

Chapter 2-01

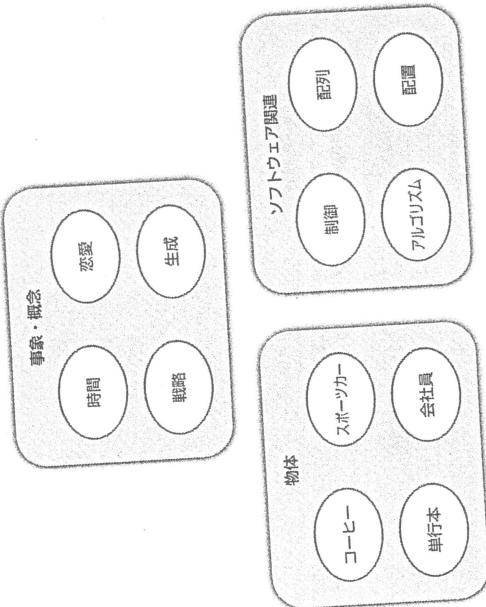
オブジェクト指向とは

UMLは、オブジェクト指向をベースとしたモデルング言語です。ではオブジェクト指向とは何でありますか？本書の読者の方々は、一度は耳にしたことがある言葉だと思います。ここでは、オブジェクト指向の1つ1つの概念を説明していきます。

オブジェクトとは

英和辞書や国語辞書で調べてみると「オブジェクト」＝「モノ」、「物体」、「指向」＝「～を目標として向かうこと」と載っています。繋げて話を補うと「オブジェクト指向」とは「モノ（物体）を目指して把握する考え方（バラダイム）」ということになります。
では、モノとはなんでしょうか？コーヒー、スポーツカー、単行本、会社…色々と挙げられるとと思います。それでは、時間はどうでしょうか？恋愛はモノと言えるのでしょうか？オブジェクト指向では、時間や恋愛もモノとして分類されます。
オブジェクト指向においては、目に見える物体はもちろんのこと、そのほかにも事象や概念もすべて含めてオブジェクト（モノ）として捉えます。

図2-1-1 オブジェクトのイメージ



これらすべて、モノ（オブジェクト）として捉える

オブジェクトは、自分自身の性質や状態を表す属性と、自分自身や外部に対しての相互作用を表す操作を持っています。例えば銀行システムの口座オブジェクトを考えてみると、名義人として「山田さん」という性質や、預金残高として「¥1,000,000」といった状態を表す属性を持っています。また、外部に対して「預金を引き出す」、「預ける」、「振り込む」、「残高照会する」という操作を提供しています。

図2-01 オブジェクト指向とは

図2-1-2 口座オブジェクト



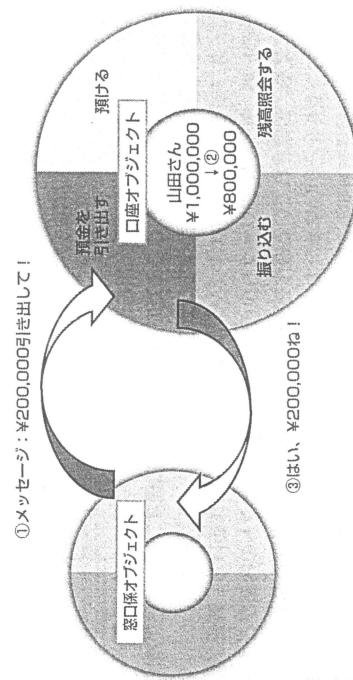
Part2
Part2

Part2
Part2

オブジェクト間のやり取り

オブジェクトは操作を呼び出すことで、他のオブジェクトに要求を出すことができます。例えば、窓口係オブジェクトが口座オブジェクトに対して「預金を引き出す」という要求を出すことができます。オブジェクト指向では、この要求を「メッセージ」、要求を出すことを「メッセージパッシング」と呼びます。

図2-1-3 メッセージパッシング



実際のシステムは、多数のオブジェクトから構成されます。そして、オブジェクトが他のオブジェクトにメッセージパッシングすることをシステムの要件機能を実現します。つまり、オブジェクトはメッセージをやり取りすることで協調動作し、目的を達成するのです。

図 2-1-4 システム全体でのメッセージパッシング

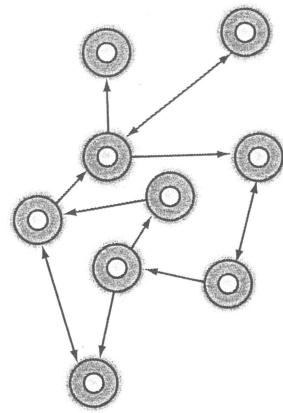
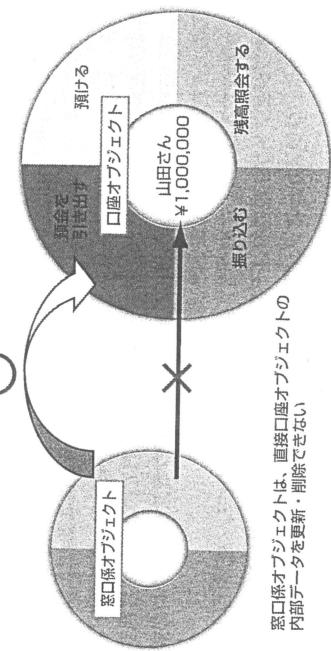


図 2-1-5 カプセル化

必ず口座オブジェクトのメソッドを使って、内部データを更新・参照する。

Part2
2-01

カプセル化

先ほどの例では、窓口係オブジェクトが口座オブジェクトに対して「預金を引き出す」というメッセージパッシングをすることによって、口座オブジェクトの預金を引き出すことができました。これは、オブジェクト指向における重要な考え方の 1 つです。オブジェクト同士は、必ずお互いの操作を通して属性を操作する必要があります。口座オブジェクトの属性である名義人や預金残高は、口座オブジェクト自身が責任をもつて管理をします。窓口係オブジェクトが直接勝手に預金残高を操作することはできません。窓口係が預金残高の増減操作を行いたい場合には、必ず口座オブジェクトの操作を呼び出すことになります。

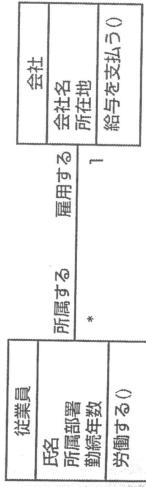
これは一見すると不便なようにも思えるのですが、実は良く考えてみると、窓口係オブジェクトは口座オブジェクトの内部詳細（例えば預金残高）を知らなくてはならないことがあります。窓口係オブジェクトは「預金を引き出す」という、口座オブジェクトの動きを知つてさえいれば、預金を引き出せるのです。もちろん預金残高が足りない場合は口座オブジェクトが預金残高をどう伝えなければなりません。しかしここでのポイントは、窓口係オブジェクトが預金残高を気にすることなく、預金を引き出す操作を行うことができるという点です。このように、オブジェクト指向ではオブジェクトごとに属性や操作を持ち、責任が分割されています。そのため、外部のオブジェクトからは、対象のオブジェクトの内部がどうなっているかを知らないでも、公開されている操作を実行して何らかの利益を受けることができます。この考え方をカプセル化といいます。カプセル化によって、外部のオブジェクトは利用しているオブジェクトの内部構造を知る必要がなくなります。そのため、オブジェクトの内部構造が変更されても外部には影響が及ばなくなります。

薬局で売っている風邪薬のカプセルを思い浮かべてください。カプセルの中身である薬の詳細は知らないとも、これを飲めば風邪が治るということを知っているのと同様です。

クラス

オブジェクトを抽象化した枠組みをクラスと呼びます。オブジェクトはクラスを具体化することだけで作成されます。「田中さんと佐藤さんは株式会社テクノロジックアートに勤務して、労働の見返りに給料をもらっています。」という内容からクラスを定義し、UML で表記すると、次のようなモデルになります。

図 2-1-6 従業員－会社クラス図



カプセル化
encapsulation。カプセル化によりオブジェクトの詳細を他のオブジェクトから隠蔽することができます。これにより詳細データの扱いを局所化することができます。

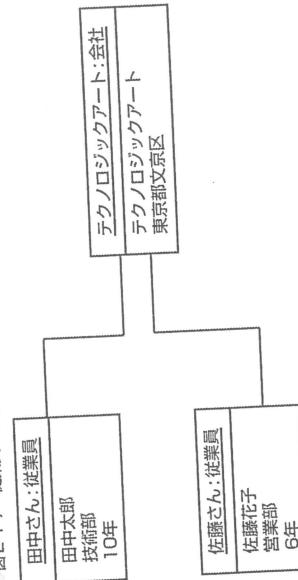
Part 2 オブジェクト指向の基礎

ここではUMLの表記法の詳細に聞いては触れませんが、「田中さん」「佐藤さん」は氏名や所属部署などの属性や、労働するという操作が共通であり、それらを抽象化して「従業員クラス」が定義されています。「株式会社テクノロジックアート」からは「会社」というクラスが定義されています。

「田中さん」と「佐藤さん」というオブジェクトを抽象化する場合、「人間クラス」を定義することもできます。また「会社クラス」をモデル化する場合、資本金という属性を持たせることもできます。このようにクラスは、対象領域の見方によってその定義が変わります。言い換えると、事象を抽象化する場合には、対象領域で必要とされている特性に注目するとともに、現在注目している問題領域や機能に対して適切な大きさでクラスを作成する必要があります。

このクラスの大きさをクラスの粒度と呼びます。このクラスの粒組みから、田中さん・佐藤さん・テクノロジックシステムが動作する時には、このクラスの粒組みから、田中さん・佐藤さん・テクノロジックアートのオブジェクトが生成され、それぞれのオブジェクトが協調しながら動くことになります。このように、クラスからオブジェクトを生成することを「インスタンス化する」といいます。

図2-1-7 従業員-会社オブジェクト図

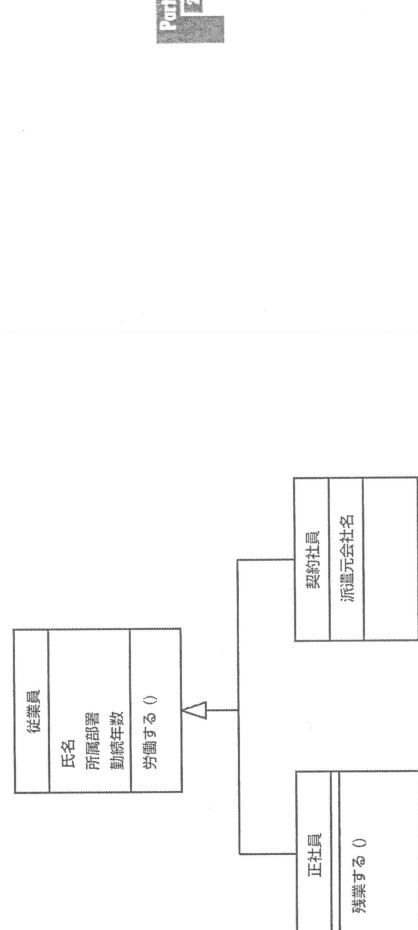


：継承・汎化

前節では、オブジェクトを抽象化したクラスについて説明しました。前節の例では、田中さん、佐藤さんはどちらも従業員クラスとしてモデル化しましたが、実は田中さんは正社員、佐藤さんは契約社員だったとします。そして、正社員は残業することができますが、契約社員は残業できないという決まりがあるとします。このような場合、従業員としての共通概念を残しつつ、差分を派生させたクラスを定義することができます。

既存のクラスの概念（属性や振る舞い）を利用して、さらに概念を追加して新しいクラスを定義することができます。より大きなくくりとなるのクラスから共通の概念をまとめあげて、より一般的なクラスを定義することができます。クラスの理解をしやすくなります。

図2-1-8 従業員クラス（継承クラス図）



ここでもUMLの詳しい説明に聞いては触れませんが、上の図は従業員クラスの概念をすべて受け継いで、正社員クラスと契約社員クラスを定義していることを示しています。このように、より一般的な概念のクラスから概念を引き継ぐことを「継承」と呼びます。正社員クラス、契約社員クラスは、どっちにしても従業員クラスを継承しているので、氏名・所属部署・勤続年数という属性や、労働するという振る舞いも持っていることになります。さらに正社員クラスは、残業するという振る舞いも持っていることを表しています。

逆に正社員クラスや契約社員クラスからみると、従業員クラスは共通の概念をまとめ上げています。このように共通の概念をまとめあげて、より一般的なクラスを定義することを「汎化」と呼びます。

継承や汎化の概念は、オブジェクト指向のもつとも基本的な考え方の1つです。これらの概念によって、物事をまとめて捉えたり、拡張する場合の手間を減らしたりすることができます。継承や汎化を使ってクラスを洗練していくアプローチは、人間の思考回路が事象や物事を分類、体系化して整理するプロセスに非常によく似ています。

：可視性ヒインターフェース

オブジェクト指向における重要な考え方の1つとしてカプセル化を説明しました。カプセル化とは、詳しい内部情報や処理を局所化するとともに、外部から隠蔽することでした。そして外部に対しては、必要な操作を公開することで、「私（オブジェクト）は***しますよ」と明示しました。この外部に対して公開したクラスの仕様をインターフェースといいます。

Part 2 オブジェクト指向の基礎

Chapter 2-01 オブジェクト指向とは

図 2-1-9 インド料理人クラス図

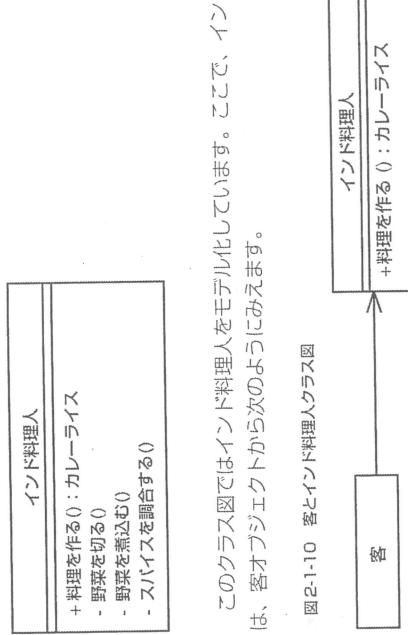
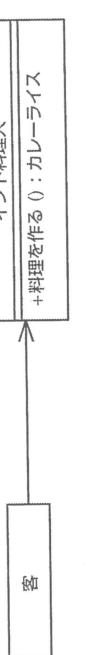


図 2-1-10 客とインド料理人クラス図



このクラス図ではインド料理人をモデル化しています。ここで、インド料理人オブジェクトからは、インド料理人オブジェクトの操作は「料理を作る()」しか見えません。インド料理人オブジェクトは、「料理を作ります。私が作るのはカレーライスですよ。」とか外部オブジェクトに対して公開していないからです。このように、外部に対して公開している部分が、インド料理人クラス（オブジェクト）のインターフェースとなります。オブジェクト指向では、クラスの属性や操作を内部に隠蔽するか、外部に公開するかは「可視性」という概念を利用して表現します。客は、カレーライスを作る具体的な工程（野菜を切る、スパイスを調合する…）を知りません。また厨房に顔を出して、料理方法や材料に口を出すことでもできません。

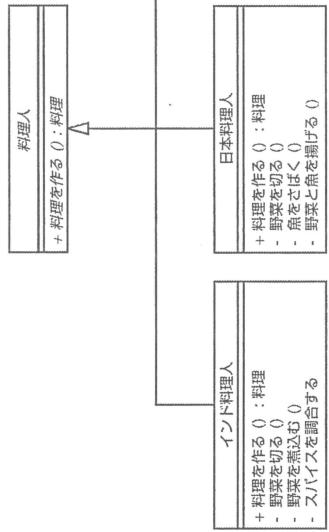
このように、内部に隠蔽する属性や操作の可視性はプライベート（private）となり、属性名やメソッド名に“-”をつけて表記します。逆に外部に公開する属性や操作の可視性はパブリック（public）となり、属性名や操作名に“+”をつけて表記します。

：多態性（ポリモーフィズム）

先ほどの例で出てきたインド料理人クラスを汎化して、料理人クラスを作成します。さらに料理人クラスを継承した日本料理人クラスとイタリア料理人クラスと、日本料理人クラスか、料理人クラスは汎化によって抽象度が上がっているため、カレーライスではなく料理を結果として返すことにします。

- 可視性 属性や振る舞いを内部に隠蔽するのかインターフェースとして外部に公開するのかを表現します。
- 同一のメッセージに対して、受け取った側が状況に応じて振る舞いを変えること。多態性は、利用する側の再利用性を促進します。

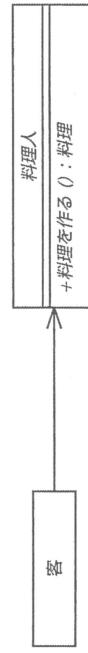
図 2-1-11 料理人クラス図



料理人クラスはイタリック表示で記述されています。これは料理人クラスが、インスタンス化されない概念的なクラスである「抽象クラス」であることを表しています。抽象クラスは概念的なクラスなので、実際の操作を行うことができません。つまり、料理人クラスには料理を作るという操作がありますが、実際料理を作ることができません（プログラムの処理は書きません）。このイタリック体の操作「料理を作る」は、料理人クラスを継承する派生クラスで、必ず何らかの料理（カレーライスかも知れませんし、天ぷら、パスタかも知れません）を作るように処理の中身が定義される必要があります。

料理人クラスのインターフェースは、図 2-1-12 のようになります。客クラスはインド料理人クラスや日本料理人クラス、イタリア料理人クラスの詳細を知らないとも料理人クラスという括りで料理を頼むことができます。

図 2-1-12 客と料理人クラス図



客クラスは料理人クラスに対して「料理を作る」というメッセージを送ります。料理人クラスは、その時にインスタンス化されているのがインド料理人クラスか、日本料理人クラスか、イタリア料理人クラスかによって、動的にカレーライスを料理するのか、和食を料理するのか、イタリアンを料理するのかを決定します。これは、料理人がその時々の状況に応じてさまざまな振る舞いをするともいえます。

このように同じ振る舞いを呼び出しても異なる機能を果たすことを「多態性」といいます（英語でポリモーフィズムと呼ばれることもあります）。この多態性は、オブジェクト指向の目指す目標の 1 つであるアプリケーションの再利用性向上のための重要な概念です。