プログラミング 15−3
(1997. 9. 12)

討論会: さまざまなソフトウェア開発文明

情報処理学会プログラミング研究会

コーディネータ: 久野　靖, 加藤和彦(筑波大学)

プログラミング研究会はプログラミング言語やその理論, 処理系に関する研究発表を中心としているが, 世の中で実際にソフトウェアを開発する人たちが重要と考える技術との乖離があるのではないかという批判もある. 数年前までは開発技術や手法はソフトウェア工学研究会, 言語はプログラミング研究会で済んだかも知れないが, 今日では「デザインパターン」,「分散オブジェクト技術」,「コンポーネント」,「フレームワーク」など, 言語機構に近い部分で, ソフトウェア工学分野では注目を浴びているのにプログラミング研究会ではまだ十分取り上げられていない技術が現われている. この問題提起に基づき, これらの技術を推進する側とプログラミング言語側の両者で討論会を行い, 研究会としての問題点や今後のあり方を考えて行きたい. パネル討論というより, 座談会形式で全員が自由に話し合う形とする.

Open Discussion: New Cultures in Software Development

Special Interest Group on Programming, IPSJ

Coodinators: Yasushi Kuno, Kazuhiko Kato (University of Tsukuba)

Major topic area of IPSJ SIGPRO (Special Interest Group on Programming) includes programming languages, its theory, and implementation techniques. However, recent advances in software development technologies contains many language-related topics that have not been coverd sufficiently in SIGPRO. Design patterns, distributed object technologies, component programming, and (application-) frameworks are example of such new technologies. These topics are already covered by other SIGs (e.g. SIGSE — SIG on Software Engineering) and language-centered conferences (e.g. OOPSLA), but we would like to discuss those theme from the following viewpoint: (1) what contribution to those technologies from language researchers are possible, and (2) what is lacking in current language resarch community. This is an open discussion, rather than a panel, and opinion from the floor are greatly welcomed.

言語研究者の「理想と現実」

久野　靖[1]
(筑波大学経営システム科学専攻)

0　そもそものきっかけ

もともと、この討論会のようなテーマについて考えるようになったのは、ソフトウェア工学研究会の主催する「オブジェクト指向シンポジウム'97 (OO'97)」を聞きに行ったのがきっかけであった。OO'97は多数(300名以上)の参加者を迎えて行われ、3並列の技術セッションやチュートリアルも多数の参加者の熱気にあふれていた。

もともとオブジェクト指向は、まず(Simula、Smalltalk-80などの)プログラミング言語において導入され、次第にデータベース、設計、分析などの分野に広がって行ったという歴史がある。もちろん、データベースやソフトウェア工学におけるオブジェクト指向の扱いはプログラミング言語におけるものとはだいぶ違っている(分野が違う以上当然である)。しかし、OO'97の発表を聞いた限りでは、そのうちの多くはプログラミング言語ともつながりが深そうに思えた。その典型的なものとして、デザインパターン、コンポーネント、フレームワーク、分散オブジェクトなどが挙げられる。しかしこれらに直接関わる発表がプログラミング研究会で行われることはあまりない、というのが自分の印象である。

それはなぜか、それでよいのか、というのが今回のような討論会テーマを提案したきっかけである。

1　言語研究者にとっては…

しかし一方で、上で挙げたようなテーマがあまりプログラミング言語研究者の感心を引かない、という点にも共感はある。次にその点について述べてみよう。

たとえば、分散オブジェクト技術の代表としてCORBAがあるが、これは非常にそっけなく言えば「(1)位置透明性と(2)相互運用性を重視した(3)RPC」であり、(1)〜(3)のいずれも分散プログラミング言語の世界としてはちっとも新しいことではない。だから、分散言語としてCORBAに沿ったものを提案したとしても、それだけでは論文の1本も書けないのでは、と思うわけである(実際にはそんなことはなく、動かしてみなければ分からない問題は沢山残っているだろうとは思うけれど…)。あと、CORBAの仕様はお金を出して買わないといけない、というのもそういうものにお金を使う経験の(ほとんど)ない言語研究者にとって障壁なのかも知れない。

また、デザインパターンなどはある意味では「いかにうまくプログラムを書くか」という問題なのだが、この種の問題も言語と密接な関わりがあるにも関わらず、言語研究者からは避けられて来たという感じが強い。それはたとえば、「プログラム書法[1]」や「ソフトウェア作法[2]」のような提言は大物にしかできないから若者が手を出すのは不遜だとか(?)、またはスタイルや形についていくら提言しても、はっきり「言語仕様のここが従来の言語と違うんですよ」とは言えないため、論文としては通りにくいということがあるかも知れない。

その観点から見ると、コンポーネントでもフレームワークでもその内容は非常に広く言えば「このように作りましょう」ということであり、言語自体は新しいわけではないから同様、ということかも知れない。また、分散オブジェクトでもスタブジェネレータやトランスレータなどは使うが、言語そのものは最終的にはCやC++やJavaが使われるわけである。

2　言語研究者の感覚…

では、言語研究者にとってこれらのテーマでひと仕事する機会はないのか? というと、そんなことはない。ただ、その現われ方が… 分散透明かつプラットフォーム相互運用かつフォールトトレラントな言語を作りました、とか(まだ自分は知らないが)コンポーネントプログラミングやフレームワークやデザインパターンに適した

[1]kuno@gssm.otsuka.tsukuba.ac.jp

言語XXを作りました、とかなってしまうわけである。

それは、言語研究者にとっては自分の身の周りある論文では1年に何十(ひょっとして何百?)もの言語が作られたり消えたりしている状況では「アタリマエ」であり、別に難しいことではない。

しかし世の中では、C++が世間の認知を得るまでに何年も掛かってここまで来たわけであり、そう簡単に新しい言語など登場してもらっては困るわけである(その意味でJavaがイキナリ登場して2年ほどでメジャーになったのは「トンデモナイ」出来事なわけである)。そこで世の中では、既存の言語(CやC++)はそのままにして、その上で「形」を制約することによって上述のような技術を実現している。それはそれで別に「悪い」ことではないはずである。

3 では歩みよりの道は?

それでは、世の中と言語研究者の間で歩み寄る方法なないのか? しかしその前に、そもそも歩み寄る必要はない、という説もあってよいかも知れない。昔から、多数の新しい言語機構が最初は実験的な言語の一部として現われ、そのうちの特に優れたものが(自然淘汰を生き残っただけで、優れているかどうかは分からないという説も…)他のプログラミング言語に引き継がれて生き残って行く、というのは普通に起きて来たことである※。だから、実験的で新しい言語を作ることに何ら遠慮は要らないとも言える。

ところが、新しい言語でなければいけない、という必然性は実はオブジェクト指向言語の普及によって低下している、というのも見逃せない点である。すなわち、従来の言語では新しい言語機構を入れようとすれば必然的に言語そのものを改造しなければならかなった。しかしオブジェクト指向言語はインタフェースさえ同一に保てば、さまざまな実装を持つオブジェクトが区別なく(実行時に)差し替えて使うことを可能にしている。このため、見ためは従来のコードそのままだが、実は分散オブジェクト、というのも可能になっているわけである。HORBやRMIはいずれもこのアプローチを採用している。これは確かに、言語研究者側からも利用者側からも受け入れやすい「歩み寄り」なのかも知れない。

4 それでもやっぱり言語は言語

ただし、このような歩み寄りが幅を効かせると、今度はこの歩み寄りによって実現可能なアプローチだけが注目され、それ以外のものが注目されなくなる、という可能性もある。それは当然、進歩を阻害するから有害であろう。というわけでとりあえず自分は、上述※のような理論に寄ることにして、「やっぱり目新しいことをするには目新しい言語しかない!」と言いながら新しい(そして誰も使ってはくれないだろう)言語を喜々として設計し実装し続けるのであった。

参考文献

[1] B. W. Kernighan, P. J. Plauger, The Elements of Programming Style (2nd ed.), Bell Labs, 1978. 翻訳: 木村 泉訳, プログラム書法(第2版), 共立, 1981.

[2] B. W. Kernighan, P. J. Plauger, Software Tools, Yourdon, 1976. 翻訳: 木村 泉訳, ソフトウェア作法, 共立, 1980.

オブジェクト指向技術がもたらすパラダイムシフトとデザインパターン[2]

萩本順三[3]
((株)エヌジェーケー
先端技術部オブジェクト指向技術室)

概要

現在ソフトウェアエンジニアリングはオブジェクト指向へとパラダイムシフトしている。しかし、ソフトウェア技術者はその変化に気がついているだろうか。本資料では、パラダイムシフトに向けて技術者の心構えについて私なりの考えを書いてみた。そして最後に、これから必要とされる技術とデザインパターンについて整理する。

Abstract

Now a paradigm of the Software Engineering is shifting toward Object-Oriented. But these such trend changes are not well understood by Software engineer. This paper describes about what exactly the paradigm-shift is and what Software engineers have to learn from such paradigm. And we show how to study technique and Design patterns in such paradigm.

オブジェクト指向は、研究段階から実用段階へと発展をとげてきた。今では、オブジェクト指向技術が、いつのまにかコンピュータソフトウェアのあらる分野に浸透している。OS、言語、フレームワーク、GUIのような技術分野に、技術者の好みに関係なく、まるでウイルスのように静かな広がりを見せている。

このような発展を見せたオブジェクト指向技術は、けして一時的な流行ではない。むしろオブジェクト指向という言葉がもてはやされる時代は過ぎ去り、今では実用的な技術として定着を見せている。今後、四半世紀はオブジェクト指向技術が構造化技術に成り代わりソフトウェア工学のベースとして存在し続けることは間違いないだろう。

このような中でソフトウェア技術者はパラダイムシフトの必要性に迫られている。このパラダイムシフトの背景にある時代の変化を3つの側面から整理してみた。

(1) ロジックからオブジェクトインターフェース重視へ

ハードウェアの進歩に伴い、ロジック中心の設計からオブジェクトインターフェース設計を重視するように設計方法が変わる。オブジェクトインターフェース設計とは、クラスの構造とオブジェクト同士がコミュニケートするためのインタフェースを仕様化することを意味しており、インターフェースとは、クラスの役割に応じて他のクラスに公開するメソッドと考えればよい。

ロジックからオブジェクトインタフェース重視へのシフトによって、効率的な設計が可能となる。このパラダイムシフトは、ロジックばかりを追求する時代から、ソフトウェアの構造をオブジェクトという人にわかりやすい構造の組み合わせとして表現する時代への移り変わりを意味する。よって、ソフトウェア技術者はこの移り変わりをしっかりと受け止め、プログラムの流れや最適化を考える前に、問題領域に対して、変化しにくい本質について構造を考えるような構造重視を癖として身につけるべきである。

(2) プラットフォーム依存から非依存

最近の言語や開発環境では、OSやGUIごとに依存する技術の修得は極端に減少することになる。また、分散オブジェクトやオブジェクト指向データベースのような技術が使われてくると、ネットワークやデータストレージまでもがオブジェクト指向に統合されるようになり個々

[2]Paradigm-shift and design patterns by object-oriented.

[3]Object Technology Group, New Technology Division, NJK CORPORATION.

の詳細な技術を身につける必要がなくなってくる。今までは、プラットフォームごとに特殊な技術が必要とされていたために他の技術者と差別化を図るのは容易であった。しかし、プラットフォームに依存しない環境では、ソフトウェア技術者は何をもって他の技術者と差別化を図ればよいのだろうか。

いままでは、数多くのプラットフォームで大量のコードを書く技術者が重宝されていた。しかし、これからは、オブジェクト指向で統合された開発環境をベースに、分析・設計・実装にオブジェクト指向を使うことで保守性があり、かつ、拡張性のある設計ができる技術者が必要とされ、また、ソフトウェアの構造をオブジェクトモデルを使って他人に解説する能力が問われるという時代がきているのである。このような環境の変化で、ソフトウェア技術者だけではなく、他の専門家たちもソフトウェア開発を行うチャンスが増えることになるだろう。そうなるとソフトウェア技術者は、今とは違った形で自己の存在アピールしなければならなくなる。たとえば、コンサルテイング、分析・設計のスペシャリストなどといったように、今までとは違った専門知識や問題解決能力などが必要とされるようになる。ここにオブジェクトモデルをベースとしたパラダイム転換の必要性を感ずる。ソフトウェア技術者はオブジェクト指向モデル（オブジェクト指向の考え方）を身につけることで、今まで以上に安定した普遍的な技術を身につけることができる。

(3) 記号暗記から構造意味記憶

これは、先に挙げた２つのパラダイムシフトをもっと抽象度をあげたものである。ハードの発展に支えられオブジェクト指向のような技術が使われてくると、記号を暗記するという能力はあまり必要なくなる。なぜなら、オブジェクト指向は、人間にわかりやすい構造をクラスとして表し意味のある名前を付けるからである。また、オブジェクト指向の抽象メカニズムによって、プログラミングの中の語彙が少なくなり暗記すべきものを減少させる。たとえば同じメソッド名を異なるクラスに定義できたり、多態性により具体的なクラスを意識しない抽象プログラミングなどが語彙を少なくする要因となる。

そのかわり、オブジェクト指向では、クラスなどの意味や構造が重視されるようになる。これは、多くのものを暗記する能力から、構造の意味を理解して記憶する能力がより重要性を増すといった、脳の活用法を変える必要性があるということを示している。

数多くのことを憶えるために意味を理解せず暗記するようなやり方では、次の時代を迎えることはできない。時間をかけて構造とその意味を把握しようと心がける。これが今の技術者にもっとも必要とされており、結果的にこのやり方が効率的であることを私は主張したい。

しかし、技術を暗記する能力はすばらしいものをもっていても、自分のものにしている人、つまり本質的な意味を理解している人は少ないように思える。これは、日本における受験戦争の悪影響なのかもしれない。また、忙しさのあまり覚えるより慣れろというノリで、多くの開発をこなしているソフトウェア企業の現状などにも問題があるのかもしれない。一方、欧米人は、一般的に構造に意味づけをして人と議論したり説明したりすることは得意と言われている。これは、ソフトウェア業界だけの問題だけではなく日本人一般の問題かもしれないが、このままでは、これがネックになり日本と欧米のソフトウェア技術レベルの差はますます開いてしまうだろう。

これからの技術者は、この問題を打開しなければならない。暗記に頼らず構造に意味を付けることで記憶する習慣が必要なのである。ソフトウェアの重要な部分を中心に、全て意味のある構造を厳密に定め、議論したり、人に説明したりするように訓練されるべきである。このような訓練には、日々の積み重ねが必要となり、自分で新たな構造をあみ出すといった少しばかりの創造性も必要とされる。さて、このような能力を付けるためにはどういう心がけが必要なのだろうか。ここで一つアドバイスすると、常識や前提を全て疑うことである。オブジェクト指向

の常識やここに私が書いたことなど、全て、白紙に近い状態に戻してしまう。そして、自分の頭で考えることで構造として知識を積み上げてみようとするのがよい。前提となる知識や常識が、自分の頭で考えることを邪魔することが多いのだ。こんなことを言うと、気が遠くなるほど時間がかかるように思えるかもしれない。しかし、今までソフトウェア技術者がやってきた単純な記号処理や暗記は、すべてソフトウェアがやってくれるような時代が到来しようとしているのだ。それは、かつて人間がやっていた単純な計算をコンピュータがやってくれるようになったことと同じであり、そこにソフトウェア技術者のパラダイムシフトの必要性がある。そのような中、ソフトウェア技術者はなにをやるべきかと考えると、やはり人にしかできない創造性豊かな作業が残されることになるのだ。

これから必要とされるもの

以上パラダイムシフトについて3つの側面を掲げてみた。さて、このような時代に必要となる設計技術とはなんだろうか。それは、オブジェクト指向設計の再利用を促進することである。そのためにはデザインパターンの研究は欠かせないものであろう。デザインパターンはオブジェクト指向の再利用を促進するための技術であるコンポーネント技術やフレームワーク技術の設計やソフトウェア開発における様々な設計思想を表現するために使われるだろう。しかし、どのようにすれば、多くのの技術者がデザインパターンを応用できるようになるのか。また、新たなデザインパターンを生み出すことができるのか。このことについてはこれからの課題となろう。

我々は、これらについて、ネットワークアプリケーションを題材に、ネットワークアプリケーションに定石として含まれるデザインを複合的なデザインパターンとして表し、それをメタフレームワークと名付けて研究している。また、デザインパターンとオブジェクト指向方法論の融合、つまり分析モデルから設計モデルへ移行する際にデザインパターンをうまく使う方法を模索している。

参考文献及び引用

- 萩本, 福村, 不破:最新オブジェクト指向技術入門実践: 9 7年 1 1月発刊予定
- 萩本:日経 Java レビュー「Java とのつきあい方」: 9 7年 9月 WEB リリース予定

言語研究者が忘れていたもの

柴山　悦哉[4]
(東京工業大学情報理工学研究科)

1　オブジェクト指向との出会い

私がオブジェクト指向言語の研究をはじめるようになったのは1984年前後のことである。当時，オブジェクト指向という考え方には魅力を感じていたのだが，漠然と疑問に思うことが一つあった。

オブジェクト指向に対する疑問: 「物」に拘りすぎると「関係」や「事」への配慮がおろそかになるのではないか?

当時は「現代思想」がちょっとしたブームになっており，個人的にも，丸山圭三郎流の解釈に基づくソシュール言語学の再評価[1]などに興味を感じていた。そのせいもあって，人間の言語運用能力は「物中心」ではなく「関係中心」にできているのではないかという疑いを感じており，「物中心」のプログラミングパラダイムには所詮限界があるのでないかと思っていた。もちろん，オブジェクト指向言語の中には，A. Borning のThingLab[2] のように，オブジェクト間の関係を制約により表現するものが既に存在していたので，オブジェクト指向言語の可能性自体についてはあまり心配していなかった。

2　オブジェクト指向の本質

その後，1980年代のある時期に突然オブジェクト指向の本質に気がついた。

オブジェクト指向の本質: オブジェクト指向的なプログラムでは，プログラムの設計情報を自然に図示することができる。

当時は設計情報という言葉を思いつかなかったので，「オブジェクト指向はMacDraw で絵を描けるところが良いのですよ」などと，わけのわからん発言を繰り返していたものである。

ともかく，オブジェクト指向のライバルとあの頃考えられていた関数型や論理型のパラダイムでは，設計情報を簡潔な図で表現することが難しく，オブジェクト指向ではやさしかったのである。

3　OOA/OOD との出会い

というようなわけで,「物」と「関係」と「図」の重要性を感じつつ，なかなかオブジェクト指向プログラミング言語の研究には結びつけることができずにいた。そんな時，オブジェクト指向分析／設計（OOA/OOD）というものの存在を知り，そこで使われるオブジェクト図を見て驚いた。今まで漠然と考えていた「物」と「関係」が「図」として表現されているではないか。

考えてみれば，Entity-Relationship モデルくらい昔から知っていたのに，データベースのスキーマ設計技法だという思い込みが強すぎた。オブジェクト指向言語が対象とするのは，動的な傾向が強いプログラムであり，静的なものではないというイメージも災いした。オブジェクト指向の本質が設計図を図示できる点にあるのなら，むしろ静的な部分にこそ注目すべきではなかったか!!

今にして思えば，Entity-Relationship モデルに若干の修正を加えれることにより，プログラムの設計情報の重要な部分が表現できるなんて，当たり前であった。こういう問題に気づかなかったのが私だけなら良いのだが，言語コミュニティ全体でも気づいていた人はあまりいなかったのではなかろうか? そう思うと，ちょっと暗い気持ちになってしまう。

4　近頃巷では

その後もいろいろあったが，そろそろ現在の話をしよう。近頃，デザインパターンカタログやソフトウェアアーキテクチャが世間で注目を集めている。ソフトウェアを構成する際に，ソフ

[4]etsuya@is.titech.ac.jp

トウェア部品を柔軟に組み合わせるためのノウハウが重要なのは当然であり，デザインパターンカタログやソフトウェアアーキテクチャの重要性は認めざるを得ないだろう。

しかし，これらはもはや博物学の世界であり，博物学的な知見を集めることこそが重要で，その表現言語はあまり重要でないような気がする。言語研究者の出番はあまりないのではなかろうか? もちろん，世の中にはソフトウェアアーキテクチャ記述言語([3]の45ページの囲み記事参照)なんて研究分野も存在するので，言語研究者と全く無関係というわけでもないのだが。

5 言語研究者の逆襲

昔から漠然と思っていた「物」と「関係」を「図」で表現するという課題に対する答として，最近，ビジュアルプログラミング言語 KLIEG を作り始めている[4, 5]。ついでに，デザインパターンやソフトウェアアーキテクチャの表現と支援までを含むシステムにしてしまった。図 1はこの KLIEG 言語で表現した並列処理のパターンの一例で，generator が生成した部分問題を，複数の worker にディスパッチし，並列に問題を解くためのパターンである。この例では，generator と worker がいわゆるホットスポットになっており，自由に部品の取り替えができる。

「デザインパターンカタログはあくまでドキュメントである」という主張があるのは知っているが，言語・環境研究者のこだわりにしたがい，あくまでマシンサポートを目指すことにした。残念ながら，デザインパターンを支援するための凝った機能やビジュアルプログラミング言語でいつも問題となるスケーラビリティの問題を解消するための機能の説明は本稿の範囲を超えている。詳しくは[4, 5]を参照していただきたい。

まあ，ともかく，戦いはこれからだ!!

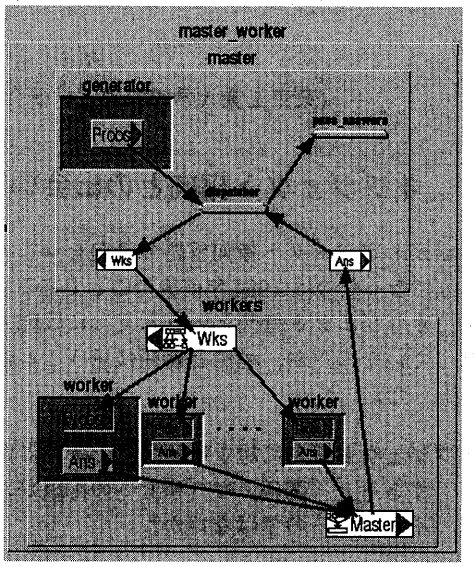

図 1: KLIEG のパターン

参考文献

[1] 丸山 圭三郎, ソシュールの思想, 岩波書店, 1981.

[2] A. Borning, The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory, *ACM TOPLAS*, Vol. 3, No. 4, pp. 353–387, 1981.

[3] R. T. Monroe, A. Kompanek, R. Melton, and D. Garlan, Architectural Styles, Design Patterns, and Objects, *IEEE Software*, Vol. 14, No. 1, pp. 43–52, 1997.

[4] M. Toyoda, B. Shizuki, S. Takahashi, S. Matsuoka, E. Shibayama, Supporting Design Patterns in a Visual Parallel Data-flow Programming Environment, Proc. of IEEE Symposium on Visual Languages, 1997 (to appear).

[5] 志築 文太郎, 豊田 正史, 高橋 伸, 柴山 悦哉, ビジュアル並列プログラミング環境 KLIEG: プロセスネットワークパターンを利用した再利用性の向上と実行表示の効率化, *WISS'96*論文集, 近代科学社, pp. 81–90, 1996.

コンポーネントプログラミング

酒匂　寛[5]
(Designers' Den)

0　論争の始まり

ソフトウェアコンポーネントという言葉はあまりに一般的過ぎるが、多くの現場のソフトウェア技術者が、否応なしにコンポーネント技術とやらに向かい合わされている。いささか旧聞に属するが1995年に米国 Byte が「オブジェクト指向プログラミングは失敗した。これからはコンポーネントウェアだ」という特集を組んだことを憶えている方もいることだろう。

思い返せば 1980 年代初頭に Smalltalk を大々的に取り上げて、世にオブジェクト指向プログラミングありきを喧伝したのも同じByte であったことを考えると、商売とは言え編集者の苦労が忍ばれるエピソードではある。

しかし、ここで目の敵にされたオブジェクト指向プログラミングとは一体何だったのだろうか。それはコンポーネントプログラミングとは無縁のものなのか。考えなくとも明らかなことであるが、コンポーネントとはオブジェクトの別名に過ぎない。では論点は一体何だったのであろう。

筆者の見るところ、世間におけるこの論争はオブジェクト指向におけるクラスベースのプログラミングと、インスタンスベースのプログラミングの違いから引き起こされたものであると思っている（この両者の違いは全く本質的なものではないのだが、ともあれ商業的には全く違うものとして捉えられている)。

クラスベースのプログラミングはインスタンシエートされていない対象を扱うのだから、実行してクラスを具体的なオブジェクトとして実体化させてみるまでその効果を見ることはできないが、インスタンスベースのプログラミングはそれ自身「実行」可能な実体を組み合わせることで行っていくので、自分が意図した通りの振る舞いを確認しながら構成しやすいという訳である。

例えば Window を一つ出して "Goodbye, World" という表示を行おうとする場合、インスタンスベースのプログラミングなら、Window のインスタンスを一つ引っ張り出してきて、その上にしかるべき文字列が表示された Label のインスタンスを貼り込めば一丁上がりという訳である。

もちろんここで述べたような GUI の構築だけがプログラミングの要素ではない。しかし不幸なことに単純な GUI を表示するだけでも大変という貧弱なプログラミング環境が「主要な」プラットフォーム上で提供されていたために、この部分を劇的に簡易にしてくれるというコンポーネントプログラミングがより注目されたというのは皮肉なことである。

以後 2 層モデルから 3 層モデルというキイワードと共に、コンポーネントプログラミング環境はその勢力を伸ばすことになった。

残念なことにコンポーネントプログラミングの優位性は、こうした「簡易プログラミング」を巡る皮相的な議論で行われている場合が多いのだが、もっとプログラミング作業そのものに根ざす議論が行われるべきではないかと思う。

1　プログラミングを支えるもの

プログラミングを行う者は何から支援を受けているだろうか。よく言われることではあるが、言語、資産、環境の3種の神器がプログラミングを行う上で大切な要素である。

言語は何らかの「記述」を行わせるものである。その意味でそれが文字で表されるものであろうと、視覚的に表されるものであろうと、計算機に対して記述を行うという行為が存在する以上言語自身はなくなることがない。頭の中で考えていることを勝手に吸い出してシステムを自己構成する仕掛けでもない限り、この状況は変化することがない。

資産はこのようにして過去に記述したものを

[5]sakoh@ba2.so-net.or.jp

貯えたものである。旧来のクラスベースのオブジェクト指向環境ではクラスだけが資産として捉えられていたが、そこから生成するインスタンスの属性も資産として捉えることができる。もしくは実際のデータを抱え込んだインスタンスも資産として考えることができる（辞書を表すオブジェクトを想像して見るがよい）。このレベルで支援を行うコンポーネントプログラミング環境は多い。一方デザインパターンというキイワードも最近頻繁にささやかれるようになった。これらもまた資産として考えられるが、クラスレベルのデザインパターンだけではなく、インスタンスレベルのデザインパターンも同様に考えることができる。こちらのレベルで支援が考えられたコンポーネント環境は残念ながら数は多くない。

環境はプログラマ（＝記述を行う者）が、資産を利用しつつ記述を行う場所である。新しく定義したクラス、そこから派生したインスタンス、アルゴリズムを記述したテキスト等全てが資産となり得る。理想のプログラミング環境は、資産を自在に引き出すことが可能で、それらの組み合わせを行うためのワークショップを提供し、制作中のオブジェクトを「手に持って」様々な角度から眺めたり、動かしたりすることを許し、成果物を再び資産として蓄積することを許す環境でなければならない。

世の中に流布する多くのコンポーネントプログラミング環境は、インスタンス資産の引き出しとインスタンスベースの記述を支援するところに主眼が置かれているようである。しかし上記の意味でまだまだ理想からは遠い。

2 言語に何を求めるか

前節に述べたようなプログラミングの3大支援要素を考える以上、言語そのものだけを取り上げて議論するのはもはや無理がある（それでも言語レベルでの Polymorphic な語彙の支援等は必須だと思うが）。

コンポーネントプログラミング環境は、言語としては既存のものに大きく優るものではない。その持つ優位性は資産管理と環境をプログラミング作業に統合しようとしている点にある。しかし過剰に資産化や環境への統合を謳うと、面白い記述ができるような方法がつまはじきされてしまう。

一人で小規模なプログラミングを行うなら言語は何でも良いだろう。しかしある程度以上の規模のシステムを、複数人で構築するためには、資産（すなわち自分を含む他人との協調）と環境（記述の動的な追跡）とをうまく持ち込む必要があるように思える。

ここではあまり深く触れないが、要求定義との関連も忘れることはできない。乱暴な言い方をするならば、要求はインスタンスレベルで記述されている。要求から実装へ向かうシームレスな開発環境が求められているとするのなら、インスタンスベースのプログラミング環境は一歩優位性を持っていることになる（UseCase を素直に表現する手段として用いることも可能である）。

コンポーネントプログラミング環境はこうした様々な統合や、開発方法論への一つの解決手段（あるいは提案）である。

参考文献

[1] Byte Ed., ComponentWare, Byte, 1995 May.

ソフトウェア文化をどう創るか

加藤 和彦[6]
(筑波大学 電子・情報工学系)

1 ソフトウェアと類似する文化

1.1 小説と映画

ソフトウェア作りに携わったことのない人達にソフトウェア作りの難しさを伝えるために，私は小説と映画をよく喩えとして用いる．

素人とプロフェッショナル 文字を書くことは誰にでも書くことはできるが，人が買ってくれるような小説を書くのは大変に難しい．巷に広く普及しているビデオカメラを使えば，映像を撮ることはひどく簡単だが，鑑賞に耐えるものを作るのは容易でない．ソフトウェアも，プログラムを書くことは，ちょっと学べばできるようになるが，人の役に立つようなプログラムを作ることは容易なことではない．

再利用の難しさ 小説でも映画でも同じような場面はしばしば現れる．しかし，既に作った場面を，そのままはめ込んで使うような再利用の仕方はほとんど困難である．同じような場面も，毎回，書き下ろし・撮り下ろしをせねばならない．これはソフトウェアの現況とよく似ている．

系統的な作者養成の難しさ 創造的な小説家を教育等によって系統的に養成することは大変に難しい．創造的な映画監督，脚本家，俳優を養成することも同様である．そしてソフトウェアの作者の養成もそうである．

1.2 建築に学ぶ

ソフトウェア作りが，小説や映画作りとよく似ているということの要は，ソフトウェア作りがアートの側面を持っているためと考えられる．ソフトウェアがアート的な側面を持っていることは否定されるべきものではない．なぜなら，ソフトウェア設計の根本的な目標は，人間の仕事・活動を効果的にサポートすることであり，どうすればそのようなことができるのかを考えることは，大いなる創造性を必要とするからだ．どうすれば創造的になれるかという問題は，ソフトウェア分野に限らない大きな問題である．ここでは仮に創造的な設計が行えた場合のことを考えよう．現在のソフトウェア文化では，創造的な設計が行われた後に，ソフトウェアを系統的に実現するような方法論は未だ確立していない．

建築は，創造的な設計が行われた後，それを系統的に実現する術をはるか昔から確立することに成功しているのではないかと思われる分野である．建築物は，建築家が創造的に設計したものを，多くの関係者が従事して実現に当たる．そして，建築家が頭の中に抱いた「想像の産物」を現実化する．大きな建築物となると，極めて多くの関係者が実現に従事することになるが，破綻を来さない[7]．これは，ソフトウェア分野の人間から見ると，極めて不思議に思える．

建築の分野からソフトウェア分野の人間が学ぶことができることは数多いのではないかと私は予測している．建築の分野でうまくいっていると私が感心することの一つは，個々の部品の規格化と，部品を組み上げたときの統制をうまく両立させていることである．言い換えると，建築物は巨視(マクロ)的な観点から見るとオリジナリティに富んだ唯一無二のように見えるものも，微視(ミクロ)的に見ると，多くの場合，規格化された（すなわち既存設計の）部品を組み合わせてあるだけである．ソフトウェアの場合，部品の再利用性の重要性はよく叫ばれ，よく理解されているが，未だ再利用することは難しいことであり，どうすれば再利用可能になるのかを我々はまだ十分には理解していない．この点に関連して私が最近注目しているのは，ソフトウェア産業界で普及が始まっているコンポーネント・ソフトウェア技術である．コンポーネン

[6]kato@is.tsukuba.ac.jp

[7]破綻を来したものは人知れず葬り去られているだけなのかもしれない．

ト技術の何が本質的なものなのか，従来のオブジェクト指向技術とはどこが違うのか，アカデミックな立場からの解明は十分になされていないが，Visual BasicやDelphiなどの実際の処理系を使ってみると，建築における規格化された部品の操作とはこういうものではないかという印象を抱かされる．

2 「骨太」な研究開発

おそらく現在は，これまでの，そう長くはないコンピュータ文化の歴史の中で，計算機文化が最も花咲いた時代であろう．日本全国津々浦々，電気店・電化製品売場には，10年前には垂涎の的だった高性能CPUと大容量主記憶を備えたパソコンが一般庶民がちょっと手を伸ばせば手の届く価格で並んでいる．書店に行けば，コンピュータ関連の単行本が所狭しと平積みにされ，コンピュータ関連雑誌も華やかな表紙を飾っている．書店の書棚の中で，コンピュータ関連の本・雑誌が占める割合の増加は目覚ましい．筆者が勤務する大学内の書籍部では，店舗の面積は同じながら，コンピュータ関連の本が占める割合は10年前に比べて3倍になった．しかし悲しきことは，その増えた分は，舶来製ソフトウェアの解説本が多数を占めていることである．そしてその解説本自体も，舶来製の本の翻訳である割合が高い．コンピュータ文化が，今，花を咲かせるといっても，日本におけるそれは徒花（あだばな）であるのかもしれない．

日本は，極東の島国という地理的な特質のためか，知的エリートが異国の文化の輸入・紹介を担うという文化を歴史的に持っている．聖徳太子の時代も，明治の時代も，そして昭和・平成の現代もそうである．最初は輸入で得た文化も，国の中で独自に育み，世界から憧憬の眼差しで見られるようになったものも少なからずある．さて，コンピュータ文化についてはどうだろうか．ハードウェア技術に関しては，日本の技術が世界のそれを支えているといって過言でないものがいくつもある．しかし，実用ソフトウェア技術に関しては，日本語処理あるいは多カ国語処理を別とすれば，日本が世界を支えているといえる技術は，残念ながら，容易には見つけることができない．

美術や音楽の分野で，「コピーは芸術のはじまり」という言い回しがある．ここで言う「コピー」とは，そっくりの複製物を作るというよりも，作者の行為をなぞることによって，その芸術の心あるいは核心を学びとり，その後の創造行為を行っていくための糧とすることを意味している．ソフトウェアも，他国で創造された作品を輸入しながらなぞることにより，その後の創造行為につなげていけるならば素晴らしい．しかし，とどまることなく，目まぐるしい速度でソフトウェア文化が進化を遂げ続けている現在では，次々と繰り出される新しい創造物を「コピー」し続けるだけで，あっという間に時は通り過ぎてしまう．舶来文化の「コピー」ももちろん謙虚な態度で続ける価値のあるものであるが，舶来の動向に左右されない，「骨太」な研究開発を意識して指向することも必要ではないだろうか．

3 正念場の時代

私が大学の情報学科に入学した，今から16年程前は，計算機という機械と計算機科学という学問分野は，右も左もわからない若者にとって神秘性のようなものを秘めていた．私が幼い頃，子供向けの番組やニュース番組でイメージ的に映し出される計算機の姿は，大型計算機システムの磁気テープシステムが回るところや，磁気テープが吐き出されるところだった．子供心に，どういう仕掛けであれで計算というものをやっているか不思議だった．あれは実は入出力装置と呼ばれるものに過ぎないことを知ったのは高校生になってからだと思う．計算機が描いた絵を見たり，計算機が作った楽曲を聞いたとき，どうやって機械にそんなことをさせることができるのか神秘的なものを感じた．実は，計算機が創造行為を行っているのではなく，創造的なのは人間であり，計算機はその創造的な人間が使う道具に過ぎないことを理解したのは大学に入ってからである．

今や計算機は日常に溢れている。今や一家に一台はあるであろう、ゲームコンピュータ、ワープロ、パソコンが計算機を日常的なものにした。それらの上で動作するソフトウェアも高度なユーザインターフェースを備えている。きれいな音も動画も出る。インターネットにもつながっている。今や計算機はテレビのようなものになりつつある。その存在は日常の風景に溶け込み、その中身の原理はわからなくとも、それはそれとして自然に受け入れられるようになってきた。もはやその存在に神秘性を感じることは難しい．もはや，計算機科学の分野を若い人にとって魅力的に見せることは容易なことではではなくなった．このような時代に，若い人を引きつけるようなソフトウェアの技術・文化の発信ができるか，今は一つの正念場の時代ではないかと思う．