プログラミング、これからの 10 年

千　葉　　滋[†]

Smalltalk-80 以来 20 年がたち、オブジェクト指向技術は広く使われるようになった。今日では「当たり前」の技術である。本稿では、まず、オブジェクト指向技術の普及を後押しした背景を考え、そこから次の 10 年の技術動向について予測する。現在、研究ステージにある技術から 2 つ選んで紹介し、それらの技術が今後実用化されると、産業界にどのようなインパクトを与える可能性があるか、それに対してソフトウェア開発者はどのように対応すればよいのか、を考える。

Next 10 years of Software Developers

Shigeru Chiba[†]

Object-oriented (OO) programming has been widely recognized as one of the really useful ideas since Smalltalk-80 was released more than twenty years ago. This article first describes why OO programming made such a success and then it foresees what technology will come out in the next ten years. It presents two research topics actively investigated now in academia, the potential impacts by those topics in the industry, and how industrial engineers should survive with them.

1. はじめに

C++、Java の普及により、今やオブジェクト指向技術は「当たり前」の技術になった。IDC の調査を引用した ZDNet News の記事[1] によると、2000 年の段階で、全世界のプログラミング言語別のソフトウェア開発者数は 1 位が C/C++ で 300 万人、2 位が Visual Basic で 230 万人、3 位がいわゆる 4GL で 180 万人、4 位が Java で 120 万人である。ちなみに 5 位は Cobol で 110 万人だそうである。1 位の C/C++ のうち、どのくらいの割合が C++ なのかは不明だが、相当数の開発者がオブジェクト指向言語を使っていることは間違いなさそうである。とくに Web アプリケーションなどの比較的先進的と考えられるソフトウェアの開発ほど、オブジェクト指向言語が使われる傾向にあるのではないだろうか。

一方、大学などの教育現場を見てみても、プログラミングの初等教育を C から Java へ切り替えているところが多いと聞く。筆者の勤務先でもプログラミングの初等教育に Java を使っている (もちろん最初はオブジェクト指向プログラミングはほとんど教えず、単なる手続き型言語として Java を教えている)。また、初等教育こそ C でおこなうが、その後に C++ または Java によるオブジェクト指向についての講義を用意しているところも多いようだ。

1.1 開発者の憂鬱

このような状況になった理由は、結局のところ、最近開発されたプラットフォーム用にソフトウェアを開発しようとすると、好むと好まざるを得ず、オブジェクト指向言語を使わざるをえない、というところではないだろうか。例えば、PC 用のアプリケーションを開発しようとすると、オペレーティング・システムが提供するライブラリ (API) を使わなければならないが、最近のものはクラス・ライブラリであり、オブジェクト指向言語なしには使えない。また、Web アプリケーションを開発しようとすると、かなりの確率で JSP や EJB を使って開発することになるだろう。これらは皆 Java でプログラムが書けなければ使えない。

また、今現在 VisualBasic を使っている開発者の非常に多くが、C# または Java に乗り換えるべく、オブジェクト指向技術への関心を高めているそうである。理由はもちろん .NET プラットフォームである。この新プラットフォームへの移行により、ますますオブジェクト指向技術の知識なしにはソフトウェア開発ができない状況が強まるだろう。

最近のアプリケーション・ソフトウェアは、オペレーティング・システムの上に、さまざまなミドルウェア、

† 東京工業大学 大学院 情報理工学研究科 数理・計算科学専攻
Dept. of Mathematical and Computing Sciences, Tokyo Institute of Technology

ライブラリ、フレームワークを積み重ね、その一番上に構築される。したがって、そのような基盤ソフトウェアに新技術が導入されると、その上のアプリケーション・ソフトウェアにも、その新技術を導入せざるをえない。アプリケーション・ソフトウェアの開発者といえども、新技術から無縁ではいられないのである。

この辺り、今でも Fortran が使われている科学技術計算の世界の逆である。科学技術計算の世界では、基盤ソフトウェア (Fortran ライブラリ) があまりにも昔に完成してしまい、その後のアプリケーション・ソフトウェア開発への新技術の導入をも阻害してしまっている感がある。

1.2 開発者の宿命

極端な言い方をすると、新技術の導入に関して、アプリケーション・ソフトウェアの開発者は、基盤ソフトウェアの開発者に支配されているのである。例えば、基盤ソフトウェアの開発者がオブジェクト指向技術の導入を決めたならば、アプリケーション・ソフトウェアの開発者もその決定に従わなければならない。従わなければ、淘汰されるだけである。

表面的にはオブジェクト指向技術を取り入れるが、アプリケーション・ソフトウェア全体をオブジェクト指向技術で構築するのは見送る、という選択肢もありうるが、長期的にはうまくゆかないだろう。例えば、JSP と EJB を使って web アプリケーションを開発すると決めたとする。当然、EJB を使うのであるから、アプリケーション・ソフトウェア全体をオブジェクト指向技術で設計するべきである。しかしあえて、オブジェクト指向言語である Java を表面的に利用するだけにとどめて開発をおこなう、という話を時折耳にする。出来上がったアプリケーション・ソフトウェアは、一応、いくつかのクラスから構成されるのだが、個々のメソッドは非常に長く、オブジェクト指向ではない旧来型のスタイルで書かれている。

このような選択肢を採用する理由は、オブジェクト指向技術を導入して設計したくとも、それを実行できる人材がいない、そのような人材を養成する余裕がない、などだろう。あるいは、もう少し積極的 (?) に、オブジェクト指向技術の有効性が納得できない、信用できない、という理由の場合もあるかもしれない。EJB を使うので、必要最小限のクラスを定義するところまでは譲歩するが、それ以上オブジェクト指向技術を導入することはせず、残りの部分は信頼のおける旧来のスタイルでプログラムを記述する、とうわけである。

ところが、そのような選択肢をとるアプリケーション・ソフトウェアの開発者は長期的には淘汰される。例えば、EJB の場合、その上で動くアプリケーション・ソフトウェア全体もオブジェクト指向技術を導入して作成しなければ、EJB を利用する効果がでない。そうでないと、役にたたないクラスをいくつも定義する分プログラミングが面倒で、かえって開発効率が悪化してしまう。

オブジェクト指向のような新技術が基盤ソフトウェアに導入された場合、その上に構築されるアプリケーション・ソフトウェアにも同じ技術を導入すれば、その基盤ソフトウェアの利点を最大限に引き出すことができる。逆に、導入しなければ、その基盤ソフトウェアの利点を引き出せないばかりか、その基盤ソフトウェアを採用しない方がよかった、ということになりかねない。アプリケーション・ソフトウェアの開発者は、基盤ソフトウェアの開発者の新技術に対する嗜好を、そのまま受け入れなければならない宿命にあるのである。

2. 基盤ソフトウェアの開発者の考え

基盤ソフトウェアを導入する目的は、つまるところ、アプリケーション・ソフトウェアの開発効率を高め、開発者の生産性を改善することである。多くのアプリケーション・ソフトウェアが共通に必要とする機能を提供できれば、その基盤ソフトウェアを使うことにより、特定のアプリケーション・ソフトウェアのために新たに書かねばならないプログラムの量が減る。さらに簡潔でわかりやすい API を通してその機能を利用できれば、それを利用する開発者の負担も減るのでなおよい。

基盤ソフトウェアの間にも競争があるので、この目的を達成するためであれば、オブジェクト指向のような新技術は、基盤ソフトウェアに貪欲に取り込まれることになる。もちろん、基盤ソフトウェアを利用する開発者に支持されなければならないので、やみくもに新技術が導入されるわけではない。その昔、Macintosh の最初の標準開発言語は Pascal であった。アセンブラよりは Pascal の方が技術的にずっと進んでいたのであるが、誰もが知っているように、Pascal はその後 C に置き換えられてしまった。Pascal は開発者に支持されなかったわけである。

だからといって、アプリケーション・ソフトウェアの開発者に受け入れられる新技術が、基盤ソフトウェアに導入されるわけではない。もしそうだとすると、前節で述べていた考えと矛盾してしまう。前節では、アプリケーション・ソフトウェアの開発者は、基盤ソフトウェアに気まぐれ (?) に導入される新技術に振り

回される運命にある、と述べた。

ではどのような新技術が導入されるのか。正確に述べると次のようなことである。基盤ソフトウェアに導入される新技術は、先進的なアプリケーション・ソフトウェアの開発者に支持される技術である。アプリケーション・ソフトウェアの開発者のうち、先進的な人々は、常に新技術の動向を見ており、よさそうなものをいち早く取り込んで、自らの生産性の向上に役立てようとしている。したがって、前節で述べた憂鬱な開発者とは彼らのことではない。憂鬱なのは、技術的に保守的な開発者達である。彼らは、先進的な利用者の趣味に沿って新技術が導入され、高度化していく基盤ソフトウェアを、たとえ望んでいなくても使わなければならない。そして使ってゆく以上、彼らも新技術を導入してゆかなければならない。

3. 今後普及するかもしれないもの

オブジェクト指向技術が当然となった今日、先進的なアプリケーション・ソフトウェアの開発者達は何を見ているのだろうか。ここでは、今後、ソフトウェア開発に導入されるかもしれない新技術のうち、2 つを選んで紹介する。

3.1 Generative Programming

Generative Programming[2),3)] (GP) とは、ソフトウェアをひとつひとつ個別に手作りするのでなく、仕様記述から自動合成するようにしよう、というパラダイムである。任意のソフトウェアをこの方法で作成することは現実的ではないが、作成するソフトウェアを特定の製品ライン (product line) に限れば十分実用性がある。

実際、web アプリケーションの開発では、おおよその半完成品を用意しておき、顧客の要求に合わせてそれをカスタマイズして、最終製品にすることなど、ある程度、現実におこなわれていることである。GP の主張は、この最後のカスタマイズを手作業でおこなうのではなく、なんらかの仕様記述言語 (プログラミング言語) でカスタマイズの方法を記述することにし、実際のカスタマイズは自動的におこなうようにしよう、ということである。

製品ラインに共通な基幹部分と無数のオプション部品をあらかじめ用意しておけば、仕様記述にしたがって、それらを自動的に組み合わせ、各種パラメータを調整した上で最終製品にすることは可能であろう。GP のアイデアは工業製品の組み立て工程から発想されたようである。実際、GP を提唱している Krzysztof Czarnecki 氏は独ダイムラークライスラーに勤務している。自動車は、基本となるシャーシの上に、異なるエンジン、異なるボディ、異なる塗装、内装、オプションをのせることで完成する。これにより、共通のシャーシから趣の異なる複数の車種を生み出し、個々の車種を顧客の予算や好みに応じてさらに細かいグレードにわけることが可能になる。ソフトウェアの開発にも、この方法を持ち込もう、というのが GP の目標である。

部品を組み合わせて最終製品にしよう、というだけでは、いわゆるコンポーネント技術と同じであるが、GP では部品同士を組み合わせるためのプログラムの自動合成 (code generation) にも重点がおかれている。例えば、最近のソフトウェア開発環境には、GUI (Graphical User Interface) 部品を画面上で配置すると、そのような配置を実現するプログラムを自動合成する機能がついている。配置される GUI 部品は、クラス・ライブラリとして提供されるのだが、実際にそれらを配置するためには、個々の GUI 部品に対応するオブジェクトを生成して、様々なパラメータを設定するためのプログラムが必要である。最近のソフトウェア開発環境は、このプログラムを自動合成するが、この自動合成の技術に重点をおいているのが GP である。部品の組み合わせ方の仕様をどのように記述させるか、得られた記述からそれを実現するプログラムをどのように合成するか、が GP の研究課題である。GUI 分野では、これらの課題は既によく研究されているが、それ以外の分野ではさらなる研究が必要である。

最も簡単な GP の例は、C++ の template 機構を使ったプログラミングである。C++ の template 機構は、コンパイル時にパラメータとして与えられた定数値や型名を元に、プログラムのひな型 (template) から普通のクラスや関数を自動合成する機構であるといえる。パラメータとして、別な template を与えることもできるので、複雑な合成も可能である。このため、あらかじめ必要な部品を template として書いておけば、template の組み合わせを指示する短く簡潔なプログラムを書くだけで、非常に長く複雑なプログラムを C++ コンパイラに自動合成させることができる。

GP の考え方は、ドメイン専用言語 (domain specific language) という名前で研究されている考え方にも重なる。この研究は、アプリケーション分野ごとに専用のプログラミング言語を用意して、より簡潔な記述でその分野のアプリケーションを開発できるようにしよう、というものである。そのような専用言語は、C++ や Java のような既存の汎用プログラミング言語に機能拡張をほどこしたものである場合もあり、独

自の文法をもつ言語である場合もある。

JSP (Java Server Pages) は、ドメイン専用言語の一例である。JSP は、サーバ側で web page を動的に生成するためのプログラミング言語と考えられる。同様の処理は Java Servlet API を使って普通の Java 言語で記述することもできるが、JSP を使えばより直感的に記述することができる。

ドメイン専用言語のアイデアは、特定のアプリケーション分野に特化した言語を用いれば、開発者はソフトウェアの本質的な部分に集中して開発ができるはずだ、というものである。汎用プログラミング言語をもちいると、煩雑なロジックの記述に大半の時間をとられてしまい、開発効率が大きく低下する。例えば、JSP を使わずに Java Servlet API だけを使って開発をおこなうと、開発者は文字列を連結して最終的にひとつの html テキストにするところまでを細かく Java 言語で記述しなければならない。一方、JSP を使えば、開発者はそのような非本質的な文字列処理を直接記述する必要がない。実装の細部から離れた抽象的なモデル上でソフトウェアを記述し、そこから実際に動作するソフトウェアを自動合成するというアイデアは、OMG が提唱する Model-Driven Architecture (MDA)[4] にも見られる。

ドメイン専用言語は、web や GUI の分野では既に実用化されているといえるが、それ以外の分野での応用はあまり進んでいない。理由として、ドメイン専用言語のコンパイラを作成するコストが大きいことがあげられる。Web や GUI のように市場が大きく、ドメイン専用言語のニーズが高い分野では、高いコストをかけてドメイン専用言語を作成することができる。一方、そうでない分野では、ドメイン専用言語を作成してアプリケーション・ソフトウェアの開発効率を向上させても、ドメイン専用言語とアプリケーション・ソフトウェアの両方の作成コストを考えると、全体ではかえってコスト高になってしまいかねない。

この問題を解決するために、拡張可能な言語の研究が活発におこなわれている。無から新しい言語を作成するのは多大な労力がかかるが、既存の言語に手を加えて部分的に機能拡張するだけなら、それほどの労力がかからないよにできるはずである。現在、マクロ処理系、template、リフレクションなど、さまざまな要素技術に基づいた拡張可能な言語が研究開発されている。そのような言語が実用化されれば、ドメイン専用言語を作成するコストが下がり、より多くの分野でドメイン専用言語が使われるようになるだろう。

3.2 Aspect Oriented Programming

アスペクト指向プログラミング (AOP)[5),6)] は、新しいソフトウェアのモジュール化の技術である。オブジェクト指向技術により、ソフトウェアのモジュール化技術は大きく進展し、既存のソフトウェアのプログラムの大半を再利用しながら、新しいソフトウェアを開発できるようになった。ところが、ソフトウェアを再利用可能なモジュール、つまりクラス、に分割するのは、現実には簡単なことではない。AOP は、既存のオブジェクト指向技術ではうまくモジュールに分割できないような事例に対応するための技術である。

オブザーバー・パターンはよく使われるデザインパターンのひとつである。このパターンは、例えば図形エディタで、図形を表すオブジェクトが変更されたとき、画面の描画を担当するオブジェクトがそれを検出して、変更を画面に反映できるようにするために使われる。前者のオブジェクトをサブジェクト、後者をオブザーバーと呼ぶ。

オブザーバー・パターンにしたがって設計すると、サブジェクトとオブザーバーのクラスに `notify`, `update` というメソッドを加えることになる。サブジェクトを変更したときには同時に `notify` を呼び、`notify` がオブザーバーの `update` を呼んで、変更を通知する。このような設計は、オブジェクト指向技術の枠内では正しい設計である。しかし、別な観点 (aspect) から見ると、`notify` や `update` は、図形の変更に対して画面の状態を正しく保つという機能を、協調して実現しているので、独立した 1 つのモジュールになっていてもよさそうである。その方が、デバッグやプログラムの再利用に便利である。

オブジェクト指向技術のような従来技術では、ソフトウェアを原則として 1 つの観点だけから見て、モジュールに分割する。このため、上の例のように別な観点から見ると 1 つのモジュールに集めるべきプログラムの断片が、しばしば複数のモジュールに分散してしまう。AOP では、このようなプログラムの断片を、モジュール横断的な関心事 (crosscutting concerns) と呼ぶ。これを、独立したモジュールに分離できるようにするのが AOP の目標である。

ソフトウェアを関心事ごとにモジュールに分割する (separation of concerns)、というアイデアは、プログラミングの歴史のごく初期に生まれたものである。構造化プログラミングはブロック構造で、オブジェクト指向プログラミングは階層構造で、ソフトウェアを分割するための技術である。最近ではリフレクションでソフトウェアを分割する技術が研究されてきた。AOP

は、この流れの最新の動きであるといえる。

AOP では、既にオブジェクト指向技術における Smalltalk (あるいは Simula か) に相当する言語が生まれている。それが AspectJ[7] である。これは AOP の提唱者の一人である Gregor Kiczales 氏らによって開発された Java を拡張して AOP を可能にした言語である。AspectJ は実用言語となることを目指して設計開発されおり、最近では、一般の雑誌等でも解説記事が掲載されるようになってきた[8]。

AspectJ は Java の拡張版なので、プログラムの大半は通常の Java で書かれる。しかしながら、通常の Java の範囲内では、モジュール横断的になってしまうプログラム断片を、AspectJ では、1 個の独立したモジュールに集めることができる。このモジュールのことを aspect と呼ぶ。これは Java のクラスに相当する概念である。

AspectJ では、メソッドが呼ばれた瞬間や、フィールドがアクセスされた瞬間を合流点 (join points) と呼ぶ。アスペクトの中に集められたプログラムの断片 (advice と呼ぶ) は、コンパイラによって、この合流点で残りの Java プログラムと結合 (weave) される。個々の advice が、具体的にどの合流点と結合されるかは、aspect の記述の中で定義される。例えば、`Point` クラスの `move` または `setX` メソッドが呼ばれる瞬間、のように定義される。要は、AspectJ が許す全ての合流点の集合の部分集合を定義することになる。AspectJ では、この部分集合のことを pointcut と呼ぶ。

AspectJ の言語仕様を斜め読みすると、CLOS の before/after メソッドを Java に導入しただけのもの、と思えるかもしれない。あるいは、リフレクションでメソッド呼び出しの実行手順を変更できるようにした言語と同じ、と思う読者がいるかもしれない。しかしながら AspectJ の優れた点は、そのような古くからあるアイデアを AOP の観点から昇華させている点である。AspectJ の言語仕様を深く理解すれば、そのようなアイデアの焼き直しではないことがわかるだろう。

4. これからの 10 年

GP も AOP も開発者の生産性を飛躍的に向上させるための技術である。GP が実用化されると、ドメイン専用言語が既にあるアプリケーション分野であれば、ソフトウェアの本質的な部分だけを書くだけで、最終製品が自動的に合成される。AOP が実用化されると、さらに高度なモジュール化が可能になり、モジュールの再利用性が高まるだろう。その結果、やはり開発者が本質的な部分だけを書けば、残りの部分は既存モジュールを再利用してソフトウェアを作成できるようになる。

このような状況では、ソフトウェアの開発に必要な人員は大きく減少する。不必要に多いプログラムの行数や、いたずらに多い開発人員の数を誇ったりする風潮は、下火になるかもしれない。人々はプログラムを書かなくなるだろう。

これはソフトウェア開発者という職業がなくなることを意味するのだろうか? もちろん、そのようなことはない。GP や AOP は、深い知識と経験をもった上級の開発者の生産性を高めるものである。それらの技術によって、上級の開発者は現在の技術水準に換算して数十人、数百人分の量のプログラムを 1 人で書けるようになるだろう。しかし、そのような上級開発者は常に必要なのである。不要になることはない。

一方で、知識や経験があまりない開発者の人口は大きく減少するかもしれない。今日の工場では、単純作業がほとんど機械に置き換えられているのと同様、知識も経験も必要ないようなプログラミング作業は、ドメイン専用言語や AOP 言語のコンパイラに置き換えられてしまうだろう。新技術の導入に消極的な開発者は、淘汰されてしまう恐れがある。

結局、10 年後に生き残るのは、新しいドメイン専用言語や AOP 言語を設計・実装する技術をもった開発者と、顧客の望むソフトウェアをシステムとして設計し、それをドメイン専用言語なり AOP 言語なりで記述できる開発者だけかもしれない。どちらの技術をも習得できなかったソフトウェア開発者は消えてゆくのだろう。

いくらか悲しい結論である。だが今、筆者はある出来事を思い出している。以前、筆者はある企業の方に、「昔は一線の開発者だったのですか」の意味で、「昔はプログラマだったのですか」と聞いて、大変嫌な顔をされたことがある。アカデミックな世界では「プログラマ」という言葉は多少の尊敬語であると思うのだが、どうも産業界では差別用語であるようだ。これは、高い技術をもった技術者から、そうでない技術者まで、ソフトウェア開発者の幅が大変広く、分類上、たくさんの名前が必要だからだろう。アーキテクト、SI、SE、プログラマー、コーダー (死語だろうか)、と技術レベルに応じて呼び名を変えなければならないのである。筆者としては、技術革新により、下位のレベルの技術者が消え、誰もが素直にプログラマと名乗れるようになれば、それはそれでよいと、密かに思う。

謝辞 本稿をまとめるにあたって、議論に協力して

くださった他のパネリストに感謝します。

参 考 文 献

1) Wong, W.: 人気の高い Java 言語，問題は陣営内に……？，ZDNN 2002 年 3 月 27 日，(http://www.zdnet.co.jp/news/0203/27/e_java.html).
2) Czarnecki, K., U. W. Eisenecker: *Generative Programming, methods, tools, and applications*, Addison Wesley, (2000).
3) http://www.generative-programming.org/
4) http://www.omg.org/mda/
5) Aspect-Oriented Programming, CACM, vol.44, no.10, pp.28-97, (2001).
6) http://www.aosd.net/
7) http://www.aspectj.org/
8) 特集アスペクト指向プログラミング、Java World, IDG Japan, July, pp.84-99, (2002).