オブジェクト指向プログラミング及び演習 1

オブジェクト指向の目的と歴史

オブジェクト指向とは

オブジェクト指向には、主に以下の目的があります。

- 設計手法としてのオブジェクト指向（OOD）
- 分析手法としてのオブジェクト指向（OOA）
- 設計と分析を総称してオブジェクト指向分析設計（OODA）
- ***プログラミングとしてのオブジェクト指向（OOP）***
  - ↑この講義ではここに焦点を当てます。
- 開発方法論としてのオブジェクト指向
- プログラミング言語仕様としてのオブジェクト指向

オブジェクト指向を学ぶ必要性

現代において「専門的にオブジェクト指向について学ぶこと」自体は、実はそれほど重要でありません。ではこの講義で取り扱うものは何なのか。ということになります。

皆さんは一年次にC言語を講義で学びました。
C言語でのプログラミングでも、構造的なプログラミングや、関数ライブラリを使った柔軟な開発は可能です。
極論を言ってしまえば、新たな道具としてオブジェクト指向を学ばなくても、大抵のプログラムは作れてします。

より良く継続的な開発を行うために

C言語でいい、とは言うものの、作成されたプログラムには「保守・運用」のフェーズが存在します。
そこで、言語として以下の要点が求められます。

1. 開発の効率化
2. 保守性の向上
3. 品質の向上

単純に作るだけでなく、これらを意識する必要性があります。

大規模で長期的に維持可能なプログラムを作る

C言語でできないかと言われるとそうでもないのですが「もっと効率良くしたい」という要望のために、オブジェクト指向という考え方が登場します。
より効率良くプログラミングを行うことを目的として、保守運用を見定めた概念であることをよく理解してオブジェクト指向を学んでください。

ただ、現代におけるプログラミングに求められるプログラミングの考え方には、オブジェクト指向以外も必要としています。
オブジェクト指向のその先もある、ということです。

プログラミング言語の歴史

おおまかに以下のような流れがあります。

1. 機械語、アセンブラ
2. ソフトウェアクライシス
3. 構造化プログラミングの登場
4. 構造化プログラミング以降
5. オブジェクト指向言語
6. 現代

機械語

電気信号を0,1で表し、2進数や16進数で表現します。
コンピュータは、0,1で表された命令をそのまま実行できます。

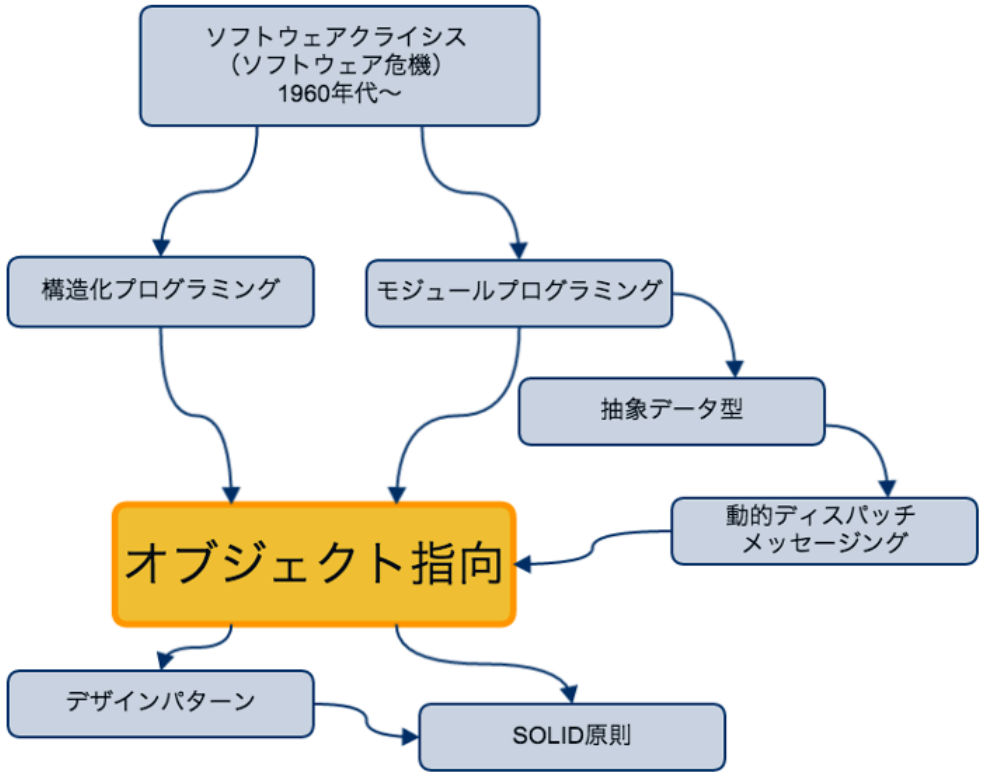
アセンブラ

コンピュータが理解する命令セットに、アルファベットの文字を割り当てることで、機械語を読み書きしやすくしました。
`MOV AX, X` や `ADD AX, DX` のような書き方で、レジスタやメモリを操作します。

ソフトウェアクライシス

1960年代の後半ごろから、コンピュータの爆発的な進化により、さらに複雑で大規模なソフトウェアが求められるようになりました。
この時点では、まだそれを実現できる言語は存在しませんでした。

右図はソフトウェアクライシス以降を表しています。



構造化プログラミングの登場

エドガー・ダイクストラにより提案された技法で、以下のような特徴があります。

- 機械と人間の間に、より人間の言葉に近いものを理解できる仮想機械を想定し、人間はそれに対してプログラミングをする
- 各仮想機械を階層として隔離して実装し、その変更は他の階層へ影響を与えない

インタプリタやコンパイラなどが間に入ることで、よりプログラミングしやすくなりました。

構造化以降（モジュールプログラミング）

モジュールプログラミングは、プログラムをモジュールに分割管理する手法をまとめた手法です。C言語の関数ライブラリ、Javaのクラスなどの基本的な考え方につながっていて、モジュール分割には以下のような考え方があります。

凝集度

- 良い機能のまとめ方と悪い機能のまとめ方を定義

- 良い機能のまとめ方
  - 通信的凝集: とあるデータに触れる処理をまとめること
  - 情報的凝集: 適切な概念とデータ構造とアルゴリズムをまとめること
  - 機能的凝集: ひとつのタスクをこなせる様にまとめること
- 悪い機能のまとめ方
  - 何を根拠に集めた機能かわからない機能群

結合度

- モジュールごとの良い連携と悪い連携を定義

- 良い連携:
  - モジュール間の「関心の分離」が行われている
- 悪い連携例:
  - 内部結合:各機能群が依存している内部データを使ってしまう
  - 共通結合:すべての機能群で1つのデータを使ってしまう

関心の分離

- 「責任」と「責務」、何をしたいのかによって分離させる

より良いモジュール分割を考える

- 良いモジュール分割とは、「データの存在を隠蔽し、関数化できること」（しかし、現実問題としてすべてにおいては不可能）

- なぜそれが難しいか
  - それぞれのモジュールが**状態**と**副作用**を持つから

副作用とは

何かしらの処理を行った（関数を呼び出した）ために、意図しているいないに係わらず他の状態を操作してしまうこと
例: 関数を呼んで処理を実行したら、グローバル変数の値が変わってしまった。など

副作用: 関数について考える

- 入力に対して出力が一定
  → 純粋関数 : 副作用なし
- 隠れた出力がある
  → 入力に対して出力は一定だが、他の機能に影響を及ぼす
- 隠れた入力がある
  → 入力に対して出力が不定
- 隠れた入出力がある
  → 入力に対して出力が不定かつ、他の機能に影響を及ぼす

副作用をどうすれば良いのか

オブジェクト指向に向かっているモジュールプログラミングでは、以下のように考えられています。

- 状態や副作用に対して、名前を付けて可視化してしまおう。
- なるべく副作用による影響を少なくするために、結合を疎にしよう。
  - これらの可視化、疎結合化によって **「関心の分離」** を実現しようとした。

結論: 状態も副作用もすべてコントロールして解決しよう。

抽象データ型

よりよいモジュール化の先にあるものとして型に注目します。
（オブジェクト指向でいうところのクラスなどで定義されるもの）

- データとそれに関連する処理をひとまとめにしたデータ型を定義できるようにする
- 状態、振る舞い、副作用をコントロールする方法を模索する
  - 機能をまとめて管理しやすくする
  - 外部に見せる必要のない部分を隠蔽することで意図しない操作を避ける

オブジェクト指向

構造化プログラミングや、モジュールプログラミングから影響を受け、取りまとめ、独自の解釈を入れた考え方が登場します。

オブジェクト指向は、単純に後継者として現れたのではありません。

オブジェクト指向登場の後も、構造化プログラミングやモジュールプログラミングに対して相互に影響を与え合い、継続的に進化します。

Simula

最初は普通のモジュールプログラミング言語（1960年代後半〜）
元々シミュレーションを記述するために作られた言語で、後に紹介するSmalltalkの影響を受け、オブジェクト、抽象データ型、動的ディスパッチ、継承などの概念を持たせることで、汎用的に使われるオブジェクト指向言語になります。

- C言語にSimulaのオブジェクト指向を取り入れたのが、C++。
  C++をベースに、オブジェクトを通したメモリアクセス、ガーベジ・コレクション、仮想マシンなどの機能を取り入れて作られたのがJavaです。

純粋オブジェクト指向言語 Smalltalk

Simulaをベースに、純粋なオブジェクト指向言語としてできた言語で、Simulaのコンセプトに「メッセージング」の概念を加えて結合されました。主な特徴は以下の通りです。

- すべての処理がメッセージ式で記述される。
- 純粋オブジェクト指向言語と呼ばれる。

ちなみに、SmalltalkのコンセプトでC言語を拡張したのが、「Objective-C」で少し前までOSXアプリやiOSアプリの主要開発言語でした。

そもそもオブジェクトとは

オブジェクト指向で用いられるオブジェクトとは、ひとつのテーマを持ったデータと処理の集まりを指します。

オブジェクトは、以下の要素で構成されます。

- 状態（データ）：プロパティや変数
- 振る舞い（処理）：メソッド（Cでいう関数）

モジュールを「何をどうしたいか」という観点で関心の分離を行ったものとして扱われます。

オブジェクト指向言語のSimulaの登場から、さらに多くの概念が登場します。その中でも生き残ったものとして、動的ディスパッチや、継承・委譲、メッセージングなどがあります。

動的ディスパッチ

どの処理を呼び出すかを決めるメカニズムで、同じ機能名であっても処理対象のデータの種類ごとで行われるべき機能は違うようにする技術。
これにより、**多態性（ポリモーフィズム）**が実現可能となりました。
多態性は、「窓口に渡す書類によって、処理内容を変えたい。」を実現できる考え方です。

継承・委譲

モジュールプログラミングでモジュール分割すると、似たような処理があちこちで点在するようになってしまった。
これを解決するために、モジュール間を親子関係で結合させ、共通の部分に関しては親の機能が使えるようになる「**継承**」が考えられました。
継承は、抽象データ型を持つ言語で使用可能で、抽象データ型を持たない言語では、処理を委譲することで対応させることができます。
これを動的継承とも呼ばれたりする場合があり、委譲を中心に考えられた言語はプロトタイプベースのオブジェクト指向言語と呼ばれたりします。

特殊な関数呼び出し メッセージング

メッセージングかなり特殊で、形式に囚われず、メッセージを受け取った関数は自由に解釈を行い処理します。(覚えなくて良い)

メッセージングの特徴

- カスケード式 : 複数のメッセージをまとめて送る
- メッセージ転送 : 受け取ったメッセージに対する処理が定義されていない場合でも自由に取り扱える
- 非同期送信 : メッセージはそれぞれが独立して処理されるため同期的に待ち受けている必要はない

オブジェクト指向以降

オブジェクト指向の登場以降、プログラミング言語以外にもさまざまな考え方や概念が登場し、またそれらが言語に影響したりと相互に作用します。いくつか紹介しておきます。

- デザインパターン
  - オブジェクト指向を用いて実現可能な設計とコードをまとめてパターン化したもの
  - GoF（Gang of Four）なんかが有名で、プログラミングに慣れてきた頃合いに調べてみると、その考え方や手法はとても参考になるでしょう。

他には、より良いプログラミング、保守運用のためのルールや規約ができました。

- SOLID原則
  - 単一責務の原則（Single Responsibility Principle）
  - 開放/閉鎖の原則（Open-Closed Principle）
  - Liskovの置換原則（Liskov Substitution Principle）
  - インターフェイス分離の原則（Interface Segregation Principle）
  - 依存関係逆転の原則（Dependency Inversion Principle）
    などなど

オブジェクト指向言語も他から影響を受けている

たとえば、関数型言語からは無名関数やラムダ式などが実装されたり、型の柔軟性を高めるためにジェネリクスの概念を導入したり……

単純に、継承を学んでクラス図を書いて、というような過去のオブジェクト指向を学ぶだけでは足りなくなっているのが現在です。

プログラミングを取り巻く環境は変化し続ける

昨今のAIの状況などもそうですが、技術は常に変化します。
この講義の名前は、「オブジェクト指向プログラミング」となっていますが、ぜひプログラミングとそれを取り巻く環境についても一緒に学んでいけたらいいなと思います。

プログラミングテクニックとしてのオブジェクト指向プログラミング

この講義では、さまざまな用語が出てきます。
すでに知っている人もいれば、知らない人もいると思いますので、ざっくりと説明しておきます。

ここからは、「オブジェクト、クラス、インスタンス」、カプセル化、インターフェイス、継承について説明します。

オブジェクトとは

コンピュータのメモリ上に展開された、プログラム内の状態と、振る舞いをまとめたものと言えます。

- 状態（データ）
  - 変数やフィールド
- 振る舞い（処理内容）
  - メソッド（関数）

状態と振る舞いは、関連を持たせて管理をすることが望ましい

クラスとは

オブジェクトはメモリ上に展開されて使用されます。
そのオブジェクトは、どんな状態を持って、どんな振る舞いをするのかを定義するのがクラスです。

よくある表現として、オブジェクトの設計図をクラスという言い方がされます。

インスタンス（実体）とは

オブジェクトは、メモリ上に展開されて使用されます。
その状態をインスタンスと呼び、メモリ上に展開して使用できるようにすることをインスタンス化（実体化）と言います。

クラス、インスタンス、オブジェクトの関係

- クラス: 設計図
- インスタンス: クラスを元に作られたオブジェクト
- オブジェクト: インスタンスを含むさまざまなものをこう呼ぶ

※厳密には、インスタンス化をしなくても内部的に使用できる状態や振る舞いというのも存在します。それらは「**静的（static）な**○○」と呼ばれ、インスタンス化を明示的に行わなくても、プログラム実行時に自動的にメモリへ展開されており使用することができるようになっています。対義語として「動的（dynamic）な○○」という言い方もあり、そちらはインスタンス化しないと使用できません。

クラスはどのように書くか

何らかのデータと、それに対する処理をまとめて書けると良いです。

「クラスは、C言語における構造体（struct）に、関数をつけられるようにしたもの」のようにイメージすると理解しやすいです。

役割ごとに分割する必要性

その方が大規模なプログラムを作る際には管理しやすくなります。
管理しやすいとは、複雑でなく、バグの発見・修正が容易なことを指します。

main関数に処理を突っ込むのではなく

自分の作っているプログラムをよく分析して、どんな登場人物（データ）がいるか、それぞれどんな役割があるかで考え、クラスに分割しましょう。

- イメージとして、子供に対して着替えの指示を出すことを考えましょう。
  - （上着を着ているか確認）「上着のボタンを外して」「右腕から袖を通して次に左腕通して脱いで」（シャツを着ているか確認）「シャツも右腕から……」（靴下を履いているか確認）「屈んで靴下脱いで」（ズボンを履いているか確認）「ズボン脱いで」（タンスの場所を子供が知っているか）「新しい服をタンスから出して」（新しい服はそろっているか）「シャツを首に通して……」……

毎回、全部指示出すのは面倒でありませんか？
子供には、**「着替えてください（服を）」** というような命令で指示を出せたらいいですよね？

全部自分のプログラムでやろうとしない

自分が作ったクラスから、役割を持つ他のオブジェクトに対して指示を出すことで、関心の分離が可能となります。
大事なポイントとして、「処理を他のオブジェクトに**お任せ**する」ということが挙げられます。

処理をお任せすることと、お任せされる側の配慮

オブジェクトに処理をお任せする以上、そのオブジェクトの状態や振る舞いに外から干渉したくない（されたくもない）

- オブジェクトが持つ状態や、状態に対する変更処理を外部から直接触られると、予期せぬ不具合や、副作用が発生する。
  - 意図しない操作をされたくない部分を、外部から操作できないようにする

→ これを**カプセル化（隠ぺい）**といいます。
適切なカプセル化を施すことで、オブジェクトの使いやすさ向上につながります。

設計時に振る舞いの名前や入出力情報だけ決める

昨今のオブジェクト指向言語には、**インターフェイス**という概念があります。
インターフェイスにはクラスの振る舞いの名前や渡されるべきデータ型、振る舞いによる結果のデータ型だけを定義しておけます。

- インターフェイスは、それ自体をデータ型として使える便利なもの
- オブジェクト指向は、プログラミングだけでなく設計としての側面も強い

オブジェクト指向言語で躓きやすい継承

難しく考える必要はありません。
同じ状態、同じ振る舞いを持つクラスをそれぞれコピペで作ると、管理が煩雑になってしまいます。

「似たデータ、似た処理をもつオブジェクトを、クラスの時点でまとめよう。」として使用するのが継承という考え方になります。
単にまとめるための手段だと知っておきましょう。

（「犬クラスは鳴く動物」「猫クラスも鳴く動物」それぞれは「鳴く動物クラス」を継承させることでどちらも「動物として鳴く機能」があるといったたとえ話もよくあります。それが必要な場面もありますが多くの場合は分かりづらさの象徴でもあります）

システム開発に「銀の弾丸などない」

古くから言われていることですが、システム開発に「銀の弾丸などない」のです。オブジェクト指向を取り入れたことによる弊害ももちろんあり得ます。
以下、一部の例を挙げます。

- ごく小規模の案件で、無理矢理取り入れることによりコードの記述量が膨大になってしまった。
- 単機能の呼び出しが連続するようなシステムなど、オブジェクト指向的な考え方が不向きなシステムだった。
- 管理しきれない副作用の発生.....など

オブジェクト指向は効率良く開発を行うための考え方

歴史を辿ると、システム開発の効率を求めた結果、オブジェクト指向という体系ができたに過ぎません。
これは通過点であり、オブジェクト指向の考え方を学ぶことですべてのことが説明できる気になったりましますが、これですべてを賄えることはありません。
この講義では、オブジェクト指向という道具を学びますが、これはただのプログラミングテクニックの1つとして、使いたいときに使えるよう、適切な使い方を身に付けましょう。