パターン適用前のソースコードを用いたデザインパターン検出

鷲崎弘宜　深谷和宏　久保淳人　深澤良彰

設計の理解支援のためオブジェクト指向プログラムからのデザインパターン検出手法が提案されている．しかしながら，デザインパターン適用後のソースコードに対する静的解析を用いた従来手法では検出精度が低い．本稿では，デザインパターン適用前のソースコードの静的解析を併用して検出精度を高めた検出手法を提案する．提案手法により，構造が類似するデザインパターン群を高精度に判別し，保守目的の参照者の設計理解を支援することができる．

Detecting design patterns from object-oriented program source-code can help maintainers understand the design of the program. However, the detection precision of conventional approaches based on the structural aspects of patterns is low due to the fact that there are several patterns with the same structure. To solve this problem, we propose an approach of design pattern detection using source-code of before the application of the design pattern. Our approach is able to distinguish different design patterns with similar structures, and help maintainers understand the design of the program more accurately.

1　はじめに

デザインパターン（以下，単にパターンと表記する）とは，ソフトウェア設計における頻出の問題，その解法や考慮した事柄をまとめた記述である [1]．各パターンはそれぞれ特定の設計意図を持つため，プログラムで適用されているパターンをあらかじめ識別できれば，保守目的の参照者は設計を容易に理解できる．しかし，大規模で複雑なプログラムにおけるパターン適用を，洩れなくかつ誤りなく人手で判断することは困難であるため，プログラムからパターンを自動的に検出する手法が提案されている．特に，オブジェクト指向プログラムから，オブジェクト指向設計のための GoF パターン [1] を検出する手法が様々に提案されており，本稿ではその検出を扱う．

従来のパターン検出手法の多くは静的解析に基づき [2] [3]，パターン適用後のクラス構造やメソッドの限定的な振る舞いを解析するため，クラス構造が互いに類似するパターンの組や，特徴的なクラス構造を持たないパターン†1の判別に失敗する場合がある．例えば，パターンの解決が与えるクラス構造を UML クラス図で表現した図 1 上部の State パターン [1] の目的は，オブジェクトの内部状態が変化したときに振る舞いを変えることである．一方，図 1 下部の Strategy パターン [1] は，State パターンと同じクラス構造を持つが，その目的はアルゴリズムをカプセル化して，利用するクライアントからは独立に変更可能にすることである．このように設計意図は全く異なるが，構造の解析だけでは判別できないパターンもある．

パターン検出精度が低い場合，保守者に設計意図が適切に伝わらないという問題を生じる．上記とは別に，静的解析と動的解析を併用したパターン検出手法 [4] も提案されている．しかし，一般に動的解析の精

Detecting Design Patterns Using Source Code of Before Applying Design Patterns.

Hironori Washizaki, 早稲田大学, 国立情報学研究所 GRACE センター, Waseda University, National Institute of Informatics. Kazuhiro Fukaya, 早稲田大学, Waseda University. Atsuto Kubo, 国立情報学研究所, National Institute of Informatics. Yoshiaki Fukazawa, 早稲田大学, Waseda University.

コンピュータソフトウェア, Vol.26, No.YY (2010), pp.ZZ–ZZ.
[レター論文] 2009 年 12 月 22 日受付.

†1　例えば Facade パターンのクラス構造は，パターン適用の意図を考慮しなければ多数の個所で出現する．

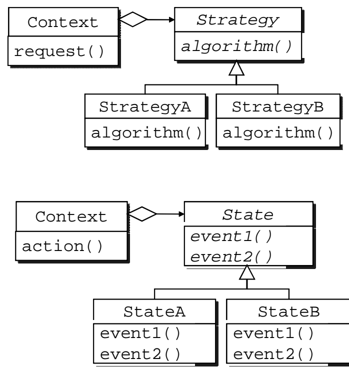

図1 解決のクラス構造：Strategy（上部）と State（下部）

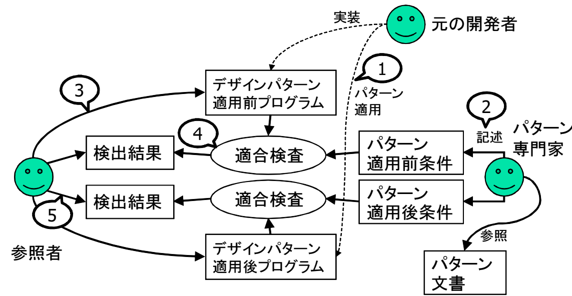

図2 提案手法の全体像と流れ

度は実行経路の網羅性に依存し，適切なテストデータを用意することは困難である．また，動的解析を用いた手法は解析における時間コストが高く，かつ実行可能なプログラムが必要となる．

そこで本稿では以降において，パターン適用後のソースコードに加えて適用前のソースコードを活用し，高精度なパターン検出を支援する手法を提案し，ケーススタディの結果を述べる．

2 パターン適用前条件を用いたパターン検出

適用前のソースコードには，パターンを適用可能な箇所があると想定し，当該個所を識別する条件を各パターンについて事前にまとめることで検出に利用する．本稿ではこの条件を「パターン適用前条件」と呼ぶ．ただし，パターン適用前条件を満たせば必ずパターンを適用できるわけではない．また，パターン適用前条件は，従来のパターン検出手法においてパターン適用後のプログラムが満たすことを検査する検出条件（パターン適用後条件）に替わるものではなく，併用することで高精度にパターンを検出する．具体的には，クラス構造が互いに類似するパターンや特徴的なクラス構造を持たないパターンの検出が可能になり，保守者の的確な設計理解を支援できる．

提案手法の全体像を図2に示す．下記の手順により提案手法が適用される．

1. 開発者（元の担当者）は，リファクタリング等による設計改善の過程でパターンを適用する．そのような改善におけるパターン適用が推奨されており[5]，提案手法はこれを前提とする．
2. パターンの専門家は，パターンの記述文書を参照し，パターン検出に利用可能な形式でパターン適用前条件とパターン適用後条件を記述する．
3. 保守目的で参照する者は，パターン適用前後の各条件に対する検査対象のソースコードを準備する．取得手段として，バージョン管理システム等で管理されている場合と，未管理ながら改善履歴が残っている場合が考えられる．
4. 参照者は，ソースコードに対してパターンの適用前・適用後条件に対する適合性を検査する．
5. 参照者は検査結果に応じて，クラスのパターンにおける役割などの対応関係を検討し，妥当であればパターンを検出したと判定する．

以降において，検出手順，適用前条件の導出方法，State/Strategy パターン [1] の判別例を示す．

2.1 パターン検出手順

検査の方向に基づき検出手順として2種を想定し，順行法，逆行法と呼ぶこととする．バージョン管理システムの利用を想定した各手順を以下に示す．

順行法（図3上部）

1. 種々のパターン適用前条件との適合検査を，古いバージョンから新しいバージョンに向かって行う．あるパターンについてパターン適用前条件と適合しなくなったバージョンを Ver.K とする．
2. Ver.K-1 と Ver.K について，当該パターンの適用後条件に対する適合性を検査する．

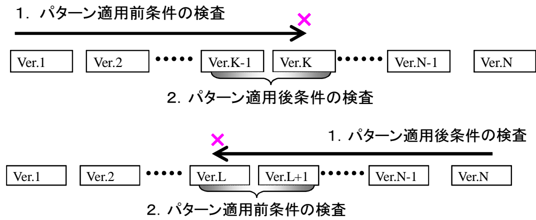

図 3 順行法（上部）と逆行法（下部）

3. Ver.K のみがパターン適用後条件に適合する場合は，Ver.K-1 から Ver.K へと改訂する際に当該パターンが適用された可能性が高い．そこで，両バージョンにおけるプログラム要素（クラスやインタフェース）の対応や変更の関係を調査し，当該パターンの適用を最終的に確認する．
   一方，Ver.K-1 と Ver.K の共に適用後条件に適合しない場合は，Ver.K について当該パターンが適用されている可能性が少なからず存在すると考えられるため，Ver.K について当該パターンの適用の有無を人手で確認し最終判定する．

逆行法（図 3 下部）

1. 種々のパターンのパターン適用後条件との適合検査を，最新バージョンから過去に向かって行う．あるパターンについて，パターン適用後条件と適合しなくなったバージョンを Ver.L とする．
2. Ver.L と Ver.L+1 についてパターン適用前条件との適合性を検査する．
3. Ver.L のみパターン適用前条件と適合する場合は，Ver.L から Ver.L+1 へと改訂する際に当該パターンが適用された可能性が高い．そこで，両バージョンにおけるプログラム要素（クラスやインタフェース）の対応や変更の関係を調査し，当該パターンの適用を最終的に確認する．
   一方，Ver.L と Ver.L+1 の共に適用前条件に適合しない場合は，Ver.L+1 について当該パターンが適用されている可能性が存在すると考えられるため，Ver.L+1 について当該パターンの適用の有無を人手で確認し最終判定する．

このように提案手法は，従来のパターン適用後のプログラムを対象とした検出手法を併用し，従来手法を単独で用いる場合と比較して効率よく，漏れをできるだけ防ぐ形でパターンを検出することを支援する．

検出対象パターンや箇所が定まっており，従来手法により正確な判別は困難であるものの当該パターンの可能性を含む形で検出が可能な状況では，逆行法が効率的で有用である．一方，順行法では従来手法における第 2 種の過誤（実際はパターンを適用しているが検出に失敗）を防ぐことができる可能性がある．具体的には，従来は全く着目していなかった箇所について，パターン適用の有無を再検討する機会を得られる．

2.2 パターン適用前条件の導出

適用前条件は，パターン記述文書の特定項目を参照して導出する．代表的なパターンカタログ（[1] [6] [7] など）の形式は未統一ながら，Canonical 形式 [7] を基として，作者が必要に応じて項目を追加，削除，改名している．表 1 に Canonical，GoF [1]，PoSA [6] の各形式の項目間の対応関係を示す．優先度は，適用前条件の導出における項目参照の優先度を示す．

「問題」には，そのパターンの意図や扱える設計課題が述べてある．「文脈」には，そのパターンを適用できる状況，パターンが扱う可能性のある粗悪な設計例とその認識方法が示されている．「フォース」には，そのパターンを適用する際に考慮すべき制約条件について述べてある．主に，上記三点を参照してパターン適用前条件を作成する．「解決」には，問題に対する解決策や設計の具体例が示されており，パターン適用後条件の作成に利用できる．

「関連パターン」は，関連あるいは共用の可能性の高いパターンへの言及である．従ってあるパターンの適用後に，その「関連パターン」を適用できる可能性がある．しかしながら，その場合には関連パターンの適用後においても，依然として関連元の当該パターンが適用されていることが多いため，前節に示した検出手順では検出困難である．そのため，本稿では関連パターンに関する情報は扱わない．

2.3 GoF パターンの検出例

提案手法により Java プログラムのソースコード中から GoF パターンを検出する例を示す．従来の静的

表 1 カタログ形式とパターン適用前条件の導出優先度

Canonical 形式	内容	優先度	GoF 形式	PoSA 形式
名前	名前	×	名前	名前
別名	別名	×	別名	別名
問題	解決しようとする問題	○	意図	課題
文脈	パターンを適用できる状況	○	適用可能性	前提
フォース	問題を解決するときに考慮すべき点	○	動機	課題
解決	問題に対する解決策	○	構造，協調関係など	解決策など
例	例	△	サンプルコード	例
結果文脈	パターン適用によって何がどうなるか	△	結果	結論
根拠	問題, 文脈, 解を支持する根拠や事例	×	（該当なし）	（該当なし）
適用事例	実際の使用例	×	使用例	適用例
関連パターン	組み合わせて使えるパターンなど	△	関連するパターン	参考

○: 参照すべき，△: 参考にできる可能性あり，×: 不要あるいは直接には無関係

解析による手法では判別できなかった State パターンと Strategy パターンを検出の対象とする．

(1) State パターンの適用前条件の導出

State パターンの適用可能性には『オペレーションが，オブジェクトの状態に依存した多岐にわたる条件文を持っている場合．この状態はたいてい 1 つ以上の列挙型の定数で表されており，度々複数のオペレーションに同じ条件構造が現れる』とある [1].

そこで，以下の 3 点を適用前条件として導出した．

- C_1: `Context` ロールを担うクラス内のメソッドが，フィールドに依存した複数の条件文を持つ．
- C_2: `Context` が状態遷移にあたり，C_1 のフィールドの値を変更している．
- C_3: C_1 のフィールドが取る値は全て定数によって表されている．

(2) Strategy パターンの適用前条件の導出

Strategy パターンの適用可能性には『クラスが多くの振る舞いを定義しており，これらがオペレーション内で複数の条件文として現れている場合．このとき，多くの条件文を利用する代わりに，条件分岐後の処理を `Strategy` クラスに移し換える』，協調関係には『`Context` オブジェクトは，クライアントからの要求を `Strategy` のオブジェクトに送る．クライアントは，通常 `ConcreteStrategy` オブジェクトを生成し，これを `Context` オブジェクトに渡す』とある [1]．また，各アルゴリズムをカプセル化した `ConcreteStrategy` はクライアントにより選択され，その選択指示なく他の `ConcreteStrategy` に処理が移ることはない．

そこで，以下の 2 点を適用前条件として導出した．

- $C_4(=C_1)$: `Context` 内のメソッドが，フィールドに依存した複数の条件文を持つ．
- C_5: `Context` 自身によって C_4 のフィールドの値が変更されることはない．

(3) State パターンと Strategy パターンの判別

両パターンの検出対象として，文献 [5] におけるの State パターン適用前後の Java プログラムソースコードを用いる．State パターン適用前の具体的な対象ソースコードの抜粋を図 4 に示す．また，同ソースコードに対して State パターンを適用した結果を UML クラス図として図 5 に示す．

これらの対象に対して，提案手法によるパターン検出を試みる．両パターンの判別が目的であり，両パターンの可能性を含む形で検出可能な従来の静的解析手法（[2] [3] など）が存在するため，逆行法を用いる．

最初に，従来手法を用いてバージョンが後のほうの図 5 に対応したソースコードを解析すると，State パターンあるいは Strategy パターンが適用されていると判定されるが，両者は判別されない．さらに，バージョンが前のほうのパターン適用前のソースコード（図 4）に対して同じく従来手法を用いると，いずれのパターンも検出されない．

続いて，図 4 のソースコードについて両パターンの適用前条件の適合性を検査する．前述した State パターン適用前条件の適合性を検査すると，それぞれ以下のように満たされることが分かる．

- C_1: クラス `SystemPermission` 内のメソッド

```
public class SystemPermission ... {
 public void claimedBy(SystemAdmin admin) {
  if (state == REQUESTED) {
   state = CLAIMED;
  } else if (state == UNIX_REQUESTED){
   state = UNIX_CLAIMED;
  } ...
 }

 public void grantedBy(SystemAdmin admin) {
  if(profile.isUnixPermissionRequired()
   && state == UNIX_CLAIMED) {
   isUnixPermissionGranted = true;
  }else if(profile.isUnixPermissionRequired()
   && !isUnixPermissionGranted()) {
   state = UNIX_REQUESTED;
   notifyUnixAdminsOfPermissionRequest();
   return;
  }
 ...
```

図 4 State パターン適用前の検出対象のコード

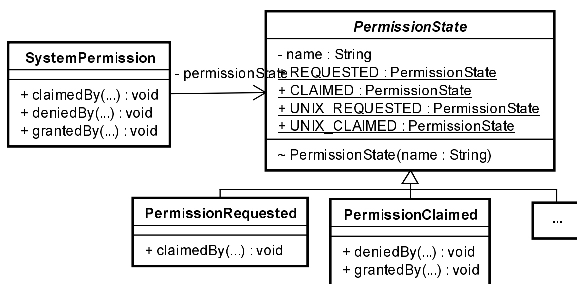

図 5 State パターン適用後の検出対象のクラス構造

claimedBy および grantedBy が，フィールド state に依存した条件文を持つ．ここで図 5 より，SystemPermission は Context 役を担うクラスと識別できる．よって満たされる．

- C_2: 上述の各メソッド内でフィールド state の値を変更している．よって満たされる．
- C_3: state の取りうる値は全て定数（REQUESTED, UNIX_REQUESTED, CLAIMED, UNIX_CLAIMED）で表されている．よって満たされる．

さらに，同じコードについて Strategy パターンの適用前条件を検査すると，以下のように条件 C_5 について満たされないことが分かる．

- C_4: C_1 と同一条件であり，同様に満たされる．
- C_5: claimedBy および grantedBy において，複数の条件文が依存するフィールド state の値を変更している．よって満たされない．

以上より，図 4 のソースコードは State パターンの適用前条件のみを満たしており，従ってその後のコードの状況を表す図 5 について State パターンが適用されているとの判定に成功した．

2.4 実装と制限

Java プログラムソースコードについて，パターン適用前条件との適合性を検査するツールを実装した．ソースコードを JavaML [8] により XML 形式文書に変換し，必要な情報を取り出して検査する（図 6）．

```
String path = "test/binary-expr"; //条件文
NodeIterator nb =
  XPathAPI.selectNodeIterator(nodeIf, path);
while((nodeb = nb.nextNode()) != null) {
 if(((Element)nodeb).getAttribute("op").
  equals("==")) { //条件判定が==
  path = "var-ref[1]"; //条件判定における変数
```

図 6 コードからの XPath を用いた情報抽出（抜粋）

提案手法の制限として，適用前条件の識別は属人性のある作業であり，条件の制約の強さが最終的な検出における正確性と網羅性に影響する．適用前条件の制約が強いほど第 2 種の過誤が増え，弱いほど第 1 種の過誤（実際はパターンを適用していないが，適用有りと判定）が増える．第 2 種の過誤増加は，パターン適用前条件を用いる利点を小さくする．第 1 種の過誤増加は順行法において，参照者が人手によりパターン適用の有無を調査する機会を増大させてしまう．ただし提案手法では，条件識別時の参照個所として優先順位を伴ってパターン文書中の項目群を挙げることで，識別作業を支援し属人性を低減させている．

3 ケーススタディ

State パターン適用前後の Java プログラム対 4 組 [5] [9] [10] [11] と，State パターンを伴わない変更前後

の Java プログラム対 3 組[5] [10] [12] （各 500 行以下程度）について，従来手法[2] [3] による State パターンの検出，および，提案手法の逆行法により State パターンの検出を試みた．結果を表 2 に示す．表中の「判別」は State パターンのみを判別して検出,「未判別」は State パターンと Strategy パターンを判別せず検出,「無し」はパターンの未検出を示す．

State パターン適用済み，かつ，従来手法で State パターンと Strategy パターンを判別せずに検出したプログラムについて，提案手法により State パターンの正確な判別に成功した．また，State パターン未適用，かつ，従来手法で State パターンと Strategy パターンを判別せずに検出したプログラムについて，提案手法により State パターンを適用していないことを示すことができた．しかしながら両者とも，逆行法を用いたため従来手法でパターンを検出できない場合には提案手法も無効となる．このような場合には，順行法を検討するべきである．

以上より，従来手法とパターン適用前条件の併用によって，検出精度が向上することを確認した．

表 2　State パターン検出（T: Tsantalis の手法[3]，F: FUJABA [2]，P(x): x を組み入れた提案手法）

	State 適用済み 4 件			State 未適用 3 件		
手法	判別	未判別	無し	判別	未判別	無し
T	0	3	1	0	3	0
P(T)	3	0	1	0	0	3
F	0	1	3	0	1	2
P(F)	1	0	3	0	0	3

4　おわりに

本稿では，パターン適用前におけるソースコードを用いたパターン検出手法を提案した．提案手法により，実験対象のプログラムに関して従来手法単独よりもパターン検出の精度が向上することを示した．

今後の課題として，オープンソースプロジェクトなどの実際的な一定規模以上のプログラム群に対する検出評価，他のパターン群の適用前条件の識別（例えば Command パターン[1] についてその適用可能性の記述より「クラス内のメソッドが引数に依存した条件文を持つこと」と識別できる），適用前条件の導出と定義の形式化などが挙げられる．

本稿の適用前条件は，広義の「不吉な匂い」[13] を特定のパターンに特化させた形と捉えることもできる．不吉な匂いを各種メトリクスで特定する試み[14] があり，提案手法におけるパターン適用前条件の導出の手がかりとすることを今後検討したい．また，開発履歴からのリファクタリング検出手法（[15] など）を併用して，パターン適用前後におけるプログラム要素の対応関係調査を支援することを今後検討したい．

謝辞 助言をいただいた査読者に感謝します．本研究の一部は (財) みずほ学術振興財団の助成を受けました．

参 考 文 献

[1] E. Gamma, et al. 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, ソフトバンククリエイティブ, 1999.
[2] J. Niere, et al. Towards pattern-based design recovery. Proc. 24th International Conference on Software Engineering, 2002.
[3] N. Tsantalis, et al. Design pattern detection using similarity scoring. IEEE Transactions on Software Engineering, Vol.32, No.11, 2006.
[4] D. Heuzeroth, S. Mandel, and W. Lowe. Generating design pattern detectors from pattern specifications. Proc. 18th IEEE International Conference on Automated Software Engineering, 2003.
[5] J. Kerievsky. Refactoring to Patterns. Addison-Wesley, 2004.
[6] F. Buschmann, et al. Pattern-Oriented Software Architecture, Wiley, 1996.
[7] http://c2.com/cgi/wiki?CanonicalForm
[8] G. Badros. JavaML: a markup language for Java source code. Proc. 9th International World Wide Web Conference on Computer Networks, 2000.
[9] E. Freeman, E. Freeman, B. Bates, and K. Sierra. Head First Design Patterns. O'Reilly, 2004.
[10] R. Martin. Agile Software Development: Principles, Patterns, and Practices. Prentice Hall, 2002.
[11] S. Metsker. Design Patterns Java Workbook. Addison Wesley Professional, 2002.
[12] H. Kabutz. Strategy pattern with generics. http://www.javaspecialists.co.za/
[13] M. Fowler, Refactoring: Improving the Design of Existing Code. Addison Wesley, 1999.
[14] M. Munro. Product metrics for automatic identification of bad smell design problems in Java source-code. Proc. 11th International Software Metrics Symposium, 2005.
[15] P. Weissgerber and S. Diehl. Identifying Refactorings from Source-Code Changes. Proc. 21st IEEE/ACM international Conference on Automated Software Engineering, 2006.