Java を対象としたソースコードインスペクションツールの開発
—検査仕様記述方法の提案—

M2010MM010 堀田淳司
指導教員：野呂昌満

1　はじめに

ソフトウェアインスペクションとは，仕様書やプログラムなどの成果物を静的に検査することで，誤りや不具合を発見する手法である．我々は，Java を対象としたソースコードインスペクションツール，Java Code Inspector(以下，JCI) の開発を行なってきた [2]．JCI は，Java ソースコードの中から不具合が生じる可能性のある箇所を指摘するツールである．我々は，開発の長期化や要求の変化に対応するために，保守性や再利用性を重視し，Product Line Software Engineering に基づいたアーキテクチャ中心開発を行なってきた．JCI の保守開発では，検査項目の追加や修正が頻繁に行なわれており，これらの変更と共にソフトウェアテストを実施している．JCI では，検査項目と検査項目に対するテストケースは同じ仕様から作成しており，これを検査仕様と呼ぶ．現在の JCI の検査仕様は自然言語を使って複数の警告例を示すことによって記述されている．実現コードの記述とテストケースを作成するには，抽象構文木やフローグラフの知識だけでなく JCI のシステム固有の処理を理解する必要がある．

本研究の目的は，検査仕様記述方法を提案し，検査仕様を改善する．検査仕様を明確にすることで，ユーザによる検査処理論理の記述と，その検査処理論理に対応したテストケースの作成を支援する．

本研究は，抽象構文木やフローグラフの解析処理を言語処理問題として検査仕様を記述する．検査項目として考慮したい特定の構文要素だけに着目し走査する．着目した構文要素の走査順に基づき検査仕様を状態遷移機械で記述する．フローグラフの処理においては基本的に変数の定義と使用等を検査するものなので，この方法が適合する．一方，抽象構文木の処理は，文脈自由文法の記述による処理に対応するものなので，プッシュダウンオートマトンの処理が必要になる．しかし，スタックを扱うのではなく，着目した構文要素に対する入れ子構造のカウントなどの簡易処理に置き換えることで，検査仕様は有限オートマトンの処理で記述できると考えた．抽象構文木とフローグラフの処理を有限オートマトンの処理に帰着することで，検査仕様を統一的に記述できる．

我々は，検査仕様の記述で必要な情報を整理するために，データ構造に対する処理方法を分析する．検査仕様記述の手がかりとして，静的解析ツール基盤 [1] で採用されている検査仕様記述 (以下，Metal) を利用する．Metal の基本操作だけでは検査仕様を記述するのは困難と判断し，Metal の拡張を行なう．検査仕様を定義し，状態遷移機械で表現した検査処理からテストケースを生成するためのアーキテクチャ設計を行なう．

2　関連研究

JCI の検査仕様記述方法の提案にあたり，我々は既存のインスペクションツールである CX-Checker[3] の仕様記述方法を調査した．CX-Checker は，ユーザが容易に検査項目の追加・変更可能な言語を持っている．抽象構文木などの情報を XML で表現することで，検査処理論理を XPath 式または DOM 式で記述できる．加えて，検出したい箇所のコード例から，それをマッチさせるための問い合わせ式を導出する機能を備えるなど，ユーザによる検査処理の実現を容易にしている．しかし，CX-Checker は構文解析結果の情報しか持たないので，制御フローを用いた検査の実現について十分な支援ができていない．

XPath 記述には，条件を満たす要素のみを取得する述語や関数が使用できる．関数には，文字列が指定された文字列から始まるか判断するものや，文字列の長さを返すものがある．CX-Checker における XPath 記述は，抽象構文木を表わした XML に対して問い合わせ式を記述する．また，DOM 式の記述では，XML を DOM で操作する Java プログラムを記述して検査を実現している．

3　JCI 概要

JCI は，Java のソースコードに対して静的解析を行ない，欠陥の可能性がある構文や制御フロー，データフローのパターンを発見，指摘するツールである．JCI の処理は，入力されたソースコードから抽象構文木を構築し，その抽象構文木に対して制御フロー解析，データフロー解析を行なうことでフローグラフを構築する．各検査処理は，ソースコードに対する解析結果の抽象構文木やフローグラフを走査することで，欠陥の可能性がある箇所を指摘する．

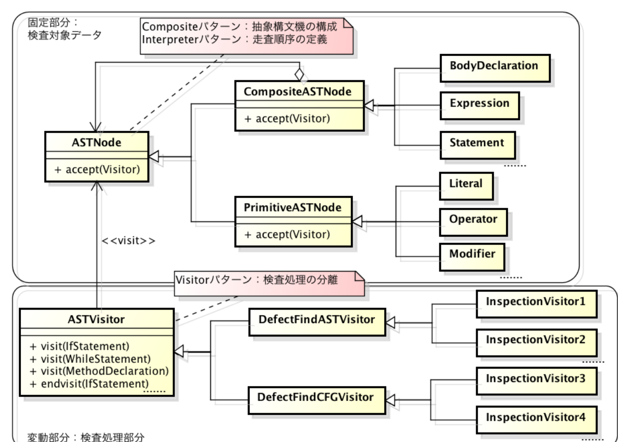

図 1 JCI のソフトウェアアーキテクチャ

JCI は変更に対して柔軟に対応できるよう，固定部分のデータ構造と変動部分の検査処理に分けてアーキテクチャ設計されている．JCI のアーキテクチャを図 1 に示す．抽象構文木の再帰的な静的構造は，Composite パターンを適用している．各構文要素に対する走査処理は Interpreter パターンを適用し，CompositeASTNode クラスにその子要素に対する走査順序を集約することで，検査実行時に必要な走査処理の再利用を行なっている．検査処理の実現には，Visitor パターンを適用する．抽象構文木やフローグラフに対する走査順序は Interpreter パターンによって実現されているので，検査処理のアルゴリズムを構文要素から分離して Visitor クラスに集約している．また，検査処理の追加や変更を，構文要素や他の検査処理の振る舞いに影響を与えずに行なうことができることから，保守性や追加変更に対する柔軟性を確保している．

4 検査項目の仕様記述

4.1 検査処理の実現

現在の JCI における検査処理の実現には，抽象 Visitor のサブクラスに構文要素の種類に応じた検査処理内容を記述する必要がある．構文要素の情報やフロー解析の結果情報は各構文要素が保持しており，それらの利用方法を把握していなければ検査処理の実現は難しい．

JCI は，Visitor による行きがけ順走査や帰りがけ順走査で構文要素の出現や終了に着目している．検査仕様を有限オートマトンで記述するにあたり，構文要素の出現や終了をイベントとして利用し，特定の構文要素の入れ子の深さを把握する処理をアクションとして実現する．検査処理を正規表現とアクションルーチンを使って実現することで検査処理内容の記述は容易になる．

状態遷移機械で検査処理内容を表現するには，構文要素の出現や終了イベントだけでなく特殊なイベントが必要となる．例えば，プログラムの終了イベントや入れ子を管理するイベントである．

4.2 フローグラフに対する検査仕様記述

データフローグラフに対するイベントの生成には，データフローを辿る Visitor を作成する必要がある．制御フローグラフの各ノードを辿った際，構文要素の種類のイベントと，変数に着目して使用や消滅のメッセージが存在すればこれらをイベントとして生成する．

データフローに対する検査項目の例として，「メソッド仮引数の値を上書きする箇所の検出」がある．この検査処理はデータフローを対象としており，メソッド仮引数の変数がメソッド内で上書きされる場合に警告する．この検査項目の状態遷移機械を図 2 に示す．

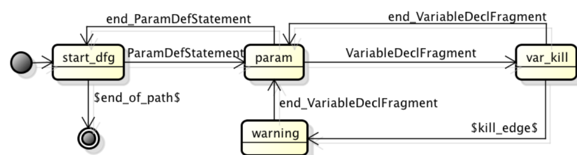

図 2 メソッド仮引数の値を上書きする箇所の検出

この検査項目の警告例を次に示す．

警告例

```
public void method(String str){
  if(str.equals(""))
    return;
  str = null; // 警告
}
```

この警告例では，start_dfg 状態の時にパラメータ宣言文が出現するので，param 状態に遷移する．仮引数が出現するので var_kill 状態に遷移する．代入文から仮引数に対して消滅イベントが生成されるので，warning 状態に遷移し，代入文を警告する．警告後，パラメータ宣言文終了イベントで start_dfg 状態に遷移し，全てのパラメータ宣言文の検査が終わったら終了する．

4.3 抽象構文木に対する検査仕様記述

抽象構文木に対する検査項目の例として，「後置式を含む if 文の条件式を検出」がある．この検査項目は，if 文の条件式に後置式が存在した場合に警告する．この検査項目の状態遷移機械を図 3 に示す．

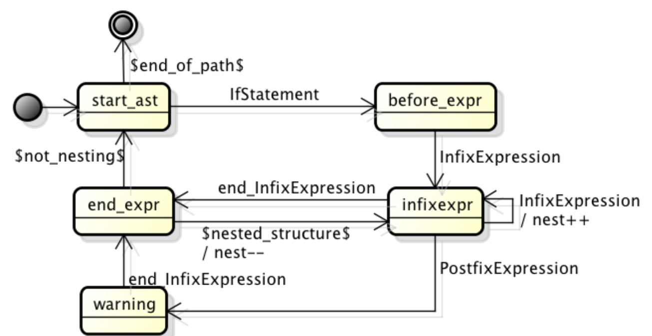

図 3 後置式を含む if 文の条件式を検出

この検査項目の警告例を次に示す．

警告例

```
if(i != 0)
 if(true || i++ < 0){ // 警告
  ...
 }
```

この警告例では，start_ast 状態の時に if 文が出現するので，before_expr 状態に遷移する．条件式は中置式なので infixexpr 状態に遷移する．条件式に後置式が存在しないので中置式終了イベントで end_expr 状態に遷移する．ネストを表現する変数は 0 なので start_ast 状態に遷移する．2 つ目の if 文の出現で before_expr 状態に遷移する．この条件式は条件式全体が中置式であり，右辺も中置式のネスト構造である．まず，条件式全体の中置式で infixexpr 状態に遷移する．左辺では遷移せず，右辺の中置式でネスト構造と判断するのでネスト変数を 1 増やし infixexpr 状態に再帰する．後置式が出現するので warning 状態に遷移し後置式が出現した箇所を警告する．警告後中置式

終了イベントで end_expr 状態に遷移する．ネスト変数は 1 なのでネスト変数を 1 減らし infixexpr 状態に遷移する．遷移した条件式は中置式終了イベントで end_expr 状態に遷移する．ネスト構造でなくなったなので start_ast 状態に遷移し，全ての if 文の検査が終わったら終了する．

5 検査仕様の記述方法

5.1 仕様記述言語

仕様記述にあたって，静的解析ツールで検査する規則を状態遷移機械を用いて定義する言語である Metal[1] を利用する．Metal は制御フローグラフを対象とし，誤りとなる可能性がある構文要素の並び方を記述することで，ソースコード中から走査順が一致する箇所を発見することを目的としている．Metal では，状態，構文要素の並びとなるイベント，その構文要素が存在した場合の振る舞いを記述する．Metal の基本操作を次に示す．

Metal の基本操作

```
state decl 変数定義;
状態:
 {イベント}==>遷移後の状態 [,{アクション}];
```

[] で囲まれた部分は記述しない場合がある．[] 以前の部分は構文要素を指定するための記述であり，この部分を複数記述することで構文要素に対する指定をより詳しく記述することができる．状態と構文要素の組み合せと，アクションを書くことで警告する内容を記述できる．

前章で述べたとおり，JCI の検査処理は状態遷移機械で記述できる．構文要素の出現や終了をイベントと捉えることで，フローグラフに対する処理は Metal で記述できる．しかし，抽象構文木の処理に対する記述は入れ子構造を考慮する必要がある．入れ子構造を把握するために要素のカウントをアクションとして定義することで，JCI の検査処理は Metal で記述できると考えた．しかしながら，これらのイベントは Metal で定義されていないので Metal を拡張する必要がある．

5.2 仕様記述言語の拡張

検査項目の開発事例を基に検査仕様の記述で必要な情報を整理し，Metal の拡張を行なう．JCI で実現されている検査処理を分析した結果，検査仕様に必要な情報を次に定義し Metal を拡張する．

- 標準化されたイベントの定義
  - 構文要素の出現
  - 構文要素の終了
  - 任意の文字列
- 特殊イベントの定義
  - end_of_path：プログラムの終了
  - $not_nesting$：ネスト構造でない
  - $nested_structure$：ネスト構造中
- 標準化されたアクションのライブラリ化
  - ネスト変数の増減

Metal を拡張することで，検査仕様の記述は制御フローグラフに対するものだけでなく，抽象構文木やデータフローグラフに対しても可能にする．加えて，それらの検査仕様の記述は統一される．

6 アーキテクチャ設計

検査処理の実現や検査項目，テストケースの生成を目的に，GoF デザインパターンを組み合わせたアーキテクチャを図 4 に示すように設計した．

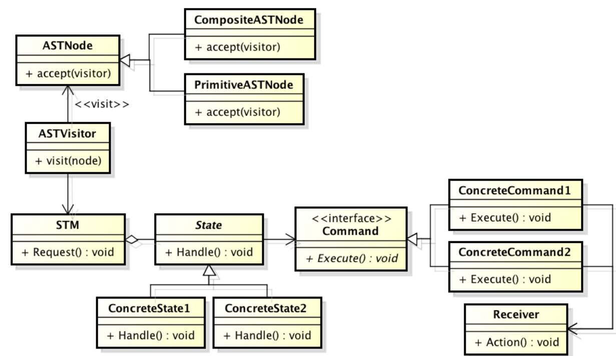

図 4 設計したアーキテクチャ

仕様の追加変更要求に対して他の状態に影響を与えずに対応できる仕組みの導入を目的に，State パターンを適用した．このパターンでは，状態で扱うイベントを規定するインターフェースを用意し，そのサブクラスのメソッドで各状態に対応する具体的な振る舞いと状態遷移を定義する．また，アクションの分離を目的に，Command パターンを適用した．このパターンでは，状態遷移におけるアクションや，Visitor による構文要素の出現や終了のイベントを呼び戻し，イベントと対応したソースコード片を出力する処理の実現をする．命令自身をクラスとして用意し，そのクラスのインスタンスを利用して，呼び出したい処理の選択や処理の前後の設定をまとめられるので，命令が複雑なものであっても扱いを簡易化できる．

7 考察

検査項目とテストケースの自動生成に関して考察する．これらは提案した検査仕様から自動生成できると考えており，検査仕様は誤ってないものとする．

7.1 テストケースの自動生成

検査仕様は自然言語を使って示された警告例の記述なので，テストケース作成者が仕様から考えられる構文要素やフローグラフの組み合せを考慮してテストケースを手作業で作成していた．また，仕様変更が行なわれた際，新たにテストケースを作成していた．我々は，検査仕様として検査対象の走査処理が明確になることで，テストケースを自動生成できると考えた．提案した検査仕様は状態遷移機械で表現されるので，イベントと遷移で辿る経路の情報からソースコード片を組み合わせてテストケー

スを生成する．テストケースの生成において，次の問題が存在する．

- 経路の循環
- 誤っているテストケースの生成

テストケースの生成には，経路を算出する必要がある．状態遷移はグラフ構造であるので，経路が循環する問題がある．この問題を解決するために，走査を終了する基準を定義して走査が循環しない機構を実現する必要がある．例えば，循環を防ぐために次の制約例を設けると，図3から導き出せる警告される経路は3つ，警告されない経路は4つ導出される．これらから生成されるテストケースは，検査仕様の妥当性の確認に用いることができる．

- 一度処理を行なった辺に対して処理を行なわない
- ネストの増減は1回

テストケースを生成するには，イベントに対してソースコード片を対応付けておく．経路に対するイベントにソースコード片を組み合わせることで，正しい構文のテストケースが生成できる．条件式など演算子を必要とする構文要素にはそれぞれの演算子を用いたソースコード片を対応付けておくことによって，複数のパターンのテストケースを生成することができる．

検査項目に対するテストケースは，警告すべき構文要素の存在の有無や特定の構文要素が異なる種類のテストケースが必要である．上記のテストケース生成方法では，警告すべき構文要素の存在の有無のテストケースしか生成できないので，特定の構文要素が異なる種類のテストケースを生成する必要がある．そこで，特定の構文要素が異なる種類のテストケースを生成するには，関連がある構文要素を類型化し，類型化した構文要素を部品として利用する．例えば，for文やif文を部品として類型化し，利用することで異なった構文要素のテストケースが生成できると考える．

7.2 検査項目の自動生成

現在のJCIにおける検査項目の実現は，検査仕様を基に抽象Visitorのサブクラスに構文要素の種類に応じた検査処理内容を記述する．検査仕様から検査項目の自動生成を行なうことで，検査項目の作成を省力化できる．仕様変更が行なわれた際，検査仕様から検査項目を自動生成を行なうので仕様変更に柔軟に対応できる．我々は，検査仕様として着目したい構文要素や着目した構文要素の振る舞いが明確になることで，検査処理が自動生成できると考える．

図4で設計したアーキテクチャを基に，検査項目の自動生成を考える．検査仕様は，Metalを用いて検査処理論理を状態遷移機械で記述されている．状態遷移機械とVisitorによって走査処理された構文要素の出現や終了イベントを入力とすることで検査処理論理のオブジェクトコードは自動生成可能だと考える．各状態とイベントによる振る舞いや遷移はStateパターンで定義してある．状態遷移を表わす検査処理は類似したオブジェクトコードになる．そこで，Commandパターンを用いてオブジェクトコードを生成するアクションを用意し，利用することで検査項目を生成する．Commandパターンの多相性を利用することで，取得した構文要素ごとに異なる処理を記述できる．実際にMetal記述を入力とし，オブジェクトコードを生成する概念図を図5に示す．

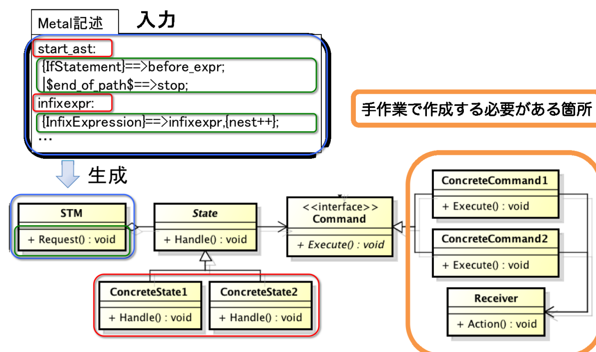

図5 検査項目の自動生成の概念図

Commandパターンを用いることで，アクションを分離して記述する．アクションは手作業で作成する必要があり，今後増える可能性があるので標準化できるようにする必要がある．一度作成したアクションは，再利用可能な部品になると考える．

8 おわりに

本研究では，既存の検査項目の開発事例から仕様記述に必要な情報を整理し，検査仕様記述方法の提案を行なった．検査仕様を定義したことにより，異なるデータ構造でも統一的に検査仕様を記述できた．検査仕様記述から検査項目と検査項目に対するテストケースを生成するアーキテクチャを設計した．今後の課題として，定数伝播やオブジェクト伝播といった数値を利用した検査項目の検査仕様記述するための拡張や，検査項目と検査項目に対するテストケースの自動生成の実現がある．また，検査仕様を記述するための，開発補助インターフェースの構築があげられる．

参考文献

[1] S. Hallem, B. Chelf, Y. Xie and D. Engler, “A System and Language for Building System-Specific, Static Analyses,” *PLDI '02 Proceeding of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pp. 69-82, 2002.

[2] 浦野彰彦，沢田篤史，野呂昌満，蜂巣吉成，“デザインパターンを用いたソースコードインスペクションツールのソフトウェアアーキテクチャ設計,”ソフトウェア工学の基礎 *XVII (*日本ソフトウェア科学会 *FOSE2010)*, 近代科学社, pp.15-24, 2010.

[3] 大須賀俊憲，小林隆志，間瀬順一，渥美紀寿，山本晋一郎，鈴村延保，阿草清滋，“CX-Checker: C 言語プログラムのためのカスタマイズ可能なコーディングチェッカ,”ソフトウェア工学最前線 *2009 (*情報処理学会ソフトウェアエンジニアリングシンポジウム *2009* 論文集*)*, 近代科学社, pp.119-126, 2009.