連載解説

パターン

―ソフトウェア開発ノウハウの再利用

第5回　パターン指向開発とパターンの今後

中山裕子（株）富士通研究所
細谷竜一（株）東芝　　山本里枝子（株）富士通研究所
吉田裕之（株）富士通　　吉田和樹（株）東芝

パターンを解説する本連載も，ついに今回が最終回である．第1回から第4回にわたり，パターンの歴史や特徴，代表的な既存のパターン集について述べ，その中でも特にデザインパターン[1]とアナリシスパターン[2), 3)]を取り上げて解説してきた．今回は，既存のパターンを再利用するだけでなく，個々の開発プロジェクトが積極的に独自のパターンを発見，蓄積し，それらを活用して開発を効率化する手法を紹介する．そして最後に，パターンに残されている課題と，今後の展開について述べる．

パターン指向開発

パターンは，ソフトウェア開発の中で繰り返し起こる問題と，それに対して皆が繰り返し使ってきた解法を記述して，再利用するものである．前回まで解説してきたように，すでに多数のパターンが発表されており，また「デザインパターン」[1]など既存のパターンを適用した事例も数多く報告されている．

しかし残念なことに，これら既存のパターンによって生産性が飛躍的に向上した，という報告はまだない．その1つの理由は，既存のパターンだけに頼っているからである．年々パターンの数が増えているとはいえ，個々のプロジェクトについて見れば，既存のパターンでカバーできる部分はまだまだ少ない．これに対してパターン指向開発では，既存のパターンだけでなく，より積極的に各プロジェクト独自のパターンを見つけて再利用することを提唱している[4), 5)]．

パターン指向開発では，パターンのそもそもの効用を「1人の開発者が考えた結果を，分かりやすい形で明文化しておけば，他の開発者が同じことを考える必要がなく，再利用できる」と捉えている．そして，各開発プロジェクト独自の開発ノウハウをパターン化・再利用する活動を徹底し，パターンを中核に開発作業を進めていく．

この章では，パターン指向開発の概要を解説し，この手法をあるパッケージ製品開発プロジェクトへ適用して生産性を飛躍的に向上できた事例を紹介する．この事例では，ソースコードの70%がプロジェクト固有のパターンに従ったものになった．

プロジェクト固有パターンの蓄積と再利用

パターン指向開発では，

1. プロジェクト内でノウハウを蓄積，再利用する．
2. アーキテクチャの一貫性を保つ．

という2つの目的のためにパターンを用いる．個々のプロジェクトが独自に蓄積するパターンを，プロジェクト固有パターンと呼ぶ．

1点目の目的のためには，プロジェクト固有パターンを一般のパターンよりも手軽に記述できるものにする．つまり，プロジェクト内ではパターンが解決する課題や背景となっている状況など詳細な情報は共有できて

いるので，一般のパターンのような厳密な記述は必ずしも必要としない．プロジェクト内部で理解できる記述で十分である．また，対象とする課題が特に難しいものでなくても，あるいはプロジェクト特有の一般性のないものでも構わない．解法の一般性もプロジェクト内で再利用可能なレベルにとどめる．大切なのは，誰かが悩んで解決した問題はもう二度と悩まないようにすることである．

もちろん既存のパターンも積極的に再利用するが，まず「コアチーム」(後述する) が必要なパターンだけ選択し，プロジェクト内での適用例を示しておくことが重要である．納期に追われる現場では，開発者全員に既存のパターン集をすべて読破してもらうのはまず不可能である．しかも，1つのプロジェクトで適用できる既存パターンは，パターンの総数から見ればごく少数なので，効率も悪い．

2点目の目的のためには，プロジェクト固有パターンを規約として利用する．同様の課題には必ず同じパターンを用いて解決するよう，プロジェクト内部に周知徹底する．

こうして，システムの各部での設計・実装上の判断のばらつきが一掃され，システム全体のアーキテクチャの一貫性が保たれるので，後の変更や機能追加，保守が容易になる．またオブジェクト指向の未経験者が多いプロジェクトでは，品質を一定に保つと同時に教育的な効果もある．規約として強い強制力を持つ点が，プロジェクト固有パターンと一般のパターンとの違いである．

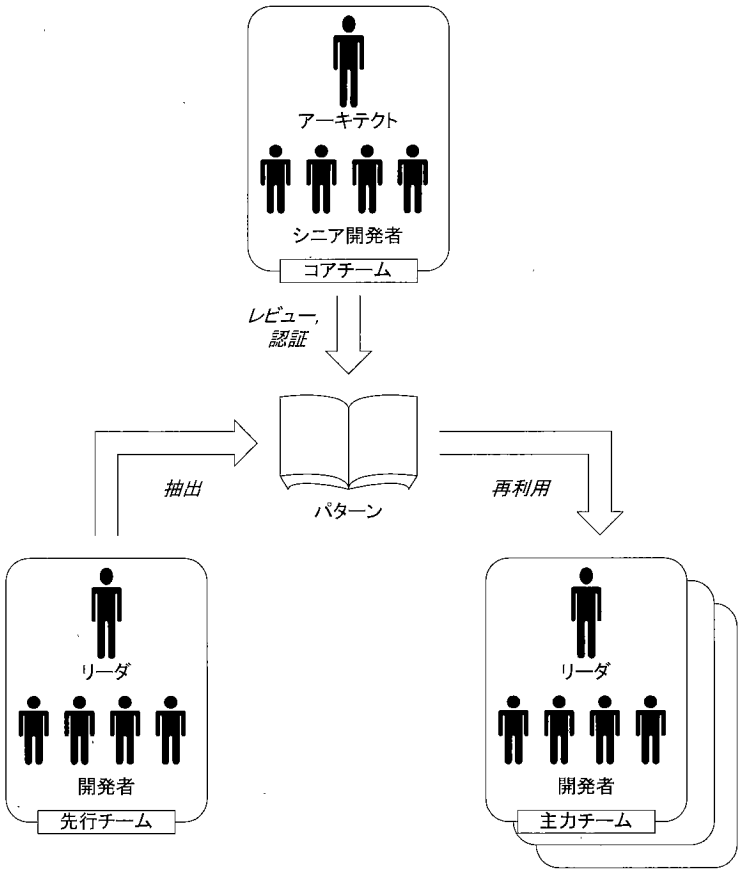

図-1　パターン指向開発における役割分担

開発チーム体制の整備

パターンを蓄積し周知徹底するには，体制作りが不可欠である．パターン指向開発では，**図-1** のようにチームを分けて役割分担することを提唱している．

まず，パターンを見つけるのは，他のチームに先駆けて開発作業を行う「先行チーム」の役割である．次に，「コアチーム」がパターンをレビュー，認証し，主力チームに周知徹底する．開発の主力部隊である「主力チーム」は，先行チームが蓄積したパターンを再利用しながら開発作業を進める．コアチームは，主力チームの作業結果をレビューして，パターンが正しく使われているかをチェックし，プロジェクト全体でアーキテクチャを一貫させる．

この流れの中で，少人数の先行チームはいわば偵察部隊の役割を果たす．この「偵察活動」によって，残りの大多数の主力チームは大きな問題にぶつかることなく開発作業を進めることができる．また異なる主力チームが同じ問題を繰り返し悩むといったこともなくなり，生産性を向上できる．ただし，先行チームがすべてのパターンを見つけることはできないので，たとえば8割程度を目標にし，主力チームが開発作業の中から逐次見つけていく．

パターン指向開発で技術面のリーダシップをとり，アーキテクチャ全体に責任を持つのは，コアチームである．コアチームは，技術をとりまとめるアーキテクトと，シニア開発者数人で構成する．コアチームには，プロジェクトの中で最も優秀な人材を集めなければならない．したがって，先行チームや主力チームのリーダやサブリーダが，コアチームを兼任する場合もあるだろう．兼任することによって，コアチームの決定事項の周知徹底も効果的に行える．

開発プロジェクト全体では，以上の役割の他に，プロジェクトマネージャと品質保証チームなどが必要になる．チーム体制の詳細については，文献4) で詳しく述べている．

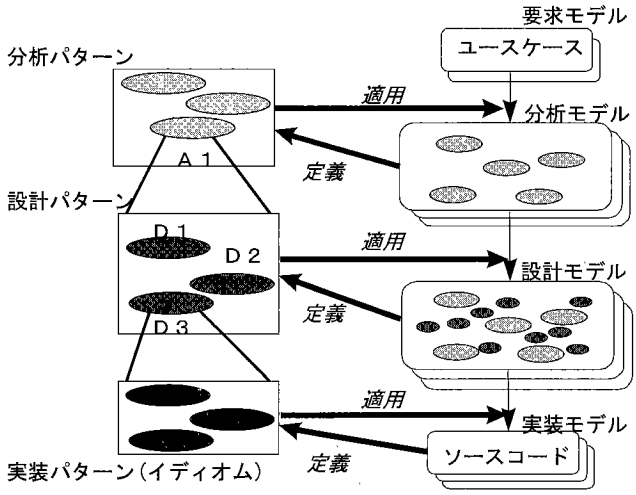

図-2　詳細化によるパターン体系

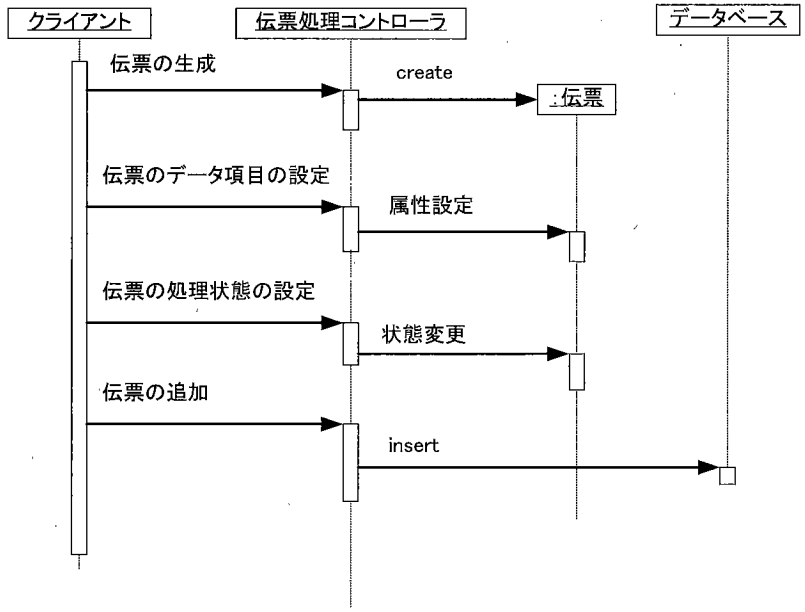

図-3　分析パターンの例

詳細化によるパターン体系

一般のパターンと同様，プロジェクト固有パターンでもパターン間の関連を示すことが重要である．中でも特に，分析・設計・実装パターンの間の一貫した「流れ」を追跡可能にしておくと，全体の見通しがよくなる[4]．この流れは，分析から設計，設計から実装への詳細化の関係である．

たとえば，**図-2**で分析パターン“A1”を適用すると，**図-3**のように何をオブジェクトにするか，またオブジェクト間でどのようなインタラクションをするか決まる．またこの結果，次にどの設計パターン“D1”～“D3”を適用すべきか（図-3の例では，伝票をデータベースへの追加する際のいくつかのパターン）が決まる．したがって設計パターン“D1”～“D3”は，分析パターン“A1”と関連付けて定義できる．また詳細化の過程で何を考慮し，それに関してどのような選択肢があるかを体系的に表すことができる[5]．

適用事例

パターン指向開発を，あるパッケージ製品開発プロジェクトに適用した例を報告する．このパッケージは，受発注などの伝票処理を中心とする販売管理システムである．

このプロジェクトの開発チーム体制であるが，コアチームはアーキテクトを中心にオブジェクト指向の専門家数名で構成した．先行チームと主力チームのメンバは，販売管理分野での経験は豊富だが，オブジェクト指向分析・設計とC++言語の経験がいずれも1年未満だった．

この体制の中で，多くのパターンが先行チームの作業から生まれた．先行チームはパターンの抽出に役立っただけでなく，開発ツールの使い方やその他の目に見えないノウハウをチーム内に普及することに大きく貢献した．オブジェクト指向の初心者ばかりで編成されたチームではあったが，パターンと先行チームの存在が，即戦力養成の決め手となった．

また，実装作業の一部を外部に発注したが，設計仕様書とともに設計パターンと実装パターンを開示することで，ソースコードの品質を内部のチームが開発したものと同等に保つことができた．

このプロジェクトでは，既存のパターンカタログから再利用したパターン以外に，プロジェクト独自のパターンとして，

- 分析パターン：15種類
- 設計パターン：20種類
- 実装パターン：20種類

を蓄積・再利用した．ここでの分析パターンは，FowlerやHayのアナリシスパターン[2), 3)]とは異なり，特定の業務処理方式をテーマとし，主にシーケンス図を使って記述している．分析パターンの例を，図-3に示す．

図-2「詳細化によるパターン体系」に示したように，分析パターンを詳細化して設計パターンを，さらに設計パターンを詳細化して実装パター

ンを定義している．パターン全体を網羅的に収集し，体系化した結果，開発の「流れ」が見通せるようになった．

1999年1月時点での報告によると，設計パターンにしたがって開発されたメソッドは，メソッド全体の実に97%に上った．さらにメソッド全体の83%は，パターン自動適用ツールによって，クラス図に自動追加できた．実装については，ソースコードの70%がパターンにしたがったものになった．それ以外の30%のうち15%弱も実装パターンにしたがった単純作業だった．残りの15%強は，主に業務個別の処理（業務ロジックとも呼ばれる）部分である．つまり，開発者は同様なプログラミングを繰り返す単純作業から解放され，難しい業務ロジックの部分に集中できたことになる．

このきわめて高いパターン適用率を得られた理由は，対象システムそのものがパターン指向開発に向いていたことにある．対象システムのほとんどの部分が，伝票の新規起票，更新，照会などビジネスアプリケーションに典型的な処理であった．どの伝票も処理の「型」は同様で，設計上，メソッドのインタラクションがほぼ同じになる．違いは，各メソッド内部でのデータのチェックロジックなど細かな部分だけである．

まとめ

このようにパターン指向開発を実践し，分析から実装までのすべての作業についてパターンの蓄積と再利用を徹底したことで，開発者による判断のバラツキが一掃できた．開発中には，パターンによって規定していない部分に集中してレビューすることで，問題を早期に発見できた．また，保守・拡張についても，各部でクラス構造と振舞いが整然と整っているので，一貫した対策がとれて作業効率がよいことが期待できる．

本連載では，初回からここまでの間に，さまざまなパターンを紹介し，またパターンを実践的に活用する方法を説明してきた．次章では，連載全体のまとめとして，パターンに関する課題と今後の展開について議論する．

今後の展開

パターンの二極化：汎用パターンとドメイン特化パターン

現在でもさまざまなパターンが，本，雑誌，ウェブ，ワークショップ，会議などの場で書かれ，発表され続けている．パターン執筆者☆1のワークショップであるPLoP（Pattern Languages of Programs）[6]では，毎年多くのパターンが提出されレビューされる．過去のPLoPに提出されたパターンのいくつかは，PLoPDシリーズ[7]～[10]に収録されてすでに出版されている．このシリーズを見てみると，とにかく何でもパターンの形式で書けることが分かる．この一連のパターン集は，本連載でも紹介したアーキテクチャパターン，設計パターン，コーディングパターンといったものだけではなく，「ソフトウェア開発工程のパターン」「コードを普及させる戦略のパターン」「顧客との関係のパターン」「パターンを書くためのパターン」などといったものまで掲載しており，ソフトウェアに関係していることを唯一の制約としながらパターンの可能性を探ろうとするPLoPの姿勢がうかがえる．

パターンは大きく分けると，分野を問わずさまざまなソフトウェア開発で用いる汎用パターンと，特定分野（ドメイン）☆2向けのドメイン特化パターンの2種類がある．汎用パターンを集めた本では，再利用設計のパターン集である「デザインパターン」[1]が特に有名である．ここに収められた設計パターンは，ソフトウェアの部分的な変更をしやすくしたり，オブジェクト指向言語を使った設計およびコードの再利用技術として注目されるアプリケーションフレームワークを構築するのに威力を発揮する．汎用パターンは素朴で，すでに多くの開発者が当たり前のように使っていたり，それが提示する問題に一度は直面したことがあるようなものである．ドメイン特化のパターン集には，たとえば製造業向けのデータモデルパターンを多く集めた「Data Model Patterns」[3]や，金融や医療などの分析モデルパターン集の「アナリシスパターン」[2]などがある（本連載第4回）．ドメイン特化パターンはより複雑で，その文脈を理解するためには適用対象となるドメインの知識を必要とする．

汎用パターンは通常，1つ1つを独立して適用するものなので，少しずつ適用しながら既存のソフトウェアを段階的に改良していくような使い方ができる（本連載第2，3回）．一方，ドメイン特化パターンは，込み入った問題から始まって，より複雑で大きい解決策を提示する．そこから導き出される成果物は具体的で，何をするためのものかがはっきりしている．たとえば，「アナリシスパターン」[2]が提示するパターンは，金融オプション取引の分析モデル，業務計画の分析モデル，企業の組織構造の分析モデルなどといった具合に，システムのユーザにとって意味のある機能を形作るための分析モデルのパターンを提示している．

汎用設計パターンの世界では，「デ

☆1 パターンを報告する人は「パターン開発者」などとは呼ばずに「パターン執筆者」と呼ぶ．なぜなら，個々のパターンはその発表者が発明したものではなく，その人が既存のアイデアをとりまとめてパターンの形式で書いたものだからだ．

☆2 ここでは，システムのアーキテクチャ，動作，運用方法などの特徴で分類する技術指向ドメインと，ユーザの業種・業務に着目したビジネス指向ドメインの両者を指す．前者ではリアルタイムシステム，通信システム，ウェブベースシステムなどのように分類する．後者には金融，医療，製造などがある．

ザインパターン」[1] を軸として，これを補うような新たなパターンや，この中の特定のパターンを改良したり発展させたものなどが雑誌，本，PLoP などで時折発表されている．このパターン集の多くの開発者に愛用されている様子が分かる．こうした動きを見る限り，汎用パターンはより大きな集団の中での使用，検証を経ながら，地に足のついた格好でゆっくりと増えていくように思える．それに対して現在，ドメイン特化パターンはいろいろなドメインから涌き出るようにして増えつつある．しかしそれらは特定の状況における問題に特化しているぶん，開発者が自分の問題に合わせてカスタマイズして使うことは容易ではない．そのうえ，それが1人の思い付きではない真のパターンなのかどうかを評価することが難しい．ドメイン特化パターンは，このまま放っておけば，きちんと評価，体系化されないまま漫然と増えていく危険がある．

パターンとパターン言語

上で述べたドメイン特化パターンを取り巻く状況を改善するキーワードは「パターン言語」である．ドメイン特化パターンは，取り扱う問題と，どんな成果物が導き出されるかが具体的である．また，成果物を導く過程では，1つだけでなく複数のパターンが組み合わされることが多い．そのため，適用の文脈から始まって，最終的な成果物に至るまでの一連のパターン適用の流れをルール化しやすい．つまり言語化されたパターンである（本連載第1回）．たとえば，第2，3回で見た「会議室予約エンジン試験フレームワーク」の構築の過程を，「試験」というドメインに特化した一連のパターン適用と見立ててみよう．この過程におけるパターン適用の文脈は「一連の試験項目を効率的に追加し，自動実行する」という問題であり，その成果物は「試験項目を，呼び出しシーケンス，配置形態等の要素に分けて実装し，実行するフレームワーク」である．この過程で，我々は問題の分析から始まって *Abstract Factory* と *Singleton* の2つのパターン[1] を適用するまでの流れを追った．また，試験項目数が膨大なときは *Prototype* パターン[1] を組み合わせたり，合否判定を自動化する場合には *Decorator* パターン[1] を適用するなど，状況によってどのようなパターンを追加適用すべきかを述べた．これら一連のパターンの適用の流れをルールの形にまとめれば「試験フレームワーク構築パターン言語」が得られるのである．ドメイン特化パターンは解決すべき問題，最終的に得られる成果物，そしてそこに至る過程を小さなパターンの適用の流れとして言語化することで，その根拠と，利用者の問題に合わせてカスタマイズする手段を提供することができるようになる．また，そのパターンの評価もしやすくなる．

実際，ドメイン特化パターンは，1つ1つ独立した形で発表されるだけではなく，数個から数十個程度をセットにした小さなパターン言語として発表されることが多い．今後ドメイン特化パターンではパターン言語の創造と改良が進むだろう．これによって，単体ではそれが的を得たものかどうか評価することや，自分の問題に合わせてカスタマイズすることが難しいドメイン特化パターンを，扱いやすく，効果的なものにすることができる．

前述の通り，汎用パターンはドメインを問わずソフトウェアの部分部分に1つずつ独立して適用するものである．また，ドメイン特化パターンと違い，汎用パターンが単独で，ユーザの目に触れるソフトウェアの機能を形作るわけでもない．こうしたことから，汎用パターンは言語化されないのが普通である．しかしこれらは，こつを得るまでは，いつ，どのパターンを適用すればいいか分かりづらい．また慣れていても，どこでどのパターンを使うかをあらかじめ決めておくことはほとんどできないので，開発工程の中で，時々リファクタリング☆3しては次に進むといった反復作業を繰り返す必要がある．

とはいえ，「会議室予約エンジン試験フレームワーク」の例で見たように，1つの汎用パターンの適用が次に適用すべき汎用パターンを自然に決めることは実際にある．したがって，数個程度の汎用パターンを組み合わせて小規模の問題を解決する過程をルール化し，小さな汎用パターン言語を作ることは可能と思われる．それは，たとえば次のようなものである：（プログラム中で，いくつかの関連するオブジェクトの生成に使うクラスの決定を1個所に封じ込めたいとき）

1. *Abstract Factory* パターン[1] を適用する．
2. 1で導入される *ConcreteFactory* をプログラム中の複数の個所で呼び出す場合は，さらに *Singleton* パターン[1] を適用する．
3. *ConcreteFactory* の設定をファイルから読み込めるようにしたければ続けて *Prototype* パターン[1] を適用する．

このように，一般のソフトウェア開発で頻発する問題を解決するための一連の汎用パターン適用を言語として提供することで，初心者はより早く汎用パターンを使い始めることができるし，熟練者は，適用すべきパターンを工程のより早い段階で決定する助けを得ることができる．

パターンとツール

パターンはソフトウェア開発の自動化の手段ではない[11]．むしろ，ソフトウェアの開発過程で開発者の思

☆3 パターンを適用するなどして，ソフトウェアの設計やコードに手を入れることで柔軟性や読みやすさ，再利用性などを向上させること．本連載第3回参照．

考を手助けするような道具である．パターンは開発者に対して，問題点を分析して理解するための文脈を与えつつも，いつ，どこに，どのようにカスタマイズしてそのパターンを適用するかの判断を要求する．もしもパターンの適用を文字通り自動化するとしたら，そこまでを機械が行わなくてはならない．

パターン適用の仕事を自動化するのではなく，それをやりやすくしてくれるようなツールは少ないながら存在する．たとえばSmalltalkで動くRefactoring Browser[12]は，メソッドを親クラスに移動するなどといった定型的なリファクタリング[13]を楽にするツールである．限られたいくつかの設計パターンについては，このツールを用いたいくつかのリファクタリングの組合せで定型的に適用できることが知られている．

近年のパターンへの注目度のわりには，ツールによってパターン適用を支援する取組みは活発でない．パターンという分野そのものが未成熟ならば，ソフトウェア業界としてのパターンに対する理解も未熟であるということだろう．またパターンコミュニティは今の段階でパターンとツールを関連付けることに慎重な態度をとっているという事情もある．ツールによるパターン支援の動きは，まずリファクタリングのためのツールや，今回紹介したパターン指向開発におけるパターンの管理・運用を支援するツールなどの登場から始まるだろう．いずれにせよ目下我々が必要とするツールは，パターン適用を自動化するものではなく，我々自身によるパターン適用をより楽に，活発に行うためのものである．

パターンと工業

パターンの役割は，単に目の前に横たわる問題のその場限りの解決だけではない．「デザインパターンとドキュメンテーション」節（本連載第3回）や「パターン指向開発」節でも触れたように，パターンを開発チームの中で共有することで，メンバ間の意志疎通を良くしたり，開発の作法を一貫させてソフトウェアの保守や拡張をしやすくすることができる．近代工業においてこうした取組みは「標準化」と呼ばれ，その発展に大きな役割を果たしている．標準化技術としてのパターンを，ソフトウェア産業に普及させるためには，よく性質の知られた汎用パターンのセットを規格化する可能性も考えられる．

工業の中でのパターンを考える上で避けて通れないのが，その生産性の評価である．たとえば会社に対してパターン指向開発を導入したいと申し出たとき，具体的にどのくらいの効果が見込めるのかを少しでも言えれば，説得しやすくなるだろう．また，パターンの適用が正しく効果を上げているかどうかを計るときには，「パターンを使わなかったらこうなる」ということが何かしら言えるとよい．パターンがもたらす生産性向上を計る指標として，次のようなものが考えられる：

- （パターン指向開発における）パターンの文書化／周知再利用の徹底のためのコスト
- 汎用パターンの学習のコスト
- 適切なパターンを探し出すためのコスト
- 設計，実装時のパターン適用のコスト
- パターン適用後の試験のコスト
- パターン適用が有効に作用する場合，無意味に終わる場合の比率
- 有効に作用した場合のコード量や開発時間の削減量

筆者の知る限り，このような観点でのパターンの生産性評価の取組みはなく，今後の研究課題といえるだろう．

最後に

本連載ではパターンの全体像に迫るべく，5回にわたって広範な話題を取り上げてきた．また，単なる予備知識に終わらず，実際にパターンを使うきっかけとなるように努めた．パターンが今後有効に発展するためには，乗り越えなければならない課題もある．しかし未熟な今の段階にあっても，パターンは使い方を誤らなければソフトウェア開発の役に十分立つ．読者には主体性を持ってパターンと向き合っていただけたらと願う．

May patterns be with you.
パターンと共にあらんことを．

参考文献

1) Gamma, E., Helm, R., Johnson, R. and Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, Reading, Massachusetts (1995). (本位田真一，吉田和樹監訳：オブジェクト指向における再利用のためのデザインパターン，ソフトバンク，東京 (1995)).
2) Fowler, M.: Analysis Patterns: Reusable Object Models, Addison-Wesley, Reading, Massachusetts, 1997. (堀内　一，児玉公信，友野晶夫訳：アナリシスパターン：再利用可能なオブジェクトモデル，星雲社，東京 (1998)).
3) Hay, D.: Data Model Patterns: Conventions of Thought, Dorset House Publishing (1996).
4) 吉田裕之，山本里枝子，上原忠弘，田中達雄：UMLによるオブジェクト指向開発実践ガイド，技術評論社，東京 (1999).
5) 山本里枝子，上原忠弘，中山裕子，大橋恭子，吉田裕之：ビジネスアプリケーションむけパターン体系とその適用，情報処理学会研究会報告，99-SE-124, pp.27-34, 情報処理学会，東京 (1999).
6) PLoP Home Page: http://st-www.cs.uiuc.edu/~plop/
7) Coplien, J. and Schmidt, D. (eds.): Pattern Languages of Program Design, Addison-Wesley, Reading, Massachusetts (1995).
8) Vlissides, J., Coplien, J. and Kerth, N. (eds.): Pattern Languages of Program Design 2, Addison-Wesley, Reading, Massachusetts (1996).
9) Martin, R., Riehle, D. and Buschmann, F. (eds.): Pattern Languages of Program Design 3, Addison-Wesley, Reading, Massachusetts (1998).
10) Harrison, N., Foote, B. and Rohnert, H. (eds.): Pattern Languages of Program Design 4, Addison-Wesley, Reading, Massachusetts (2000).
11) Johnson, R., 中村宏明，中山裕子，吉田和樹：パターンとフレームワーク，共立出版，東京 (1999).
12) Refactoring Browser Home Page: http://st-www.cs.uiuc.edu/~brant/Refactory/
13) Fowler M. et al.: Refactoring: Improvning the Design of Existing Code, Addison-Wesley, Reading, Massachusetts (1999).

(平成12年4月7日受付)