特集

ソフト&ハード，両方知っててあたりまえ！

第4章

ハードウェア技術者のための オブジェクト指向& C++ 言語入門

――オブジェクト指向は ハードウェア設計の考え方に近い

山崎 正実

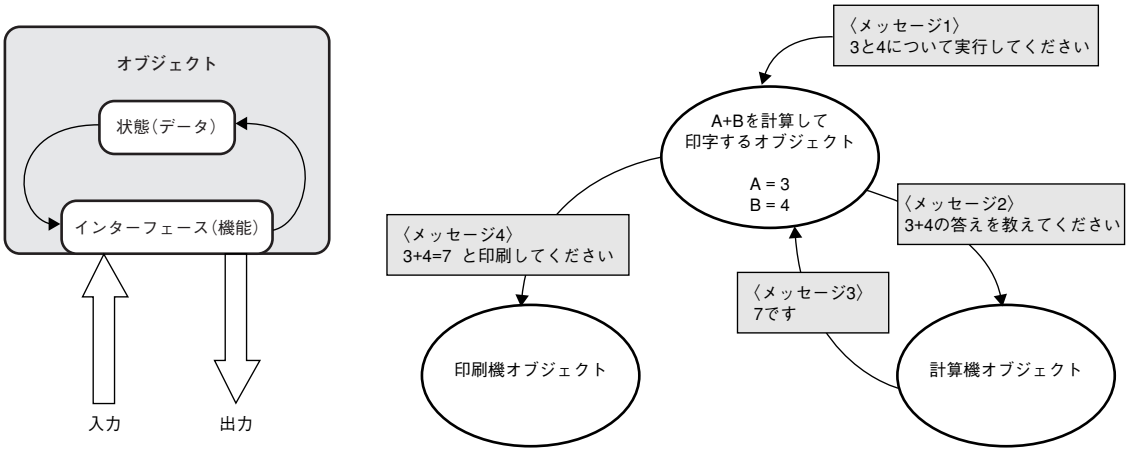
ここでは，ハードウェア設計の知識のある技術者を対象に，オブジェクト指向プログラミングの考え方とC++ 言語の概要について解説する．オブジェクト指向では，システムを，それを構成する個々のオブジェクトとオブジェクト間のメッセージの通信に分けて考える．これは，ハードウェアのモデリングの考え方に近い．後半では，スタックを例に，C++ を使ったハードウェアのモデリング手法について紹介する．なお，本記事で紹介したサンプルのC/C++ データは，本誌付属のCD-ROMに収録されている．（編集部）

オブジェクト指向プログラミング（OOP：object oriented programming）は，ハードウェア設計者にとっても，システムのプロトタイピングやテストベンチ開発を行ううえで非常に強力なツールになります．ここでは，C++ 言語を取り上げ，簡単なハードウェアのモデリングを通してオブジェクト指向プログラミングの考え方を紹介したいと思います．

オブジェクト指向は，システムを構成しているオブジェクトという概念によりシステムを実現，あるいはモデル化する方法です．オブジェクトとは，目的とするシステムを実現するための，ある役割を担う“もの”です．オブジェクトは，その機能を果たすための手続きと内部状態（データ）をもっています．たとえばコンピュータ・システムでは，プロセッサやメモリ・システム，バス，入出力装置などをオブジェクトと考えることができます．

オブジェクト指向では，システムは，それを構成する個々のオブジェクトとオブジェクト間のメッセージの通信によってモデル化されます．**図1** はその概念図ですが，ハードウェア設計の機能ブロック図とよく似ています．このため，ハードウェア設計者の方々には，これまでやってきたハードウェア設計とどこが違うのだろうか？ と思われるかもしれません．

はっきり言ってしまえば，あまり変わりません．VHDL ではコンポーネント，Verilog-HDL ではモジュールと呼ばれるものがオブジェクト指向プログラミングにおけるオブジェクトにあたります．ハードウェアの設計では，オブジェクトがサブシステムからRTL モジュール，回路素子へと段階的に詳細化



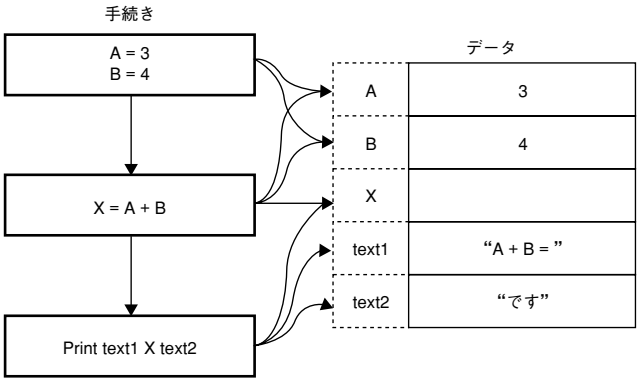
〔図1〕オブジェクト指向プログラミング

オブジェクトは機能を果たすための手続きと内部状態（データ）をもっている．オブジェクト指向プログラミングでは，オブジェクト間のメッセージ通信をもとに処理が進んでいく．

されていくという特徴はありますが，システムが個々のオブジェクトとオブジェクト間の通信で構成されるという，オブジェクト指向のパラダイムの外には出ていないと思われます．

ところが，従来型のソフトウェア設計の世界は違いました．ソフトウェアでは一般に，一つの手続きを効率的に多数のデータに適用できるように，オブジェクトを手続き（プログラム）と処理対象となるデータに分けて実装します．ノイマン型コンピュータにとって，この方法がもっとも効率的な実装形態だからです．処理を行う際には，手続きに対してデータを渡して処理するという形になります．このような実装形態は，オブジェクト指向に対して，手続き指向と呼ばれています（**図2**）．

このように，オブジェクト指向はソフトウェア設計の世界では特別な概念なのですが，ハードウェア設計の世界ではごく普通の（あたりまえの）概念なのです．



〔図2〕手続き指向のプログラム
手続き指向では，手続き（プログラム）と処理対象となるデータを分けて実装する

1．オブジェクト指向プログラミング

まず，オブジェクト指向プログラミングとはなんなのかについて説明していきます．

●なぜオブジェクト指向プログラミングなのか？

ソフトウェア設計の世界では，プログラムの大規模化，複雑化にともなって，品質と納期の確保が困難になり，いわゆる「ソフトウェアの危機」が1970年代ごろから盛んに叫ばれるようになってきました．このことを背景として，構造化プログラミングなど，数々のプログラミング手法が開発されました．オブジェクト指向もその一つで，1980年代ごろから注目され始め，一般的に使われるようになってきました．さて，オブジェクト指向が台頭してきたのは，なぜでしょうか？

一般に，大規模化，複雑化するシステムをきちんと開発するためには，次のような手法がとられます．

- **システムをサブシステムに分割し，サブシステムごとに並行開発する**
- **過去の設計資産を再利用して，実質的な開発量を減らす**

サブシステムに分割して開発を行う手法は，分割統治法（divide and conquer）と呼ばれます．ハードウェア設計者の方なら，ソフトウェアとのインターフェースであるアドレス・マップを勝手に変更してしまい，ソフトウェア開発者への変更通知を忘れ，後でひんしゅくを買った経験がありませんか？ この手法を適用するうえでもっとも大事なことは，サブシステム間のインターフェースをなるべく早く明確に定義することです．また，シンプルでわかりやすいものにすることも非常に重要です．

メモリ・マップドI/Oによるソフトウェア/ハードウェア・インターフェースでは，ハードウェアとして実装したとおりの内部アドレス・マップをそのままソフトウェア/ハードウェア・インターフェースとすることがあります．しかし，分割統治法の観点から見ると，これはよくありません．設計の修正・変更，仕様の追加・変更などで実装方法が変わることはよくある話で，それがサブシステム間インターフェースの変更要因になるからです．実装の変更がほかのサブシステムに影響を与えるようなことは，極力防がなければなりません．このため，外部インターフェースは内部の実装に依存しない抽象的なものにしたいのです．また，機能（インターフェース）追加も容易に実現できるようになっていなければなりません．

「過去の設計資産を活用しよう！」．技術者であれば一度は聞いたことのあるスローガンでしょう．しかし，言うはやすく，行うは難し，であることも事実だと思います．ハードウェア設計でも，プロセッサや画像処理用のマクロ・セルといった大物になるとさすがにあきらめて再利用を考えるようですが，一般的には，自分の資産や所属するチームが開発した資産以外は，なかなか再利用が進まないのが現実だと思います．

このあたりの事情を探ってみると，以下のような，まことにもっともと思われる理由で断念していることが多いようです．

- **機能の修正・追加が必要．いらない機能は削りたい．中身がぐちゃぐちゃで，どこを直したらよいのかわからない．**
- **機能の独立性が低く，インターフェースがとりにくい．たとえば外にシーケンサを置いて，さまざまな制御信号やタイミング信号を入力しなければならなかったり，データ構造が不整合だったりする．**

これらのことを整理すると，分割統治法や設計再利用が容易なプログラムは，以下の要件を満足する必要があると思います．

- **機能の独立性が高い**
- **インターフェースが抽象化されており，実装と分離されている**
- **インターフェースに影響を与えないで，容易に機能を追加・変更できる**

このような必要性から出てきた手法がオブジェクト指向プ