

オブジェクト指向技術入門

2024 年 7 月更新

はじめに

オブジェクト指向技術にはいくつかの方法論があり、そこで使われている用語や概念は多少異なる場合があります。特定の手法について学習される場合には、その手法の書籍も併せてご使用下さい。

目次

| | |
|-------------------------|----|
| 1. オブジェクト指向技術とその目的 | 4 |
| 1.1 ソフトウェア開発 | 4 |
| 1.2 ソフトウェア開発における位置付け | 4 |
| 1.3 アーキテクチャとは | 4 |
| 1.4 手法とは | 5 |
| 1.5 プロセスとは | 5 |
| 1.6 ツールとは | 5 |
| 1.7 アプローチとは | 5 |
| 1.8 パラダイムシフトとは | 6 |
| 1.9 オブジェクト指向技術の出現 | 6 |
| 1.10 方法論の出現 | 6 |
| 1.11 オブジェクト指向技術に期待できるもの | 6 |
| 2. オブジェクト指向技術の主要な概念 | 7 |
| 2.1 オブジェクト | 7 |
| 2.2 オブジェクトを使ってモデル化する | 8 |
| 2.3 クラス | 8 |
| 2.4 オブジェクトの追跡可能性 | 10 |
| 2.5 オブジェクトのアイデンティティ | 10 |
| 2.6 オブジェクトを使う利点 | 10 |
| 2.7 オブジェクトを発見する | 11 |
| 2.8 カプセル化 | 12 |
| 2.9 属性 | 13 |
| 2.10 オブジェクト、クラス、インスタンス | 14 |

| | |
|--|-----------|
| 2.11 オブジェクトとインスタンス | 15 |
| 2.12 オブジェクトの候補 | 15 |
| 2.13 クラス図 | 16 |
| 2.14 クラス図と観点 (Martin Fowler, 1997) | 22 |
| 2.15 仕様の観点 | 22 |
| 2.16 関連 (インスタンス間の関係) | 22 |
| 2.17 集約とコンポジション | 25 |
| 2.18 制約 | 25 |
| 2.19 多相性 (ポリモーフィズム) | 26 |
| 2.20 関連に関するその他の概念 | 30 |
| 3. オブジェクト指向システム開発 | 32 |
| 3.1 従来のシステム開発 | 32 |
| 3.2 オブジェクト指向開発の特徴 | 32 |
| 3.3 オブジェクト指向開発の流れ | 33 |
| 3.4 開発プロセスと作成するモデル | 35 |
| 3.5 モデル | 35 |
| 3.6 分析プロセス | 35 |
| 3.7 構築プロセス (設計モデルの作成) | 39 |
| 3.8 構築プロセス (実装モデルの作成) | 39 |
| 3.9 テストプロセス (テストモデルの作成) | 40 |
| 3.10 ユースケースモデル | 40 |
| 4. 統一モデリング言語 UML | 42 |
| 4.1 UMLの概要 | 43 |
| 4.2 ユースケース | 43 |
| 4.3 クラス図 | 45 |
| 4.4 相互作用図 | 48 |
| 4.5 パッケージ図 | 49 |
| 4.6 振舞い図 | 50 |
| 4.7 アクティビティ図 | 51 |
| 4.8 ステレオタイプ | 53 |

| | |
|--|-----------|
| 4.9 OCL (Object Constraint Language) | 55 |
| 4.10 モデリングツール | 56 |
| 5. オブジェクト指向プログラミング言語 Java | 57 |
| 5.1 オブジェクト指向プログラミング言語 | 57 |
| 5.2 Java の歴史 | 57 |
| 5.3 Java の特徴 | 57 |
| 5.4 Java を始める | 59 |
| 5.5 基本型と参照型 | 59 |
| 5.6 変数宣言 | 60 |
| 5.7 演算子 | 61 |
| 5.8 制御フロー文 | 62 |
| 5.9 列挙型 <i>enum</i> | 64 |
| 5.10 匿名クラス | 64 |
| 5.11 <i>this</i> と <i>super</i> | 64 |
| 5.12 ボクシングとアンボクシング | 65 |
| 5.13 <i>javadoc</i> | 66 |
| 5.14 ジェネリクス (総称型) | 66 |
| 5.15 コレクション | 67 |
| 5.16 ストリーム | 67 |
| 5.17 ラムダ式 | 68 |
| 5.18 オブジェクト | 68 |
| 5.19 クラス | 70 |
| 5.20 クラスの拡張 | 74 |
| 5.21 <i>final</i> 宣言 | 84 |
| 5.22 <i>Object</i> クラス | 85 |
| 5.23 ラップクラス (<i>Wrapper Class</i>) | 85 |
| 5.24 <i>Class</i> クラス | 85 |
| 5.25 抽象クラスと <i>abstract</i> メソッド | 86 |
| 5.26 インタフェース | 87 |
| 5.27 例外クラス (<i>Exception</i>) | 89 |
| 5.28 スレッド | 90 |

5.29 パッケージ 91

6. オブジェクト指向技術の導入..... 92

7. 永続化戦略..... 92

1. オブジェクト指向技術とその目的

本章では、ソフトウェア開発技術におけるオブジェクト指向技術の位置付けと、その目的について説明します。

1.1 ソフトウェア開発

ソフトウェアには、Windows などの OS、Java コンパイラなどの言語処理系、Spring などのフレームワーク、データベース管理システム、自動車の自動運転などの制御系システム、家電製品に搭載される組み込み系、企業で使われる基幹系業務システムなどがあります。それらを開発するためには、設計という作業が行われます。

開発プロセスがウォーターフォール型であれアジャイル型であれ必要な作業です。

設計にもいろいろな手法があります。企業内のチームや個人が築き上げた独自手法もあります。オブジェクト指向は手法の 1 つにです。適用される領域は、所謂ソフトウェアだけに限らず、ビジネスモデリングなどにも適用されています。

1.2 ソフトウェア開発における位置付け

例

| | | |
|---------|------------------|----------------|
| アーキテクチャ | オブジェクト指向 | 機能/データ分離指向 |
| 手法 | OOD/OOA、OMT、OOSE | SADT、RDD、SA/SD |

1.3 アーキテクチャとは

オブジェクト指向とは、ソフトウェア開発の中のアーキテクチャに位置付けられます。

(例)

- 機能/データ分離指向アーキテクチャ
- オブジェクト指向アーキテクチャ

アーキテクチャとは、もともとは建築学における設計技術や建築様式のことです。コンピュータの世界では、設計思想などを意味します。システム・アーキテクチャ、アプリケーション・アーキテクチャ、ハードウェア・アーキテクチャ、CPU アーキテクチャなど。

(例)

- Von Neumann アーキテクチャ

(オブジェクト指向アーキテクチャとは異なる)「機能とデータを分離するアーキテクチャ」は、Von Neumann アーキテクチャに起源しています。機能とデータを分離する構造は、ハードウェアの構造（メモリ、中央制御装置、演算ユニット、入力、出力の 5 要素構成）にうまく適合した設計思想です。

機械語以来、高レベルの言語においても「プログラムとは、メモリ内にあるデータを操作するための制御文で構成され、目的の結果を得るもの」という考え方があります。この考え方が、上流工程の分析、設計においても、機能中心法として使われてきました。

1.4 手法とは

あるアーキテクチャによるソフトウェアの開発手続きや手順です。様々な手法が提唱されています。

1.4.1 オブジェクト指向アーキテクチャによる手法の例

- ・ OOSE：オブジェクト指向ソフトウェア工学 OOSE(1987) Ivar Jacobson 他
- ・ OOSA:オブジェクト指向システム分析（1988）Shlaer, Mellor
- ・ OOA/OOD：オブジェクト指向分析(1990) Peter Coad, Ed Yourdon
- ・ Booch 法：オブジェクト指向分析と設計(1990) Grady Booch
- ・ OMT：オブジェクトモデル化技法(1991) Rumbaugh 他

1.4.2 機能/データ分離指向アーキテクチャによる手法の例

- ・ SADT：Structured Analysis and Design Technique；構造化分析/設計技法(1985) Ross
- ・ RDD：Requirement Driven Design based on SREM；SREM を使った要求駆動型設計(1985) Alford
- ・ SA/SD：Structured Analysis and Structure Design；構造化分析/構造化設計(1979) Yordon 他

1.5 プロセスとは

手法は机上レベルの理論。プロセスとは、手法を実際のシステム開発の現場でも適用できるレベルにまで具体化したものです。プロセスの目的は、誰が行っても同じ品質、同じ結果が得られることです。

5 人規模のプロジェクトに適用するプロセスと、100 人規模に適用するプロセスは同じとは限りません。会社毎、部署毎、プロジェクト毎に使用する開発標準などがこれに相当します。

1.6 ツールとは

各アーキテクチャに基づいた（あるいは適した）開発環境やデータベース、プログラミング言語などがあり、使用するアーキテクチャに合わせて選択する必要があります。

1.7 アプローチとは

アプローチという言葉もアーキテクチャ、方法論、手法といった意味で使われます。よく取り上げられるものとして次の 3 つがあります。

1.7.1 プロセス中心アプローチ (POA: Process Oriented Approach)

まず機能に着目し、システム全体を処理（プロセス）の集まりとしてモデル化します。データ形式は、各処理に合わせる形で決定します。

1.7.2 データ中心アプローチ (DOA: Data Oriented Approach)

要求される機能はシステム毎や時間の経過とともに変化します。それに対して、基盤となるデータは安定しています。この点に着目したアプローチが DOA です。最初に基盤となるデータ構造を決め、各処理は決められたデータ構造を前提に設計します。

1.7.3 オブジェクト中心アプローチ (OOA: Object Oriented Approach)

オブジェクトとそれらの相互作用としてシステムをモデル化します。

DOA と OOA については、オブジェクト指向技術の広がり、特に業務システム分野での適用事例の増加とともに、様々な議論があります。例えば、DOA と OOA を対立させる考え方、あるいは DOA を OOA の一部ととらえる考え方などがあります。

1.8 パラダイムシフトとは

従来型の開発技法に習熟した技術者がオブジェクト指向技術を習得しようとする場合、ソフトウェア開発の広い範囲において、それまでの考え方などを切り替えなければならないポイントがあります。この切り替えをパラダイムシフトと呼びます。スムーズにパラダイムシフトできる場合もありますが、時間がかかるケースの方が多いようです。しかし、これを避けてはオブジェクト指向技術を活用することはできません。

例えば、「銀行口座」という言葉から、「銀行口座テーブル」を思い浮かべるのか、あるいは「銀行口座オブジェクト」を思い浮かべるのか、この切り替えが必要です。

1.9 オブジェクト指向技術の出現

最初はプログラミング言語を中心に発展してきました。(1980年代)

- Smalltalk XEROX PARC 研究所
- C++ AT&T ベル研究所
- Eiffel Bertrand Mayer
- CLOS Common Lisp Object System, X3J13 ANSI Lisp 標準化 G

1.10 方法論の出現

1990年代に入り、方法論（手法の研究）が提唱され、一般のシステム開発（ビジネス・アプリケーションの開発など）にも適用しやすくなりました。

1.11 オブジェクト指向技術に期待できるもの

開発方法の工業化（部品、コンポーネントの利用）

自動車分野では新型車の場合でも部品の再利用率は80%程度と言われています。再利用がなければ、高い品質や信頼性を実現・維持することは難しいでしょう。

このような仕組みは、ソフトウェア開発においても必要であり、次のような効果をもたらします。

- 部品単位での信頼性の確保
- システム全体としての品質向上
- 短期開発
- コスト削減
- 属人性の低減、

全体を部品（オブジェクト）で構成するというアプローチはオブジェクト指向と一致します。オブジェクト指向技術によ

ってソフトウェア産業においても自動車産業のような工業化が進むでしょう。

当然、再利用されるような部品（オブジェクト）を作るためには、そのオブジェクトが属する世界（ドメイン）に関する深い知識やオブジェクト設計スキルが必要です。優れたオブジェクトは誰からも理解されやすく、使いやすいものです。

1.11.1 パターン（設計、実装の定石）の利用

ソフトウェア開発に要するコストの大部分は、多くのプロジェクトや開発者が同じような間違いを繰返すために生じるコストだと言われています。

パターンを利用すれば、同じような試行を繰返す必要はなくなり、（間違うことも大事な経験ですが）その分のコストを削減できます。

広く知られたパターンは高度に洗練されており、信頼できます。優れたパターンを取り入れることで開発期間が短縮され、品質に関しても安心できます。これは、そのパターンの範囲だけではなく、その周辺にも良い効果が得られると思います。

また、開発者のスキルアップも最短距離で行えます。

パターンには、ある企業内部や個人で使用されているものもありますが、オープンソース・ソフトと同じように公開されているものも数多くあります。

Java や C# が普及した現在、パターンの利用も広まっています。

パターンによる信頼性の向上や開発コストの削減効果は、オブジェクト指向技術がもたらした大きな恩恵の 1 つです。

- ・ 分析パターン（Analysis Patterns: Reusable Object Models, 1997, Martin Fowler）
- ・ 設計パターン（Design Patterns: Elements of Reusable Object-Oriented Software, 1995, Erich Gamma 他）

2. オブジェクト指向技術の主要な概念

本章では、オブジェクト指向分析、設計、プログラミングで用いられる主要な概念について説明します。これらの概念に対する理解は、実戦を重ねる中で深めていく必要があります。しかし、基本的な知識もなく実戦を重ねても効果はありません。ここに挙げた概念は、モデリングをする時だけではなく、実装（プログラミング）する際にも必要な知識です。

2.1 オブジェクト

オブジェクトの代表格は、現実の問題領域（アプリケーション・ドメインやシステム化対象領域）に存在する「モノ」です。また、私たちが自然に「モノ」として考える対象物以外にも、たくさんのオブジェクトがあります。これらを見分ける技術は、オブジェクトを抽出するという経験を重ねることで向上します。また、パターンを学習したり、他の人が作った優れたオブジェクトモデル（オブジェクトの集まり）を知ることで、より早く正しい方向で習得できます。自己流だけや、間違った経験を重ねるよりも、はるかに効率的です。

抽出したオブジェクトは、現実の世界よりも、はるかに多くのリスポンシビリティ（responsibilities 責務）を持つようになります。例えば、1 冊の書籍（オブジェクト）に、その書籍名を問うと、その書籍は自分の書籍名を応答する（責務を持つ）ようになります。この辺りは、現実の世界と全く同じではありませんが、オブジェクト指向システムの中では、それぞれのオブジェクトは、まるで人間であるかのように振舞います。

（例）

銀行の預金口座を管理するシステムを構築するとします。その問題領域（ドメイン、システム化対象領域）には、A さんの普通預金口座や B さんの普通預金口座、C さんの定期預金口座などのオブジェクトがあります。各オブジェクトは、残

高など自分の状態（情報）を知っています。A さんが自分の普通預金口座から預金を引き出すと、A さんの普通預金口座の状態（残高）が変化します。

（例）

社内の書籍を管理するシステムを構築するとします。それぞれの書籍は、いくつかある書棚の 1 つに収納されています。ある書棚に問い合わせると、その書棚に収納されている全ての書籍のリストを応答します。

2.2 オブジェクトを使ってモデル化する

システムは、オブジェクトの集りとしてモデル化します。各オブジェクトが、自分に割り当てられた責務を果たすことで、全体として、システム要件を満たしていきます。システムは、あるオブジェクトが他のオブジェクトにメッセージを送り始めたときに始動します。「メッセージを送る」とは、他のオブジェクトに対して、ある責務の遂行（仕事）を依頼することです。Java プログラミング的に言えば、あるメソッドを使用することです。（従来型のプログラミング的には、ある関数を呼び出すことです）

（オブジェクトの例）

- ・ 預金者オブジェクト
- ・ 普通預金口座オブジェクト
- ・ 定期預金口座オブジェクト
- ・ 普通預金口座番号オブジェクト
- ・ 支店オブジェクト
- ・ 預入れ取引オブジェクト
- ・ 引き出し取引オブジェクト

2.3 クラス

オブジェクトを使ってモデル化したシステムを、コンピュータ上で動かすためには、各オブジェクトをソフトウェアで実現する必要があります。

Java はクラスベースという OOP スタイルを導入します。

クラスベースとは、クラスをもとにインスタンス（オブジェクト）を生成するスタイルです。

例えば、A さんの普通預金口座、B さんの普通預金口座、2 つのオブジェクトがあります。この 2 つのオブジェクトは別々のものです。しかし、共通する性質（属性）があります。例えば、ともに残高という状態をもつこと、ともに預金者という状態をもつこと等です。もちろん、残高の金額そのものや、預金者は異なります。この共通した性質を定義したものが、クラスです。そして、このクラス（仕様や雛型とも呼ばれます）をもとに、コンピュータ上で、ソフトウェア的に生成されたものがインスタンスです。このインスタンスが現実のオブジェクトに対応します。

下図は、クラスとオブジェクトです。左側が UML クラス図、右側の 2 つは UML オブジェクト図です。

| 普通預金口座 |
|-----------|
| -普通預金口座番号 |
| -預金者 |
| -残高 |
| +預け入れる() |
| +引き出す() |
| +解約する() |

| Aさんの普通預金口座：普通預金口座 |
|-------------------|
| 普通預金口座番号 = 123456 |
| 預金者 = A |
| 残高 = 1000 |

| Bさんの普通預金口座：普通預金口座 |
|-------------------|
| 普通預金口座番号 = 234567 |
| 預金者 = B |
| 残高 = 2000 |

以下に、普通預金口座クラスの Java コードの例を示します。UML クラス図の記法や Java の文法の詳細については、ここでは触れません。ただ、2つの形がよく似ている点に着目してください。

```
package oop.chap2;

/**
 * 普通預金口座
 */
public class 普通預金口座 {
    private String 普通預金口座番号;
    private String 預金者;
    private long 残高;
    public void 預入れる(long 預入れ額) {
        //預入れ処理
    }
    public void 引き出す(long 引き出し額) {
        //引き出し処理
    }
    public void 解約する() {
        //解約処理
    }
}
```

次は、インスタンスを生成するための Java のコード例です。

```
普通預金口座 kouza = new 普通預金口座();
```

必要な性質（属性）はどれか？それは問題領域によって異なります。預金口座を管理するというシステムでは、各預金者が持っている通帳の色やデザインは無視して良いかもしれません。もし、完全なオブジェクトをソフトウェアで容易に実現できるのであれば迷う必要はありません。すべての性質を持ったクラスを定義すればよいのです。しかし、それは現実的ではありません。通常は、必要な性質を識別し、クラスを定義することになります。

【補足】

クラスベース（Class-Based）はクラスをもとにオブジェクト（インスタンス）生成します（Java や PHP）。

プロトタイプベースは既存のオブジェクトから新しいオブジェクトを生成します（JavaScript）。ただ、JavaScript でも class 構文を使うとクラスベースなプログラミングができます。

2.4 オブジェクトの追跡可能性

モデルとは、問題領域（システム化対象領域、ドメイン）やソフトウェアのある側面を表したものです。モデルには、分析モデルや設計モデル、実装モデル（ソースコード）などがあります。開発作業が進むとともにモデルは詳細になり、含まれるオブジェクトの数も増えていきます。

オブジェクト指向開発では、オブジェクトは追跡可能です。追跡可能とは、初期(前工程)のモデルにあったオブジェクトは、詳細化された次工程のモデルの中でも容易に識別できるという特徴です。つまり、分析モデルにあったオブジェクト（から定義されたクラス）は、ソースコードの中で容易に識別できます。例えば、普通預金口座オブジェクトは、分析モデルの中でも、実装モデル（ソースコード）の中でも、普通預金口座クラスとして、そのままの名前で存在します。

追跡可能性は、次のようなケースで有用です。例えば、ユーザーからの変更要求があったとします。その時、その変更内容を説明する文書や言葉の中には、オブジェクトの名前が使われているはずです。そして、それらの用語のほとんどが、オブジェクトとして抽出されてるはずです。また、変更内容は、あるオブジェクトのある振舞い（責務の遂行の仕方）に対する修正であることが多いでしょう。そのような場合、変更対象であるオブジェクトを、ソースコードの中で容易に識別できるのです。

もしも、追跡可能でない場合、ユーザーからの変更要求をソースコードに反映するためには、要求内容を翻訳する必要があります。つまり、要求内容にある用語ばどを、機能やデータの構造に置き換えて考え直さなければなりません。（参照 後述の「現実とモデルの意味的乖離がなくなる」）

2.5 オブジェクトのアイデンティティ

全てのオブジェクトは一意に識別されるアイデンティティ（識別子）を持っています。リレーショナル・データベース・モデルにおける（設計者が指定する）主キーのような概念とは別に、一意に識別できるアイデンティティがあります。オブジェクトはそれぞれ固有の存在として区別されます。オブジェクトの記述的な性質によって区別されるわけではありません。例えば、Smalltalk では2つの整数オブジェクトが同じ値であっても、それらは別のオブジェクトとして扱われます（同値と同一の違い）。Java の場合、基本タイプについては、同値と同一が同じに扱われますが、例えば、int ではなく Integer を使用すれば同値と同一は区別されます。

（例）

普通預金口座クラスのインスタンスである A さんの普通預金口座インスタンスと、B さんの普通預金口座インスタンスは、普通預金口座番号ような情報がなくても一意に識別できるアイデンティティが、それぞれのインスタンスに対して、Java プログラムの実行環境によって与えられます。

2.6 オブジェクトを使う利点

2.6.1 開発者、利用者共に理解しやすい

オブジェクトは実世界の“もの”に基づいています。そのため、設計されたモデルを元の実世界に対比して理解することができます。従って、利用者、発注者、分析・設計者、プログラマ、保守担当者にとって理解しやすく、関係者間でのコミュニケーション・ミスも生じにくいという利点があります。

また、他の開発者が、すでに開発されたオブジェクト（クラス）を再利用する場合にも、それが何であるかを理解しやすくなります。例えば、GUI の世界では、コンポーネント（オブジェクト）の再利用は常識です。それは、そのコンポーネントが何であるかを、他の開発者が理解しやすいからだと言えます。また、理解しやすいオブジェクトモデルになって

いるからです。

- 例えば、預金口座管理ドメインにおいても、普通預金口座のようなオブジェクト（クラス）が流通し、それを再利用することが一般的になれば、ソフトウェア開発の姿も大きく変わるでしょう。

2.6.2 現実とモデルの意味的乖離がなくなる

機能指向やデータと機能をわけける方法で設計されたモデルは、（要件定義書を除けば）利用者や発注者がそれを理解する事は困難です。これは、現実のモデルと意味的に乖離しているためです。また、利用者や発注者から新しい要求があった場合には、開発者はその要求を“機能とデータを分離した構造”に変換して考える必要があります。例えば、新しい要求は、幾つかのテーブルとプログラム群に対する要求に変換しなければなりません。さらに、この変換は属人的な作業になる場合が多く、結果（要求の実現方法）も様々になる可能性があります。つまり、担当者によって実現方法が違う場合も少なくありません。上述した追跡可能なモデルでは、新しい要求を意味的に乖離した構造に置き換える必要がなく、実現方法（変更するオブジェクト、クラス）も、必然的に決まるようになります。

2.6.3 変更に強い

現実世界の“もの”は、システムの種類や要件に関わらず高い普遍性を持っています。つまり、システムの改版や、他システムで再利用する場合でも、適切にモデリングされたオブジェクトの責務や意味が大きく変わることはありません。また、変更が必要な場合にも、変更すべきオブジェクトは特定しやすく、正しくカプセル化（後述）されていれば変更箇所も局所化されます。

2.6.4 リファクタリングしやすくなる（継続的な洗練を可能にします）

データと機能を分離したシステムよりもリファクタリングが容易になります。ただし、それなりの設計や経験は必要です。

「良いモデル」と「すこし無理があるモデル」もあるでしょう。

例

Java 8 から日付・時刻の新しい API が追加されました。

Instant, LocalTime, LocalDate, LocalDateTime, ZonedDateTime などです。

これらは古い API である Date, Calendarなどを改善したものです。

例

SpringSecurity（ライブラリ、フレームワーク）

WebSecurityConfigurerAdapter が不要になったため、5.7 で非推奨化、6.0 で削除になりました。

このように優秀な技術者が作成したモデルであってもリファクタリングが必要になってきます。

2.7 オブジェクトを発見する

2.7.1 CRC カード手法（Class, Responsibilities, Collaborators）

オブジェクトを抽出するときの方法として、下図のような CRC カードを利用する方法があります。これは、Smalltalk の研究グループが、オブジェクト指向的“思考方法”を伝えるために考案したものです。オブジェクトは単なるデータの格納庫ではない、という事を忘れないように構成されています。

CRC カード（6 インチ×4 インチ）（約 15 cm×10 cm）



Class

クラス名を書きます。

Responsibilities

属性やメソッドではなく、このクラスのリスポンスビリティ（責務）を書きます。（データの格納庫ではありません）

Collaborators

協力者となるクラスを書きます。これによって、クラス同士の関係が浮かび上がります。

2.7.2 責務駆動設計 [Wirfs-Brock 他 1990]（参考）

アプリケーションをクラスとその責務、クラス間の協調によってモデル化する手法です。最初に、システム中のクラスやオブジェクトを識別し、次に、システムの責務を分析し、それらをシステム中のクラスに割り振ります。最後に、責務を満たすために必要なオブジェクト間の協調を、クラス間の協調として定義します。

このモデルを出発点に、クラス階層やサブシステムなどを設計します。

2.8 カプセル化

カプセル化とは、あるオブジェクトにアクセスする方法は、公開されたインタフェースに従ったメッセージの送信だけで、それ以外は、外部から完全に隠蔽することです。あるオブジェクト A は、別のオブジェクト B のインタフェースを、公開された責務・振舞いとして、安心して使用できます。また、オブジェクト B も確実にその責務を果たすことを保証しなければなりません。その代わり、オブジェクト A が、オブジェクト B の内情について干渉することはできません。適切にモデリングされたオブジェクトは、分かり易いクラス名と分かり易いインタフェースを持ち、自然とカプセル化されるはずです。

(例)普通預金口座クラスの公開されたインタフェースは、次の 3 つの責務・振舞いです。

預入れる

引き出す

解約する

| 普通預金口座 |
|-----------|
| -普通預金口座番号 |
| -預金者 |
| -残高 |
| +預け入れる() |
| +引き出す() |
| +解約する() |

```
package oop.chap2_2;

import java.math.BigDecimal;

public class 普通預金口座 {

    private static BigDecimal 利率 = new BigDecimal("0.001");
    private BigDecimal 残高; //ここではデータベースから復元した金額が設定されているものとします

    public void 預入れる(BigDecimal 預入れ額) {
        残高 = 残高.add(預入れ額); //BigDecimal 型の加算
    }
    public void 引き出す(BigDecimal 引き出し額) {
        残高 = 残高.subtract(引き出し額); //BigDecimal 型の減算
    }
    public void 解約する() {
        //解約処理
    }
} (注)「引き出す」の中の残高不足処理などは省略します。
```

普通預金口座の残高は隠蔽されており、2つのインタフェース「預入れる」、「引き出す」以外から変更されることはありません。

解約処理は省略しています。無視してください。

この例を、機能とデータを分離する場合と比較してみましょう。データはリレーショナル・データベースに格納されるものとします。その場合、普通預金口座テーブルの中に、Aさんの普通預金口座情報とBさんの普通預金口座情報が格納されることになるでしょう。普通預金口座テーブルの「残高」列の値を更新する機能（プログラム）の数は1つではないかもしれません。その場合、更新の仕方（ルール）は各プログラムに分散します。ルールが変われば、変更箇所は複数になります。そして、正しく更新するか否かは、各プログラムに依存してしまいます。

一方、カプセル化された「残高」は、そのオブジェクトの決まったインタフェースを使わなければ更新されることはありません。更新の仕方が分散することはありません。

2.9 属性

属性は、主に責務の遂行に必要な、ある状態を保持するために使用されます。例えば、普通預金口座オブジェクトは、その責務“引き出す”を遂行するために、属性“残高”を持っています。

補足：「プロパティ」と呼ばれるものと多くの文脈で同義です。C#.net では別の意味で使用されます。

2.9.1 インスタンス属性とクラス属性

属性には、インスタンス属性とクラス属性があります。インスタンス属性はインスタンス毎に固有の状態を保持します。クラス属性は、クラス毎に固有の状態を保持します。インスタンス属性がインスタンスの数だけ存在するのに対し、あるクラスのクラス属性は、そのクラスに1つだけ存在します。クラスに属性は、そのクラスの全てのインスタンスに共通な状態を保持します。

(例)普通預金口座クラスの各インスタンスは、それぞれ自分の残高を保持します。残高は、インスタンス毎に異なる状態だからです。一方、普通預金利率は、普通預金口座の全てのインスタンスに共通だとすると、この普通預金利率は、各インスタンスがそれぞれ保持するよりも、普通預金口座クラスに1つ保持されるのが自然です。この場合の普通預金利率は、クラス属性です。

2.9.2 属性、操作を UML で記述する

| 普通預金口座 |
|--|
| -普通預金口座番号 : String -預金者 : String -残高 : BigDecimal -利率 : <u>BigDecimal</u> |
| +預け入れる(in 預入れ額 : BigDecimal) : Boolean +引き出す(in 引出し額 : BigDecimal) : Boolean +解約する() : Boolean |

※先頭の + は public、- は private。

※下線は static

※UML に準拠する、しないというよりも使用するツールに従うことになります。本書は Visio2010 です。

※setter, getter メソッドは省略する場合があります。

2.10 オブジェクト、クラス、インスタンス

2.10.1 オブジェクトからクラスを定義する

クラスとは、似たような振る舞いと情報構造をもつオブジェクトを定義（抽象化）したものです。コンピュータ上でオブジェクト（インスタンス）を生成するためには、クラス（インスタンスの定義情報）が必要になります。

2.10.2 似たような…の基準

現実の世界に準じるべきです。例えば、A さんの普通預金口座オブジェクトと B さんの普通預金口座オブジェクトから普通預金口座クラスが定義できそうです。しかし、A さんの普通預金口座オブジェクトと B さんの所有する自動車オブジェクトからクラスを導出するケースは少ないかもしれません。

目的や基準を間違えると、分かり難いクラスが導出されてしまいます。このようなクラスは、生産性や品質を低下させることにもつながります。例えば、実装時に差分プログラミングの効果に着目するあまり、不自然なクラスや継承を追加してしまい、分析・設計時にはなかった分かり難いクラスが現れるため、オブジェクトの追跡が困難になり、変更しにくく、保守しにくいモデルになることがあります。

2.11 オブジェクトとインスタンス

ほぼ同義語ですが、使い分けの例としては、「オブジェクトは、実世界を分析して得たもの（例：個人顧客、法人顧客、顧客番号、…）で、インスタンスは、コンピュータ上のプログラムによってクラスから生成されたものです。（例：`Customer customer = new Customer("murayama","C0001");`）」という分け方があります。

全てのオブジェクトは、あるクラスに属する

- ・ インスタンスは、クラスから生成されるオブジェクトです。
- ・ 生成されたインスタンスは、そのクラスの振舞いと情報構造を持ちます。

2.11.1 クラスとタイプ(型)

- ・ 型はインタフェースだけを定義したものです。
- ・ クラスはインタフェースの実装を含むものです。
- ・ 型仕様と、型のために定義された実装を結合したものをクラスとします。(ODMG)
- ・ CORBA IDL はタイプを定義します。
- ・ Java では (abstract) class 定義と interface 定義があります。
- ・ Java のインタフェース (interface) は、public メソッドと定数のみを定義します。
- ・ Java の抽象クラス (abstract class) は、メソッドの一部を定義することもできます。
- ・ protected, static メソッドを持つことができます。

2.12 オブジェクトの候補

「オブジェクト指向システム分析」シュレィアー／メラーによる例（すこし古典的）

2.12.1 有形物

人、商品、伝票

2.12.2 役割

人や構成により演じられる役割

医師、患者、顧客、従業員

医師は患者にもなる

2.12.3 出来事

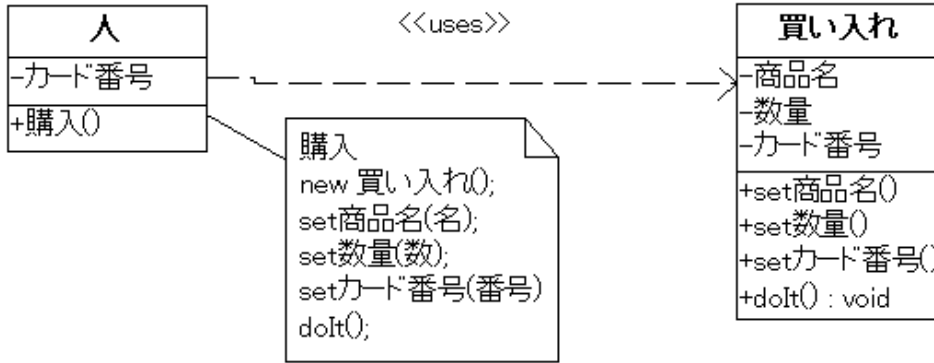
飛行、事故、故障、サービス要求

2.12.4 相互作用

買い入れ、結婚

2.12.5 仕様

製品などの仕様



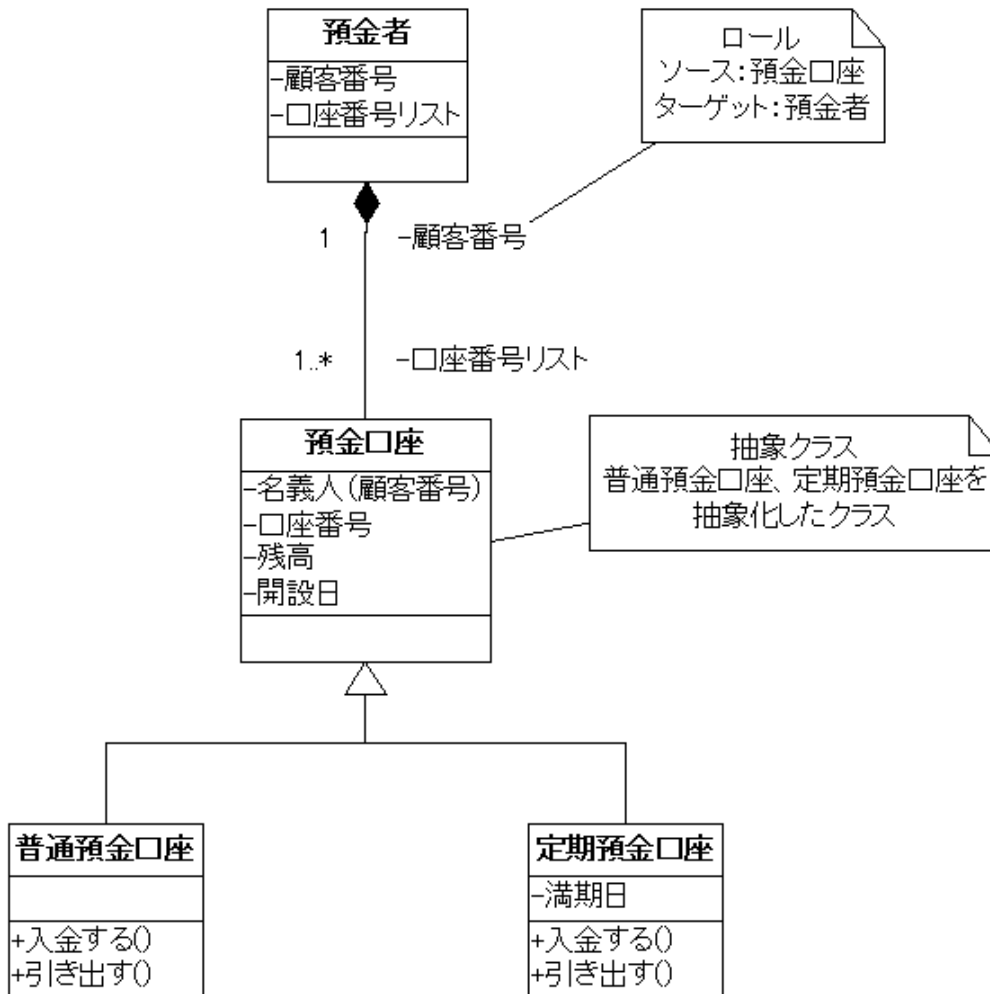
2.13 クラス図

オブジェクトタイプとそれらの間に存在する静的な関係を記述したものです。

静的な関係には

- ・ 関連
- ・ 継承（サブタイプはインターフェースの継承、サブクラスは実装の継承）

があります。



クラス図を書くときは、リスポンシビリティ指向ではなく、データ指向のクラスモデルにならないように注意します。

[注意] ER 図

2.13.1 汎化、一般化(クラス間の関係)

- ・ 汎化（一般化）はクラス間の関係を表わします。
- ・ いくつかのクラスに共通な性質を抜き出し、より一般的なクラスを作ることができます。
- ・ より一般的なクラスを継承関係の上位に置き、もとのクラスで継承し共有することができます。
- ・ 何が類似し、何が異なるかを簡潔に捉えることで、モデル化を容易にし分かりやすくできます。
- ・ 何が共通な性質であるかは、実世界の基準に基づかなければなりません。
- ・ モデルを分かりやすくすることが目的です。単なる共通部の抽出を目的とした継承や多階層の継承はモデルを分かりにくくし、劣化させます。

2.13.2 継承

- ・ あるクラス（スーパークラス）の責務を拡張するための仕組みです。
- ・ プログラミング言語でのコード再利用を示す。(差分プログラミング)

2.13.3 特化、特殊化

- ・ あるクラスの特性を継承し、必要な操作と情報構造を追加して、新しいクラスを作ることです。
- ・ 親のクラスの操作や情報構造を、再定義または削除することもできます。(振舞い互換でない)

ケース1：〈拡張〉新たに属性やメソッドを追加する。

クラスAが使われている場所で、クラスAの子孫が利用できるとき、振舞い互換という。

ケース2：継承したデータやメソッドを置き換える。(オーバーライドする)

ケース3：〈制限〉継承したインタフェースの一部を取り除く。(あまり使用しない。例：private にする等。)

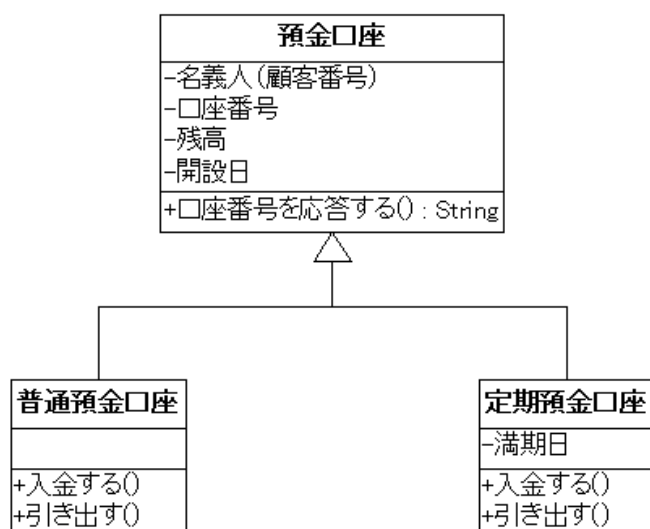
2.13.4 クラス図の例

名義人、口座番号、残高、開設日は普通預金口座と定期預金口座に共通の属性です。

「口座番号を応答する」は普通預金口座と定期預金口座に共通の操作です。

「入金する」と「引き出す」は共通の操作ではありません。

定期預金口座には属性として満期日が必要です。(普通預金口座にはありません)



2.13.5 Java ソースの例

```

/**
 * このサンプルはコンパイル確認までで、実行していません。
 */
package oop.chap2_3;

import java.math.BigDecimal;
import java.time.LocalDate;

/**
 * 預金口座クラス (抽象クラス)
 */
public abstract class 預金口座 {

    private String 名義人;
    private String 口座番号;
    private BigDecimal 残高;
    private LocalDate 開設日;

    //コンストラクタ
    public 預金口座(
        String 名義人,
        String 口座番号,
        BigDecimal 残高,
        LocalDate 開設日){
        this.名義人 = 名義人;
        this.口座番号 = 口座番号;
        this.残高 = 残高;
        this.開設日 = 開設日;
    }

    public String getAccountNumber() { //口座番号を応答する
        return 口座番号;
    }
}

package oop.chap2_3;

import java.math.BigDecimal;
import java.time.LocalDate;

public class 普通預金口座 extends 預金口座 {

    //コンストラクタ
    public 普通預金口座(
        String 名義人,
        String 口座番号,

```

```

        BigDecimal 残額,
        LocalDate 開設日){
    super(名義人, 口座番号, 残額, 開設日);
    //この例では預金口座クラスのコンストラクタを呼び出すだけ
}

public void deposit(int p_amount) {
    // 普通預金口座の場合の入金処理
}

public void withdraw(int p_amount) {
    // 普通預金口座の場合の引出し処理
}
}

package oop.chap2_3;

import java.math.BigDecimal;
import java.time.LocalDate;

/**
 * 定期預金口座クラス
 */
public class 定期預金口座 extends 預金口座 {
    private LocalDate 満期日; //定期預金だけの属性

    //コンストラクタ
    public 定期預金口座(
        String 名義人,
        String 口座番号,
        BigDecimal 残額,
        LocalDate 開設日,
        LocalDate 満期日){
        super(名義人, 口座番号, 残額, 開設日);
        this.満期日 = 満期日;
    }

    public void deposit(int p_amount) {
        // 定期預金口座の場合の入金処理
    }

    public void withdraw(int p_amount) {
        // 定期預金口座の場合の引出し処理
    }
}

```

2.13.6 クラスと具象クラス

- ・ 抽象クラスはインスタンスを持ちません。

- ・ 具象クラスはインスタンスを持つことができます。

2.13.7 スーパークラスとサブクラス

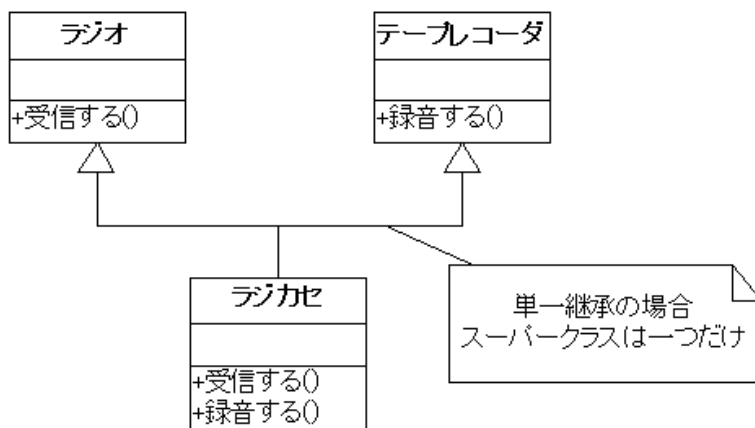
- ・ スーパークラスは特殊化のもとになるクラス、サブクラスは特殊化された結果のクラスです。
- ・ クラス階層をより分かりやすく、捉えやすくするためにクラス階層を作成することができます。
- ・ 再利用しやすくするためにクラス階層を作成することができます。
- ・ 差分プログラミングの効果を出すためにクラス階層を作成することができます。

2.13.8 継承を考えるときの注意点

- ・ 共通関数やデータの正規化が目的ではありません。(データ指向、プロセス指向で混乱しない)
- ・ 開発が進むにしたがって、新しいクラスが必要になることがあります。スパゲッティ継承に注意します。
- ・ 新しいクラスの追加は、クラス階層の変更を伴うこともあるので、最適なモデル、変更負荷、重要度等を検討し、結果として再構成することもあります。
- ・ 委譲を使うべきかを検討します。

2.13.9 多重継承

- ・ 2つ以上の既存クラスの性質を利用できます。
- ・ オブジェクトが演じる役割をモデル化する場合に適切です。
- ・ → オブジェクトの候補の役割オブジェクト
- ・ オブジェクト記述を再利用できます。
- ・ Java は単一継承のみです。
- ・ 多重継承では問題点も多く指摘されています。
- ・ (問題点) クラス階層が複雑で理解し難くなります。
- ・ (問題点) 2つの親クラスが同じ名前の操作を持つ場合など理解し難くなります。
- ・ (問題点) 反復継承は複雑になります。



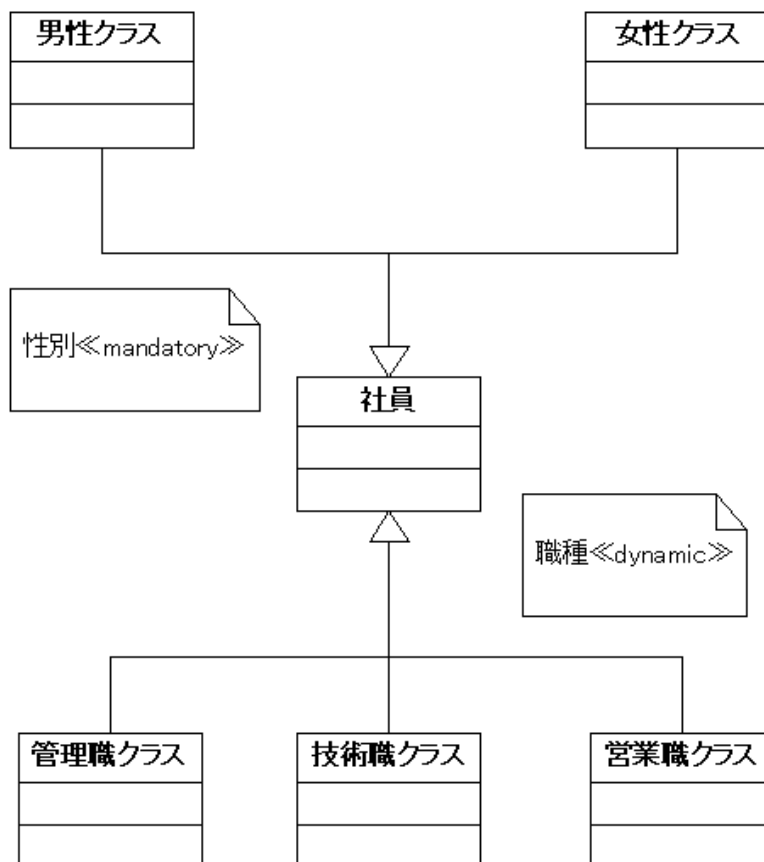
2.13.10 単一分類と多重分類

- ・ 分類（classification）とは、オブジェクトとそのタイプとの関係を示します。
- ・ ほとんどのオブジェクト指向プログラミング言語では、単一の静的分類です。
- ・ 単一分類では、オブジェクトはスーパータイプから継承する単一のタイプに属します。
- ・ 多重分類では、オブジェクトを、継承の関係を持つ必要のない複数のタイプで記述できます。
- ・ （多重継承とは、複数のスーパータイプを持つタイプを定義できることで、各オブジェクトはある単一のタイプに分類されます。）

2.13.11 静的分類と動的分類

動的分類では、オブジェクトはサブタイプ構造内の範囲で、属するタイプを変更できます。

静的分類では変更できません。（参考：アナリシスパターンのロールモデル）



2.13.12 Java のインタフェース

- ・ Java は単一継承のみ可能です。（多重継承はできません）
- ・ その代わりに、“実装は継承できませんが多重継承に準ずる”方法としてインタフェースがあります。（注：abstract クラス）

2.13.13 継承の使用に適した場合

- ・ サブクラスは「何々の特別な種類」であって、「何々によって果たされる役割」ではない場合。
- ・ オブジェクトはいったん分類されると、そのクラスのオブジェクトであり続ける場合。

2.13.14 継承ではなくコンポジションを利用すべき場合

- ・ 継承を使うとクラス階層の上下でカプセル化が弱くなることが懸念される場合。(スーパークラスの変更がサブクラスに波及するなど)
- ・ サブクラスを移り変わるオブジェクトをモデル化する場合。
- ・ 他のオブジェクトに振舞いを委譲することで責務を拡張すべき場合。

[参考文献 Java オブジェクト設計 ピーター・コード+マーク・メイフィールド]

2.14 クラス図と観点(Martin Fowler,1997)

[参考文献 UML モデリングのエッセンス マーチン・ファウラー、ケンドール・シコット]

- ・ モデルの開発は、概念的レベル、仕様レベル、実装レベルと洗練されます。
- ・ ダイアグラム、特にクラス図を作成する場合には、次の3つの観点のうちの、どの観点を作成するのか、どの観点を作成されたものかを常に認識しておく必要があります。

2.14.1 概念的観点

- ・ 問題ドメインにおける概念を表わすダイアグラムを作成します。
- ・ 概念モデルは、それを実装するソフトウェアをほとんど無視して作成されるべきものです。

2.15 仕様の観点

- ・ ソフトウェアの実装ではなく、インタフェースについて作成します。従ってクラスではなく、(実装を持たない) タイプについて作成することになります。

2.15.1 実装の観点

- ・ 実際にクラスを与えてその実装を明らかにします。
- ・ ソースコードを作成します。

※Martin Fowler 氏は後に観点の導入は複雑過ぎたと述べています(筆者の記憶違いかもしれません)

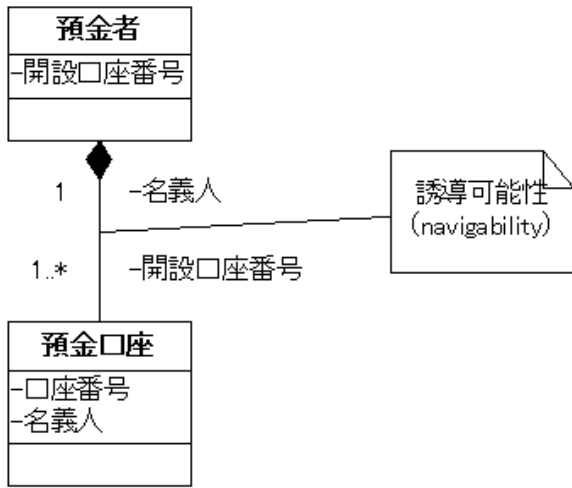
※エリック・エバンス氏のドメイン駆動設計ではモデルは1つを推奨しています

※観点という考え方を知ることは意味があると思います。

2.16 関連(インスタンス間の関係)

2.16.1 概念的観点

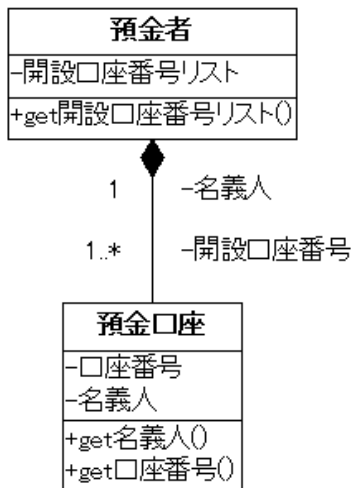
- ・ 関連は方向性を持つ2つのロールを持ちます。
- ・ ロールの元になるクラスはソース、宛先になるクラスはターゲットです。
- ・ 関連に関与するオブジェクトの数を多重度で表わします。



2.16.2 仕様の観点

- ・ 関連はレスポンスビリティを表わします。

例： 預金者クラスは開設済みの（0 個以上の）口座番号を応答するメソッドを持ちます。



2.16.3 実装の観点

- ・ 関連は各クラスに参照または参照の集合が実装される事を示します。
- ・ 実装の観点なので用語としては、オブジェクト識別子ではなく参照またはポインタで記述します。



```
package oop.chap2_4;
```

```
import java.util.ArrayList;
```

```
public class 預金者 {
    private ArrayList<預金口座> 開設口座リスト;
    public 預金者() {
        super();
        this.開設口座リスト = new ArrayList<預金口座>();
    }
    public ArrayList<預金口座> get 開設口座リスト() {
        return 開設口座リスト;
    }
}
```

```
package oop.chap2_4;
```

```
public class 預金口座 {
    private String 口座番号;
    private String 名義人;
    public 預金口座() {
        super();
        this.口座番号 = "";
        this.名義人 = "";
    }
    public String get 口座番号() {
        return 口座番号;
    }
    public String get 名義人() {
        return 名義人;
    }
}
```


2.16.4 誘導可能性 (navigability)

- ・ 概念的観点のダイアグラムでは誘導可能性を示す矢印が必要な場合は多くありません。
- ・ 単方向関連、双方向関連があります。
- ・ 仕様や実装の観点では必要な定義であり、重要な意味を持ちます。
- ・ 実装レベルでは、どちらのクラスに参照を持つか、両方のクラスに持つか等が決まります。
- ・ UML では、矢印のない関連は誘導可能性が未知であるか、双方向である事を示します。

2.16.5 静的関連・動的関連

- ・ 静的関連： 長期間にわたって存在する関連です。
- ・ 動的関連： 2つのオブジェクトが相互にやり取りしている関連です。
- ・ オブジェクト指向では、オブジェクトの視点から関連を見ます。仮に自分が、そのオブジェクトであるとして、他のオブジェクトをどのように参照しなければならないかを考えます。従って、関連は方向を持ちます。
- ・ データ中心のモデル化では2つのオブジェクトを（同時に見て）結合 (join) するものとして関連を考えます。従来の手法で関係データベースを用いる場合、オブジェクト間の関連は、外部キーと複数の表とのジョインで暗黙的に表わされています。オブジェクト指向では、関連は明示的に実装されます。通常は、クラスに実装されたインスタンス変数として存在します。

2.17 集約とコンポジション

2.17.1 集約

- ・ 全体オブジェクトー部分オブジェクトの関係を表わします。
 - ・ 集約と関連は区別しにくいことが多いです。
- （例）口座が解約（削除）された場合でも、取引は削除されません。

2.17.2 コンポジション

- ・ より強い集約の一種です。
- ・ 部分オブジェクトは唯一の全体オブジェクトに属します。
- ・ 連鎖削除。全体オブジェクトを削除すると、連鎖的に部分オブジェクトも削除されます。
- ・ 口座が解約（削除）されると取引も削除されます。

2.18 制約

下図において、制約とは、

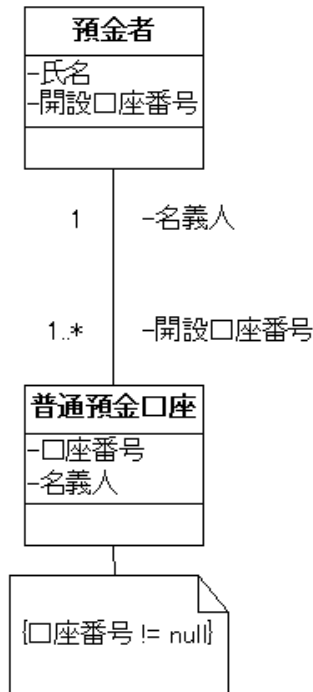
預金者は、0個以上の預金口座を持つこと。

預金口座は、唯一の名義人（預金者）が所有すること。

預金口座には、必ず口座番号があること。

です。

- ・ 実体（オブジェクト、クラス、属性、関連）が取り得る値の範囲を示します。
- ・ UML での制約は、{ } で囲まれた中に記述します。
- ・ 構文は定義されていないので、自然言語、論理式等を使えます。
- ・ また、OMG で定義するオブジェクト制約言語 OCL も使用できます。
- ・ 関連ロール、多重度を含め制約の表示は、クラス図の作成において、その大部分を占めます。



2.19 多相性(ポリモーフィズム)

2.19.1 用語

多態、ポリモーフィズムともいいます。

2.19.2 定義

あるインスタンスが、別のインスタンスにメッセージを送る場合に、送り手のインスタンスが、受け手のインスタンスが属するクラスに関係なく、メッセージを送ることができ、異なるクラスに対しては、異なる振舞いをする。 （受け手によってメソッド名を変えたり、パラメータを変えたりする必要がないこと）

2.19.3 制限付き多相性

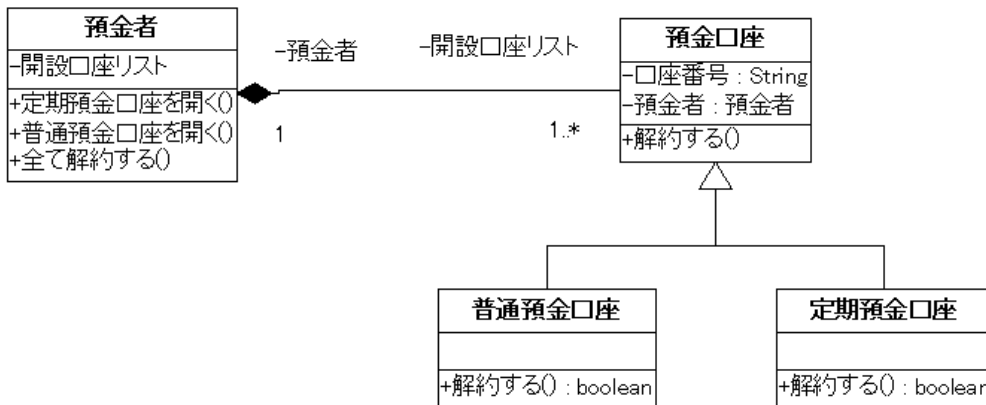
例： 開設口座に関連できるのは預金口座クラスとその子孫クラスのインスタンスであること。

<例>

「開設口座リスト」に登録されている全ての預金口座を解約する場合を考えます。(closeAllAccounts())

全ての預金口座を解約するためには、「開設口座リスト」に登録されている全てのオブジェクトに対して同じメッセージ

「解約する」を送ればよいことになります。受け手のオブジェクトが普通預金口座クラスか定期預金口座クラスに属するかを知る必要はありません。送られたメッセージ「解約する」は、受け手のオブジェクトがどのクラスに属しているかによって異なる振舞い（解約手続き）をします。この性質を多相性と呼びます。



```

package oop.chapt2_5;

/**
 * 預金口座
 * 抽象クラスなのでインスタンスは存在しない
 */
public abstract class Account {
    private String accountNumber; // 口座番号
    private Customer customer;

    /**
     * 実際には 各コンストラクタには引数(Customer 等)が必要と思われるが この例題では省略する
     */
    public Account() {
        super();
        accountNumber = "";
    }

    /**
     * 抽象メソッド close()
     * @return
     */
    public abstract boolean close();
}

```

```

package oop.chapt2_5;

/**
 * 普通預金口座
 */
public class OrdinaryAccount extends Account {
    public OrdinaryAccount() {
        super();
    }

    public boolean close() {
        boolean result = true;
        // 普通預金の解約チェックを行い、結果を result に設定する
    }
}

```

```

        // 解約できる場合のみ解約する
        return result;
    }
}

package oop.chapt2_5;

/**
 * 定期預金口座
 */
public class FixedAccount extends Account {
    public FixedAccount() {
        super();
    }
    public boolean close() {
        boolean result = true;
        // 定期預金の解約チェックを行い、結果を result に設定する

        // 解約できる場合のみ解約する
        return result;
    }
}

package oop.chapt2_5;

import java.util.ArrayList;

/**
 * 預金者
 */
public class Customer {
    private ArrayList<Account> accountList; // 開設口座リスト
    public Customer() {
        super();
        accountList = new ArrayList<Account>();
    }
    public void openOrdinaryAccount() {
        OrdinaryAccount oa = new OrdinaryAccount();
        // 普通預金口座を開設し、開設口座リストに追加する
        accountList.add(oa);
    }
    public void openFixedAccount() {
        FixedAccount fa = new FixedAccount();
        // 定期預金口座を開設し、開設口座リストに追加する
        accountList.add(fa);
    }
    public void closeAllAccounts() {
        for (Account a : accountList) {
            // 開設済みの全ての預金口座(普通か定期)を解約する
            a.close();
        }
    }
}

```

2.19.4 動的束縛

プログラミング言語として多相性を実現するために必要な仕組み。

動的束縛 ←→ 静的束縛

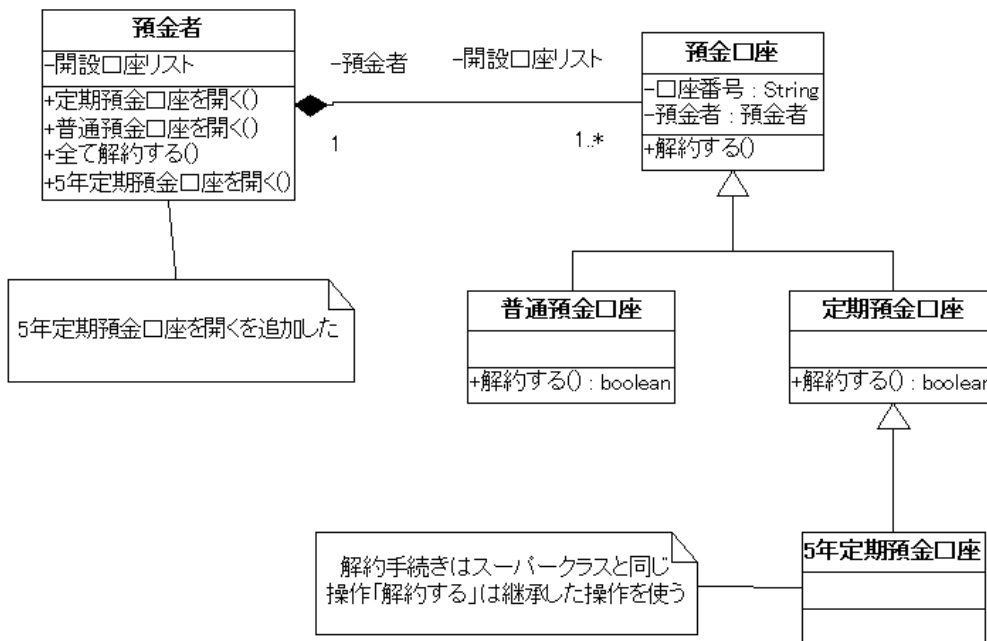
<例>

新たに 5 年定期預金口座を追加した場合を考える。

「解約する」方法は、定期預金口座と全く同じである。

5 年定期預金クラスのオブジェクトに、メッセージ「解約する」を送った場合、どうなるか？

預金者クラスの開設口座リストには、普通預金口座クラス、定期預金口座クラスのオブジェクトに加えて、5 年定期預金クラスのオブジェクトも登録される。例 1 で示した、全預金口座を解約する方法は同じで良いか？



```
package oop.chapt2_6;
```

```
import java.util.ArrayList;
```

```
import oop.chapt2_5.Account;
```

```
import oop.chapt2_5.FixedAccount;
```

```
import oop.chapt2_5.OrdinaryAccount;
```

```
/**
```

```
 * 預金者
```

```
 * openFixed5YearsAccount() を追加した
```

```
 */
```

```
public class Customer {
```

```
    private ArrayList<Account> accountList; // 開設口座リスト
```

```
    public Customer() {
```

```
        super();
```

```
        accountList = new ArrayList<Account>();
```

```
    }
```

```
    public void openOrdinaryAccount() {
```

```

        OrdinaryAccount oa = new OrdinaryAccount();
        // 普通預金口座を開設し、開設口座リストに追加する
        accountList.add(oa);
    }

    public void openFixedAccount() {
        FixedAccount fa = new FixedAccount();
        // 定期預金口座を開設し、開設口座リストに追加する
        accountList.add(fa);
    }

    public void openFixed5YearsAccount() {
        FixedAccount f5a = new Fixed5YearsAccount();
        // 定期預金口座を開設し、開設口座リストに追加する
        accountList.add(f5a);
    }

    public void closeAllAccounts() {
        for (Account a : accountList) {
            // 開設済みの全ての預金口座(普通か定期)を解約する
            a.close();
        }
    }
}

package oop.chapt2_6;

import oop.chapt2_5.FixedAccount;

/**
 * 5 年定期預金口座
 */
public class Fixed5YearsAccount extends FixedAccount {
    public Fixed5YearsAccount() {
        super();
    }
}

```

2.20 関連に関するその他の概念

幾つかの方法論で使われる概念を列挙します。

古い表現もありますが、考え方を知っておくと良いと思います。

2.20.1 合成 (has-a)

has-a と同様に方向を持つ関係です。

所有の関係を表わす。

飛行機は、胴体と翼とエンジンと…の部品から組み立てられる

オブジェクトをより詳細に記述するため

再利用する部品を作るため

2.20.2 格納(holds-a)

has-a と同様に方向を持つ関係である。

格納するオブジェクトはコンテナリストとも呼ぶ。

格納、埋め込み関係を表わす。

2.20.3 実装(is-implemented-using)

顧客リストはノートを使って実装される。

顧客リストクラスは、ノートクラスをインスタンス変数に持つ。

顧客リストクラスはノートクラスのサブクラスではない。

2.20.4 構成関連(パーティション関連)

あるオブジェクトが、他のオブジェクトから構成される。

家族は、人間から構成される。

オブジェクト「家族」は、オブジェクト「人間」を結び付けるために存在する。

2.20.5 連想(Knows-about)

オブジェクトAがオブジェクトBを知っているとき、AはBの共通インターフェース（多態）のどのメソッドも呼び出すことができる。これ以外の関係（has-a や holds-a 等）は存在しない。

人と人（面識）

3. オブジェクト指向システム開発

この章の内容

- オブジェクト指向開発の特徴
- オブジェクト指向開発の流れ
- 開発プロセスと作成するモデル
- モデル
- 分析プロセス
- 構築プロセス（設計モデルの作成）
- 構築プロセス（実装モデルの作成）
- テストプロセス（テストモデルの作成）
- ユースケースモデル

3.1 従来のシステム開発

3.1.1 機能とデータを分ける方法論

S A D T（Structured Analysis and Design Technique）構造化分析/設計技法

R D D（Requirement Driven Design base on SREM）S R E M要求駆動型設計

S A / S D（Structured Analysis and Structure Design）構造化分析/構造化設計

3.1.2 フォンノイマンハードウェアアーキテクチャに起源する

ハードウェアは、メモリ、中央制御装置、演算ユニット、入力、出力で構成される。

プログラムはメモリ内にあるデータを操作するための制御文を用いる。

ハードウェア・アーキテクチャに合致したこのスタイルは高級言語においても引き継がれ、データとプログラムを分離する指向が続いてきた。

機能中心法。

3.1.3 ウォータフォールモデル

理想的な新規開発だけを記述している。

要件定義が確定すると、全ての要件を満たすデータベースが設計される。

各処理の設計者はそのデータベースを理解する必要がある。

各処理はデータベースの構造を前提として設計・開発される。

仕様の追加や修正によってデータベースに変更が生じると、その影響範囲は各処理に及ぶことが多い。

3.2 オブジェクト指向開発の特徴

3.2.1 オブジェクトによるモデリング

理解の容易性（オブジェクトをもとにモデリングする。オブジェクトは実世界をもとに抽出する。）

オブジェクトの普遍性

変更の局所性

カプセル化（品質の単位になる。）

再利用、バージョンアップ（継承、サブタイピングを使用できる。）

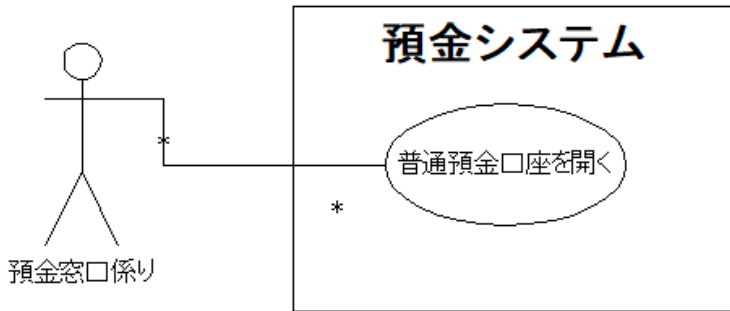
3.2.2 ユースケースの利用

ユースケースで要求モデルを作成する。

ユースケースを分析・設計の出発点とする。

従来は自然言語等で記述されていた要件定義書等に相当する。

これをより統一化された記法で表わし、システムのライフサイクルを通して使用する。



3.2.3 UML(Unified Modeling Language 統一モデル記述言語)

OMG により 1997.12 UML1.1 として標準化された。

以前は各手法毎に異なる記法が用いられ多少混乱する傾向にあった。

開発プロセスはシステムの規模、問題ドメイン等で異なるので、いかなるプロセスにおいても使用可能であり、そのための拡張性を持った記法メタモデルとして策定されている。

OMG（Object Management Group）はオブジェクト技術の標準化団体。CORBA(Common Object Request Broker Architecture)を策定している。

3.3 オブジェクト指向開発の流れ

| | | |
|------------|------------|-------------|
| オブジェクト指向分析 | オブジェクト指向構築 | オブジェクト指向テスト |
|------------|------------|-------------|

ユースケースを含むオブジェクトの追跡性がある。

3.3.1 オブジェクト指向分析

オブジェクトの発見

アプリケーションドメインにある名詞は有力な候補になる。

データ中心であったり、機能中心では、有効なオブジェクトを抽出できない。

有効なオブジェクトは、理解容易であり、再利用しやすい。

設計モデル、実装モデルで初めて現れるオブジェクトもある。（ファクトリーオブジェクトやプロキシオブジェクトなど）

オブジェクトの整理

クラス階層を作成する。

作成基準は、実際の世界での類似性をもとにする。

オブジェクトどうしのやり取りの記述

あるオブジェクトが、システムで果たす役割を把握する。

他のオブジェクトとのやり取りを、シナリオ、ユースケースに表わす。(O O S E)

オブジェクトの属性と操作の定義

他のオブジェクトが送ることができるメッセージ、利用できる操作を定義する。

オブジェクトの内部の定義

※ 分析結果の理解容易性

オブジェクト指向分析では、人間が自然に現実を見る方法をもとにするために、機能とデータを分けた場合の分析結果よりも理解しやすい分析結果が得られる。

3.3.2 オブジェクト指向構築

分析モデルをもとに、ソースコードとして実装すること。

実装環境に合わせて、分析モデルを変形しなければならない場合もある。

可能な限り、分析モデルとの対応が取れるように設計モデル、言語に変換する。


コンポーネントを利用する。(以前に開発されたソースコードを利用する)

3.3.3 オブジェクト指向テスト

従来型の単体テストは、ルーチン、モジュールについて行う。オブジェクト指向システムではオブジェクト単位となり従来型よりも大きな単位となる。統合テストに移るときのハードルは低くなる。

テスト済みの親クラスの操作を、継承する子クラスのオブジェクトについても、新たにテストが必要である。

3.4 開発プロセスと作成するモデル

| プロセス | モデル |  <p>追跡可能性。 あるモデルのオブジェクトから他のモデルのオブジェクトを追跡できる。</p> |
|--------------------|--------|---|
| 分析プロセス | 要求モデル | |
| | 分析モデル | |
| 構築プロセス 設計サブプロセス | 設計モデル | |
| 構築プロセス 実装サブプロセス | 実装モデル | |
| テストプロセス | テストモデル | |

3.5 モデル

システム開発はモデルを開発する事である。

各モデルは作成するシステムのある側面を捉えるためのものである。

順次作成するモデルの中に複雑さを次第に導入することにより、システムの複雑さを管理する。

モデル間の変換は機械的ではなく、漸近的であり、才能ある開発者による創造的な作業である。

全てのモデルは追跡可能性を持つ。すなわち、あるモデルに現れたオブジェクトは他のモデルに現れるオブジェクトとして追跡できる。

3.5.1 OOSE法

要求モデル

分析モデル

設計モデル

実装モデル（ソースコード）

テストモデル（実装モデルのテスト結果）

3.5.2 OMT法(参考)

オブジェクトモデル

動的モデル

機能モデル

3.6 分析プロセス

分析プロセス→構築プロセス（設計サブプロセス）→構築プロセス（実装サブプロセス）→テストプロセス

分析プロセス：要求モデル（ユースケースモデル、インタフェース記述、ドメインモデル）、分析モデル

分析プロセスでは要求モデルと分析モデルを作成する。

3.6.1 要求モデル（分析プロセスでは、要求モデルと分析モデルを作成する）

要求仕様から要求モデルに変換する。

要求モデルでは、システムを使用するユーザがどのようにシステムを使うのかを記述する。

発注者、ユーザ主導で開発する。

システム範囲・境界を定義する。

要求モデルの構成要素

ユースケース・モデル

インタフェース記述（ユースケース・モデルを支援する上で有効な場合）

問題ドメインモデル

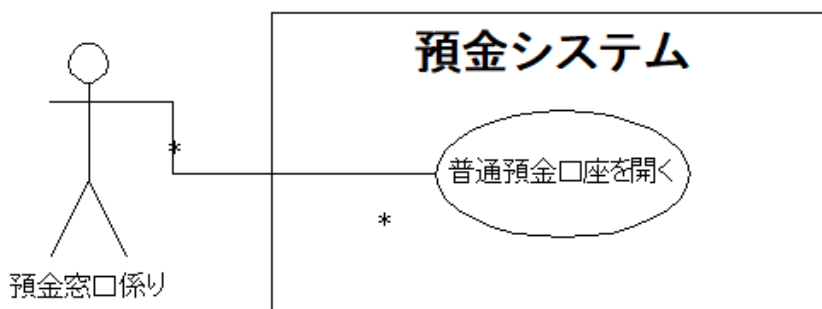
3.6.2 ユースケース・モデル

分析プロセス→構築プロセス（設計サブプロセス）→構築プロセス（実装サブプロセス）→テストプロセス

分析プロセス：要求モデル（ユースケースモデル、インタフェース記述、ドメインモデル）、分析モデル

アクタとユースケースを使用する。

アクタとはシステムの外部に存在し、ユーザや外部のシステムが行う役割を表現する。



アクタはクラス、ユーザ自身や外部システム（カード決済システム）はアクタ・クラスのインスタンス。

ユースケースは、ユーザがシステムを利用するときに行う一連の処理を指す。

全てのユースケースの記述の集合はシステムの機能を完全に指定する。

アクタを取り出す。次にアクタが必要とするユースケースを取り出していく。

3.6.3 インタフェース記述

分析プロセス→構築プロセス（設計サブプロセス）→構築プロセス（実装サブプロセス）→テストプロセス

分析プロセス：要求モデル（ユースケースモデル、インタフェース記述、ドメインモデル）、分析モデル

ユースケースモデルを支援する上で有効な場合に作成する。

GUI プロトタイプを作成する。

他のシステムとのインタフェースを定義する。

3.6.4 ドメインオブジェクトモデル

分析プロセス→構築プロセス（設計サブプロセス）→構築プロセス（実装サブプロセス）→テストプロセス

分析プロセス：要求モデル（ユースケースモデル、インタフェース記述、ドメインモデル）、分析モデル

概念的観点のクラス図である。

問題ドメインから直接抽出されたオブジェクトで構成される。

ユーザが使用する概念を直接表現したものである。

ユースケースモデルと協調して作成し、ユースケースを記述するための名詞の一覧になる。

Coad/Yourdon 法、Booch 法などでは最初のオブジェクトモデルが実装におけるクラスに直接マッピングされるが、OOSE ではドメインオブジェクトモデルを作成した後で、変更に強い分析モデルを作る。

ドメインオブジェクトは以降のプロセスでより洗練され詳細化される。

オブジェクト名

論理的属性

静的インスタンス関連

継承

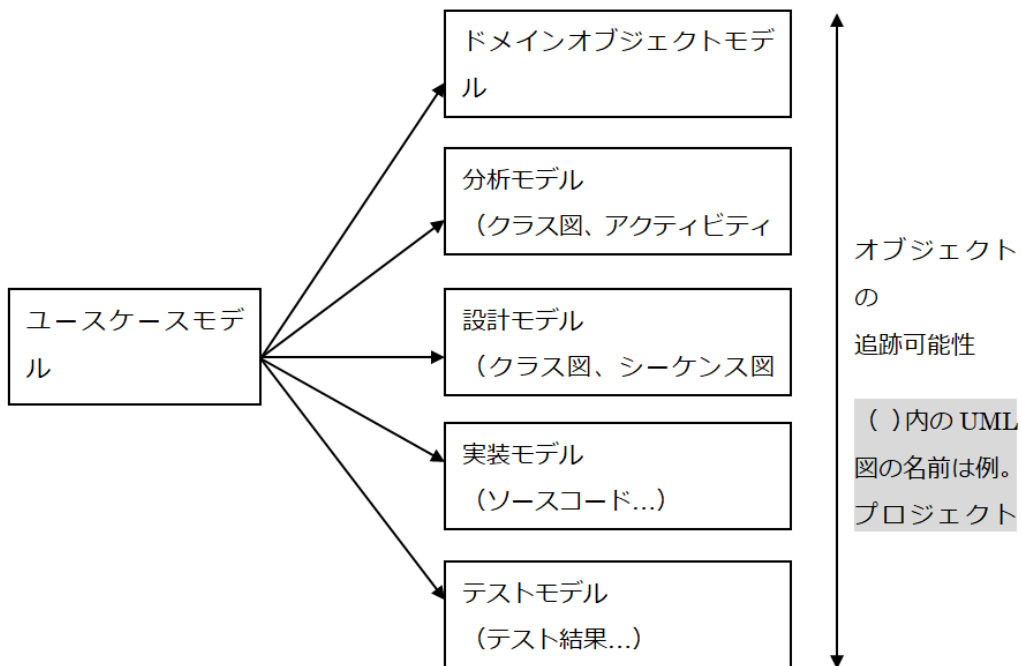
動的インスタンス関連

操作

ドメインオブジェクトの多くは、分析モデルの実体オブジェクトとして現れる。

3.6.5 ユースケース主導設計

ユースケースモデルは、全てのモデルのもとになる。



従来の仕様書と対比する

要件定義書

外部設計書

内部設計書

ソースコード

テスト仕様書

利用者、発注者が理解できる範囲は？

上流工程からの記述の追跡性は？

ライフサイクルを通して利用できるものは？

3.6.6 分析モデル

分析プロセス→構築プロセス（設計サブプロセス）→構築プロセス（実装サブプロセス）→テストプロセス

分析プロセス→要求モデル（ユースケースモデル、インタフェース記述、ドメインモデル）、分析モデル

発注者やユーザによって要求モデルが確認された後で、分析モデルの作成を開始する。

3種類のオブジェクトを使ってモデル化する。 → ステレオタイプ

実体オブジェクト （ほとんどのドメインオブジェクトはここに分類される）

インタフェースオブジェクト （GUI等のインタフェースに依存する振舞いと情報をモデル化したもの）

制御オブジェクト （実体オブジェクトやインタフェースオブジェクトの振舞いとして自然にモデル化できないような振舞いがある。 そのような振舞いをモデル化したもの。 例：全口座残高の計算など。実装時にはクラスメソッドを使うようなもの。）

分析モデルを作らずに次のプロセスにはいる手法もある。

UML ではステレオタイプの概念が取り入れられている。

（OOSE）「最も安定したシステムは、現実世界のものを反映したものだけを使って構成されるものではない」。3種類のオブジェクトに現れるような人工的なドメインオブジェクトを考慮する事で、変更を局所化できるような強いモデルが得られる。インタフェースの変更はインタフェースオブジェクトに局所化される。

3.6.7 ステレオタイプ (stereotype 単純化された定型概念)

オブジェクトの高次の分類

Rebecca Wirfs-Brock(1990)

コントローラ

コーディネータ

Jacobson(1994)

インタフェース

コントロール

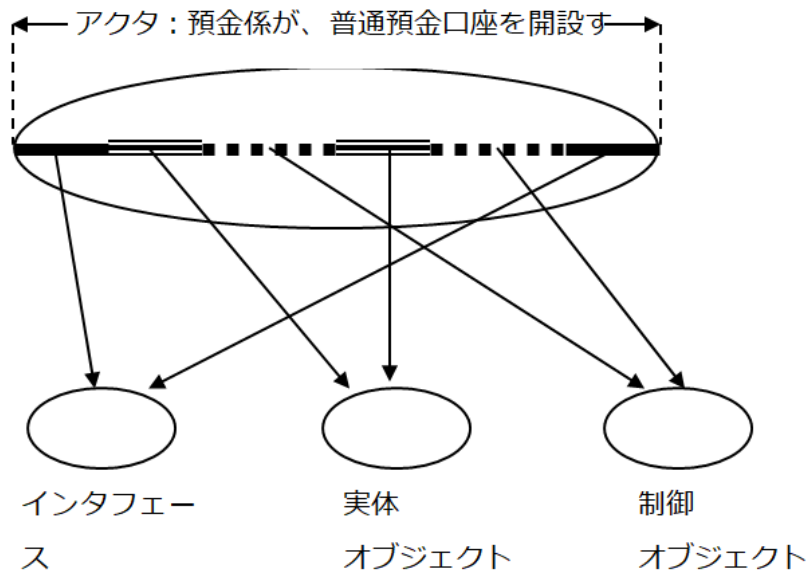
エンティティ

参考資料 1 S A S Dとオブジェクト指向による設計例の比較

変更・追加時の局所性

3種類のオブジェクトを使う場合の効果

ユースケースモデルから分析モデル



システム環境に直接依存するユースケースの機能はインタフェースオブジェクトに割り当てる。

どのインタフェースオブジェクトに置く事も不自然であるデータ領域の取り扱いや情報の処理を行う機能は実体オブジェクトに割り当てる。(一般にドメインオブジェクトから導かれる)

1つあるいは2、3のユースケースに限定され、いかなる他のオブジェクトに置くことも不自然な機能は制御オブジェクトに割り当てる。

基本的な責務割り当ての原則は、変更の局所性を達成できることである。

3.6.8 アナリシスパターン

Martin Fowler: Analysis Patterns: Reusable Object Models.

3.7 構築プロセス(設計モデルの作成)

分析プロセス→構築プロセス(設計サブプロセス)→構築プロセス(実装サブプロセス)→テストプロセス

分析モデルを実装環境に適合させる。(分析モデルは理想的な実装環境を前提とする)

分析モデルの構造が設計モデルの骨格であることが理想であるが、関係データベース、分散環境、レスポンス、実装言語、並行プロセス等を導入する場合、分析モデルを変更する必要がある。このため設計モデルとして新しいモデルを開発する。

設計モデルの作成を含む構築プロセスでは、一般に複雑さが増加する。

複雑さを管理するために、サブシステム、UMLのパッケージと依存関係を導入する。

(1) デザインパターン

Erich Gamma 他 Design Patterns: Elements of Reusable Object-Oriented Software

3.8 構築プロセス(実装モデルの作成)

分析プロセス→構築プロセス(設計サブプロセス)→構築プロセス(実装サブプロセス)→テストプロセス

3.8.1 ソースコード。

オブジェクト指向言語は必須ではないが、オブジェクト指向の基本的概念が簡単に言語構文に変換できるという理由でオブジェクト指向言語が望ましい。

例えば関係データベースを使用する場合、型変換、検索などが含まれ複雑になる。

3.9 テストプロセス(テストモデルの作成)

分析プロセス→構築プロセス（設計サブプロセス）→構築プロセス（実装サブプロセス）→テストプロセス

テスト結果を記述したものである。

オブジェクトモジュール、ユースケース、システム全体の順にテストを行う。

3.10 ユースケースモデル

分析プロセス→構築プロセス（設計サブプロセス）→構築プロセス（実装サブプロセス）→テストプロセス

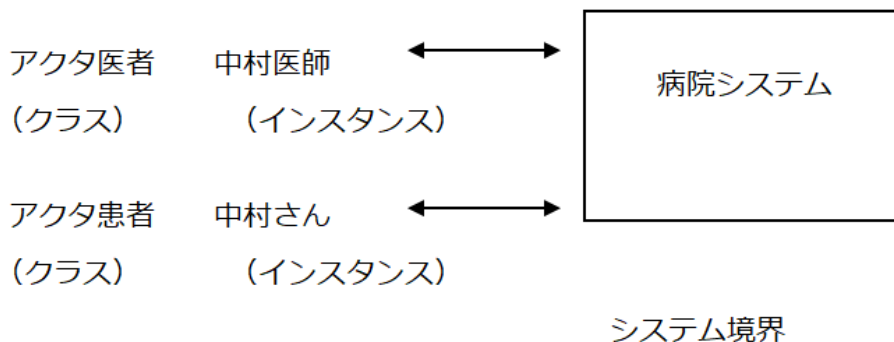
3.10.1 アクタ

システムの外部に存在し、システムと情報交換する全てのもの。（人や機械、他のシステム）

主アクタと副アクタがある。例）現金自動支払い機システムでの顧客（主アクタ）と修理係（副アクタ）。副アクタは、主アクタがシステムを使用できるようにするためにシステムを扱う。ユースケースの識別は主アクタから始める。

アクタクラスのインスタンスがユーザである。

アクタはユーザが行う役割を表わす。



3.10.2 ユースケース

各ユースケースは、最初にアクタによって起動される事象で構成される。

アクタとシステムとの間のインタラクションを定める。

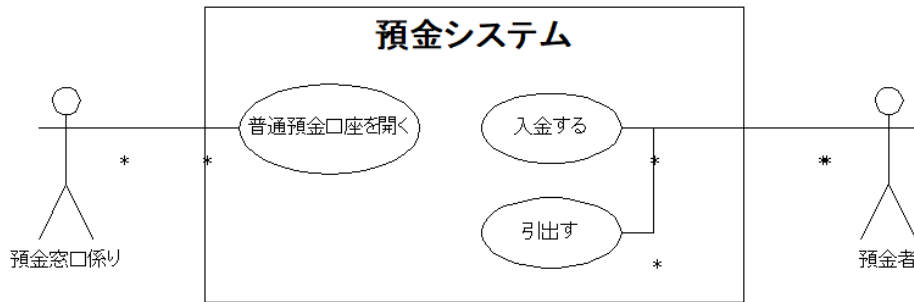
ユースケースの全体集合によって、全てのシステム機能を規定できる。

ユースケースはシステムのある限定された部分の機能に焦点を当てている。

ユースケースの数が増える毎に漸近的にシステム全体を分析できる。

異なった機能領域に対するユースケースを独立に開発し、その後で統合する。これは、並列的な開発を行う場合にも有

効である。



3.10.3 基本系列と代替系列

そのユースケースを理解するのに最も適した事象の列を基本系列とする。

基本系列の変形やエラー系列は代替系列として記述する。

3.10.4 ユースケースは振舞いと状態を持つオブジェクト

(注) 預金システムをオブジェクトとみなし、ユースケースをシステムで実行される操作という見方を OOSE ではしない。最終的なシステム自身をオブジェクトとする見方はしない。

実際にはかなりの数のユースケースを書くことになる。書き出すことで、もとの要求仕様書の不明瞭な点分かる。記述の視点や範囲が決められているので書きやすく、読みやすい。

ユースケースを分割するよりも、長く広範囲の方が有効である場合が多い。

3.10.5 ユースケースの拡張関連

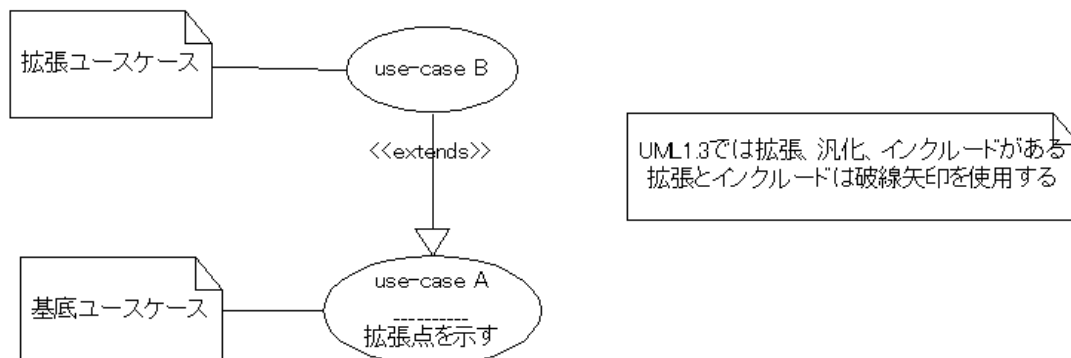
初めに単純で標準的なユースケースを作成する。(基本系列) その後、そのユースケースにバリエーションが必要になった場合、元のユースケースに直接バリエーションを追加できるが、場合によっては基本系列が見えにくくなったりする。そこで、バリエーションを別のユースケースとして作成し拡張関係によりユースケースを完成させることができる。

(選択的な振る舞い) バリエーションや例外処理などのサブフローのこと。

それぞれ独立したユースケースとして記述する。

基底となるユースケースの記述の中に、基底ユースケースのどこに拡張ユースケースを組込むかを記述する。(拡張点の記述)

サブフローは全て別にしなくても構わない。複雑度との兼ね合いになる。

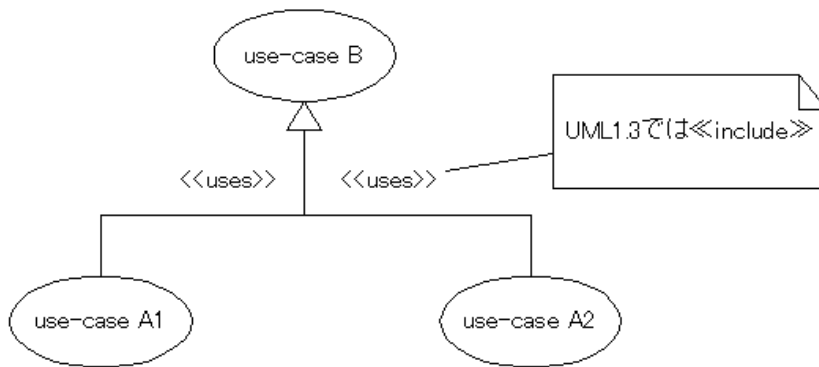


- (例) 基底ユースケース： 定期預金口座を解約する。
 拡張点： その定期預金口座が担保設定されている場合
 拡張ユースケース： 担保解除処理を行う。

3.10.6 ユースケースの使用関連

ユースケース B はユースケース A1 とユースケース A2 によって使用される。

ユースケース A1, A2 には共通の振舞いがあり、両方のユースケースへの繰返し記述を避けるために利用する。



- (例) ユースケース B： 顧客を認証する。
 ユースケース A1： 普通預金口座を解約する。
 ユースケース A2： 普通預金口座から引出す。

使用関連も拡張関連も継承の一種と見ることができる。ただし、オブジェクト指向プログラミング言語での継承と同じではない。

3.10.7 抽象ユースケース、具象ユースケース

抽象ユースケースは、実際にはインスタンスを持たない。

拡張ユースケースのほとんどは抽象ユースケースである。(基底ユースケースに組みこまれて具象ユースケースになる)

具象ユースケースは、実際にインスタンスを持つ。

抽象ユースケースの記述は具象ユースケースの中で利用される。

ユースケースのインスタンスが具象ユースケースの記述に従って動作し、ある時点から具象ユースケースに変わって抽象ユースケースの記述に従って動作が継続し、再び具象ユースケースに戻る。

4. 統一モデリング言語 UML

この章の内容 (UML1.3)

- ユースケース
- クラス図
- 相互作用図（シーケンス図）
- 相互作用図（コラボレーション図）
- パッケージ図
- ステートチャート図（状態図）
- アクティビティ図
- ステレオタイプ
- OCL（Object Constraint Language）

4.1 UMLの概要

UML（Unified Modeling Language）は方法論ではなく、モデリング言語である。

モデリングを行うための言語を統一したものが UML である。

モデリングとは「観察の対象となる領域（対象領域）を人工的に投影する作業」を指す。

1997 年 11 月 OMG（Object Management Group）標準となる。

UML1.3

1999 年 6 月発行

4.1.1 ユースケース図(use case diagram)

ユースケース図

4.1.2 相互作用図(interaction diagram)

シーケンス図

コラボレーション図（協調図）

4.1.3 静的構造図(static structure diagram)

クラス図

オブジェクト図

4.1.4 振舞い図(behavior diagram)

ステートチャート図

アクティビティ図

4.1.5 実装図(implementation diagram)

コンポーネント図

配置図

4.2 ユースケース

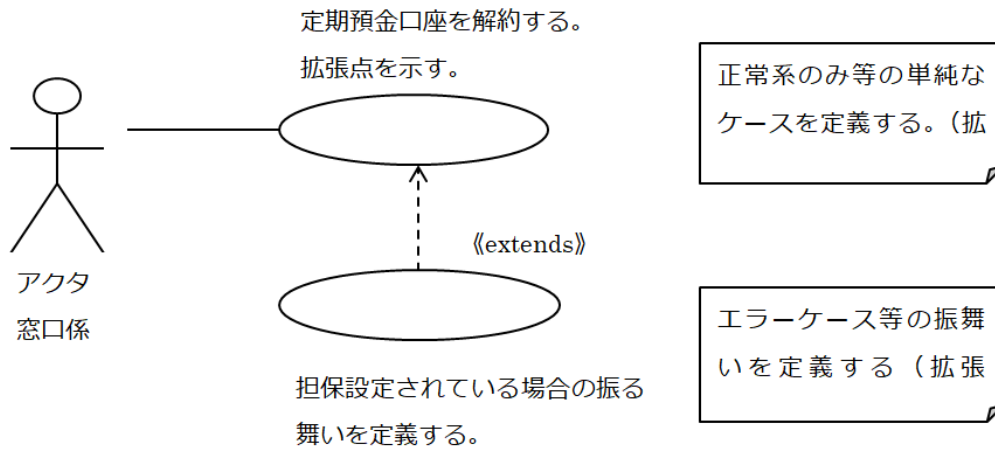
関連 アクタとユースケースの関係

拡張

インクルード

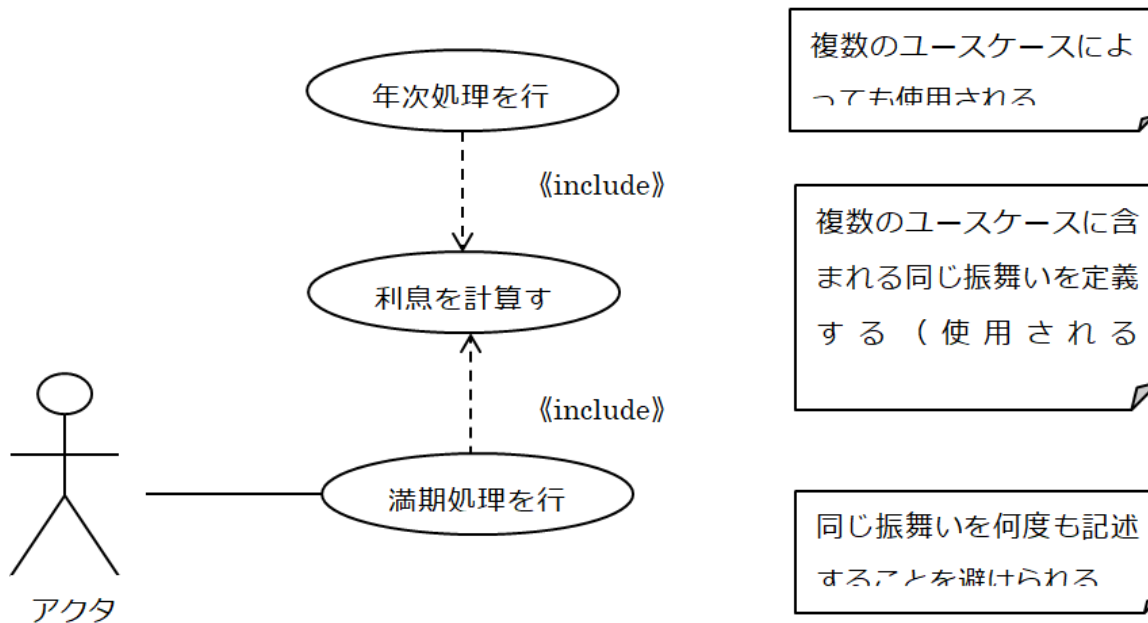
汎化

拡張



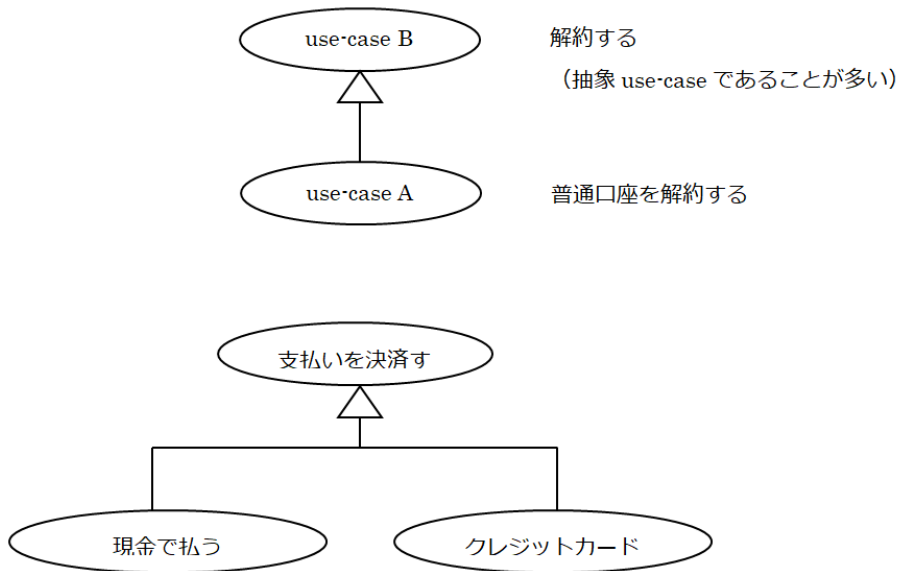
基底ユースケースは、拡張ユースケースがなくても完結する。

インクルード



include する側のユースケースは単独では完結しない。

汎化



メリット：理解しやすくなる。

4.2.1 ユースケースを使う

ヒアリングの結果のまとめ

システム要件定義書

プロジェクト計画のベース

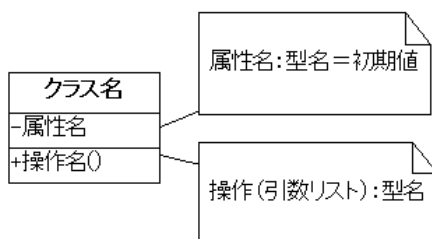
繰り返し開発、イテレーションの単位

ユースケースはシステムの外見を表わす。従って、システム内のクラスとの関係は（ここでは）考えない。

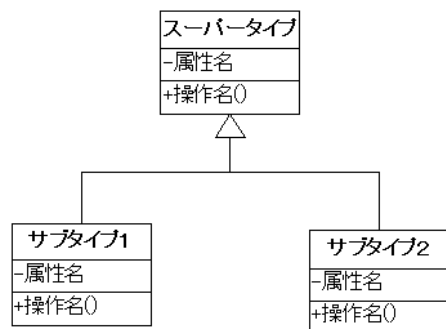
[参考文献 UML モデリングのエッセンス マーチン・ファウラー、ケンドール・シコット]

4.3 クラス図

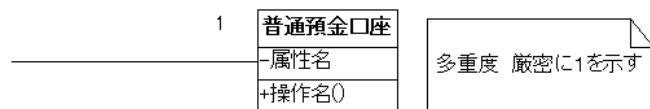
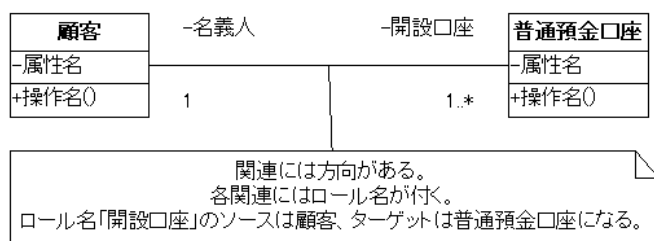
4.3.1 クラス



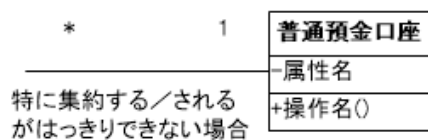
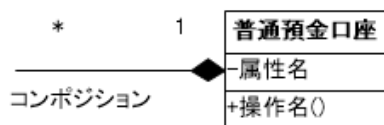
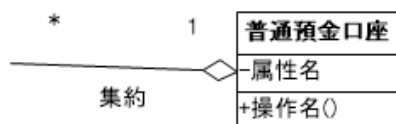
4.3.2 汎化・特化



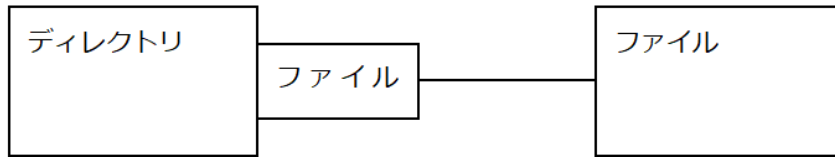
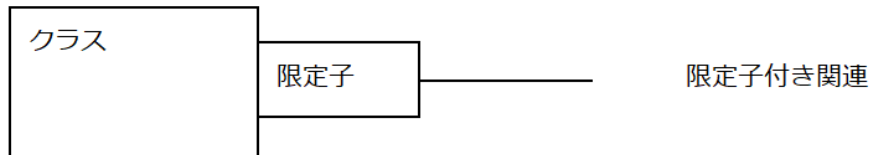
4.3.3 関連、ロール、多重度



4.3.4 集約、コンポジション



4.3.5 限定子

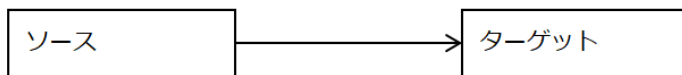
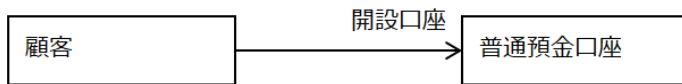


限定子によって関連の多重度を減らすことができる。

この例では、ファイル名が限定子。

限定子ファイル名によって 1 対多の関連を、1 対 1 に減少させる。

4.3.6 誘導可能性、依存関係



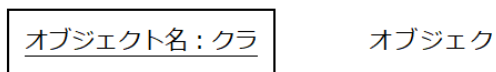
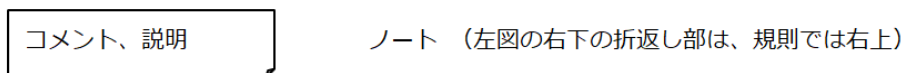
実装モデルにおいて、誘導可能性（navigability）の有無は重要な意味を持つ。

上の例では、顧客オブジェクトは普通預金口座オブジェクトの参照を持つことを示す。

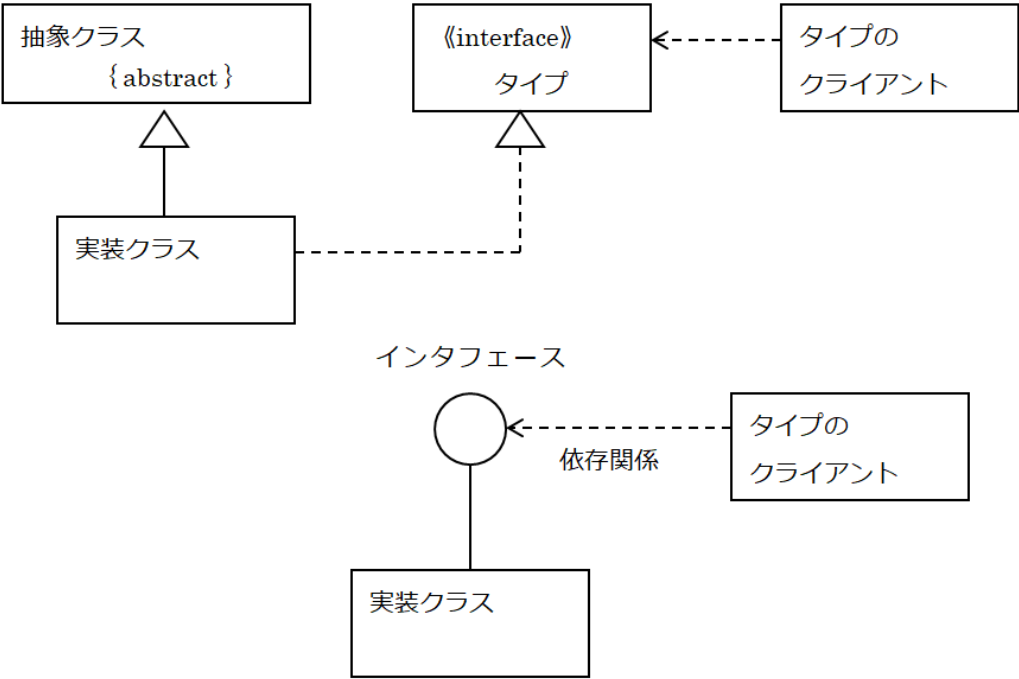
4.3.7 補助的記法

{制約の記述}

《ステレオタイプ名》

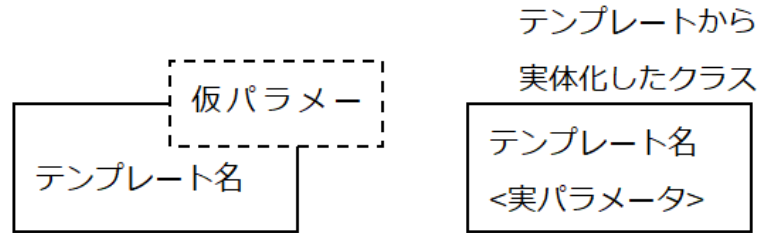


4.3.8 インタフェース

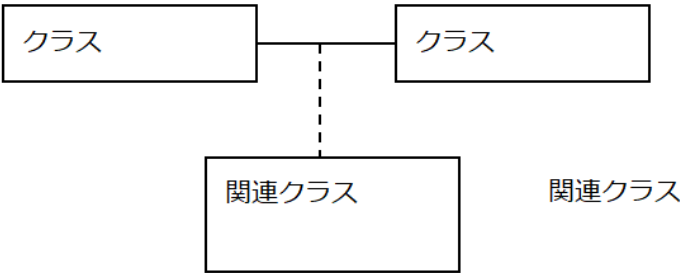


4.3.9 パラメタライズド・クラス、テンプレート・クラス

パラメタライズドクラス



4.3.10 関連クラス

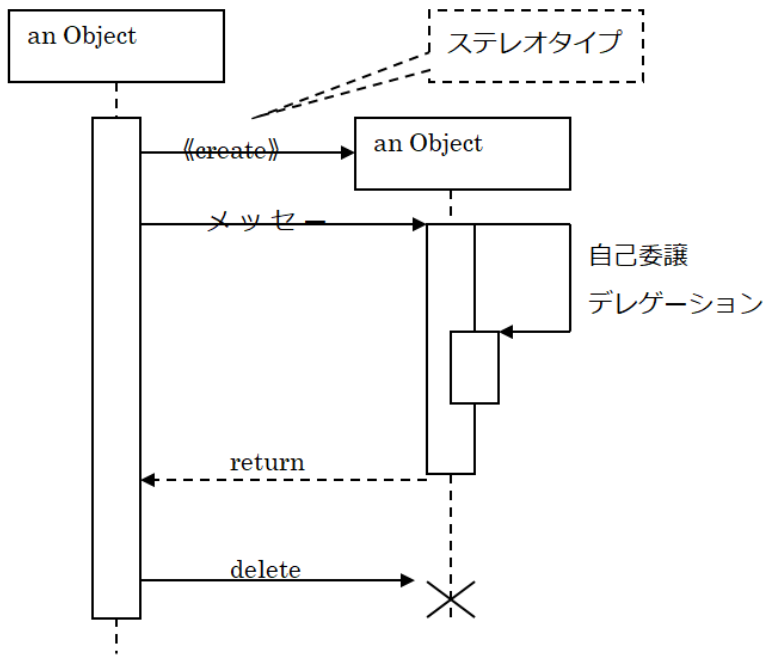


4.3.11 クラス図を使う

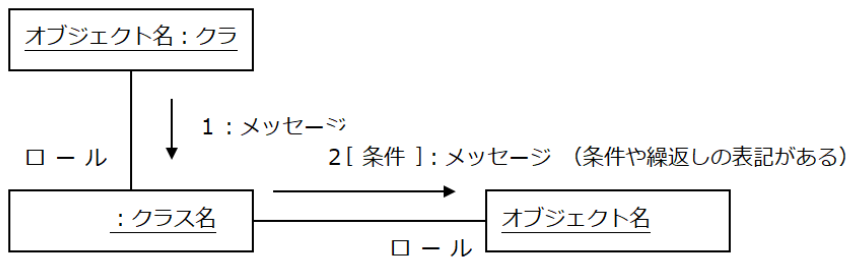
すべての表記を使うことに注力しない。
観点を意識する。(概念の観点、仕様の観点、実装の観点)
あらゆるものに対してモデルを作成しない。

4.4 相互作用図

4.4.1 シーケンス図



4.4.2 コラボレーション図



4.4.3 相互作用図を使う

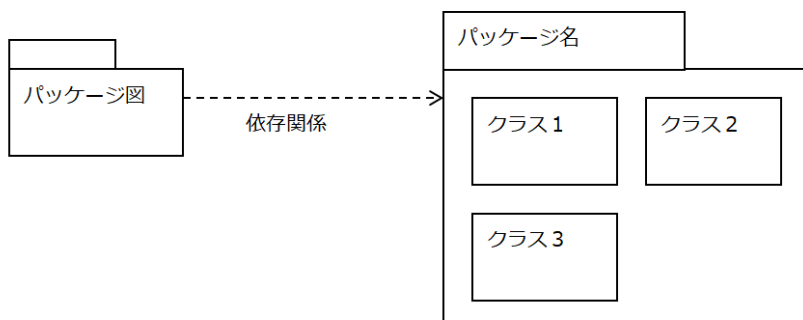
1つのユースケースにおけるオブジェクト群の振舞いを示す。

オブジェクト間のコラボレーションを示す場合に適する。

あるオブジェクトの振舞いを正確に定義する場合は適さない。(この場合には、ステートチャート図 (状態図)、アクティビティ図を使う)

[参考文献 UML モデリングのエッセンス マーチン・ファウラー、ケンドール・シコット]

4.5 パッケージ図

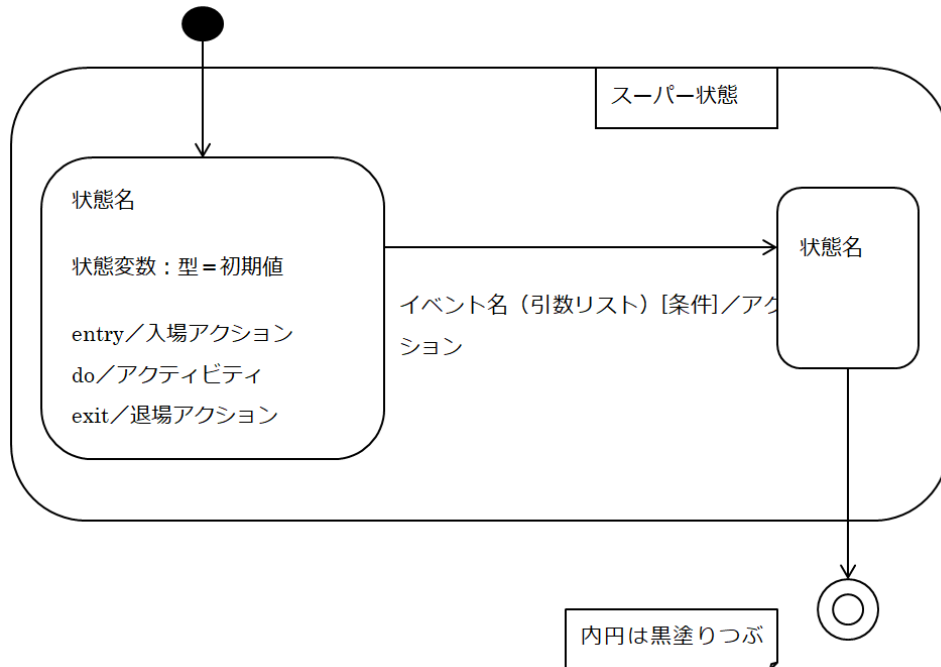


4.5.1 パッケージ図を使う

大規模なプロジェクトでは不可欠。

4.6 振舞い図

4.6.1 ステートチャート図(状態図)



4.6.2 ステートチャート図を使う

複数のユースケースにわたる 1 つのオブジェクトの振舞いを記述する。

(複数のユースケース、すなはち、そのドメインにおけるそのオブジェクトの振舞い)

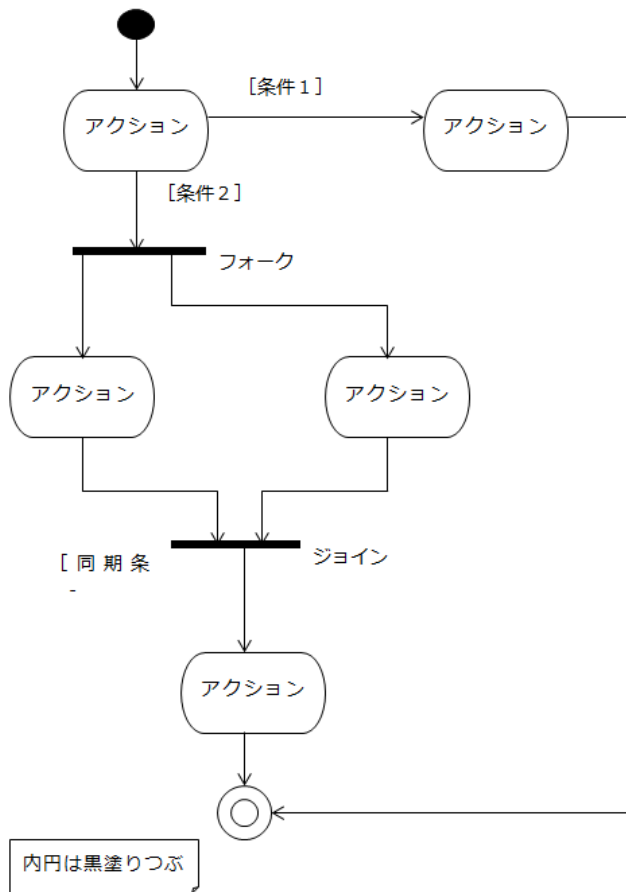
ステートチャート図での記述に適したオブジェクトの例

ユーザインタフェース

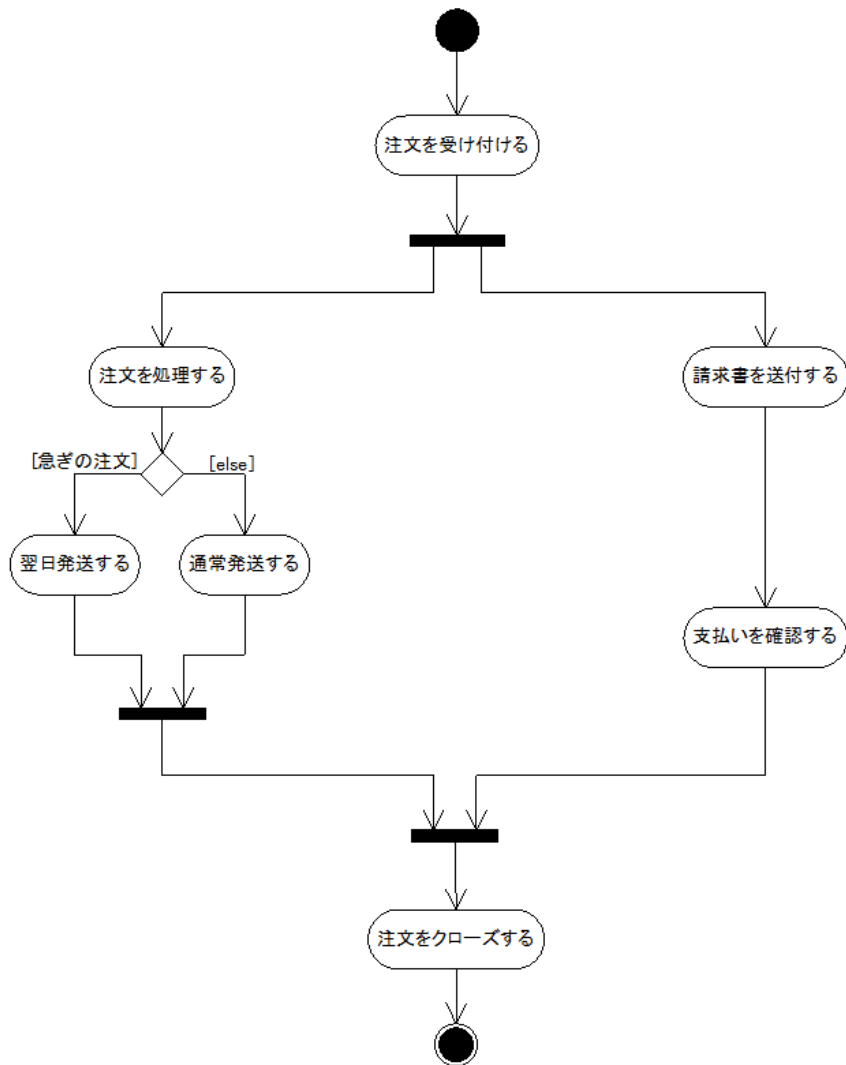
制御 (MVC の Control)

[参考文献 UML モデリングのエッセンス マーチン・ファウラー、ケンドール・シコット]

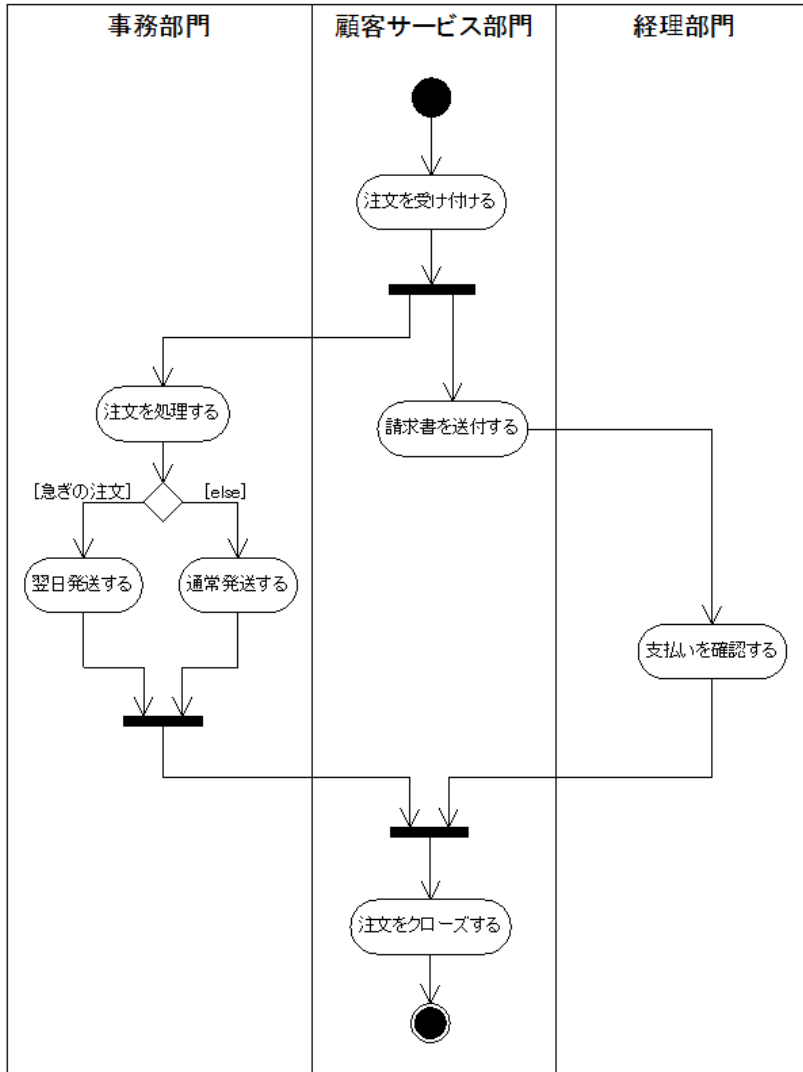
4.7 アクティビティ図



4.7.1 アクティビティ図(例1)



4.7.2 アクティビティ図(レーンの使用例)



4.7.3 アクティビティ図を使う

[参考文献 UML モデリングのエッセンス マーチン・ファウラー、ケンドール・シコット]

ユースケースを分析する。

アクションをオブジェクトに結びつける必要がない。(必要がある場合は適さない)

ワークフローを理解する。

複雑な逐次アルゴリズムを記述する。

UML に準拠したフローチャートと見なす。

マルチスレッドのアプリケーションを扱う。

4.8 ステレオタイプ

UML ではいくつかのステレオタイプを定義している。

4.8.1 抽象化 (Abstraction)

《drive》

《realize》

《refine》

《trace》

4.8.2 関連 (Association)

《implicit》

4.8.3 関連の終端 (AssociationEnd)

《association》

《global》

《local》

《parameter》

《self》

4.8.4 振舞いの特徴 (BehavioralFeature)

《create》

《destroy》

4.8.5 呼出しイベント (CallEvent)

《create》

《destroy》

4.8.6 クラス (Class)

《implementationClass》

《type》

4.8.7 クラシファイヤ (Classifier)

《metaclass》

《powertype》

《process》

《thread》

《utility》

4.8.8 コメント (Comment)

《requirement》

《responsibility》

4.8.9 コンポーネント (Component)

《document》

《executable》

《file》

《library》

《table》

4.8.10 制約 (Constraint)

《invariant》

《postcondition》

《precondition》

4.8.11 流れ (Flow)

《become》

《copy》

4.8.12 汎化 (Generalization)

《implementation》

4.8.13 オブジェクトフロー状態 (ObjectFlowState)

《signalflow》

パッケージ (Package)

《facade》

《framework》

《stub》

《toplevel》

パーミッション (Permission)

《access》

《friend》

《import》

4.8.14 ユーセージ (Usage)

《call》

《create》

《instantiate》

《send》

4.9 OCL (Object Constraint Language)

不変条件の表現

事前条件/事後条件の表現

演算子の優先順位

コメント

4.10 モデリングツール

~~Rational Rose for Java (日本ラショナルソフトウェア)~~

~~Cittera (Koneso) (オージス総研)~~

~~MagieDraw Pro (エッチ・アイ・シー)~~

~~WithClass (グレースィティ)~~

~~Pattern Weaver (テクノロジックアート)~~

~~Together ControlCenter (トウゲザ・ソフト・ジャパン)~~

~~WebGain StructureBuilder Enterprise Edition (ウェブゲイン・ジャパン)~~

~~BridgePoint (東陽テクニカ)~~

~~Visio (マイクロソフト)~~

~~ArgoUML(<http://argouml.tigris.org/>)~~

5. オブジェクト指向プログラミング言語 Java

5.1 オブジェクト指向プログラミング言語

引用 [Martin Fowler 1997]

「オブジェクト指向言語でプログラムを作成する方法を学ぶことは難しいことではありません。

問題になるのは、“オブジェクト指向言語がもたらす利点を生かす” ということを経験するには時間がかかるという点です。

(大きなパラダイムシフトが必要)」

5.2 Java の歴史

主な歴史（概略なので正確ではありません）

1995 年 サン・マイクロシステムズが Java を発表。

1996 年 JDK1.0 版

2004 年 J2SE1.5.0

2006 年 OpenJDK

2010 年 オラクルがサン・マイクロシステムズを買収

2014 年 Java SE8

2017 年 AdoptOpenJDK

2020 年 Adoptium

2024 年 Java SE22

5.3 Java の特徴

5.3.1 オブジェクト指向プログラミング言語

- オブジェクト指向設計モデルがスムーズに実装できます。
- カプセル化
- 継承
- 多態性（多相性、ポリモーフィズム）
- クラスベース
- Java、C#、 Kotlin、 Ruby、 Python、、、

5.3.2 シンプル

- C++などの複雑さは言語仕様から除いてあります。（何でも出来る指向の C++は肥大化しているとも言えます）

5.3.3 インタプリタ型

- コンパイルするとバイトコード（中間コード）が生成され、バイトコードは JVM 内のインタプリタで実行されます。
- プロトタイピングに向いています。

5.3.4 クラスライブラリ

- JDK の一部として（標準として）、良く利用する機能を持ったクラスが用意されています。

5.3.5 安定性

- 強い型付けの言語
- コンパイル時と実行時の二重チェック
- リファレンス（C++のポインタ演算はできません）

5.3.6 セキュリティ

- 従来の言語では、アプリケーションの安全性は言語の仕様とは独立しています。
- Java ではセキュリティのパッケージが JDK に含まれています。

5.3.7 プラットフォーム依存しない

- 再コンパイルせずに複数のプラットフォームで動きます。
- Write Once Run Anywhere
- 主要な OS 用の JVM（Java 仮想マシン）が提供されています。

5.3.8 マルチスレッド

- 並列プログラミングによる並列処理が可能です。
- 同期プリミティブが高機能、豊富

5.3.9 ソフトウェア・プラットフォーム

- Java 仮想マシン（Java Virtual Machine）によりマシンや OS に対して独立しています。
- Windows、MacOS、Linux 用の JVM が提供されています。

5.3.10 ガーベッジ・コレクタ（garbage collector）

- プログラムが使わなくなったメモリを自動的に開放します。
- C++で new したものを、delete や free し忘れるとメモリリークを引き起こす。
- JavaVM のインタプリタが管理するメモリ領域以外はアクセスできない。（セキュリティ強化、障害予防）

5.3.11 カプセル化 (Encapsulation):

データとそれに関連するメソッドを 1 つの単位（クラス）にまとめ、外部からデータを直接アクセスできないようにします。これにより、データの保護と内部の実装の隠蔽が可能になります。

アクセス修飾子（public, private, protected, なし）を使って、データの可視性を制御します。

5.3.12 継承 (Inheritance):

既存のクラス（スーパークラス）を基に新しいクラス（サブクラス）を作成する機能です。

サブクラスはスーパークラスの特性を継承し、さらに独自の特性を追加することができます。

再利用性とコードの簡潔さを高め、階層的な関係を表現します。

5.3.13 ポリモーフィズム (Polymorphism):

同じ操作を異なるデータ型のオブジェクトに対して行うことができる機能です。

これにより、同じメソッド名でも異なる実装を持つことができます。

オーバーライド（メソッドの再定義）やオーバーロード（同じメソッド名で異なる引数リストを持つ複数のメソッド）を利用します。

5.3.14 抽象化 (Abstraction):

必要な情報だけを取り出し、詳細を隠すことで、システムの複雑さを管理しやすくします。

抽象クラスやインターフェースを使って、共通の特性や動作を定義し、具体的な実装をサブクラスや実装クラスに任せます。

5.3.15 OOPL の特性を活かした実装

デザインパターンは優れた例です。

5.4 Java を始める

5.4.1 統合開発環境 IDE をインストールする

以前は次の手順でプログラミングを行いました。

- (1) JDK をダウンロードする
- (2) IDE をインストールする
- (3) 環境変数を設定する

ここまでは初回のみ。以下は開発サイクルの中で繰り返されます。

- (4) テキストエディタで .java ファイルを作成する
- (5) javac コマンドでコンパイルして .class (バイトコード) ファイルを作成する
- (6) java コマンドで .class ファイルを起動する

※現在は特別な理由がない限り、IDE を使ってプログラミングします。

Preiades all-in-one や IntelliJ などを使用します。

5.4.2 サンプルコードを動かす。

サンプルコードを真似てコードを書き、動かしてみます。

```
package oop.chapt5;

public class HelloWorld {
    public static void main (String[] args) {
        System.out.println("HelloWorld");
    }
}
```

5.5 基本型と参照型

```
package oop.chapt5;

/**
 * 基本型 (プリミティブ型) と参照型 (後述) とリテラル
 */
public class Sample1 {

    /** 基本型 論理値 リテラル true, false*/
    boolean bl = true;

    /** 基本型 16 ビット Unicode 文字 */
    char c = 'A';
```

```

/** 基本型 8ビット符号付整数 */
byte b = 10;

/** 基本型 16ビット符号付整数 */
short s = 20;

/** 基本型 32ビット符号付整数 */
int i = 30;

/** 基本型 64ビット符号付整数 */
long l = 40;

/** 基本型 32ビット浮動小数点数 */
float f = 10.0f;

/** 基本型 64ビット浮動小数点数 */
double d = 10.0;          // = 1.0e1, = 0.1E2

/** 参照型 リテラル null */
Sample1 sample1 = null;
double d2 = 3.1412; //数値リテラル
double d3 = 3.1412; //数値リテラル
}

```

5.6 変数宣言

```

package oop.chapt5;

/**
 * 変数宣言
 */
public class Sample2 {

    /** フィールド変数（インスタンス変数かクラス変数（後述）） */
    int value = 0;

    /** フィールド変数（インスタンス変数かクラス変数（後述）） */
    final String name = "初期値";    //final 変数

    /** フィールド変数（インスタンス変数かクラス変数） */
    int[] ia = new int[5];    //配列変数

    /**
     * メソッド
     */
    public void method(String param) { //パラメータ

        int i = 0;    //ローカル変数（プリミティブ型か参照型）

    }
}

```

5.7 演算子

```
package oop.chapt5;

/**
 * 演算子
 */
public class Sample3 {

    public void method() {

        int i = 0;
        int j = ++i; //インクリメント演算子
        int k = --i; //デクリメント演算子

        if (i > j) {} //関係演算子（大なり）
        if (i >= j) {} //関係演算子（以上）
        if (i < j) {} //関係演算子（小なり）
        if (i <= j) {} //関係演算子（以下）
        if (i == j) {} //関係演算子（等しい）
        if (i != j) {} //関係演算子（等しくない）

        if (!(i > j)) {} //論理否定
        if ((i > j) & (i > k)) {} //論理積（AND）
        if ((i > j) | (i > k)) {} //論理和（OR）
        if ((i > j) ^ (i > k)) {} //排他的論理和（XOR）
        if ((i > j) && (i > k)) {} //条件積（左側が先に評価され必要な場合のみ次が評価される）
        if ((i > j) || (i > k)) {} //条件和（左側が先に評価され必要な場合のみ次が評価される）

        int a = 0xF00F;
        int b = 0x0FF0;
        int c = 0xAAAA;

        int d = a & b; //0x0000 二項ビット演算子（ビット積 AND）
        int e = a | b; //0xFFFF 二項ビット演算子（ビット和 OR）
        int f = c ^ e; //0x5555 二項ビット演算子（排他的ビット和 XOR）

        int x = a << 2; //2 ビット左シフトで右側をゼロで埋める
        int y = b >> 2; //2 ビット右シフトで左側を符号ビット（最上位）で埋める
        int z = b >>> 2; //2 ビット右シフトで左側をゼロで埋める

        //instanceof 演算子
        String s1 = "ABC";
        if (s1 instanceof String) {
            System.out.println("s1 は String クラスのインスタンスです");
        }
    }
}
```

```

//条件演算子 ?:
x = ((a < b) ? a : b);
/*
    if (a < b) {
        x = a;
    } else {
        x = b;
    }
*/

//代入演算子
x = 1;
x = y = z = 1;
x += 1;          //x = x + 1;
x *= 2;          //x = x * 2;

//文字列結合演算子
String s2 = "DEF";
String s3 = s1 + s2; //"ABCDEF"
s3 += s2;          //s3 = s3 + s2;

//new 演算子
String s4 = new String("GHI");

}
}

```

5.8 制御フロー文

```

package oop.chapt5;

import java.util.ArrayList;

/**
 * 制御フロー文
 */
public class Sample4 {

    public int method() {

        int a = 1;
        int b = 2;

        //if 文
        if (a < b) { //条件式
            //真の場合
        } else {
            //偽の場合
        }

        //switch 文
        switch (a) { //a は整数式
            case 2:

```

```

        //a がラベル(整数定数 2)と等しい場合の処理

        break;    //ある場合は次のラベルの評価をしない。switch を終了する
    case 3:
        //a がラベル(整数定数 3)と等しい場合の処理
    default:
        //a がどのラベルとも違う場合の処理
    }

//while 文
while (a < b) {
    //繰り返す処理
}

//do-while 文
do {
    //繰り返す処理
} while (a < b);

//for 文
for (int i = 0; i < a; i++) {
    //繰り返す処理
}

//拡張 for 文
ArrayList<String> list1 = new ArrayList<>();
for(String s : list1) {
    System.out.println(s);
}

//Stream#forEach メソッド
list1.stream().forEach(s -> System.out.println(s));

//break 文
for (int i = 0; i < a; i++) {
    //繰り返す処理
    if (i == b) {
        break;    //ループを終了する
    }
    //繰り返す処理
}

//continue 文
for (int i = 0; i < a; i++) {
    //繰り返す処理
    if (i == b) {
        continue; //ループ本体の終りに制御を移し次にループ式を評価する
    }
    //繰り返す処理
}

```

```

        //return 文
        return a;
    }
}

```

5.9 列挙型 enum

****列挙型 (enum) ****は、Java で特定の値の集合を定義するために使用されます。

例えば、曜日、季節、カードのスートなど、限られた数の値を表現したいときに使います。列挙型は、定数のグループを簡単に作成でき、コードの可読性と保守性を向上させます。

例

```

public enum Day {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY
}

public class EnumExample {
    public static void main(String[] args) {
        Day today = Day.SATURDAY;
        System.out.println("Today is: " + today);
    }
}

```

5.10 匿名クラス

匿名クラスは、一度きりしか使わないクラスを定義するための方法です。

通常、匿名クラスはインターフェースや抽象クラスのインスタンスを作成するために使われます。

クラスの名前をつけずに、その場でクラスを定義してインスタンス化できます。

例

```

public class AnonymousClassExample {
    public static void main(String[] args) {
        // Runnable インターフェースを匿名クラスで実装
        Runnable runnable = new Runnable() {
            @Override
            public void run() {
                System.out.println("Anonymous class running");
            }
        };

        // 新しいスレッドを開始
        new Thread(runnable).start();
    }
}

```

この例では、Runnable インターフェースを匿名クラスで実装し、スレッドを開始しています。

匿名クラスを使うことで、簡潔にインターフェースを実装することができます。

5.11 this と super

this と super は、Java のキーワードで、それぞれ現在のオブジェクトや親クラス（スーパークラス）への参照を表します。

5.11.1 this

this は、現在のオブジェクトを指すキーワードです。クラスのインスタンスメソッドやコンストラクタの中で使われます。

特に、インスタンス変数とローカル変数の名前が同じ場合に、インスタンス変数を明示的に参照するために使います。

```
public class ThisExample {
    private int number;

    public ThisExample(int number) {
        this.number = number; // インスタンス変数を指す
    }

    public void printNumber() {
        System.out.println(this.number); // インスタンス変数を指す
    }

    public static void main(String[] args) {
        ThisExample example = new ThisExample(10);
        example.printNumber(); // 出力: 10
    }
}
```

5.11.2 super

super は、親クラス（スーパークラス）のメンバー（フィールドやメソッド）にアクセスするために使います。親クラスのコンストラクタを呼び出すときにも使用されます。

```
class Animal {
    public void makeSound() {
        System.out.println("Animal sound");
    }
}

class Dog extends Animal {
    @Override
    public void makeSound() {
        super.makeSound(); // 親クラスのメソッドを呼び出す
        System.out.println("Dog barks");
    }
}

public class SuperExample {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.makeSound(); // 出力: Animal sound
                        //      Dog barks
    }
}
```

5.12 ボクシングとアンボクシング

ボクシングとアンボクシングは、プリミティブ型と対応するラッパークラス（オブジェクト型）の間で変換するプロセスです。

5.12.1 ボクシング

プリミティブ型（int、char、boolean など）を対応するラッパークラス（Integer、Character、Boolean など）のオブジェク

トに変換することをボックスングといいます。

```
int num = 10;
Integer boxedNum = num; // ボックスング
```

5.12.2 アンボックスング

ラッパークラスのオブジェクトをプリミティブ型に変換することをアンボックスングといいます。

```
Integer boxedNum = 10;
int num = boxedNum; // アンボックスング
```

5.13 javadoc

javadoc は、Java ソースコードに埋め込まれたコメントを使って、HTML 形式の API ドキュメントを自動生成するツールです。開発者は、クラス、メソッド、フィールドなどの詳細をコメントとして記述し、javadoc を使って読みやすいドキュメントを生成します。

例

```
/**
 * このクラスは例のためのクラスです。
 * @author ChatGPT
 */
public class JavadocExample {

    /**
     * これはメインメソッドです。
     * @param args コマンドライン引数
     */
    public static void main(String[] args) {
        System.out.println("Hello, javadoc!");
    }

    /**
     * 数値を加算するメソッドです。
     * @param a 一つ目の数値
     * @param b 二つ目の数値
     * @return 加算結果
     */
    public int add(int a, int b) {
        return a + b;
    }
}
```

これらのコメントを使って、javadoc ツールはクラスやメソッドの詳細なドキュメントを生成します。

5.14 ジェネリクス（総称型）

ジェネリクス（総称型）は、クラスやメソッドが扱うデータの型をパラメータ化する仕組みです。ジェネリクスを使うことで、型安全性を高め、再利用性の高いコードを書くことができます。例えば、リストやマップのようなコレクションに対して、特定の型の要素だけを格納できるようにするために使用されます。

例

```
import java.util.ArrayList;
import java.util.List;

public class GenericsExample {
    public static void main(String[] args) {
        List<String> stringList = new ArrayList<>(); // String 型のリスト
        stringList.add("Hello");
        stringList.add("Generics");

        for (String s : stringList) {
            System.out.println(s);
        }
    }
}
```

5.15 コレクション

コレクションは、データのグループを管理するためのクラスのセットです。代表的なコレクションには、リスト、セット、マップなどがあります。コレクションを使うことで、データの追加、削除、検索などの操作を簡単に行うことができます。

例

```
import java.util.ArrayList;
import java.util.HashSet;
import java.util.HashMap;
import java.util.List;
import java.util.Set;
import java.util.Map;

public class CollectionExample {
    public static void main(String[] args) {
        // リスト
        List<String> list = new ArrayList<>();
        list.add("Apple");
        list.add("Banana");
        System.out.println("List: " + list);

        // セット
        Set<String> set = new HashSet<>();
        set.add("Apple");
        set.add("Banana");
        set.add("Apple"); // 重複は無視される
        System.out.println("Set: " + set);

        // マップ
        Map<Integer, String> map = new HashMap<>();
        map.put(1, "One");
        map.put(2, "Two");
        System.out.println("Map: " + map);
    }
}
```

5.16 ストリーム

ストリームは、Java 8 で導入されたコレクションの操作を効率的に行うための API です。ストリームを使うことで、データのフィルタリング、マッピング、集計などの操作を簡潔なコードで記述できます。ストリームは一度だけ使えるデータ

の流れで、元のコレクションを変更しません。

例

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class StreamExample {
    public static void main(String[] args) {
        List<String> list = Arrays.asList("Apple", "Banana", "Cherry", "Date");

        // フィルタリングと変換
        List<String> result = list.stream()
                                .filter(s -> s.startsWith("A"))
                                .map(String::toUpperCase)
                                .collect(Collectors.toList());

        System.out.println(result); // 出力: [APPLE]
    }
}
```

5.17 ラムダ式

ラムダ式は、Java 8 で導入された匿名関数の一種で、簡潔に関数を記述する方法です。特に、コレクションの操作やストリームの処理でよく使われます。ラムダ式を使うことで、コードの可読性が向上し、冗長なコードを避けることができます。

例

```
import java.util.Arrays;
import java.util.List;

public class LambdaExample {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

        // ラムダ式を使ってリストの要素を 2 倍にして出力
        numbers.forEach(n -> System.out.println(n * 2));
    }
}
```

これらの概念を理解することで、Java 8 の機能を活用して、より効率的で読みやすいコードを書くことができるようになります。

5.18 オブジェクト

分析・設計によって抽出されたオブジェクト。(人、顧客、口座)

オブジェクト指向で設計・実装されたシステムは、オブジェクト同士が相互作用（メッセージを送る、メソッドを呼び出す）ことでシステム要件を実現します。

5.18.1 クラスとオブジェクト(インスタンス)の関係

例

銀行口座クラス

A さんの銀行口座オブジェクト

Bさんの銀行口座オブジェクト

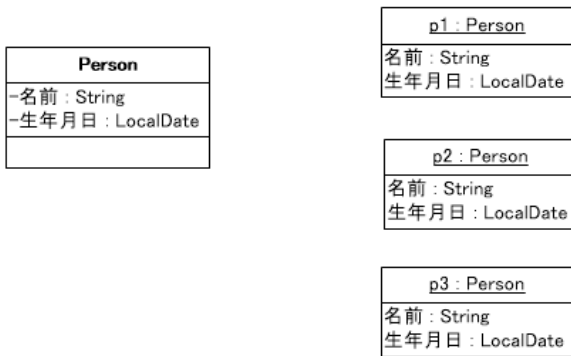
Aさんの銀行口座インスタンス：実装寄りの文脈で使用されます

Bさんの銀行口座インスタンス

5.18.2 オブジェクトの同一性と同値性

例

Person クラスとそのインスタンスが3つあります。(次図 UML 参照)



同じ名前 AND 同じ生年月日でも、インスタンスが別であれば、同一ではありません。

同じ名前 AND 同じ生年月日なので、2つのインスタンスは同値です。

以下のコードで確認できます。

```

package oop.chapt5;

import java.time.LocalDate;
import java.util.Objects;

public class Person {

    private String name;
    private LocalDate birthDay;

    public Person(String name, LocalDate birthDay) {
        this.name = name;
        this.birthDay = birthDay;
    }

    @Override
    public int hashCode() {
        return Objects.hash(birthDay, name);
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        Person other = (Person) obj;
  
```

```

    return Objects.equals(birthday, other.birthday) && Objects.equals(name, other.name);
}

//テスト開始
public static void main(String[] args) {
    Person p1 = new Person("令和太郎", LocalDate.of(2000, 1, 1));
    Person p2 = new Person("令和太郎", LocalDate.of(2000, 1, 1));

    if (p1 == p2) {
        //実行されません
    } else {
        System.out.println("同名・同生年月日でも同一ではありません");
    }
    if (p1.equals(p2)) {
        System.out.println("同名・同生年月日なら同値です");
    } else {
        //実行されません
    }

    Person p3 = p1;
    if (p1 == p3) {
        System.out.println("同一です");
    }

    Integer integer1 = new Integer(1);
    Integer integer2 = new Integer(1);

    if (integer1 == integer2) {
        //実行されません
    } else {
        System.out.println("同じ整数値でも同一ではありません。別のインスタンスです");
    }
    if (integer1.equals(integer2)) {
        System.out.println("同じ整数値なら同値です");
    } else {
        //実行されません
    }

}
}

```

ここでは同値性の判定に名前と生年月日を使いましたが、

もし Person クラスのインスタンスの同値性の判定に生年月日を使用したくない、というような場合には、

Person#equals メソッドを変更します。そこでどの属性を使うかを指定できます。上のサンプルコードの equals メソッドを確認してください。

5.19 クラス

5.19.1 クラスを定義する

オブジェクトを生成するための雛形としてクラスを定義します。

Java はインスタンスを生成するためにクラス定義から `new` するクラスベースのプログラミング言語です。プロトタイプベースの言語などもあります。

(クラス定義の主要な要素)

- クラス名
- フィールド
- メソッド
- コンストラクタ

【練習】ここで UML クラス図を書いてみましょう。

5.19.2 アクセス制御

- `public` 全てのクラスからのアクセスを許します。
- `protected` サブクラス、同じパッケージ内からのアクセスを許します。
- `private` そのクラス以外からのアクセスを許しません。(データの隠蔽)
- 指定なし アクセス修飾子無しの場合、同じパッケージ内のコードからのみアクセス出来ます。

(注) クラス単位であってインスタンス単位ではありません。ちなみにインスタンス単位の言語もあります。

5.19.3 オブジェクトの生成

同一性と同値性のサンプルコードに含まれていた

```
Person p1 = new Person("令和太郎", LocalDate.of(2000, 1, 1));
```

や

```
Integer integer1 = new Integer(1);
```

です。

5.19.4 コンストラクタ

新しく生成されたオブジェクトは初期値を持ちます。

フィールド(属性、プロパティ)は、タイプに応じて `0`、`false`、`null` に単純に初期化されます。

これ以外の初期化を行う場合は、コンストラクタの中に初期化のコードを書きます。

コンストラクタはクラス名と同じ名前を持ちます。

メソッドではないので戻り値を持ちません。

引数を持たないコンストラクタはデフォルトコンストラクタと呼びます。

複数のコンストラクタを持つことができます。(オーバーロード、後述)

コンストラクタを持たないクラスには、Java 言語がなにもしないパラメータなしのコンストラクタを用意します。

例

```
public Person(String name, LocalDate birthDate) {
    super();
    this.name = name;
    this.birthDate = birthDate;
}
```

5.19.5 メソッド

分析、設計モデルの操作に相当します。

例

```
package oop.chapt5_2;

import java.time.LocalDate;
import java.time.Period;

public class Person {

    private String name;
    private LocalDate birthDay;

    public Person(String name, LocalDate birthDay) {
        this.name = name;
        this.birthDay = birthDay;
    }

    //テスト開始
    public static void main(String[] args) {
        Person p1 = new Person("令和太郎", LocalDate.of(2000, 1, 1));

        System.out.println("現在の年齢は " + p1.getAge());
    }

    //自分の年齢を応答する
    public int getAge() {
        if ((this.birthDay != null)) {
            LocalDate currentDate = LocalDate.now();
            return Period.between(this.birthDay, currentDate).getYears();
        } else {
            return -1;
        }
    }
}
```

5.19.6 メソッドのオーバーロード

メソッドのシグネチャとはメソッド名、パラメータの数とタイプの組合せです。

シグネチャが異なれば、同じ名前のメソッドを複数定義できます。

下の例では、同じ操作名 `getAge` がオーバーロードされています。

```
package oop.chapt5_2;

import java.time.LocalDate;
import java.time.Period;

public class Person {

    private String name;
    private LocalDate birthDay;

    //コンストラクタ
    public Person(String name, LocalDate birthDay) {
        this.name = name;
        this.birthDay = birthDay;
    }
}
```



```

    }

    //テスト開始
    public static void main(String[] args) {
        Person p1 = new Person("令和太郎", LocalDate.of(2000, 1, 1));

        System.out.println("現在の年齢は " + p1.getAge());

        LocalDate targetDay = LocalDate.of(2030, 12, 31);
        System.out.println(targetDay + "時点の年齢は " + p1.getAge(targetDay));
    }

    //現在の年齢を応答する
    public int getAge() {
        if (this.birthDay != null) {
            LocalDate currentDate = LocalDate.now();
            return Period.between(this.birthDay, currentDate).getYears();
        } else {
            return -1;
        }
    }

    //指定された年月日時点の年齢を応答する
    public int getAge(LocalDate targetDay) {
        if ((this.birthDay != null) && (targetDay != null)) {
            return Period.between(this.birthDay, targetDay).getYears();
        } else {
            return -1;
        }
    }
}

```

コンストラクタもオーバーロードできます。

5.19.7 クラスメンバーとインスタンスメンバー

- クラスメンバー

クラス変数とクラスメソッドのことです。

- クラス変数（static 変数）

そのクラスに対して 1 個だけ存在します。

インスタンスが 1 つも生成されていない状態でも存在します。

- クラスメソッド

static メソッド。

そのクラスに対して唯一つ存在します。

メソッドへのアクセスはオブジェクト参照ではなく、クラス名を使います。

オブジェクト参照ではないので、this は使用できません。

- インスタンスメンバー

インスタンス変数とインスタンスメソッドのことです。

個々のインスタンス毎に存在します。

例

```
package oop.chapt5_3;

import java.math.BigDecimal;

public class 普通預金口座 {

    //クラス変数
    private static double 利率 = 0.01;

    //インスタンス変数
    private BigDecimal 残高;
    private String 口座番号;
    private String 名義名;

    //クラスメソッド
    public static double get 利率() {
        return 利率;
    }

    //インスタンスメソッド
    public BigDecimal get 残高() {
        return this.残高;
    }
}
```

5.19.8 ネイティブメソッド

java 言語以外で書かれたコードを使用するような場合、ネイティブメソッドを使うことができます。

(例) C 言語で作成したユーザ認証処理を、Java で作成するログイン処理から使用する。

JNI (Java Native Interface)

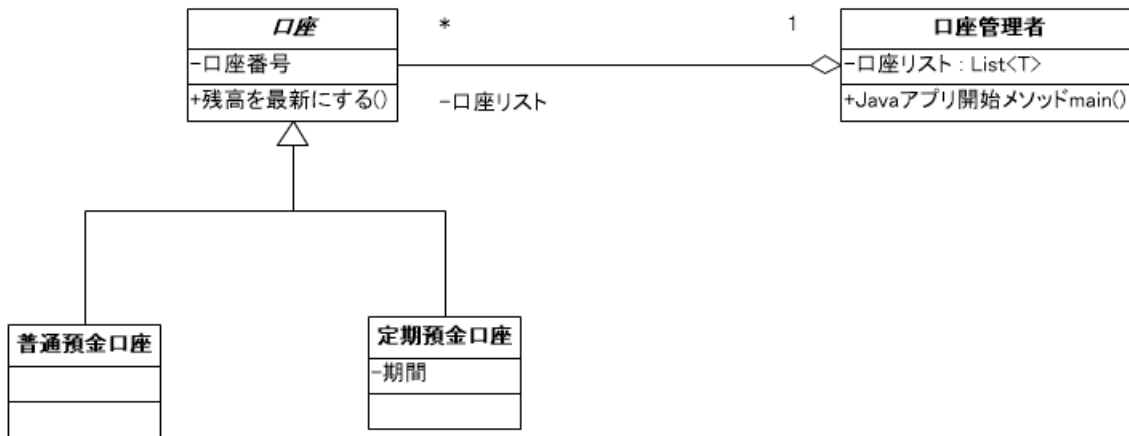
5.20 クラスの拡張

例 5.11

この例のドメインは、銀行口座の管理業務です。

管理できる口座はすべて同じ銀行の口座とします。つまり、同じネットバンキングサービスを利用します。

残高を参照するための API は普通預金用と定期預金用の 2 つがあります。



```

package oop.chapt5_6;
import java.math.BigDecimal;
/**
 * 口座
 */
public abstract class 口座 {

    private String 口座番号;
    private BigDecimal 残高;

    protected 口座(String 口座番号) {
        this.口座番号 = 口座番号;
        this.残高 = new BigDecimal(0);
        System.out.println("[INFO] 口座コンストラクタを実行 : 口座番号=" + 口座番号);
    }

    public String get 口座番号() {
        return 口座番号;
    }

    public void set 口座番号(String 口座番号) {
        this.口座番号 = 口座番号;
    }

    public BigDecimal get 残高() {
        return 残高;
    }

    public void set 残高(BigDecimal 残高) {
        this.残高 = 残高;
    }

    //抽象メソッド
    public abstract void 残高を最新にする();
}

```

```

package oop.chapt5_6;
import java.math.BigDecimal;
/**
 * 普通預金口座
 */
public class 普通預金口座 extends 口座 {

    public 普通預金口座(String 口座番号) {
        super(口座番号);
        System.out.println("[INFO] 普通預金口座コンストラクタを実行 : 口座番号=" + 口座番号);
    }

    //抽象メソッドの実装
    public void 残高を最新にする() {
        System.out.println("[INFO] ネットバンキング API で普通預金口座の残高を最新にしました。 : 口座番号=" + this.get
口座番号());
        BigDecimal 最新残高 = new BigDecimal(100); //ネットバンキング API で取得する
        this.set 残高(最新残高);
    }
}

package oop.chapt5_6;
import java.math.BigDecimal;
/**
 * 定期預金口座
 */
public class 定期預金口座 extends 口座 {

    private int 期間;

    public 定期預金口座(String 口座番号, int 期間) {
        super(口座番号);
        this.期間 = 期間;
        System.out.println("[INFO] 定期預金口座コンストラクタを実行 : 口座番号=" + 口座番号);
    }

    //抽象メソッドの実装
    public void 残高を最新にする() {
        System.out.println("[INFO] ネットバンキング API で定期預金口座の残高を最新にしました。 : 口座番号=" + this.get
口座番号());
        BigDecimal 最新残高 = new BigDecimal(100); //ネットバンキング API で取得する
        this.set 残高(最新残高);
    }

    public int get 期間() {
        return 期間;
    }
}

```

```

    public void set 期間(int 期間) {
        this.期間 = 期間;
    }
}

package oop.chapt5_6;
import java.util.ArrayList;
/**
 * 口座管理者
 */
public class 口座管理者 {

    private static ArrayList<口座> 口座リスト = new ArrayList<>();

    //Java アプリケーション開始メソッド
    public static void main(String[] args) {
        System.out.println("--- テスト開始 ---");
        口座管理者 管理者 = new 口座管理者();
        管理者.doTest();
        System.out.println("--- テスト終了 ---");
    }

    //テストする
    private void doTest() {
        System.out.println("--- この会社が所有しているすべての口座情報を追加します ---");
        this.add 口座(new 普通預金口座("1001"));
        this.add 口座(new 普通預金口座("1002"));
        this.add 口座(new 定期預金口座("8001", 3));

        System.out.println("--- すべての口座の残高を最新にします ---");
        for(口座 kouza : 口座リスト) {
            kouza.残高を最新にする();
        }
        //for 文ではなく Stream を使った場合
        口座リスト.stream().forEach(k -> k.残高を最新にする());
    }

    public void add 口座(口座 kouza) {
        口座リスト.add(kouza);
    }
}

```

--- テスト開始 ---

--- この会社が所有しているすべての口座情報を追加します ---

[INFO] 口座コンストラクタを実行：口座番号=1001

[INFO] 普通預金口座コンストラクタを実行：口座番号=1001

[INFO] 口座コンストラクタを実行：口座番号=1002

[INFO] 普通預金口座コンストラクタを実行：口座番号=1002

[INFO] 口座コンストラクタを実行：口座番号=8001

[INFO] 定期預金口座コンストラクタを実行：口座番号=8001

--- すべての口座の残高を最新にします ---

[INFO] ネットバンキング API で普通預金口座の残高を最新にしました。：口座番号=1001

[INFO] ネットバンキング API で普通預金口座の残高を最新にしました。：口座番号=1002

[INFO] ネットバンキング API で定期預金口座の残高を最新にしました。：口座番号=8001

[INFO] ネットバンキング API で普通預金口座の残高を最新にしました。：口座番号=1001

[INFO] ネットバンキング API で普通預金口座の残高を最新にしました。：口座番号=1002

[INFO] ネットバンキング API で定期預金口座の残高を最新にしました。：口座番号=8001

--- テスト終了 ---

ここから更新ポイント 2024-7-18

5.20.1 this と super

- this

非 static メソッドの中で、自分自身を指すオブジェクト参照です。

- super

非 static メソッドの中で、カレントオブジェクトをスーパークラスのインスタンスと見なしたときの参照です。

5.20.2 継承(拡張)とポリモルフィズム

口座クラスが使用できる場所では、そのサブクラスである普通預金口座も同様に扱えます。

拡張されたクラス（普通預金口座や定期預金口座）も元のクラス（口座）と同様に扱えます。

同様に扱えるとは、口座クラスと同じ責務を果たせるということです。

5.20.3 メソッドのオーバーライド

オーバーライドとは、サブクラスで親クラスのメソッドを再定義することです。

この際、オーバーライドするメソッドは親クラスのメソッドと同じ「メソッドのシグニチャ」（メソッド名、引数の数と

型、戻り値の型)を持つ必要があります。

これにより、サブクラスで同じ名前のメソッドを定義することで、そのメソッドが呼び出された際には、サブクラスの定義が優先されるようになります。

オーバーライドする際には戻り値の型も含めて「メソッドのシグニチャ」が完全に一致している必要があります。

つまり、メソッド名と引数の数と型、そして戻り値の型が親クラスのメソッドと完全に一致していなければなりません。

(注) static メソッドはオーバーライド出来ません。

(注) オーバーロードとは、同じクラス内またはサブクラスで、同じメソッド名で複数の異なるシグニチャ(メソッド名、引数の数や型)を持つことです。これにより、同じメソッド名で異なる引数を受け取るメソッドを複数定義することができます。オーバーロードされたメソッドは、引数の型や数が異なるため、コンパイラがどのメソッドを呼び出すかを解決します。

5.20.4 メソッド結合

- 動的メソッド結合(Dynamic method binding)

オブジェクト指向プログラミングの本質的な部分です。

例 5.11 の口座管理者クラスの main メソッドの中に次のコードがあります。

実行すると、普通預金口座または定期預金口座の「残高を最新にする()」メソッドが呼び出されます(メソッド結合)

```
for(口座 kouza : 口座リスト){
    kouza.残高を最新にする();
}
```

ちなみに

Smalltalk の場合は常に動的メソッド結合を行います。

C++の場合は、virtual 関数(仮想関数)を使用した場合には、動的メソッド結合になります。

- 静的メソッド結合(Static method binding)

コンパイル時に呼出すメソッドを決定します。

C などの関数呼出しと同じになり、実行効率はあがりますが、プログラミングに制約が生じます。

5.20.5 同じフィールド名

スーパークラスと同じ名前のフィールドを定義すると、スーパークラスのフィールドは残りますが、単に名前を使っただけではアクセス出来なくなります。super などの参照を用いなければなりません。

<例>

```
package oop.chapt5_7;

/**
 * 最初にコマンドラインから始動されるクラス
 */
public class Starter {
    public static void main(String[] args) {
        ClassA ref_a;
        ClassB ref_b;
        System.out.println("ClassA のインスタンスを生成する");
        ref_a = new ClassA();
    }
}
```

```

        System.out.println ("public フィールド参照の場合、ref_a.same_name=" + ref_a.same_name);
        System.out.println ("メソッド参照の場合、ref_a.getSameName()=" + ref_a.getSameName());
        System.out.println ("メソッド参照の場合、ref_a.getSameName2()=" + ref_a.getSameName2());
        System.out.println ("¥nClassB のインスタンスを生成する");
        ref_b = new ClassB();
        System.out.println ("public フィールド参照の場合、ref_b.same_name=" + ref_b.same_name);
        System.out.println ("メソッド参照の場合、ref_b.getSameName()=" + ref_b.getSameName());
        System.out.println ("メソッド参照の場合、ref_b.getSameName2()=" + ref_b.getSameName2());
        System.out.println ("¥nref_a に ref_b を代入する(キャストする)");
        if (ref_b instanceof ClassB) {
            ref_a = ref_b;
        }
        System.out.println ("public フィールド参照の場合、ref_a.same_name=" + ref_a.same_name);
        System.out.println ("メソッド参照の場合、ref_a.getSameName()=" + ref_a.getSameName());
        System.out.println ("メソッド参照の場合、ref_a.getSameName2()=" + ref_a.getSameName2());
    }
}

/**
 * スーパークラス ClassA
 */
class ClassA extends Object {
    String same_name;
    public String getSameName() {
        return (same_name);
    }
    public String getSameName2() {
        return (same_name);
    }
    public ClassA() {
        same_name = "A";
        System.out.println ("コンストラクタ ClassA()が実行された");
    }
}

/**
 * ClassA のサブクラス ClassB
 */
class ClassB extends ClassA {
    String same_name;
    public String getSameName2() { //オーバーライドしたメソッド
        return (same_name);
    }
    public ClassB() {
        same_name = "B";
        System.out.println ("コンストラクタ ClassB()が実行された");
        System.out.println ("コンストラクタ ClassB()中から same_name=" + same_name);
        System.out.println ("コンストラクタ ClassB()中から super.same_name=" + super.same_name);
    }
}

```


<実行結果>

ClassB のインスタンスを生成する
 コンストラクタ ClassA()が実行された
 コンストラクタ ClassB()が実行された
 コンストラクタ ClassB()中から same_name=B
 コンストラクタ ClassB()中から super.same_name=A
 public フィールド参照の場合、ref_b.same_name=B
 メソッド参照の場合、ref_b.getSameName()=A
 メソッド参照の場合、ref_b.getSameName2()=B

ref_a に ref_b を代入する(キャストする)
 public フィールド参照の場合、ref_a.same_name=A
 メソッド参照の場合、ref_a.getSameName()=A
 メソッド参照の場合、ref_a.getSameName2()=B

<実行結果はここまで>

既存のスーパークラス（ClassA）の実装者が、（既に開発済みの）サブクラスを破壊せずに新しい public、protected フィールドを追加できるように、このような仕様になっているようです。

メソッドの場合、参照のタイプではなく、実際のタイプで呼出されます。

フィールドの場合、実際のタイプではなく、参照の宣言タイプで決定されます。

アクセサメソッドによってアクセスするように設計すべきです。

5.20.6 タイプ変換

Java は強い型付けの言語です。（コンパイル時にほとんどの場合に対してタイプチェックが行われます）

- ワイディング、キャストアップ、安全なキャスト

<例>

```
class Starter {
    public static void main(String[] args) {
        ClassA ref_a;
        ClassB ref_b;
        ref_a = new ClassB(); //安全なキャスト
    }
}

class ClassA {
    public ClassA() {
        super();
    }
}

class ClassB extends ClassA {
    public ClassB() {
        super();
    }
}
```

- ナローイング、キャストダウン、安全ではないキャスト

ケース 1

```
class Starter {
    public static void main(String[] args) {
        ClassA ref_a;
        ClassB ref_b;

        ref_a = new ClassB();    //安全なキャスト
        ref_b = (ClassB)ref_a;    //ナローイング
    }
}
```

ケース 2

```
class Starter {
    public static void main(String[] args) {
        ClassA ref_a;
        ClassB ref_b;

        ref_a = new ClassB();    //安全なキャスト
        ref_b = (ClassB)ref_a;    //ナローイング
        ref_b = ref_a;
    }
}
```

<ケース 2 のコンパイル結果>

Starter.java:7: 互換性のない型

出現: ClassA

要求: ClassB

```
        ref_b = ref_a;    ^
```

エラー 1 個

ケース 3

```
class Starter {
    public static void main(String[] args) {
        ClassA ref_a;
        ClassB ref_b;
        ref_a = new ClassA();
        ref_b = (ClassB)ref_a;
    }
}
```

<ケース 3 の実行結果>

Exception in thread "main" java.lang.ClassCastException: ClassA
at Starter.main(Starter.java:6)

5.20.7 instanceof

```
class Starter {
    public static void main(String[] args) {
        ClassA ref_a;
        ClassB ref_b;
        ref_a = new ClassA();
        if (ref_a instanceof ClassB) {
```

```

        ref_b = (ClassB)ref_a;
    }
}
}

```

5.20.8 拡張したクラスのコンストラクタ

スーパークラスのコンストラクタを、新しいクラスの最初の実行文で呼出さない場合は、他の命令が実行される前に自動的にスーパークラスの引数なしコンストラクタが呼出されます。

ケース 1

```

public class Starter {
    public static void main(String[] args) {
        ClassA ref_a;
        ClassB ref_b;
        System.out.println ("ClassB のインスタンスを生成する");
        ref_b = new ClassB();
    }
}

class ClassA {
    public ClassA() {
        super();
        System.out.println ("コンストラクタ ClassA()が実行された");
    }
}

class ClassB extends ClassA {
    public ClassB() {
        super();
        System.out.println ("コンストラクタ ClassB()が実行された");
    }
}

```

<ケース 1 の実行結果>

```

ClassB のインスタンスを生成する
コンストラクタ ClassA()が実行された
コンストラクタ ClassB()が実行された

```

ケース 2

ケース 1 の ClassB のコンストラクタの 1 行目に `super();` はなくても実行結果は同じですが、デフォルトコンストラクタ(引数なしコンストラクタ) 以外を実行する必要がある場合は、必ず 1 行目に書く必要があります。 通常、デフォルトコンストラクタであっても 1 行目に `super();` を書くことが推奨されます。

```

public class Starter {
    public static void main(String[] args) {
        ClassA ref_a;
        ClassB ref_b;
        System.out.println ("ClassB のインスタンスを生成する");
        ref_b = new ClassB();
    }
}

class ClassA {

```

```

public ClassA() {
    System.out.println ("コンストラクタ ClassA()が実行された");
}
public ClassA(int i) {
    System.out.println ("コンストラクタ ClassA(int i)が実行された");
}
}

class ClassB extends ClassA {
    public ClassB() {
        super(1); //デフォルトコンストラクタ以外を実行する
        System.out.println ("コンストラクタ ClassB()が実行された");
    }
}

```

<ケース 2 の実行結果>

ClassB のインスタンスを生成する

コンストラクタ ClassA(int i)が実行された

コンストラクタ ClassB()が実行された

- this()

ケース 3

自分のコンストラクタを実行する。

```

class ClassA {
    public ClassA() {
        System.out.println ("コンストラクタ ClassA()が実行された");
    }
    public ClassA(int i) {
        this(); //自分のコンストラクタを実行する
        System.out.println ("コンストラクタ ClassA(int i)が実行された");
    }
}

```

ケース 3 の実行結果

ClassB のインスタンスを生成する

コンストラクタ ClassA()が実行された

コンストラクタ ClassA(int i)が実行された

コンストラクタ ClassB()が実行された

- super()

コンストラクタの最初に super()や super(…)がない場合は、super()が実行されます。もし、スーパークラスに引数なしコンストラクタが定義されていない場合は、コンパイルエラー「適合するコンストラクタがない」となります。

5.21 final 宣言

メソッドやクラスを final 宣言します。

final 宣言されたクラスのサブクラスは作れません。

final クラスのメソッドは暗黙的に final です。

final 宣言されたメソッドは、インライン展開など、最適化しやすくなります。

```
/**
 * 5 年定期預金口座
 */
class Fixed5YearsAccount extends FixedAccount {
    static final int 期間 = 5;
    public Fixed5YearsAccount() {
        super();
    }
}
```

5.22 Object クラス

全てのクラスは直接・間接に java.lang.Object クラスを拡張します。

```
public boolean equals (Object obj)
```

このオブジェクトと obj で参照されたオブジェクトの同値性を調べてみます。デフォルトの実装では、オブジェクトはそれ自身に対してのみ同値です。

```
public int hashCode()
```

オブジェクトのハッシュ値を返します。デフォルトの実装では異なるオブジェクトに対して、通常は、一意な値を返します。Hashtable オブジェクトにオブジェクトを格納するために使われます。

```
protected Object clone()
```

オブジェクトの複製を返します。 ※cloneable インタフェース。

```
public final Class getClass()
```

そのオブジェクトが属するクラスを Class タイプのオブジェクトによって返します。

```
protected void finalize() throws Throwable
```

ガーベッジコレクション時にオブジェクトを finalize します。

5.23 ラップクラス (Wrapper Class)

Java では数値や論理値を表わすのにクラスではなく int や boolean といった原始型 (Primitive Type) が存在します。

ラップクラスとして Integer や Boolean が用意されています。Integer クラスは文字列を数値に変換する、16 進表現を 10 進表現に変換する等の機能を提供します。これらの機能が必要でなければ、原始型を使用する場合があります。

5.24 Class クラス

全てのクラスとインタフェースは自らを表わす Class オブジェクトを持ちます。このオブジェクトはクラスやインタフェースに関する基本的な情報を調べたり、クラスの新しいオブジェクトを作るのに使われる Class クラスを使うことでプログラム中のタイプシステムをたどることができます。

関連 リフレクション

5.24.1 java.lang.Object クラスの getClass()メソッド

```
public final Class getClass()
```

オブジェクトの実行時クラスを返します。

5.24.2 Class クラスのメソッド例

- `getFields()`

この `Class` オブジェクトが表すクラスまたはインタフェースのすべてのアクセス可能な `public` フィールドをリフレクトし、`Field` オブジェクトを保持している配列を返します。

- `getMethods()`

この `Class` オブジェクトが表すクラスまたはインタフェースのすべての `public` メンバメソッドをリフレクトし `Method` オブジェクトを格納している配列を返します。

5.24.3 使用例

```
package oop.chapt5_8;
import java.lang.reflect.Method;
class Starter {
    public static void main(String[] args) {
        ClassA ref_a = new ClassA();
        Class c = ref_a.getClass();
        System.out.println("ref_a is an instance of " + c.getName());
        Method[] methods = c.getMethods();
        System.out.println("<method name list>");
        for (int i=0; i<methods.length; i++) {
            System.out.println("[ " + i + " ] " + methods[i].getName());
        }
    }
}

class ClassA {
    public ClassA() {
        super();
    }
    public void methodA() {
    }
}
```

<実行結果>

```
ref_a is an instance of oop.chapt5_8.ClassA
<method name list>
[0] methodA
[1] wait
[2] wait
[3] wait
[4] equals
[5] toString
[6] hashCode
[7] getClass
[8] notify
[9] notifyAll
```

5.25 抽象クラスと abstract メソッド

抽象クラスは `abstract` 宣言したクラスです。

クラスで実装されていないメソッドは `abstract` 宣言します。

1 つでも `abstract` メソッドがある場合、そのクラスは `abstract` 宣言する必要があります。

```
package oop.chapt5_6;
import java.math.BigDecimal;
/**
 * 口座
 */
public abstract class 口座 {
    private String 口座番号;
    private BigDecimal 残高;
    protected 口座(String 口座番号) {
        this.口座番号 = 口座番号;
        this.残高 = new BigDecimal(0);
        System.out.println("[INFO] 口座コンストラクタを実行 : 口座番号=" + 口座番号);
    }
    public String get 口座番号() {
        return 口座番号;
    }
    public void set 口座番号(String 口座番号) {
        this.口座番号 = 口座番号;
    }
    public BigDecimal get 残高() {
        return 残高;
    }
    public void set 残高(BigDecimal 残高) {
        this.残高 = 残高;
    }

    //抽象メソッド
    public abstract void 残高を最新にする();
}
```

5.26 インタフェース

クラスは設計と実装が混合しているのに対して、インタフェースは純粹に設計のみを表現する。

メソッドは全て暗黙的に `abstract` となる。(`abstract` は書かない)

メソッドは全て `public` である。

メソッドは `static` にはなり得ない。(`static` はクラスに対して定義できるもの)

フィールドは全て `static` かつ `final` である。(`static`、`final` は書かない)

複数のインタフェースを実装できるため、インタフェースは多重継承を実現できる。

抽象クラスはメソッドの一部が実装されていてもよい。また、`protected` や `static` メソッドも持てる。しかし、インタフェースは、`public` メソッドと定数以外は持てない。

インタフェース自身も `extends` により 2 つ以上のインタフェースから拡張できる。

<例>

```
import java.lang.reflect.*;
import java.util.*;
/**
 * 最初に起動されるクラス
 */
class Starter {
    public static void main(String[] args) {
        Radio radio = new Radio();
        Class c = radio.getClass();
        System.out.println("ref_a is an instance of " + c.getName());
        Method[] methods = c.getMethods();
        System.out.println("<method name list>");
        for (int i=0; i<methods.length; i++) {
            System.out.println("[ " + i + " ] " + methods[i].getName());
        }
        if (radio instanceof Radio) {
            System.out.println("radio は Radio クラスのインスタンスを参照している");
        }
        if (radio instanceof Recorder) {
            System.out.println("radio は Recorder クラスのインスタンスを参照している");
        }
        if (radio instanceof Clock) {
            System.out.println("radio は Clock クラスのインスタンスを参照している");
        }
    }
}

/**
 * Radio クラス
 * Recorder と Clock を実装する
 */
class Radio implements Recorder, Clock {
    public Radio() {
    }
    public void tune(int station) {
        //選局するためのコード
    }
    public void record() {
        //Recorder の record(録音する)を実装するためのコード
    }
    public void play() {
        //Recorder の play(再生する)を実装するためのコード
    }
    public void setTime(Date time) {
        //Clock の setTime(時刻を合わせる)を実装するためのコード
    }
    public Date getTime() {
        Date current = null;
        //Clock の getTime(時刻を見る)を実装するためのコード
        return current;
    }
}
```



```

/**
 * Recorder インタフェース
 */
interface Recorder {
    public void record();
    public void play();
}

/**
 * Clock インタフェース
 */
interface Clock {
    public void setTime(Date time);
    public Date getTime();
}

```

<実行結果>

ref_a is an instance of oop.chapt5_9.Radio

<method name list>

```

[0] setTime
[1] getTime
[2] tune
[3] record
[4] play
[5] wait
[6] wait
[7] wait
[8] equals
[9] toString
[10] hashCode
[11] getClass
[12] notify
[13] notifyAll

```

radio は Radio クラスのインスタンスを参照している

radio は Recorder クラスのインスタンスを参照している

radio は Clock クラスのインスタンスを参照している radio は Radio クラスのインスタンスを参照している

radio は Recorder クラスのインスタンスを参照している

radio は Clock クラスのインスタンスを参照している

5.26.1 インタフェースを使うとき

多重継承が必要な場合に使います。

※ Java は単一継承モデルです。従って、拡張できるスーパークラスは高々 1 つです。

5.27 例外クラス (Exception)

Java では、プログラムの実行中に発生するエラーや異常な状況を「例外 (Exception)」と呼びます。

例外が発生すると、通常のプログラムの流れが中断され、例外を処理するためのコードが実行されます。

Java には、例外を扱うための特別なクラスが用意されています。

5.27.1 主な例外クラス

- Exception クラス:

すべての例外の基本クラスです。これを継承してカスタム例外を作成することもできます。

通常、プログラムのロジック上で処理可能なエラーを表します。

- RuntimeException クラス:

Exception クラスを継承するクラスで、プログラムの実行時に発生する例外を表します。

例：NullPointerException, ArrayIndexOutOfBoundsException

チェックされない例外（Unchecked Exceptions）と呼ばれ、コンパイル時には検出されません。

- IOException クラス:

入出力操作に関するエラーを表します。

ファイルの読み書き中に発生するエラーなどが含まれます。

- SQLException クラス:

データベース操作中に発生するエラーを表します。

SQL クエリの実行中に発生する問題を扱います。

5.27.2 例外処理の基本構文

```
try {
    // エラーが発生する可能性のあるコード
} catch (ExceptionType e) {
    // 例外が発生した場合の処理
} finally {
    // 例外の発生に関係なく、必ず実行される処理
}
```

- try ブロック: エラーが発生する可能性のあるコードを囲みます。
- catch ブロック: 発生した例外をキャッチし、その処理を行います。
- finally ブロック: 例外の有無に関わらず、必ず実行されるコードを記述します（省

5.28 スレッド

スレッドとは、プログラムの実行単位のことです。Java では、マルチスレッドプログラミングをサポートしており、一つのプログラム内で複数のスレッドを実行することができます。これにより、同時に複数の処理を並行して行うことができ、プログラムの効率を向上させることができます。

Java でスレッドを使用する方法は主に 2 つあります。

- Thread クラスを拡張する方法
- Runnable インターフェースを実装する方法

Thread クラスを拡張する方法では、Thread クラスを継承して新しいクラスを作成し、その中で run メソッドをオーバーライドします。

一方、Runnable インターフェースを実装する方法では、Runnable インターフェースを実装したクラスを作成し、その中で run メソッドを定義します。

どちらの方法もスレッドの動作を定義するために run メソッドを使用します。

サンプルプログラム

1. Thread クラスを拡張する方法

```
class MyThread extends Thread {
    @Override
    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println("Thread: " + i);
            try {
                Thread.sleep(1000); // 1 秒間スリープ
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

public class Main {
    public static void main(String[] args) {
        MyThread thread = new MyThread();
        thread.start(); // スレッドの開始
    }
}
```

2. Runnable インターフェースを実装する方法

```
class MyRunnable implements Runnable {
    @Override
    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println("Runnable: " + i);
            try {
                Thread.sleep(1000); // 1 秒間スリープ
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

public class Main {
    public static void main(String[] args) {
        Thread thread = new Thread(new MyRunnable());
        thread.start(); // スレッドの開始
    }
}
```

これらのサンプルプログラムでは、新しいスレッドが作成され、そのスレッド内で run メソッドが実行されます。各スレッドは、0 から 4 までの数字を 1 秒ごとに出力します。Thread.sleep メソッドを使用してスレッドを指定した時間だけ一時停止させることができます。

5.29 パッケージ

Java のパッケージ (package) は、クラスやインターフェースを分類、整理するための仕組みです。

以下にパッケージの簡単な説明と使用例をいくつか挙げます。

5.29.1 パッケージの概要

- パッケージの定義:

パッケージは、関連するクラスやインターフェースをグループ化するための名前空間です。

パッケージ名は、通常ドメイン名を逆にした形式（例：com.example.project）で命名されます。

- パッケージの宣言

クラスファイルの最初にパッケージを宣言します。

例 `package com.example.project;`

- インポート

他のパッケージ内のクラスを使用するためには、そのパッケージをインポートする必要があります。

例 `import java.util.List;`

Java のパッケージはクラスやインターフェースを整理し、コードの再利用性を高め、名前の衝突を避けるために非常に重要です。標準パッケージを効果的に利用することで、様々な機能を簡単に実装できます。

- パッケージ名の命名規則

Java のパッケージ名にはいくつかの命名規則があります。主な規則として、すべて小文字で書かれることと、複数の単語はドットで区切ることが推奨されています。

以下の例を考えてみましょう。

例 `account_management`: アンダースコアを使うのは Java のパッケージ名として一般的ではありません。

例 `accountManagement`: キャメルケースもパッケージ名としては一般的ではありません。

例 `Accountmanagement`: 大文字を使うのは推奨されません。

最も適切な名前は以下のように、ドットで単語を区切り、すべて小文字にするスタイルです。

推奨される名前

`account.management`

6. オブジェクト指向技術の導入

<http://www.fk-nextdesign.sakura.ne.jp/ddd/index.html>

7. 永続化戦略

https://atmarkit.itmedia.co.jp/fdb/rensai/javapersis01/javapersis01_1.html

*** 終り ***