C#の世界観

suzusime

平成 28 年 6 月 4 日

1　オブジェクト指向に関して

我々はクラスを使って「もの」を表現し、その聯関を以てプログラムを設計することになりますが、これはオブジェクト指向と呼ばれる考え方によるもの（であるはず）です。詳しいオブジェクト指向の定義などは私にはよく分からないのですが、これはすなわち、「プログラムとして実現するために、世界をどのように見て、どのように解釈して、どのように抽象するか」という一つの方法論である訳で、大切なことだと思うので至らぬ理解かも知れませんが解説していきます。

1.1　クラスと実体（インスタンス）——内包と外延

世界の見方として、哲学には内包と外延という考え方があります。

『精選版日本国語大辞典』での定義を引き写しておくと、**内包（comprehension）**とは「概念が適用されるすべての事物に共通する性質の総体」であり、**外延（extension）**とは「概念が適用される事物の全体の、元の概念に対する称」です。例としては、「人間」という概念に対して、その内包が「理性をもつこと」や「動物であること」であると挙げられています。一方で「人間」に対する外延を考えれば、「朝永振一郎」とか「ポール・ディラック」、内包の定義によっては「友利奈緒」とか「長門有希」なんかも入るかも知れません。

C#に於けるクラスと実体はちょうどこの内包と外延の関係を持っています。

例えば Human クラスを作ったとして、そこに年齢を表すメンバ変数 age をもたせれば、それは「年齢という情報を持つ」という内包を定義したことになります。一方で、`Human nagato = new Human();` のようにして実体を生成すれば、その実体が外延になるのです。

このように、内包と外延という見方は、オブジェクト指向に於いて基本的です。

1.2　継承——特殊化

継承は、ある概念（＝クラス）の内包が、別のある概念のもつ内包を含んでいるときに行われます。

例えば、「長方形」の内包が「四角形である」「すべての角が直角である」の 2 つであり、「正方形」の内包が「四角形である」「すべての角が直角である」「すべての辺の長さが等しい」の 3 つであったとき、正方形の内包は長方形の内包を含んでいるといえます。

このようなとき、「正方形は長方形である」と言うことが出来ますから、**is-a 関係**であると言われます。念のため注意しておきますと、ここでの is は $=$ でも $\in$ でもなくて、$\subset$ を表します。

但し、ある概念をプログラムで表現するときには内包として選べるものがかなり制限されてしまうことに注意してください。上で挙げた「四角形である」「すべての角が直角である」などはどうもプログラムで表現するのは難しそうです。そこで、例えば長方形に対して内包として「長辺の長さ a をもつ」「短辺の長さ b をもつ」をもたせ、正方形に対して「辺の長さ a をもつ」をもたせてしまうと、もはや正方形の内包は長方形の内包を含んでいるとはいえません。

ゲームとして考えやすい例としては、「画面上の物体」のクラスは座標を持ち、「敵」のクラスは座標と HP、攻撃力、守備力をもち行動として攻撃と守備ができる、「魔法を使える敵」は座標と HP、攻撃力、守備力に加えて MP をもち行動として行動と守備と呪文詠唱ができる……というものがあるでしょう。このようにプログラムで実現できる内包は

「〜をもつ」というものが多くなると思います。

プログラムにおいては、上の例に於ける「魔法を使える敵」のクラスを「敵」のクラスを利用して定義することができます。これが**継承**です。このとき、前者を**派生クラス**、後者を**基底クラス**と呼びます。

継承を行うことで、同じコードを何度も書く必要がなくなる（実装の継承という面）だけでなく、派生クラスが基底クラスの特殊化であることを明示できます（インターフェースの継承という面)。このことにより、例えば「敵」の実体を入れる配列に対して「魔法を使える敵」の実体を突っ込む、といったことが型安全性を保った上で可能になります。

1.3 合成（コンポジション）

上述の継承はある一つの概念から別の概念を作り出す一つの手段でした。代表的なもう一つの手段として**合成**があります。

合成は、いわばある概念を用いて他の概念を定義する事です。例えば「2 次元実数ベクトル空間」という概念に「2 つの実数の組である」を内包として考えれば、これは「実数」という概念を内部で用いているということになります。

これを C#で実現する手段が、単に別のクラスの実体をメンバ変数にもたせるというものです。これは **has-a 関係**をもつと言われます。あまりに単純でわざわざ「合成」などと名前を付けるほどではないかと思うかも知れませんが、継承と合成との区別は設計上重要です。

ゲームらしい例を出すとすれば、敵のクラスが自分の撃った弾の実体を入れた配列をもっている、というものが挙げられます。

1.4 カプセル化

C#でクラスを書くときに、public や private という言葉がでてきたと思います。これらはアクセス指定子と呼ばれるもので、その変数や関数を呼び出すことの出来る範囲を指定するものです。

public なものはどこからでも呼び出せますが、private なものは、そのクラスのメンバ函数からしか呼び出せません。protected なものはそのクラスだけでなく派生先のクラスの中からも呼び出せます。たいていの場合、メンバ変数はすべて private にし、外部から触れたいものについては、それを読み出し、変更する public なメンバ函数を定義します。C#にはそれを簡単に行うための**プロパティ**という機能もあります。

これらは**カプセル化**（の一部であるデータ隠蔽）を実現する機能です。なぜこのような面倒なことをするのでしょう。全部 public にしてしまったほうが簡単で良いのではないか、そう思うかも知れません。が、これが推奨されるのには理由があります。ここにはふたつ理由を挙げておきます。

第一点は、バグを埋め込みにくくなることがあります。

例えば人間のクラスが年齢をメンバ変数としてもっているとき、年齢は外部から読み出すことはあっても書き込むことはないでしょう。こういうものには取得函数（getter）だけを用意し、設定函数（setter）は用意しなければ、誤って外部から書き込むことはありません。他にも、所持金額をメンバ変数にもっているとすれば、これは外部から設定したいかも知れませんが、その際に間違って負の値を設定しようとしてしまうかも知れません。こういうのも、設定函数を間にかませれば、もし負の値が与えられたときに代わりに 0 を代入する、といった処理を行うことが出来ます。

第二点は、実装の変更に対して柔軟になることです。

例えば、三次元ユークリッド空間上の点を表すクラスがあったとしましょう。それは x,y,z 座標を内部に持っていましたが、計算の都合上極座標をもっていたほうが良いので変更したい、となったとします。このとき、もし外部からじかにクラスの持つ x,y,z 座標を参照していたとしたら、参照している箇所をすべて書き換えないと行けなくなります。一方、取得函数、設定函数をとおして参照していれば、その取得函数と設定函数の中を書き換えてさえしまえば、他は書き換えずとも内部の実装を変更できるのです。つまり、private な部分は容易に書き換えられ、public な部分は一度書いてしまうとなかなか書き換えられないのです。

このように、出来る限り private にすることによって、実装の変更にかかる手間が少なくなります。

1.5　ゲームの設計について

繰り返しますが、オブジェクト指向プログラミングでは、望む系を「オブジェクト（もの)」の相互作用とみなして記述します。すなわち、我々は望む系の挙動を観察（想像）して、そこからものを切り出していかねばなりません。

例えば将棋ゲームをつくるとしましょう。すると、まず駒が必要なことはわかるでしょう。が、それだけではありません。盤面も必要ですし、棋士も必要です。それを C#のプログラムにするにあたっては、駒と盤面と棋士のクラスを作ろう、「持ち駒」のリストも必要だから、これは持ち駒のリストを棋士のクラスに持たせることで実現しよう……とこんなふうに考えていきます。

シューティングゲームならば、自機がいて、敵がいて、自分と敵のそれぞれの弾は打ち手が管理していて……いや、でも自機や敵の撃った弾以外の障害物もほしいし、いっそ当たり判定を持つものは全部一つの管理クラスに持たせた方が……とか色々考える必要が出てきます。たぶん答えを簡単に定める方法はないでしょう。とりあえず紙か何かに書いてみて、なんとなく問題がなさそうならそれで試しに作ってみると良いと思います。試行錯誤しているうちにプログラムを書く速度も上がるでしょうし、失敗することで身につくことがたくさんあると思います。

ただ、オブジェクト指向は長年使われているものなので、どういう使い方が効果的か、という知見を先人達が積み上げてきています。クラスの書き方が少しわかってきたなと思ったら、そういうものを調べてみると良いと思います。例えば「継承よりも合成が良い」と言われていたりします。もちろん継承は絶対駄目ということではないでしょうが、「何も考えずに継承でクラスを作るまえに、合成を使っても望む構造は実現できないか考えてみるべきである、なぜなら合成のほうが柔軟であるから」とそういうふうなことがインターネットの海に潜ればあちこちに書かれています。そういう記事を読んで何故合成が柔軟だと主張しているかを理解すれば、自ずと自分がこれからプログラムを設計する際にも活きてくると思います。

他に「デザインパターン」と呼ばれるものもあります。これは、もっと具体的に「〜のような構造を実現したければこのようにクラスを作ってやればいい」という指針を与えるものです。有名なものは GoF による 23 のデザインパターンですが、これを解説したりおかしいから改良すべきと指摘したりと、たくさんの記事があるので、興味があれば浚ってみると良いと思います。

2　コーディング規約について

コーディング規約とは、コードを書くときに守るべきとされる規範です。例えば C#では空白や改行は基本的に無視されますから、これらをめちゃくちゃにいれてもプログラム自体はきちんと動いてしでも、きちんとブロックごとに字下げを入れてやると、全体の構造が一目で見渡せて読みやすいはずです。

コーディング規約を守ることによって。見やすい、他の人が見ても意味が伝わりやすい、などの利益を得られます。

3　値型と参照型

C#には値型と参照型という概念があります。それぞれ特徴があるので注意しましょう

4　参考文献

- スコット・メイヤーズ著、小林健一郎訳『Effective C++ 第 3 版』（丸善出版）