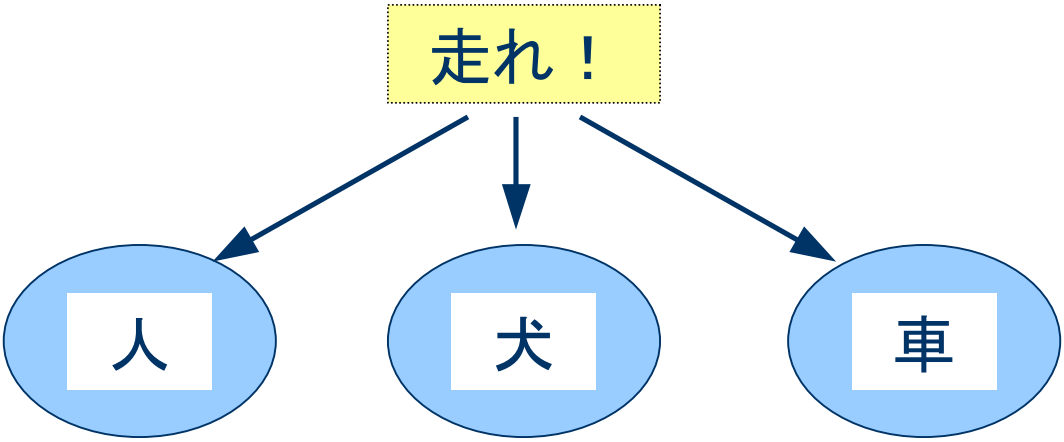
3　オブジェクト指向プログラミング (Object Oriented Programming)

オブジェクト指向プログラミングの歴史は、1968 年に発表されたプログラミング言語 SIMULA67 から始まりました。C 言語が世に出る 4 年以上前のことです。
SIMULA67 をベースに、あらゆるオブジェクト指向プログラム言語の基本となった Smalltalk が発表されたのは 1972 年です。
C++言語は 1980 年に開発されましたが、国際標準ができたのは 1997 年のことです。
そして Java 言語が作られたのは 1990 年です。

ここではオブジェクト指向プログラミングに重要な 5 つの概念、**『抽象化』『隠蔽』『継承』『多態性』『委譲』**について、特に Java にフォーカスして概要をまとめます。

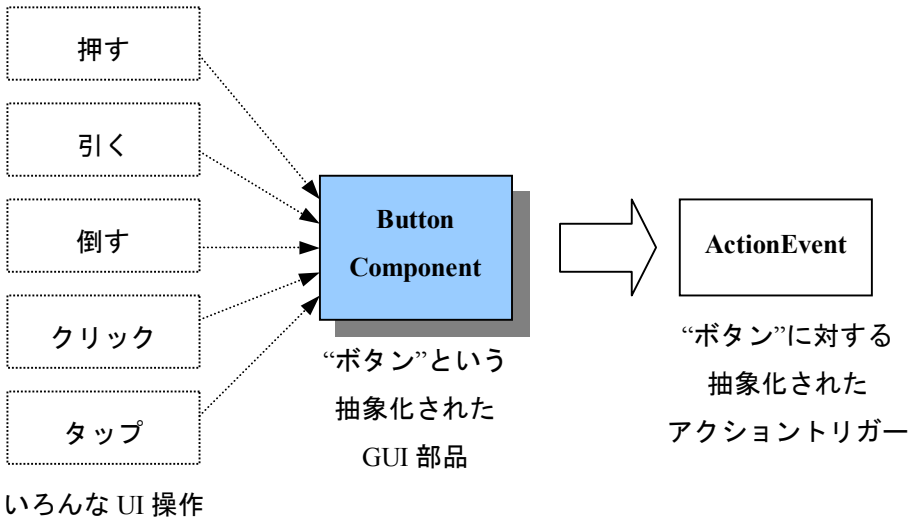
3.1 抽象化 (Abstraction)



『プログラマは全てを知っていなければならない』という旧来のプログラミングパラダイムに対して、『概念プログラミング』という進化をもたらすためにもっとも重要な概念が抽象化です。

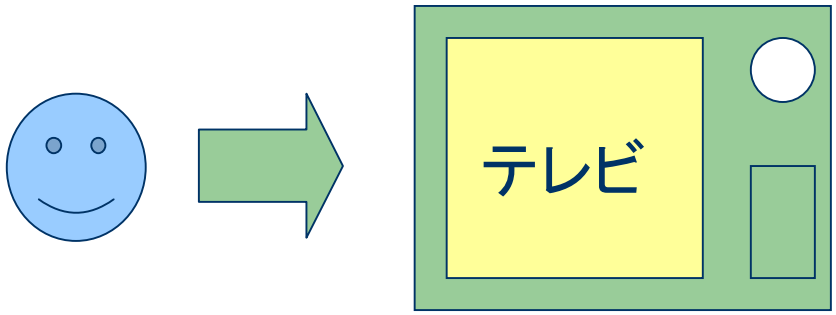
プログラマは『ボタン』という抽象的なコンポーネントを用いてアプリケーションプログラムを記述することができ、『ボタン』の内部や、『ボタン』が動作するためのシステムサービス、オペレーティングシステム、コンポーネントイベントモデル、などの内部を知る必要がありません。

『ボタン』がどのような描画イメージを持っているか、どのようなプッシュアクションをするのかとは無関係に、『ボタンコンポーネントは何らかのアクションを発生させる』という抽象機能のみで、利用することができるようになります。



3.2 隠蔽 (Encapsulation)

隠蔽とは『外部に対して提供するサービス以外は、外部から見えないようにする』という仕組みです。



テレビの利用者は、『テレビ』の中身がどのようになっているかを知ることなく、利用することができます。

テレビを利用するために、電子回路、電子物性工学、電磁波動方程式を理解する必要はありません。

例え中身が入れ替わったとしても、『テレビ』としては以前と同様に利用することができるでしょう。

クラスプログラミングでは、データの実体は外部から隠蔽されるべきです。

クラスが保持するデータは(たいてい)何らかの実装依存表現形式であり、外部に提供するには不適切です。

クラスは外部に対して『何らかの機能』を提供すべきです。

3.3 継承 (Inherit)

継承はオブジェクト指向というパラダイムにおいて作り出された概念です。

あるモノを改造しようとする場合、モノそれ自体を改造するのではなく、そのモノとの差分だけを付け加えることができれば便利です。

そのための概念を継承と呼びます。

オブジェクト指向では、継承を 2 つのパターンに分類します。

Java 言語では Interface 継承が**型の継承**、Class 継承が**実装の継承**に分類されますが、Class 継承を型の継承として利用することもできます。

C++言語では、public 継承が**型の継承**、private / protected 継承が**実装の継承**に分類されます。

3.3.1スーパークラス (基底クラス)

継承を定義する場合、継承の元となるクラスを、Java ではスーパークラス、C++言語では、基底クラスと呼びます。

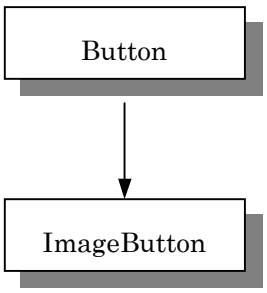
3.3.2サブクラス (派生クラス)

継承を定義する場合、継承したクラスを、Java ではサブクラス、C++言語では、派生クラスと呼びます。

3.3.3実装の継承 (Implementation Inheritance)

実装の継承は"is-a"関係ではない継承であり、構文的継承(syntactic inheritance)、アルゴリズムの継承とも呼ばれます。

実装の継承の目的は、元になるモノが既に持っている実装を再利用しつつ、新たな独自実装を付加することにあります。



ImageButton クラスは、Button クラスのすべての機能を持っており、加えて「文字ラベル」だけでなく「画像ラベル」も使用できるボタンコンポーネントです。したがって画像の描画処理だけが追加されています。

オブジェクト指向技術が広まりつつあった頃に、オブジェクト指向の特徴として喧伝されたのが実装の継承ですが、現在では、実装の継承は必ずしも良い方法ではないと認識されるようになっています。

なぜでしょう？　隠蔽化との両立について考えてみる。

3.3.4型の継承 (Type Inheritance)

型の継承は”is-a”関係であり、インターフェイスの継承(Interface Inheritance)、意味継承(Semantic Inheritance)、振る舞いの継承(Behavioral Inheritance)、仕様の継承とも呼ばれます。

型を継承したサブクラスは、親のクラスと「同じ穴のムジナ(同類)」であることを意味し、かつ、親のクラスを特化(Specialize)しています。

型の継承は、オブジェクト指向において重要な概念である抽象化の実現に大きく関わっています。

“is-a”関係とは？

実装の継承との具体的な違いは？

java.util.Stack は java.util.Vector を継承しています。
これはどのような種類の継承でしょう？
java.io.IOException は java.lang.Exception を継承しています。これは？

3.3.5汎化 (Generalization)

実装の継承において、スーパークラスが提供する実装の観点は**汎化**です。

例えば ImageButton クラス、ToggleButton クラス、RoundButton クラスなどに対して、スーパークラスの Button クラスが実装すべき機能は、ボタン機能に関する共通処理的な実装になります。

3.3.6特化 (Specialization)

実装の継承あるいは型の継承において、サブクラスが提供する実装の観点は**特化**です。

スーパークラス(スーパーインターフェイス)において定義あるいは実装された機能に対し、さらに特定の処理や実装を与えることは、そのサブクラスが提供する特定目的のための実装と言えます。

3.3.7多重継承 (Multiple Inheritance)

C++言語では多重継承をサポートしていますが、Java 言語では多重継承が禁止されています。

ただし Java 言語で禁止されている多重継承は Class 継承の場合であり、Interface

は多重に継承することが許可されています。

> どうして Class の多重継承が禁止され、Interface の多重継承が許可されているのでしょうか？
> 多重継承の弊害は？

3.3.8再定義 (Override)

スーパークラスで定義されているメソッド(あるいはフィールド)を、サブクラスで定義し直すことを再定義(オーバーライド)と呼びます。

オーバーライドするには、メソッド定義が、スーパークラスでのメソッド定義とまったく同一である必要があります。

(→*メソッドシグネチャ*)

C++言語の場合は、オーバーライドされたメソッドが仮想関数でない限り、動的束縛ではありません。

Java 言語の場合は、すべてのメソッドが動的束縛です。

オーバーライドされたスーパークラスのメソッドを呼び出したい場合は、super 修飾子を使用します。

```
public void paint(Graphics g) {
   super.paint(g);              //   スーパークラスの paint メソッドを呼び出す
}
```

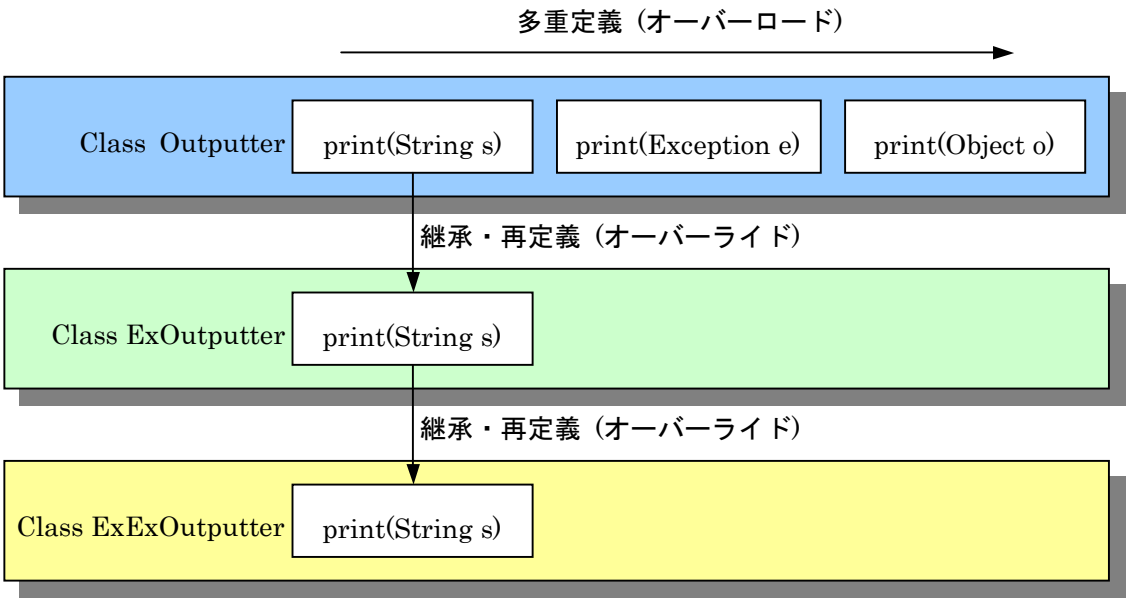
3.3.9多重定義 (Overload)

クラスの中に、メソッドシグネチャの異なるメソッドを複数定義することを多重定義(オーバーロード)と呼びます。

オーバーロードのもっとも典型的な例は、System.out.println メソッドです。

3.4 多態性／多相性 (Polymorphism)

プログラム上において、同じ呼び出し方であるのに、異なる動きをさせる仕組みを多態性と呼びます。



```
public class Outputter {
   public void print(String s) { System.out.println(s); }
   public void print(Exception e) { print(e.getMessage()); }
   public void print(Object o) { print(o.toString()); }
}

public class ExOutputter extends Outputter {
   public void print(String s) { System.out.println("[" + s + "]"); }
}

public class ExExOutputter extends ExOutputter {
   public void print(String s) { super.print("{" + s + "}"); }
}
```

```
public class Processor {
   public static void process(Outputter out, Object param) {
      out.print(param);
   }
}

public class Main {
   public static void main(String[] args) {
      Processor.process(new Outputter(), "hogehoge");
      Processor.process(new ExOutputter(), new Integer(123));
      Processor.process(new ExExOutputter(), new IllegalArgumentException("hogehoge"));
      System.exit(0);
   }
}
```

3.4.1多重定義による多態性 (Adhoc Polymorphism)

多重定義(Overloading)による多態性では、言語が型を強制し、型に応じたメソッドを多重定義することによって実現します。

3.4.2継承による多態性 (Subtype Polymorphism)

継承によってメソッドが再定義された場合、実際に呼び出されるメソッドはスーパークラスのメソッドではなく、サブクラスのメソッドとなります。

つまり同じ型のクラスであっても、そのサブクラスによる実装が異なれば、呼び出された結果も異なります。

3.4.3総称による多態性 (Parametric Polymorphism)

総称(Universal)多態性は、引数から型を特定する機能を用いて実現します。

Ada の generic units や C++の Template が該当します。

Java では、Java Generics として同一変換(homogeneous translation)および閉じた多態(bounded polymorphism)を用いた型総称性が導入されます。

Java Generics	C++ Template
同一変換 (homogeneous translation)	異種変換 (heterogeneous translation)
閉じた多態 (bounded polymouphism)	閉じていない多態 (unbounded polymouphism)

3.4.4動的束縛 (Dynamic Binding)

ソースコード上でのメソッド呼び出し先が、処理コンテキストに依存しない仕組みを静的束縛(Static Binding)、処理コンテキストに依存して変わる仕組みを動的束縛(Dynamic Binding)と呼びます。

C++言語の継承による多態性では、仮想関数宣言されたメソッドのみが動的束縛となりますが、Java 言語ではすべてのメソッドが動的束縛です。

3.5 委譲 (Delegation)

委譲は、サブルーチン呼び出しと等価です。

あるクラスのメソッドを拡張または変更したい場合、もっともわかりやすいのは継承してサブクラス化することです。

しかし実装の継承はデメリットが大きく、機能の独立性を損ない、メンテナンスが難しくなります。

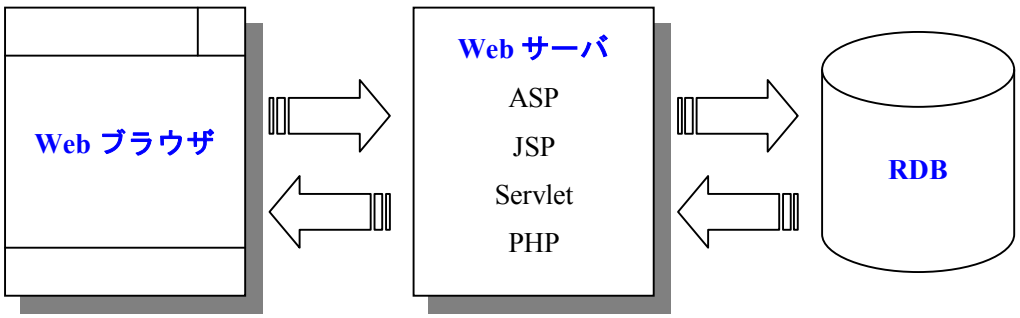
そこで、拡張される処理部を他のクラスに分離しておく手法が委譲です。

委譲とは、構造化プログラミングでのサブルーチン呼び出しに相当します。

4 MVC (Model / View / Controller)

オブジェクト指向プログラミングに限らず、アプリケーションシステムを設計する上で基本となる考え方が **MVC** です。
Web アプリケーション開発で、**三階層モデル**(＝三層構造:データベース層、ビジネスロジック層、プレゼンテーション層)と呼ばれているものとほぼ同じ概念です。
この MVC モデルは機能ごとの実装の独立性が高く、汎用的で、再利用性が高く、拡張が容易である特徴があります。
オブジェクト指向プログラミングにおいても、MVC はアプリケーション開発の基本原則です。
また Java コンポーネントにも、MVC の考え方が採り入れられています。
(→*Java2 の javax.swing.JTable を参照*)

MVC が実現されているもっとも典型的なシステム例として、オンラインショッピングサイトを示します。



4.1 Model

Model とは、抽象化されたデータソースを表します。

現実にアプリケーションが動作するためには、アプリケーションデータを扱う必要があります。
このアプリケーションデータは、システムによってはファイルであり、あるいはメモリ上のオブジェクトであり、あるいはリレーショナルデータベースであり、あるいはネットワーク上のリモートデータであるかもしれません。

本来アプリケーションが動作するためには、実データが『どこに』『どんな形態で』『どんなデータ表現形式で』存在していようとも、関係ないはずです。
このようなデータの抽象化を行うのが、Model の役目です。

4.2 View

View とは、アプリケーションの UI を表します。

ほとんどのアプリケーションは、何らかの UI を持ちます。
ユーザは UI に対して処理要求を行い、UI から処理結果を取得します。
UI に求められる機能は、たったそれだけです。

アプリケーション UI は、そのアプリケーションが動作するプラットホームに応じて変化します。
画面サイズの大小(例えば PC や PDA や携帯電話機)や、入力方式の相違(例えばキーボード入力、ペン入力、音声入力)、出力方式の相違(ディスプレイ、印刷、音声出力)など。
しかしアプリケーションプログラムの機能そのものは、アプリケーション UI の違いには関係ないはずです。
そのため、アプリケーション View はアプリケーションの機能とは完全に独立して実装されるべきです。

4.3 Controller

Controller とは、アプリケーションの機能処理を表します。

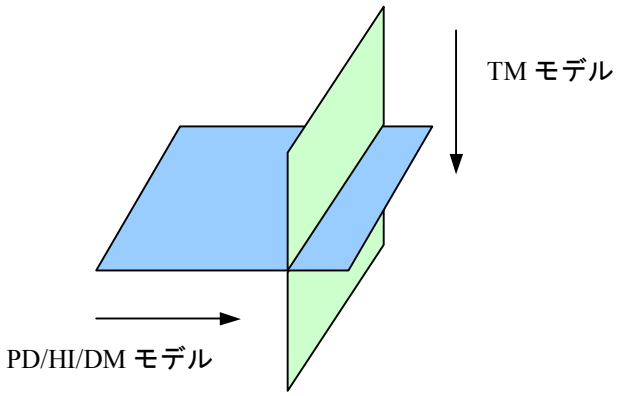
アプリケーションは Controller が中心となって、View と Model を制御し、適切に動作します。

4.4 もう一つの分類方法

MVC モデルは優れたシステムモデリング手法ですが、実際の実装に結びつく概念が含まれていません。
そこで、もう一つのアプリケーションモデルを示します。

- Human Interface (HI)
- Problem Domain (PD)
- Task Management (TM)
- Data Management (DM)

HI とは UI であり、利用者に対する入出力処理に相当します。
PD はビジネスロジックであり、アプリケーションが解決すべき問題そのものを表します。
TM とは、アプリケーションが問題を解決するための動作を管理します。
DM とは、アプリケーションが扱う永続データを管理します。



このモデルは、DM／HI／PD が MVC に相当し、システム実現手段という観点の TM が実装されると考えられます。

MVC と TM をそれぞれモデリングすると、通常これらは概念的に直交するでしょう。

Aspect-Oriented Programming とは？