オブジェクト指向技術入門

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、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修得される場合には、その方法論の書籍も併せてご使用下さい。

構 成

オブジェクト指向技術とその目的

ソフトウエア開発では様々な方法論や言語、開発ツール等があり、新しい手法や製品が次々と出てきます。 一体、オブジェクト指向とはどこに位置し、どんなことが期待できるか？について理解します。

オブジェクト指向技術の概念

「クラス」や「オブジェクト」といったオブジェクト指向の基本概念を理解し、複数のオブジェクトと、それらの協調作用によってシステムをモデル化する、という考え方を習得します。 これらは以降の章でも繰返し出てきますので、繰返すことで理解を深めます。

オブジェクト指向システム開発

オブジェクト指向に適したシステム開発の流れを理解します。

統一モデリング言語 UML

オブジェクト指向開発でも広く使われているモデリング言語を習得します。 UML は、標準化団体 OMG によって策定されているもので、方法論毎に異なっていた記法が統一されました。

オブジェクト指向プログラミング言語 Java

オブジェクト指向言語としての特徴を理解します。また、UML から Java コードに、スムーズにマッピングできることを理解します。

オブジェクト指向技術の導入

導入時のチェックポイントを把握します。

目次

1. オブジェクト指向技術とその目的

本章では、ソフトウエ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 オブジェクト指向とプログラミング言語 Java

Java を使い始めるときには、以下の点に注意する必要があります。

・ Java はオブジェクト指向プログラミング言語です。
・ 従って、オブジェクト指向設計された結果（設計モデル）は、Java を使ってスムーズに実装できます。
・ もしも、オブジェクト指向設計されていなければ、Java を使うメリットはありません。この場合、Java を使うことで、従来よりも工数が増え、生産性が低下するケースがほとんどです。

1.2 Java で最初のプログラムを書く

以下は、よく知られている「Hello World」サンプルです。

1.2.1 C 言語の場合

プログラミング言語 C　[B.W.カーニハン/C.M.リッチー著 石田晴久訳より]

```
#include <stdio.h>
main()
{
        printf("hello, world¥n");
}
```

1.2.2 Java言語の場合

プログラミング言語 Java　[ケン･アーノルド/ジェームス･ゴスリン著 光澤敦訳より]

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

このサンプルを作成し実行することは簡単です。
そのため、次のような重要なポイントを見落としてしまうケースがあります。特に、オブジェクト指向言語以外でのプログラミング経験者は注意が必要です。

「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。問題は、"オブジェクト指向言語がもたらす利点を活かすこと" を習得するには時間がかかるという点です。オブジェクトの長所を活かすためには、パラダイムシフトを行わなければなりません。」[Martin Fowler 1997] より

1.2.3 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、ソフトウエ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。

1.2.4 キーワード class とは

HelloWorld サンプルコードの最初にある「class」は、Java 言語のキーワード（予約語）です。Java プログラミングでは頻繁に使用される最も重要なキーワードです。しかし、HelloWorld のようなサンプルではなく、自分自身で新たにプログラムを書き始めると、次のような疑問が起きるはずです。

・ class の次に書く名前は？
（因みに、「HelloWorld」という名前は良い例ではないと言われています）

Java の文法を学ぶだけでは、この答え（＝クラス名）を得ることはできません。クラス名の意図を理解したり、決めたりするためには、オブジェクト指向（思考）が必要です。

また、従来型の設計書の中にも（必要がないので）「クラス名」は登場しません。クラス名に相当するものもありません。もしも、従来型の設計書をもとに Java でプログラミングを行おうとすると、重要な情報が不足することになります。

1.3 ソフトウエア開発における位置付け

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
プロセス	手法の拡張、実践方法	手法の拡張、実践方法
ツール	CASE、データベース、 言語（Java、C#等）	CASE、データベース、 言語（C、COBOL 等）

1.3.1 アーキテクチャ

オブジェクト指向とは、ソフトウエ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

因みに、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

ードウエア・アーキテクチャ、CPU アーキテクチャなど。

1.3.2 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、ハードウエアの構造（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。

1.3.3 手法

あるアーキテクチャによるソフトウエアの開発手続きや手順です。

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エア工学 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

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

1.3.4 プロセス

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。

1.3.5 ツール

各アーキテクチャに基づいた（あるいは適した）CASE などの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。

1.3.6 アプローチ

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） 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。

（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。

（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。

1.4 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 Sysytem，X3J13 ANSI Lisp 標準化 G

1.4.1 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。

1.5 オブジェクト指向技術に期待できるもの

1.5.1 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、ソフトウエ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エ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。

1.5.2 パターン等の利用（設計、実装の定石）

ソフトウエ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、パターンの利用も急速に広まっています。

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）

オープンソース・ソフト

Java 言語　（2006.11 オープンソース化 GPL ライセンス）

このような再利用の考え方は従来からあるものですが、無償で提供されているオブジェクト群（フレームワーク）やデザインパターンなど、オブジェクト指向技術のもとでより成功していると言えるでしょう。

その要因の１つが、“オブジェクト同士の協調によってシステムを実現する”というオブジェクト指向の考え方です。他人が作成したオブジェクト（クラス）でも、理にかなったオブジェクトであれば、誰にとっても理解しやすく、取り入れやすいからだと言えます。

2. オブジェクト指向技術の概念

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。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、いくつか在る書棚の 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、あるメソッドを呼ぶことです。（あえて言えば、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

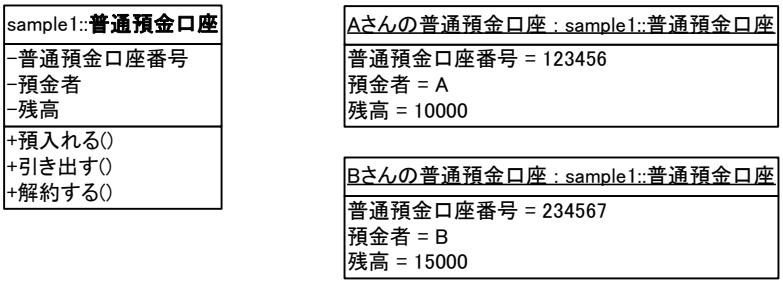
・定期預金口座オブジェクト
・普通預金口座番号オブジェクト
・支店オブジェクト
・預入れ取引きオブジェクト
・引き出し取引きオブジェクト

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エアで実現する必要があります。そのために Class-Based という考え方を導入します。

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、ソフトウエ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。



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。

```
/**
 * 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　koza = new 普通預金口座();
```

必要な性質はどれか？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エ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、完全に実現できないのであれば、必要な性質を識別し、クラスを定義する必要があります。

2.4 オブジェクトの追跡可能性

モデルとは、問題領域やソフトウエ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、ソフトウエ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。

2.7 CRC カード手法（Class, Resposibilities, 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。

2.7.1 CRC カード（6インチ×4インチ)(約15cm×10cm）

Class 普通預金口座	
Responsibilities 記帳する	Collaborators 印刷

2.7.2 Class

クラス名を書きます。

2.7.3 Responsibilities

属性やメソッドの代わりに、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)

2.7.4 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。

2.8 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。

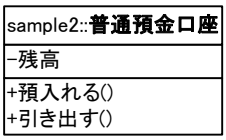
2.9 カプセル化

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2つの責務・振舞いです。

預入れる

引き出す




```
/**
 * 普通預金口座
 */
public class 普通預金口座 {
	private long 残高;
	public void 預入れる(long 預入れ額) {
		残高 = 残高 + 預入れ額;
	}
	public void 引き出す(long 引き出し額) {
		残高 = 残高 - 引き出し額;
	}
}
```

（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。

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もありません。

2.10 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。

2.10.1 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。

////////////////////////// ここまで　2005.11.20 ///////////////

2.10.2 属性を UML 表記する

[書式]　可視性　名前： タイプ = デフォルト値
[例]　- cardNumber : String = “”
可視性： +（パブリック)、#（プロテクテッド)、-（プライベート）
名前： 文字列
タイプ： 属性と同じ構文（String, int 等）
デフォルト値：　（オプション）

2.11 操作

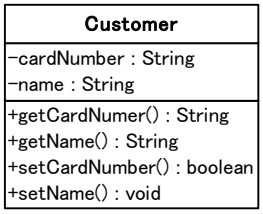
分析・設計レベルでは、オブジェクトの責務にほぼ対応し、実装レベルではメソッドに対応します。

2.11.1 操作を UML 表記する

[書式]　可視性　名前　（パラメータリスト）　：　戻り値のタイプ｛プロパティ文字列｝
[例]　+ getName () : String
可視性： +（パブリック)、#（プロテクテッド)、-（プライベート）
名前： 文字列
パラメータリスト： 属性と同じ構文（オプション）

戻り値のタイプ： 言語に依存した指定（オプション）
プロパティ文字列： 操作に使用するプロパティ値（オプション）

2.11.2 顧客クラスの UML 表記例



※　setter, getter メソッドは省略する場合が多い。

2.12 オブジェクト、クラス、インスタンス

2.12.1 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。

2.12.2 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。

目的や基準を間違うと、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、変更に弱く、保守しにくいモデルになることがあります。

2.12.3 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 p = new Customer(“murayama”,”C0001”);）」という分け方があります。

2.12.4 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。

2.12.5 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。

2.13 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／メラーよる例（すこし古典的）

2.13.1 有形物
人、商品、伝票

2.13.2 役割
人や構成により演じられる役割

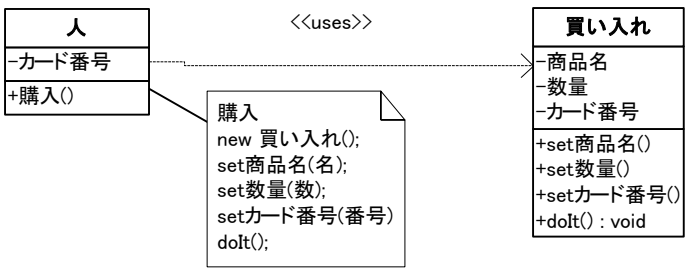
医師、患者、顧客、従業員

医師は患者にもなる

2.13.3 出来事
飛行、事故、故障、サービス要求

2.13.4 相互作用
買い入れ、結婚

2.13.5 仕様
製品などの仕様



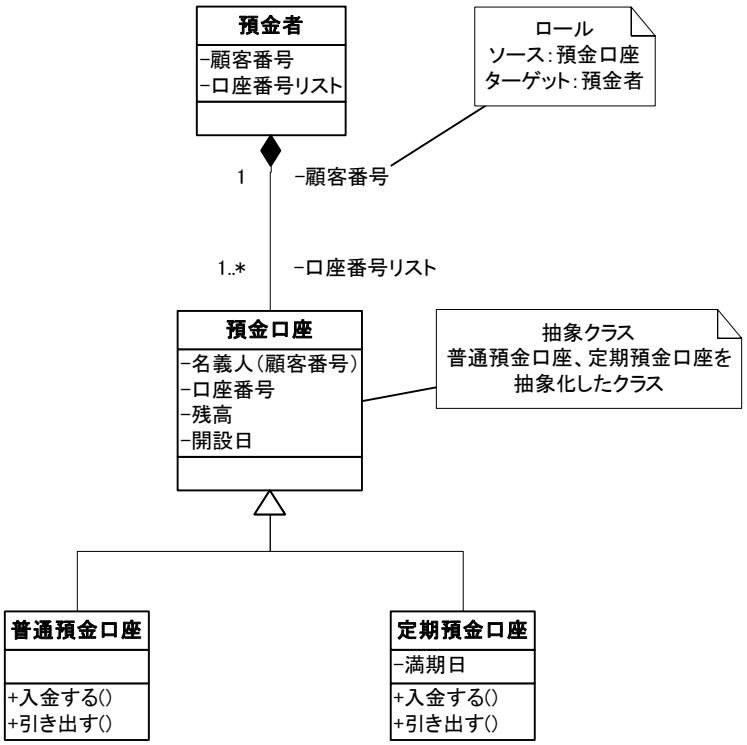
2.14 クラス図

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

2.14.1 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くてはなりません。

・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。

2.14.2 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）

2.14.3 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。）

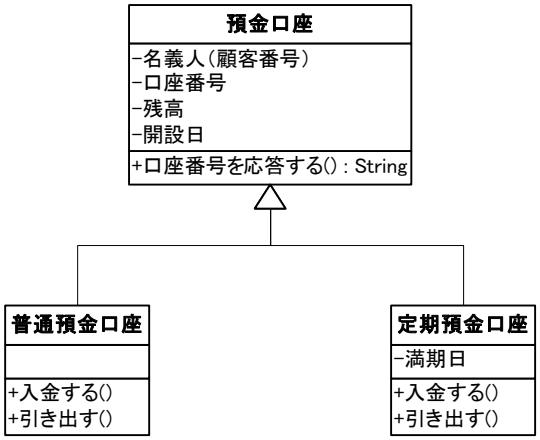
2.14.4 クラス図の例

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交差に共通の操作です。

「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）



2.14.5 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。
 * 以下のコードは、1 つのファイル"普通預金口座.java"に登録しています。
 * そのため、普通預金口座だけを public class としています。
 */
import java.util.Date;
/**
 * 預金口座クラス（抽象クラス）
 */
abstract class 預金口座 {
    private String 名義人;
    private String 口座番号;
    private long 残高;
    private Date 開設日;
    public 預金口座（String p_customer,
                    String p_account,
                    int p_amount,
```

```
                        Date p_open) {   //コンストラクタ
        名義人 = p_customer;
        口座番号 = p_account;
        残高 = p_amount;
        開設日 = p_open;
    }
    public String getAccountNumber (){   //口座番号を応答する
        return 口座番号;
    }
}
/**
 * 普通預金口座クラス
 */
public class 普通預金口座 extends 預金口座 {
    public 普通預金口座 (String p_customer,
                        String p_account,
                        int p_amount,
                        Date p_open) {   //コンストラクタ
        super(p_customer, p_account, p_amount, p_open);
        //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
    }
}
/**
 * 定期預金口座クラス
 */
class 定期預金口座 extends 預金口座 {
    private Date 満期日;    //定期預金だけの属性
    public 定期預金口座 (String p_customer,
                        String p_account,
                        int p_amount,
                        Date p_open,
                        Date p_expiration) {   //コンストラクタ
        super(p_customer, p_account, p_amount, p_open);
        満期日 = p_expiration;
    }
    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
        }
}
```

2.14.6 抽象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。

2.14.7 スーパークラスとサブクラス

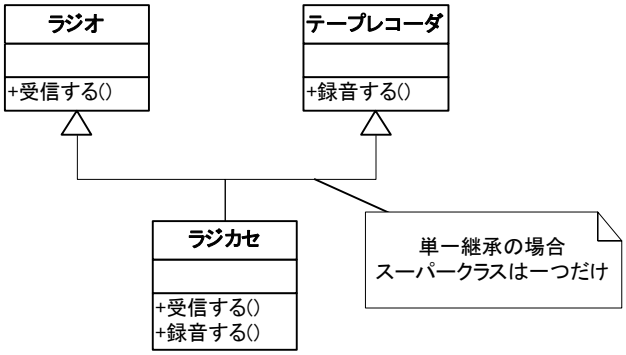
- スーパークラスは特殊化のもとになるクラス、サブクラスは特殊化された結果のクラスです。
- クラス階層をより分かりやすく、捉えやすくするためにクラス階層を作成することができます。
- 再利用しやすくするためにクラス階層を作成することができます。
- 差分プログラミングの効果を出すためにクラス階層を作成することができます。

2.14.8 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が、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えます。
- 委譲を使うべきかを検討します。

2.14.9 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。

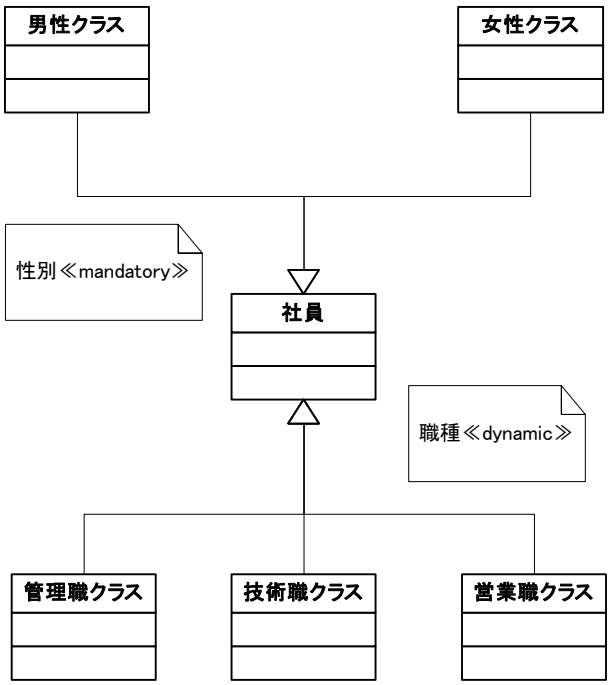


2.14.10 単一分類と多重分類

- 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。）

2.14.11 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）



2.14.12 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）

2.14.13 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。

2.14.14 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]

2.15 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、どの観点で作成するのか、どの観点で作成されたものかを常に認識しておく必要があります。

2.15.1 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エアをほとんど無視して作成されるべきものです。

2.15.2 仕様の観点

・ ソフトウエアの実装ではなく、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。

2.15.3 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。

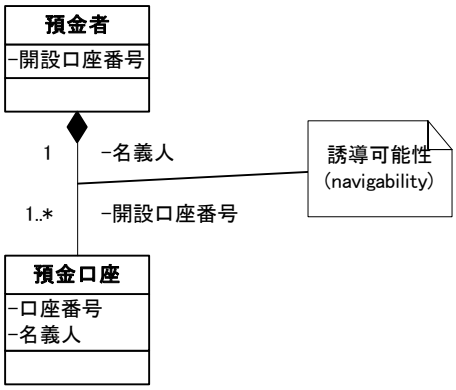
2.16 関連（インスタンス間の関係）

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。

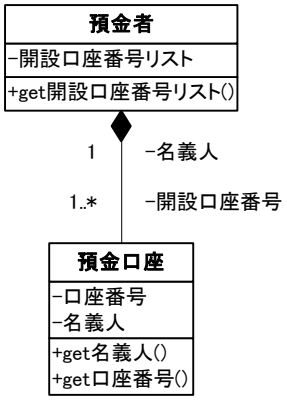
・ 関連に関与するオブジェクトの数を多重度で表わします。



2.16.2 仕様の観点

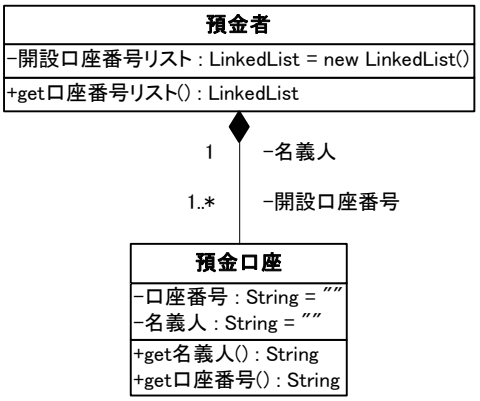
・ 関連はリスポンシビリティを表わします。

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。




```
/**
 * 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。
 * 以下のコードは、1 つのファイル"預金者.java"に登録しています。
 * そのため、預金者だけを public class としています。
 */
import java.util.LinkedList;
public class 預金者 {
    private LinkedList 開設口座リスト;
    public 預金者() {
        super();
        開設口座リスト = new LinkedList();
    }
    public LinkedList get開設口座リスト() {
        return 開設口座リスト;
    }
}
class 預金口座 {
    private String 口座番号;
    private String 名義人;
    public 預金口座() {
        super();
        口座番号 = "";
        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。

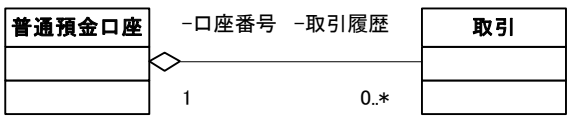
2.16.5 静的関連・動的関連

- 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

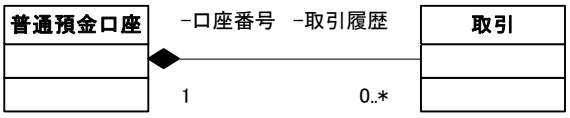
2.17.1 集約

- 全体オブジェクト－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、

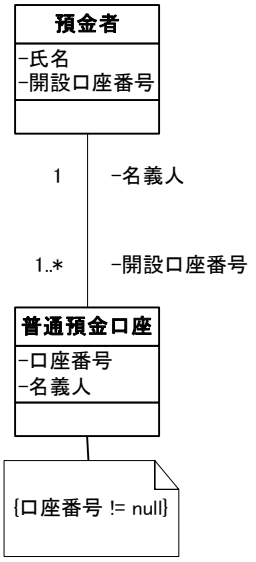
（1） 預金者は、０個以上の預金口座を持つこと。

（2） 預金口座は、唯一の名義人（預金者）が所有すること。

（3） 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

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

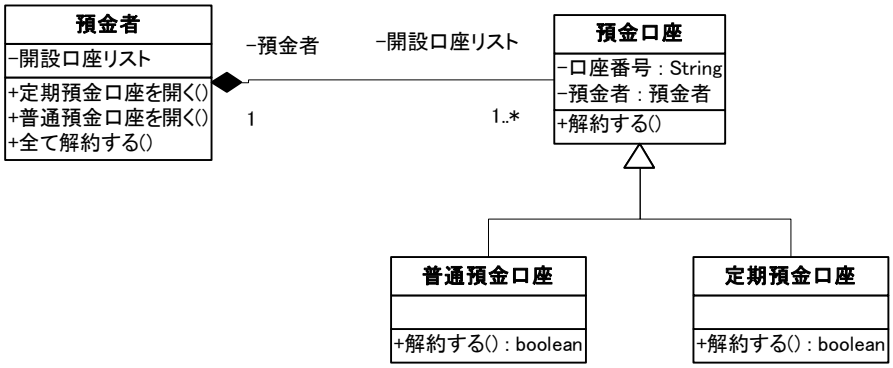
あるインスタンスが、別のインスタンスにメッセージを送る場合に、送り手のインスタンスが、受け手のインスタンスが属するクラスに関係なく、メッセージを送ることができ、異なるクラスに対しては、異なる振舞いをすること。 （受け手によってメソッド名を変えたり、パラメータを変えたりする必要がないこと）

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。




```
/**
 * 実際には
 * 各コンストラクタには引数(Account(Customer c)等)が必要と思われるが
 * この例題では省略する
 */
import java.util.*;
/**
 * 預金口座
 * 抽象クラスなのでインスタンスは存在しない
 * 抽象メソッド close() を持つ
 */
abstract class Account {
    private String accountNumber; // 口座番号
    private Customer customer;
    public Account() {
        super();
        accountNumber = "";
    }
    public abstract boolean close();
}

/**
 * 普通預金口座
 */
```

```
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;
	}
}

/**
 * 定期預金口座
 */
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;
	}
}

/**
 * 預金者
 */
public class Customer {
	private LinkedList accountList; // 開設口座リスト
	public Customer() {
		super();
		accountList = new LinkedList();
	}
	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() {
		Account a;
		int size = accountList.size();
		for (int i=0; i<size; i++) {
			a = (Account)accountList.get(i);
```

```
                // 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

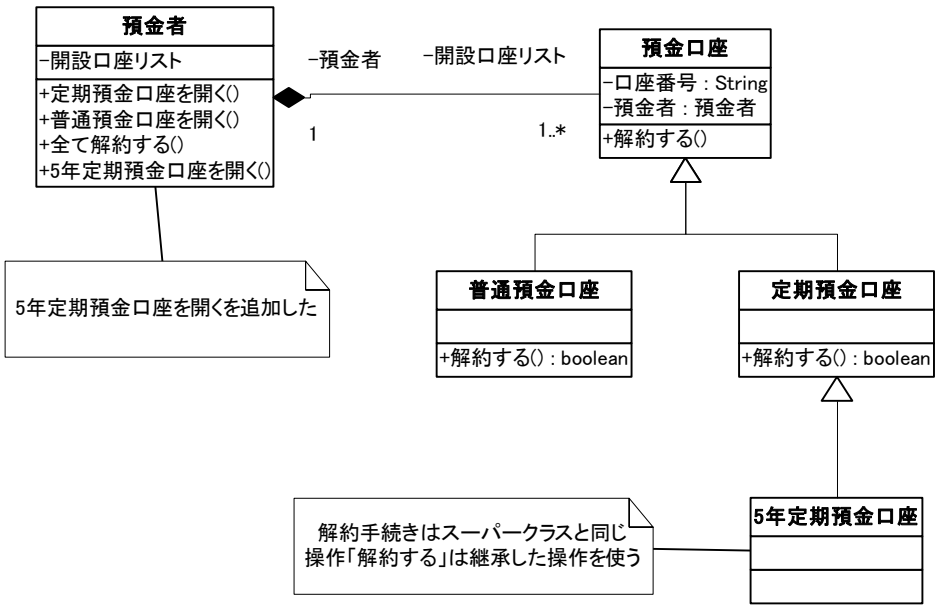
＜例＞

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？




```
/**
 * 預金者
 * openFixed5YearsAccount() を追加した
 */
public class Customer {
    private LinkedList accountList; // 開設口座リスト
    public Customer() {
        super();
        accountList = new LinkedList();
    }
    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() {
        Account a;
        int size = accountList.size();
        for (int i=0; i<size; i++) {
            a = (Account)accountList.get(i);
            // 開設済みの全ての預金口座(普通か定期)を解約する
            a.close();
        }
    }
}

/**
 * 5 年定期預金口座
 */
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する。

2.20.1 合成(has-a)

has-a と同様に方向を持つ関係である。

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

従来のシステム開発
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

ＳＡＤＴ（Structured Analysis and Design Technique）構造化分析/設計技法
ＲＤＤ（Requirement Driven Design base on SREM）ＳＲＥＭ要求駆動型設計
ＳＡ/ＳＤ（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エアアーキテクチャに起源する

ハードウエアは、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。
プログラマ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。
ハードウエア・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。

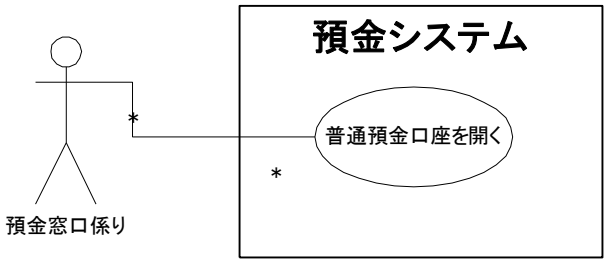
3.2 オブジェクト指向開発の特徴

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 19997.12 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

（1） 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）

（2） 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。

（3） 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。（ＯＯＳＥ）

（4） 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
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

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。

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。

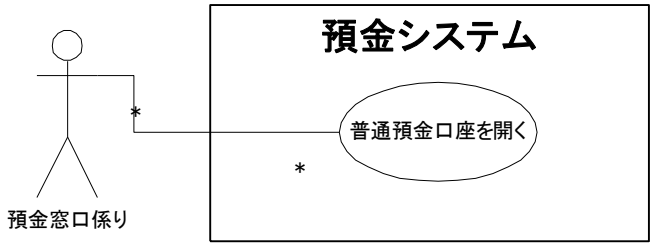
3.6.1 要求モデル（分析プロセスでは、要求モデルと分析モデルを作成する）

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

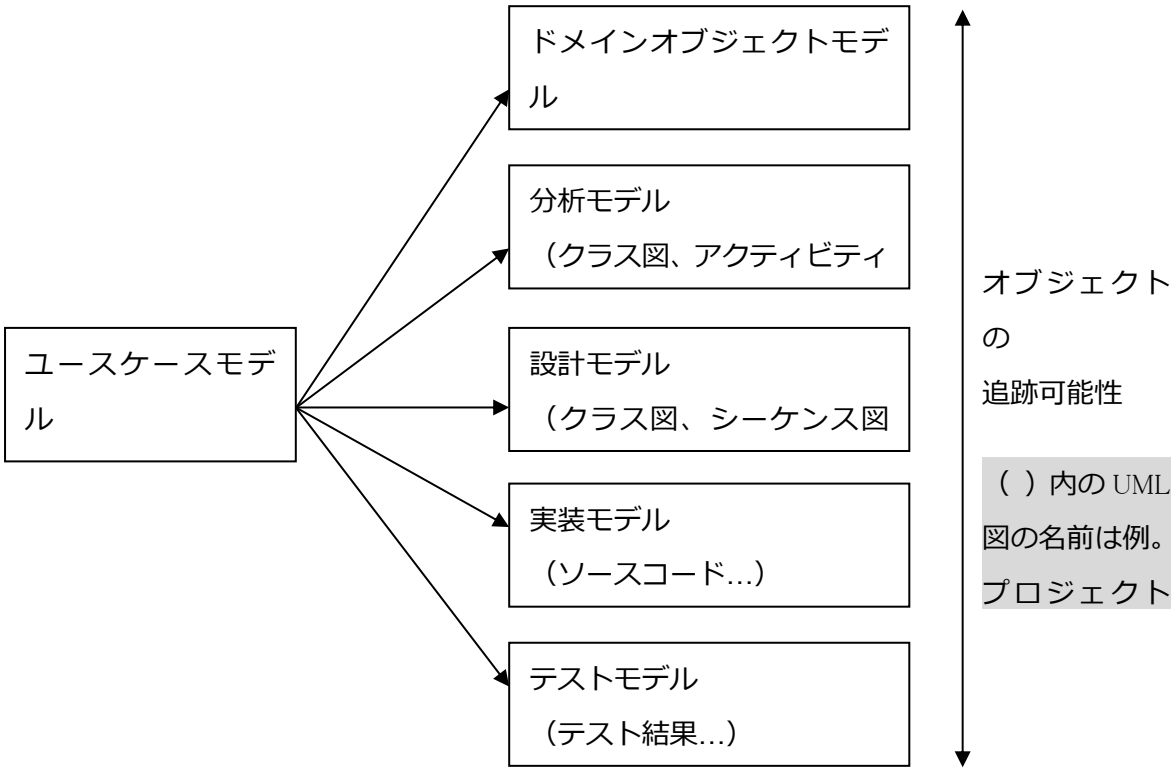
動的インスタンス関連

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

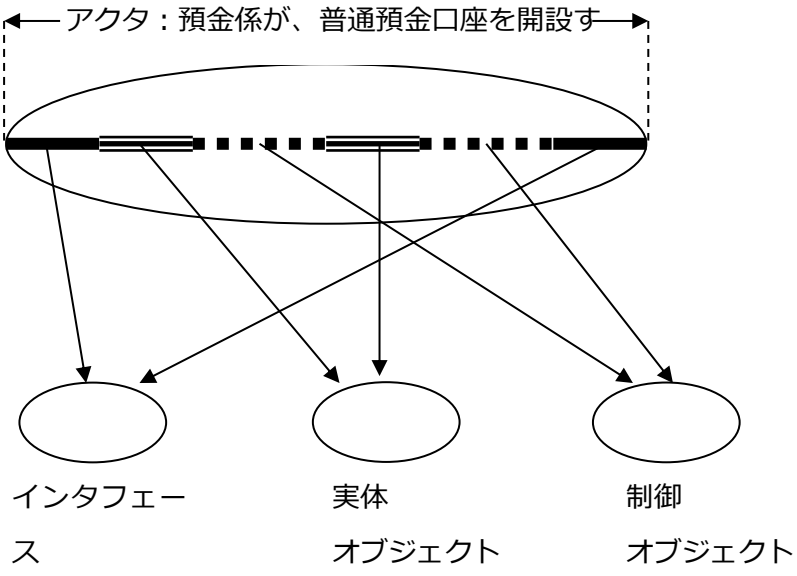
Rebecca Wirfs-Brock(1990)
コントローラ
コーディネータ

Jacobson(1994)
インタフェース
コントロール
エンティティ

3.6.8 参考資料1 SASD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

3.6.9 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。

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。

3.7.1 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

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

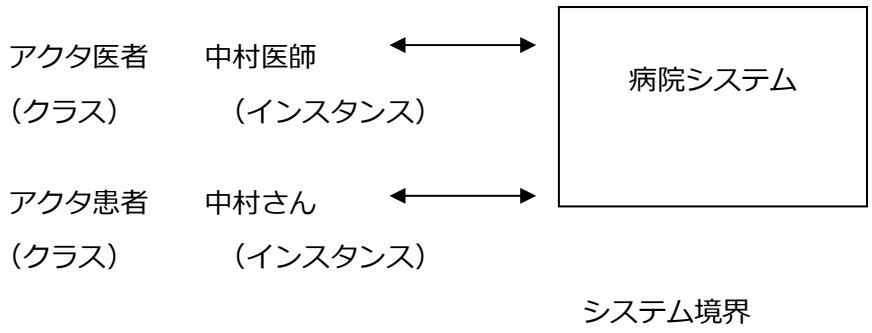
テスト結果を記述したもでのである。
オブジェクトモジュール、ユースケース、システム全体の順にテストを行う。

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

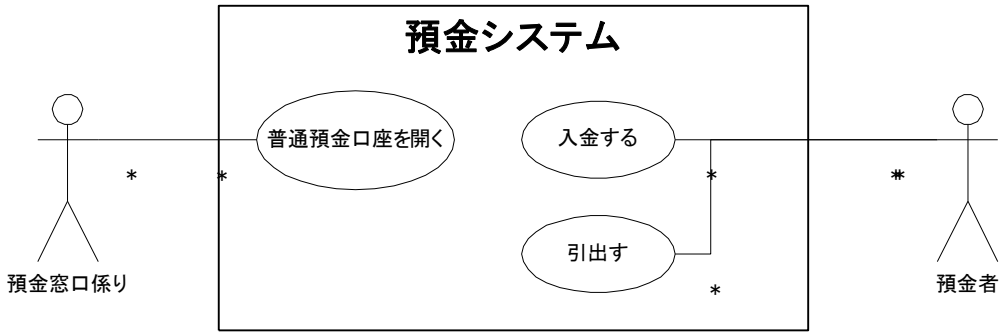
3.10.1 アクタ

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。

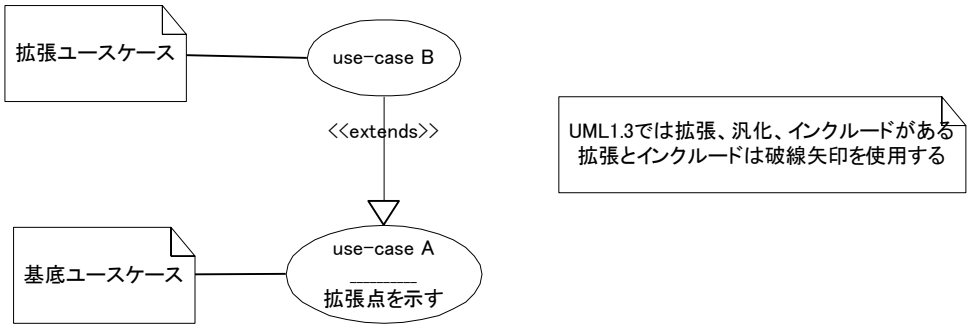
3.10.4 ユースケースは振舞いと状態を持つオブジェクト

（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が分かる。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、長く広範囲な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

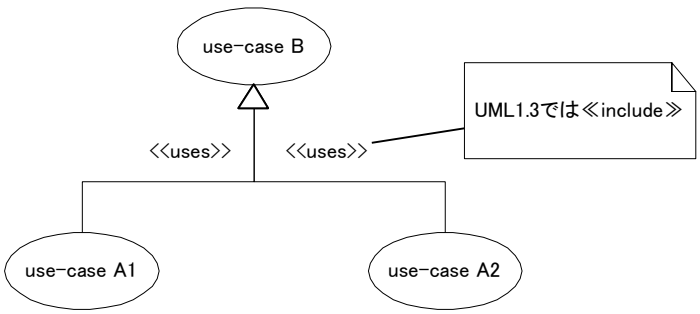
初めに単純で標準的なユースケースを作成する。(基本系列) その後、そのユースケースにバリエーションが必要になった場合、元のユースケースに直接バリエーションを追加できるが、場合によっては基本系列が見えにくくなったりする。 そこで、バリエーションを別のユースケースとして作成し拡張関係によりユースケースを完成させることができる。
(選択的な振る舞い) バリエーションや例外処理などのサブフローのこと。
それぞれ独立したユースケースとして記述する。
基底となるユースケースの記述の中に、基底ユースケースのどこに拡張ユースケースを組込むかを記述する。(拡張点の記述)
サブフローは全て別にしなくても構わない。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）

ＵＭＬの概要
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。

4.1.1 UML1.3

1999 年 6 月発行

4.1.2 ユースケース図（use case diagram）

ユースケース図

4.1.3 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）

4.1.4 静的構造図（static structure diagram）

クラス図
オブジェクト図

4.1.5 振舞い図（behavior diagram）

ステートチャート図

アクティビティ図

4.1.6 実装図(implementation diagram)

コンポーネント図

配置図

4.2 ユースケース

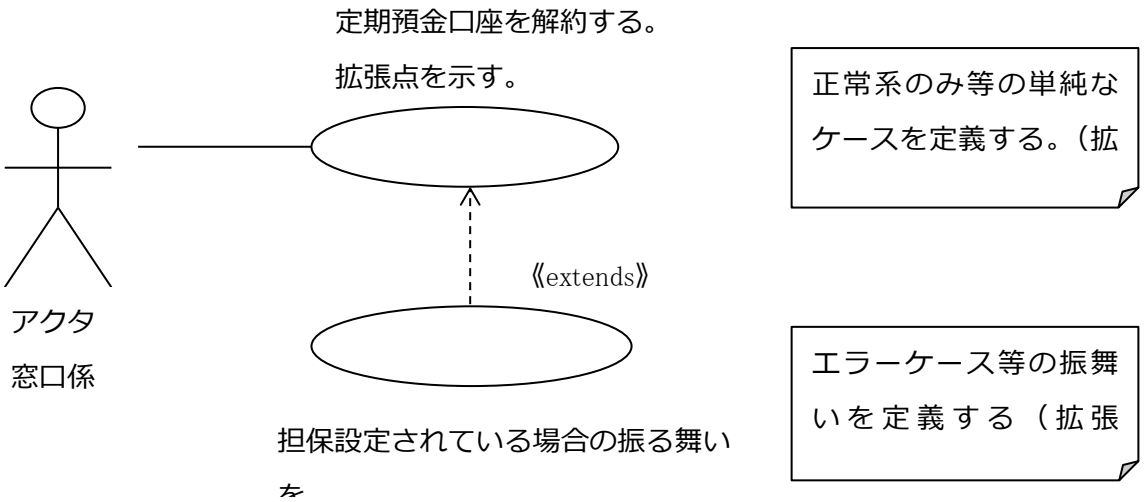
関連　アクタとユースケースの関係

拡張

インクルード

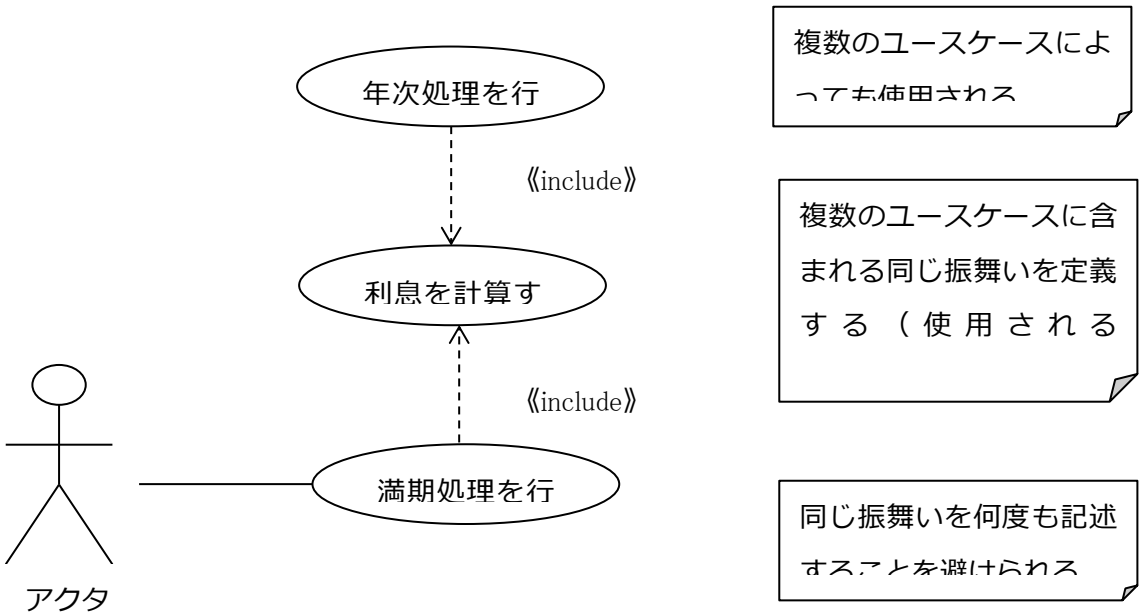
汎化

4.2.1 拡張



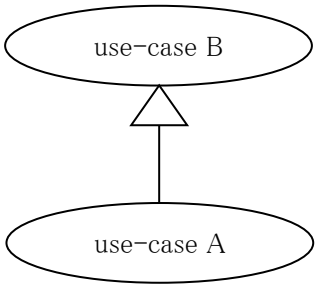
基底ユースケースは、拡張ユースケースがなくても完結する。

4.2.2 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。

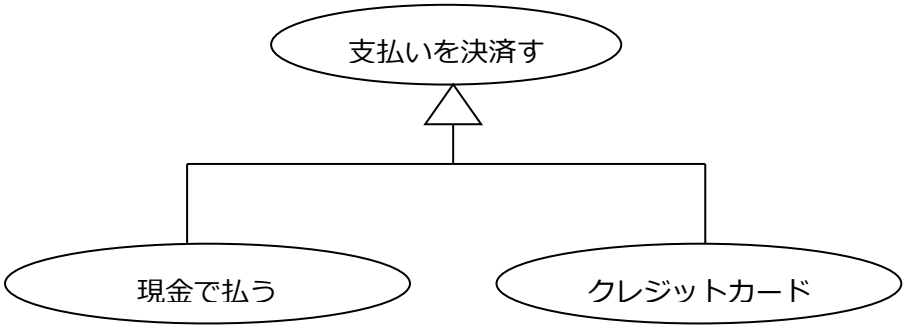
4.2.3 汎化



解約する

（抽象 use-case であることが多

普通口座を解約する



メリット：理解しやすくなる。

4.2.4 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

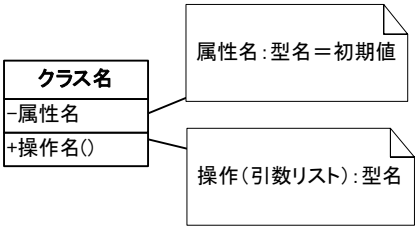
繰返し開発、イテレーションの単位

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。

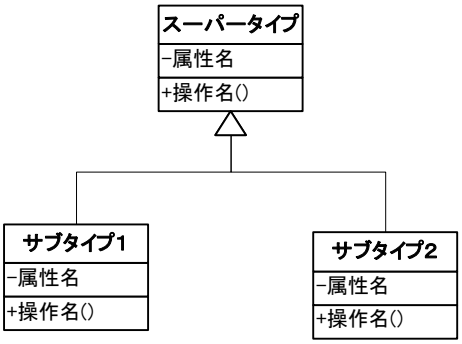
[参考文献　UML モデリングのエッセンス　マーチン・ファウラー、ケンドール・シコット]

4.3 クラス図

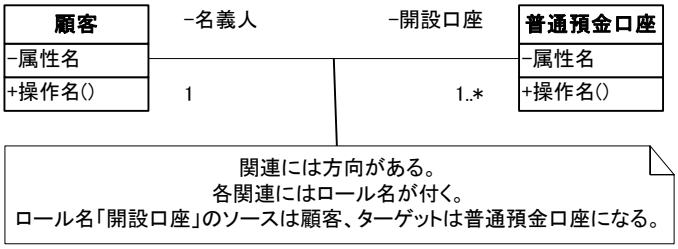
4.3.1 クラス

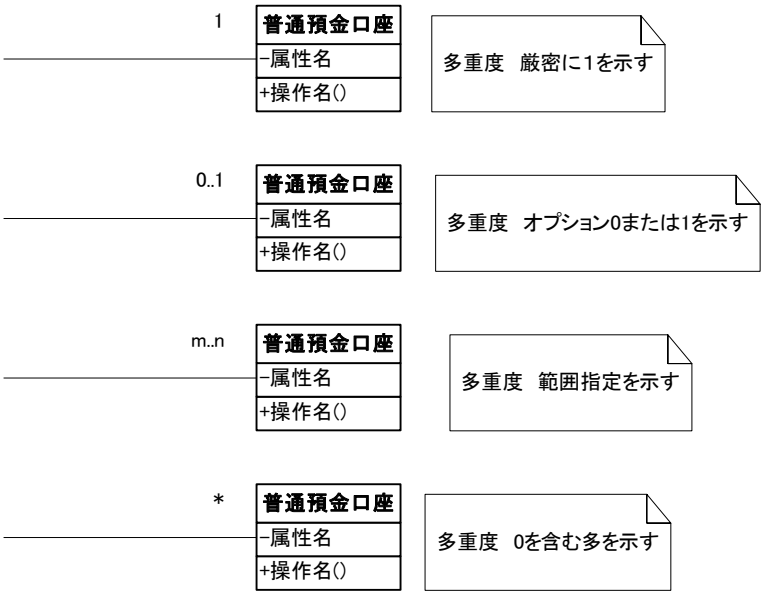


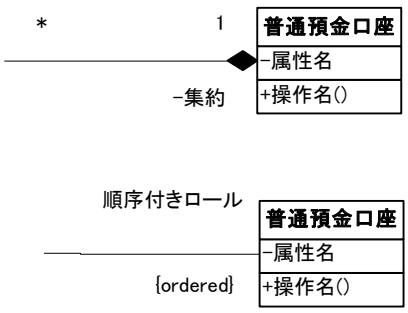
4.3.2 汎化・特化



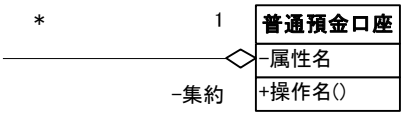
4.3.3 関連、ロール、多重度



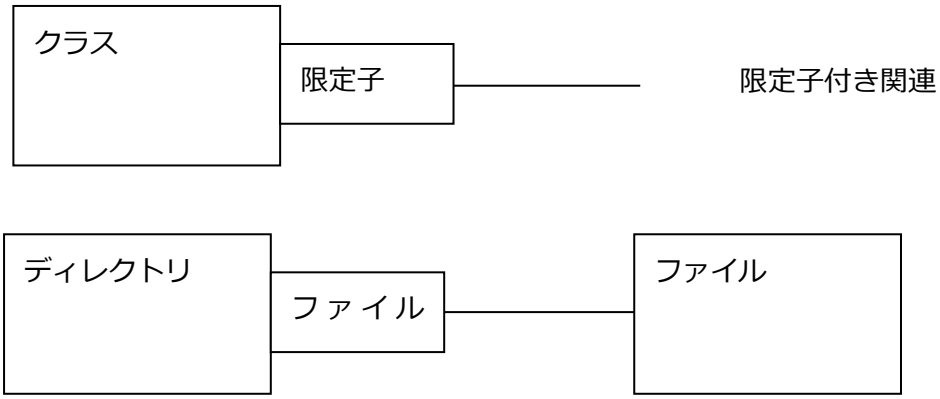




4.3.4 集約、コンポジション



4.3.5 限定子

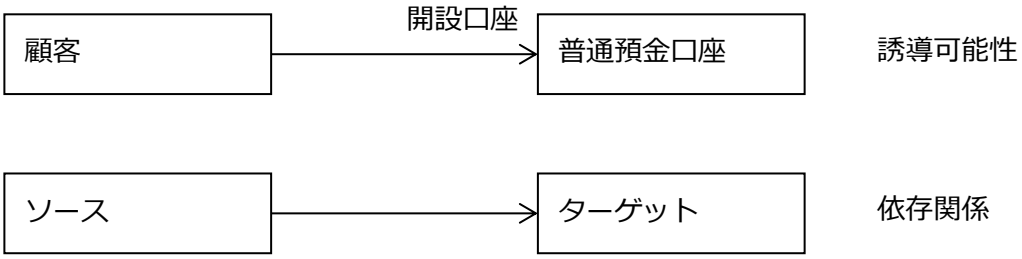


限定子によって関連の多重度を減らすことができる。

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



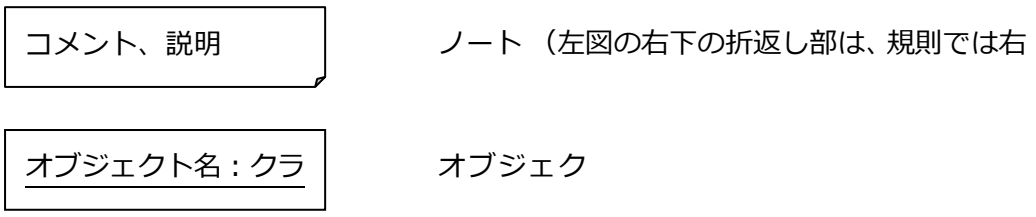
実装モデルにおいて、誘導可能性（navigability）の有無は重要な意味を持つ。

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

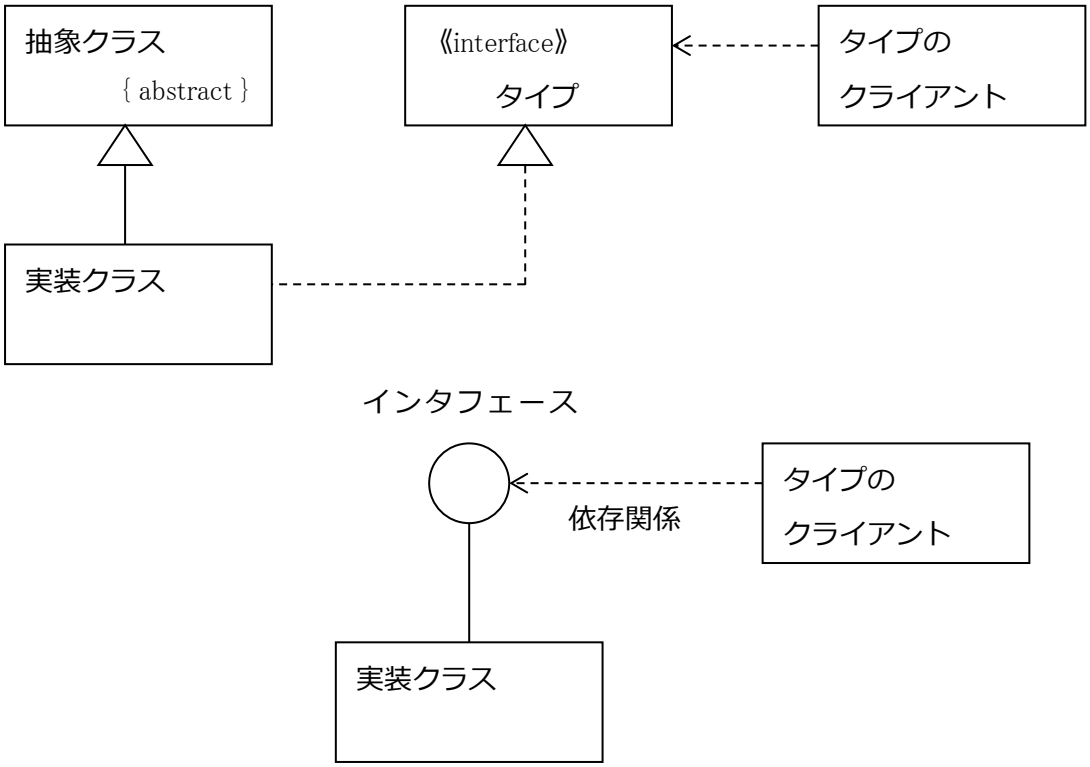
4.3.7 補助的記法

{制約の記述}

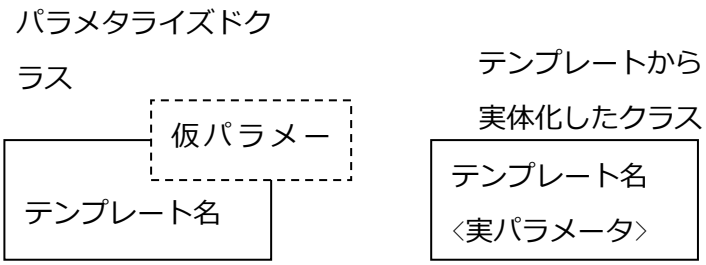
《ステレオタイプ名》



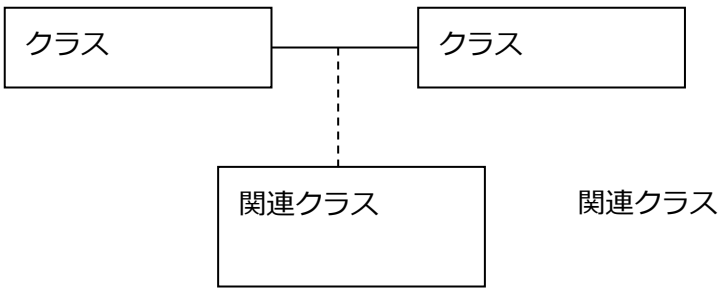
4.3.8 インタフェース



4.3.9 パラメタライズド・クラス、テンプレート・クラス



4.3.10 関連クラス



4.3.11 クラス図を使う

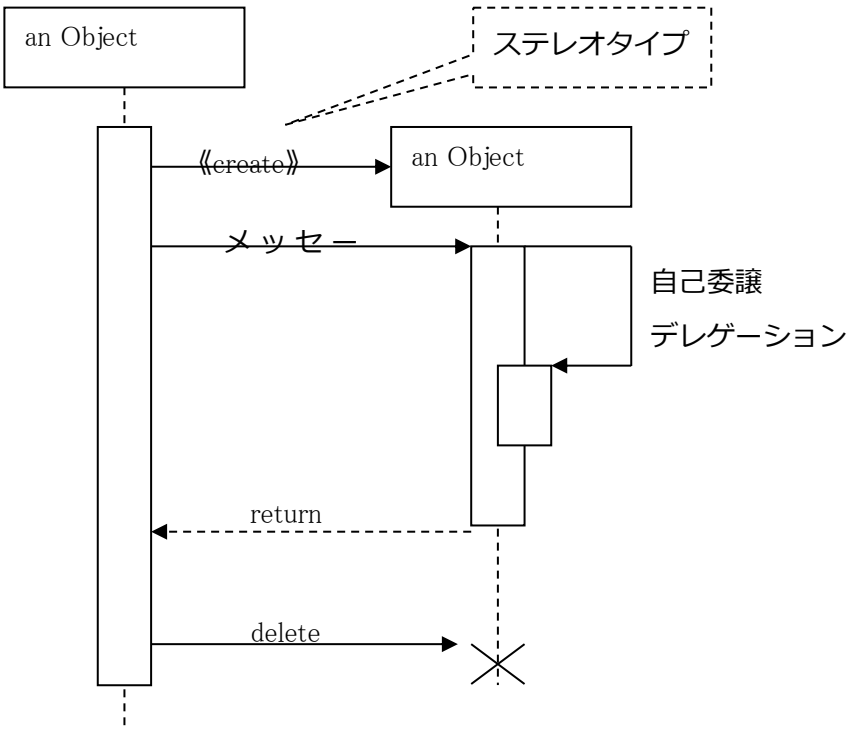
すべての表記を使うことに注力しない。

観点を意識する。（概念の観点、仕様の観点、実装の観点）

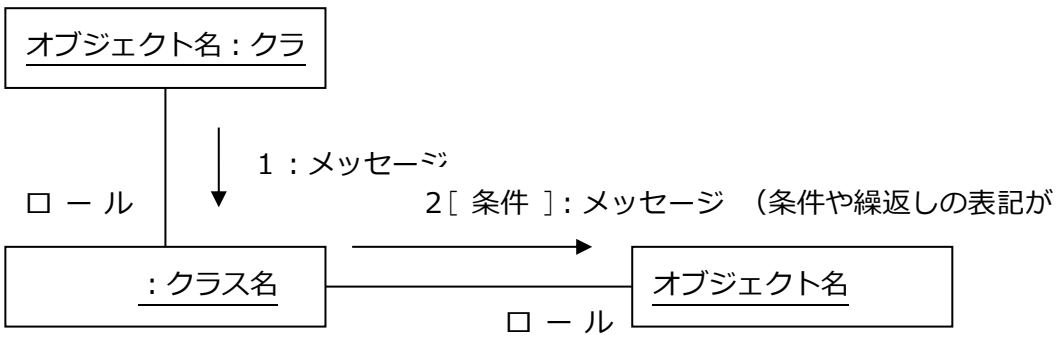
あらゆるものに対してモデルを作成しない。

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

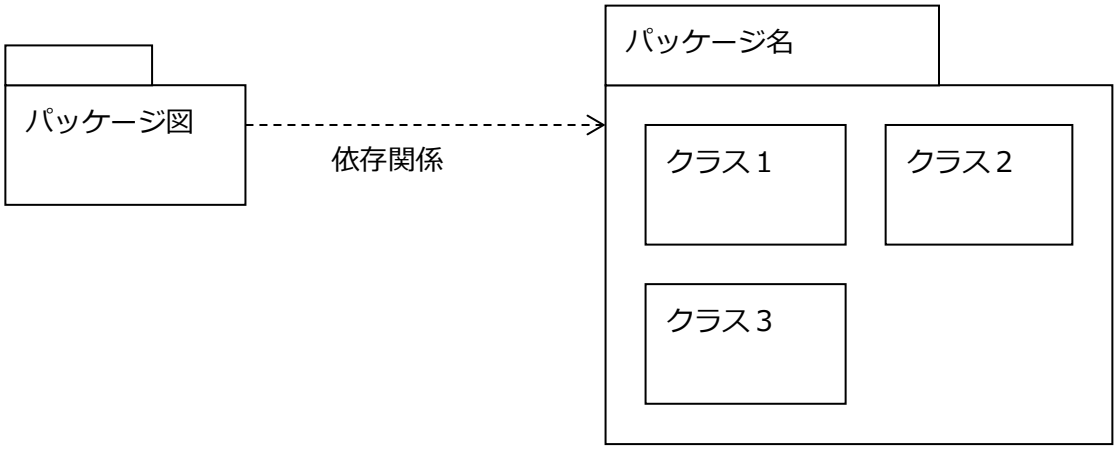
１つのユースケースにおけるオブジェクト群の振舞いを示す。

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

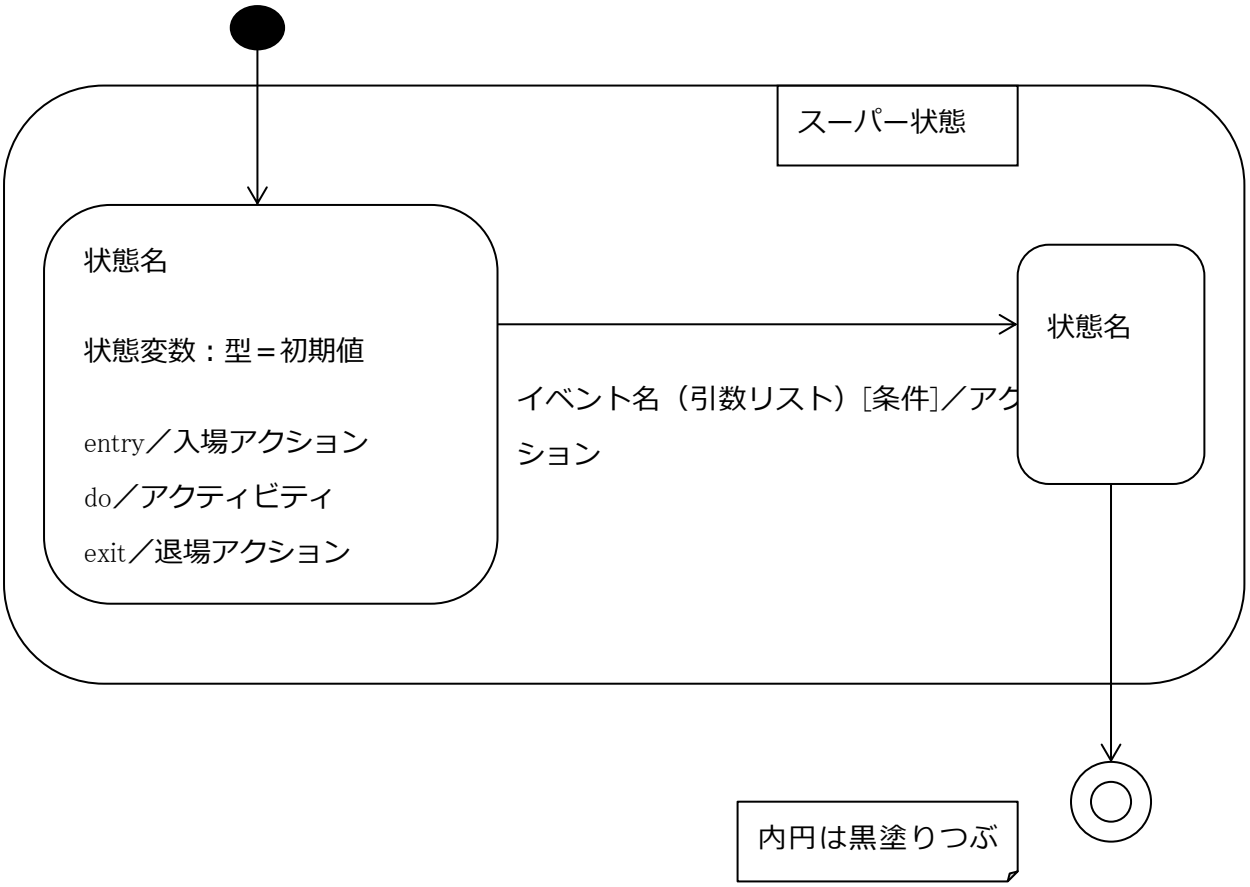


4.5.1 パッケージ図を使う

大規模なプロジェクトでは不可欠。

4.6 振舞い図

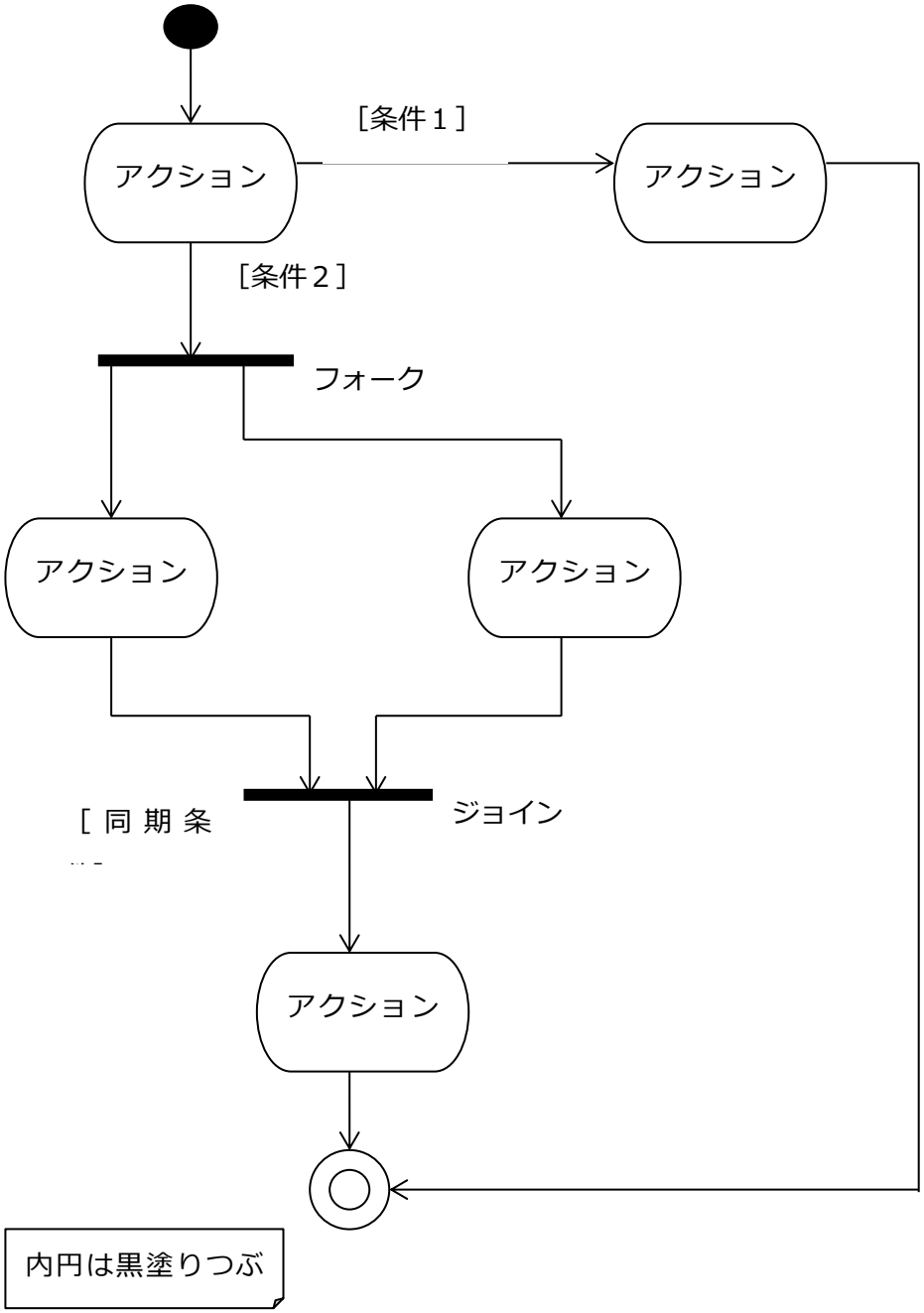
4.6.1 ステートチャート図（状態図）



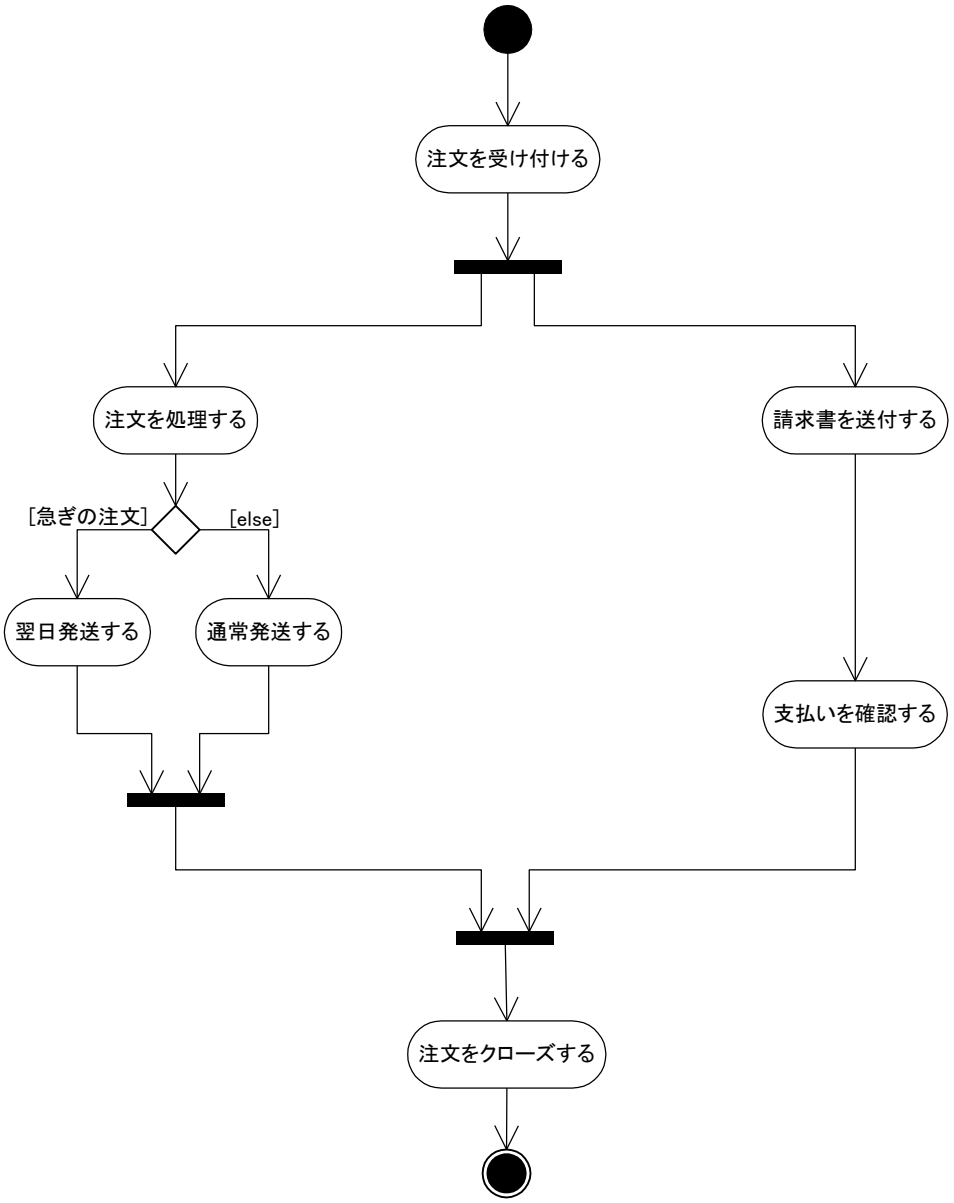
4.6.2 ステートチャート図を使う

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]

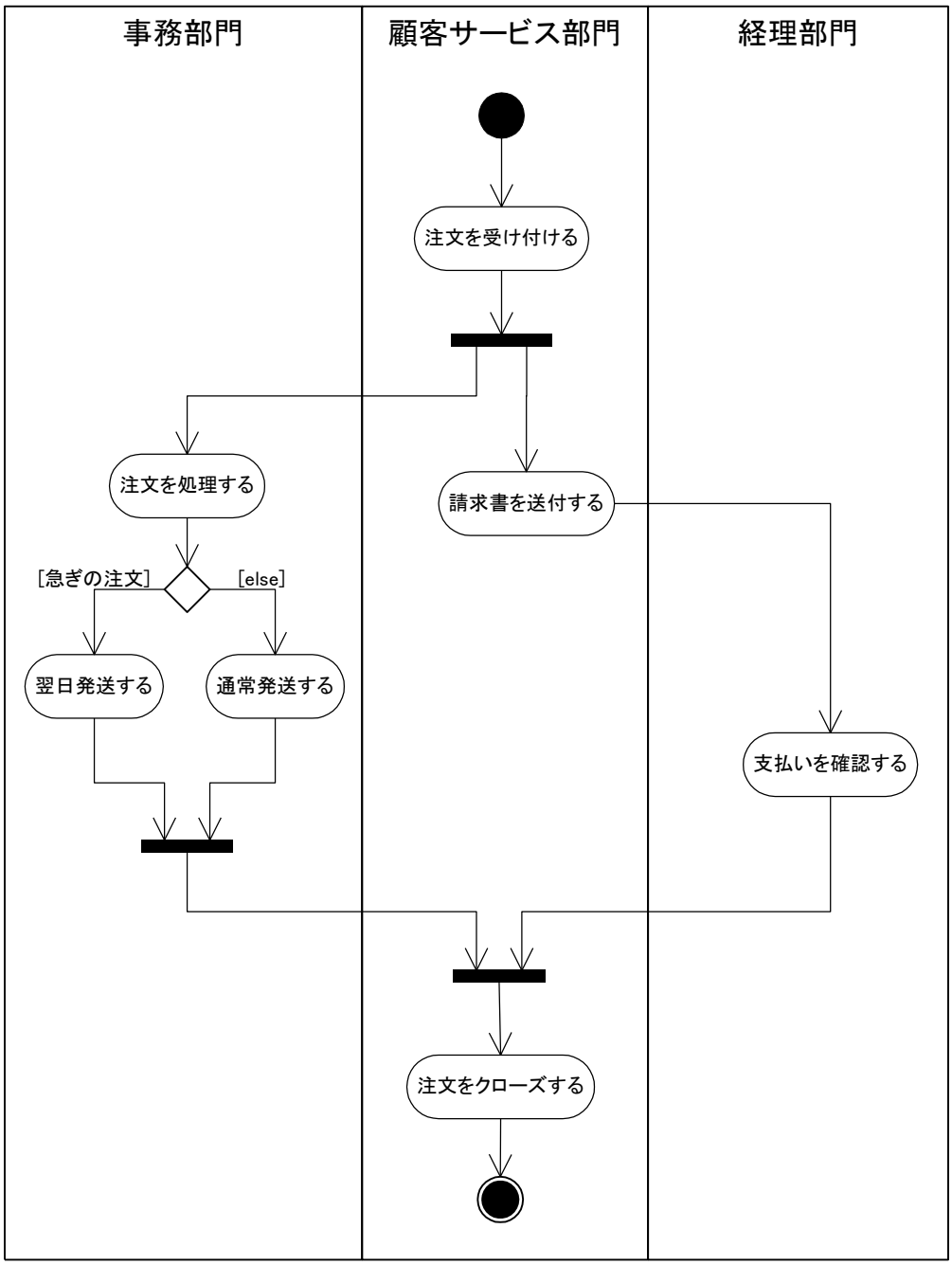
4.7 アクティビティ図


アクション
［条件１］
アクション
［条件２］
フォーク
アクション
アクション
［ 同 期 条
ジョイン
アクション
内円は黒塗りつぶ

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）

《metacless》

《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（Genaralization）

《implementation》

4.8.13 オブジェクトフロー状態（ObjectFlowState）

《signalflow》

4.8.14 パッケージ（Package）

《facade》

《framework》

《stub》

《toplevel》

4.8.15 パーミッション（Permission）

《access》

《friend》

《import》

4.8.16 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エア)

Cittera (Konesa) (オージス総研)

MagicDraw Pro (エッチ・アイ・シー)

WithClass (グレープシティ)

Pattern Weaver (テクノロジックアート)

Together ControlCenter (トゥゲザーソフト・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/)

4.11 関連する動向

MDA（Model Driven Architecture）

Action Semantics

UML2.0

5. オブジェクト指向プログラミング言語 Java

この章の内容

5.1 オブジェクト指向プログラミング

5.1.1 新たな言語やプログラミングスタイルをはじめる場合

- BASIC から C へ。　　goto 文から構造化プログラミングへ。
- 関数プログラミング型。 関数を組合わせてプログラムを構成するスタイル。Lisp、APL 等。

・イベントドリブン型。　　ユーザの操作等のイベントに対応する処理を記述するスタイル。

5.1.2 オブジェクト指向言語をはじめる場合

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” ということを学ぶ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)
オブジェクト指向言語の長所を生かすためには、パラダイムシフトが必要です。[Martin Fowler 1997]

・オブジェクト指向技術が必要になります。
・構造化プログラミングの技術はメソッドの実装に活かされます。
・Java を使うからオブジェクト指向設計するのではなく、オブジェクト指向設計するから Java が必要です。
・プログラムやモジュールなどの概念や単位ではなく、オブジェクト（クラス）という概念、単位に置き換わります。

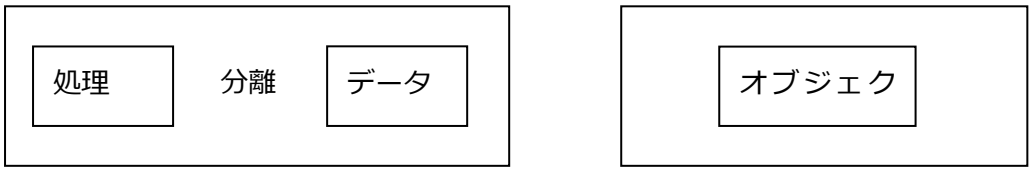
5.1.3 オブジェクト指向言語の例

・Java
・C++
・Smalltalk
・C#

5.2 オブジェクト指向と手続き指向の比較

5.2.1 データとオブジェクト

・手続き型のデータオブジェクトは値が入っているだけで、その値の意味は手続きによって与えられます。
・オブジェクトは、その他にそれ自身が機能を持ち、能動的であり、それ自身で完全な存在です。
・データと機能を分離する開発では、プログラマはデータ構造を暗黙的に理解していることが条件です。 例えば、プログラマは複数のテーブルと、テーブル間の関連を理解しておくことが必要です。　オブジェクト指向の場合は、あるクラスが知っています。



5.3 Java の出現

当初は「アプレット開発用言語」との見方が多かったが、現在は商用システムの主力開発言語として使用されています。 オブジェクト指向プログラミング言語、ネットワーク時代のプログラミング言語として進化、普及しています。
1995 年 サン・マイクロシステムズが Java を発表。

1995 年 JDK 1.0β版（Java Development Kit） （10 月）

1996 年 JDK1.0 版 （1 月）

1998 年 JDK1.2 （12 月）

2000 年 J2SE1.3

2002 年 J2SE1.4

2004 年 J2SE1.5.0 （9 月）

2005 年 1 月現在 J2SE1.4.2 / J2SE5.0

5.4 Java の特徴

5.4.1 オブジェクト指向言語

- 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。
- コンポーネントの作成・利用が容易になります。

5.4.2 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言えます）

5.4.3 インタプリタ型

- コンパイルによりバイトコード（中間コード）を生成します。
- プロトタイピングに向いています。

5.4.4 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。

5.4.5 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++のポインタ演算はない）

5.4.6 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全く独立している。
Java ではセキュリティのパッケージが JDK に含まれる。

5.4.7 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く。

5.4.8 移植性

抽象 GUI クラス（AWT → Swing Look&Feel も統一）

5.4.9 マルチスレッド

高機能マルチメディアアプリケーションに効果がある
同期プリミティブが高機能、豊富

5.4.10 ソフトウエア・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る。
Solaris、Windows95/98/NT、OS/2、MacOS、UNIX、NextStep、BeOS、Linux

5.4.11 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開放する。
Ｃ++で new したものを、delete や free しなくてよい。（new はあるが delete はない）
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）

5.5 Java を始める

ＪＤＫをダウンロードする。
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く、動かす。
新しくコードを書く。 → 何をクラスにするのか？
要件書を渡される。 → 何をクラスとするのか？ このメソッドはどこに書くのか？

5.6 HelloWorld (1)

コマンドプロンプトから開始する Java アプリケーションを書く。

5.6.1 ステップ1 Javaファイルを作成する（テキストエディタ）

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

5.6.2 ステップ2 Javacでclassファイルを作成する（コンパイルする）

```
>javac HelloWorld.java
```
※ コンパイルする

5.6.3 ステップ3 実行する

```
>java HelloWorld
HelloWorld
```

5.7 基本型と参照型

（Types¥Sample.java）

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

    /** 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 */
    Sample sample = null;
}
```

5.8 変数宣言

（Variables¥Sample.java）

```
/**
 * 変数宣言
 */
public class Sample {

    /** 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）

    }
}
```

5.9 演算子

（Operators¥Sample.java）

```
/**
 * 演算子
 */
public class Sample {

    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");

    }
}
```

5.10 制御フロー文

（Statements¥Sample.java）

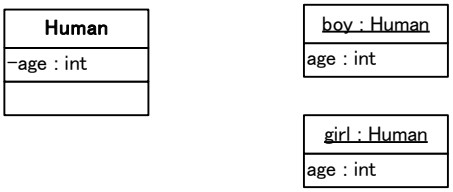
```
/**
 * 制御フロー文
 */
public class Sample {

    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
        }
```

```
        //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;
    }
}
```

5.11 HelloWorld (2)

5.11.1 サンプル HelloWorld (2) のオブジェクト図

最初にStarterクラスのオブジェクトがコマンドラインから起動されると、Starterクラスのmainメソッドはクラスのインスタンスを２つ生成する。



5.11.2 Javaファイルを作成する

（HelloWorld¥Starter.java）

```
/**
 * 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) {
		Human boy, girl;
		boy = new Human(10);
		girl = new Human(12);
	}
}

/**
 * 人間クラス
 */
class Human {
	private int age;
	/**
	 * コンストラクタ
	 */
	public Human(int p_age) {
		super();
		age = p_age;
		System.out.println("HelloWorld and I am " + age + " years old.");
	}
}
```

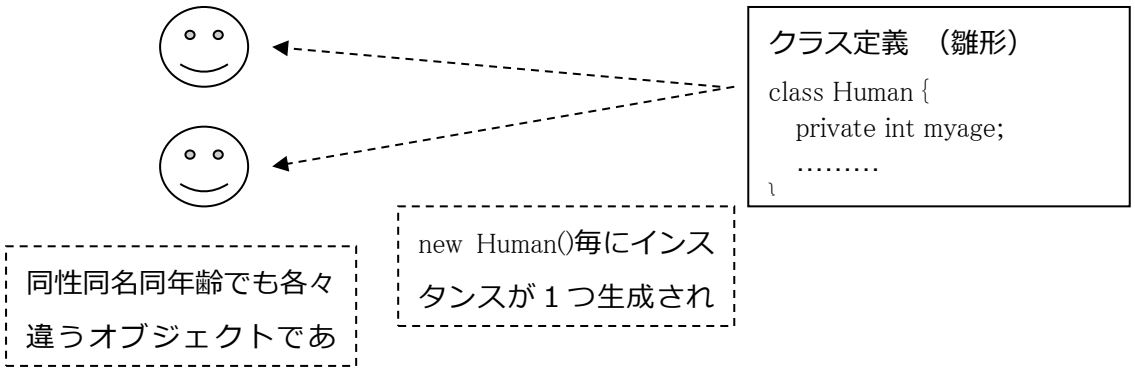
5.11.3 実行する

```
>javac Starter.java
>java Starter
HelloWorld and I am 10 years old.
HelloWorld and I am 12 years old.
```

5.12 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）
カプセル化されるもの。
オブジェクト指向プログラミングでのプログラム単位。
オブジェクトは内部状態を持ち、オブジェクト自身が管理する。
オブジェクト自身がメッセージを解釈し、メソッドを実行する。
オブジェクトにアクセスする唯一の方法は、メッセージをおくること。

5.13 クラスとオブジェクト（インスタンス）の関係



5.13.1 オブジェクトの同一性

全ての属性が同じ値でも、同じオブジェクトとは限らないこと。同性同名同年齢。
Java の文字列はオブジェクトであり、内容がおなじ“abc”であってもオブジェクトとしては同じとは限らない。
手続き型言語では、変数とオブジェクトを同一視する傾向がある。オブジェクト指向プログラミングでは、区別して考える。値が同じであることと、オブジェクトが同一であることとは、違う。
例として、ウインドウモデルでの２つのウインドウオブジェクトを考える。２つのウインドウが全く同じ場所で同じ内容を表示していても、オブジェクトとしては２つ別々に存在する。

（Identity¥Starter.java）

```
/**
 * オブジェクトの同一と同値
 */
public class Starter {

    /**
     * コマンドプロンプトからの開始メソッド
     * @param args 未使用
     */
    static public void main(String[] args) {
        Starter starter = new Starter();
        starter.start();
    }
```

```
    /**
     * テスト開始
     */
    private void start() {

        ClassA a1 = new ClassA(1);
        ClassA a2 = new ClassA(1);
        ClassA a3 = new ClassA(2);
        ClassA a4 = a1;

        System.out.println("a1 == a1 --> " + (a1 == a1));
        System.out.println("a1 == a2 --> " + (a1 == a2));
        System.out.println("a1 == a4 --> " + (a1 == a4));
        System.out.println("a1.equals(a2) --> " + (a1.equals(a2)));
        System.out.println("a1.equals(a3) --> " + (a1.equals(a3)));
    }

}

/**
 * このテストでのみ使用するクラス
 */
class ClassA {

    /** 属性 */
    private int attribute = 0;

    /**
     * コンストラクタ
     * @param value 属性値
     */
    protected ClassA(int value) {
        attribute = value;
    }

    /**
     * Object#equals メソッドをオーバーライドする
     * @param object
     */
    public boolean equals(Object object) {
        boolean result = false;
        if (object instanceof ClassA) {
            result = (attribute == ((ClassA)object).getAttribute());
        }
        return result;
    }

    /**
     * Object#hashCode メソッドをオーバーライドする
     * HashMap 等でパフォーマンスが著しく低下するが、equals, hashCode の実装要件は満たす
     */
```

```
	public int hashCode() {
		return 0;
	}

	/**
	 * 属性の getter メソッド
	 * @return 属性値
	 */
	public int getAttribute() {
		return attribute;
	}
}
```

（実行結果）

```
C:¥>java Starter
a1 == a1 --> true
a1 == a2 --> false
a1 == a4 --> true
a1.equals(a2) --> true
a1.equals(a3) --> false
```

5.14 クラス

5.14.1 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する。

クラス名

フィールド

メソッド

コンストラクタ

（HelloWorld¥Starter.java）

```
/**
 * クラス名：人間クラス
 */
class Human {

	/** 自分の年齢 */
	private int age;

	/**
	 * コンストラクタ
	 * 引数：年齢(初期値)
	 */
	public Human(int p_age) {
		super();
		age = p_age;
		System.out.println("HelloWorld and I am " + age + " years old.");
```

```
    }

    /**
     * 自分の年齢を応答するメソッド
     * 引数：なし
     */
    public int getAge() {
        return age;
    }
}
```

⇒　クラス図を書いてみる

5.14.2 アクセス制御

public　　全てのクラスからのアクセスを許す。

protected　サブクラス、同じパッケージ内からのアクセスを許す。

private　　そのクラス以外からのアクセスを許さない。 データの隠蔽。

（注）クラス単位であってインスタンス単位ではない。インスタンス単位の言語もある。

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る。

5.14.3 オブジェクトの生成

（HelloWorld¥Starter.java）

```
/**
 * 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) {
        Human boy, girl;
        boy = new Human(10);
        girl = new Human(12);
    }
}
```

5.14.4 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つ。 （フィールドは、タイプに応じて¥u0000、false、null に単純に初期化されるが、これ以外の初期化を行う場合、コンストラクタによって行う。）

コンストラクタはクラス名と同じ名前を持つ。 （下の例：Human()）

メソッドではないので戻り値は持たない。

no-arg コンストラクタ 引数を持たないコンストラクタ

複数のコンストラクタを持てる。

コンストラクタを持たない全てのクラスには、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する。

（HelloWorld¥Starter.java）

```
/**
 * クラス名：人間クラス
 */
class Human {

    /** 自分の年齢 */
    private int age;

    /** 自分の名前
    private String name;

    /**
     * コンストラクタ(1)
     * 引数：年齢(初期値)のみ
     */
    public Human(int p_age) {
        super();
        age = p_age;
        System.out.println("HelloWorld and I am " + age + " years old.");
    }

    /**
     * コンストラクタ(2)
     * 引数：年齢(初期値)、氏名
     */
    public Human(int p_age, String p_name) {
        super();
        age = p_age;
        name = p_name;
        System.out.println("HelloWorld and I am " + age + " years old.");
    }

    /**
     * 自分の年齢を応答するメソッド
     * 引数：なし
     */
    public int getAge() {
        return age;
    }
}
```

5.14.5 メソッド

分析、設計モデルの操作に相当する

他に、実装上必要となる操作やパフォーマンス上追加する操作等がある

（HelloWorld¥Starter.java）

```
/**
 * 自分の年齢を応答するメソッド
 * 引数：なし
 */
public int getAge() {
	return age;
}
/**
 * 自分の年齢を変更するメソッド
 * 引数：新たな年齢
 */
public void setAge(int p_age) {
	age = p_age;
}
```

5.14.6 メソッドのオーバーロード

各メソッドはシグネチャをもつ。 シグネチャとはメソッド名、パラメータの数とタイプである。
シグネチャが異なれば、同じ名前のメソッドを定義できる。
下の例では、同じ操作名 change がオーバーロードされている。

```
/**
 * 自分の年齢を変更するメソッド
 * 引数：新たな年齢
 */
public void change(int p_age) {
	age = p_age;
}
/**
 * 自分の年齢と氏名を変更するメソッド
 * 引数：新たな年齢
 *		新たな氏名
 */
public void change(int p_age, String p_name) {
	age = p_age;
	name = p_name;
}
```

5.14.7 クラスメンバーとインスタンス変数

クラスメンバーは、クラス変数とクラスメソッドのこと。

クラス変数（static）は、そのクラスに１個存在する。

インスタンスが１つもなくても存在する。

インスタンス変数は、個々のインスタンス毎に存在する。　例：普通預金口座クラスとインスタンス変数：残高

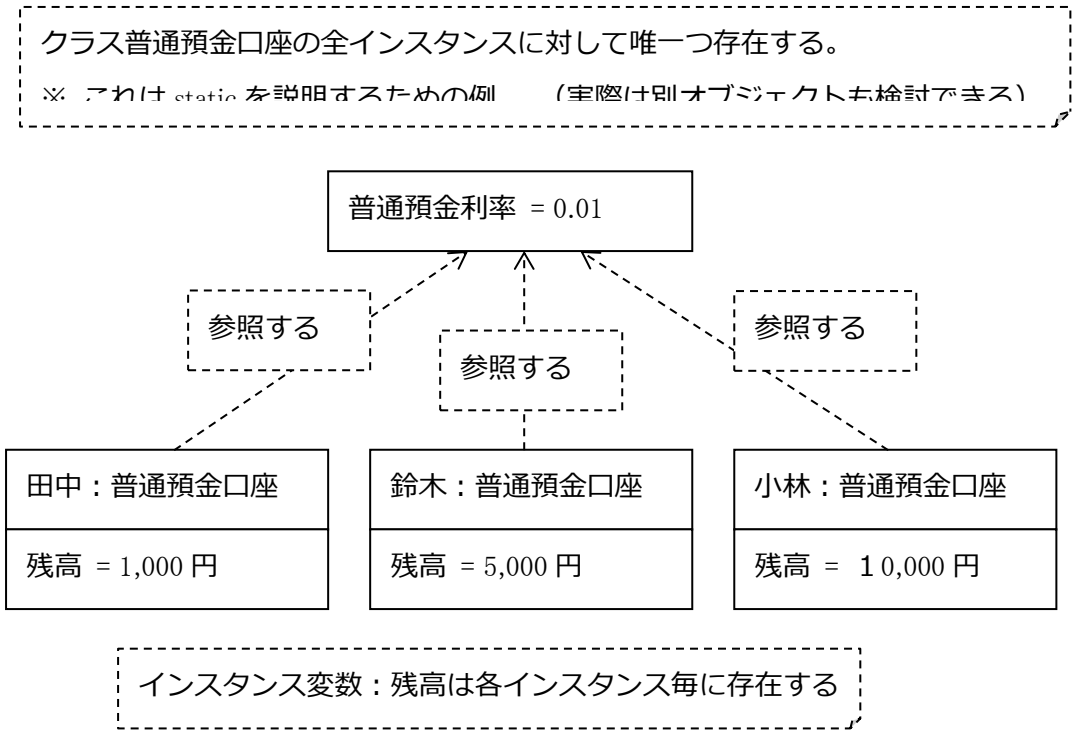
クラスメソッド

static メソッド。

そのクラスに対して唯一つ存在する。

メソッドへのアクセスはオブジェクト参照ではなく、クラス名を使う。

オブジェクト参照ではないので、this は使用できない。



＜例＞

（ClassMember¥Starter.java）

```
/**
 * 最初にコマンドラインから始動されるクラス
 * 1. OrdinaryAccount(普通預金口座)クラスのインスタンスを 3 つ生成する
 * 2. 1 つの口座を解約する(close メソッド)
 * 3. 利率を変更する(OrdinaryAccount クラスの static メソッド setRate)
 * 4. 残り 2 つの口座を解約する
 */
public class Starter {
    public static void main (String[] args) {
        OrdinaryAccount oa1 = new OrdinaryAccount(1000);
```

```
        OrdinaryAccount oa2 = new OrdinaryAccount(5000);
        OrdinaryAccount oa3 = new OrdinaryAccount(10000);

        oa1.close();
        OrdinaryAccount.setRate(0.2);
        oa2.close();
        oa3.close();
    }
}
/**
 * 普通預金口座クラス
 * 残高はインスタンス変数：balance (口座毎に必要)
 * 利率はクラス変数：rate (全ての普通預金口座で共通)
 */
class OrdinaryAccount {
    protected static double rate = 0.1;
    private long balance;
    private Customer custmer;
    /**
     * 普通預金の利率を設定する
     * 引数：利率
     */
    public static void setRate(double p_rate) {
        rate = p_rate;
    }
    /**
     * コンストラクタ
     * 引数：開始残高
     */
    public OrdinaryAccount(int p_balance) {
        super();
        balance = p_balance;
    }
    /**
     * 解約する
     * 払い戻し金額はこの操作が呼ばれた時点での rate を使って計算する
     * 引数：なし
     * 戻り値：払戻し金額
     */
    public long close() {
        long return_amount = 0;
        System.out.println("現在の利率は、" + rate + "%です。¥n");
        return return_amount;
    }
}
/**
 * 顧客クラス
 */
class Customer {
```

```
    private String name;
    public Customer(String p_name) {
        super();
        name = p_name;
    }
}
```

＜実行結果＞

```
>java Starter
```
現在の利率は、0.1%です。

現在の利率は、0.2%です。

現在の利率は、0.2%です。

5.14.8 参考

メッセージはオブジェクト毎に解釈されるものなので、インスタンス変数と同じようにオブジェクト毎に異なるメソッドを定義できる言語もある。
例えば、ＧＵＩ用の部品オブジェクトで、イベント毎のメソッドを別々に定義するような場合に相当する。

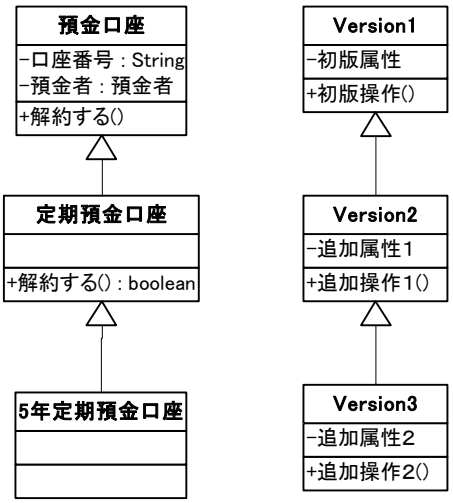
5.14.9 ネイティブメソッド

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る。
（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呼出す。
ＪＮＩ（Java Native Interface）

5.15 クラスの拡張

5.15.1 継承（UML クラス図）

継承の例　“新しい商品が追加された例”　と　“バージョンアップに適用した例”



（適用例）
Version1 の元々の振舞いを保障したまま（変更しないまま）、新しい振舞い（メソッド、操作）を追加できる。　変更は新しい振舞いを使用する他のクラスだけになる。

Version1 の振舞いを変えた（オーバーライドした）新しい Version2 を提供する。

5.15.2 継承（Java コード）

（BankAccount¥Customer.java）

```
/**
 * 預金口座
 * 抽象クラスなのでインスタンスは存在しない
 * 抽象メソッド close() を持つ
 */
abstract class Account {
	private String accountNumber; // 口座番号
	private Customer customer;
	public Account() {
		super();
		accountNumber = "";
	}
	public abstract boolean close();
}

/**
 * 定期預金口座
 */
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;
	}
}

/**
 * 5 年定期預金口座　（解約方法は同じなので実装する必要はない）
 */
class Fixed5YearsAccount extends FixedAccount {
	public Fixed5YearsAccount() {
		super();
	}
}
```

5.15.3 this と super

this
非 static メソッドの中で、自分自身を参照する特別なオブジェクト参照として使用できる。

```
private double x;
private double y;
```

```
public void move(double x, double y) {
	this.x = x;
	this.y = y;
}

public void setX(double newX) {
	x = newX;
	/*
	    this.x = newX;
	*/
}
```

5.15.4 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て、それへの参照として振舞う。

5.15.5 継承（拡張）とポリモルフィズム

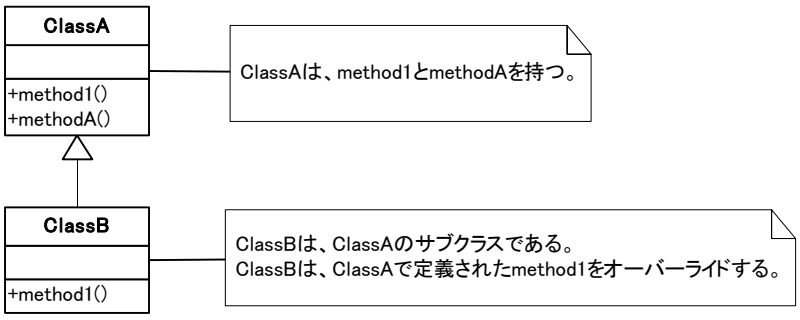
クラス ClassA が使用できるところでは、そのサブクラスである ClassB も同様に扱える。
拡張されたクラス（ClassB)も元のクラス（ClassA）と同様に扱える。

5.15.6 メソッドのオーバーライド

シグネチャ（メソッド名、パラメータの数、タイプ）が同じメソッドを定義する。（Method1()）
static メソッドはオーバーライド出来ない。
（注）オーバーロードはメソッド名以外のシグネチャが異なる。

5.16 メソッド結合

＜例＞




```
/**
 * 最初にコマンドラインから始動されるクラス MethodBinding¥Starter.java
 * ClassA のインスタンスの参照を、ClassA 型の参照に代入し、method1()を呼ぶ。
 * ClassB のインスタンスの参照を、ClassA 型の参照に代入し、method1()を呼ぶ。
 * ClassA のインスタンスの参照を、ClassA 型の参照に代入し、methodA()を呼ぶ。
 * ClassB のインスタンスの参照を、ClassB 型の参照に代入し、methodA()を呼ぶ。
```

```
 */
class Starter {
    public static void main(String[] args) {
        /**
         * 参照(変数)ref_a、ref_b のタイプを宣言する
         */
        ClassA ref_a;
        ClassB ref_b;
        System.out.println("¥n(1) 参照のタイプではなく、" +
                           "参照先のオブジェクトのタイプで" +
                           "呼ばれるメソッドが決まることを確かめる");
        System.out.print ("ref_a=new ClassA()、ref_a.method1()の結果 --> ");
        ref_a = new ClassA();
        ref_a.method1();
        System.out.print ("ref_a=new ClassB()、ref_a.method1()の結果 --> ");
        ref_a = new ClassB();
        ref_a.method1();
        System.out.println("¥n(2) クラスの継承木を溯り、" +
                           "初めて遭遇するメソッドが" +
                           "呼出されることを確かめる");
        System.out.print ("ref_a=new ClassA()、ref_a.methodA()の結果 --> ");
        ref_a = new ClassA();
        ref_a.methodA();
        System.out.print("ref_b=new ClassB()、ref_b.methodA()の結果 --> ");
        ref_b = new ClassB();
        ref_b.methodA();
    }
}
/**
 * クラスＡの定義
 */
class ClassA extends Object {
    public ClassA() {
        super();
        //System.out.println ("ClassA のコンストラクタが実行された");
    }
    public void method1() {
        System.out.println ("ClassA の method1 が実行された");
    }
    public void methodA() {
        System.out.println ("ClassA だけが持つ methodA が実行された");
    }
}
/**
 * クラスＡを継承したクラスＢの定義
 */
class ClassB extends ClassA {
    public ClassB() {
```

```
        super();
        //System.out.println ("ClassB のコンストラクタが実行された");
    }
    public void method1() {
        System.out.println("ClassB の method1 が実行された");
    }
}
```

<実行結果>

>java Starter

(1) 参照のタイプではなく、参照先のオブジェクトのタイプで呼ばれるメソッドが決まることを確かめる

ref_a=new ClassA()、ref_a.method1()の結果 --> ClassA の method1 が実行された

ref_a=new ClassB()、ref_a.method1()の結果 --> ClassB の method1 が実行された

(2) クラスの継承木を溯り、初めて遭遇するメソッドが呼出されることを確かめる

ref_a=new ClassA()、ref_a.methodA()の結果 --> ClassA だけが持つ methodA が実行された

ref_b=new ClassB()、ref_b.methodA()の結果 --> ClassA だけが持つ methodA が実行された

5.16.1 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ある。

Smalltalk の場合は常に動的メソッド結合を行う。

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る。

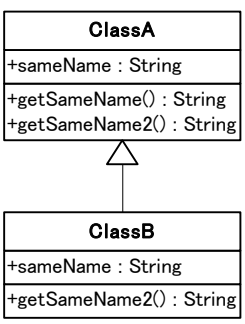
5.16.2 静的メソッド結合(Static method binding)

コンパイル時に呼出すメソッドを決定する。

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

5.17 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るが、単に名前を使っただけではアクセス出来なくなる。 super その他の参照を用いなければならない。



<例>

（SameFieldName¥Starter.java）

```
/**
 * 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＞

```
>java Starter
ClassA 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
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()=A

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   <- ①
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。

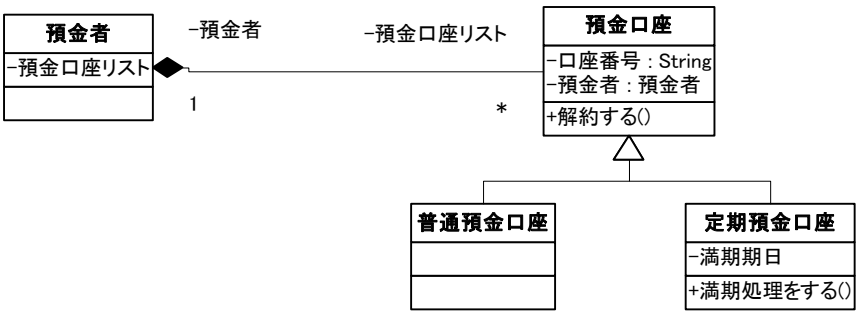
メソッドの場合、参照のタイプではなく、実際のタイプで呼出される。

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る。 -> ①

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ある。

5.18 タイプ変換

Ｊａｖａは強い型付けの（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る）言語である。



5.18.1 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();
	}
}
```

5.18.2 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＞

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

5.18.3 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;
            }
        }
}
```

5.18.4 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る。

（Constructor¥Starter.java）

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>

```
>java Starter
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あるが、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 行目に書かなければならない。

通常、デフォルトコンストラクタであっても 1 行目に super(); を書くようにする。

```
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>

```
>java Starter
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
```

ケース 3

自分のコンストラクタを実行する。　this(…)

```
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

```
>java Starter
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
```

5.18.5 super()

コンストラクタの最初に super()や super(…)がない場合は、super()が実行される。 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」となる。

5.19 final 宣言

メソッドやクラスを final（最終版）宣言する。
final 宣言されたクラスのサブクラスは作れない。
final クラスのメソッドは暗黙的に final である。
final 宣言されたメソッドは最適化しやすい。 インライン展開が可能になる。

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

5.20 Object クラス

全てのクラスは直接・間接に java.lang.Object クラスを拡張する。

5.20.1 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る。 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ある。

5.20.2 pubulic int hashCode()

オブジェクトのハッシュ値を返す。 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、一意な値を返す。 Hashtable オブジェクトにオブジェクトを格納するために使う。

5.20.3 protected object clone()

オブジェクトの複製を返す。 ※cloneable インタフェース。

5.20.4 pubulic final Class getClass()

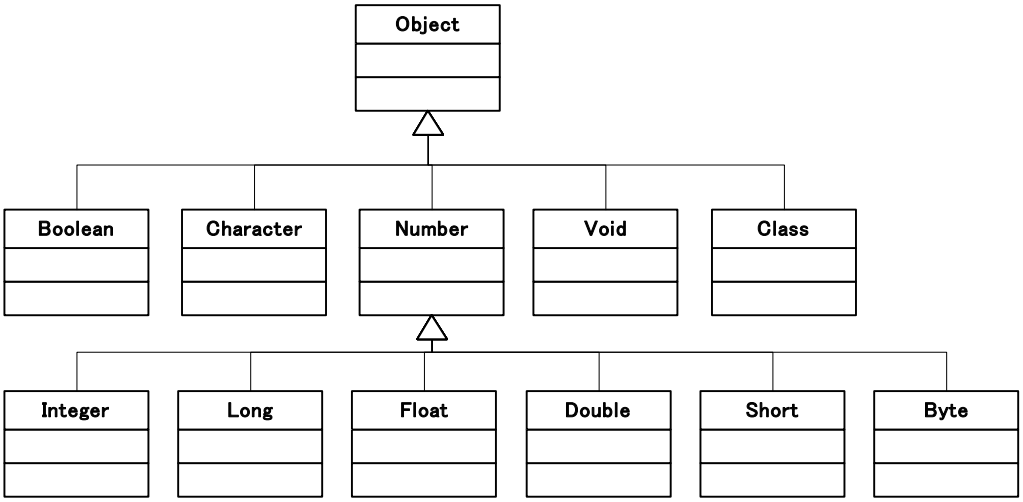
そのオブジェクトが属するクラスを Class タイプのオブジェクトによって返す。

5.20.5 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 する。

5.21 ラッパクラス（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）が存在する。ラッパクラスとして Interger や Boolean が用意されている。　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する。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多い。



5.22 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 オブジェクトを持つ。 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る。
関連 リフレクション

5.22.1 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。

5.22.2 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する、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する Method オブジェクトを格納している配列を返します。

（ClassClass¥Starter.java）

使用例

```
import java.lang.reflect.*;
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＞

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

5.23 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。
クラス中で実装されていないメソッドは abstract 宣言する。
１つでも abstract メソッドがある場合、そのクラスは abstract 宣言しなければならない。

```
/**
```

```
 * 預金口座
 * 抽象クラスなのでインスタンスは存在しない
 * 抽象メソッド close() を持つ
 */
abstract class Account {
    private String accountNumber;   // 口座番号
    private Customer customer;
    public Account() {
        super();
        accountNumber = "";
    }
    public abstract boolean close();   // 実装は普通預金口座、定期預金口座の各クラスで行う
}
```

5.24 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粋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。

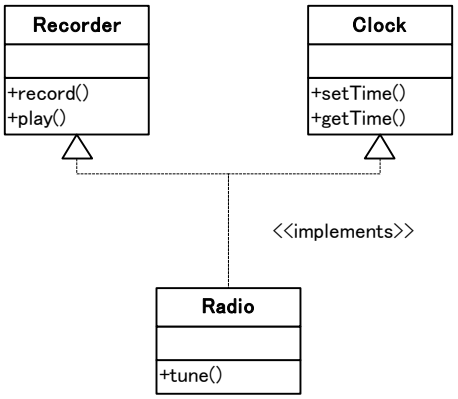
メソッドは static にはなり得ない。(static はクラスに対して定義できるもの)

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＞

(Interface¥Starter.java)

```
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＞

```
>java Starter
ref_a is an instance of Radio
```

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

5.24.1 インタフェースを使うとき

多重継承が必要な場合に使う。　※ Java は単一継承モデルである。(拡張できるスーパークラスは高々 1 つである)
リピータを抽出する（クラス図の中で繰返し現れるメソッドシグニチャを抽出する） [40]
命名規則例　“-able”, “-ible”, “-er”, “I-“など。(Linkable) [40]

5.25 例外クラス（Exception）

各メソッドの呼出し毎に可能性のあるエラーについて全てコーディングすると、正常なコードの流れが不明瞭になる場合がある。 このような場合、Java では try、catch、finally 構文を使って記述する。

（Exception¥Starter.java）

＜例＞

```
import java.util.*;
import java.text.*;
/**
 * 最初にコマンドラインから起動されるクラス
 */
class Starter {
	public static void main(String[] args) {
		String s = "2001/01/01";
		Date date;
		//-----------------------------------------
		// 例外を throw しない Converter クラスを使う場合
		//-----------------------------------------
		Converter cvt = new Converter();
		date = cvt.toDate(s);
```

```
        if (date != null) {                // <--- 忘れることが多くバグの原因になりやすい
            System.out.println("date = " + date);
        }
        //-----------------------------------------------------
        // 例外を throw する Converter2 クラスを使う場合
        //-----------------------------------------------------
        Converter2 cvtex = new Converter2();
        // 次の try を忘れるとコンパイルエラーになる （次節注意）
        // try 本体にエラー時のコードがないため、本来のコードが見やすい
        try {
            date = cvtex.toDate(s);
            System.out.println("date = " + date);
        } catch (MyParseException myexp) {
            System.out.println("My Exception Code = " + myexp.getMyCode());
        } finally {
            System.out.println("正常時も例外時も常に実行されます");
        }
    }
}

class Converter {
    public Converter() {
        super();
    }
    public Date toDate(String s) {
        Date ans = null;
        SimpleDateFormat format = new SimpleDateFormat("yyyy/MM/dd");
        try {
            ans = format.parse(s);
        } catch (ParseException ex) {
            System.out.println("ConvertError:" + s);
        }
        return ans;
    }
}

class Converter2 {
    public Converter2() {
        super();
    }
    public Date toDate(String s) throws MyParseException {
        Date ans = null;
        SimpleDateFormat format = new SimpleDateFormat("yyyy/MM/dd");
        try {
            ans = format.parse(s);
        } catch (ParseException ex) {
            throw new MyParseException("form Converter2");
        }
        return ans;
    }
}

/**
```

```
 * コンパイル時にチェックされる自分のシステム用の例外
 */
class MyParseException extends Exception {
	private String myCode;
	public MyParseException(String s) {
		super();
		myCode = s;
	}
	public String getMyCode() {
		return myCode;
	}
}
```

5.25.1 例外クラス

コンパイル時にチェックされない例外

RuntimeException クラスとそのサブクラス

Error クラスとそのサブクラス

これ以外は全てコンパイル時にチェックされる。

5.26 スレッド

5.26.1 実装方法

方法 1： java.lang.Thread を継承（extends）する

方法 2： java.lang.Runnable を実装（implemants）する

5.26.2 Thread を継承したスレッドプログラムの例

「プログラミング言語 Java」第３版１０章より引用

（Thread¥PingPong.java）

```
public class PingPong extends Thread {
		private String word;
		private int delay;

		public PingPong(String word, int delay) {
			this.word = word;
			this.delay = delay;
		}

		public void run() {
			try {
				for (;;) {
					System.out.print(word + "　");
					Thread.sleep(delay);
				}
			} catch (InterruptedException ex) {
				return;
			}
```

```
        }

        public static void main(String[] args) {
                new PingPong("ping", 33).start();
                new PingPong("PONG", 100).start();
        }
}
```

（実行結果）

```
C:¥>java PingPong
ping PONG ping ping ping PONG ping ping ping PONG ping ping ping PO
NG ping ping ping PONG ping ping ping PONG ping ping ping PONG ping
 ping ping PONG ping ping ping PONG ping ping ping PONG ping ping
ping PONG ping ping ping PONG ping ping ping PONG ping ping ping PO
NG ping ping ping PONG ping ping ping PONG ping ping ping PONG ping
```

5.26.3 スレッドの同期

（Thread¥Starter.java）

```
/**
 * 例題開始用インタフェースオブジェクト
 * 例題：スレッド同期の例
 * 1.口座を１つ開設する（初回入金 3000 円）
 * 2.同じ口座に対して、２ヶ所のＡＴＭから同時に操作しても残高が正しいことを確認する
 */
public class Starter {
    public static void main(String[] args) {
        Starter starter = new Starter();
        starter.start();
    }

    private void start() {
        BankAccount account = new BankAccount(3000);

        // ＡＴＭ操作（１）を実行するスレッドを定義する
        // 操作は、1000 円預入、次に 1000 円引出、最後に 1000 円預入れる
        // 正しい結果として、残高は 1000 円増える
        int[] sumList1 = {1000, -1000, 1000};
        UserTransaction transaction1 = new UserTransaction(account, sumList1);
        Thread thread1 = new Thread(transaction1);

        // ＡＴＭ操作（２）を実行するスレッドを定義する
        // 操作は、2000 円預入、次に 2000 円預入、最後に 3000 円引出す
        // 正しい結果として、残高は 1000 円増える
        int[] sumList2 = {2000, 2000, -3000};
        UserTransaction transaction2 = new UserTransaction(account, sumList2);
        Thread thread2 = new Thread(transaction2);
```

```
        // ＡＴＭ操作（１）と（２）を開始する
        thread1.start();
        thread2.start();

        // ＡＴＭ操作（１）と（２）の完了を待つ
        // 全ての操作が完了後、残高を表示する
        try {
            thread1.join();
            thread2.join();
            System.out.println("残高=" + account.getBalance() + "円");
        } catch (InterruptedException ex) {
            System.out.println("中断された");
        }
    }
}


/**
 * 口座クラス
 */
public class BankAccount {

    private int balance; //口座の残高

    /**
     * 口座クラスのコンストラクタ
     * @param 初回入金額
     */
    public BankAccount(int initialBalance) {
        super();
        balance = initialBalance;
        System.out.println("BankAccount:口座開設(" + balance + "円)");
    }

    /**
     * 残高を照会する
     * @return 現在の残高
     */
    public synchronized int getBalance() {
        return balance;
    }

    /**
     * 引き出す（残高不足を許す）
     * @param 引き出し額
     * @return なし
     */
    public synchronized void withdraw(int sum) {
```

```
            System.out.println("BankAccount:引出し(" + sum + "円)");
            int lastBalance = balance - sum;
            sleep(10);
            balance = lastBalance;
        }

        /**
         * 預け入れる
         * @param 預け入れ額
         * @return なし
         */
        public synchronized void deposit(int sum) {
            System.out.println("BankAccount:預入れ(" + sum + "円)");
            int lastBalance = balance + sum;
            sleep(1);
            balance = lastBalance;
        }

        /**
         * テストのための遅延用メソッド
         * @param ミリ秒数
         * @return なし
         */
        private void sleep(int msec) {
            try {
                Thread.sleep(msec);
            } catch (InterruptedException ex) {
                System.out.println("中断された");
            }
        }
}

/**
 * ＡＴＭからの取引トランザクション
 * 取引とは、あるＡＴＭから同じ口座に対して行った連続した操作（預入／引出）の集合
 */
public class UserTransaction implements Runnable {

    private BankAccount account;
    private int[] sumList; //続けて行った預入／引出の金額のリスト

    /**
     * コンストラクタ
     * @param 対象となる預金口座
     * @param 金額のリスト（続けて行った預入／引出の金額のリスト）
     */
    public UserTransaction(BankAccount account, int[] sumList) {
        super();
```

```
        this.sumList = sumList;
        this.account = account;
    }

    /**
     * 続けて行った預入／引出を実行する
     * @return なし
     */
    public void run() {
        if (account != null) {
            for (int i=0; i<sumList.length; i++) {
                int sum = sumList[i];
                if (sum < 0) {
                    account.withdraw(-sum); //金額が負の場合は引出し
                } else {
                    account.deposit(sum); //金額が非負の場合は預入れ
                }
            }
        }
    }
}
```

5.27 パッケージ

5.27.1 import

(例１)

```
class Date1 {
   public static void main (String[] args) {
      java.util.Date    now = new    java.util.Date();
      System.out.println( now );
   }
}
```

(例２)

```
import java.util.Date;
class Date1 {
   public static void main (String[] args) {
      Date    now = new    Date();
      System.out.println( now );
   }
}
```

5.27.2 package 宣言

（例１） package 宣言がない場合 → 無名パッケージ

```
class ClassA extends Object {
    int name;
    public ClassA() {
        name = 1;
        System.out.println ("コンストラクタ ClassA()が実行された");
```

```
    }
}
```

(例 2)　package 宣言する

```
package demo;
public class ClassA extends Object {
    private int name;
    public int getSameName() {   //アクセサメソッド
        return (name);
    }
    public ClassA() {
        name = 1;
        System.out.println ("コンストラクタ ClassA()が実行された");
    }
}
```

(例 3)　public な class、interface

```
package demo;
public class ClassA extends Object {   //public なクラスはファイル（.java）に唯一
    private int name;
    public int getSameName() {   //アクセサメソッド
        return (name);
    }
    public ClassA() {
        name = 1;
        System.out.println ("コンストラクタ ClassA()が実行された");
    }
}
public interface MyInterface {
public int getInt();
}
```

```
public なクラス interface demo.MyInterface は、ファイル "MyInterface.java" で定義されなければなりません。
public interface MyInterface   {
                               ^
```

エラー 1 個

5.27.3 package 名

JP.co.xxx.demo
com.xxx.example

5.27.4 スコープ

アクセス修飾子なしで宣言されたメンバは同じパッケージ内のコードからのみアクセスできる。

public クラスあるいはインタフェースは、パッケージ外からアクセスできる。

public でないタイプはパッケージ内スコープのみを持つ。つまり同じパッケージ内の全プログラムから使うことができるが、パッケージの外からは隠され、ネストしたパッケージからもアクセスできない。

5.27.5 UML 表記

5.28 JDK 開発キット（JDK1.2、Java2 SDK）

5.28.1 Javaコンパイラ

javac　.java から .class を生成する。

5.28.2 Javaインタプリタ

java　.class を解釈実行する。

5.28.3 C言語用ファイル生成コマンド

javah

5.28.4 逆アセンブラ

javap　.class を逆アセンブルする。

5.28.5 ドキュメント生成コマンド

javadoc　HTML 形式のドキュメントを生成する。

5.28.6 デバッガ

jdb

5.28.7 クラスパッケージ

Java core API　（java.ではじまる名前のパッケージ群として JDK で提供されるＡＰＩ）

java.io　ストリーム入出力、ファイルシステム用。

java.lang　自動的にインポートされ、Java 自身を管理するためのクラスや、Obect クラス等を含む。

java.util　可変配列や日付などのユーティリティパッケージ

6. JavaSE と JavaEE (JakartaEE)

6.1 違い

例	JavaSE	JavaEE
javax.swing パッケージ	含む	ない
javax.servlet パッケージ	ない	含む

6.2 JavaSE

- OpenJDK
- Adopt Open JDK … OpenJDK をビルドしたもの
- Oracle JDK

JDK は開発環境を指し、コンパイラを含む。そのコンパイラと JRE が参照実装。

6.3 JavaEE / JakartaEE

- OracleEE
- OpenEE
- JakartaEE … Java EE を Oracle から Eclise Foundation に移管
- GlassFish … Java EE の参照実装。Eclipse Foundation 2022.1.10 Latest
- Eclipse GlassFish … Jakarta EE の参照実装
- Tomcat … Web コンテナ機能のみ。JSP Servlet の参照実装。Apache 2022.4.1 Latest
- WebLogic, Jetty, WebSphere など

6.4 JavaEE サーバの構成要素

- Web コンテナ … Servlet コンテナ Servlet エンジン
- EJB コンテナ

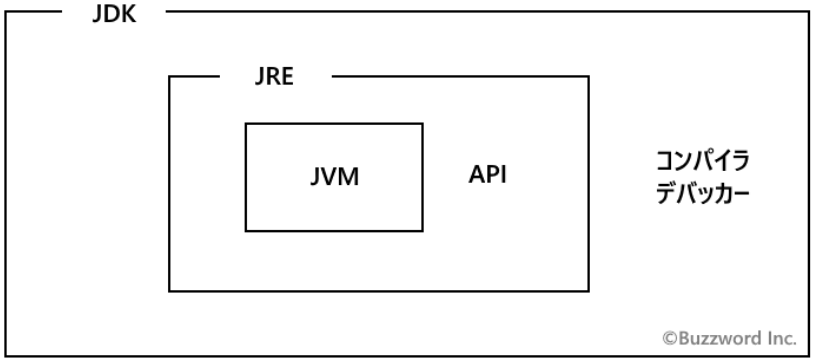
6.5 Spring

実行には Web コンテナ（Tomcat など）を必要とし、EJB コンテナは使用しない。

サーバに必要なトランザクション機能や永続化機能、DI 機能を提供する。

Struts2 や SAStruts も同様。

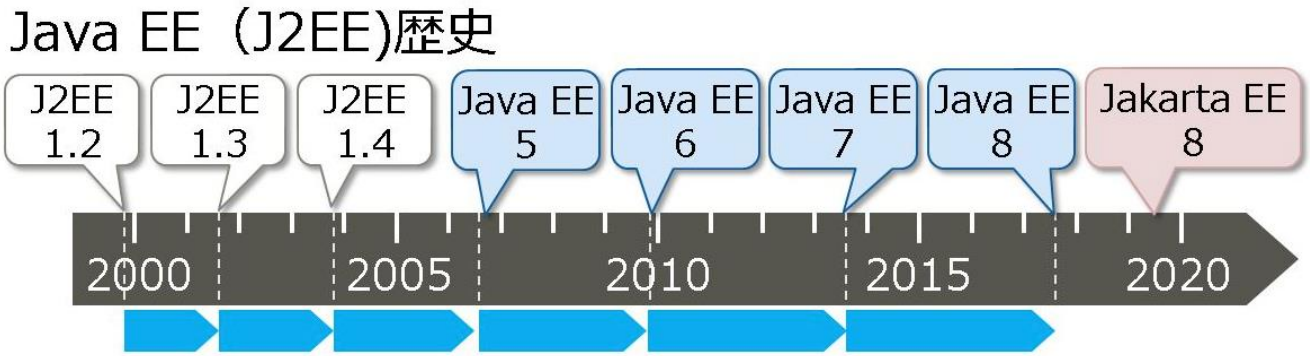
6.6 JDK JRE JVM



上図は https://www.javadrive.jp/start/install/index5.html から参照

6.7 仕様策定

JCP …Java Community Process
EclipseFoundation



上図は富士通社ブルグから

Spring Boot
2014, ver1.0
Spring Framework（2002，アンチ J2EE,ロッド・ジョンソン）が成長し、ある面、肥大化したために登場。

Struts2 …Apache Struts
Spring Framework
JSF …Jave Server Faces
Java EE …
Play Framework …
Apache Wicket …UI に特化したフレームワークです。Java らしいコードを書いて WEB アプリケーションを作る

7. オブジェクト指向技術の導入

7.1 はじめて導入する

7.1.1 導入の動機

(1) 生産性・品質の向上

(2) 再利用の期待

(3) ユーザ（発注元）の要求

(4) トレンド

(5) 複雑な要件への対応

7.1.2 簡単なシステムを動かす

(1) 豊富なサンプル（雑誌、インターネットなど）

(2) 導入効果を実感できるか

7.1.3 従来技術の未経験者の場合

(1) 特に疑問はない

7.1.4 従来技術の経験者の場合

(1) デメリットを感じることが多い　（もっと簡単にできるが、なぜ）

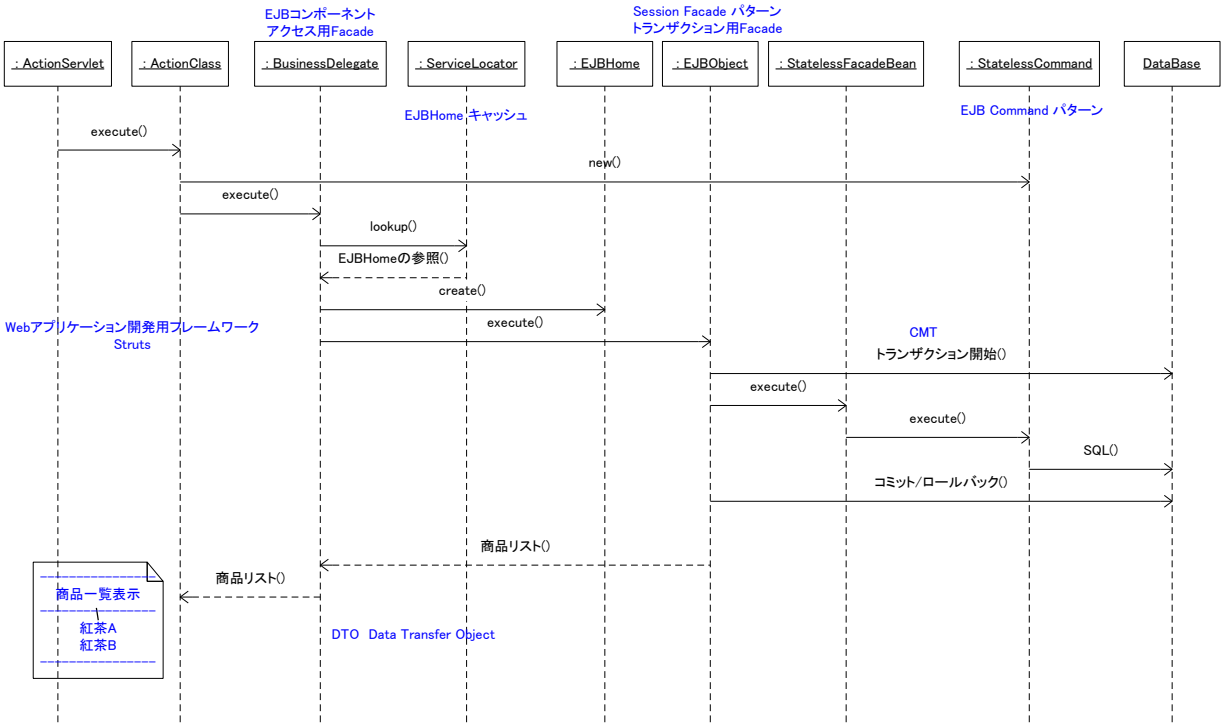
(2) パラダイムシフトを敬遠する　（覚えなくていけない概念、技法、ツールなどが、結果に比べて必要以上に多いように感じる）

(3) 学習コストは認められない　（時間がない。 保守しなければならない従来技法のシステムが残っている）

(4) 導入目的を確信できない　（新しい技術を習得するだけのメリットを感じない）

(5) 複雑になった　（オープンソース、オープンなデザインパターンなど組み合わせや、役割が複雑）

7.1.5 サンプル シーケンス図「商品一覧を表示する」



7.2 OOA か DOA か

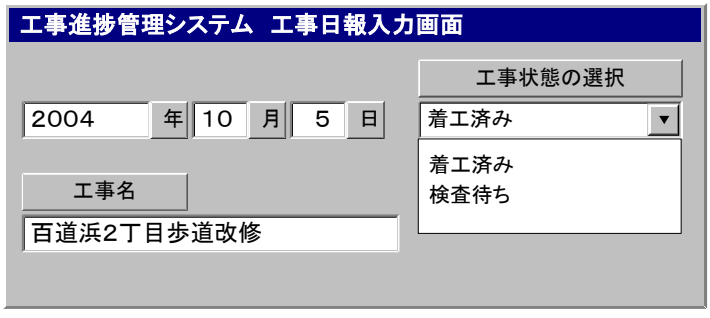
システムを開発する場合、使用するアーキテクチャを決めければなりません。その中で、OOA（Object Oriented Analysis）を使うのか、DOA（Data Oriented Approach）を使うのかを決める必要があります。

7.2.1 DOA

特に業務アプリケーションと呼ばれる分野では広く使用されています。リレーショナルデータベースとの相性がよい手法です。しかし、Java や C#といったオブジェクト指向言語の利点や効果を十分に引き出せる手法ではありません。インピーダンスミスマッチ、O/R マッピングといった問題がトラブルになるケースも少なくありません。

7.3 設計モデルを作成する

この章では、道路工事の進捗管理システム（Web アプリケーション）の開発プロジェクトを想定します。このシステムには、工事の状態（「着工承認待ち」、「着工待ち」、「着工済み」など）を設定する画面がいくつかあります。



状態遷移にはルールがあります。例えば、「着工承認待ち」からは「着工承認待ち」または「着工待ち」に設定（遷移）できます。「着工済み」からは「着工済み」または「検査待ち」に設定（遷移）できます。

ここで、いくつかの設計例を挙げます。

7.3.1 設計例1

(1) JSP + サーブレット + リレーショナルデータベースを使用します。

(2) 画面毎に JSP、サーブレットが各１つあります。

(3) サーブレットは直接データベースにアクセスします。

(4) リクエストに対する制御を（Struts 等を使用しないで）作成します。

(5) データの意味、扱い方は各サーブレットに書かれていることになります。

7.3.2 設計例2

(1) MVC の VC フレームワーク + リレーショナルデータベースを使用します。

(2) フレームワークから呼び出される（フレームワークに含まれる）クラスのサブクラスを画面毎に作成します。

(3) リクエストに対する制御はフレームワークが行います。

(4) 各サブクラスは必要な場合、SQL コードを含みます。

7.3.3 設計例3

(1) MVC の VC フレームワーク（Struts など）+ Data Accessor Object + リレーショナルデータベースを使用します。

(2) SQL コードは Data Accessor Object が隠蔽します。

(3) Data Accessor Object はテーブルレコード（データオブジェクト）を応答します。

(4) テーブルレコードの扱い（列の意味）は、Data Accessor Object の呼出し側に依存します。

7.3.4 設計例4

(1) MVC の VC フレームワーク + MODEL + Data Accessor Object + リレーショナルデータベースを使用します。

(2) Data Accessor Object は（テーブルレコードではなく）MODEL オブジェクトを応答します。

(3) Data Accessor Object はテーブルレコードから MODEL オブジェクトを生成し応答します。従って、データベースは隠蔽されることになります。

(4) 隠蔽には手間がかかり、その効果が見出し難いといえます。（特に初期段階）

(5) MODEL オブジェクトがほとんどテーブルレコードと一致し、setter/getter だけのデータオブジェクトになりやすい傾向があります。

(6) 各画面の項目もテーブルレコードとほとんど一致するため、MODEL 隠蔽の効果は分かり難いです。

7.3.5 設計例5

(1) MVC の VC フレームワーク + MODEL + STATE パターン + Data Accessor Object + リレーショナルデータベースを使用します。

7.4 パターンを導入する

次に、Model オブジェクトを使うメリットを考えます。

工事の状態の設定画面では、次に遷移可能な状態だけを含むリストが表示されるものとします。設定可能な工事の状態リストは画面毎には決められず（固定ではなく）、対象とする工事の現在状態によって決まるものとします。以下、この要件を実現する方法を考えます。

7.4.1 ケース1　設計例1～4のサーブレットの中や JavaBeans の中のコードで実現

（擬似コード）

IF　現在の状態が「着工承認待ち」 THEN 次に遷移可能なリストに、「着工承認待ち」／「着工待ち」をセットする

ELSE　IF　現在の状態が「着工済み」 THEN 次に遷移可能なリストに、「着工済み」／「検査待ち」をセットする

・・・・・・以下省略

（特徴）

（1） 各画面に関係を現すコードが分散する。

（2） バグの原因となりやすい。保守し難い。

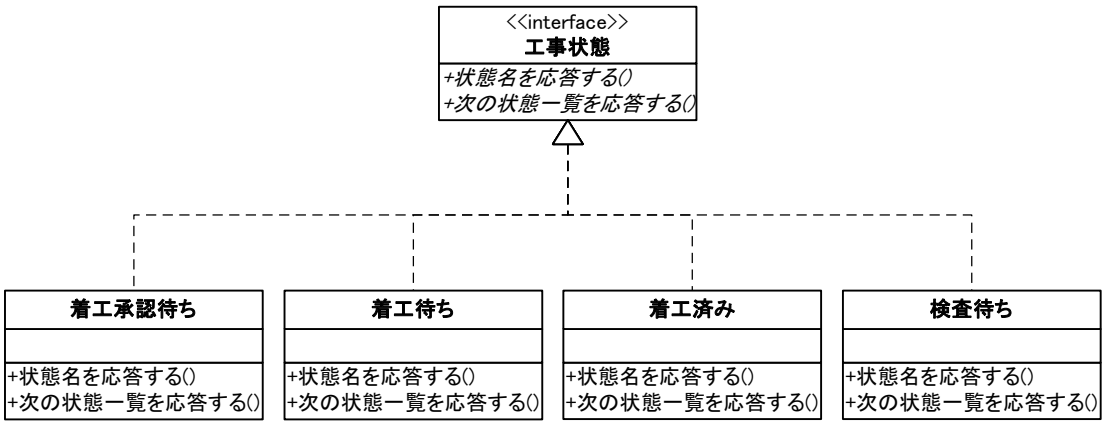
7.4.2 ケース2　各状態と各状態から遷移可能な次の状態の関連をもつ（リレーショナルデータベース）テーブルを持つ

（特徴）

（1） 保守が難しい。（状態の追加・削除・移動が難しい）

（2） デバッグし難い。

7.4.3 ケース3　デザインパターン（State パターン）を使用する



（特徴）

（1） 分かり易い、変更しやすい、追加しやすい。

（2） コメントや UML 図にパターン名を書くことで他者が理解しやすい。

（3） 実装方式として枯れている。

（4） 使用するには設計モデルに書かれていなければならない。実装段階では間に合わない。

7.4.4 OOA ではなく DOA を選択すべき場合

（1） オブジェクト指向の経験者がいない。

（2） あまり複雑ではないシステム。

7.4.5 DOA と Java を選択した場合の留意点

（1） 機能とデータが分離するため、コンポーネント化は難しい。

（2） テーブルスキーマの変更が広範囲に及ぶような構造になりやすい。

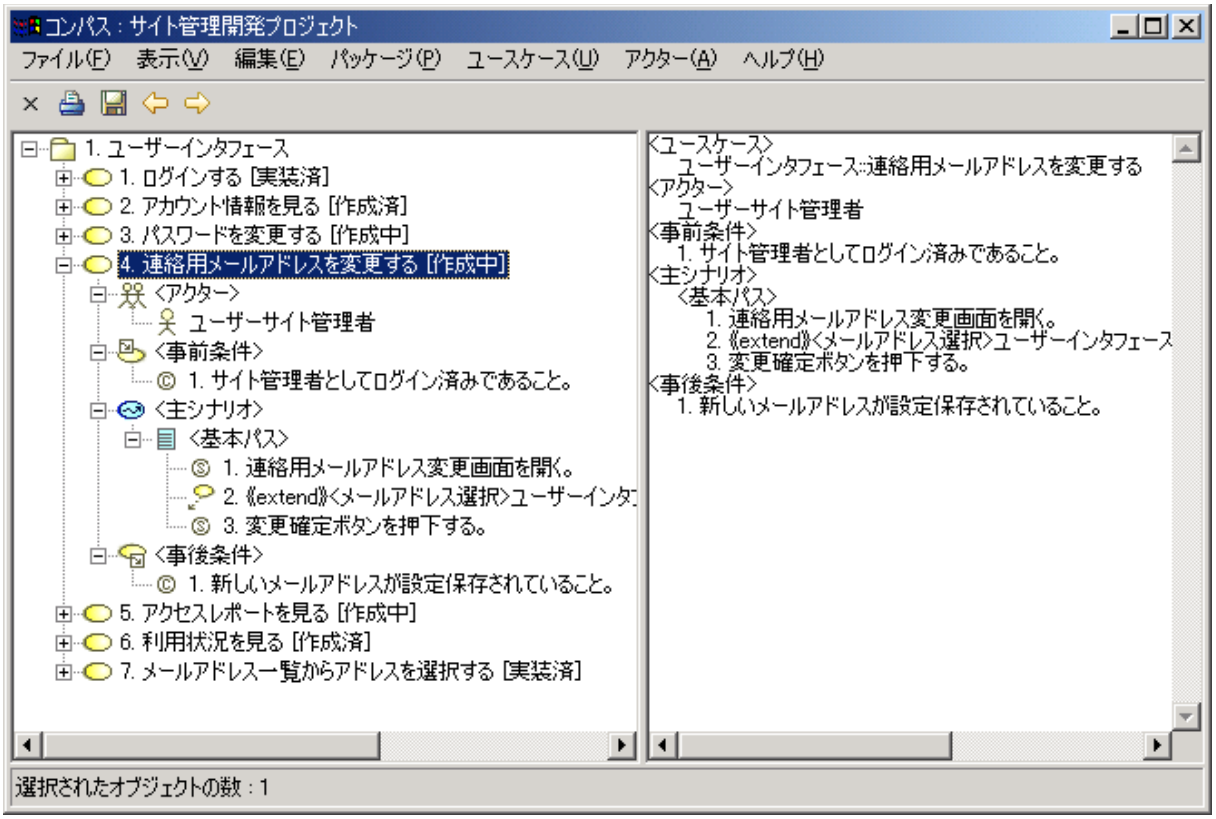
7.5 MVC の MODEL とその責務

7.5.1 例題について

「ユースケース・モデリング・ツール Compass」をサンプルにします。Compass は GUI を持つ Java アプリケーションで、この中で、入力値の妥当性チェックをどのように実装するのかを検討します。

7.5.2 Compass のユーザーインタフェース画面（GUI）

（1） 主画面



（2） 利用者は以下の手順で、拡張ユースケースを設定できます。

①パスを選択する。

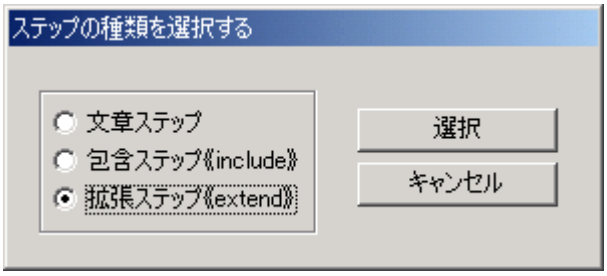
②ステップを追加する。

③ステップの種類として拡張ステップを選択する。

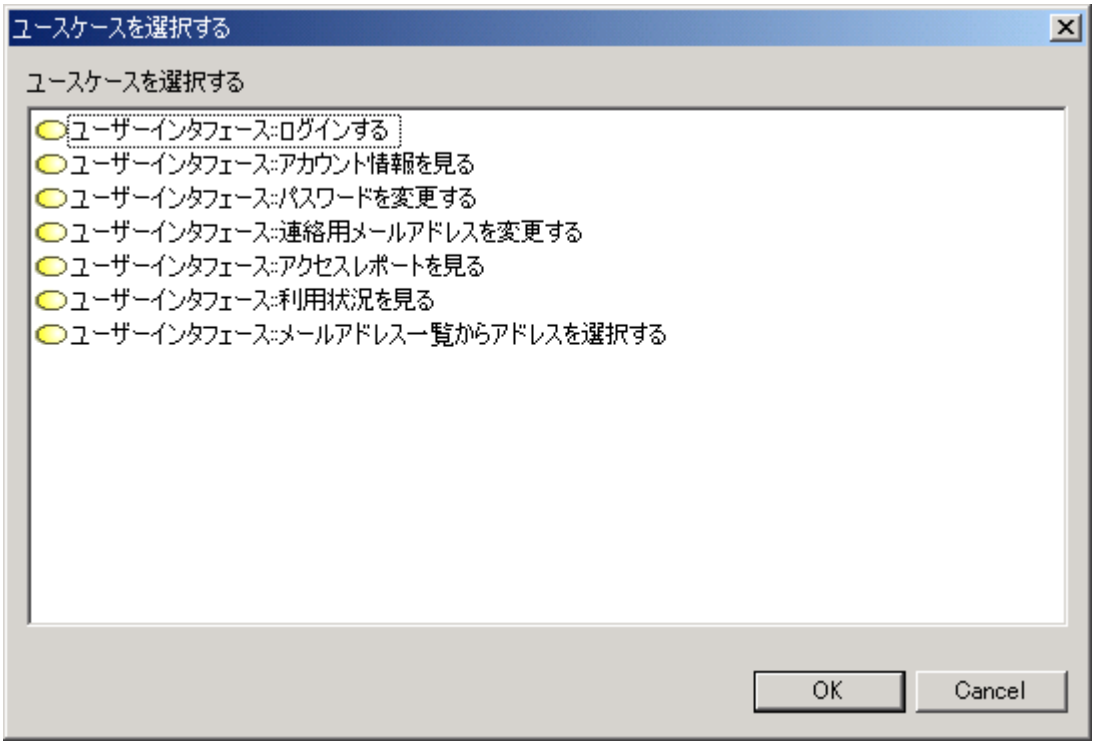
④拡張点の名前を定義する。（省略）

⑤ユースケーの一覧から拡張ユースケースを選択する。

（3） ステップの種類を選択画面



（4） 拡張ユースケースの選択画面



7.5.3 拡張ユースケースとしての正当性チェックの方法

拡張ユースケースとして自分自身を選択することはできませんので、パスが選択されている状態で、次のチェックが必要になります。

（1） ケース１

パステーブル、シナリオテーブル、ユースケーステーブルまたは、パス－シナリオ関連テーブル、シナリオーユースケース関連テーブルを見て判定します。

（2） ケース２

Path#getUsecase() → Scenario#getUsecase() で判定します。

パスオブジェクトは自分が含まれているユースケースを、自分を直接含んでいるシナリオオブジェクトに問い合わせます。シナリオオブジェクトは自分自身を直接含んでいるユースケースオブジェクトを応答します。

7.5.4 正当性チェックのルールが変わったときの影響

例えば、Path#isExtendableUsecase(Usecase usecase)メソッドをパスクラスに追加したとします。すなわち、あるユースケースが拡張ユースケースとして正当であるか否かの判断は、パスオブジェクトの責務とします。

ここで、ケース１とケース２を比較してみましょう。変更個所を特定する作業の容易性、変更の局所性、変更自体の容易性、変更のリスク、変更によって必要になるテストケースの数、将来の保守性などを比較すればケース２の方が優れていると言えます。

＊＊＊　終り　＊＊＊