2005－SE－147（17）
2005／3／18

ソフトウェアパターン研究の発展経緯と最近の動向 *

鷲崎 弘宜[1]　大杉 直樹[2]　権藤 克彦[3]　服部 哲[4]　久保 淳人[5]　下滝 亜里[6]
小林 隆志[3]　藤枝 和宏[4]　大月 美佳[7]　丸山 勝久[8]　榊原 彰[9]

[1] 国立情報学研究所　[2] 奈良先端大学院大学　[3] 東京工業大学　[4] 北陸先端大学院大学　[5] 早稲田大学
[6] 大阪産業大学　[7] 佐賀大学　[8] 立命館大学　[9] 日本アイ・ビー・エム

要 旨

本稿では，ソフトウェアパターンの発展の経緯を概観し，ソフトウェア工学やプログラミング言語の分野で著名な国際会議および論文誌などで発表された論文を中心に，ソフトウェアパターンへの工学的取り組みに関する最近の話題を概説する．

Progress and Current Trends of Researches on Software Patterns

Hironori Washizaki[1]　Naoki Ohsugi[2]　Katsuhiko Gondow[3]　Satoshi Hattori[4]
Atsuto Kubo[5]　Asato Shimotaki[6]　Takashi Kobayashi[3]　Kazuhiro Fujieda[4]
Mika Ohtsuki[7]　Katsuhisa Maruyama[8]　Akira Sakakibara[9]

[1]National Institute of Informatics　[2]Nara Institute of Science and Technology　[3]Tokyo Institute of Technology　[4]Japan Advanced Instituteof Science and Technology　[5]Waseda University
[6]Osaka Sangyo University　[7]Saga University　[8]Ritsumeikan University　[9]IBM Japan

Abstract

This paper surveys the progress and current trends of the research topics regarding software patterns, which have been presented in major relevant conferences and journals.

1　はじめに

ソフトウェアパターンとは，ソフトウェア開発における特定の文脈上で繰り返し出現する問題と，その問題について実証済みの解決策，および，解決策を採るに至った複数の異なる制約や理由（それらをフォースと呼ぶ）を記述したものである．

ソフトウェアパターンへのソフトウェア工学的取り組みは，パターンカタログの電子化に端を発する．続いて，デザインパターンの普及に伴い，自動プログラミング技術の応用による単一デザインパターンの展開作業の自動化手法の提案が盛んとなった．しかしながら，デザインパターンを扱うことの難しさは，パターンをコードやモデルに展開することよりも，対象とするモデルやコードの状況・問題を理解することや適切なデザインパターンを選択することにあること [1] が認識されるに従い，デザインパターンの形式的仕様記述に基づいた自動検出や複合といった，展開作業以外のデザインパターン活用工程の技術が研究されるようになってきた．また，設計以外の種々のソフトウェア開発工程におけるパターンの発見と拡充に伴い，扱うパターンの種類も拡大されてきた．

本稿では最初に，ソフトウェアパターンへのソフトウェア工学における取り組みの発展の経緯を解説する．続いて，ここ4〜5年における主にデザインパターンに関する主要な研究技術として，パターンの条件，形式的仕様，リバースエンジニアリング，メタプログラミング／リフレクション，メトリクス，および要求工学を取り上げる．また，プロセスパターンへの工学的取り組みについても解説する．

2　パターン研究発展の経緯

2.1　ソフトウェアパターンとコミュニティ

パターンという用語は1980年代前半において，自動プログラミング技術の文脈において，アプリケーション生成系や変換システムにおけるコード生成規則 [3] を指すものとして使用されていた．この定義は，現在のパターンコミュニティにおけるパターンとは幾らか異なるものの，ソフトウェア開発の過程

* 本報告は，情報処理学会ソフトウェア工学研究会パターンワーキンググループ [2] の有志によって2004年度に実施されたパターン研究調査活動に基づく．

における過去の経験を再利用する考え方としては共通していると考えられる．

今日的なソフトウェアパターンへの取り組みは，Christopher Alexander による建築学におけるパターンランゲージに触発されて，オブジェクト指向プログラミング言語 Smalltalk を用いた GUI 設計のためのパターンランゲージの提案に端を発する [4]．以来，Hillside Group[5] を代表とするパターンコミュニティが誕生し，関連するソフトウェアパターン集合をカタログとしてまとめて出版するなどの様々な活動が継続的に行われてきた．1995 年に Gang Of Four（GoF）と呼ばれる 4 人の著者によってデザインパターン [6]（GoF デザインパターン）が発表されたのを皮切りに，ソフトウェアプロダクトおよび開発プロセス・組織に関する様々なパターンの発見がなされてきた．また，デザインパターンやプロセス・組織パターンの発展は，eXtreme Programming に代表されるアジャイル開発プロセスの出現を促した．

2.2 ソフトウェアパターン研究の経緯

ソフトウェアパターンが数多く提案されるに従い，提案済みのソフトウェアパターンを題材としたソフトウェア工学の見地からの研究が数多く行われてきた．ソフトウェア工学におけるパターンへの取り組みとは，パターンを活用したソフトウェア開発の間接または直接な自動化の実現と言い換えることができる．パターンを活用したソフトウェア開発とは，ソフトウェア開発の様々な局面において顕在化される種々の問題に対する経験をソフトウェアパターンの形で再利用することにより，熟練者によってもたらされるような優れた・誤りの少ないソフトウェアや開発組織構造・プロセスを得る作業である．

ソフトウェアパターンを活用した開発作業の自動化は，ソフトウェア工学研究やプログラミング研究において確立された種々の技術を用いてソフトウェアパターンを定義することにより発展してきた．パターンをどのような形式で定義して計算機で扱うのかによって，パターン活用作業のうちで自動化できる事柄が決まる．パターンの形式的定義の発展経緯と，その発展に付随する研究成果を表 1 に示す．

(1) 半構造文書としてのパターン

元来，ソフトウェアパターンの形式は，Alexander による建築学におけるパターンランゲージの記述形式にならって，目的や状況，問題，および解決といった項目と，その項目に対応する事柄の自然言語記述の集合として定義されてきた．例として，GoF デザインパターンの一種である Observer パターンの記述を図 1 に示す．図 1 では，Observer パターンの定義は，名前，文脈，問題，解決，関連パターンの 5 つの項目より構成されている．これらの項目の種類や意味は，パターンの種類や記述する作者によってまちまちであるため，もともとのソフトウェアパターン文書は半構造文書ということができる．

ソフトウェアパターンを単なる半構造文書として扱うならば，計算機によるパターン活用の支援は，

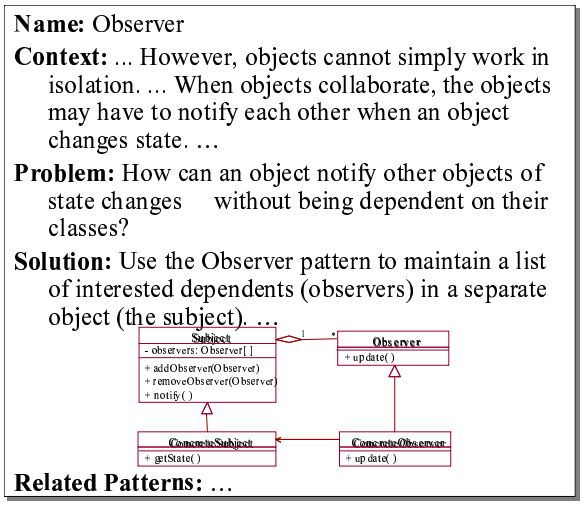

図 1: Observer パターンの記述文書

キーワード検索や単純な関連付けに基づく集積といった，一般的な文書処理技術に基づいた技術に限定される．以上の理由から，パターンを活用した開発自動化の成果は最初に，大小様々なソフトウェアパターンカタログの電子化として現れた．1995 年に Web 上でのソフトウェアパターンリポジトリ Portland Pattern Repository（PPR）[7] が登場し，PPR は世界最大のパターンリポジトリとして継続的に発展してきた．また，PPR を発展させる形で新たなパターンリポジトリ PatternShare[8] が 2005 年に公開されている．それらのパターンリポジトリは電子的パターンカタログとして機能し，新たなパターンの集積や，開発者が直面する問題に関連するパターンの検索・比較検討といった作業を支援する．

(2) ロール／クラス集合としてのデザインパターン

開発自動化の取り組みは，カタログの電子化に続いて，自動プログラミング技術を応用することにより，主にデザインパターンのより直接的な再利用作業の自動化について行われてきた．具体的には，1990 年代後半から 2000 年代初頭にかけて，GoF デザインパターンが解決として与える構造にのみ着目して，同パターンの特定のオブジェクト指向プログラミング言語環境に基づく自動的な実装コードの生成 [9] や，モデリングツールの機能拡張による設計モデルデザインパターンの自動的な適用 [10] について，活発に取り組まれてきた．その成果は，今日において代表的な商用モデリングツール・プログラミング環境の多くにおいて，デザインパターン展開機能として実用に至っている．

GoF デザインパターンは，ライブラリあるいは単体として具体的なオブジェクト指向クラス集合を再利用するのではなく，特定の役割（ロール）を持った複数のクラスの組み合わせの仕方の再利用を与える．典型的な展開機能においては，個々のデザインパターンが解決として与える構造にのみ着目して，事前にその構造を，関連付けられたロールの集合（図 1 におけるクラス構造）として定義しておき，各ロールを展開先のモデル／コード中の構成要

表 1: パターンの形式定義と研究発展の経緯

時期	対象パターン	定義	研究	元となる技術
1995–	全て	自然言語，半構造文書	カタログの電子化	文書処理
–1999	デザイン	オブジェクト指向クラス集合	コード生成	自動プログラミング，OOP
2000–	デザイン	形式的仕様，関心事	コード／モデル生成，検出	仕様記述，AOP
2000–	プロセス	自然言語，モデル表現など	検査，関連解析	仕様記述，文書処理

素（クラスやインタフェース）に対応付ける形で展開を実現する．

(3) 仕様が記述されたデザインパターン

固定されたロール集合としてのデザインパターン定義は，パターンが与える最終的な固定された解法の再利用のみを実現し，デザインパターン本来の目的である”設計に至った理由も含めた意思疎通”[11]を実現しなかった．ソフトウェア設計活動は，対象とする元の設計状況の理解や，変更時の構成要素間の調整といった，開発者による判断を必要とする高度で複雑・知的なものである．その思考プロセスを計算機によって支援するには，デザインパターンが与える設計モデル上の構造や振る舞いの意味をより厳密に定義することや，パターンを適用した後にその事実を後から振り返って確認可能とすることなどが求められる．

そこで 2000 年代に入ると，形式的仕様記述法やメタプログラミング技術を活用することで，個々のデザインパターンの仕様を厳密に定義する技術や，形式的仕様に従ってデザインパターンを既存のモデル／コードから検出する技術，および，もとの設計モデル／コードとデザインパターンに関わるモデル／コードをそれぞれ別個に管理・統合する技術などが活発に提案されてきた．これらの個々の要素技術については，以降の章において詳しく述べる．

(4) 対象パターンの種類の拡充

同じく 2000 年代に入るとパターン活用活動の自動化は，デザインパターン以外の他のパターン（プロセスパターンや組織パターンなど）についても幾らか取り組まれるようになった．プロセスパターンへの取り組みについて以降の章で詳説する。

2.3 パターン活動スパイラルモデル

パターンを扱う活動は，パターンの抽出活動とパターンの利用活動の 2 つから構成される [12]．両活動のプロセスを図 2 に示す．両活動はそれぞれ，継続した取り組みによって，関連するパターン集合の抽出／利用を実現することが大切であるため，図 2 ではスパイラルとして描いている．

抽出活動とは，ソフトウェア開発に関する知識から，パターンとして再利用可能な知識を発見し，特定の形式にしたがって記述し，組織内または組織外のコミュニティにおいて洗練し，洗練後に得られたパターンが実際に有効であることを異なる開発プロジェクトなどにおいて評価する活動である．利用活動とは，開発者がソフトウェア開発において直面する問題と文脈を理解し，その状況の解決に役立ちそうなパターンをコミュニティにおける同意を得て形成されたパターン集合（カタログ）から選択し，利用者の重視する制約に合わせた修正と拡張を経てパターンを適用し，適用した結果について調整と評価を行う活動である．

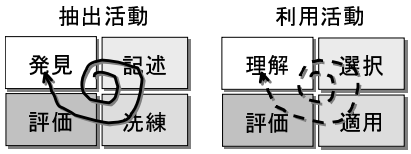

図 2: パターン活動プロセスのスパイラル

しかしながら，抽出と利用の両活動は独立していない．両活動の関連した様子を，2 重螺旋モデルとして図 3 に示す．図 3 は，以下の事柄を表す．

- 抽出または利用活動に続いて，抽出または利用活動の実施: パターンは，単体では存在し得ないと指摘される [13]．パターンを単体ではなく，続いて関連するパターン候補を発見する様子や，抽出したパターンを利用する様子，あるパターンの適用後に生じる状況と問題に役立つ別のパターンの利用を検討する様子などを表す．
- 抽出活動と利用活動が表裏一体: 両活動を構成する各工程間の活動をまたいだ強い関連を示す．例えば，対象の問題や状況を理解する工程において，新たなパターン候補を発見できる可能性がある．
- 両活動の工程を支援するパターン活動支援技術: 各工程を支援するパターン活動自動化の成果を楕円で表す．例えばメトリクス技術は，両活動の評価工程を支援する．

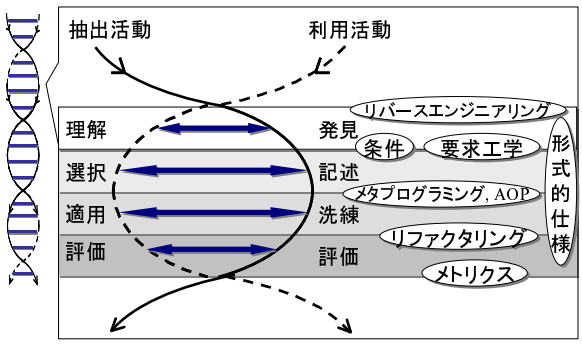

図 3: パターン活動プロセスの 2 重螺旋構造

3　パターン活動支援技術

以降では，上述のパターン活動を支援するパターン研究の最近の成果を技術単位で解説する．

3.1　ソフトウェアパターンの条件

前章で述べたように，パターンのもともとの定義が曖昧であることが，パターンを工学的／科学的に扱うことの主要な困難さである．そこで，既存の，あるいは新たに記述しようとするパターンが，正しくパターンであることの条件の確立について古くから取り組まれてきている．これまでに良く知られる条件として，パターンが文脈・フォースの体系・ソフトウェア構成の 3 つ組を成すという文書表現上の条件 [14] や，3 回以上利用されているという利用実績上の条件 [15]，および，パターンがレビューを受けていることを含めた基準 [16] などがある．

しかしながら，上述の条件が真に妥当なものであるとの保証はない．実際に，主要なパターンの多くは，条件の幾つかを満たさない．そこで最近では，Winn らが，パターンの多様さに起因してパターンの必要条件を見出すことは難しいことを指摘して，必要条件の代わりに，ソフトウェア設計のための設計パターンが備えるべき以下の 9 つの本質的特性を提案している [17].

- ソフトウェア／ドメインの設計を暗示する
- 複数の抽象レベルにおける表現を結びつける
- 機能的特性と非機能的特性の両者を扱う
- 得られるソフトウェア設計中に痕跡が残る
- ホットスポットを捉える
- パターンランゲージの一部を形成する
- 繰り返し利用によって有効性が確認済み
- 特定のドメインに根ざしている
- 設計上の巨視的な方針を捉える

ただし，これらの特性を備えていることをもって，対象とする知識文書が絶対的に設計パターンであるとは主張していない．論文では，特性一覧を用いて，既存のソフトウェア設計に関するパターンの幾つかをパターンあるいはパターンではないものと区別することを試みている．”パターンらしさとは何か”という根源的な課題への取り組みとして興味深い．

3.2　パターンの形式的仕様に基づく検出・検査・理解・選択支援

ソフトウェアの形式的仕様とは，意味が一意に定まるように数学的に厳密に記述された仕様である．形式的仕様記述法は，仕様中に数学的な誤りが存在しないことの検証や，プログラムの自動生成などに広く応用されてきている．パターン研究の分野では，プログラムの状況理解を主な目的として，デザインパターンに形式的に記述された仕様を与えることで，プログラムコード中でその仕様を満たす部分をデザインパターンとして検出することが試みられてきた [18, 19]．また最近では，同じく形式的仕様記述を応用することで，パターンに基づいて得られるモデルの自動検査や，デザインパターンの正確な理解の支援，および，パターンの選択の支援が試みられている．

Konrad らは，組込みシステム開発の分析工程において用いる幾つかのオブジェクト分析パターンが与える分析モデルの雛形について，その状態遷移の仕様を時相論理式によって記述しておくことで，それらのパターンの雛形に基づいて得られたモデルをモデル検査器 SPIN によって自動的に検査する方法を提案している [20]．高品質なソフトウェアを得るための有望な技術が自動検証であり，その成果をパターン指向開発に応用した試みとして興味深い．

Soundarajan らは，従来非形式的に記述されていたデザインパターンに対して，設計者がそのパターンを適用する際に満たさなければならない要求を正確かつ曖昧のないように理解するための仕様の形式化手法を提案している [21]．提案する仕様は，標準的な契約による設計 (DbC: Design by Contract) に類似した記法により形式化されており，責任を担う部分と報酬を担う部分で構成されている．責任とは，クラスの構造，そのクラスの特定のメソッドの振る舞い，クラス間のやり取りに関して設計者が守らなければならない条件を指す．報酬とは，パターンを適用した際に受け取る恩恵（具体的には，責任に含まれるすべての要求が満たされる場合に外部に公開されることが保証されている特別な振る舞い）を指す．論文では，Observer パターンを例題とし，パターン全体の仕様，パターンに参加している個々の役割に関する形式的仕様（図 4）を示している．図 4 では，Observer パターンを構成するロールとして 1 つの Subject 役と任意数の Observer 役をそれぞれ定義し，それらのロール間に成立する関係や制約を述語や論理式を用いて宣言している．例えば，`auxiliaryConcepts` 句において以下の関係や制約を宣言している．

- Subject 役の状態 $\alpha\sigma$ と，Subject 役に依存する Observer 役の状態 $\alpha\sigma$ が一貫している
- Subject 役の状態 $\alpha\sigma_2$ は，状態 $\alpha\sigma_1$ から変化したものである
- Subject 役の状態 $su\alpha\sigma_1$ と $su\alpha\sigma_2$ が同一で，かつ，$su\alpha\sigma_1$ と $ob\alpha\sigma$ が一貫している場合は，$ob\alpha\sigma$ は $su\alpha\sigma_2$ についても一貫している

パラメータ化された上述の付属概念を用いることで，パターンの仕様は，それぞれのインスタンスに対してテイラーリングする余地を残し，柔軟性を損なわずに正確性を追求することができる．

Mehlitz らは，コンポーネントのプログラムコードについて，その利用のための制約条件や保証される特性に関する形式的仕様記述を付加したモジュールを対象ドメインに特化したパターン（D4V パターンと呼ぶ）として記述して，再利用する手法を提案している [22]．D4V パターンは，保証される特性を用いて索引付けされる．解決されるべ各システム要求について，各 D4V パターンが保証する特性との合致を調査することで，適切な D4V パターンを選択する．論文では例として，イベント多重化コン

```
pattern Observer {
 role: Subject, Observer*;
 state:
  Subject: set[Observer] _observers;
  Observer: Subject _subject;
  pattern: null;
 auxiliaryConcepts:
  relation:Consistent(Subject.α σ,Obsever.α σ);
  relation:Modified(Subject.α σ1,Subject.α σ2)
  constraint:
   [¬Modified(su α σ1,suα σ2 )
    ∧Consistent(su α σ1,obα σ)]
   ⇒Consistent(su α σ2,obα σ)
```

図 4: Soundarajan らによる Observer パターンの形式的仕様記述の一部 [21]

ポーネントに対して，優先順位付きイベント処理・非ブロック・非同期が要求され，特性を満たす D4V パターンが選択される様子が示されている．

3.3 リバースエンジニアリングと新デザインパターン抽出

ソフトウェア工学におけるリバースエンジニアリングとは，プログラムの理解・保守や特定部分の再利用を目的として，既存のプログラムを解析する技術である．パターン研究の分野では，リバースエンジニアリングを活用して，既存のプログラムから共通に出現するプログラムコードをパターンとして抽出することや，リバースエンジニアリング活動そのものをパターンカタログにまとめることなどが試みられてきている．

Tonella らは，一般的な関係（継承や関連，メソッド呼び出しなど）を共有するクラス群を識別するために，コンセプト分析（Concept Analysis）手法を用いて既存のソースコードからオブジェクト指向設計パターンを発見する手法を提案している [23]．プログラムコードに対して，その内部に存在するクラスとクラス間の関係を，提案するアルゴリズムに基づき複数のコンセプトという単位に分類する．分類したコンセプトにおいて同値性が見られるものをまとめることで，最終的なパターンを発見している．論文では，25000LOC 程度の C++アプリケーションに対して，提案手法を用いてパターン検出の評価実験を行っている．発見されたパターン（コード中の共通の特徴と呼ぶべきもの）に対する意味付けを放棄しているものの，多様なプログラムコードに自動的に適用できる点が興味深い．

さらに Arevalo らは，Tonella らのクラス間関係に基づくコンセプト分析手法 [23] について，半順序集合（ラティス）を考えることで，クラス発見手法の効率化とパターン間の協調関係の定義に成功している [24]．コンセプトの構築には，サブクラスかどうか，抽象クラスかどうか，利用関係にあるかどうかの 3 つの特性を用いる．各コンセプトにおけるクラスをノード，クラス間の関係をアークとする Intent Relation Graph に対して，すべてのノードが結合されているコンセプトだけを残す．さらに，等価なコンセプトを 1 つにまとめることにより，従来手法 [23] よりも探索空間を削減している．さらに，コンセプトの半順序関係（包含関係）に基づいた幾つかの協調関係を導入することで，類似パターンの発見を実現している．パターンの発見のみならず，パターン間の関連までを含めた抽出を行っている点が興味深い．

Niere らは，既存のプログラムコードからパターンを発見する際の問題を，実装におけるバリエーションと捉えて，ASG（Abstract Syntax Graph）の導入によりその問題を解消し，パターン発見の精度を向上させる手法を提案している [25]．ASG とは，ソースコードの構成要素を頂点，それらの間に成立する関係（属性としての所有やメソッドとしての所有など）を辺で表現した有向グラフである．ASG に対して特定のグラフとのマッチングを行うことで，パターン（サブグラフ）を見つける．パターンが見つかった場合に，関連するノードに対して，パターン名を記した注釈を付加する．パターンの発見作業者は，この注釈を見ながらさらなるグラフマッチングの適用を考え，最終的にアプリケーション内部のパターンを識別していく．パターンの抽出に ASG を用いる点と，利用者とのインタラクションを前提としている点が興味深い．

[26] では，Demeyer らが実際に体験したリエンジニアリングの経験が，リエンジニアリングパターンのカタログとして文書化されている．リエンジニアリングとは，既存のソフトウェア資産をリバースエンジニアリングによって解析して，設計上の問題を調査し，その問題を修復する（フォワードエンジニアリング）活動である．デザインパターンが設計のみを扱うのに対して，リエンジニアリングパターンはコード，設計，ドキュメント等の多様な対象を扱う．パターンカタログとしてまとめることにより，ソフトウェア工学的活動を分かりやすく・導入しやすくすることに成功している．

3.4 メタプログラミング／リフレクションによるデザインパターンの扱い

プログラミング言語研究の発展に伴い，プログラムの拡張性や生産性の向上を目的として，プログラム生成・操作系のプログラミング技術（メタプログラミング）や，リフレクション技術を発展させて，要求を（しばしば横断的な）関心事の集合に分割して関心事の単位でプログラムを作成・合成するアスペクト指向プログラミング技術（AOP）などが研究されてきた．パターン研究の分野では，これらの主に実装プログラミング言語上の生成・合成の仕組みを応用して，デザインパターンの独立した実装と合成について取り組まれてきた．

[27] では，動的オブジェクト指向プログラミング言語 CLOS によって GoF デザインパターンを明示的に実装する手法が提案されている．従来の実装手法では，クラス間の関連はコード自身で表現されており，パターンの表現が明示的でないという欠点を持つ．この論文では，CLOS の静的なメタプログラミングの仕組みを活用して，各デザインパターンに

ついて実装コードの生成規則をメタクラスとして事前に定義し，そのメタクラスの再利用によって，デザインパターンの実装を明示的に用意して再利用することに成功している．

Hannemann らは，GoF デザインパターンの多くが，構成するロールやロールを担当する実装クラスが対象プログラム内で横断的に出現する性質に着目して，アスペクト指向プログラミング言語 AspectJ を用いてデザインパターンを実装する手法を提案し，AspectJ を用いた実装によってモジュール性（局所性，再利用性，合成透過性，およびプラグイン可能性）の改善が得られるかどうかを評価している [28]．この論文では，23 個の GoF デザインパターンのうち 17 個についてモジュール性の改善が得られることを示している．AspectJ を用いた Observer パターンの実装例を図 5 に示す．図 5 では，Observer パターンの核となる仕組みを `ObserverProtocol` アスペクト，そのアスペクトを具体的に対象プログラムに展開する仕組みを `FigureObservation` アスペクトとしてそれぞれ別個に定義している．これらのアスペクトを実際に展開した場合に得られる設計の例を図 6 に示す [29]．デザインパターンの核となる部分の実装の独立した再利用，および，依存性の逆転（展開対象プログラムがデザインパターンに依存するのではなく，パターンの展開方法を定義したアスペクトが展開対象に依存すること）を実現している．

```
public abstract aspect ObserverProtocol {
 protected interface Subject  { }
 protected interface Observer { }
   protected abstract pointcut
     subjectChange(Subject s);
 after(Subject s): subjectChange(s) {
  Iterator iter = getObservers(s).iterator();
  while(iter.hasNext())
   updateObserver(s,(Observer)iter.next()); }
 protected abstract void
  updateObserver(Subject s, Observer o);

public aspect FigureObservation extends
                          ObserverProtocol {
  declare parents: Line implements Subject;
  declare parents: Display implements Observer;
  declare parents: Chart implements Observer;
  protected pointcut subjectChange(Subject s):
  call(void Line.setColor(Color)) && target(s)
```

図 5: Hannemann らによる Observer パターンの AspectJ による定義の一部 [28]

3.5 リファクタリングとデザインパターン

リファクタリングとは，プログラムの外部に対する本質的な振る舞いを保ったままで，理解性や保守性などの向上を目的とした，プログラムの内部構造を変更する作業である．デザインパターンの適用は，同様に非機能的特性の向上を目的としたプログラム設計を促すため，リファクタリングと密接な関係にある．そこで，両者の関係を詳細に明らかにすることが試みられてきている．

Tokuda らは，ソフトウェアが発展する過程にお

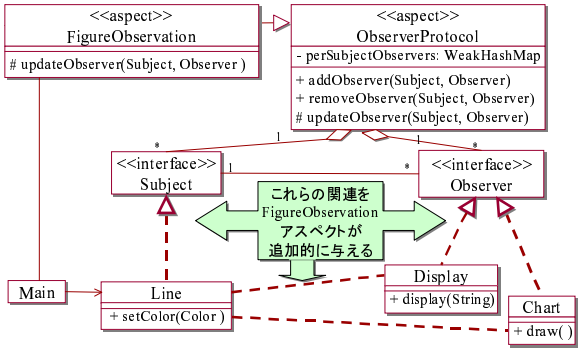

図 6: Observer パターンのアスペクトの展開例 [29]

いてデザインパターンが設計の目標を与えていると仮定し，基本的なリファクタリングを繰り返し適用することで，既存プログラムの設計を改良する手法を提案している [30]．提案しているリファクタリングは，パラメータ化された Ingerit（親子関係を構築），Factory Method（インスタンスを生成するメソッドを追加），Substitute（あるクラスへの参照から別のクラスへの参照への置換），CreateClass（新規クラスの作成），および，CreateInstanceVariable（新規インスタンス変数のクラスへの追加）である．論文では，これらの基本変換を用いて，プログラムコードがデザインパターンを有する形に推移する様子が具体的に示されている．

[31] では，既存の設計（あるいはコード）を改良するために，デザインパターンを標的とした 27 個のリファクタリングをカタログ形式で紹介している．紹介されているリファクタリングは，既存コードを特定のパターンに置き換えるもの，既存コードを特定のパターンに近づけるもの，特定パターンから離れるものに分類されている．リファクタリングとパターンとの関係をカタログとして整理している点で，ソフトウェア開発コミュニティへの貢献度は大きい．

3.6 メトリクスとデザインパターン

メトリクスとは，高品質なソフトウェアを開発し運用するために，ソフトウェアや開発プロセスの備える品質特性が要求される度合いに達しているかどうかを判定する品質測定法（メトリック）の集合である．これまでパターンのコミュニティでは，パターンが与える品質上の”良さ”は，言葉では表せないものであるとして QWAN（Quality without a Name）と呼ばれていた．そこで，デザインパターンの特性や与える影響を定量的に調査し評価するために，メトリクス技術が用いられてきている．

[32] では，適切なデザインパターンに基づく設計の方が，場当たり的な設計に比べて欠陥が少ないことを仮定して，商用アプリケーションにおける欠陥率の測定によりその仮定が成り立つかどうかを分析した結果が述べられている．測定においては，スケーラビリティと精度を十分に考慮して実装されたシステムを構築している．実験では，5 つのパターン (Singleton，Template Method，Decorator，Observer，Factory) とそれらの組み合わせに対し

て，ロジスティック回帰分析を適用して次の事柄を報告している．

- 一般的なコードと比較して，Factory パターンを適用したコードは欠陥率が低いが，Observer パターンを適用したコードは欠陥率が高い
- Singleton パターンと Observer パターンにおいては，それらのパターンのサイズの増加と欠陥頻度に相関が見られる
- Template Method パターンに関しては，さまざまな状況において何度も適用されているため，欠陥頻度に関する明確な傾向は見られない

この報告は，デザインパターンを適用すると設計が良くなるという主張に対して，それが事実あるいは神話なのかを示す一つの指標になる．

一方，Bieman らは，デザインパターンが与える可変性を調査した結果を報告している [33]．論文では，デザインパターンが与える可変性は，パターンのロールを担うクラス（パターンクラス）そのものの変更ではなく，サブクラスの追加・修正によって実現されるものと仮定して，幾つかの実用プログラムの改変履歴を調査したところ，実際には，パターンクラスの方がより変更される傾向にあり，先の仮定は当てはまらなかったことを報告している．

また，メトリクス技術を，新たな設計パターンの発見に活用する試みもある．設計パターンを特定する作業は，複雑なグラフにおいてパターンを見つけることに等しく，可能なクラスの組み合わせとプログラムのサイズという点から一般的に困難である．そこで [34] では，ソフトウェア設計の基調（モチーフ）において，特定の役割を担うクラスの外部的特性（メソッドやフィールドの数，親子関係，凝縮度，結合度）に基づく数値（フィンガープリント）を導入している．いくつかのアプリケーションに対して，この数値を機械的に学習させることで，デザイン基調において特定の役割を担うクラスおよびクラス群が持つべき数値を割り出している．得られた数値を用いて，特定のデザイン基調において明らかに役割を担っていないクラスを探索空間から取り除くことで，パターンの発見工程において，より重要な役割を担うクラスを特定している．さらに，この数値はデザイン基調を用いて実現されたプログラムの品質特性を定量的に評価する際の指標となることが期待できる．メトリクスに基づく数値を，探索空間の削減としてパターン発見の初期段階に適用することで実用性を狙っている点が興味深い．

3.7 要求工学技術の応用

要求工学とは，要求の定義や獲得といった要求分析活動を支援する工学的技法の分野である．最近では，この分野における主要な成果であるゴール指向分析法を，パターンの表現や選択に応用することが試みられてきている．ゴール指向分析法とは，要求の具体化を目的として，要求を，要求の達成のためのゴールへと段階的に分解・詳細化する手法である．

Ong らは，パターン文書中に記述されたフォース（解決に至った理由や考慮すべき事柄の記述）を非機能的特性に関する要求として捉え，非機能要求群をゴール指向分析法を用いて表した枠組みを用いて，フォース間の対立や協調・導出といった関係を網羅したフォース階層図を作成することで，既存のパターンの理解を支援し，さらに，パターンランゲージの洗練を支援する手法を提案している [35]．例えば図 7 では，”Home and Visitor Databases”パターンについて，考慮する要求を分解・詳細化していく過程においてフォース間（”Attacking ...” と ”More hardware ...”）のトレードオフの関係を分かりやすく表している．

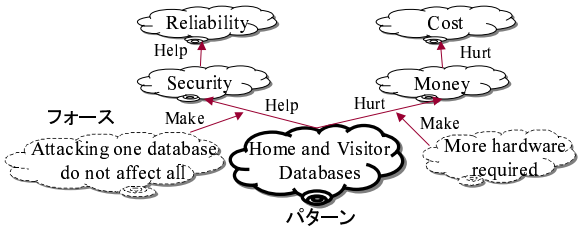

図 7: フォース間関係の表現例 [35]

また Hsueh らは，デザインパターンについて品質とその影響の観点から分析したモデルを用意し，ゴール指向分析法の適用によって得られたサブゴールのうちで，非機能要求の達成やトレードオフの関係にあるサブゴールの解消のための設計に役立つデザインパターンを選択する手法を提案している [36]．

3.8 プロセスパターンの扱い

パターンの概念を開発プロセスの知識・経験再利用に応用して，ソフトウェアを開発するための実証済みの一連の活動手順を表すものがプロセスパターンである．最近ではプロセスパターンの広まりと共に，ソフトウェア工学的技法をプロセスパターンの活用に応用する動きが出てきた．

Dittmann らは，従来多く提案されてきたプロセス記述言語に類似する形でプロセスパターン記述言語を策定し，その言語によってプロセスパターンが解決する問題やパターン間の関係などを定義することにより，記述内容の曖昧さを減少させることに成功している [37]．また，[38] では，効率的な開発プロセスに共通して見られる経験的法則を表した 2 つのプロセスパターンを取り上げて，それらに数学的な裏付けを与えることを試みている．具体的には，開発プロセスにおける情報交換を確率ペトリネットでモデル化し，確率ペトリネットの定常状態解析によって，取り上げたプロセスパターンが情報交換の効率について妥当な解決策を与えることを確認している．これらの試みは，自然言語で記述されたプロセスパターンについて，他の形式に基づく定義を与えるという点について共通している．

一方，自然言語で記述されたままのプロセスパターンについて，文書処理技術を応用する試みもある．[39] では，文書処理技術に基づくパターン間関

連解析手法 [40] を応用して，複数のプロセスパターン間の自動的な関連解析を試みている．解析の結果，異なるパターンカタログに属するプロセスパターンの間について，カタログをまたいだ新たな役立つと考えられる関連を抽出することに成功している．

4 おわりに

本稿では，最近のソフトウェアパターン研究から主要な技術を幾つか取り上げて解説した．また，ソフトウェアパターン活動を構成するプロセスのモデル化，および，各プロセスとパターン支援技術の対応付けを試みた．ここまで取り上げたように，ソフトウェアパターンの活動支援に応用された技術（およびパターンの概念を導入した分野）は広範にわたって存在する．今後も，ソフトウェアパターン研究はソフトウェア工学やプログラミング研究といった研究分野における様々な成果を取り込むと同時に，パターン研究の成果が他の分野に影響を与えるといった相互に作用した発展が期待される．その成果は様々な会議や論文誌において”横断的に”発表されるため，パターン研究の分野の進展を概観するためには，今後も本稿と同様に様々な情報源についての継続的な調査が必要である．

謝辞 山波良博氏（立命館大学）より調査研究のご協力をいただきました．また，古宮誠一先生（芝浦工業大学）よりパターンの概念について有益なご指摘をいただきました．ここに深く感謝いたします．

参考文献

[1] C. Chambers, et al.: A Debate on Language and Tool Support for Design Patterns, Proc. POPL’00

[2] http://patterns-wg.fuka.info.waseda.ac.jp/

[3] 大野, 阿草: ソフトウェア工学からみた自動プログラミング, 情報処理, 28(10), 1987

[4] K. Beck and W. Cunningham: Using a Pattern Language for Programming, ACM SIGPLAN Notices, 23(5), 1987

[5] http://hillside.net/

[6] E. Gamma, et al.: Design Patterns: Elements of Reusable Object-Oriented Software, Addision-Wesley, 1995

[7] http://c2.com/ppr/

[8] http://patternshare.org/

[9] F. Budinsky et al.: Automatic code generation from design patterns, IBM Systems Journal, 35(2), 1996

[10] T. Kobayashi et al.: Object-Oriented Modeling of Software Patterns, Proc. ISPSE2000

[11] K. Beck and R. Johnson: Patterns Generate Architectures, Proc. ECOOP’94

[12] 鷲崎, 深澤: ソフトウェアパターン研究の現在と未来, 情処研報, 2003(SE-141)

[13] F. Buschmann et al.: Pattern-Oriented Software Architecture: A System of Patterns, John Wiley & Sons, 1996

[14] D. Alur et al.: Core J2EE Patterns: Best Practices and Design Strategies, Pearson Education, 2001

[15] W. Tracz: RMISE Workshop on Software Reuse Meeting Summary, In Software Reuse: Emerging Technology, IEEE CS, 1988

[16] W. Brown et al.: The Software Patterns Criteria: Proposed Definitions for Evaluating Software Pattern Quality, 1998, http://www.antipatterns.com/whatisapattern/

[17] T. Winn and P. Calder: Is This a Pattern?, IEEE Software, 19(1), 2002

[18] C. Kramer and L. Prechelt: Design Recovery by Automated Search for Structural Design Patterns in Object-Oriented Software, Proc. WCRE’96

[19] D. Heuzeroth et al.: Automatic Design Pattern Detection, Proc. Workshop on Program Comprehension at ICSE’03

[20] S. Konrad et al.: Object Analysis Patterns for Embedded Systems, IEEE Trans. Soft. Eng., 30(12), 2004

[21] N. Soundarajan and J.O. Hallstrom: Responsibilities and Rewards: Specifying Design Patterns, Proc. ICSE’04

[22] P.C. Mehlitz and J. Penix: Design for Verification Using Design Patterns to Build Reliable Systems, Proc. 6th CBSE, 2003

[23] P. Tonella and G. Antoniol: Object Oriented Design Pattern Inference, Proc. ICSM’99

[24] G. Arevalo et al.: Detecting Implicit Collaboration Patterns, Proc. WCRE’04

[25] J. Niere et al.: Towards Pattern-Based Design Recovery, Proc. ICSE’02

[26] S. Demeyer et al.: Object-Oriented Reengineering Patterns, Morgan Kaufmann, 2002

[27] D. Dincklage: Making Patterns Explicit with Metaprogramming, Proc. GPCE’03

[28] J. Hannemann and G. Kiczales: Design Pattern Implementation in Java and AspectJ, Proc. OOPSLA’02

[29] 羽生田 監修, 金澤, 井上, 森下, 鷲崎，佃, 細谷，瀬戸川, 山野, 沖田 著: ソフトウェアパターン, ソフトリサーチセンタ，2005（出版予定）

[30] L. Tokuda and D. Batory: Evolving Object-Oriented Designs with Refactorings, Automated Software Engineering, 8(1), 2001

[31] J. Kerievsky: Refactoring to Patterns, Addison-Wesley Professional, 2004

[32] M. Vokac: Defect Frequency and Design Patterns: An Empirical Study of Industrial Code, IEEE Trans. Soft. Eng.，30(12), 2004

[33] J.M. Bieman et al.: Design Patterns and Change Proneness: An Examination of Five Evolving Systems, Proc. Metrics 2003

[34] Y. Gueheneuc et al.: Fingerprinting Design Patterns, Proc. WCRE’04

[35] H. Ong et al.: Rewriting a Pattern Language to Make it More Expressive, Proc. ChiliPLoP 2003

[36] N. Hsueh and W. Shen: Handling Nonfunctional and Conflicting Requirements with Design Patterns, Proc. APSEC 2004

[37] T. Dittmann et al.: Improved Support for the Description and Usage of Process Patterns, Proc. SDPP’02

[38] S. Hattori and K. Ochimizu: A Mathematical Foundation to Validate Some Empirical Organizational Patterns, Proc. CITSA 2004

[39] H. Washizaki et al.: Relation Analysis among Patterns on Software Development Process, Proc. PROFES 2005（to appear）

[40] A. Kubo et al.: Analyzing Relations among Software Patterns based on Document Similarity, Proc. ITCC 2005（to appear）