ソフトウェアパターンについて

堀内　明

1．はじめに

オブジェクト指向のソフトウェア開発は普及しているが、ソフトウェアの再利用はそれほど進んでいない。開発の対象とソフトウェアの最小単位であるクラスとの間にあるギャップが1つの原因である。このようなギャップを埋めるアプローチとして、デザインパターンの考え方が提案された。デザインパターン[1]とは、ソフトウェアの設計過程で出現する問題とその解決方法をパターン化したものである。デザインパターンの考え方は、アーキテクチャの決定・アプリケーションの分析・プログラミング・開発プロセス[2]などソフトウェア開発の諸局面にも適用されつつある。このようなソフトウェア開発に関連する諸パターンのことを、本稿ではソフトウェアパターンと呼ぶ。

デザインパターンは、問題と設計案とを対にして1つのパターンを表現している。直観的に優れた表現方法であるため、ソフトウェア再利用の面で画期的なものである。パターンの概念は、ソフトウェアアーキテクチャ[3]やイディオム[4]などの局面に垂直展開された。デザインパターンはソフトウェアの局所的な構造を表現するためマイクロアーキテクチャと呼ばれる場合もある。それに対して、アプリケーション全体の構造のことをソフトウェアアーキテクチャと呼び、ソフトウェアアーキテクチャに共通

する構造をパターン化したものをアーキテクチャパターン[5]と呼んでいる。

デザインパターンは、特定のアプリケーションドメインから独立した設計のパターン化によって広範に再利用できることが特長である。しかし、汎用のデザインパターンだけではアプリケーションドメイン固有の問題解決にとって不充分であることが明らかになってきた。現在では、アプリケーションドメインの問題を解決するためのデザインパターンが提案されている。

1990年に、Bruce Andersenがソフトウェアアーキテクチャのハンドブックを作る試みを開始し、その後Eric Gammaが合流した。その成果はOOPSLA'91[6]でのワークショップに提案された。ハンドブックには共通の形式が定められており、パターン・テンプレートの基礎になっている。このような活動を通じて次第にコンセンサスが形成され、パターンコミュニティ[7]が成長を始めた。その後もパターンコミュニティは拡大を続け、1994年にソフトウェアパターンに関する国際会議であるPloP (Pattern Language of Programming)[8]が初めて開催された。PloPはその後も毎年開催されており、数多くのパターン[9]が提案されている。日本では、1999年にパターンコミュニティとしてJPLoP[10]が活動を開始した。JPLoPも多くのパターン[11]を提案している。

2. アーキテクチャパターン

2.1 概念

ソフトウェアのアーキテクチャは、ソフトウェアの構成要素とその間の関係を定義している。ソフトウェアの構成要素としては、サブシステム、モジュール、プロセス、関数などがあり、関係としては、データフロー、メッセージのやりとりなどがある。ソフトウェアアーキテクチャには静的側面と動的側面がある。定義の参照関係は静的側面であり、モジュールの

制御方法は動的側面である。クライアント・サーバというアーキテクチャは、クライアントとサーバという要素が相互にどのような依存関係にあるかを定義する静的側面と、サーバがクライアントにどのようなサービスを提供するかを定義する動的側面を持っている。

ソフトウェアの実現すべき諸機能が、それぞれ異なるアーキテクチャを要求するとしたら、設計に要する負担が膨大なものになる。しかし、表層的には異なる機能であっても分析・分類することによって、基本的なアーキテクチャ[12] を見出すことができれば設計の負担を軽減させることが可能となる。

2．2　具体例

初期の頃Shawは、ソフトウェアの制御構造に着目してアーキテクチャパターン[13] を提案した。ソフトウェアアーキテクチャを整理・分類した点で、アーキテクチャパターンの起源として位置付けられている。その後、デザインパターンの進展に影響され、Buschmannもアーキテクチャパターンを提案[14] した。Buschmannの提案したパターンは次のようなものである。

（1）レイヤ

アプリケーションを異なる抽象度のサブタスク群に階層分割するアーキテクチャパターンである。

（2）パイプとフィルタ

データストリームを処理するアプリケーションのアーキテクチャパターンである。処理はフィルタコンポーネントとしてカプセル化されており、フィルタ間を結合するパイプによってデータが受け渡される。

（3）黒板

情報の収集、共有などを通じて非決定論的に問題を解決するための

アーキテクチャパターンである。問題に関する決定的な解決手段が存在しない場合、複数のサブシステムが知識や能力を結集して、部分解や近似解を構築する。データを集中管理する黒板と黒板を更新する知識ソースから構成される。

（4）ブローカ

複数のコンピュータ上に分散するコンポーネントを遠隔呼び出しの手法により連携させる。分散システムを構築するためのアーキテクチャパターンである。

（5）クライアント・サーバ

ユーザの機能を実現するクライアント、クライアントが要求するサービスを提供するサーバから構成される。クライアント・サーバはブローカと同様に、サービスを提供する側と受け取る側の通信やインタラクションを重視したアーキテクチャパターンである。ネットワーク上のビジネスアプリケーションに適用される。

（6）MVC

GUIなどの対話型ソフトウェアシステムをモデル、ビュー、コントローラという3つのコンポーネントで構成するアーキテクチャパターンである。モデルはデータを管理し、ビューはユーザへのデータの表示を行い、コントローラはユーザ入力に応じてビューを制御する。ビューとコントローラがユーザインタフェースを構成している。MVCでは1つのモデルに複数の異なるユーザインタフェースが結合しても矛盾なく表示が行われる。MVCの最初の事例としてはSmalltalkがある。Windowsでは、MFCにおいてDocument－Viewというアーキテクチャを採用している。

（7）PAC

複数の階層構造を構成するエージェントが協調することにより、対話型の複合的ソフトウェアシステムを実現するためのアーキテクチャ

パターンである。各エージェントは、presentation、abstraction、controlという3つのコンポーネントから成り立っている。GUIを持つ複合的なアプリケーションに適用される。

（8）マイクロカーネル

システムへの要求変更に対して柔軟に適用することができるようなアーキテクチャパターンである。最小機能の核となる部分と拡張部分やカスタマイズ部分を分離するアーキテクチャである。マイクロカーネルは拡張部分を協調させるソケットになっている。MachやWindowsNTなどがこのアーキテクチャを採用している。

（9）リフレクション

ソフトウェアシステムの構造や振る舞いを動的に変更できるアーキテクチャパターンである。アプリケーションをメタレベルとベースレベルという2階層に分け、メタレベルが変更の機構を定義し、ベースレベルがアプリケーションの実体となっている。

3．アナリシスパターン

3．1　概念

アナリシスパターンとは、アプリケーションの分析時に頻出するパターンであり、実際のソフトウェアの実装ではなく、ビジネスプロセスの構造を反映した典型的なモデルを示したものである。アナリシスパターンを収集、解説した著書として有名な『アナリシスパターン』[15] の著者Martin Fowlerは、アナリシスパターンのことを「コンピュータシステムをどのように設計するかというよりは、ビジネスを人々がどのように捉えているかを表現したもの」と定義している。

アナリシスパターンは、デザインパターンとは異なりドメインに依存する傾向があるが、実際の分析工程においてモデルの規範として利用できるようにするためには、具体的に記述されていることが必要である。そのた

めアナリシスパターンは、財務会計、在庫管理など具体的なアプリケーションのモデル[16]として提示されている。アナリシスパターンは典型的な分析モデルの例を示したものである。したがって、そのまま再利用できるわけではないが、分析過程において重要な点はアプリケーションドメインやアプリケーションの本質的性質を抽出することであり、そのためのガイドライン[17]として有効性を発揮できる。

Coad[18]は、アナリシスパターンをモデリング手法の一般論が取り扱わなかった実例を示すものとして位置付けている。実例はアプリケーション、戦略、パターンから構成される。アプリケーションはPOSや在庫管理という具体的なアプリケーションのモデリング過程を、戦略はパターンの抽出や適用の各段階における着眼点などの指針を、パターンはモデルのひな型を示している。

Hay[19]は、データモデルのパターンを取り上げている。データモデリング技法だけでは分析の目的である業務の本質を記述することができないことを指摘し、広く共通に利用できるモデルの有用性を主張している。パターンは、組織、会計、MRPなどの問題領域ごとに定義されている。

Fowlerのアナリシスパターン[20]は様々な業務のモデリングにおいて共通化できる構造を抽出したものであり、モデリングの実例として利用できるものとして定義されている。Fowlerは、パターンが複数の問題領域を横断して利用されることを予定しており、よりパターンの抽象度が上がりパターンの適用範囲が拡大することを目指している。

3．2　具体例

（1）責任関係

責任関係というカテゴリーは、人や組織、ポストなどを統一的にモデル化し、それぞれが他者に対して責任を負うことを示すパターンから成り立っている。これにより、組織構造、契約、雇用という多くの問題を表現でき

るモデルが提供される。このカテゴリーは、次のようなパターンから構成されている。

- パーティ
- 組織構造
- 責任関係の知識レベル
- 階層責任関係
- パーティ型の汎化
- 業務範囲
- ポスト

人、組織、ポストを抽象化したものとして、パーティと呼ばれるオブジェクトを導入している。それにより、それぞれに共通したモデル要素を統一的に表現できる。例えば、「個人が組織に電話をかける」、「組織が組織に請求書を送付する」、「営業担当者と顧客との関係」などを容易に記述できる。また、人と組織に関して並列的にポストという要素を組み込むことにより、人とそのポストを切り分けて管理することができる。例として、人の異動が容易に表現できるようになり、一人に複数のポストを割り当てることもできるようになる。ワークフローの記述も人を対象とするよりもポストを対象に行う方が汎用性が高くなる。

責任関係は組織階層のような単なる階層構造よりも複雑な関係がパーティ間に存在する場合に利用される。例えば、2つのパーティ間に責任関係というクラスをおいてそれぞれから委任者と責任者という関連を設定する。このモデルは、「あるパーティが別のパーティとある期間にわたって何らかのルールにもとづいた責任関係を持つ」ことを示すことができる。

責任関係パターンは、業務上のビジネスルールを記述する1つの方法である。ルールを明示的に記述することにより、ワークフローやビジネスプロセスなどをオブジェクトレベルで表現することが容易になる。責任関係パターンは、B2B取引、電子受注、電子調達などにも広く利用できるパ

ターンである。

（2）観測

観測というカテゴリには次のようなパターンがある。

- 量
- 換算率
- 複合単位
- 測定
- 観測概念の継承
- 手続き
- 観測プロセス

「量」パターンはものの数とその単位を組み合わせたものとして量を明示的にモデル化するためのパターンである。「換算率」は、単位の変換を容易に行うための指針を提供している。明示的に単位を取り扱い、汎用的かつ統一的な単位換算が可能になる。

4．デザインパターン

4．1　概念

デザインパターンは設計段階において利用されるパターン[21)~22)]である。パターンコミュニティの中で古くから研究が進んでいる領域であり、多くのパターンが提案されている。デザインパターンは、頻出するオブジェクト群とその構成を記述したものであり、ソフトウェアを設計する場合の定石を集めたものである。デザインパターンは、既に知られている設計のガイドラインを参照することにより、設計段階における誤りを減らし、設計コストを抑える役割をはたすことができる。

デザインパターンは，オブジェクト指向設計において頻繁に出現する重要な設計に名前を付け説明を加えたものである。デザインパターンは、過

去の優れた設計を収集したものであるため、それらを用いることによりソフトウェアの変更や拡張が容易になる。しかし、デザインパターンの無秩序な適用は、設計を複雑にしたり、性能を犠牲にしたりすることになる場合がある。したがって、高い柔軟性、拡張性を実現するためにはデザインパターンを適用するための体系的な方法論[23]が必要である。

デザインパターンを活用する大きな利点の1つとして、設計者の知識の文書化がある。デザインパターンは、ある設計問題をどのように解決するのかを、明確に文書化する。ソフトウェアを構築するための最初のステップとして、ソフトウェアに対する変更や拡張要求を明らかにすることが重要であり、それらの要求と活用すべきデザインパターンとを関連付けることにより、効果的にデザインパターンを活用できる。ソフトウェアの中で変更、拡張要求が発生する場所はホットスポットと定義[24]されている。多くのデザインパターンは解決すべき設計問題を持っているが、デザインパターンの文書の中では、必ずしも変更、拡張といった観点から記述されているわけでない。そのため、ホットスポットという観点からデザインパターンを再構成することにより、開発者は、ホットスポットに対する変更や拡張に関する要求と特定のデザインパターンとを関連させることができるようになる。この関連付けは、設計問題とそれを解決するパターンの関連を明確にするだけでなく、そのパターンが活用された理由の記述にもなる。

4．2　具体例

（1）Observerパターン

Observerパターンは、あるオブジェクトの状態が変化したことを別のオブジェクトへ通知するためのパターンである。Observerパターンは、MVC（Model－View－Controller）アーキテクチャを実現させるための道具として説明されている。Observerは、GUIアプリケーションにおけ

るデータとその表現の切り分けや連携のために利用される。あるデータがあってそのデータにもとづいてスプレッドシートやグラフなどのビューが画面に現れる場合、モデルの状態変化が発生するとそのモデルに依存しているすべてのビューに通知され、再描画が行われる。

どんなビューが自分に依存しているか固定的には決定できない場合やモデルにどんなビューが依存するかを決定したくない場合に変更通知が効果的となる。モデルとビューを切り分けることで両者の結合を疎に保ち、モデルは自分のデータがどのように表現されるかを関知する必要がない。また、ビューの実装が変更されたり新たにビューが生成されたとしても、それがモデルに及ぼす影響を最小限に抑えることができる。このように、状態の変化を引き起こすオブジェクトとその状態変化を受理するオブジェクトとの間に一対多の関係を定義するのがObserverパターンである。

Observerパターンは、GUIだけでなく、分散システムなどでも利用されている。CORBAでは、イベントサービスとノーティフィケーションサービスにおいて、分散版Observerパターン[25]が利用されている。

（2）Mediatorパターン

Mediatorパターンは、オブジェクト同士が直接参照し合わないようにして、その結合度を弱めるために用いられる。オブジェクトモデルでは、オブジェクトとオブジェクトが互いに参照し合い、メソッドを呼び出しながら処理が進行するが、参照し合うオブジェクトの数が多くなるとその参照関係がスパゲティ状態になる場合がある。あるオブジェクトは他のオブジェクトがなければ動作しないので、部分的な拡張や置換などは難しい。

このような場合、Mediatorパターンは各オブジェクト間に仲介オブジェクトを配置することにより、スパゲティ化を防止する。各オブジェクトは自分の相手を仲介オブジェクトに問い合わせてから呼び出すので、多対多の相互作用を1対多にし、その相互作用の制御を仲介オブジェクトに局所化させることができる。これによりオブジェクト間の相互作用の保守、拡

張が容易になる。

（3）Commandパターン

Commandパターンは、要求をオブジェクトとしてカプセル化し、要求するオブジェクトと要求を実行するオブジェクトとを分離させる。例として、GUI構築用ツールを利用してエディタのGUI機能を実装する場合が考えられる。エディタでは、「ファイルを開く」というボタンをクリックして適当なファイルを選択すると、該当するファイルの内容が画面に表示される。このとき、ファイルを開く操作要求を行うのは「ファイルを開く」というGUIボタンであり、実際にファイルを開くのはエディタである。このように、処理要求を行うオブジェクトとそれを実行するオブジェクトが異なる場合、Commandパターンが適用できる。上記の例のように、必然的に機能を別々のオブジェクトに分散させなければならない場合と、意図的に分散させたい場合があるが、いずれの場合にもCommandパターンは適用できる。

（4）Multicastパターン

Multicastパターンは、Observerパターンの拡張・変形[26]として位置付けられている。Observerパターンを利用すると、SubjectとObserverとの間で転送されるイベントの種類が複数になるといくつかの問題が発生する。イベントをクラスとしてモデル化すると、Observerがイベントオブジェクトからイベントデータを取り出すのにダウンキャストする必要がある。また、イベントの数が増えると、イベントの種類別の分岐が複雑になる。Multicastパターンは、この問題を解決することができる。Multicastパターンの場合には、イベントの種類別に異なるクラスを定義し、イベントを受け取って処理する手順を記述する。Observerパターンでは、イベントの受信者がイベントの発生源を知っているのに対し、Multicastパターンでは、イベントの受信者はイベントの発生源を知る必要がない。

（5）Extension Object パターン

Extension Object パターンは、クラスに拡張機能とそのインタフェースを追加する場合の方法を示したものである。一般にクラスの機能拡張を行う場合には多重継承を利用するが、機能追加の数が増加すると継承関係に膨大な組み合わせが発生し管理コストが増大する。このような場合、Extension Object パターンを使用すると、多重継承によらない機能追加が可能になる。また、実行時に Extension オブジェクトを Subject に接続したり切断できるようなプラグインメカニズムを実装できる。Extension Object パターンは、既存クラスのインスタンス同士を組み合わせることによる機能拡張の一例である。Extension Object パターンは、任意のインタフェースを自由に追加できる点で優れたパターンである。

5．プログラミングパターン

5．1　概念

プログラミングパターン[27] とは、コーディングの際に繰り返し現れる手順やコーディングのスタイルをパターンとして整理したものである。プログラミングパターンは、プログラミング言語に依存しているため、デザインパターンと比べると抽象度は低く、再利用の範囲も限定される。しかし、設計を実装化するための具体的なパターンとしては有用であり、プログラミングパターンを利用することにより、実装工程の作業量を減少させることが可能である。プログラミングパターンは、C[28]、C＋＋[29)~30)]、Smalltalk[31]、Java[32)~35)] などの言語に関してパターン集が存在する。

プログラミングパターンは、コーディングパターンとスタイルパターンとに分けられる。コーディングパターンは、頻繁に用いられる論理をどのようにコーディングするかを定めたものである。オブジェクト指向の場合には、オブジェクトの生成、属性の参照、属性の更新、オブジェクトの削除といった基本操作や集合オブジェクトに対する操作がコーディングパター

ンである。データベースへのアクセスやGUIの操作というような繰り返し使用される手順もコーディングパターンである。スタイルパターンとは、インデントの付け方、クラス名やメソッド名の付け方といったコードの書き方の作法に関するパターンである。コーディング規約と呼ばれる場合もある。

プログラミングパターンを用いる実装工程は、分析工程、設計工程に続く工程であり、設計モデルをプログラムに変換する工程である。ソフトウェアの大規模化によるコード量の増加やバグ対応により実装工程は工数のかかる工程になっている。実装作業においては、バグの修正や保守の際に他の人が作成したコードを理解しなければならないが、他人が作成したコードはその内容を理解するまでに多くの時間を必要とし、工数を増加させる。プログラミングパターンの使用は、コードの理解容易性、可読性を向上させ、コードを理解するための時間を減少させることができる。実装に用いられる設計モデルには設計者の意図が含まれており、実装作業においては設計者の意図通りにプログラムを実装しなければならないが、実装担当者がこの意図を読み取れずに実装を行い、問題を発生させる場合がある。このような実装工程の持つ問題に対してプログラミングパターンは非常に効果的である。

5．2　具体例

（1）オブジェクトの基本操作

オブジェクトには、生成、属性の参照、属性の更新、削除という基本操作が必要である。設計担当者はこれらのメソッドを用意し、実装担当者は実装パターンを用意する。各基本操作はパターンとして次のように定義される。

・生成

オブジェクトの生成法は、プログラミング言語で定められている。

プログラミングパターンでは、オブジェクト生成のためのコードの書き方を定義する。オブジェクトの生成法が複数ある場合には、コーディングパターンでオブジェクトの生成法を特定することにより、1つのプロジェクト内でオブジェクトの生成法が統一される。

・参照

参照はオブジェクト属性の参照方法を定義するものである。プログラミングパターンでは、参照のためのメソッド名をスタイルとして定義し、それに従ったメソッドを用意する。

・更新

更新はオブジェクト属性の変更方法を定義するものである。プログラミングパターンでは、更新のためのメソッド名をスタイルとして定義し、それに従ったメソッドを用意する。

・削除

オブジェクトの削除方法はプログラミング言語によって異なる。削除を明示的に行う場合とガーベージコレクタが自動的に行う場合がある。プログラミングパターンは、明示的に行う場合のコードを定義している。

（2）オブジェクトの関連

オブジェクト指向設計においては、クラス間の関連をクラス図を使って設計する。関連には多重度、方向性、継承というような種類があり、実装工程においてはこの設計にもとづいてオブジェクト間の関連を実装する。

・関連

オブジェクト間の関連は、ロール名を名前とするクラスの属性として実装される。多重度、方向性、集約はこの属性をどのように実装するかについて記述する。

・関連の多重度

関連には、関連先のオブジェクトの個数を示す多重度が付けられる。

関連のプログラミングパターンでは、この多重度の意味を踏まえた実装方法を提供する。

・集約

集約では、部分と全体を表現するオブジェクトの関係を示す。設計においては、集約を共有集約と複合集約とに分ける。共有集約の場合には、部分オブジェクトが複数の全体オブジェクトに共有され、複合集約の場合には、部分オブジェクトは１つの全体オブジェクトに従属する。

・関連の方向

関連の方向には双方向と片方向とがある。設計段階では、この方向性も考慮される。片方向の関連を実装するためには、関連を示す属性に関連先のクラスのインスタンスへの参照を保持させるようにする。双方向の関連を実装するためには、自身から関連先のクラスのインスタンスへの参照以外に、関連先のクラスのインスタンスから自身への参照をも保持させるようにする。双方向の関連は、片方向の関連と比べると保守のための負担が多い。双方向の関連を持つオブジェクトの１方を削除するときには、関連先オブジェクトが持つ削除対象オブジェクトへの関連を消去する必要がある。設計段階では、保守の負担を軽減させるため、極力片方向の関連になるよう努力すべきである。しかし、場合によっては双方向の関連を保証しなければならないこともありうる。このような場合に、確実な双方向関連を実装するため、プログラミングパターンを利用すべきである。

（３）集合に対する操作

集合を表すオブジェクトは、プログラミング言語用に用意されたコレクションクラスやコンテナクラスと呼ばれるライブラリクラスを使って実装する。その扱い方は、使用する言語やクラスライブラリに依存する。集合への基本操作としては、要素の追加、削除、参照、検索がある。これらの

操作は集合のライブラリクラス中にメソッドとして標準的に用意されているので、プログラミングパターンとしては、どのような場合にどのメソッドを用いるかを記述すればよい。しかし、集合の要素全体への操作に関しては、特別のプログラミングパターンを用意する必要がある。集合要素全体への操作とは、集合の要素ごとに順次アクセスする操作である。この操作は、クラスライブラリ中に操作クラスが用意されている場合もあるが、その利用方法は統一されていない。したがって、プログラミングパターンとして統一しておく必要がある。

6．フレームワーク

フレームワークとは再利用可能なソフトウェアアーキテクチャであり、類似する複数の問題に関して汎用的な設計を提供するものである。フレームワークはソフトウェアアーキテクチャを実現するための一手段[36]である。フレームワークは、半完成のアプリケーションであり、動作可能なアプリケーションを開発するためには、フレームワークにアプリケーション固有のコードを追加する。フレームワークは、抽象クラスや再利用性の高いクラス、インタフェースなどから構成されている。開発者は抽象クラスのサブクラスを作成したり、インタフェースに実装クラスを定義したりして、アプリケーションを構築する。例えば、GUI構築用フレームワークとマルチチメディア処理用フレームワークを組み合わせて利用する場合やワークフロー管理フレームワークと分散システテム構築用フレームワークを組み合わせて利用する場合がある。フレームワークという概念はGUI構築用のものから発展したが、現在ではGUIに限らず様々なフレームワークが利用されている。

フレームワークにおいては、アプリケーションコードが主導権を握ってライブラリを呼び出すのではなく、フレームワーク側に主導権が移されている。コールバックやポリモーフィズムなどが多用されており、アプリケー

ション固有のロジックを開発者が後から追加する。フレームワークには、アーキテクチャパターン、デザインパターンが含まれている。優れたフレームワークはアプリケーション開発のコストと時間を低減させるが、汎用的であり高い再利用性をもたらすフレームワークであればあるほど理解が難しいという問題もある。フレームワークの例としては、OMGが標準化している産業別フレームワークやIBM San Francisco フレームワーク[37)~38)]などがある。OMGの産業別フレームワーク[39)]を次に示す。

（1）個別産業に依存しない汎用フレームワーク

- Workflow Management
- Knowledge Management
- Internationalization and Time
- Mobile Agent Facility
- Printing Facility
- Task and Session
- Common Warehouse Metadata Interchange

（2）製造

- Distributed Simulation
- Product Data Management
- MES Data Acquisition
- Product & Process Engineering

（3）医療

- Person Identification
- Lexicon Query
- Resource Access Decision Facility
- Clinical Observation Access Service
- Healthcare Data Interpretation
- Clinical Image Access Service

• Medical Transcript Management

（4）金融

• Currency

• General Ledger

• Party Management

（5）電気通信

• Notification Service

• Audio Video Streams

• Telecom Log Service

• Management of Event Domains

• Service Access & Subscription

• Telecom Wireless Communication

（6）エレクトロニックコマース

• Registration & Discovery

• Public Key Infrastructure

• Negotiation Facility

（7）運輸・交通

• Display Manager for Air Traffic Control

• Surveillance

• ITS Center to Center Communication

• Flight Planning

• Space、Satellite and Ground System

• Interoperability for Rail

（8）生命科学

• Genomic Maps

• Bibliographic Query Service

• Macromolecular Structure

- Entity Identification Service
- Gene Expression
- Chemical Structure

フレームワークの抽象度と適用範囲には密接な関係がある。抽象度が高いと適用範囲は広くなるが実際のシステムとの差が大きくなるのに対し、抽象度が低いと適用範囲は限定されるが実際のシステムとの差が小さくなる。開発者の立場では、抽象度が高くてもフレームワークを利用できることが望ましいが、抽象度が高い場合には技術者の能力差が顕著に現れる可能性が高い。フレームワークの利点を最大限に生かすためには、抽象度の低い問題に特化したフレームワークを選択することが重要である。

フレームワークを階層化すると変更要求に対し該当する層のみを交換すればよいので柔軟性が高くなる。新たな問題領域に対するフレームワークを構築する場合は、上層のフレームワークをそのまま利用することができるので拡張性が高くなる。したがって、フレームワークは単一の抽象度で構築するよりも多重の抽象度で階層化することが望ましい。

フレームワークを定義[40]する場合には、追加・削除される可能性のある部分や複数の実現方法が考えられる部分を検討し、追加・変更・削除を柔軟に行えるよう工夫することが重要である。フレームワークの中で柔軟性を保持すべき部分をホットスポットと呼ぶ。フレームワークを構築する場合、最も重要な作業は変更可能性の程度を吟味することである。ホットスポットの設計に適したデザインパターンを次に示す。

- アルゴリズム

  Strategy、Visitor
- イベントに対する対応

  Observer、Multicast
- オブジェクトの状態

State

- オブジェクトの振舞い

  Decorator

- インタフェース

  Adapter、Extension

- 実装方法

  Bridge

- オブジェクト間の通信方法

  Mediator

- コンテナクラスのアクセス方法

  Iterator

- インスタンスの生成方法

  Factory、Method、Abstract、Factory、Prototype

7．パターンコミュニティ

7．1　PLoP

PLoPは、ソフトウェアパターンに関する国際会議であり、毎年カンファレンスやワークショップが開催され、多くのパターンが抽出、提案されている。ワークショップにおいて推敲されたパターンは、PLoPD[41]にまとめられている。既にカタログ化されているデザインパターン[42]の一例を次に示す。

（1）Creational Patterns

- AbstractFactoryPattern
- BuilderPattern
- FactoryMethodPatternTest
- PrototypePattern
- SingletonPattern

（2）Structural Patterns

- Adapter Pattern
- BridgePattern
- CompositPattern
- DecoratorPattern
- FacadePattern
- FlyweightPattern
- ProxyPattern

（3）Behavioral Patterns

- ChainOfResponsibilityPattern
- CommandPattern
- InterpreterPattern
- IteratorPattern
- MediatorPattern
- MementoPattern
- ObserverPattern
- StatePattern
- StrategyPattern
- TemplateMethodPattern
- VisitorPattern

7．2　JPLoP

日本においては、Japan PLoP が定期的にワークショップや研究会を開催しており、パターンに関する議論が行われている。完成度が認められたパターン[43] として次のものがある。

- Persistor
- Query Iterator

- Transaction Context
- SQL Utility
- Header Detail
- Multi-Phase Startup

現在議論の対象となっているパターン[44] として次のものがある。

- 近似評価パターン
- 引用からの品質
- メーリングリストのパターン
- 定時トランザクション
- ポータルシステム構築のためのパターン
- Patterns for Informal Unit Testing
- MultipleCurrency
- スイッチングメソッド
- カスタマイザ
- オプションカード
- アタッチャオブジェクト
- Neutral Interface
- Dispatcher

8．おわりに

デザインパターンはカタログとして広く普及しているが、アナリシスパターンはあまり普及していない。その原因の一つとして、記述形式の問題がある。アナリシスパターンは、デザインパターンのようなパターンテンプレートを持っていない。そのためカタログとしては、使いにくいという欠点がある。今後はアナリシスパターンに適した記述形式の開発が必要である。また、問題領域間の相互関係が多くの人々によって議論されるべきである。

(注)

1) E. Gamma, R. Helm, R. Johnson, J. Vlissides : Design Patterns : Elements of Reusable Object-Oriented Software,Addison-Wesley, 1994, pp.13-23.

2) アルフォンゾ・フュジェッタ、アレキサンダー・ウルフ編　岸田孝一監修　坂本啓司、中小路久美代監訳：『ソフトウェアプロセスのトレンド』、海文堂、1997、pp.3-14.

3) 中谷多哉子、青山幹雄、佐藤啓太編：『ソフトウェアパターン』、共立出版、2001、pp.78-92.

4) 鈴木純一、長瀬嘉秀、田中祐、松田亮一著：『ソフトウェアパターン再考』、日科技連、2000、pp.42-56.

5) F. Buschmann, et al. : Pattern-Oriented Software Architecture : A System of Pattern, John Wiley & Sons, 1996, pp. 26-35.

6) OOPSLA : Object Oriented Programming, Systems, Languages and Applications

7) パターンコミュニティの歴史に関しては、http://c2.com/cgi-bin/wiki HistoryOfPatterns が詳しい。

8) http://st-www.cs.uiuc.edu/~plop

9) カンファレンス、ワークショップの成果は、http://hillside.net/patterns/conferrences/ にまとめられている。

10) http://www.kame-net.com/jplop/

11) http://www.kame-net.com/jplop/PatternRepository/Current-PatternsList.htm

12) L. Bass, et al. : Software Architecture in Practice, Addison-Wesley, 1998, pp.87-96.

13) M. Shaw, D. Garlan : Software Architecture-Perspectives on an Emerging Discipline, Prentice-Hall, 1996, pp.102-109.

14) 前掲５) pp.59-79.
15) M. Fowler : Analysis Patterns : Reusable Object Models, Addison-Wesley, 1996, pp.56-68.
16) 中山裕子、山本理枝子、吉田裕之：パターンの発展と現状、情報処理、Vol.41,No.1,2000,pp.77-78.
17) 吉田和樹、細谷竜一：アナリシスパターンの比較、情報処理、Vol.41,No.4,2000,pp.427-428.
18) P. Coad, D. North, M. Mayfield : Object Models : Strategies, Patterns, Applications, Prentice-Hall, 1995, pp.89-95.
19) D. Hay : Data Model Patterns : Conventions of Thought, Dorset House, 1996, pp.43-45.
20) 前掲15) pp.92-99.
21) 細田竜一、吉田和樹：デザインパターンの適用―基本編、情報処理、Vol.41, No.2,2000,pp.183-189.
22) 細田竜一、吉田和樹：デザインパターンの適用―応用編、情報処理、Vol.41, No.3,2000,pp.293-299.
23) 増田剛、坂本憲広、牛島和夫：発展型ソフトウェア構築のためのデザインパターンの活用とその評価、ソフトウェア発展、岩波書店、2000、pp.4-15.
24) W. Pree : Design Patterns for Object-Oriented Software Development, ACM Press, 1995, pp.43-45.
25) http://www.omg.org/library/c2indx.html
26) J. Vlissides : Pattern Hatching : Design Patterns Applied, Addison - Wesley, 1998, pp.12-18.
27) プログラミングパターンは、実装パターンやイディオムと呼ばれる場合がある。
28) B. J. Kernigham, P. J. Plauger : The Elements of Programming

Style, McGraw-Hill, 1974.

29) J. Coplien : Advanced C++ Programming Styles and Idioms, Addison-Wesley, 1992.

30) S. Meyers : Effective C++ : 50 Specific Ways to Improve Your Programs and Designs, Addison-Wesley, 1997.

31) K.Beck:Smalltalk Best Practice Patterns ,Prentice-Hall,1997.

32) M. Feleisen, D. P. Friedman : A Little Java, Few Patterns, MIT Press, 1998.

33) M. Grand : Patterns in Java, Vol.2, John Wiley & Sons, 1999.

34) D .Lea : Concurrent Programming in Java, Addison-Wesley, 1997.

35) N. Warren, P. Bishop : Java in Practice : Design Styles and Idioms for Effective Java, Addison-Wesley, 1999.

36) K. Beck and R. Johnson : Patterns Generate Architectures, In Proceedings of OOPSLA'94, 1994, pp. 34-36.

37) http://www.ibm.com/software/ad/sanfrancisco/

38) 三ツ井欽一：「IBM San Franciscoフレームワーク：大規模ビジネスオブジェクトフレームワークの実践」、Bit別冊『ソフトウェアパターン』、共立出版、1999、pp.67-86.

39) http://www.omg.org/techprocess/meetings/schedule/index.html

40) 中山裕子、細田竜一、吉田裕之、山本里枝子、吉田和樹：パターン指向開発とパターンの今後、情報処理、Vol.41,No.5,2000,pp.584-589.

41) PLoPD : Pattern Languages of Programming and Design

42) http://wiki.cs.uiuc.edu/PatternStories/DesignPatterns

43) http: / / www.kame-net.com / jplop / PatternRepository / HallOf-Fame. htm

44) http: / / www.kame-net.com / jplop / PatternRepository / Current-PatternsList./htm