ソフトウェアパターンへのいざない

青山 幹雄
南山大学
数理情報学部 情報通信学科
mikio.aoyama@nifty.com

ソフトウェアパターンへのいざない

ソフトウェアパターンとは何か，どう使うか，を実践事例を交えて紹介する．

1. ソフトウェアパターンとは

「ソフトウェアパターン」はソフトウェア開発でうまくいった設計や手順などのノウハウ，定石，あるいはベストプラクティスといったものを再利用しやすいように記述したものである．このような再利用は，以前から試みられてきたが，なかなかうまくいかなかった．ソフトウェアパターンでは，「パターン言語(Pattern Language)」と呼ばれる書き方で表現することにより，再利用しやすくしている．

パターン言語は，図-1 に示すように，問題とその解とを対として，ある書式で記述する書き方である．従来の設計の再利用では解だけが記述されていたため，問題を抱えてはいるが解が分からない人にとっては再利用が難しかった．パターン言語では，問題とそれをとりまく条件を記述することにより，問題から解を見出す過程を支援する．参考書で「傾向」と「対策」を対として記述する方法と同じ発想と言える．

パターン言語は，現代建築学の研究者クリストファー・アレキサンダー(Christopher Alexander)がアーキテクチャのパターン表現として提案した[Alex77]．1987 年頃からオブジェクト指向の研究者の間でパターンの考え方をソフトウェアへ応用する試みが始まった[John99]．1994 年に，ガンマ(Gamma)らのいわゆる GoF (Gang of Four)がオブジェクト指向設計をパターン化した「デザインパターン」[Gamm94]を刊行し，急速に広まった．

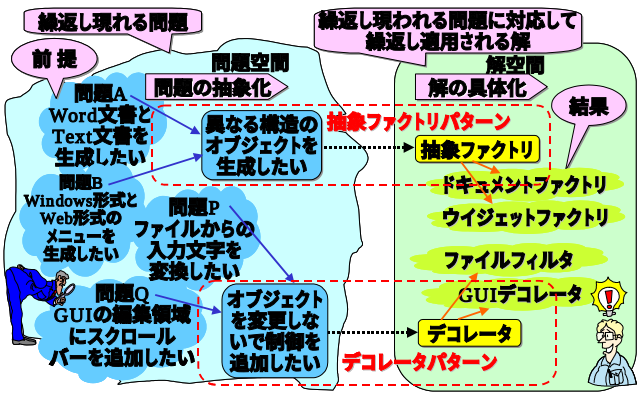

図-1　パターンの考え方

パターンの考え方は多くの人の共感を得た．設計に留まらず，自然に，分析などのプロダクトやその開発プロセスに拡張する活動が起こった．そのため，これらを包括して「ソフトウェアパターン」と呼ぶようになった[Naka99][Schm96]．1990 年代の後半は，「パターン運動(Pattern Movement)」と呼ばれるほどにパターンの適用拡大が試みられた．アーキテクチャパターン[Busc96]，アナリシスパターン[Fowl97]，プログラムパターン[Copl92]などの成果物のパターン化．ソフトウェア開発プロセスをパターン化したプロセスパターン[Ambl98]や失敗のパターン化であるアンチパターン[Brow98]などが提案されている．その中で，デザインパターンは実践でもっとも広く活用されつつある．

2. デザインパターン：パターンの出発点

デザインパターンはソフトウェアパターンの出発点であり，かつ，研究と実践の中心である．デザインパターンを，内容とそれを表現するパターン言語の両面から紹介しよう．

2.1 デザインパターンの考え方

デザインパターンは繰り返し使われる局所的な設計をパターン言語でドキュメント化したものである．まさに，定石と言える．

デザインパターンでは，あるまとまった複数のオブジェクト間の依存関係に規則性があることに着目している．ここで，依存関係は，オブジェクトの生成，複数のオブジェクトの組み合せ(構造の設計)，オブジェクトの実行制御(振る舞いの設計)の 3 種類に分類できる．これに加えて，並行処理の制御のデザインパターンも提案されている．

デザインパターンはオブジェクト間の依存関係を疎にしたり，局所化することにより，柔軟性や再利用性の高い設計の手本を示す．デザインパターンを適用して，設計の柔軟性や再利用性を高めることができる．さらに，デザインパターンの考え方を理解することにより，類似の局面で良い設計のヒントを得ることができる．それは，設計力の向上につながる．

2.2 デザインパターンの内容

表-1は主なデザインパターンを示す．表の上欄に挙げた 23 個のパターンは GoF によって「デザインパターン」の本で提案された．その後，多くのデザインパターンが提案されている．これらのパターンをまとめたパターンカタログが書籍の形で公開されている[Copl95][Vlis96][Mart97][Harr99]．

表-1 主要なデザインパターン

	生　成	構　造	振る舞い
GoF	ファクトリメソッド 抽象ファクトリ ビルダ プトロタイプ シングルトン	アダプタ ブリッジ コンポジット デコレータ ファサード フライウェイト プロキシ	インタープリタ テンプレートメソッド 連鎖(Chain of Responsibility) コマンド イテレータ メディエータ メメント オブザーバ ステート ストラテジー ビジター
その他	オブジェクトプール[Gran98]		リアクタ[Schm00] パブリッシュ/サブスクライブ[Busc96]

2.3 パターン言語

パターンを分かりやすく表現するために，共通の書式で記述する必要がある．デザインパターンでは，アレキサンダーのパターン言語に基づいた書式を用いている．著者により細部の違いはあるが，次の記述項目が共通している．

(1) パターン名
(2) 概要あるいは目的:一言でいって何か
(3) 問題:解こうとしている問題とその背景，適用可能性
(4) 解法:クラス図やシーケンス図で記述した設計例，実装の指針，サンプルコード
(5) 結果:パターンの効果と限界，使用例，関連パターン

パターン言語の記述の特徴はその具体性にある．問題は背景や内容を具体的に記述する．それに対し，解法はクラス図やシーケンス図で設計例を示す．これによって，問題に直面した設計者が問題の言葉で理解できるようにしている．Java などに言語を限定して完結したソースコードも示すこともある[Coop00][Gran98][Naka01]．一方，問題の記述には，ある程度の抽象化も行い，結果の欄で使用例を示すなど，異なる問題に適用するための工夫もされている．

2.4 デザインパターンの効果

デザインパターンの効果は次の 2 つの面がある．

(1) 設計力の向上

図-2 は抽象ファクトリパターンを利用しなかった場合と利用した場合のオブジェクト間の依存関係を示す．抽象ファクトリによって，クライアントと生成対象オブジェクトの間がドキュメントファクトリという単一のクラスのインタフェースで統一され，依存関係が簡潔となる．生成するオブジェクトの追加･変更が局所化される．

(2) 設計理解力の向上

デザインパターンのもう一つの効果は，設計に名前を付けたことにより，クラスより一段抽象度の高いレベルで設計内容を伝えることが可能になったことである．例えば，電子回路では増幅器や発信器と言えば，その設計が何であるか理解できる．これは，工学分野では当然のことである．ソフトウェアの設計においても，パターンの名前が設計を表す名称として使われつつある．「ここはオブザーバを使っている」と言えば，個々のクラスとその間の関係を説明しなくても何の設計か分かるからである．

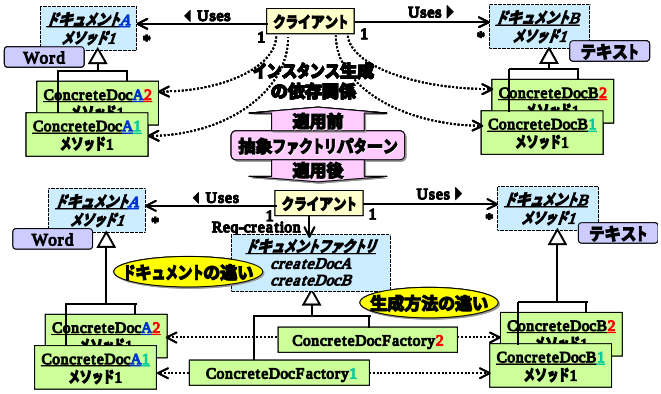

図-2 抽象ファクトリの使用前と後での依存関係の違い

3. パターンを使う

デコレータパターンを例に，デザインパターンの使い方を紹介しよう．題材は，Java のファイル入出力である．

まず，デコレータパターンの利用方法を示す．次に，デコレータパターンと他のパターンを組み合せてファイル入出力フレームワークを設計する方法を示す．

3.1 デコレータパターン

デコレータ(Decorator)パターンは，特定のオブジェクト(クラスではないことに留意)の外に別のオブジェクトをラッピングすることにより，処理を付加できる．その意味で飾る(decorate)と呼ぶが，ラッパー(Wrapper)とも呼ばれている．応用範囲が広いパターンである．

3.2 Java ファイル入出力におけるデコレータパターンの利用

Java のファイル入出力パッケージ(Java.io.*)は，パイプ＆フィルタのアーキテクチャをとっている．

ファイル入出力では，バイト単位，文字(16 ビット)単位，行などの処理単位やバッファリングを行うか否かなど，処理する上で多くの組み合せがある．これを継承を用いて実現すると，継承関係の組み合せ爆発が起こり，クラス階層が巨大になってしまう．この問題に対し，Java のファイル入出力では，デコレータパターンを適用して解決している[Ecke99]．

図-3 にファイル入力の主要なクラス図を示す(各クラス名の頭の Java.io は省略している)．基本的なファイル入力処理を行う FileInputStream に対し，フィルタ(FilterInputStream)によって必要なフィルタを連結して処理することにより解決している．FilterInputStream はフィルタ，すなわちラッパーのインタフェースを規定するだけで，何もしない抽象クラスである．

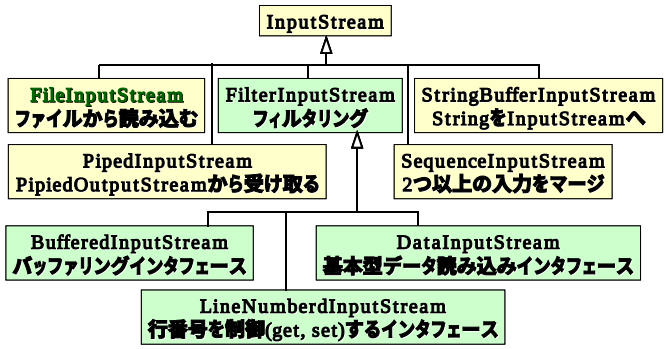

図-3 Java.io パッケージのファイル入力の主要なクラス階層

さらに，このような基本的なフィルタだけでなく個別のフィルタを作って入出力処理を行いたいことがある．文字変換やレコードの変換などである．この場合も，同様に，個別のフィルタを連結して実現できる．このようなデコレータパターンの使用例を図-4 のクラス図に示す．

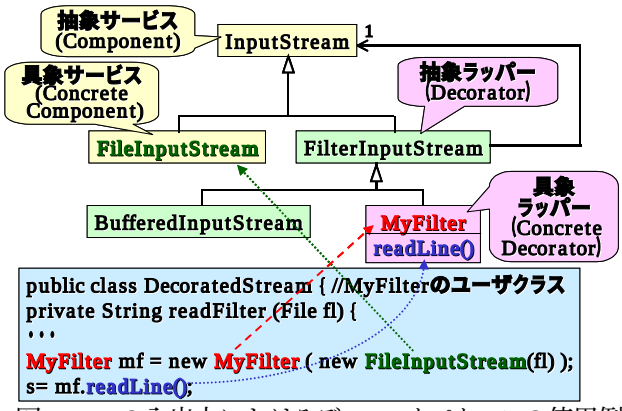

図-4 Java の入出力におけるデコレータパターンの使用例

ここで，MyFilter という個別のフィルタを FilterInputstream のサブクラスとして実装している．これを用いて，フィルタリングする場合，図に示すように，次のようにフィルタをネストして連結するだけでよい[Coop00].

```
MyFilter mf = new MyFilter (new FileInputStream (fl));
```

図-5 は，図-4 のクラス間の振る舞いとデータの流れを示したものである．

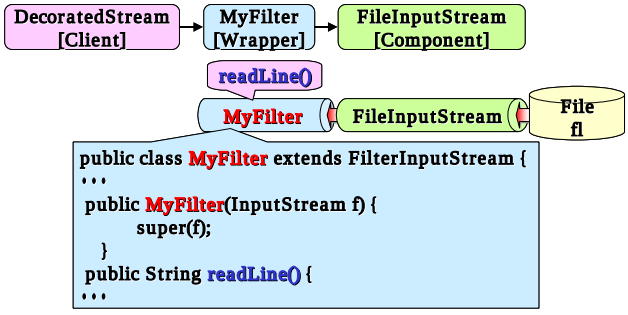

図-5 Java.の入出力におけるデコレータパターンの使用例

このように，デコレータパターンを用いてパイプ＆フィルタアーキテクチャの中でフィルタのラッピングを組み合せることによりストリーム入出力処理の多様性を解決している．単純さと柔軟性をあわせ持つエレガントな設計と言える．

3.3 なぜ，デコレータパターンか：関連パターンとの違い

実際の設計では様々な制約条件を考慮する必要があるので，細部が異なることが多い．同様に，デザインパターンにも類似のパターンがある．ここでは，デコレータパターンの類似のパターンであるストラテジパターンとの比較を通して，パターンの選択について考えよう．

(1)関連パターンとの比較：デコレータとストラテジ

ストラテジパターンは，同一のインタフェースで異なるアルゴリズムなどで実装したクラスにパラメータで振り分けるパターンである．例えば，ある配列データを折れ線グラフや棒グラフなどの異なるグラフで表示する場合，あるいは，あるデータ系列に対し，異なるアルゴリズムで並べ替えや計算などを行う場合に適用する．

表-2 にデコレータパターンとストラテジパターンの設計の違いを示す．

表-2 デコレータとストラテジの違い

設計特性	デコレータ	ストラテジ
設計概念	外からくるむ(ラッピング)	中で振り分ける(カスタマイズ)
インタフェース制約	コンポーネントインタフェース＝デコレータインタフェース	コンポーネントインタフェース≠ストラテジインタフェース(一致する必要はない)
変 更	殻(ラッパー)を変更(コンポーネントの変更不要	中身(コンポーネント)を変更

(2)パターンの選択

パターンの選択では設計の狙いや意図が選択の指針となる．このような意図や狙いをまとめてパターンのフォース(forces)と呼ぶことがある．

パターンの選択は設計そのものに他ならない．したがって，パターンを選択するためにはパターンを取巻く設計上の制約や設計の意図を明確にする必要がある．これは，フォースを明らかすることである．

例えば表-3 に示すように，パターンの構造とフォースの両面から評価する方が適したパターンの選択できる[Fshu96].

表-3 パターンの選択

	フォース不適合	フォース適合
構造不適合	レベル 1：パターンとして不適切	レベル 3：同じ目的を実現するように設計されているが設計が洗練されていない可能性がある
構造適合	レベル 2：パターンの一部が発見できるが，巧妙に実装されている可能性がある	レベル 4：適合する

3.4 パターンを組み合せたファイル入出力フレームワーク

デコレータパターンと他のパターンを組み合せて，ファイル入出力フレームワークを設計する例を示そう[Aoya00][Wool99]. 図-6 に問題の概要を示す. 複数のフィールドから成るレコードのデータストリームを EOF まで読み込み，そのレコードを取り出して，オブジェクトとして格納する.

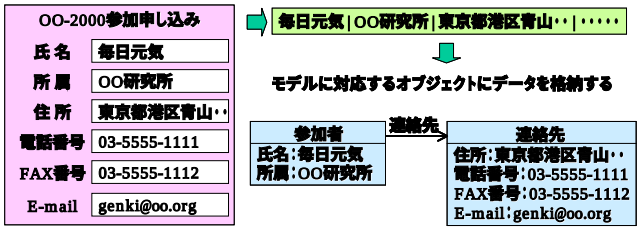

図-6 ファイル入出力の問題

この問題は，次の 3 つの問題に分割できる.

1) 複合フィールドから成るレコードの読み込み
2) レコードストリームの読み込み
3) 読み込んだデータのオブジェクトへの格納

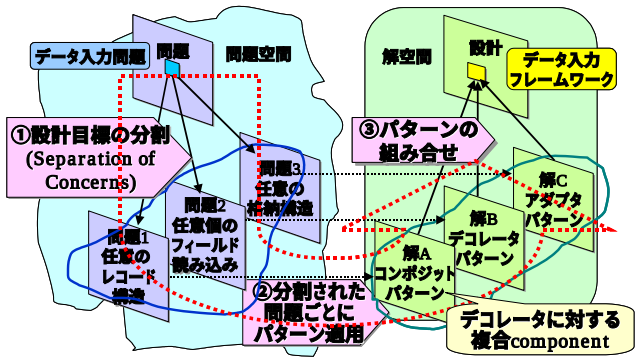

図-7 ファイル入出力問題の分解と解への対応付け

これらの問題は，図-7 に示すように，それぞれ，次のデザインパターンによって解決できる.

a) 複合レコード⇒コンポジットパターン
b) レコードストリーム入力⇒デコレータパターン
c) オブジェクトへの変換⇒アダプタパターン

この 3 つのデザインパターンを図-8 に示すように組み合せて，ファイル入出力フレームワークとなる.

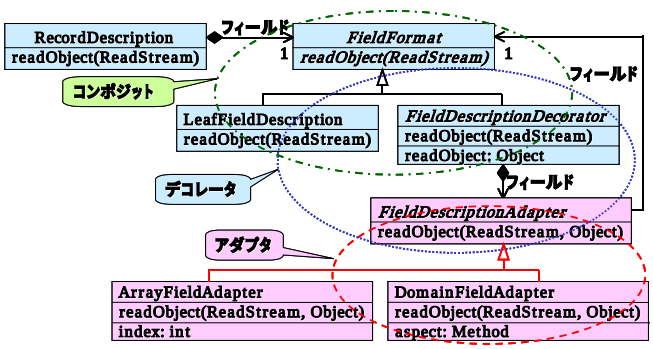

図-8 パターンを組み合せたファイル入出力フレームワーク

4. デザインパターンからソフトウェアパターンへ

4.1 パターンの広がり

パターンの考え方がソフトウェアパターンとしてソフトウェア開発のあらゆるプロダクトとプロセスへ応用されようとしている. 図-9 はこの中の主なソフトウェアパターンを示す.

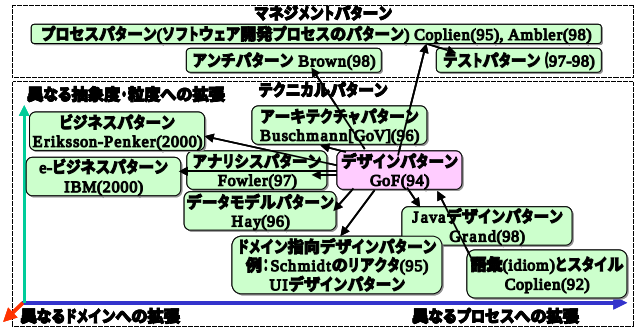

図-9 デザインパターンからソフトウェアパターンへの展開

図に示すように，これらのソフトウェアパターンを本稿ではテクニカルパターンとマネジメントパターンに分類する. テクニカルパターンは分析や設計のプロダクトのパターンである. マネジメントパターンは開発プロセスやその管理に関するパターンである.

これらのパターンは，まだ，かならずしも実証され，実践されているわけではないことに留意すべきである.

また，一般的に言って，テクニカルパターンは，具体的な分析･設計内容をパターン化している. 一方，マネジメントパターンは，抽象化した行動指針に近い. これは，個々のマネジメントの状況が多様であるため，一般化するためには，ある程度抽象化せざるを得ないからである.

テクニカルパターンでも，対象によってパターン言語(表現と内容)が異なる. アーキテクチャパターンはプロダクトの構造が主たる設計対象となるのでデザインパターンと似ている. しかし，最近では構造に加え，性能などの非機能的要求を併せてパターン化する研究もある[Klei99]. これは，アーキテクチャがシステムの性能の主要な決定要因となるからである.

筆者らは，分散オブジェクト環境を対象とする性能設計のパターン化を行ってきたが，この場合も，性能という非機能的特性と構造の両面からパターン化を行っている[Iris00].

また，デザインパターンを設計時ではなく，実行時に利用して，動的にアダプタを生成する方法もある. 例えば，JDK 1.3 から提供されている動的プロキシパターン[Rich00]と JavaBeans のイントロスペクション機能を組み合せて，アダプタを自動生成できる. これによって，非標準(カスタム)イベントを用いた JavaBeans でも実行時に動的に組み合せることができる[Saik00].

必ずしもマネジメントパターンに限らないが，失敗のパターン化である「やってはいけないパターン」として「アンチパターン」も提案されている[Brow98].

4.2 分析のパターン

分析では，パーティ(人や組織などを一般化した概念)などの実世界を構成する基本的な要素や取引などの基本的な関係がアナリシスパターン[Fowl97]やビジネスパターン[Erik00]としてパターン化されている．本節では，まず，オブジェクト指向分析でしばしば遭遇する「多対多の関連」をパターン化する[Evit00]．次に，ビジネスパターンを紹介する．

図-10 は商品とその顧客の関連を示す．ある商品には複数の顧客がおり，ある顧客は複数の商品を購入することから，「多対多の関連」をもつ．このような関連は，「授業科目と教師」や「複数の図書館と利用者」など，繰り返し出現する．

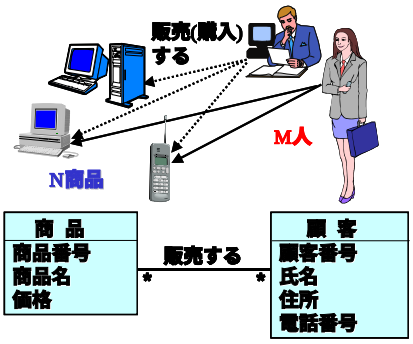

図-10 顧客と商品間の多対多の関連

一般に，「多対多の関連」は，データ構造が不定となり(データモデリングの言葉で言えば正規形となっていない)，そのまま実装できない．したがって，このような場合，図-11 に示すように，関連を表す販売クラスを仲介として「2 つの 1 対多の関連に解く」ことが必要になる．

関連クラスを作ることで，実は，このモデルは柔軟性も高まっている．例えば，図-12 に示すように，販売クラスには販売日や販売価格を属性として持つことは自然である．これらは販売ごとに設定されるもので，上側の販売クラスのない場合にはこのような属性の設定はできない．

このパターンは簡単ではあるが，分析においては知っておくべきことである．その意味で，パターンとして組織内で再利用できるようにしておくと良い．

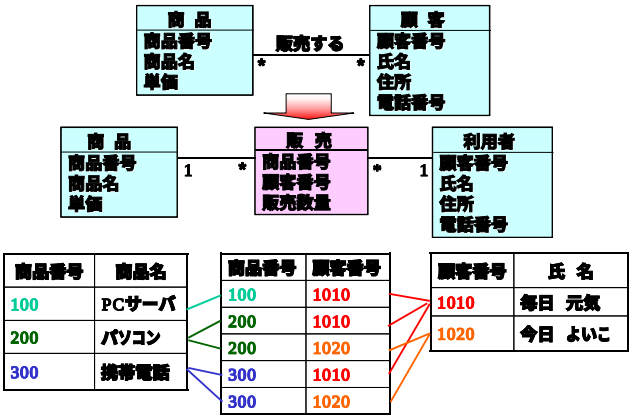

図-11 販売クラスによる関連の分解

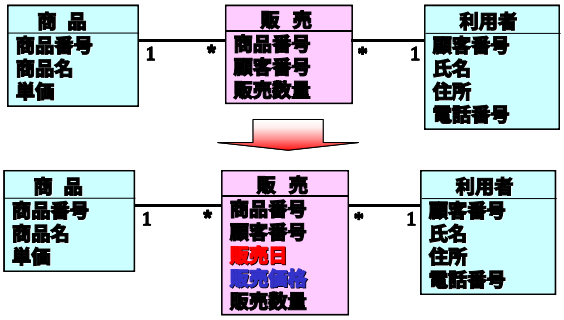

図-12 販売クラスの属性の追加

Eriksson らはこのようなビジネスの基本的な要素や取引をビジネスパターンと名付けて，幾つかのビジネスパターンをカタログとしてまとめている[Erik00]．このようなビジネスパターンが有効であるためには，パターンで使う用語やその意味が統一されている必要がある．このため，従来は，データ辞書などを作成してきた．ここでは，ビジネスを構成する用語間の意味関係をクラス図で表し，用語のメタモデルを定義している．

例えば，図-13 は，ビジネスの基本要素とその間の関係を定義している．

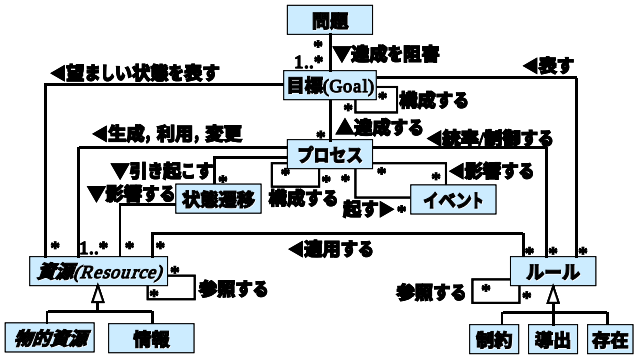

図-13 ビジネスのメタモデル

図-14 は契約ビジネスパターンとそれを保険契約へ適用した例を示す．

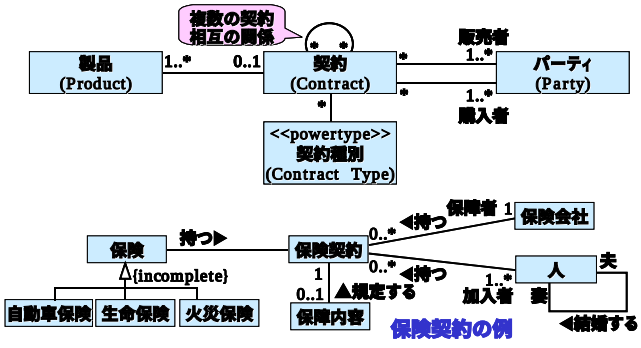

図-14 契約ビジネスパターンと保険契約への適用例

5. パターンを作る

パターンを作ることは，パターンを使うより稀であると思われる．また，一般に，パターンを使うよりは作るほうが難しい．しかし，「多対多の関連」の解消で見たように，分析や設計の多くの局面で，あったら便利というパターンは少なくないであろう．また，大規模プロジェクトではプロジェクト固有であっても再利用の効果が期待できるパターンもある．このようなパターンを発掘する活動をパターンマイニング(Pattern Mining)と呼ぶ．

発掘したパターンの原石は，図-15 に示すように，洗練して広く利用できるパターンに高める必要がある．このために，詩作における推敲の方法を応用したライターズワークショップが PLoP (Pattern Language of Program design)を始めとして開催されている．できたパターンは本として刊行されている [Copl95][Vlis96][Mart97][Harr99]．国内では，JPLoP (http://www.kame-net.com/jplop/)がワークショップを開催している．

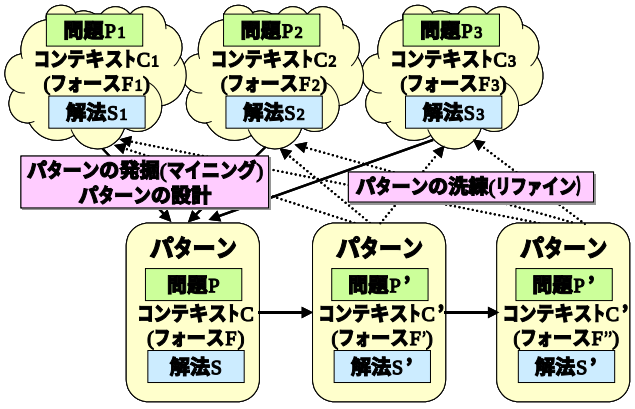

図-15 パターンの発掘と洗練

表-4 Java におけるパターンの適用例

	パターン名	適用例
生成	ファクトリメソッド	Java.net.URLConnection の getContent
	抽象ファクトリ	Java.awt.Toolkit
	プロトタイプ	JavaBeans
	シングルトン	Java.lang.Runtime
構造	アダプタ	Java.awt.event.WindowAdapter
	ブリッジ	Java.awt の Component（Button, TextField）とそれを実装するための Java.awt.peer(ButtonPeer, TextFieldPeer)
	コンポジット	Java.awt.swing の Container(Panel, Frame など)
	デコレータ	Java.io.FilterInput(Output)Stream でのフィルタリング
	ファサード	Java.net.URL
	フライウェイト	Java.lang.String(プログラム内で同じ文字列が複数ある場合)
	プロキシ	JavaRMI（Remote Method Invocation）
挙動	イテレータ	Java.util.Iterator
	メディエータ	Java.awt.swing.FocusManager
	オブザーバ	Java のイベントモデル(Java.util.Observer, Java.util.Observable)
	ストラテジー	Java.util.zip パッケージのチェックサム計算(Adler32, CRC32)

6. パターンの実践事例と実践のヒント

6.1 デザインパターンの適用事例

デザインパターンは実際に多くの設計で使われている．ここでは，デザインパターンによるフレームワークの設計，デザインパターンによるアプリケーション開発の両面で幾つかの事例を紹介する．

(1) デザインパターンによるフレームワークの設計

1) Java におけるデザインパターンの利用

表-4 は JDK などで使われているデザインパターンの例を示す[Gran98]．Adapter や Observer など，デザインパターンの名称が使われていることに注意しよう．上で述べたように，このような命名によって，これが何の設計であるか，名前を見れば分かるのである．

2) POS フレームワークの設計

図-16 はデザインパターンを組み合せて POS フレームワークを設計した例である[ITC00]．この例では，次のようにパターンが使われている．

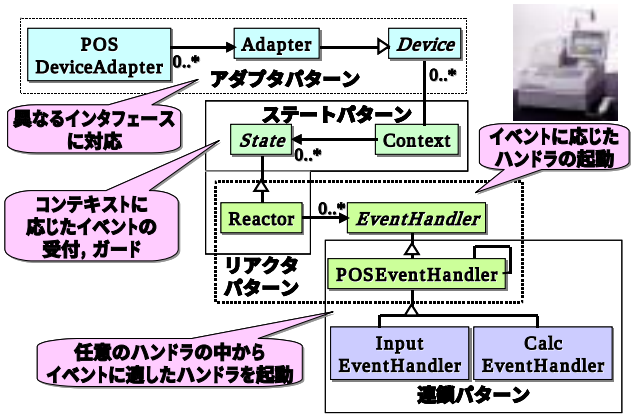

図-16 デザインパターンを組み合せた POS フレームワーク

a) アダプタ：入力信号の論理イベントへの変換
b) ステート：状態遷移の制御
c) リアクタ：状態に応じたイベントの振り分け
d) 連鎖：イベントに応じた処理の起動

3) IBM San Francisco フレームワーク

IBM の San Francisco フレームワークは大規模な業務フレームワークとして知られている．この中で，GoF のデザインパターンに加え独自に設計した様々なデザインパターンが利用されている[Mond00].

(2)デザインパターンによるアプリケーション開発

アプリケーション開発でもパターンの

利用は行われているが，パターンと意識しないで利用していることが少なくないであろう．したがって，アプリケーション開発におけるパターンの利用を組織的に推進している事例の報告は少ない．国内この中では，次の事例がある．

1) 流通業コンビニ店舗システムへの適用：GoF のデザインパターンなどを適用して大規模コンビニ店舗システムを構築した事例がある[Naka99].
2) 業務アプリケーション開発へての適用：データベースアクセスなど，GoF にはない開発対象固有のパターンを作り，適用した[Yama98]. また，パターンを適用するためのプロセスと開発方法論も開発している．

6.2 実践のヒント

ソフトウェアパターンはドキュメントであり，特別なツールなどを必要としないことから，コスト面での導入障壁は低い．しかし，開発のあり方を変えることから，理解し，実践する人の側面が課題となる．デザインパターンを中心にソフトウェアパターンを実践するヒントを，実践の対象，技術面，管理面から紹介しよう．

(1) ソフトウェアパターンの活かし方

開発現場でソフトウェアパターンを利用する方法として，例えば，次の 2 つがある．

1) 開発への適用：デザインパターンをフレームワーク設計やアプリケーション開発に適用する．あるいは，本稿で述べたJavaの入出力パッケージのようにフレームワークなどを利用する際に，その設計原理を理解する方法としてデザインパターンがどう使われているかを理解する．
2) オブジェクト指向教育への適用：オブジェクト指向を理解する素材としてデザインパターンを利用する．

(2) 技術面：理解・実践する

デザインパターンを導入する上で，パターンの理解とパターンを利用する開発方法論の 2 つがある．

a) パターンの理解：パターンを理解する一つの方法は，身近な事例でパターンカタログを作成することである[Naka99][Naka00][Suzu00]. 図-17 は筆者の研究室の学部 4 年生がパターンを理解するために Web 上に作成したパターン理解支援システムである．パターンごとに構造を示すクラス図とその説明を上下 2 つのフレームに分けて表示する．例を視覚的に表示できるパターンは Java で実装し，その動作は右下に表示されているように Java アプレットで示す．Java ソースコードはクラス図内の各クラスをクリックすると右上の画面のように表示される．このように，現場では，身近な適用事例を用いて，自らの言葉でパターンを説明することが，理解と実践を促進するのではないか．また，筆者らは，類似したパターン群の中でパターン間の関係を示すことが，パターンの違いを理解したり，選択の指針になるのではないかと考えている[Aoya99].

図-17 デザインパターン理解支援システムの画面例

b) パターン指向開発方法論

パターンを利用した開発方法論については，山本らの提案がある[Yama98]. まず，第一歩は，デザインパターンなどの名称が組織の共通設計用語となることである．

(3) 管理面

パターンを実践するための管理面は再利用の推進と同様，開発者の動機付けにある．デザインパターンを理解し，適用した良い設計を評価し，組織の行動原理とする必要がある．また，Java のファイル入出力フィルタのように，パターンの実装がそのまま再利用できる場合には，組織で共有するリポジトリなどの仕組みを作る必要がある．

一方，悪しき設計がもたらす失敗からも学ぶ姿勢も必要であろう[Brow98][Hata00].

7. まとめ：ソフトウェアの達人への道

本稿では，例題や事例を交えてソフトウェアパターンの考え方を紹介した．

ここで紹介したパターンや例題が「そんなこと当たり前で良く分かっている，何を今さら」と思われた方は，より多くのソフトウェアパターンを編み出して，公開し，達人をめざして頂きたい．「なるほど」と思われた方は，ソフトウェアパターンをうまく使いこなして，良い設計・開発をめざして頂きたい．

囲碁や将棋では，定石（将棋では定跡と書くがこれは囲碁では布石に相当）を習得しないかぎり，有段者にはなれない．デザインパターンの提案される以前に，ソフトウェア工学と囲碁・将棋の両方に造詣が深い先輩から「定石を 100 ぐらい使えないと有段者にはなれないように，ソフトウェアでも同じことが言えるのではないか」という意見を伺ったことがある．定石の数はともかく，パターンの意味するところも同じである．また，囲碁や将棋でも，定石は実戦で使って初めて身につくものである．さらに，定石はそれを覚えることが目的ではなく，手段であることも同じである．パターンの背後にある設計の考え方を身に付け，良い設計が実践できることが重要である．

パターンをマスターしてソフトウェアの達人になろう．

参考文献

[Alex77] C. Alexander, S. Ishikawa, and M. Silverstein, *A Pattern Language: Towns-Buildings-Construction*, Oxford University Press, 1977 [平田翰那(訳), パターン・ランゲージ:町・建物・施工, 鹿島出版会, 1984].

[Ambl98] S. W. Ambler, *Process Patterns*, Cambridge University Press/SIGS Books, 1998.

[Aoya99] 青山幹雄, ソフトウェアパターンのモデル化とパターン進化のパターン, 情報処理学会ソフトウェア工学研究会, No. 124-6, Oct. 1999, pp. 35-42.

[Aoya00] 青山幹雄, 実践ソフトウェアパターン, オブジェクト指向 2000 シンポジウム資料集, 情報処理学会, Aug. 2000, pp. 63-79.

[Brow98] W. J. Brown, R. C. Malveau, H. W. "Skip" McCormick III, and T. J. Mowbray, *Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis*, John Wiley & Sons, 1998 [岩谷 宏(訳), アンチパターン, ソフトバンク, 1999].

[Busc96] F. Buschmann, et al., *Pattern-Oriented Software Architecture: A System of Patterns*, John Wiley & Sons, 1996[金澤典子ほか(訳), ソフトウェアアーキテクチャ, トッパン, 1999].

[Coop00] J. W. Cooper, *Java Design Patterns: A Tutorial*, Addison Wesley, 2000 [安藤慶一(訳), Java 実例プログラムによるデザインパターン入門講座, ピアソン・エデュケーション, 2001].

[Copl92] J. Coplien, *Advanced C++ Programming Styles and Idioms*, Addison Wesley, 1992[安村通晃, 大谷浩司, 渦原 茂(訳), C++プログラミングの筋と定石, トッパン, 1994].

[Copl95] J. Coplien and D. Schmidt (eds.), *Pattern Languages of Program Design*, Addison Wesley, 1995.

[Ecke99] B. Eckel, *Thinking in Java*, Prentice Hall PTR, 1999 [安藤慶一(訳), Bruce Eckel の Java プログラミングマスターコース(上)(下), ピアソン・エデュケーション, 1999].

[Erik00] H.-K. Eriksson and M. Penker, *Business Modeling with UML: Business Patterns at Work*, John Wiley & Sons, 2000.

[Evit00] P. Evitts, *A UML Pattern Language*, Macmillan Technical Publishing, 2000.

[Fowl97] M. Fowler, *Analysis Patterns: Reusable Object Models*, Addison Wesley, 1997[児玉公信, 友野昌夫(訳), アナリシスパターン, アジソン・ウェスレイ・パブリッシャーズ・ジャパン, 1998].

[Fshu96] F. Fshull, W. Melo and V. Basili, *An Inductive Method for Discovering Design Patterns from Object-Oriented Software Systems*, Technical Report, CS-TR-3597, University of Maryland, Jan. 1996.

[Gamm94] E. Gamma, et. al., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, 1994 [本位田真一, 吉田和樹(監訳), デザインパターン, ソフトバンク, 1995].

[Gran98] M. Grand, *Patterns in Java, Vol. 1, A Catalog of Reusable Design Patterns Illustrated with UML*, John Wiley & Sons, 1998 [原 潔ほか(訳), UML を使った Java デザインパターン, カットシステム, 2000].

[Harr99] N. Harrison, B. Foote, and H. Rohnert(eds.), *Pattern Languages of Program Design 4*, Addison Wesley, 1999.

[Hata00] 畑村洋太郎, 失敗学のすすめ, 講談社, 2000.

[Iris00]入沢賢一, 青山幹雄, 分散オブジェクト環境の性能設計, 電子情報通信学会ソフトウェアサイエンス研究会, No. SS-2000-55, pp. 15-22, Mar. 2001.

[ITC00] 情報技術コンソーシアム(ITC), 次世代コンポーネントウェア技術に関する研究開発報告書, Mar. 2000.[John99] R. Johnson, 中村宏明, 中山裕子, 吉田和樹, パターンとフレームワーク, ソフトウェアテクノロジー シリーズ Vol. 1, 共立出版, 1999.

[Klei99] M. Klein, et al., Attribute-Based Architecture Style, *IFIP Conference on Software Architecture*, Feb. 1999, pp. 225-243.

[Mart97] R. Martin, D. Riehle, and F. Buschmann (eds.), *Pattern Languages of Program Design 3*, Addison Wesley, 1997.

[Mond00] P. Mond, J. Carey and M. Dangler, *San Francisco Component Framework: An Introduction*, Addison Wesley, 2000.

[Naka99]中谷多哉子, 青山幹雄, 佐藤啓太(編著), ソフトウェアパターン, 共立出版, Oct. 1999.

[Naka00] 中山裕子ほか, 連載:パターン:ソフトウェア開発のノウハウの再利用, 情報処理, Jan.-May, 2000.

[Naka01] 中川 隆, Java デザインパターンプログラミングの実際, テクノプレス, 2001.

[Pree94] W. Pree, *Design Patterns for Object-Oriented Software Development*, Addison Wesley, 1994 [佐藤啓太, 金澤典子(訳), デザインパターンプログラミング(補訂版), トッパン, 1998].

[Rich00] M. Richter and T. Suezawa, Dynamic Proxy Classes: Toward Metalevel Programming in Java, *Java Report*, Vol. 5, No. 8, Aug. 2000, pp. 32-41.

[Saik00] 斉木太郎, 青山幹雄, JavaBeans コンポーネントプレイヤ, 電子情報通信学会ソフトウェアサイエンス研究会, No. SS-2000-61, pp. 31-38, Mar. 2001.

[Schm96] D. C. Schmidt, et al (eds.), Software Patterns, *Comm. of the ACM*, Vol. 39, No. 10, Oct. 1996, pp. 36-82.

[Schm00] D. C. Schmidt, et al (eds.), *Pattern-Oriented Software Architecture: Vol. 2, Patterns for Concurrent and Network Objects*, John Wiley & Sons, 2000.

[Suzu00] 鈴木純一ほか, ソフトウェアパターン再考, 日科技連出版, 2000.

[Vlis96] J. Vlissides, J. Coplien, and N. Kerth (eds.), *Pattern Languages of Program Design 2*, Addison Wesley, 1996.

[Wool99] B. Woolf, Framework Development Using Patterns, M. E. Fayad, et al. (eds.), Implementing Application Frameworks, John Wiley & Sons, 1999, pp. 621-628.

[Yama98] 山本里枝子, パターンを用いたチーム開発のためのプロジェクト管理, 上原三八, 鯵坂恒夫(編), オブジェクト指向最前線'98ー情報処理学会 OO'98 シンポジウム, 朝倉書店, 1998, pp. 203-210.

