宣言型メタプログラミングによる不吉な臭いの要因検出手法

伊藤雄貴[†1]　森本康彦[†1]　中村勝一[†2]　宮寺庸造[†1]

概要:ソフトウェア開発において，リファクタリングを行い不吉な臭いの要因を除去するプロセスは必要不可欠である．しかし，膨大な量のソースコードからソフトウェア開発者が自力で不吉な臭いの要因を見つけ出す作業は大変困難なものである．そこで本研究では Java を対象に，宣言型メタプログラミングによってソースコード内から不吉な臭いの要因を検出する手法を提案した．さらに提案手法を Eclipse プラグインとして実装した．これにより，ソフトウェア開発者に対してソースコード内に含まれる不吉な臭いの要因を提示することが可能となり，リファクタリング作業の効率向上が期待できる．

1. はじめに

拡張や保守が容易に行える良質なソフトウェアを開発する上で，リファクタリングによってソースコード内の不吉な臭いの要因を除去するプロセスは必要不可欠である．

しかし，膨大な量のソースコードからソフトウェア開発者が自力で不吉な臭いの要因を見つけ出すには莫大な時間と労力を必要とし，その作業は大変困難なものである．この問題を解決するために，ソースコード内に含まれる不吉な臭いの要因を検出し，その要因と，それに対して効果的なリファクタリング手法をソフトウェア開発者に提示するツールが必要である．

ソースコード内から不吉な臭いの要因を検出する手法に関する研究は既に数多く行われている[1][2][3][4]．しかし，特定種類の不吉な臭いを対象にした研究が大部分を占めており，多種多様な不吉な臭いを扱った研究は少ない．

このような背景を踏まえ本研究の目的は，汎用的なプログラミング言語である Java を対象にソースコード内から多種多様な不吉な臭いの要因を検出する手法を提案することである．提案手法では，対象とするソースコードと不吉な臭いの要因をそれぞれ宣言型メタプログラミングによって表現し，双方を対比することでソースコード内から不吉な臭いの要因を検出する．さらに，提案手法を Eclipse プラグインとして実装する．これにより，ソフトウェア開発者に対してソースコード内に含まれる不吉な臭いの要因と，その要因に対して効果的なリファクタリング手法を提示することが可能となり，リファクタリング作業の効率向上が期待できる．

2. 手法の検討

ソースコードから不吉な臭いの要因を検出する手法として，宣言型メタプログラミングの利用が挙げられる[3][4]．宣言型メタプログラミングとは，ベースとなる言語で構成されたプログラムを推察，改修するために宣言型プログラミング言語（特に論理型プログラミング言語）をメタレベルで用いる技法である[5]．これを用いれば多種多様な不吉な臭いの要因を論理プログラムという統一された手法で記述することができると考えられる．そのため本研究では，ソースコードから不吉な臭いの要因を検出するために宣言型メタプログラミングを用いることとした．この際に用いるメタ言語としては Prolog を採用する．Prolog は述語理論に基づいたプログラミング言語であり，そのプログラムは論理的事実を表す Fact という述語と，論理的規則を表す Rule という述語で構成される．

多種多様な不吉な臭いの要因を検出するためには，ソースコードの詳細な構造に着目する必要がある．ソースコードの詳細な構造情報を得る手法としては，抽象構文木の利用が挙げられる．抽象構文木とは，各ノードにシンボル名や型情報が付与された構文木である．この抽象構文木を論理型プログラミング言語で表現することで，ソースコードの詳細な構造情報を宣言型メタプログラミングによって推察することが可能となる．

提案手法の概要を図 1 に示す．提案手法では対象となる Java のソースコードを Prolog の Fact で，不吉な臭いの要因を Prolog の Rule で表現し，これらの Fact と Rule に対してクエリーを発行することによって，不吉な臭いの要因を検出する．

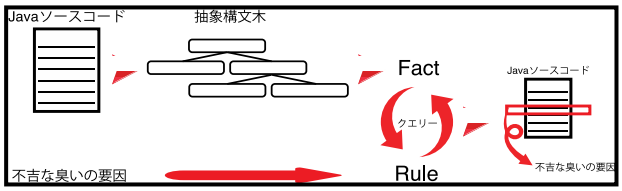

図 1　提案手法の概要

3. Prolog による不吉な臭いの要因検出手法

3.1 Fact によるソースコードの構造の表現

提案手法では，対象となるソースコードから生成される抽象構文木を Prolog の Fact で表現する．抽象構文木を Fact で表現するにあたり，抽象構文木の各ノードやノード間の関係に対応する Fact の雛形を定めた．ソースコードは抽象構文木の解析を経てこの雛形に則った Fact によって表現される．

抽象構文木の各ノードに対応する Fact は，不吉な臭いの要因検出時に一意に特定する必要がある．そのため，抽象構文木の各ノードに対応する Fact には一意な ID を付与す

†1 東京学芸大学
Tokyo Gakugei University
†2 福島大学
Fukushima University

る．不吉な臭いの要因として検出された Fact はこの ID によって識別される．

代表的な Fact の雛形を表 1 に示す．表中の X，Y は ID を意味する変数である．提案手法では，対象となるソースコードの抽象構文木を走査することで表のような各ノードやノード間の関係に対応する Fact を生成する．

表 1　代表的な Fact の雛形

Factの雛形	概要
block(X).	ノードXがブロックである.
ifStatement(X).	ノードXがif文である.
whileStatement(X).	ノードXがwhile文である.
methodInvocation(X).	ノードXがメソッド呼び出し文である.
numberLiteral(X).	ノードXが数リテラルである.
body(X, Y).	ノードXがノードYの本体部である.
statements(X, Y).	ノードXがノードYのステートメントである.
thenStatement(X, Y).	ノードXがノードYのthen部である.
type(*type*, X).	ノードXの過多情報が*type*である.

3.2 Rule による不吉な臭いの要因の表現

前述した抽象構文木を表現する Fact と対比するために，不吉な臭いの要因は，Prolog の Rule によって表現する．この Rule は不吉な臭いの要因となるようなソースコードの構造を表現するものであり，前節で述べた Fact 間の論理的関係の記述により構成される．

本研究で扱う不吉な臭いの要因は文献[6], [7]を基に決定した．文献で述べられるリファクタリング手法の適用動機に不吉な臭いの要因が出現していると考え，これらを Rule によって表現する．

例として，「1 つのメソッドの中で，ローカル変数に情報を累積している」という不吉な臭いの要因に対する Rule を図 2 に示す．図のように，不吉な臭いの要因を表す Rule は，抽象構文木のノードを表す Fact と，それらの Fact 間の関係の記述によって構成されており，不吉な臭いの要因を含むようなソースコードの構造を表現している．

```
excessOfAccumulation(X) :- inLoopstatement(X), accumulationMethodInvocation(X).
inLoopstatement(X) :- inWhileStatement(X); inForStatement(X); inEnhancedForStatement(X).
inWhileStatement(X) :- ancestor(Y, X), whileStatement(Y).
inForStatement(X) :- ancestor(Y, X), forStatement(Y).
inEnhancedForStatement(X) :- ancestor(Y, X), enhancedFroStatement(Y).
accumulationMethodInvocation(X) :- methodInvocation(X), ( method("List<*>.add(*)", X);
                    method("List<*>.addHead(*)", X); method("List<*>.addEnd(*)", X); ...).
parent(X, Y) :- body(Y, X); expression(Y, X); thenStatement(Y, X); statements(Y, X); ...
ancestor(X, Y) :- parent(X, Y); parent(Z, Y), ancestor(X, Z).
```

図 2　不吉な臭いの要因を表現する Rule の例

3.3 クエリーによる不吉な臭いの要因検出

本研究で提案する手法では，ソースコードの構造を表す Fact と，不吉な臭いの要因を表す Rule に対して Prolog のクエリーを発行する．クエリーを発行することで Rule を満たす Fact を全て検出し，検出された Fact を基に抽象構文木を辿ることでソースコード中の不吉な臭いの要因を検出する．

4. 手法の実装

本研究で提案する手法に基づき，ソースコード内から不吉な臭いの要因を検出するツールを開発した．本ツールは Eclipse のプラグインとして実装した．本ツールを Eclipse にインストールすると図 3 に示すような独自のビューが追加される．ビュー内のプラグイン実行ボタンを押すと，編集中のソースコードを対象に上記のような仕組みで不吉な臭いの要因検出が行われる．検出結果として，検出された不吉な臭いの要因の概要とソースコード内での位置（行番号），及びその要因に対して有効なリファクタリング手法がビューに表示される（図 3 下部）．これにより，リファクタリング作業の支援となる．

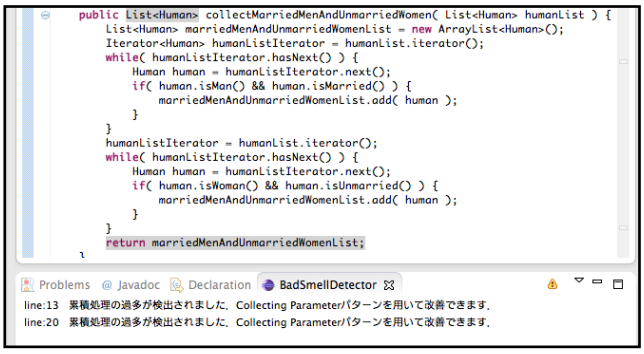

図 3　Eclipse プラグインの動作画面例

5. まとめと今後の課題

本論文では，宣言型メタプログラミングを用いてソースコード内から不吉な臭いの要因を検出する手法を提案した．

また，提案手法に基づき，不吉な臭いの要因を検出するツールを統合開発環境 Eclipse 上で動作するプラグインとして開発した．これにより，ソースコード内の不吉な臭いの要因と，その要因に対して有効なリファクタリング手法をソフトウェア開発に提示することが可能となり，リファクタリング作業の支援となる．

今後は，本論文で提案した Rule 表現の妥当性を検証する必要があると考える．また，本論文で提案した手法を多岐にわたるソフトウェアに対して適用し，その効果を検証する．

参考文献

1)　村松裕次，中川晋吾，出口博章，水野忠則，太田剛，酒井三四郎：リファクタリング箇所特定支援のためのパターン記述，情報処理学会論文誌 46(12) 3054-3065 (2005)

2)　三宅達也，肥後芳樹，井上克郎：ソフトウェアメトリクスとメソッド内の構造を用いたリファクタリング支援手法の提案，電子情報通信学会技術研究報告 SS2008 - 13～26 Vol.108 No.173 pp.73 – 78 (2008)

3)　J.Rajesh, D.Janakiram: JIAD A Tool to Infer Design Patterns in Refactoring, ACM SIGPLAN International Conference (2004)

4)　Tom Tourwe, Tom Mens: Identifying Refactoring Opportunities Using Logic Meta Programming, Software Maintenance and Reengineering, Proceedings. Seventh European Conference (2003)

5)　Johan Brichau, Kim Mens: Declarative Meta Programming, http://soft.vub.ac.be/research/Old - DMP/

6)　Martin Fowler: Refactoring Improving the Design of Existing Code, Addison Wesley Professional (1994)

7)　Joshua Kerievsky: Refactoring to Patterns, Addison Wesley Professional (2004)