静的解析ツールの系統的な部品化に関する研究

2006MI071 金 賢修　　2007MI272 加藤 遼介
指導教員　野呂 昌満　　蜂巣 吉成

1　はじめに

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, 実際に動作させせることなく種々の誤りや不具合があるか検証し, 品質を向上させる技法である. 本研究室では Java ソースコードに対する品質向上を目的として, コードインスペクションツールである Java Code Inspector[2]（以下 JCI）を開発している. JCI は入力されたソースコードに対して構文解析やフロー解析などをおこない, その結果を利用してプログラムの欠陥となる可能性がある箇所を検出する.

JCI は全部で 36 種類の検査項目を提供し, プログラムの不正な処理をおこなっている箇所や好ましくないコーディングスタイルを使用している箇所を検出することができる. ただし, 利用者が特定の構文要素や独自に定義したコーディングスタイルに対して検査をおこないたいと考えた場合, 抽象構文木やフローグラフなど JCI の内部構造を理解した上で検査項目を記述する必要がある.

本研究の目的は JCI に対して, 利用者が容易に検査項目を開発できる環境を構築することである. 目的を達成するためのアプローチとしてわれわれは部品化をおこなった. 検査項目間で同様の処理が複数回記述されており, これらを部品とし, 部品を利用して利用者は個々の要求に応じて独自に検査を作成することで既存の JCI よりも容易に問題を解決することができる.

研究の課題として, JCI の内部構造を隠ぺいできる単位で部品化することと, 部品化にあたって既存の JCI に対する変更箇所を少なくすることが挙げられる. 既存の JCI は利用者が検査項目を開発することを考えて作られてはおらず, 開発には内部構造を理解する必要があり, その開発には困難が伴なう. また, JCI は今後も機能を拡張していくことが予想されるので, 追加する機能が既存の機能に与える影響を局所化することが求められる.

本研究の目的を達成するために, われわれは構文要素とそれに関する処理の整理をおこなった. JCI の抽象構文木は現在まとめられている単位以外に, 条件判定における利用のされ方によっては異なる単位でまとめることができることが分かった. したがってわれわれは既存の検査項目間で構文木の各構文要素がどういう条件判定で利用されているかを整理し, 類型化をおこなった. そして, 既存の検査項目では類型化した構文要素に対してどのような処理がおこなわれているかを整理した. 整理した結果, 複数の構文要素にまたがる処理は部品化する視点によって様々なものが現れることが分かった. 成果を JCI に反映させるために, どの GoF デザインパターン [1] を適用するか比較検討をおこなった. 比較する基準としては, 既存の JCI に対する影響, 部品追加の容易さ, そして内部をどの程度隠ぺいできるかを挙げた.

本研究の成果として, 検査項目を作成する際に必要な部品を系統的に部品化できた. また GoF デザインパターンを適用することで直接構文木の各要素に機能を追加するよりも現在の JCI に対しての影響を局所化しつつ部品を利用できるようにした.

2　JCI

2.1　検査処理の概要

JCI の検査処理の概要を図 1 に示す. JCI は入力されたソースコードに対して構文解析をおこない, 抽象構文木を作成する. そして, 作成した抽象構文木を基にフローの解析をおこない, データフローグラフ及び制御フローグラフを作成する. 個々の検査処理はソースコード解析結果としての抽象構文木やフローグラフを走査し, 欠陥の可能性となる箇所を検出する.

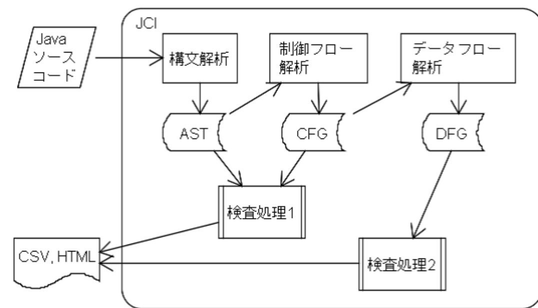

図 1　JCI の検査処理の概要

処理のうち, 構文解析や意味解析, フロー解析, 結果を出力する処理は開発者が検査項目を開発する際に意識しなくても良い箇所である. これらの処理のうち, 構文解析や意味解析, フロー解析は Java の言語仕様に依存するので変更される頻度は低い.

検査処理の部分は検査に対する要求によって多様に変化するので, 変更する頻度が高い. JCI では検査機能の拡張に対する柔軟性と保守性, 再利用性が重要視されており, 検査処理の追加や変更がツール全体に影響をおよぼさないことが求められる.

2.2　JCI のソフトウェアアーキテクチャ

前節で議論した要求を踏まえて JCI のアーキテクチャを図 2 のように設計した. JCI は GoF デザインパターンを用いて設計されている. Java のソースコードから抽象構文木を作成するさい, 抽象構文木の再帰的な構造を表現するために Composite パターンを利用し, Interpreter パターンを利用してその抽象構文木の走査順序を定義している. また, Visitor パターンによって抽象構文木のデータ構造と検査処理が分離されている.

検査処理の追加や変更を，構文要素や他の検査処理に影響を与えずにおこなうことができるので，保守性や拡張に対する柔軟性を確保することができている．

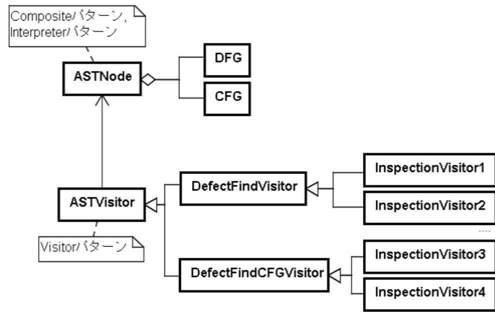

図 2　JCI のアーキテクチャ

3　JCI の検査項目作成

3.1　検査項目の作成手順

JCI の検査項目を作成する際の手順を図 3 に示す．

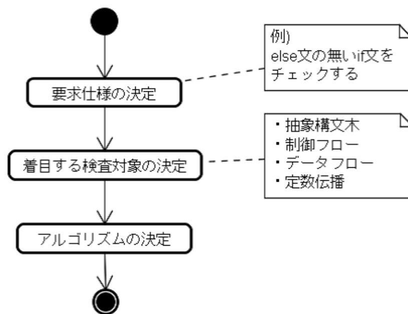

図 3　検査項目の作成順序

開発者は最初にどのような構文に対してどのような検査をおこなうか，要求仕様を決定する．そして要求仕様を基に JCI 上で着目する検査対象を決定する．アルゴリズムの決定においては，実際に抽象構文木や各フローグラフのどのような機能を用いて検査をおこなうかを決定する．

3.2　データ構造

検査項目の開発において，開発者は大きく分けて以下の 2 つを理解する必要がある．

- 抽象構文木
- フローグラフ

JCI は入力されたソースコードの各構文要素を検査するために抽象構文木を作成する．開発者は検査したい構文要素が抽象構文木上ではどの項目に当たるのか調査し，ノードの機能を利用して検査項目を記述する．

フローグラフにはデータフローグラフと制御フローグラフの 2 種類がある．データフローは変数などの定義点，使用点を辿るものであり，制御フローはプログラムが実行される経路を辿るものである．開発者は要求仕様がどちらのフローグラフを利用するかを調査した上で，それぞれのフローグラフを走査する Visitor のサブクラスとして検査項目を記述する．

3.3　部品化によって実現する機能

前節では利用者が考える検査対象となる構文要素とそれに対する処理よりも，抽象構文木やフローグラフなどさらに細かい単位で検査項目を記述する必要があることを示した．利用者が容易に検査項目を開発するには，JCI の内部構造を隠ぺいし，より大きい単位で検査を記述できるように部品化する必要がある．

部品化するにあたってわれわれは既存の検査項目を状態遷移の集合と捉え，どのような構文要素が条件判定で用いられ，どんな処理がおこなわれているか調査した．

4　部品化

検査項目作成に必要なソフトウェアを整理し部品化する．各部品は JCI の内部構造を隠ぺいし，内部構造を理解しなくても検査項目作成できるように部品化した．

4.1　部品の単位

JCI の各検査項目の条件判定を調査し，検査の対象となる構文要素と，各構文要素に対しておこなう処理を整理した．整理した結果を基に各構文要素の類型化と，複数の構文要素に対して存在する処理を整理し，各処理の抽象化をおこなった．構文要素を類型化した結果を表 1 に示す．

表 1　構文要素の類型化

構文要素の類型化
繰り返し文，分岐文，例外処理，条件式を持つ，中断文，修飾子を持つ，型を持つ，名前を持つ，null の可能性を持つ，値を持つ，演算子を持つ，式，Javadoc を持つ，初期化式を持つ，変数，宣言，配列

おこなう検査の仕様によって検査の対象となる構文要素は多様であり，構文要素を類型化してもすべての検査仕様を網羅することは困難である．このことは各構文要素を個々の単位で利用することを可能にし，検査仕様によって類型化した部品の追加を可能にすることで対応できる．また構文要素に対する処理も検査の仕様によって多様な処理が存在する．各処理を抽象化した結果を表 2 に示す．

表 2　処理の抽象化

処理の種類	細分化した処理
ある要素が存在するか	使用する箇所があるか，代入される箇所があるか Javadoc があるか
比較して同じであるか	同じ型か，同じ値か，同じ修飾子か，同じ名前か，同じ式か，同じ変数か
数値を比較して設定値以上，以下であるか	数値が設定値以上か，ループの回数は設定値以上か，分岐の数は設定値以上か

各処理は抽象化した処理をそのまま部品として利用することも可能だが，細分化した各処理を利用することでより多様な検査を作成できる．細分化される各処理は検査の仕様によって多様であり，一意に定めるのは困難である．このことは現在の検査項目を調査する過程で定め

た処理の部品に追加して，新たな処理の部品を追加可能にすることで対応できる．

4.2 部品の利用

部品を利用して検査項目を作成する場合，複数の構文要素を一まとめにして扱うことで個々の構文要素の詳細を気にせずに検査を作成できる．部品の利用者は複数の構文要素に対しておこなわれる処理が，実際にどのようにおこなわれるかを意識する必要はない．しかし，実際にはまとめられた構文要素の数の分，各構文要素に対して同じ内容の処理をおこなう必要がある．構文要素ごとに同じ内容の処理をおこなう例を図 4 に示す．

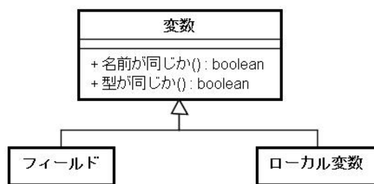

図 4　構文要素ごとの処理

図 4 の例では変数に対する「名前が同じか」の処理が一つの部品となるが，利用者は構文要素と処理を別の部品としてとらえて利用する．利用者が意識する部品の構造を図 5 に示す．図 5 は変数に対して別の変数と同じであるかの検査を表している．

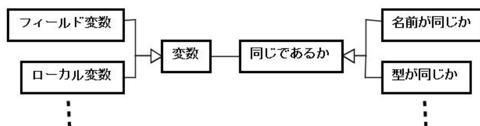

図 5　利用者視点での部品

利用者は対象とする構文要素，もしくは類型化した構文要素の部品を対象に処理の部品を組み合わせる．実際の構造を知る必要はなく，検査対象の要素とおこなう処理の部品のみを知れば良い．

4.3 JCI の変更

前述で説明した部品を利用できる環境を整備するために，JCI に多様な部品を用意し，それらを組み合わせて利用できる仕組みを追加する．部品の実現のために JCI のアーキテクチャに変更を加える．アーキテクチャを変更する理由として JCI が開発された当初 [3] には JCI をソフトウェア部品として整理し，開発者ではなく利用者が新たに検査項目を作成することを想定していなかったことが挙げられる．

部品に求められる要求として，部品の再利用性と追加・変更の柔軟性がある．各構文要素をまとめて利用することや，まとめた構文要素ごとに多様な処理を組み合わせることから部品は再利用が容易な形で追加する必要がある．また現在整理した各部品は既存の検査項目の仕様を基に整理しているが，検査の仕様によって多様な部品が存在しうる．このことから部品の追加・変更は柔軟におこなえる必要がある．

4.4 Command パターンの適用

部品化した結果を JCI のアーキテクチャに反映させるために，Command パターンを適用した．Command パターンを適用したさいのアーキテクチャを図 6 に示す．

Command パターンは動作とそれに伴なうパラメータをカプセル化するパターンであり，部品化における抽象化した処理が動作に，パラメータは対象となる抽象構文木の各構文要素が相当する．抽象化した処理を実現するために実際に必要となる処理は構文要素ごとに異なるが，Command パターンではポリモーフィズムを利用することで，構文要素ごとに異なる処理を記述することができる．

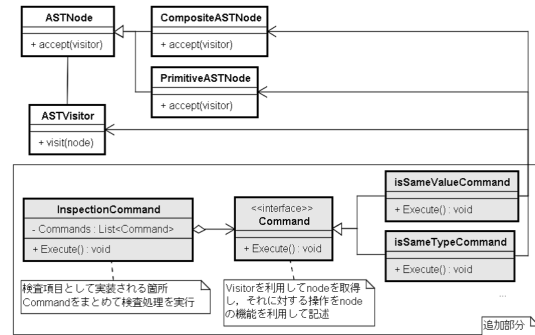

図 6　Command パターンを適用したさいの構造

検査項目は利用者が抽象化した処理とその対象となる構文要素を組み合わせることで実現する．Command パターンは複数の命令を組み合わせて利用することを考えて設計されている．部品化では抽象化した処理と対象になる抽象構文木の類型化をおこない，それらの組み合わせで検査項目を表現するので，Command パターンは部品の特徴と一致していると考えた．

新たな部品の追加は，Command インターフェースを実現するクラスを作成し，Visitor や構文要素の各機能を利用することで実現する．このさい，Visitor や構文要素の機能は変更されないので，部品の追加や変更に対する既存の構造への影響を局所化することができる．

5 考察

5.1 GoF デザインパターンの適用

アーキテクチャを変更するにあたって，既存の JCI に対する影響を局所化することが重要になる．整理した結果をそのまま抽象構文木の構造に反映した場合，Interpreter パターンで定義された走査順序を書きなおす必要があり，既存の検査項目にも影響が及ぶ．

GoF デザインパターンには担保したい性質と設計が対応付けられているので，抽象構文木の構造を直接変更するよりも影響を局所化できる．また，抽象化の視点によって様々な部品ができるということが整理をおこなった結果判明したので，今後の部品の追加が容易となるような設計ができたことから，GoF デザインパターンを利用するのは妥当であると考えた．

5.1.1 Decorator パターン

前章で挙げた Command パターン以外に部品を表現するパターンとして Decorator パターンを挙げる．Decorator パターンを適用した場合のアーキテクチャを

図 7 に示す.

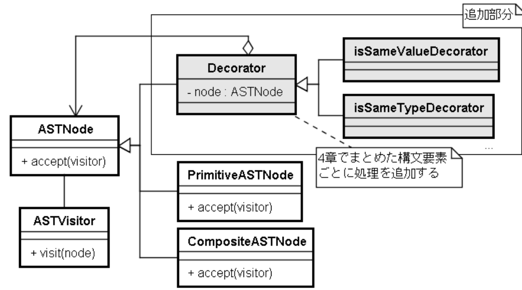

図 7　Decorator パターンを適用したさいの構造

Decorator パターンは既存のオブジェクトに新しい機能や振る舞いを動的に追加することを可能とするパターンである．Command パターンでは部品を抽象化した処理とそれに対する構文要素を 1 つのクラスとして表現するが，Decorator パターンでは構文要素に対して抽象化した処理を追加する Decorator として表現する．

検査項目は検査の対象となる構文要素に Decorator を適用し，抽象化した処理を利用することで実現する．Decorator パターンも Command パターンと同様に，組み合わせて使われることを考えて設計されており，部品の特徴と一致していると考えた．

部品を追加するさいは Decorator クラスのサブクラスとして追加する抽象化した処理を表すクラスを作成する．Decorator によって既存の Visitor や構造に対して変更をする必要はなく，部品の追加や変更に対する影響を局所化することができる．

5.2　GoF デザインパターンの比較

前述した 2 つのパターンを比較した結果を次に示す．Command パターンと Decorator パターンはいずれも前章および前節で述べたように部品の追加や変更に対する影響を局所化することができる．内部構造については，Command パターンでは抽象化した処理とそれに対する構文要素，Decorator パターンでは構文要素と抽象化した処理の組み合わせを理解するだけでよく，利用者は JCI の構造の詳細を理解する必要はない．したがって，どちらのパターンも柔軟性を保ったまま内部を隠ぺいすることができる．しかし，Command パターンは部品を羅列するだけで検査項目を実現することができるが，Decorator パターンは構文要素に Decorator を適用し，その上で追加した機能を組み合わせて検査項目を実現するので必要となる手続きが多くなる．今後，GUI などを用いて利用者が実際にコードを記述しなくても検査項目を記述できるようにしたいなどの要望が出た場合，Command パターンの方がより容易に実現することができると考えられる．以上から Command パターンの方が適していると考えた．

5.3　部品の妥当性

既存の各検査項目の条件判定を調査し，構文要素と処理に着目して整理した．整理した結果を基に構文要素の類型化と処理の抽象化を行ない，部品として利用できるように JCI のアーキテクチャの変更を行なった．各部品は対象となる構文要素と行なう処理を組み合わせることによって検査作成に利用できる．部品を組み合わせた検査の作成例を図 8 に示す．

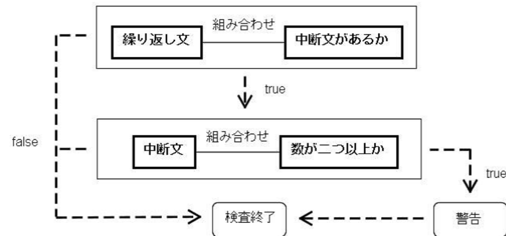

図 8　中断文が複数ある繰り返し文の検出する検査

図 8 では for 文，while 文，do 文，拡張 for 文を類型化した繰り返し文に対して「break 文があるか」，「return 文があるか」などの処理を抽象化した「中断文があるか」を検査する．結果が true であった場合は，次の「中断文の数が二つ以上か」を検査することで，中断文が複数ある繰り返し文を検出する．この条件判定の内，「繰り返し文に中断文があるか」の処理は既存の JCI では制御フローを辿ることで判定を行なっている．しかし，この例のように部品を組み合わせて利用する際にはこのことを意識する必要はない．検査作成の例から，各構文要素と処理は部品として再利用可能なこと，制御フローなどの内部構造をしらなくても利用できることから，部品として妥当だと考えた．

6　おわりに

本研究では現在実装されている検査項目の条件判定に着目し，整理をおこなうことで，系統的に部品化をおこなうことができた．整理した部品は GoF デザインパターンを適用することで JCI に対する影響を小さくしつつ部品を使えるようにすることができた．

今後の課題として，実装をおこなうことで今回考察したパターンが妥当であったのか議論をした上で，もっとも適したパターンを示すことと，JCI に検査作成のための環境を整備して実際に検査項目を作成することで部品とアーキテクチャの検証をおこなうことが挙げられる．

参考文献

[1] E.Gamma，R.Helm，R.Johnson，and J.Vlissides，*Design Patterns: Elements of Reusable Object-Oriented Software*，Addison-Wesley,1995.

[2] 浦野 彰彦，沢田 篤史，野呂 昌満，蜂巣 吉成，張 漢明，吉田 敦：”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，” 第 17 回ソフトウェア工学の基礎ワークショップ FOSE2010，2010.

[3] 後藤 洋：”Java ソースコードの CDI(Code Inspection) の開発　〜アーキテクチャの構築〜，” 南山大学大学院数理情報研究科 2008 年度修士論文要旨集，pp.130-133，March 2009.