8 オブジェクト指向プログラミング (

8–1 オブジェクト指向言語の特徴とその恩恵

オブジェクト指向言語の特徴：

- カプセル化と情報隠蔽 · · · 関連したデータと操作を1つにまとめて部品化し、外に見せる必要のないものは隠蔽できる。
  (C++言語では 、クラス定義できるということ。⟶ 第3.2節)
- 新規データ型についての拡張 · · · 新規にデータ型を追加して基本データ型と同じ様に簡単に使える。 (C++言語では 、
  ◇ 定義したクラスの名前をデータ型名として使うことができ、
  ◇ 適切な「演算子多重定義」により、それらの型に関する演算も基本データ型の場合と同じ様に簡単に使える様になる、
  ということ。⟶ 例1.4, 例題4.4)
- 継承 · · · 既存のデータ型を利用して新しいデータ型を定義できる。
  (C++言語では 、⟶ 第5.1節)
- 関数の動的結合による多態性 · · · 多態変数でオブジェクトを保持した時、動的にオブジェクト自身のメンバー関数を呼び出せる。
  (C++言語では 、⟶ 第5.2~5.4節)

これらの特徴はどれもC++言語以前の言語にも散見される。ただ、
C++以前はオブジェクト指向の考え方を取り込む動きは少なかった。

しかし、
1980年台にC++が出現すると、次の理由で
大々的にC++が受け入れられ、
オブジェクト指向の考え方が浸透していった。

- ◇ C言語からC++言語への移行は容易、
- ◇ C++言語は処理効率の面でも実用的、
- ◇ カプセル化、情報隠蔽、継承、多態性、型安全、等の恩恵

C++言語の普及により、
その後のJava言語を始めとしたオブジェクト指向言語の普及にも繋がった。

抽象データ型、カプセル化、情報隠蔽：

関連したデータと操作を1つにまとめたものを、
ソフトウェア部品として扱いたい。
$\longrightarrow$ 抽象的な型(抽象データ型という)をもったデータと考える。

どういうソフトウェア部品が良いか?:

一人で作って利用するとは限らない。
$\longrightarrow$ ソフトウェア部品を提供する側の視点と
利用する側の視点を区別することが大切

ソフトウェア部品利用者の視点

- 利用し易いのが良い。
- 理解し易いのが良い。
- 安価で、効率的で、強力なのが良い。
- 内部の、利用上知る必要のない箇所が隠蔽されていれば 、
  - ◇ その分、理解し易くなる。

ソフトウェア部品提供者の視点

- 再利用や拡張を容易に行えるのが良い。
- 利用者が誤用しにくいのが良い。
- 内部を隠蔽できれば 、
  - ◇ その部分を利用者に説明する必要がなくなる。
  - ◇ その部分は、(利用者にコード修正をお願いすることなく)後で自由に改良を加えることができる。
  - ◇ 利用者の裏技的な誤使用の可能性を少なくすることができる。

⇨ ソフトウェア部品利用者には、最低限必要なデータや操作だけを利用可能にし、残りは隠蔽するのが良い。

ソフトウェア部品の再利用、継承：

プログラミング言語が好んで使われるためには、
ライブラリの生成やソフトウェア部品の再利用に関する性能も重要。
これに関しては、

- 既存のクラスを基に新規にクラスを派生定義する継承を行えば、
  既存クラスのコードが派生クラスと共有され、再利用される。

- 多くの抽象データ型をうまく整理して階層的に派生定義出来れば、
  コードの共有も進み、またインタフェースの統一にも役立つ。

- 継承によって形成されるクラス階層において、
  派生クラスのオブジェクトを基底クラスのオブジェクトとして見ると、
  派生クラスの定義の際に追加した要素は隠れることになる。

  ⇨ クラス階層に沿った情報隠蔽も自然発生していると見做せる。

多態性： 一般のプログラミング言語において 、
変数や関数,定数,オブジェクト,式などの要素(に関する型の扱い)は

- 単態性を持つ ⇔ 単一の型のものしか対応させられない。
- 多態性(多相性)を持つ ⇔ 複数の型のものへの対応付けが許される。
  - 多態性を実現するに当たって 、
    - 普遍的多態性(純多態性)
      ⇔ 1つのコードで複数の型への対応付けが為される時
    - 場当たり的多態性
      ⇔ 見かけ上複数の型への対応付けが為されているが
      最終的な振舞いに至るまでの処置が型ごとに異なる時

Cardelli&Wegner(ACM Computing Survey, 1985)に従うと、
多態性は次の4つに分類される。

- 強制型変換(**coercion,** 自動型変換) ··· 場当たり的多態性の一種。
  コンパイラが必要に応じて型変換のコードを補うことによって、
  関数や演算子が複数のデータ型に対処できる様にする。

- 多重定義 (overloading)··· 場当たり的多態性の一種。
  型ごとに個別に有限個の単態性要素をプログラマが用意し、
  コンパイラがこれらを切り替えて複数の型への対応付けを行う。
  ⟶ **5.3節, 2.11節**

- サブタイプ多態性 (subtyping,inclusion)··· 普遍的多態性の一種。
  多態変数でオブジェクトを保持した時、
  動的にオブジェクト自身のメンバー関数を呼び出せる、
  ということ。 C++言語では 、
  ◇ 基底クラス型オブジェクトへのポインタ変数**p**は多態変数で、
  ◇ 基底クラス上で仮想関数**f()** が定義されていた場合、
  「**p->f()**」と書いた部分は、コンパイラにより
  実行時に**p**の指すオブジェクト内の**f()** を動的に呼び出す
  様に翻訳される。
  ⟶ **5.2∼5.4節**

- パラメータ多態性 (parametric polymor...)··· 普遍的多態性の一種。

パラメータ付きクラス定義、総称的プログラミング。

例えば、

◇ テンプレートクラスが定義され、

◇ そのテンプレートを具現化したクラスを書いた時、

コンパイラは具現化の指定に従って元々のテンプレートからパラメータなしのクラス定義を実際に創り出し、利用する。

⟶ 第6節

オブジェクト指向の恩恵：

- クラスを定義し、そのクラスのインスタンス(ソフトウエア部品)を必要なだけ生成して利用する。
  ⇨ (恩恵0) コードの簡素化 ··· 類似コードをあちこちに書かなくて済むので。

以下、オブジェクト指向の3大要素

- カプセル化 ··· 情報隠蔽を進めてオブジェクト(ソフトウェア部品)の独立性を高める。
  ⇨ (恩恵1.1) モジュール性 ··· 他のオブジェクトと切り離して、オブジェクト毎にソースコードを作成・保守することができる。
  (恩恵1.2) 情報隠蔽の恩恵 ··· 内部の実装方式がちゃんと隠蔽されていて隠蔽しているはずの事柄に依存したコードが他のオブジェクト中に現れないことが保証されるなら、オブジェクト内部の実装方式を自由に変更することができる。

情報隠蔽を行なっている場合は、
情報の保持方法の変更は外部に公開されている関数の処理内容を変更するだけで済む。

(恩恵1.3) コードの再利用 ··· 一般的な処理を行うオブジェクトの場合、他からの独立性を高めることにより、コード再利用の可能性が高まる。

(恩恵1.4) ソフトウェア全体の保守の容易さ ··· 1つのオブジェクトで異常が発生した場合でも、そのオブジェクトを代替オブジェクトに差し替えたり、場合によってはそのオブジェクトを全体から切り離して残りの部分を運用したり (**fail soft**)、ということを行い易い。

- 継承 ··· 既存のクラスの内容を引き継いで新たな別のクラスを定義できる。

  ⇨ (恩恵2.1) 効率的なプログラミング ··· 簡単に既存クラスを拡張できる。また、類似クラスができそうな場合は、

それらの共通部分を親クラスとして構成することにより、類似コードをあちこちに書かなくて済む。

(恩恵2.2) 間違いの可能性の減少 ··· 各種機能を整理して配置し、類似コードが多数に場所に分散するのを極力避けることが出来るので、コードの修正忘れも少なくなる。

- 多態性 ··· 同じメソッドに対してオブジェクトごとに異なる振る舞い。
  ⇨ (恩恵3.1) コードの簡素化 ··· 同種のインスタンスを統合的に扱えるので。

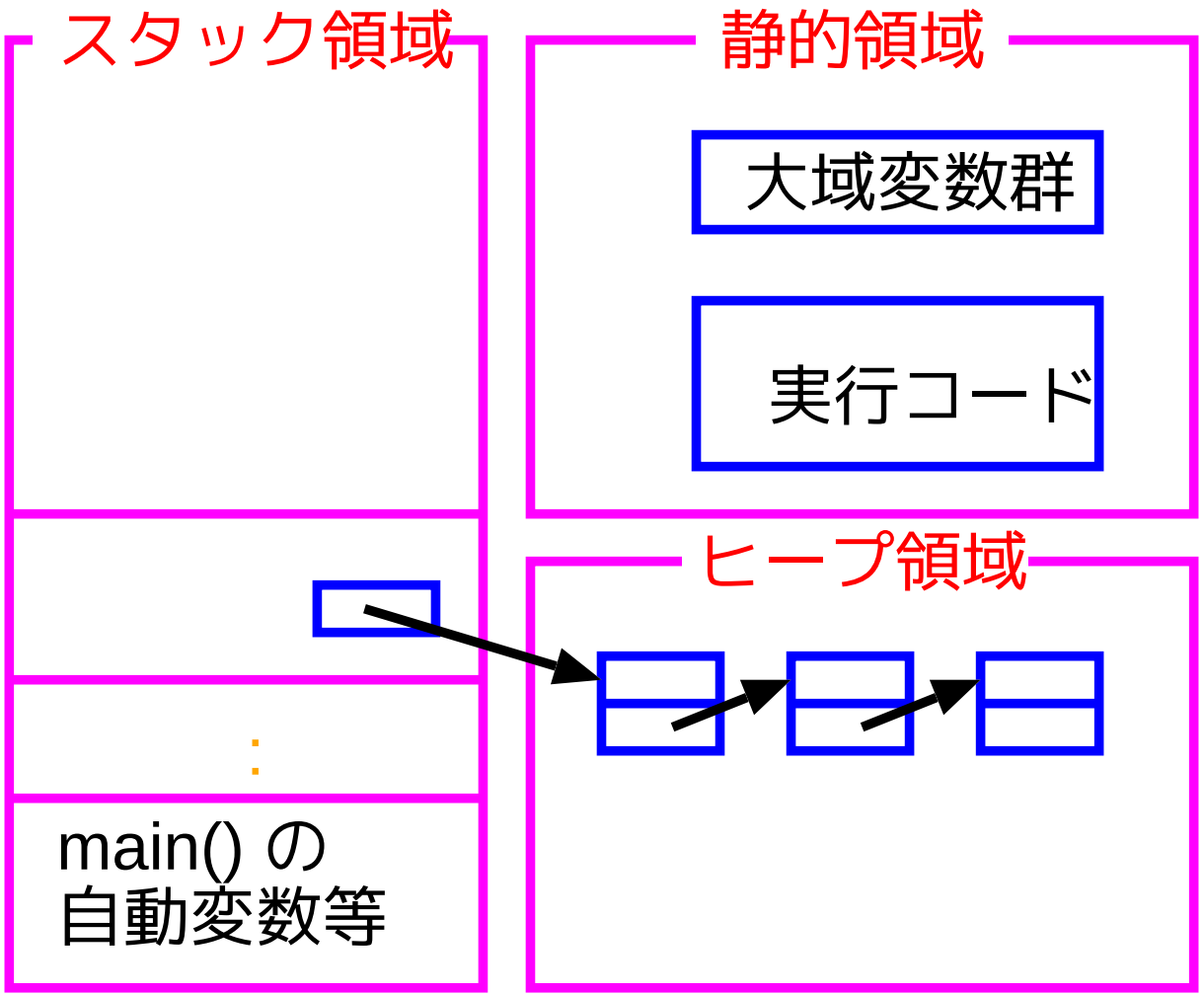
  (恩恵3.2) クラス利用の簡単化 ··· 強制型変換は暗黙に行われる。サブタイプ多態性では、クラス利用者は基底クラスの仮想関数で定められたインタフェースだけを理解していれば十分。また、多重定義により同じ目的の関数に別々の名前を付ける必要がなくなった。

注意：　以上の恩恵を充分に引き出せてないプログラムは
C++で書いてあっても非オブジェクト指向と言える。

8–2 オブジェクト指向言語におけるメモリ領域の

一般に、プログラム実行時にはメモリ領域は 静的領域、スタック領域、ヒープ領域 の3つに分けて管理される。

- 静的領域 ··· 実行コード、大域変数等を格納。プログラム実行から終了まで内部配置は固定。
- スタック領域 ··· 関数呼び出しの柔軟な実現のために利用。
- ヒープ領域 ··· 動的なデータ記憶領域を確保したい時に利用。



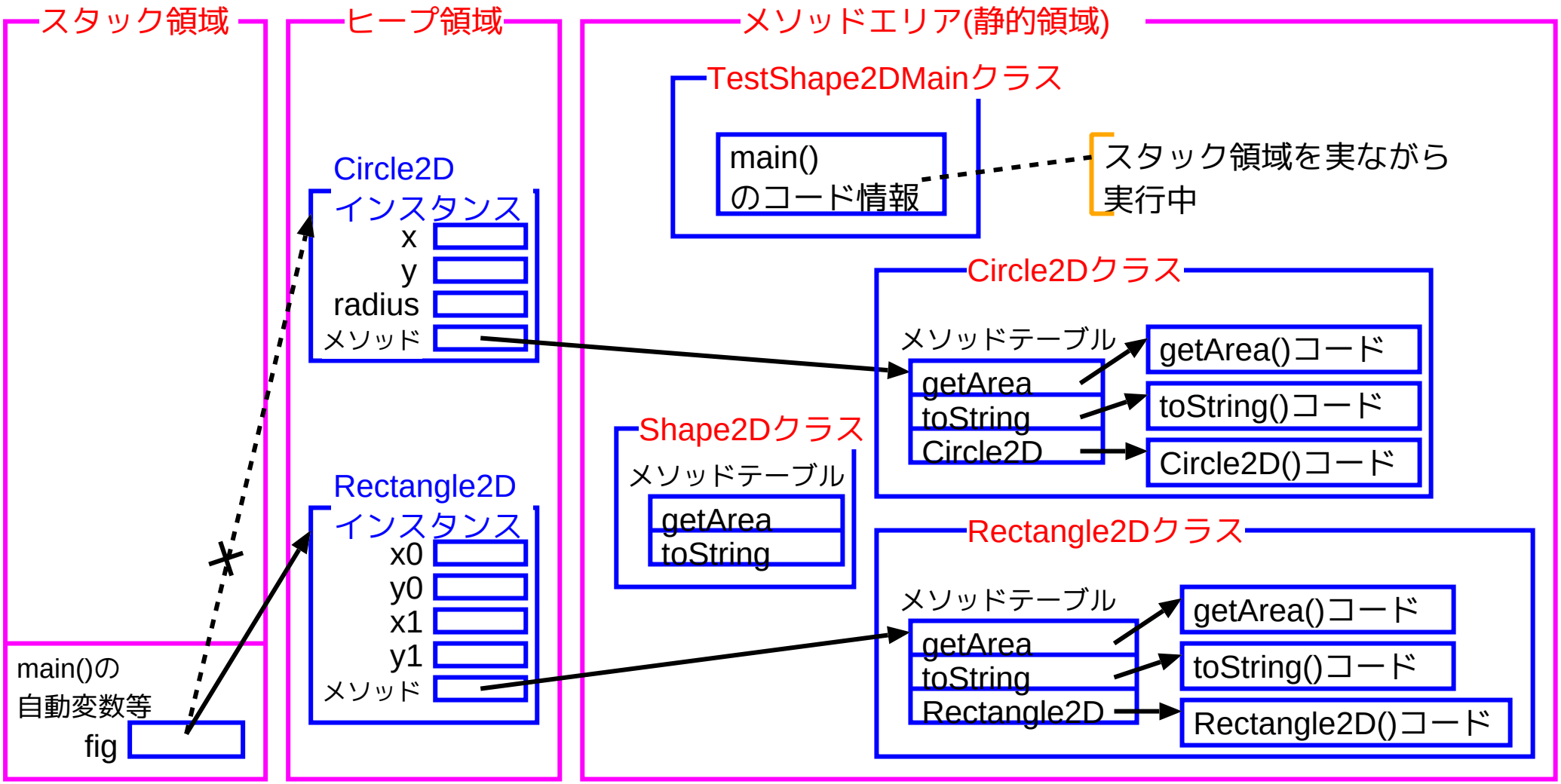
オブジェクト指向プログラムを実行する際のメモリの使い方、

基本的な枠組 $\Longrightarrow$ 上記の通り。
細部 $\Longrightarrow$ 従来と異なる点もある。例えば、

- 例えば **Java言語(1991頃〜)では** クラス情報は必要になった時点でロードされる方式。
  $\Longrightarrow$クラス情報を格納する領域はもはや静的ではない。
  $\Longrightarrow$メソッドエリアと呼ばれている。

- **Javaを始めとする最近のオブジェクト指向言語の場合は** 、
  インスタンスオブジェクトは全て動的に生成される。
  $\Longrightarrow$ヒープ領域は頻繁に利用される。

  例えば、例題5.7で与えたC++プログラムに相当するものをJava言語で書いたとして、その実行途中には、メモリ内部の状態は次の様になっているものと考えられる。


スタック領域
ヒープ領域
メソッドエリア(静的領域)
TestShape2DMainクラス
main()
のコード情報
スタック領域を実ながら
実行中
Circle2D
インスタンス
x
y
radius
メソッド
Circle2Dクラス
メソッドテーブル
getArea
toString
Circle2D
getArea()コード
toString()コード
Circle2D()コード
Shape2Dクラス
メソッドテーブル
getArea
toString
Rectangle2D
インスタンス
x0
y0
x1
y1
メソッド
Rectangle2Dクラス
メソッドテーブル
getArea
toString
Rectangle2D
getArea()コード
toString()コード
Rectangle2D()コード
main()の
自動変数等
fig

特にC++言語の場合、

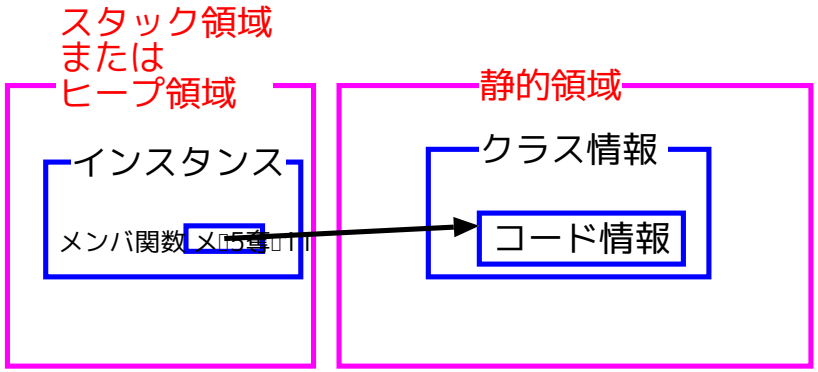
- 全てのクラス情報はプログラム開始時に静的領域に一括してロード。

- C++言語(1979年頃～)を設計するに当たって、
  C言語のシステムプログラミングに適した性格は重要視された。
  ⇨ インスタンスをスタック領域に配置する選択肢も残された。

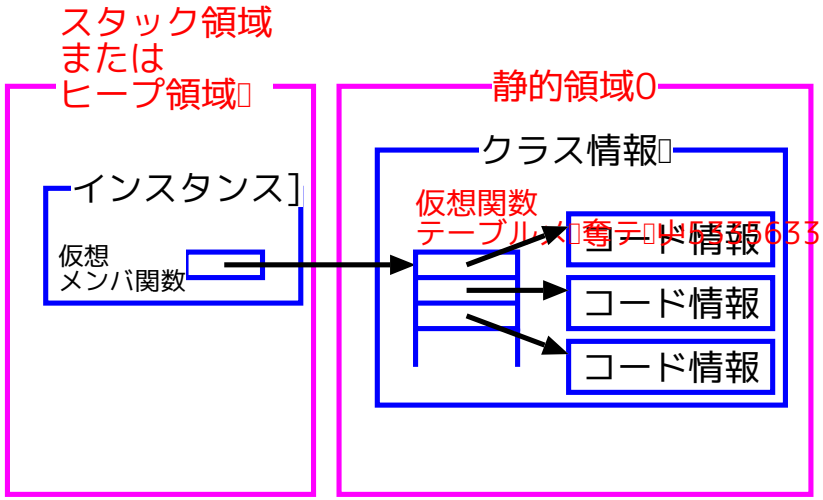
- インスタンスをヒープ領域に配置するためには 、
  ①空き領域演算子newを用いてインスタンスのための領域を
  　ヒープ領域内に確保し、そこにインスタンスを構成する。
  ②確保された領域へのポインタ値をスタック領域内の変数に保持。

  (このポインタを保持する変数を宣言したブロックを出る際は、当然、
  　①(必要ならデストラクタを呼び出し、その後で)
  　②delete演算子を呼び出して確保していた領域を開放する。)

- メンバ関数のコード情報は、
  クラス情報の一部として静的領域内に配置され、
  そこへのポインタが個々のインスタンス領域の中に置かれる。



  また、多態性の実装を容易に行うために、
  クラス情報の領域内には、仮想関数テーブルが配置され、個々のインスタンスの中にはその仮想関数テーブルへの参照情報だけが置かれることもある。



8–3 オブジェクト指向の設計

オブジェクト指向ソフトウェア設計の流れ： {Pohl(1999)10.1.2節}

クラス継承／階層はソフトウェア全体の設計に影響を及ぼす。

⇨ オブジェクト指向の下ではソフトウェア設計は次の流れで進む。

① 必要な抽象データ型群 (i.e. クラス群) を特定する。

② 前ステップで特定したクラスの間の関係を精査し、
クラス継承によりコードとインタフェースの共通化を進める。

③ 仮想関数を用いて、
必要に応じてオブジェクトに多態的な動きをさせる様にする。

ソフトウェア設計／クラス設計の際に考慮すべきこと：

{Pohl(1999)10.3節}

相反することもある次の事柄を考慮し、バランスの良い設計にする。

- 完全性 · · · クラス設計の場合は、
  用意されたデータメンバとメンバ関数を使って
  行いたいことを全て行えるか、
  ということ。ブール代数においては、
  ◇ 演算の集合 **{nand}** は完全であるが、
  ◇ 通常、基本的な演算集合として **{not, and, or}** を用いている。
  この例の示す様に、
  ◇ 完全性だけでは不十分なことが多く、
  ◇ 実用のために十分な表現力が必要である。

- 表現力 · · · クラス設計の場合は、
  十分なデータメンバとメンバ関数が用意されているか、
  ということ。

- 可逆性 · · · クラス設計の場合、
  用意されたどの主要メンバ関数に対しても
  逆の働きをするメンバ関数が用意されている、
  ということ。
  ◇ 数値データ型の場合 $\longrightarrow$ 加算と減算は互いに逆演算の関係にある。
  ◇ テキスト編集の場合 $\longrightarrow$ **undo** が全ての操作の逆操作に相当する。

- 直交性 · · · クラス設計の場合は、
  生成されたインスタンスを利用する際に
  利用目的のために必要な操作系列が一意に決まる
  (**i.e.** 冗長性がない)、
  ということ。例えば
  ◇ 平面上で図形を操作する場合、
  互いに直交した操作集合として {水平移動, 垂直移動, 回転}

- オッカムの剃刀 (**Occam's Razor**) $\cdots$ 元々は

  「ある事柄を説明するためには、

  必要以上に多くを仮定するべきでない」という指針

  で、より一般には

  「不要なものを (剃刀で) 切り落とす」

  ということ。

  クラス設計の際に継承によりコード共有を進める、

  というのはこの指針に沿ったものと言えなくもない。

- 無矛盾性 (**consistency**) $\cdots$

- 単純性 (**simplicity**) $\cdots$

- 効率性 (**efficiency**) $\cdots$

オブジェクト指向設計の原則:

{日経ソフトウェア編(2009)「ゼロから...」第3部1章p.181}

- 単一責務の原則(SRP, Single Responsibility Principle) ···
  分割された個々のプログラムに複数の目的・機能を負わせるべきでない、という指針。(その方が、将来の部分的な変更もやり易い。)

- 開放閉鎖の原則(OCP, Open-Closed Principle) ···
  構築するクラス群は、
  機能拡張可能(open)で、
  拡張の際には既存コードには手を加えなくて済む(closed)、
  様なものが良い、という指針。

良いプログラムを構築するための考え方と実践方法：

{日経ソフトウェア編(2011)「Javaツール完全理解」第2部2章}

- ソフトウェアの価値の3条件
  - ◇ シンプル ··· プログラムの理解や修正が容易になる。
  - ◇ コミュニケーション可 ··· プログラムの書き手と読み手がソースコードを通じて十分にコミュニケーションできる。
  - ◇ 柔軟性 ··· 変更に対する柔軟性がある。(初期開発費用より修正費用の方が大きいので、これも重要。)

- プログラミングの原則
  - ◇ **YAGNI (You Aren't Going to Need It.)** ···
    今必要なことだけをやる。
  - ◇ **DRY (Don't Repeat Yourself.)** ···
    コードの重複を避け、必要があればできるだけ再利用する。
  - ◇ **PIE (Program Intently and Expressively.)** ···
    意図が明確に伝わる様にコードを書く。

- 代表的なベストプラクティス
  - ◇ リファクタリング ··· 外部に対する振舞いを変えずに、ソースコードの内部構造を整理し簡素化すること。
  - ◇ テストファースト ··· 実装者の視点で実装コードを書く前に、利用者の視点でテストコードを書く。これによって余分な複雑さを排除できることを期待する。
  - ◇ ドメイン駆動設計 (**Domain-Driven Design, DDD**) ··· 問題領域(**domain**)をモデル化し、それを中心に据えてソフトウェアを設計する。(モデル自体もドメイン知識を使って反復的に深化させていく。)

8–4 ソフトウェアの部品化と再利用

プログラミング言語の進化 プログラミング言語に備わっているべき事柄は、...

- アルゴリズムを容易にコード化できるための表現能力
- ソフトウェアの寿命が伸びた
  $\Longrightarrow$ 保守も大事
  $\Longrightarrow$ 出来上がったプログラムの理解や修正の容易さも重要
- ソフトウェアには高い品質が必要
  $\Longrightarrow$ プログラムの中に余計な複雑さや単純な間違いが入り込みにくいということも大切
- ソフトウェアの生産性を上げたい
  $\Longrightarrow$ 実績のあるプログラムを再利用する仕組み

⇨ プログラミング言語がどの様に進化してきたのかを、次の様にまとめることができる。

	表現能力	保守性	品質保証	再利用性	導入された機構/考え方
機械語					
アセンブリ言語	△				
高級言語	○			△	• サブルーチン $\Longrightarrow$ 表現能力,再利用性向上
構造化	○	△	△	△	• 3つの基本構造 $\Longrightarrow$ 保守性向上 • サブルーチンの独立性を高める $\Longrightarrow$ 再利用性多少向上
オブジェクト指向	○	○	○	○	• クラス (関連する変数とメソッドをまとめる仕掛け) $\Longrightarrow$ 保守性向上,再利用性向上 • 例外機構 $\Longrightarrow$ 品質向上 • 型チェックの強化 $\Longrightarrow$ 品質向上 • 多態性 $\Longrightarrow$ 再利用性向上 • 継承 $\Longrightarrow$ 再利用性促進

↓ 時間

ソフトウェア再利用技術発展の流れ

オブジェクト指向より前の構造化言語では、

$\Longrightarrow$ 再利用できるソフトウェアと言えばサブルーチンだけ

コード変換, 入出力処理, 数値計算, ... の汎用ライブラリ程度

オブジェクト指向の考えが導入されると、

(関連性の強いサブルーチンや大域変数を1つのクラスとしてまとめて粒度の大きいソフトウェア部品を作り出す仕組みがある)

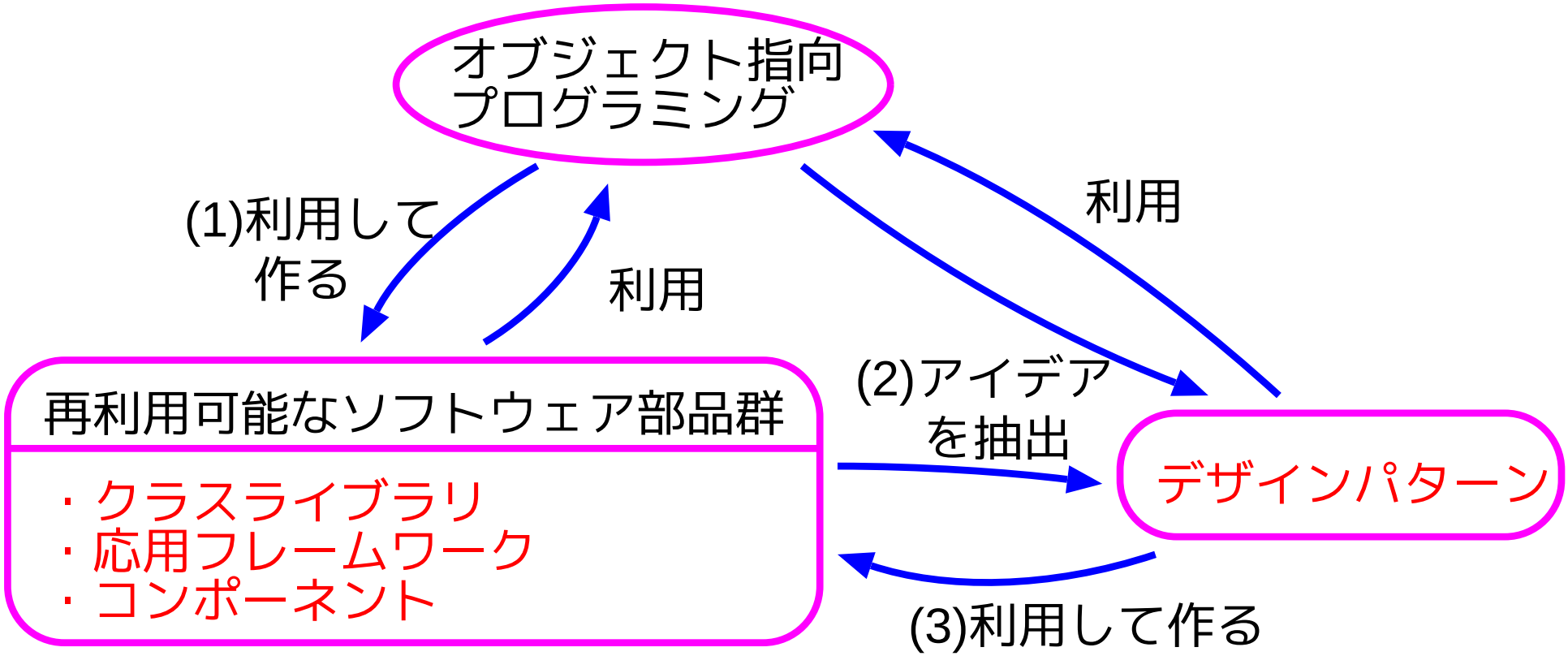
$\Longrightarrow$ ソフトウェア再利用の可能性は大きく広がる。

$\Longrightarrow$ ソフトウェア部品として既に存在しているソースコードや実行形式モジュールを使い回すのが当り前。

再利用可能なソフトウェア部品としては、現在、

- クラスライブラリ、
- (応用) フレームワーク、
- コンポーネント、
- デザインパターン · · · ソフトウェア設計のアイデアを後で利用できるように文書化したもの

と呼ばれるものがある。



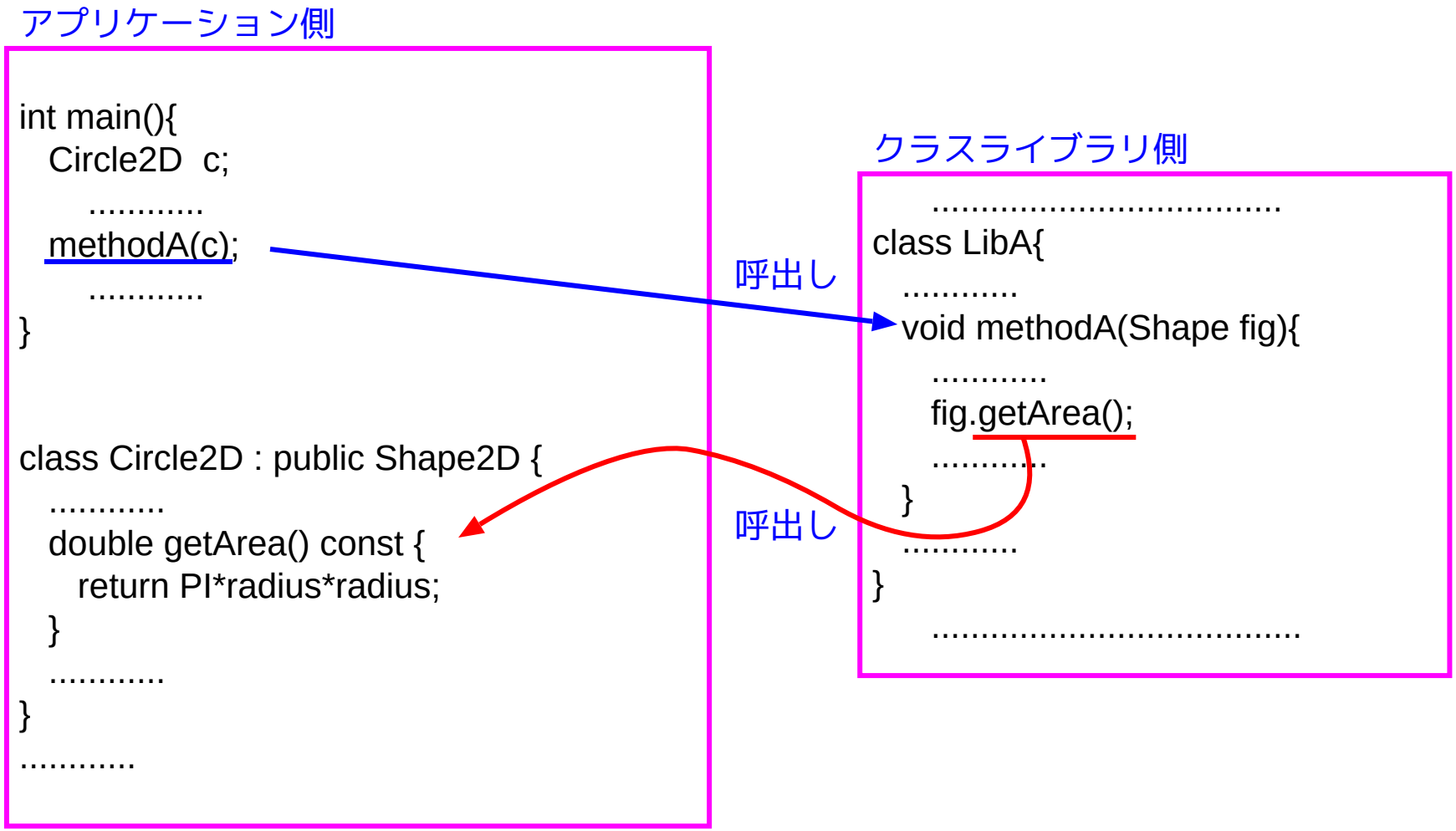
クラスライブラリ　　・・・汎用的な機能をもつクラスを多数蓄積したもの

オブジェクト指向の前と比べて

- ソフトウェアが格段に豊富になった。
- ライブラリの利用の仕方も広がった。

クラスライブラリの場合は次の様な**3**つの利用の仕方が可能

- 用意されたクラスのインスタンスを作成して
  付属のメンバ関数等を利用。
  (クラスの利用, 従来のライブラリ関数呼び出しに相当。)
- ライブラリのコードからアプリケーション固有の処理を呼び出す。
  (多態性の利用。)



- ライブラリ内のクラスを拡張・補正して、新しいクラスを作成。(継承の利用。)

例えば、**2011年時点のJava(J2SE7.0; JDK1.7, Java SE Development Kit 1.7)**には、**GUI**, 入出力, ネットワーキング, ... 等のために、合わせて**4000** にものぼるクラスライブラリが整備されていた。

言語仕様は最小限に抑えて
必要な機能はクラスライブラリとして提供される
$\Longrightarrow$ 言語仕様の互換性を保ちながら
クラスライブラリの拡張によってバージョンアップを行うことが可能。

⇨ この豊富なライブラリを使いこなすことが、
オブジェクト指向言語習熟への1つの道である。

(応用) フレームワーク

用途 (ドメイン) が類似したソフトウェアは多くの共通要素を持つ

⇨ ◇基本的な制御の流れをもったソフトウェアの枠組みを予め用意し、
◇個別の処理のために将来組み込むクラスやライブラリを明らかにしておけば、

利用者の個別の要求のある部分だけを追加する

だけで色々な利用者のニーズに合った応用プログラムを作成できる。

こういった考えで作られた、「半完成品」の応用プログラムを応用フレームワークあるいは単にフレームワークという。

クラスライブラリも応用フレームワークも再利用可能なソフトウェア部品群という点では同じ。ただ、目的と再利用部品の使われ方が違う。

コンポーネント ···(クラスライブラリ, 応用フレームワークと全然違う)

平澤(2004)によれば、

- クラスよりも粒度が大きく、
- (ソースコード形式でなく)バイナリ形式で提供される、

そして、

- ソフトウェア部品の定義情報も提供される、
- 機能的に独立性が高く内部の詳細を知らなくても利用できる、

というものを一般にコンポーネントと呼ぶ。(広く浸透していない。)

利用の仕方も特徴的で、

ソースコードを書くのではなく、
視覚的なツールを用いて直接関連する部品を配置・設定・接続する

ことによって、短時間で応用ソフトウェアを組み立てる。

具体例:

マイクロソフト社VisualBasic(1990年代前半~)で導入 $\longrightarrow$ ActiveX

Java環境ではJavaBeansと呼ばれる仕組み

Beansと呼ばれるコンポーネントを

BDK 等の(ビルダ)ツールで接続してソフトウェアを組み立てる。

デザインパターン

..... オブジェクト指向に基づいて
再利用性や柔軟性の高いソフトウェアを開発しようとする際に、
様々な場面で適用される「お決まりの設計指針」

有意義で適用範囲の広いデザインパターンが見つかれば、
①全てのソフトウェア開発者がその恩恵を受けることができ、また
②それが開発者間の共通認識として定着すれば開発者間のコミュニケーションも容易になる。

この様な状況はアルゴリズムやデータ構造の場合と同じ。

具体的なデザインパターンとしては、
E.Gamma, R.Helm, R.Johnson, J.Vlissides という**4**人の技術者達 (**GoF**, **the Gang of Four**) が発表した次の**23**種類 (**GoF**のデザインパターンと呼ばれている) が有名で、これらは**Java**言語のクラスライブラリの作成にも大いに利用されている。

GoF による分類	パターン名	再利用を妨げる要因のどれに効果が期待されるか?						
		クラス名を固定したインスタンスの生成	特定の処理内容への依存	プラットフォームに依存した application program interface の使用	特定のアルゴリズムへの依存	クラス同士の密接な依存関係	継承によるデメリット	クラス数の急激な増加
生成に関するパターン	Abstract Factory	○		○		○		
	Builder				○			
	Factory Method	○						
	Prototype	○						
	Singleton							
構造に関するパターン	Adapter							
	Bridge			○		○	○	○
	Composite						○	
	Decorator						○	○
	Facade					○		
	Flyweight							
	Proxy							
振舞いに関するパターン	Chain of Responsibility		○			○	○	
	Command		○			○		
	Interpreter							
	Iterator				○			
	Mediator					○		
	Memento							
	Observer					○	○	
	State							
	Strategy				○		○	
	Template Method				○			
	Visitor				○			

GoFの著作においては、

これらのデザインパターンは次の様な項目に分けて説明されている。

- パターン名 ···
- 目的 ··· そのデザインパターンがどの様な設計課題に対処するか、何をもたらすか、原理と意図、等を簡潔に。
- （別名 ···　　）
- 動機 ··· 設計上の問題点、及び、そのパターン内のクラスやオブジェクトの構造がどの様にその問題を解決するか、のシナリオ。
- 適用場面 ··· このデザインパターンを適用できる状況。
- 構造 ··· クラス間の関係、要求のシーケンス、...を図で。
- 構成要素 ··· 使われるクラス、オブジェクトと、各々の役割。
- 協調関係 ··· 各構成要素がどの様に協調して役割を果たすか。
- 結果 ··· そのパターンが要求に対してどの様に効果を発揮するか。
- 実装 ··· 実装方法や注意点。
- サンプルコード ··· そのデザインパターンを使って実装した例。
- 利用例 ··· そのデザインパターンが実際のシステムで利用された例。
- 関連するパターン ··· 別のデザインパターンとの関係。

例8. 1 (Iteratorパターンの活用)　例えば、大きさ10の配列 arr[] の個々の要素に対してメンバ関数getName()の実行依頼を出し、戻って来た文字列を全て表示するのに

```
for (int i=0; i<10; i++)
    cout << arr[i].getName() << endl;
```

これは、
　　オブジェクトの集合体を表すのに配列を用いるということに依存したコード

⇨ データを vector<要素の型> 等のSTLコンテナ v に入れ、
Iteratorパターンの指針に従えば、上のコードは

```
for (vector<要素の型>::iterator p=v.begin();
                                p!=v.end(); p++)
    cout << p->getName() << endl;
```

という風に、集合体の実装方法に依存しないコードに置き換わる。

—

例8. 2 (Singletonパターンの活用)

インスタンス生成を1度だけに限定したいクラスもある。

⇨ **Singletonパターン**

例8. 3 (Strategyパターンの活用)

処理の大枠は固定するが、
その中で使うアルゴリズム(strategy)は色々と切り替えて使いたい、
という場合もある。

⇨ **Strategyパターン**
