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
—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—

M2009MM028 浦野彰彦
指導教員：沢田篤史

1　はじめに

本稿では，株式会社キャナリーリサーチ (以下，キャナリーリサーチ) と南山大学との共同研究，及び OJL として実施した，ソースコードインスペクションツール JCI(Java Code Inspector)[4] の開発の報告を行なう．我々は，要求の変化に対して柔軟かつ即応的に対応可能なアーキテクチャの設計を目的として，ソフトウェアアーキテクチャスタイルと GoF(Gang of Four) デザインパターン [1] をベース技術と位置づけ，アーキテクチャ設計に利用した．さらに，PLSE(Product Line Software Engineering)[2] をベース技術に対するメタ技術と位置付け，ソースコードインスペクションツール開発のための専用手法として，ソフトウェアアーキテクチャを中心とした開発環境構築を実施した．これらの技術や手法の利用によって，生産性の高い，柔軟なツール開発を実施できた．

ソースコードインスペクションにおいて，プログラムのソースコードに対する完全な実行前検査の実施は不可能である．すなわち，障害が起こりうるすべてのコードパターンを定義し，その定義に基づき発見することは不可能である．なぜならば，特定のコードパターンと障害の関係は，検査に対する要求によって定義され，さらにその要求は検査の対象となるドメインによって異なるからである．このような理由から，検査に対する要求は多様に変化し，それに追従して発見したいコードパターンも変化する．

多様なコードパターンの発見が可能なソースコードインスペクションツールを開発するためには，要求の変化に対して柔軟に対応可能な仕組みが必要である．要求の変化によって変更が生じる箇所を局所化したソフトウェアアーキテクチャを設計することで，検査の追加や変更が容易になり，保守作業に対する柔軟性を保証できる．さらに，検査で共通に利用される処理やデータをコンポーネント化し，必要に応じてそれらを再利用することで，要求の変化に対して即応的に対処することが可能になる．

本研究の目的は，多様な検査項目を備えたソースコードインスペクションツールの開発を可能とするために，要求の変化に対して柔軟かつ即応的に対応可能なソフトウェアアーキテクチャを設計することである．本研究では，以下の手順でソフトウェアアーキテクチャの設計を行なった．

- ドメイン分析
- 概念的アーキテクチャ設計
- 詳細アーキテクチャ設計

本稿では，実開発で生じた要求への対応を通じて，設計したソフトウェアアーキテクチャを中心とする開発を実施したことで，要求に対して柔軟に対応できたことを示す．

2　JCI 概要

JCI は，Java ソースコードの中から不具合を生じる可能性のある部分を指摘するツールである．入力のソースコードに対して，コーディングスタイルや構文規則，制御フロー，データフローの観点から静的に検査を行なう．

我々が提案した JCI の検査処理の概要を図 1 に示す．入力されたソースコードから抽象構文木を構築するとともに，フロー解析結果から制御フローグラフとデータフローグラフを構築する処理が必要である．個々の検査項目は，ソースコードに対する解析結果として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ことになる．

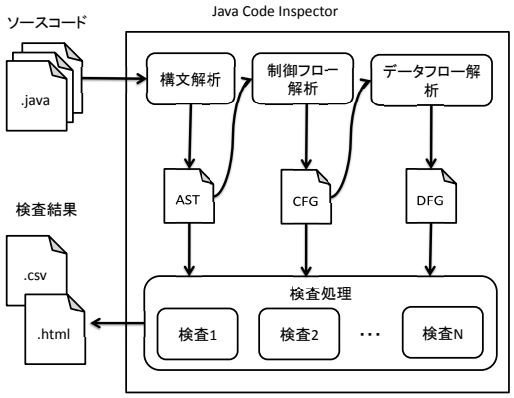

図 1　JCI の検査処理概要

3　アーキテクチャ設計

3.1　PLSE に基づく開発プロセス

我々は，ソフトウェア部品の再利用によるソースコードインスペクションツールの開発環境構築を目的として，PLSE に基づく開発プロセスを実施した．PLSE では，想定される製品系列の固定部分と可変部分に基づいてソフトウェア部品群を定義し，製品ごとの要求に応じた部品の組み合わせによる開発を行なうことで，開発の省力化を狙う．このような製品開発は，製品系列の固定部分と可変部分が明確なドメインが適しており，我々はソースコードインスペクションツールの開発に対しても適用可能であると判断した．我々が実施した PLSE に基づく開発プロセスの概念図を図 2 に示す．本論文では，核資産開発における概念的アーキテクチャ設計，詳細アーキテクチャ設計について議論する．

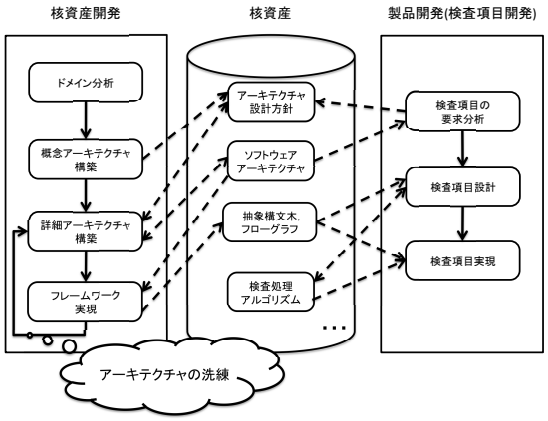

図 2 PLSE に基づく開発プロセス

3.2 概念的アーキテクチャ設計

3.2.1 ドメインの特徴

前章で示した JCI の一連の処理のうち，字句解析や構文解析，フロー解析，及びこれらの処理から出力される成果物である抽象構文木やフローグラフは，検査対象の言語仕様や文法に依存している．検査対象のプログラミング言語の言語仕様が変更されない限り，抽象構文木の構造などが変更されることは無いので，変更の頻度が低いと判断した．一方で，個々の検査項目を実現する検査処理は，検査に対する要求によって多様に変化することから，追加や変更の頻度が高いと判断した．これらの特徴から，JCI の固定部分と可変部分を以下のように定義した．

- 固定部分
  - 抽象構文木
  - 制御フローグラフ，データフローグラフ
- 可変部分
  - 検査処理

1 章で述べたように，検査処理に対する多様な要求を実現するためには，要求の変化に対して柔軟に対応する必要がある．言い換えるならば，検査に対する要求が変化したさいに発生する，変更にかかる保守作業を軽減する必要がある．頻繁に変更が起きる検査処理によってツール全体の保守性が左右されることから，我々はツールの可変部分に対して特に，保守作業が必要な箇所の特定が容易であること (解析性) と，変更が必要な範囲が小さく変更が容易であること (変更容易性) を保証する必要があると判断した．これらの特徴に加えて，検査処理の新規開発においては，抽象構文木やフローグラフ，及びそれらに対する走査処理を再利用することで開発の生産性の向上を見込めることから，ツールの固定部分の再利用性も併せて保証する必要があると判断した．

3.2.2 JCI の概念的アーキテクチャ

前節のドメイン分析の結果に基づいて，ソースコードインスペクションツールの概念的アーキテクチャを設計する．我々はドメイン分析の結果から，JCI のアーキテクチャに対して以下の要求が存在すると判断した．

- 抽象構文木やフローグラフの再利用性
- 検査処理に対する解析性
- 検査処理の変更容易性

これらの要求に加えて，JCI は言語処理系のドメインに分類されることから，言語処理系のアーキテクチャの基本的な構造を踏襲することが要求される．コンパイラをはじめとする言語処理系のドメインにおいては，実例に基づいた安定したアーキテクチャが Shaw らによって提案されており [3]，アーキテクチャスタイルのうち，Pipe and Filter スタイルと Repositories スタイルを採用している．Pipe and Filter スタイルを採用したことで，各解析処理の独立性が保証され，追加や変更が容易になる．さらに Repositories スタイルを採用したことで，各解析処理 (手続き) と計算実体 (データ構造) が分離され，それぞれのコンポーネントに対する変更が局所化される．これらの特徴を踏まえて設計した JCI の概念的アーキテクチャを図 3 に示す．

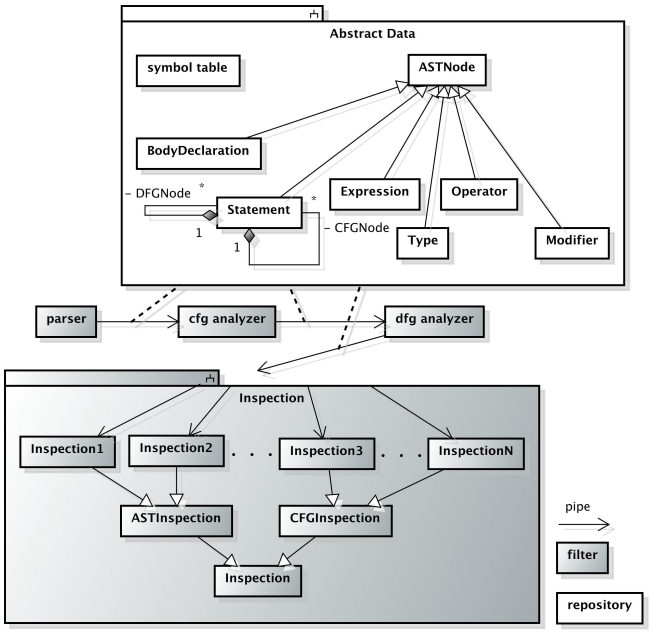

図 3 JCI の概念的アーキテクチャ

3.3 詳細アーキテクチャ設計

設計した概念的アーキテクチャに基づいて詳細アーキテクチャを設計する．詳細アーキテクチャの設計では，前述したアーキテクチャに対する要求を保証することを目的として，GoF デザインパターンを適用する．GoF デザインパターンのパターンカタログには，各パターンを適用する目的，解決される設計上の問題点，適用したさいに生じるトレードオフといったアーキテクチャ設計に必要な情報が記述されており，パターンを詳細な構造を定義するためのマイクロアーキテクチャとして捉えることができる．以降の節で，適用するデザインパターンについて議論する．

3.3.1 抽象構文木

本節では，抽象構文木のデータ構造の設計について議論する．概念的アーキテクチャ設計では，抽象構文木の再利用性を保証していないので，本節ではこれを保証することを目的として，構造に関するデザインパターンのうち，Composite パターン，Facade パターンについて比較

を行なう．
Composite パターンを適用した場合，抽象構文木を再帰的な集約構造として表現することで各ノードを一様に扱うことが可能になり，抽象構文木全体の再利用が容易になる．その一方で，新たなノードの追加を制限できないことから，不要なインタフェースの定義を招き，設計が一般化されてしまう可能性がある．
Facade パターンを適用した場合，統一的なインタフェースを定義することで抽象構文木の構造を検査処理から隠蔽し，抽象構文木に対する変更を局所化できる．さらに，検査処理との結合が疎になるので，抽象構文木の再利用性が高まる．
いずれのパターンを適用した場合でも，抽象構文木の再利用性は保証される．しかしながら，Facade パターンを適用した場合，定義した統一インタフェースを通して外部から利用される対象は抽象構文木のみであり，複雑なサブシステムのインタフェースの単純化という Facade パターンの利点を活かすことができない．Composite パターンを適用した場合，ノードの追加による設計の過度な一般化が懸念されるが，前述したように我々は抽象構文木の変更の頻度は低いと判断したので，この欠点は考慮しなくてよいと判断した．以上の理由から，我々は抽象構文木のデータ構造の設計に Composite パターンを適用した．

3.3.2 検査処理

本節では，検査処理の設計について議論する．アーキテクチャに対する要求として，検査処理に対する解析性と変更容易性が存在するので，これらの要求をアーキテクチャで実現可能なデザインパターンを選択する．本節では振舞いに関するパターンのうち，Chain of Responsibility パターン，Interpreter パターン，Visitor パターンについて比較を行なう．
Chain of Responsibility パターンは Composite パターンとの親和性が高く，親子関係にあるノードのリンクを定義することで抽象構文木の走査が可能になる．Chain of Responsibility パターンを適用することで，検査処理で各ノードに対する走査順序を再利用できるようになるが，検査処理が各ノードに散在してしまい，検査処理の解析性が低くなる．
Interpreter パターンを適用する場合，各ノードに検査処理のためのメソッド (Interpret メソッド) を定義することになる．Composite パターンで定義した構造に対して適用する場合，Composite ノードに子ノードに対する走査順序を集約できるので，検査処理に必要な走査の再利用が可能になる．一方で，検査処理を追加するさいに，すべてのノードに対して Interpret メソッドを新たに定義する必要があるので，変更が必要な範囲が大きくなってしまう．
Visitor パターンを適用する場合，各検査処理は抽象 Visitor を拡張することで実現される．ノードに対する走査順序は Interpreter パターンで Accept メソッドとして定義され，各ノードに対する具体的な処理は Visitor に集約される．Visitor パターンを適用することで，検査処理の解析性が保証されるが，その反面，抽象構文木に対する変更が起こったさいに，変更が必要になる範囲が大きい．
検査処理の解析性や変更容易性の実現では，いかに検査処理を抽象構文木から分離し，局所化するかが鍵となる．すなわち，データ構造に横断する手続きの適切なモジュール化が重要である．Chain of Responsibility パターンや Interpreter パターンでは，走査順序の再利用が容易であるという利点があるが，いずれのパターンを適用した場合でも，検査処理を抽象構文木のノードに対して定義する必要があり，データ構造に対する手続きを分離できない．Visitor パターンと Interpreter パターンの組合わせでは，抽象構文木の走査順序と各ノードに対する処理が分離されており，適切なモジュール化がなされていると判断できる．このような理由から，我々は検査処理の設計に Visitor パターンと Interpreter パターンを組み合わせて適用した．

4 考察

我々は，JCI の詳細アーキテクチャを，Composite パターン，Visitor パターン，Interpreter パターンを組み合わせることで設計した．設計したアーキテクチャを図 4 に示す．

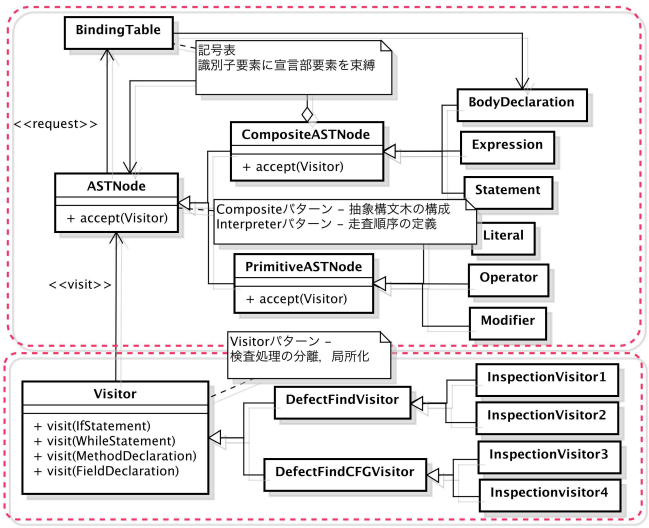

図 4 JCI の詳細アーキテクチャ

4.1 アーキテクチャの進化

本節では，我々が実開発で生じた要求に対応するために行なったアーキテクチャの変更について議論し，代替案との比較を通じてその妥当性について考察する．

4.1.1 機能拡張

検査機能の高精度化を図る目的で，コンパイラの最適化技法の一つである定数伝播を検査処理に共通する基本機能と位置付け，これを再利用可能なコンポーネントとするためにアーキテクチャの変更を含む機能拡張を行なった． 定数伝播とは，ソースコード中の変数の値を既知の定数値に置き換えることでプログラムの最適化を図る技法である．定数伝播によって変数の値を静的に計算し，その計算結果を利用することで，デッドコードや null オブ

ジェクトに対する参照，配列の範囲外参照などの箇所を発見する検査が実現できる．
定数伝播処理は，制御フローグラフを走査しながら，抽象構文木やデータフローグラフの情報を利用 して変数の値を計算することで実現できる．したがって，定数伝播にかかる処理を実現する新たな Visitor を追加すれば良いことになるが，この処理を複数の検査項目で共通に利用するためには，計算結果をいずれかの場所に格納する必要が生じる．
我々は，定数伝播処理の計算結果を導入するにあたって，Decorator パターンの適用を検討した．Decorator パターンには，既存のデータ構造に対して，取り外し可能な形で機能拡張できるという利点がある．すなわち，データ構造に対する機能の追加や変更を，付加した Decorator に局所化することで，データ構造に対する変更容易性を保証できる．一方で，Decorator パターンによる機能拡張は，場当たり的なデータ構造の拡張として捉えることも可能である．データ構造に対するこのような変更を繰り返すと，構造の複雑化によるツール全体の保守性の低下を招き，アーキテクチャが劣化する可能性がある．
我々は，アーキテクチャの洗練が必要になることを見越して，定数伝播の計算結果の導入に Decorator パターンを採用せず，代わりに抽象構文木の文要素に新たにフィールドを追加する方法を採用した．Decorator パターンを適用した場合と異なり，抽象構文木のノードに対して変更を加えることになるので，すべての検査処理を実現する Visitor の振舞いが変わらないことを保証しなければならない．しかしながら，機能拡張による抽象構文木の構造の複雑化を防ぐことができることから，アーキテクチャ設計の変更を行なう価値があると判断した．
JCI が提供する検査項目のうち，配列の添字が定義内にあるか否かの検査，null オブジェクト参照の検査， デッドコードの検出などの項目で定数伝播計算を利用している．これらの検査の精度向上に定数伝播機能は貢献していることから，この判断は妥当であると考えた．

4.1.2 性能改善

JCI の開発にあたり，検査対象としてキャナリーリサーチが想定していた Java ソースコードは，1MLOC を超えるものであった．この規模のソースコードに対する検査を，2GB 程度のメモリを搭載した比較的安価なコンピュータで可能にすることが，JCI に対する要求として挙がっていた．開発の初期においては，検査機能の充実や精度向上を重視していたが，製品化にあたり，空間効率の改善を目的としたアーキテクチャ変更を行なう必要が生じた．
改善にあたって，プロファイラなどを用いて調査した結果，抽象構文木の末端要素と定数伝播の計算結果データがメモリを大量に消費していることが判明した．例えば，public 修飾子や static 修飾子といった頻出する構文要素に対して，個々にインスタンスの生成を行っていることが抽象構文木の末端要素でのメモリの浪費につながっていた．
この結果を受けて我々は，Flyweight パターンを適用することでアーキテクチャの設計変更を行なった．Flyweight パターンを適用することで，インスタンス生成の管理を局所化することができる．つまり，インスタンス生成の関心事を，データ構造や手続きから分離しモジュール化することで，その解析性や変更容易性を保証することができる．
アーキテクチャの変更を行なった結果，JCI 全体で使用するメモリ容量を約 11%削減することができたことから，変更は妥当であると考えた．変更後の詳細アーキテクチャを図 5 に示す．

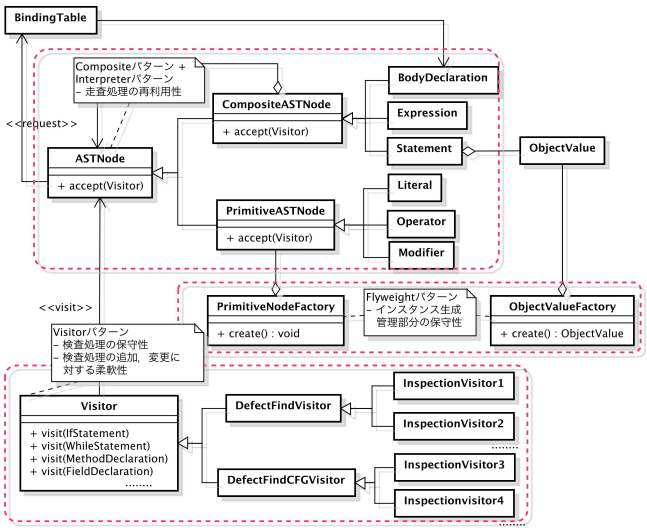

図 5 変更後の詳細アーキテクチャ

5 おわりに

本研究では，OJL 開発を通じて，ソースコードインスペクションツール JCI のソフトウェアアーキテクチャを設計した．アーキテクチャ設計では，要求の変化に対して柔軟かつ即応的に対応可能なツール開発の実現を目的として，検査で共通に利用されるデータや処理の再利用性や，検査に対する解析性や変更容易性を保証した．定数伝播処理の導入と空間効率の改善を目的としたアーキテクチャの変更では，代替案との比較を通じて設計の妥当性を確認できたと考えている．

参考文献

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

[2] K. Pohl, G. Bockle, and F. Linden, *Software Product Line Engineering Foundations, Principles, and Techniques*, Springer-Verlag, 2005.

[3] M. Shaw, and D. Garlan, *Software Architecture - Perspective on an Emerging Discipline*, Prentice Hall, 1996.

[4] 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, 2010, pp. 15-24.