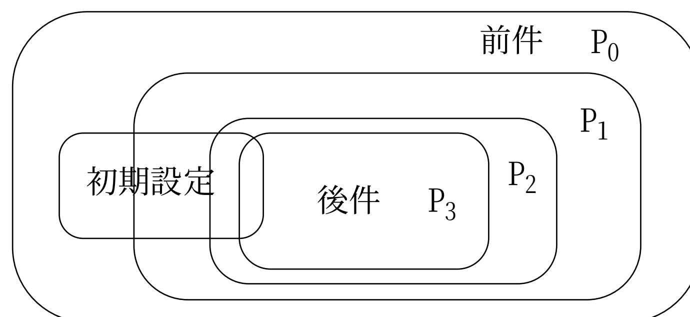

図 71 後件・前件・初期設定の関係

後件を緩めて、前件との中間になる条件を求めるには、以下の方法がある。

- **積項を取り除く**

  項 A, B, C があって、A ∧ B ∧ C のような形をしているとき「積項」と呼ぶ。そのうちの 1 つの項を取り除いて条件を緩める。

---

98. 開発現場の多くのプロジェクトでは、宣言的仕様もなく手続き的仕様を作成するか、手続き的仕様もなくプログラムを作成するので、さらに欠陥が多い。
99. 一種の開発プロセスのパターン。『D. グリース著、筧かつ彦訳、プログラミングの科学、培風館、1991 年』あるいは、『The RAISE Language Group, The RAISE Development Method, Prentice Hall, 1995』参照のこと。

例えば、3 章算法 (Algorithm) パターンのところで紹介した辞書パターン中の HashTable クラスの検索操作の後件の場合を考えてみる。後件は以下のようになっていた。

(self.hash(result) = self.hash(key） and

要素 Link.allInstances->exists(each : Link | key = each.key)) or

((要素 Link.allInstances->exists(end : Link | end.nextLink->isEmpty) and

要素 Link.allInstances->forAll(each : Link | key <> each.key))

implies result = false)

ここで、

A = self.hash(result) = self.hash(key）

B = 要素 Link.allInstances->exists(each : Link | key = each.key)

C = 要素 Link.allInstances->exists(end : Link | end.nextLink->isEmpty)

D = 要素 Link.allInstances->forAll(each : Link | key <> each.key)

E = result = false

と置けば、後件は

(A and B) or ((C and D) implies E)

となり、以下のように変形していくことができる。

(A and B) or (not (C and D) or E) -- implies の定義

(A or (not (C and D) or E)) and (B or (not (C and D) or E)) -- 分配則

(A or not C or not D or E) and (B or not C or not D or E) -- ド・モルガンの法則

(A or not C or not D or E) and (B or not C or E) -- B = not D により簡約化

ここで、後の方の項 (B or not C or E) を取り除き、

(A or not C or not D or E)

すなわち

(sel.hash(result) = self.hash(key)）or

not (要素 Link.allInstances->exists(end : Link | end.nextLink->isEmpty)) or

not (要素 Link.allInstances->forAll(each : Link | key <> each.key)) or

(result = false)

とするのだ。

- **定数を変数に置き換える**

  例えば、$1 \leqq d \leqq 10$ というような条件があったら、$1 \leqq d \leqq i \wedge 1 \leqq i \leqq 10$ に置き換えてみるという方法である。

- **変数の領域を広げる**

  例えば、$1 \leqq d \leqq 10$ という条件があったら、$0 \leqq d \leqq 100$ に置き換えてみるという方法である。

- **和項を追加する**

  これは、A という項があったら、$A \vee B$ を考えてみようという方法であるが、試行錯誤になってしまい、行き当たりばったりのプログラムと同様になりかねないので避ける。

さて、同一ハッシュキーの線形リストの firstLink から lastLink まで反復させて手続き的仕様を作ろうと決心すると、反復の上限を決める「蓋」の条件は、先ほど取り除いた積項の否定を使えばよいことが分かっている。

今の場合、

not (B or not C or E) = not B and C and not E

すなわち、

not (要素 Link.allInstances->exists(each : Link | key = each.key)) and
要素 Link.allInstances->exists(end : Link | end.nextLink->isEmpty) and
result <> false

変換すると

(要素 Link.allInstances->forAll(each : Link | key <> each.key)) and
要素 Link.allInstances->exists(end : Link | end.nextLink->isEmpty) and
result <> false

である。

また、反復の最中変わらない不変条件は、残りの積項すなわち

(self.hash(result) = self.hash(key)) or
not (要素 Link.allInstances->exists(end : Link | end.nextLink->isEmpty)) or
not (要素 Link.allInstances->forAll(each : Link | key <> each.key)) or

(result = false)

となる。反復が終了することを保証する「限度関数」t は

d = 探索した Link オブジェクトの個数

として

t = 要素 Link.allInstances->size － d

とすれば、t は常に 0 以上であり、かつ d は増加するのだから、段々 0 に近づいていくことが分かり、反復が終了することを保証できる。

手続き的仕様の大筋は以下のようになるだろう。

初期設定
while 蓋の条件 **do**
  ループを次へ進める
end;

初期設定は、反復処理に入る前に、反復中の不変条件を満たすように設定しなければならない。まず、不変条件の最初の 2 つの条件

(self.hash(result) = self.hash(key)）or

not (要素 Link.allInstances->exists(end : Link | end.nextLink->isEmpty))

を満たす仕様を考える。これは、要するに指定された key と同じハッシュアドレスを持つ線形リストの要素は、反復中は空でないということであるから、

result := self->at[100] (hash(key)).list.firstLink;

として、線形リストの先頭を取りあえず結果に入れておけばよい。不変条件の 3 番目と 4 番目の条件は、見つけるべき要素があるかないかであるということなので、これは反復中にどちらかの条件が満たされる結果になればよいので、初期設定する必要はない。手続き的仕様は以下のようになる。

result := self->at(hash(key)).list.firstLink;
while 蓋の条件 **do**
  ループを次へ進める
end;

100.sequence->at(i) で sequence の i 番目の要素を示す。

蓋の条件は、以下のように変形できる。

```
key  <> result.key  and result <> false
```

蓋の条件の 1 番目の項

「要素 Link.allInstances->forAll(each : Link | key <> each.key)」

は、線形リスト中の任意の場所で成り立つのだから key <> result[101] .key としてよい。

蓋の条件の 2 番目の項

「要素 Link.allInstances->exists(end : Link | end.nextLink->isEmpty)」

は、反復中に線形リストの要素で nextLink が空になるもの (要するに lastLink で指されている最後の要素）があることを示しているので、これは反復から抜ける条件として使い、蓋にはしない。ここまでで、手続き的仕様は以下のようになる。

```
result := self->at(hash(key)).list.firstLink;
while  key  <> result.key  and result <> false  do
    if result.nextLink->isEmpty then
        result := false -- 検索失敗
        exit  --   操作を抜ける
    else
        反復を次へ進める
    end
end;
```

反復を次に進めるには、nextLink を辿ればよいので、

```
result := result.nextLink
```

とする。結局、手続き的仕様は以下のようになる。反復を抜けた時の result が求める結果である。

101.forAll(each : Link | key <> each.key) 中の each は束縛変数と呼び、他の名前に置き換えても意味は変わらないので、ここでは result に置き換えた。

仕様 26 ハッシュ表の検索操作の仕様

(1) hash(key : oclAny[102]) : oclAny

```
result := self->at(hash(key)).list.firstLink;
while key <> result.key and result <> false do
    if result.nextLink->isEmpty then
        result := false;  -- 検索失敗
        exit --  操作を抜ける
    else
        result := result.nextLink
    end
end;
```

(3) 算法 (Algorithm) の検索と適用

上で述べたように、仕様の段階的詳細化はかなりの工数が掛かる。もちろん、ソフトウェアの信頼性を高めるためには必要な作業なのだが、先人が段階的詳細化を行ったものについて、我々がまた同じ作業を行うのは無駄である。先人の作業は「算法 (Algorithm)」としてまとめられている。実は、上で述べたハッシュ表や線形リストの検索も、段階的詳細化をしなくても算法 (Algorithm) としてまとめられていたのだ。

このような算法 (Algorithm) を探し出すためには、例えば、以下のような本を揃えておき、必要なときに必要な算法 (Algorithm) を見つけることができるようにしておく。

102.oclAny は OCL のすべての型とクラスのスーパー型を示す。ここでは「任意のオブジェクト」の意味で使っている。

- **島内剛一他編、アルゴリズム辞典、共立出版、**1994 **年**
- **石畑清著、アルゴリズムとデータ構造、岩波書店、**1989 **年**
- A.V. **エイホ他著、大野義夫訳、データ構造とアルゴリズム、培風館、**1987 **年**
- R. **セジウィック著、野下浩平他訳、アルゴリズム、近代科学社、**1992 **年**

例えば、題名の検索で出てきた正規表現の検索算法 (Algorithm) の一部は、石畑氏の著書に Pascal プログラムとして記述されている。

また、最近のオブジェクト指向言語のクラスライブラリーをよく調べると、これらの算法 (Algorithm) の基本部分はかなり実装されている[103]。従って、自分が使っている言語のライブラリーはよく調べておく必要がある。

例えば、算法 (Algorithm) パターンのところで説明した 2 分木から要素を検索する算法 (Algorithm) は、以下のようになる。

仕様 27 2 分木の検索

```
(1)  treeSearch(key : oclAny, root : 2 分木 ) : oclAny
        if  root->isEmpty  then
            result := empty   -- 検索失敗
        elsif  key = root. 要素 then
            result := root. 要素   -- 検索成功
        elsif  key  <   root.leftTree  then
            result := treeSearch(key, root.leftTree)   -- 左部分木を検索
        else
            result := treeSearch(key, root.rightTree)   -- 右部分木を検索
        end
```

103.UNIX の C 言語ライブラリーでも、かなり実装されていた。

4.4.5 状態遷移の実装方法決定

状態遷移図は有限状態マシン[104]を表し、あらゆる有限状態マシンはプログラムとして実装できることが分かっている。そして、有限状態マシンの実装方法は、以下に示す３つの方法がある。

- **プログラム中の位置を使って状態を表す**
- **有限状態マシンを直接実装する**
- **並行タスクを利用する**

(1) プログラム中の位置を使って状態を表す

この方法は、伝統的な方法であり、今のところ最も一般的に使われている。有限状態マシンを簡単に実装できる反面、構造化の原則に違反したコードになるのが普通[105]なので、修正が困難であるという欠陥を持つ。

この方法では、プログラム中の制御位置が、プログラムの状態を暗黙に示す。例えば、利用者オブジェクトの状態遷移図を実装した、借用か返却のどちらかのコマンド[106]を処理するコマンド操作を考えると、手続き的仕様の大筋は以下のようになる。

仕様 28 コマンド処理操作の手続き的仕様大筋

(1) **コマンド** (コマンド : Command) : **貸出**

```
    ( 命令 , 本実体 ) := コマンド解析107 ( コマンド );
    if 命令 = # 借用 or 命令 = # 返却 then
        self. エラー処理 (’ 存在しない命令を受けた。’)
    elsif self. 借用数 = 0 then -- 「未借用」状態
        result := self. 未借用処理 ( 命令 , 本実体）
```

104.有限個の状態とその上での状態遷移の規則が定義された、計算機構の数学モデルのこと。
105.goto 文を使うのが、一番簡単！な実装方法になる。
106.コマンド (Command) パターンを採用したことになる。
107.通訳 (Interpreter) パターンを使う処理になる。

```
elsif self. 借用数 > 0 then --「借用中」状態
    result := self. 借用中処理 ( 命令 , 本実体 )
elsif
    self. エラー処理 (' 想定していない状態になった。')
end
```

(2) **未借用処理** (**命令** : String, **ある本実体** : **本実体**): **貸出**

```
if 命令 = # 借用 then
    result := self. 借用 ( ある本実体 )
elsif 命令 = # 返却 then
    self. エラー処理 (' 借りていない本を返そうとしている。')
end
```

(3) **借用中処理** (**命令** : String, **ある本実体** : **本実体**) : **貸出**

```
if 命令 = # 借用 then
    result := self. 借用 ( ある本実体 )
elsif 命令 = # 返却 then
    result := self. 返却 ( ある本実体 )
end
```

上の例の場合は、小さな状態遷移であり、変更もあまり考えられないので問題は少ないが、一般にこの方法では、状態が増えたり減ったりする度に if 文を挿入したり削除しなければならないので変更余波が大きくなる。

この欠点を克服するには、状態 (State) パターンを使う。利用者オブジェクトの場合に状態 (State) パターンを適用するとすると、以下の図のようになる。ここで、操作や属性は状態遷移に関わるものだけ記述した。ここで、ロール名 state は、利用者オブジェクトの状態（未借用か借用中）を表す利用者状態クラスのサブクラス（利用者未借用クラスか利用者借用中クラス）のインスタンスを指す。ロール名 context は、状態を表すオブジェクトから、利用者オブジェクトを参照するためのポインターの役割をする。

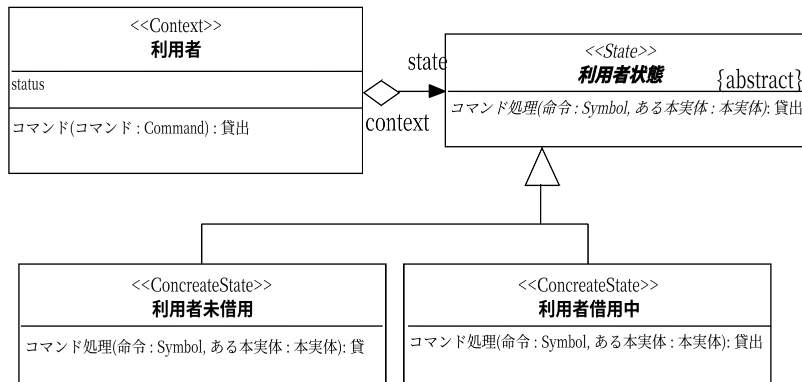

図 72 利用者と状態 (State) パターン

各クラスの操作は以下のようになる。

まず、利用者クラスは下記のように、status 属性に状態のみ格納しているが、状態遷移処理を利用者状態クラスのサブクラスにほとんど任せてしまう。

仕様 29 **利用者クラスのコマンド操作の仕様**

(1) **コマンド** (コマンド : Command) : **貸出**

```
( 命令 , 本実体 ) := コマンド解析 ( コマンド );
if  命令 = # 借用  or  命令 = # 返却   then
    self. エラー処理 (’ 存在しない命令を受けた。’)
self.state. コマンド処理 ( 命令 , 本実体 )
```

利用者状態クラスは抽象クラスであり、コマンド処理操作のインターフェースを定義するだけで、実際の操作の定義は具象クラスであるサブクラス群に任せている。

仕様 30 利用者状態クラスの操作仕様

(1) **コマンド処理 (命令** : String, **ある本実体** : **本実体)** : **貸出**

```
-- 抽象操作なので、サブクラスに実装を任せる。
self.subclassResponsibility
```

利用者未借用クラスは「未借用」状態の処理に責任を持つクラスである。以下の実装で、このクラスは次に遷移すべき「借用中」状態を知っているようにした。すなわち、利用者オブジェクトの statasu 属性を利用者借用中オブジェクトに書き換えることによって、次の状態に遷移するようにしている。

仕様 31 利用者未借用クラスの操作仕様

(1) **コマンド処理 (命令** : String, **ある本実体** : **本実体)** : **貸出**

```
if 命令 = # 借用 then
    result := self. 借用 ( ある本実体 )
    -- 利用者の状態を利用者借用中に遷移させる。
    -- self.context.state は利用者の状態を表す。
    -- 利用者借用中 .new は「利用者借用中」クラスの
    -- インスタンスを生成する
    self.context.state := 利用者借用中 .new
elsif 命令 = # 返却 then
    self. エラー処理 (' 借りていない本を返そうとしている。')
end
```

上のコマンド処理操作の中では、命令による分岐が発生しているが、プログラミング言語によってはこの分岐をなくすことができる[108]。

利用者借用中クラスは「借用中」状態の処理をするオブジェクトを定義したクラスである。図書館の最大貸出数を越えようとしたとき、貸出限度超過操作へ行く (goto) するようにしている。

仕様 32 利用者借用中の操作仕様

(1) **コマンド処理** (**命令** : Symbol, **ある本実体** : **本実体**) : **貸出**

```
if 命令 = # 借用 then
    result := self. 借用 ( ある本実体 )
elsif 命令 = # 返却 then
    result := self. 返却 ( ある本実体 );
    if self. 借用数 = 0 then
        -- 利用者の状態を利用者未借用に遷移させる。
        self.context.state := 利用者未借用 .new
    end
end
```

(2) 有限状態マシンを直接実装する

状態遷移図をパラメータとして受け取り実行する、いわば「状態マシン・エンジン」を実装する方法である。(1) の方法に比べ、実装はパラメータを与えるだけであり、修正はパラメータの変更だけなので、再利用性・保守性共に優れている。また、動作の実体を定義しないうちから、状態遷移図の基本フローを確認できるため、プロトタイピングにも向いている。

108.例えば Smalltalk では、String として受け取った操作名を実行できるので、パラメータとして受け取った「命令」を使って、

result := self. 命令 (ある本実体)

というようなプログラムを書くことができるので、条件分岐が不要になる。もちろん、この場合、返却の場合のエラー処理は、返却操作内に移動させる必要がある。

ただし、「状態マシン・エンジン」があらかじめ定義されていない場合、自分たちで作らなければならないので、「状態マシン・エンジン」の最初の実装のみ、(1) の方法よりやや手間が掛かる。

「状態マシン・エンジン」は通常インタープリターであり、通訳(Interpreter) パターンが使える。すなわち、以下のような文法[109] の言語をインタープリターとして実行するプログラムを作ればよい。

状態遷移 → 前状態　遷移　後状態

遷移 → イベント名　引数リスト　'['　ガード条件　']'　'/'　動作式

...

例えば、利用者オブジェクトの場合だと、

{ 未借用 , 借用 (ある本実体) / 借用数加算 , 借用中 }

{ 借用中 , [借用数 = 0], 未借用 }

{ 借用中 , 返却 (ある本実体) / 借用数減算 , 借用中 }

{ 借用中 , 借用 (ある本実体) / 借用数加算 , 借用中 }

{ 借用中 , [借用数 > self. 図書館 . 最大貸出数] / 貸出限度超過 , 借用中 }

というようなパラメータを受け付けて、実行するプログラムを作る必要がある。

(3) 並行タスクを利用する

この方法は、オブジェクトが本来持っている並行性をうまく表せる。OS や言語のランタイム機構を使って、イベントをプロセス間呼出として使う。各プロセス内では、(1) の方法と同じく、プログラム内の制御位置で状態を表す。

4.4.6 関連の実装方法決定

ほとんどのオブジェクト指向言語では「関連」という概念は実装されていない。従って、クラス図上の「関連」をオブジェクト指向言語が実装している型ないしはオブジェクトで実現しなければならない。

109.BNF 記法による記述。

(1) ポインターによる実装

関連の 1 方向分を、ポインター型の属性を持つことで実装する方法である。相手方の多重度が 1 ならば単純なポインターで実装できるし、多重度が多ならばポインターの集合で実装できる。多の側に {ordered} の制約が付いているなら、線形リスト構造または木構造を用いる。限定付関連は辞書 (Dictionary) パターンを使って実装できる[110]。

ポインターによる実装は単純だが、関連が結ぶクラスにポインター型の属性を付加しなければならないため、ソースプログラムが公開されていないクラス・ライブラリー中にあるクラスに適用することができない。また、疎な関連[111] の場合、実際に存在しないリンクに対応する、空のポインターを多数持つことになり、空間効率が悪くなる。

(2) 関連自体をオブジェクトする実装

関連自体を辞書クラスなどのオブジェクトとして実装する方法である。関連するオブジェクトの組を可変長オブジェクトに格納する。限定付関連の場合は、限定子とそれに関連する 2 つのオブジェクトの「3 つ組」を可変長オブジェクトに格納する。

関連オブジェクトによる実装は、ソースプログラムが公開されていないクラス・ライブラリー中にあるクラスに適用でき、疎な関連にも適用できる利点がある。

ポインターによる実装に比べて、若干効率が悪くなる恐れがあるが、ハッシュ表または木構造を採用することで効率の低下を最小限に留めることができる。

4.4.7 実装図の作成

システムの設計が終わると、プログラムの構造を表す構成図と、実行時の構造を表す配置図を作成する。

構成図は、ソースコートや実行モジュールの構造を示す図であり、最近の開発環境では同等の構造を容易に作成・管理できるので、ここでは説明を省略する。

110.辞書 (Dictionary) パターンの物理的構造は、この場合、木構造かハッシュ表を使うのが普通であるが、通常のオブジェクト指向言語では「辞書クラス」としてすでに定義されていることが多い。

111.関連するクラスの大半のオブジェクトには関係ない関連のこと。

配置図は、実行時のオブジェクトやプロセスやその他のコンポーネントの配置状況を記述するためのものである。以下に、貸出と検索についての配置図を示す。

ここでは、司書クライアントの Mac に貸出処理画面と検索画面という 2 つのアプリケーション[112]があり、図書館サーバーの UNIX の利用者クラスと著作物クラスに依存していることを示している。

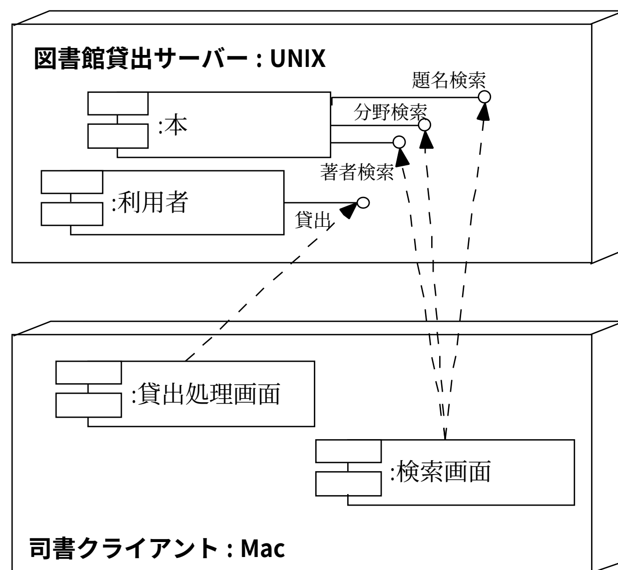

図 73 配置図の例

上の図で、貸出・題名検索などの名前が付いた「○の付いた線」は、実行時のコンポーネントが提供するサービスを示し、破線の矢印は「依存関係」を示す。例えば、検索画面は著作物オブジェクトの題名検索・分野検索・著者検索の 3 つのサービスに依存している。

112.最近では各種の GUI 作成ツールなどがあり、クライアント側のアプリケーションの設計は本質的に難しい問題ではないので、この本ではあまり触れない。サーバー側の「モデル」が大事なのだという立場である。もっとも、多くのプロジェクトでは、クライアント側のアプリケーションの画面設計だけを行って「設計が終わった」として、多くのトラブルを招いているのが実状である。

4.4.8 再利用性・保守性分析

ここまでで、一応の設計が終わったことにして、次に必要なのが再利用性と保守性の分析である。このためのチェック項目を以下に示す。

- **汎用性のある抽象クラスを作る**
- **多相を実現する**
- **パラメータ化クラスを作る**
- **勘所（Hotspot）を発見する**
- **小さなモジュールを作る**
- **強結合のモジュールを作る**
- **少なく小さいインタフェースで実現する**

以下に、概要を述べる。

(1) 汎用性のある抽象クラスを作る

汎用性のある抽象クラスが、再利用に役立つことが分かっている。逆に言えば、設計モデルにこのようなクラスがなければ、まだ検討が足りないことを示している。

このシステムでは、分析パターンなどの手助けもあり、責任・責任タイプ・当事者・人・組織・利用者・著作物・HashTableといった、汎用性の高い抽象クラスを抽出できた。

(2) 多相を実現する

借用や返却に関連する操作名は、各クラスで共通化できている。また、モデル中英語で示した操作名（remove, hashなど）は、既存のクラス・ライブラリーとの共通化を意識している。ただし、多相の実現は実装言語とクラスライブラリーとの関係が深いので、ここで最終結論は出ない。

(3) パラメータ化クラスを作る

ハッシュ表や 2 分木は、任意の型・クラスに対して定義したのでパラメータ化クラスに対応している。

(4) 勘所（Hotspot）を発見する

一番変更が予想される貸出規則の変更は、図書館クラスの中に定義して、他のクラスにはほとんど影響が出ないようにした。

また、次に変更が予想される、利用者の種類や司書との関係の変更は、責任 (Accountability) パターンを使って、責任クラスや責任タイプ・クラスのインスタンスの変更で済ませることができる。

さらに、図書館で管理する対象に、本以外の CD やビデオが追加されても、著作物のサブクラスを作れば、変更余波は少ない。

(5) 小さなモジュールを作る

各操作は数行から、多くても 10 行ほどであり、十分小さい。また、各クラスも操作数は高々 10 個ほどであり、十分小さい。

今までに出てきたクラス図では、UML の約束に従って、各属性に値を設定したり、各属性の値を参照する操作は省略していた。そういう操作を入れても、操作数は高々 20 個ほどであり、従って、クラス全体もせいぜい 200 行ほど[113] で十分に小さい。

もちろん、実装言語によってはもう少し行数が増えるかもしれないが、それでも大きなモジュールにはならないだろう。

(6) 強結合のモジュールを作る

各クラスの責任がはっきりしていて、各操作も単一の機能を実現しているので結合性は高い。

113.注釈は除いた行数である。

(7) 少なく小さいインタフェースで実現する

各操作は、数個のパラメータをやり取りしているだけで、かつ、各パラメータは単純なデータ型か小さなオブジェクトである。また、関係が深いクラス間ではメッセージのやり取りがあるが、関係が浅いクラス間でのメッセージのやり取りはない。従って、少なく小さいインタフェースが実現できていて、変更を局所化できるし再利用性も高い。

4.4.9 設計モデルの検証

設計モデルは、ドメインモデルや分析モデルと同じやり方で検証していく。設計が終わった時点では、すべての操作は実装されていて、しかも手続き的仕様になっているはずなので、より精密な検証ができるはずである。

ただし、この時点ではすべての場合を尽くすようなテストケースは事実上作成できない。テストケースの「組合せの爆発」が起き、何百万通りのテストケースを作っても、システムのすべてのパス（Path）の組合せをチェックすることは不可能だからだ。実用的なシステムでは、分岐の 85%[114] とすべての命令をカバー[115] するようなテストケース作成が目標となる。このような効率の良いテストケースの作り方やテスト方法の詳細は本書の対象外なので、参考文献を当たって欲しい。[116]

114.C_1 カバレージ 85% という言い方をする。

115.C_0 カバレージ 100% と言う。

116.テストについては『宮本勲著、ソフトウェア・エンジニアリング：現状と展望、産学社、1982 年』に網羅的な解説がある。『G.J. Myers 著、松尾正信訳、ソフトウェア・テストの技法、1980 年、近代科学社』も参考になる。