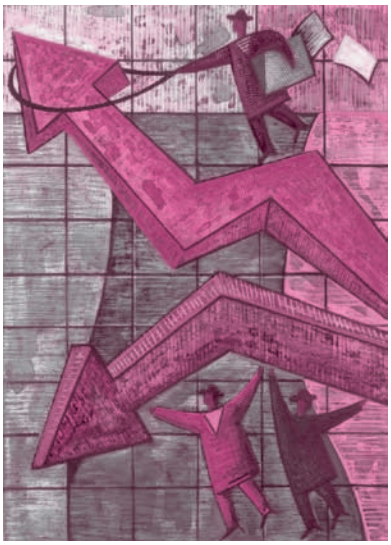

ソフトウェアパターン概観

鷲崎弘宜 早稲田大学

[ソフトウェアパターンとは]

ソフトウェアパターンと聞いて，デザインパターンを思い浮かべる方が多いのではないか．つまりパターンとは，先人が典型的な設計をまとめた結果であり，それを再利用できる，という捉え方である．

確かに，Gamma ら 4 人組がまとめたオブジェクト指向プログラミングのためのデザインパターン集（GoF パターンと呼ばれる）[1]は，膨大なソフトウェアパターンの中で最も知られており，ソフトウェア開発にパターンの考え方を定着させた金字塔である．しかし本稿では，決してそれだけではなく，確かな広がりと古くて新たな使い方を伴って，パターンのムーブメントが今静かに再び起こりつつあることを説明したい．

まず今日，ソフトウェアパターンは設計に限らず開発のあらゆる段階，さらには，プロセスやマネジメントに至るまで発見され，蓄積され，開発や保守運用をより豊かなものへとすることに役立てられている．

パターンの出自としても，次の 2 つの定義を与えられる．1 つは，ソフトウェアの開発や保守運用上の特定の文脈で「偶然ではなく繰り返される」問題と解決をまとめたものである．ノウハウや手本，実証済みの知識と言うことができる．たとえば，オブジェクト間の通知を拡張可能な形で設計したいという問題に，Observer パターン[1]はよい解決策を与える．

もう 1 つは，特定の文脈で周知共有したいビジョンや方針をまとめたものである☆1．たとえば「データベース利用時は必ず O/R マッピングフレームワークと自前のファクトリ生成系を用いることとし，これを『マッパー・ファクトリ』と呼ぶ」といった決定である．

これらの定義は出自において異なるが，ソフトウェアに対して複雑さを隠蔽した抽象表現を与えるという点で共通する．

[ソフトウェアパターンの記述]

Observer パターンを例として，パターンの記述を見てみよう．オブジェクト指向プログラムの振舞いは，オブジェクト間のメッセージ通信で成り立つ．しかし，通信の存在はオブジェクト間の依存関係を意味するため，不必要に複雑な構造を生み出す可能性がある．

たとえば公演のキャンセル待ちを扱う場合，公演側から個々の通知先を特定して通知するのであれば，公演側は通知を待つ全オブジェクトの型を事前に把握しておく必要があり，通知先の新たな追加や変更は難しい．この場合は図 -1 に示すように，通知を待つ側を抽象化した共通の「キャンセル待ち」インタフェースを用意し，公演側からは「キャンセル管理」を介して同インタフェースを実装したオブジェクト群（チケットシステムなど）に一斉通知すればよい．

あるいは，特定の Web サイトの更新状況を，他

☆1 ビジョンとしてのパターンについては，本特集のコラム「企業におけるパターン指向ソフトウェア開発の実践」や記事「これからのみんなのことば，みんなのかたち」を参照されたい．

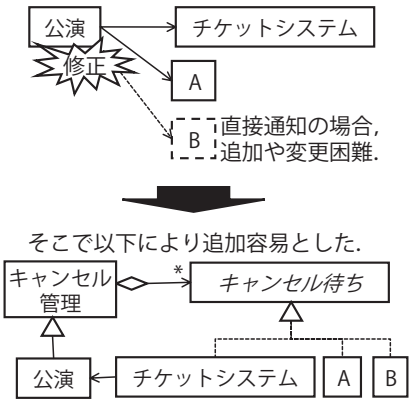

図 -1　キャンセル待ちの設計

のシステムやリーダで把握したいとしよう．この場合，Web サイト側で事前にすべての把握したい側の型を把握するのは無理がある．この場合，共通の「更新待ち」インタフェースを用意し，Web サイトには同インタフェースを実装したオブジェクト群を事前に登録しておけばよい．

これらの例は扱う対象こそ異なるが，詳細を省けば，実は似たようなことをしている．そこで，この似たようなことを特定の形式で書き下した結果が以下に示す Observer パターンとなる．

- 名前 : Observer
- 文脈（文献 1）では「適用可能性」）: オブジェクトが，他のオブジェクトに対して，それがどのようなものなのかを仮定せずに通知できるようにする．
- 問題（「動機」）: 1 つのシステムを協調動作するクラスの集まりに分割する際に，関連するオブジェクト間の無矛盾性を保つ必要がある．しかし，無矛盾にするためにクラス間の結合性を高めたくはない．
- 解決（「構造」「協調関係」）: 変更通知したい全オブジェクトに Observer インタフェースを実装し，変更元オブジェクトへ事前に登録しておく．変更元オブジェクトの状態変更時には一括通知を実現する（図 -2）．
- 結果 : プログラム全体で 1 つの Subject を管理するだけでよく，プログラムの複雑さを回避できる．

このようにパターンは通常，文脈（こういうとき），

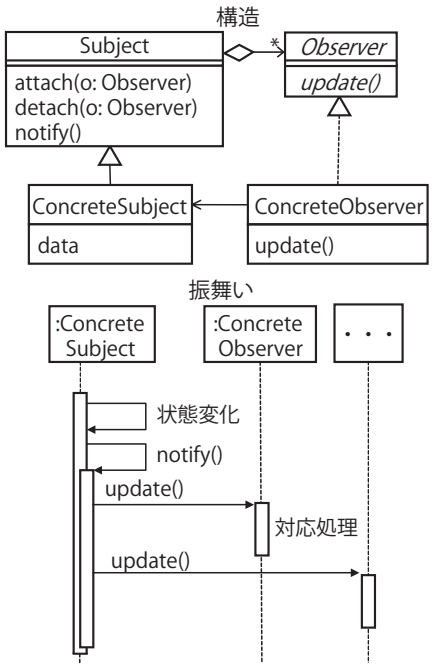

図 -2　Observer パターンが与える解決

問題（こうしたければ・こういうことを考慮して），解決（こうしなさい），結果（こうなる）をそれぞれ異なる項目として明示した形式で記述される．ノウハウや経験談がベタなテキストとして与えられるのではなく，一定の構造化がなされることで，パターンの理解や比較が容易となるのである．

パターンは，必ず成り立つという規則を見出しにくいソフトウェア開発において，類似の事例から言わば帰納的に得た結果であり，個々の状況の違いを考慮しながら具象化して適用することになる．たとえば Observer パターンの場合，公演側に ConcreteSubject の役目，キャンセル待ち側に ConcreteObserver の役目を担わし，対応する Subject 役と Observer 役を用意してキャンセル待ち通知の仕組みを具体化する．ただし適用の前に，暗黙に具体的な状況を抽象化してパターンの「文脈」「問題」との適合性を確認することが不可欠である（図 -3）．

[なぜパターンが必要なのか]

ソフトウェアパターンのもたらす効果として，規律と創造，再利用，対話の 3 点が挙げられる．

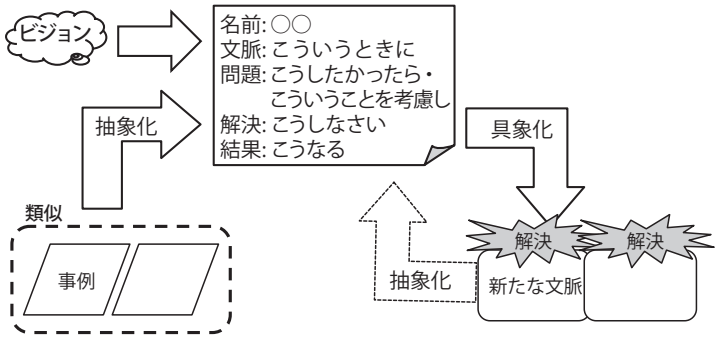

図 -3　パターンと抽象化・具象化

第 1 に，ソフトウェアそのものの表現には物理制約がなく，自由度が高い．そのため規律なしでは，場当たり的でまとまりのない構造が簡単に得られてしまう．そこでパターンは，ソフトウェアの開発に一定の秩序をもたらし，構成を整えることに役立つ．ただしパターンは，「A ならば B」という規則ではなく，「A ならば B や C を考慮して D」という具合である．たとえば Observer パターンの場合，通知側の仕組みの再利用を考えないのであれば，複雑さの低減のため Subject 役と ConcreteSubject 役を一体化する設計もあり得る．このようにパターンは，創造的（たとえばメタファ）と機械的（たとえばコンポーネント）のちょうど中間の抽象表現を与え，規律を保ちつつも一定の創造性を許容する「やわらかな」開発を実現する．

第 2 に，過去の成功体験者やビジョン策定者の思考過程と成果を効率よく再利用できる．つまり，「同じ問題に対する同じ解法を，何度も何度も最初から考え直さずにすむというわけだ」[2]．これは，過去にある状況で問題の解決に成功したのであれば，同じような状況で似た問題の解決に成功する確率が高いという考え方に基づく．ただし，同じような状況で，同じ過ちを犯す可能性も高い．そこで，共通の失敗事例はしばしば回避策とともにアンチパターンとしてまとめられる[3]．

第 3 に，ソフトウェア開発とは，現実世界や理想的な状況について解釈した結果を，抽象化と具象化を繰り返して，最終的に計算機への指示として表現する活動である．その過程においてパターンは，対象の複雑さを隠蔽して抽象化の方法や結果，理由を効率よく示し議論検討する「単語」となる．つまり，いちいち内容を伝えるよりも，「ここは Observer でいこう」と伝えて共有すればよいのである．同じ領域について関連するパターンを集めた成果はパターンカタログと呼ばれる．さらに，パターンの集合に対して適用可能な順序や関係を与えた成果はパターンランゲージ[2]と呼ばれる．これは言わば「言語」として，関係者の対話やイメージの整合，さらには新パターンの発見やパターン間の関係の発展を促し，皆が主体的に参加する形での開発や改善に繋がることとなる．

[ソフトウェアパターンの広がりと分類]

ソフトウェアパターンは，ソフトウェアの開発や保守運用，マネジメントのあらゆる段階について，成功事例やビジョンをまとめる形で蓄積されてきた．開発に関するものはプロダクトパターン，その他のマネジメントやプロセス定義・改善にかかわるものはマネジメントパターンと称され区別される．

さらに各パターンは，抽象度に基づいて分類できる．パターンは抽象表現であるため，適合する状況の広さと，解決にかかる手間の間にはトレードオフの関係がある．問題領域や実装技術に依存せず，適合する状況の広いパターンは汎用型と呼ばれる．たとえば GoF パターンは，分野を問わずあらゆるオブジェクト指向プログラミングに適用できるが，最終的な実装にあたり適用対象におけるプログラミング言語やミドルウェア等の制約を考慮した具体化が必要である．

一方，特定の問題領域や技術に特化したパターンは特化型と呼ばれる．たとえば，金融分野に特化した分析パターンや，セキュリティ対策に特化した設計パターンなどが該当する．これらが適合する状況は狭いが，もし当てはまれば，具体的な解決をほぼそのまま再利用できる確率が高い．

さらに，特化型を発展させて，特定の組織やプロ

目的	工程	パターンの種類	パターンカタログ	
			汎用型	特化型
ソフトウェア開発	ドメイン分析	分析パターン	「もの―こと―もの」，データモデルパターン，モデルパターン・カタログ	データモデルパターン，アナリシスパターン
	要求分析			
	システム分析			
	アーキテクチャ設計	アーキテクチャパターン	アーキテクチャスタイル，POSA	J2EE パターン，PofEAA，エンタープライズ統合パターン
	プログラム設計	デザインパターン	GoF デザインパターン，POSA	J2EE パターン，PofEAA，マルチスレッドパターン集
	実装	イディオム	オブジェクト指向実装イディオム	Smalltalk ベストプラクティス，C++ プログラミング定石，Java スタイル，POSA
	テスト	テストパターン	オブジェクト指向テストパターン	Java テスティングパターン
	保守／改訂	リファクタリングなど	リエンジニアリングパターン，リファクタリング	
プロジェクト管理	組織構造最適化	組織パターン	開発工程の生成的パターンランゲージ	
	プロセス改善	プロセスパターン	開発工程の生成的パターンランゲージ	フレームワーク進化パターン

表 -1　ソフトウェアパターンのカタログの分類

ジェクトに固有のパターンを考えることができる．これは組織固有型と呼ぶことができ，ビジョンや方針としてのパターンが該当する．たとえば前述の『マッパー・ファクトリ』は，既存の O/R マッピングフレームワークと自前のファクトリ生成系を用いる当該組織に固有のパターンであり，同組織における開発要員に知らしめることで生産性と一貫性を高められる．しかしこれが，他の組織においても有用であるとは限らない．

表 -1 に，汎用型と特化型の代表的なパターンカタログを種別ごとに示す．各カタログの詳細は文献 4）等を参照されたい．種々のパターンをボキャブラリとして抑えることは，厳しさや複雑さを増す今日のソフトウェア開発において必須の教養と言える．そこで本稿では以降において，各工程について代表的なパターンを解説する．

■分析のパターン

分析パターンとは，現実世界や理想的な状況を何らかの形で解釈する（たいていモデル化する）過程と成果を，パターンとして表したものである．分析においては設計と異なり，注目する側面が同一であれば結果がぶれることは少ない．そこで，対象とする問題領域について既知の分析パターンがあれば，たいていそのままモデル化のたたき台として活用を検討できる．

代表的なカタログとして，汎用型には Hay の『Data Model Patterns』など，特化型には在庫管理や金融業務などに特化した Fowler の『アナリシスパターン』などがある．

汎用型分析パターンの例として「もの―こと―もの」[5] を取り上げる．これは，業務状況の構造を，主体となる「もの」，記録すべき「こと」，対象となる「もの」の 3 つ組で捉える指針である．もたらす構造の例を図 -4 左に示す．分析ではしばしば「もの」に目をとられるが，ものの関係上に生じる「こと」を明確に捉えることを示唆する基本的かつ重要なパターンといえる．

特化型分析パターンの例として，図 -4 右上に Fowler の勘定（二肢トランザクション）パターンを示す．勘定パターンは「もの―こと―もの」を会計取引に特化させたパターンであり，お金の移動を記録可能な分析モデルの手本といえる．たとえば図 -4 右下では勘定パターンに基づき，当初残高 15,000 円の A 口座から，当初残高 0 円の B 口座へ，4 月 1 日に 5,000 円を振り込んだ事実を表現できている．

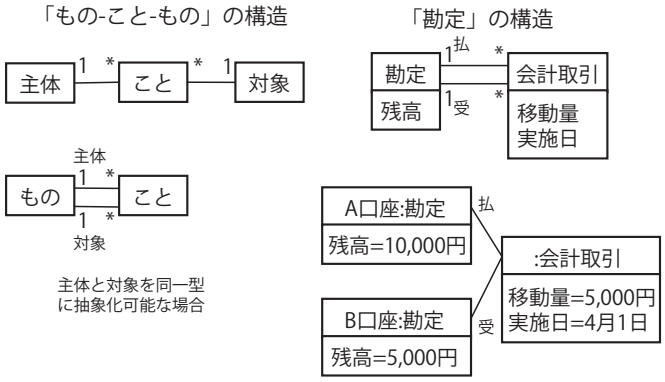

図 -4　分析パターンの例

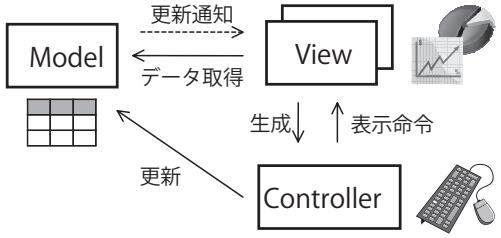

図 -5　MVC の解決が与えるアーキテクチャ

■設計のパターン

ソフトウェアの設計工程は，アーキテクチャ設計（基本設計）とプログラム設計（詳細設計）の 2 段階に大別できる．前者においては主にマクロな構造を扱うアーキテクチャパターン，後者では主にミクロな構造を扱うデザインパターンを活用できる．さらにこれらはしばしば，連携して用いられる．

アーキテクチャパターン（およびアーキテクチャスタイル☆2）とは，機能要求と非機能要求を加味して，必要なコンポーネントおよびコンポーネント間の関係への制約を課す典型的な骨格とその導出過程をまとめたものである．たとえば多層やブローカ，パイプ＆フィルタなどはよく知られている．要求や状況が過去の経験・実績と比較してまったく異なるということは稀であり，パターンをアーキテクチャのたたき台として活用できることが多い．

アーキテクチャパターンの代表的なカタログとして汎用型には，Shaw らの『Software Architecture』におけるアーキテクチャスタイルや Busshman らの『Pattern-Oriented Software Architecture』など，特化型にはエンタープライズアプリケーションに特化した Fowler の『Patterns of Enterprise Application Architecture』や非同期メッセージングによる統合に特化した Hohpe らの『Enterprise Integration Patterns』などがある．

アーキテクチャパターンの例として，Model-View-Controller（MVC）[6] を取り上げる．MVC は，同一データ（たとえば表）について同期を取りつつ多様な GUI（たとえば円グラフや棒グラフ）を持つ対話型ソフトウェアを設計したい場合に，核となるデータと機能をカプセル化した Model，情報をユーザに提示する View，入力装置からユーザ入力を受け取る Controller の 3 コンポーネントからなる図 -5 の構成を与える．MVC の適用結果は，Smalltalk 言語や Java 言語に標準の GUI フレームワーク，さらには，Web アプリケーションなどに見ることができる☆3．

一方，（アーキテクチャ設計を扱わない）狭義のデザインパターンは，アーキテクチャの設計後に個々のコンポーネントやコンポーネント間の関係を決定づけるにあたり適用できる．デザインパターンの代表的なカタログとして，汎用型には GoF パターン（たとえば Observer パターン）など，特化型には Java における並行性設計に特化した Doug Lee らの『Concurrent Programming in Java : Design Principles and Patterns』などがある．

■実装その他のプロダクトパターン

ソフトウェアの分析・設計以降においても，さまざまなパターンが蓄積されており，その活用を通じて効率的かつ効果的に開発を進められる．

実装においては，イディオム（いわば「慣用句」）☆4 と呼ばれる実装のパターン群を利用できる．イディ

☆2 アーキテクチャパターンは，「アーキテクチャスタイル」と同義の場合と，より特定の側面に焦点をあてたものとして捉えられる場合がある．本稿では同義として捉えている．
☆3 Web アプリケーションの構成における MVC は，元来の MVC とは異なる面もあるため「MVC2」として区別することがある．
☆4 コーディングパターン，プログラミングパターンとも呼ばれる．

オムはプログラミング言語やライブラリの特徴を活かして，ソースコードの読みやすさや拡張性，あるいは列挙型の実現といった問題の解決に寄与する．イディオムの代表的なカタログとして，Langr の『Essential Java Style : Patterns for Implementation』，Coplien の『C++ プログラミングの筋と定石』など，各プログラミング言語に応じて存在する．イディオムの中には，言語独立で汎用的なものもある．たとえば，Langr による「どのように実現するかではなく，何を達成するかを示すメソッド名とせよ」という Intention Revealing Method Name イディオムは，Java に限らず有効な考え方と言える．

実装後は，テストにおいて各種のテストパターン（たとえば Binder の『Testing Object-Oriented Systems : Models, Patterns, and Tools』）を利用できる．保守においては，リバースエンジニアリングを通じたプログラムの理解と修正を扱う Demeyer らの『Object-Oriented Reengineering Patterns』や，読みやすさ等の向上のため外部への挙動を変えずにプログラム構造を変える Fowler の『リファクタリング』などの利用を検討できる．

リファクタリングは，デザインパターンと密接な関係にある．すでにあるプログラムに後からデザインパターンを適用する場合，もとの機能を損なわず安全に変更するための手段となる．その典型的な流れは，Kerievsky の『パターン指向リファクタリング入門』にまとめられている．たとえば「Observer によるハードコードされた通知の置き換え」パターンにおいて，「メソッドの移動」や「インタフェースの抽出」といったリファクタリングの連続適用を通じて Observer パターンを適用できることが示されている．

■マネジメントパターン

組織構成やチーム編成の指針を表す組織パターンや，生産性の高い開発プロセスに共通する優れたプロセスを表すプロセスパターンも数多くある．両者は密接にかかわるため，しばしば 1 つにまとめて公開される．

代表的なものとして，「段階的実施」「ペア開発」といったパターンをまとめた Coplien の『開発工程の生成的パターンランゲージ』や，「作業分割」といったパターンをまとめた Cunningham の『エピソード』などがある．これらは，今日のアジャイル開発プロセスの潮流の 1 つとして捉えられる．

[パターン間の関係]

「No pattern is an island」[6] との言葉に象徴されるように，それぞれのパターンは孤立したものではなく，たいてい他のパターンと関係している．その関係に留意することで，パターンをより効果的に用いることができる．特に類似の関係や利用の関係は重要である．

まず，各パターンは特定の文脈において，品質や制約といったしばしばトレードオフの関係にあるさまざまな事柄を考慮したうえで決定の方針を示している．類似の文脈について重視する事柄や問題が異なれば，異なる（しかしいくらか類似した）パターンが存在することとなる．

そこでパターンの選択にあたっては，各パターンの記述における「問題」や「適用可能性」，「動機」，「結果」といった項目に着目し，当該パターンが問題の解決にあたり重視している事柄や品質その他への良い・悪い影響を読み解く必要がある．これらは，その場に働く力という意味で「フォース」と呼ばれ，カタログによっては独立した項目として明示されていることもある（たとえば文献 6）．どのような力が働くかによって，その解消や調停に適した解決方針は異なってくるため，パターンは規則ではないのである☆5．

たとえば Observer パターンの場合，項目「結果」において，過度の更新通知の危険性や，すべての Observer 役へ統一インタフェースへの準拠を強制することに伴う複雑性が述べられている．そこで，

☆5 フォースを把握せずに規則としてパターンを用いると，柔軟性や創造性を欠き，いわゆる「ワンパターン」となる．

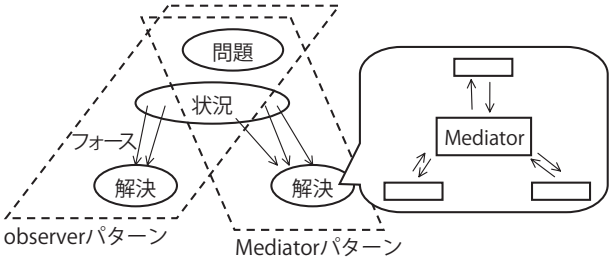

図 -6　同一問題への異なるパターンと Mediator

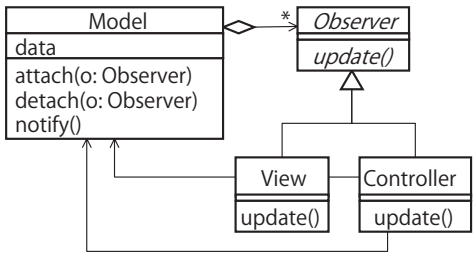

図 -7　MVC アーキテクチャの Observer パターンによる詳細化

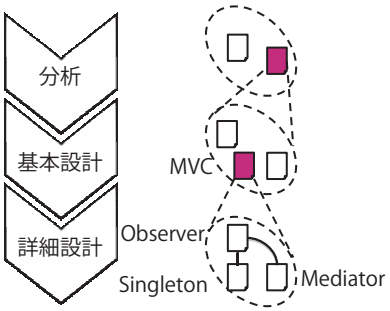

図 -8　パターン間の関係とパターン指向開発

Observer パターンの適用にあたり Observer 役同士の間にも更新通知を行わせたい場合，Observer 役の間に追加で Observer パターンを適用してしまうと，上述の複雑性は無視できないものとなり，設計のスパゲティ化を引き起こす．このような場合は，文脈としては適合するものの，重視するフォースが現状にそぐわないということで適用を断念し，他の方法，たとえば他の類似のパターンを検討するとよい．オブジェクト間の相互作用に関する他のパターンとしては，1 つの仲介役モジュールが集中的に他のモジュール間の相互作用を交通整理する Mediator パターン [1]（図 -6）を検討できる．

類似に加えて，利用の関係についても留意が望ましい．これは，あるパターンが与える解決において他のパターンを内部的に利用できること，あるいは，異なるパターンを併用してより大きな効果を生み出せるような関係を指す．たとえば Observer パターンでは，複数の異なる ConcreteSubject 役を同時に扱いたい場合，Singleton パターン [1] を併用することで，複数の ConcreteSubject <-> ConcreteObserver の対応関係を管理するグローバルかつただ 1 つのオブジェクトを設けるとよい．これは同一のデザインパターンという段階における「横」の利用関係と言える．

一方，段階を超えた「縦」の利用関係も存在する．特に，アーキテクチャパターンの適用により全体への制約を課すアーキテクチャを得たうえで，構成する個々のコンポーネントの詳細化にあたりアーキテクチャ上の制約を満たせるように特定のデザインパターンを適用できる場合がある．

たとえば，MVC パターンを適用したアーキテクチャの詳細化にあたり，Model から View への緩やかな（密結合ではない）更新通知の実現が求められる．その 1 つの解として，図 -7 に示すように Observer パターンの適用を検討できることが知られている．

このようにパターンに加えてパターン間の関係をも蓄積し，分析から設計さらには以降に至るまで体系的にパターン群を再利用して効率的かつ一貫性を持って開発を進めることをパターン指向ソフトウェア開発と呼ぶ（図 -8）．上述のように，自由度の高いソフトウェア開発においては一定の規律がその成功に不可欠である．そこで，特定の問題領域における開発経験の蓄積を経て，パターンおよびパターン間の関係を抽出し，再利用することはごく自然なことと言える．ただしその実現には，その抽出と再利用を進める体制や環境の整備が欠かせない．

[ソフトウェアパターンの展望]

パターンやパターン指向開発の考え方は現在，さまざまな実用的な開発技法へと組み入れられつつある．たとえば，モデル駆動開発技法ではモデル変換

やコード生成についてパターン化が不可欠であり，『エリック・エヴァンスのドメイン駆動設計』はそもそもパターンカタログとしてまとめられており，プロダクトライン開発技法では再利用可能資産としてのパターンの集積と再利用が欠かせない．

今後もソフトウェア工学の各種技法や，それ以外の社会学・組織論などと連携しながら，未解明の事柄を明らかにしつつ活用発展されていくと考えられる．たとえば「パターンはなぜうまく機能するのか？」「パターンが人々の対話にもたらす質とは何か？」といった問いかけは，古くから認識され今なお完全には未解明である．そこで，デザインパターンと欠陥率の関係の分析や，プロセスパターンのシミュレーションを通じた検証など，パターンに対する科学的・工学的取り組みが活発化しつつある[7]．

ソフトウェアパターンそのものについては，汎用的なものは出尽くしたと指摘されることがあり，今後は開発者や関係者それぞれ自身による特化型や組織固有型のパターンの抽出と拡充が期待される．たとえば，今日の情報システムにおけるセキュリティの重要性の高まりを受けて，分析や設計におけるセキュリティパターンが近年急速に抽出されつつある．また，Ajax やクラウドといったソフトウェアの新たな実現・実行・提供方法の出現を受けて，日々対応するパターンが蓄積されつつある（たとえば『Ajax デザインパターン』など）．

これらのプロダクトパターンに加えて，アジャイル開発の一定の成功を受けてあらためて組織構成やプロセス改善，マネジメントにおける全員参加型プロセスの重要性が見直され，マネジメントパターンの蓄積やランゲージ化も進むことが期待される．同時に，既存のパターン群を再認識し，洗練や結び付けを経てパターンランゲージを構成する動きも活発化すると考えられる．

そのような新たなパターンやパターンランゲージについて，皆でレビューし改善および共有する場として，国際会議 Conference on Pattern Languages of Programs（PLoP）が代表的である．PLoP は，世界有数のソフトウェアパターンコミュニティである Hillside Group の支援を受けて世界各地で開催され，ここから数多くの汎用型・特化型のパターンがまとめられ世界に発信されている．

日本においても 2010 年にアジア地域を対象とした初の PLoP である第 1 回 AsianPLoP を開催し，本特集で言及するセキュリティパターンや学習パターンを含めてさまざまなパターンについて諸外国から活発な提案と議論があった．2011 年は 10 月 5 ～ 7 日に東京・早稲田大学にて第 2 回を開催予定である☆6．幅広くパターンを扱う予定であり，ぜひ参加されたい．新たなムーブメントの形成をご一緒させていただければ幸いである．

☆6 http://patterns-wg.fuka.info.waseda.ac.jp/asianplop/

参考文献

1）Gamma, E., Helm, R., Johnson, R., Vlissides, J. 著，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 改訂版，ソフトバンクパブリッシング（2000）．
2）Alexander, C. ほか著，平田翰那 訳：パタン・ランゲージ：環境設計の手引，鹿島出版会（1984）．
3）McCormick, H. W. ほか著，岩谷　宏 訳：アンチパターン：ソフトウェア危篤患者の救出，ソフトバンククリエイティブ，（2002）．
4）深澤良彰 監修，鷲崎弘宜，丸山勝久，山本里枝子，久保淳人著：ソフトウェアパターン：パターン指向の実践ソフトウェア開発，近代科学社（2007）．
5）児玉公信：UML モデリングの本質，日経 BP 社（2004）．
6）Busshman, F. ほか著，金澤典子ほか訳：ソフトウェアアーキテクチャ：ソフトウェア開発のためのパターン体系，近代科学社（2000）．
7）鷲崎弘宜ほか：ソフトウェアパターン研究の発展経緯と最近の動向，情報処理学会第 147 回ソフトウェア工学研究会（2005）．

（2011 年 6 月 23 日受付）

鷲崎弘宜（正会員）　washizaki@waseda.jp

博士（情報科学）．早稲田大学グローバルソフトウェアエンジニアリング研究所所長・理工学術院准教授，国立情報学研究所客員准教授．ソフトウェア工学の研究教育に従事．対外活動に ISO/IEC JTC1/SC7/WG20 国内委員会主査，本会論文誌編集委員，日本ソフトウェア科学会論文誌編集委員ほか．著書に『ソフトウェアパターン』（近代科学社），『ソフトウェアパターン入門』（ソフトリサーチセンタ）ほか．