# Javaソースコードの CDI (Code Inspection) ツールの開発
## ～ ライブラリ情報の追加によるアーキテクチャの変更 ～

M2008MM027 長大介
指導教員：野呂昌満

## 1　はじめに

インスペクションとはソフトウェアの静的検証技術のひとつで，決められた手順で成果物を検査し，品質を向上させる技術である．私は本 OJL(On the Job Learning) において，これまでの OJL で開発されたコードインスペクションツール (CDI ツール)[4] を引き継ぎ，品質向上および機能改善をおこなった．この CDI ツールはプログラミング言語 Java[1] のソースコードを検査の対象とする．

CDI ツールの検査機能に対する要求は多種多様に変化することから，CDI ツールのアーキテクチャは，検査項目の追加や変更に対して柔軟な設計となっている．このアーキテクチャは，検査対象のデータと検査処理を分離している．その結果，プロダクトラインソフトウェアエンジニアリング [2, 3](PLSE) に基づいた検査項目開発を可能にし，検査項目の開発を省力化した．このアーキテクチャに基づいた検査項目の開発では，検査の共通処理である構文解析処理などが核資産として再利用可能である．

CDI ツールの検査項目に対する要求の中には，ライブラリの情報が不可欠なものも存在する．検査処理においてライブラリの情報を使用することで，検査処理をさらに厳密化，詳細化，多機能化することができる．しかし，開発対象の CDI ツールには，ライブラリの情報を扱っていないという制約があり，各検査項目はその制約のもとに開発されてきた．CDI ツールは，検査対象のソースコードに対する抽象構文木のみを扱っていて，ライブラリコードに対する抽象構文木は定義されていない．さらに，検査対象のソースコードとライブラリコードの間では，抽象構文木の特性が異なる．

本 OJL の目的は，CDI ツールを拡張し，ライブラリの情報を用いた検査処理を開発可能にすることである．この目的に対するアプローチとして，CDI ツールのアーキテクチャ変更し，検査対象データの拡張に対して対応可能とした．変更後のアーキテクチャでは，検査に用いるデータの記号表に対して Proxy パターンを適用し，複数のデータが記号表を介して結合する．新たに導入する検査データの題材として，ライブラリコードの抽象構文木を用いて，CDI ツールに導入した．実際にこのアーキテクチャに基づいた検査項目の開発をおこない，ライブラリの情報を用いた検査項目が開発可能であることと，CDI ツールの検査項目の追加や変更に対する柔軟性は損なわれなかったことを確認した．

本 OJL のメタ技術は，アーキテクチャの拡張方法としてのデザインパターンである．ベース技術は，静的解析技術，オブジェクト指向である．専用手法は，アーキテクチャの拡張による検査対象データに対する拡張性の導入である．

## 2　CDI ツールのアーキテクチャ

CDI ツールのアーキテクチャの設計指針は，字句解析および構文解析などの検査に共通する処理と，意味解析を分離することである．CDI ツールの検査機能に対する要求は多種多様に変化するので，意味解析処理の追加および変更に対して柔軟なアーキテクチャが求められる．ソースコードに対して意味解析をおこない，欠陥の可能性がある箇所を発見するには，ソースコードを抽象表現するデータ構造が必要である．このデータ構造は，検査の対象とするプログラミング言語の使用や文法によって決定するので，変更の頻度は低い．反面，意味解析処理は，CDI ツールの検査に対する要求によって多種多様に変化するので，追加や変更の頻度は高い．

CDI ツールのアーキテクチャの概略を図 1 に示す．このアーキテクチャにおいては，意味解析処理の追加および変更に対して柔軟に対応が可能である．このアーキテクチャに基づいた検査項目の開発では，PLSE に則った核資産の再利用が可能である．その際の核資産は，アーキテクチャおよび抽象構文木などの検査に共通する部分である．

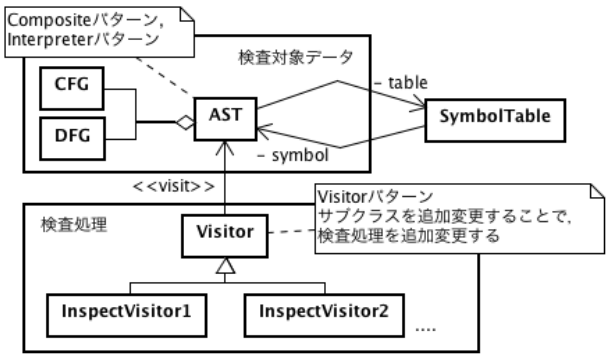


図 1　CDI ツールのアーキテクチャ

CDI ツールでは，Java 言語の文法を Composite パターンを用いて抽象構文木としてモデル化している．抽象構文木に対して Interpreter パターンを適用し，抽象構文木の走査順序を定義している．さらに，検査処理を抽象構文木に横断する関心事としてとらえ，Visitor パターンを用いて，手続きとして抽出，局所化している．このアーキテクチャに基づいた検査項目の開発では，検査処理を実現する Visitor を追加変更すればよい．

CDI ツールにおいて，ソースコード上の識別子に束縛される構文要素を解決する処理は，記号表がおこなう．記号表を用いる場合の例として，メソッド起動式とメソッド宣言の対応を挙げる．メソッド起動式に対する部分構文木と記号表の関係を図 2 に示す．

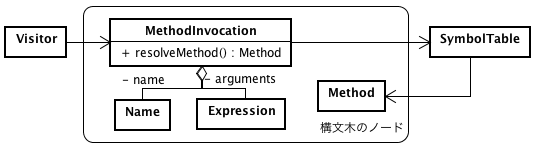


図 2 メソッド起動式からの記号表参照

検査モジュールから，メソッド起動式に対応するメソッド宣言を解決する場合の挙動を図 3 に示す．検査モジュールはメソッド起動式のノードに対して，対応するメソッド宣言をリクエストする．そのさい，メソッド起動式のノードは，メソッド宣言の解決処理を記号表に委譲する．

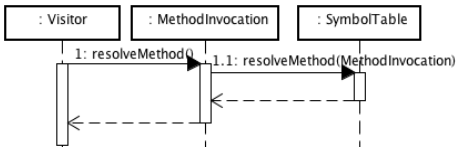


図 3 メソッド起動式からメソッド宣言の取得

## 3 アーキテクチャ変更に対する要求

CDI ツールのアーキテクチャの変更における要求は以下の 2 点がある．

- ライブラリコードのデータ構造に対する要求
- CDI ツールへのライブラリコードのデータ導入に対する要求

### 3.1 ライブラリコードのデータ構造に対する要求

ライブラリコードを表現するデータ構造には，ソースコードを表現するデータ構造にならい，抽象構文木を採用する．ただし，ライブラリコードの抽象構文木は，ソースコードの抽象構文木とは異なるデータ構造になる．

ライブラリコードに関する情報は，検査対象のソースコード上の識別子を解決する目的で用いられる．ライブラリコードの抽象構文木は，ライブラリにおいて提供されるクラスに関する情報と，そのクラスが提供するインタフェースに関する情報のみを含めば良い．また，ライブラリコードは CDI ツールにおける検査対象のデータではないので，ライブラリコードの抽象構文木に対する処理の追加変更に対する柔軟性は，必ずしも必要ではない．

### 3.2 ライブラリコードのデータ導入が可能なアーキテクチャへの変更に対する要求

CDI ツールのアーキテクチャの変更にあたっては，検査処理の追加変更に対する柔軟性を損なわないようにする必要がある．CDI ツールのアーキテクチャは，1 つの検査対象のデータと複数の検査処理とに分離されていることから，検査処理の追加変更に対して柔軟である．

CDI ツールのアーキテクチャを変更するさいには，CDI ツールが持つ検査処理に対する柔軟性を維持する必要がある．これを維持するには，変更後のアーキテクチャにおいても，1 つの検査対象データと複数の検査処理から成るアーキテクチャである必要がある．

## 4 検査処理に用いるデータ構造の設計

検査処理に用いるデータ構造は，検査対象の抽象構文木と，抽象構文木に対する記号表がある．抽象構文木および記号表を設計するにあたり，次の 2 つのアプローチを考えることができる．

1. ソースコードとライブラリを 1 つのデータとして表現する
2. ソースコードとライブラリを別々のデータとして表現する

### 4.1 抽象構文木に対する設計

ソースコードの抽象構文木とライブラリの抽象構文木について，1 つの抽象構文木として表現する場合と，それぞれ独立の抽象構文木として表現する場合を考える．

#### 1 つの抽象構文木として表現する場合

Composite パターンを用いて，ソースコードの抽象構文木とライブラリの抽象構文木を 1 つの抽象構文木として表現した場合のクラス図を，図 4 に示す．

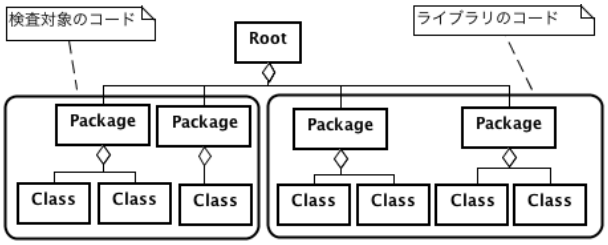


図 4 1 つの抽象構文木とした場合のクラス図

Composite パターンを用いて複数の抽象構文木を結合する場合，結合後の全体の抽象構文木と，結合される部分構文木は同一視可能である必要がある．この制約は，結合される抽象構文木とライブラリコードの抽象構文木を過度に一般化する．また，CDI ツールが検査の対象とする抽象構文木の構造が変化することから，既に開発された検査項目への影響が波及することが危惧される．

#### それぞれ独立の抽象構文木とする場合

ソースコードの抽象構文木とライブラリの抽象構文木をそれぞれ独立に表現する場合のクラス図を，図 5 に示す．

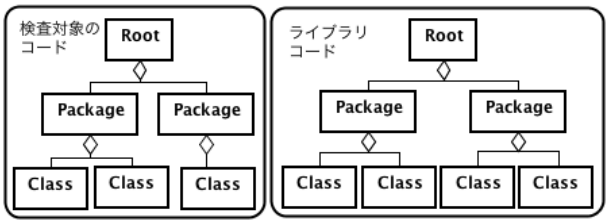


図 5 抽象構文木を独立にする場合のクラス図

それぞれの抽象構文木を独立とする場合，それらの抽象構文木は同一視できる必要はないので，新たな抽象構文木の設計に関する制約が少なくなる．加えて，ライブ

ラリコードは検査対象の抽象構文木に含まれる識別子を解決する目的で使用されるので，2 つの抽象構文木は記号表を介して結合すればよく，1 つの抽象構文木とする必要はない．

2 つの案を比較検討をおこなった結果．新たなデータの導入に際する自由度の観点から，異なる検査対象のソースコードとライブラリコードは別々の抽象構文木として表現する案を採用した．この案を採用したことで，今後新たに導入されるデータの構造に対する制約を小さくすることができる．

### 4.2 記号表に対する設計

記号表の設計について，次の 2 つの案を検討する．

- 検査対象のソースコードから識別子の解決をする記号表と，ライブラリコードから識別子の解決をおこなう記号表の 2 つを導入する
- 検査対象のコードとライブラリコードの両方から，識別子の解決をおこなう記号表を 1 つ導入する

2 つの案を比較検討をおこなった結果．それぞれの抽象構文木に対して記号表を導入する案を採用した．1 つの記号表にすべての抽象構文木からの識別子を解決する責任を与えると，記号表はすべての抽象構文木と密に結合し，複数の抽象構文木に横断する要素となることから，適切ではない．

## 5 デザインパターンの適用

それぞれの抽象構文木は記号表を介して結合するので．デザインパターンの選択においては，複数の記号表をどのようにサポートするかが鍵となる．デザインパターンを適用する上での前提条件として，それぞれの抽象構文木は独立であり，それぞれの抽象構文木に対して記号表があるとする．適用可能なデザインパターンとして，次の 2 つを比較検討する．

- 検査モジュールに対して Strategy パターンを適用する場合
- 記号表に対して Proxy パターンを適用する場合

### 5.1 Strategy パターンを適用した場合

識別子の解決をおこなうさいに利用する記号表を保持する Strategy を定義する．各 Strategy は識別子の解決処理において，検査対象のコード，ライブラリコード，またはその両方のいずれかの記号表を使用する．検査モジュールは，これらの Strategy のうち 1 つを保持する．

Strategy パターンを適用し，ライブラリコードのデータ構造を導入した場合のアーキテクチャを図 6 に示す．また，メソッド起動式を例にした場合の挙動を図 7 に示す．

CDI ツールの検査処理は，ライブラリコードまで含めて識別子を解決する必要がある検査と，そうでない検査があり，記号表に対する要求は検査処理ごとに異なる．Strategy パターンを適用すると，検査モジュールごとに，検査に使用する記号表の指定が可能になる．その反面，検査モジュールは記号表と結合するようになる．

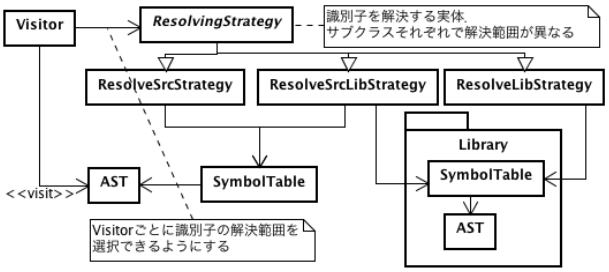


図 6 Strategy パターン適用時

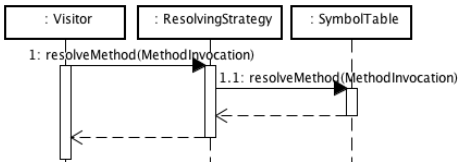


図 7 Strategy パターン適用時の挙動

新たに検査に用いるデータを追加するには，新たなデータから識別子の解決をおこなう記号表を用意し，その記号表から識別子を解決する Strategy を定義すればよい．対象とするデータ構造の数が多く，Strategy の数が膨大になってしまう場合には，Decorator パターンによる Strategy の実現も考えられる．

### 5.2 Proxy パターンを適用した場合

2 つの記号表の両方を用いて識別子の解決をおこなう記号表の Proxy を導入する．Proxy は識別子の解決に使用する記号表をすべて集約する．識別子の解決時には，Proxy に集約されている記号表に解決処理を委譲する．抽象構文木は，識別子の解決を記号表ではなく，Proxy に解決を依頼をする．

Proxy パターンを適用してライブラリコードのデータ構造を導入した場合のアーキテクチャを図 8 に示す．また，メソッド起動式を例にした場合の挙動を図 9 に示す．

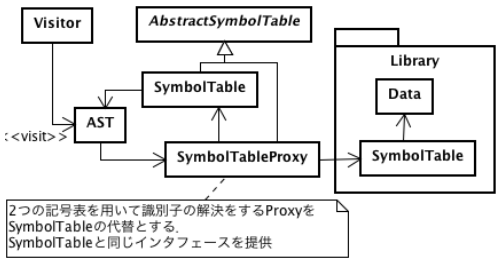


図 8 Proxy パターン適用時

CDI ツールにおける従来の記号表の処理は抽象構文木内に隠蔽されている．Proxy パターンを適用した場合は記号表の構成が変化するので，検査処理に対するアーキテクチャ上の影響は発生しない．

新たに検査に用いるデータを追加するには，Proxy は新しいデータに対する記号表を集約すればよい．

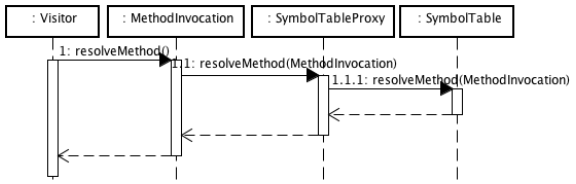


図 9 Proxy パターン適用時の挙動

### 5.3 適用するデザインパターンの比較

Strategy パターンと Proxy パターンの比較において，検査モジュールと検査対象データとの結合度の観点から，Proxy パターンを適用した．

Strategy パターンおよび Proxy パターンのどちらを用いても，検査に使用するデータ構造に対する拡張性を導入可能なので，アーキテクチャの不変性の観点から両パターンの比較をおこなった．変更前のアーキテクチャでは，記号表に識別子を解決する処理を委譲することは抽象構文木の役割となっていて，記号表を用いた処理は検査処理に対して隠蔽されている．検査モジュールは変更後のアーキテクチャでも，記号表を用いた識別子解決の処理は隠蔽をするほうが望ましい．

Proxy パターンを適用する場合，アーキテクチャ上は記号表の構成が変更される．この場合，検査モジュールに対して記号表は隠蔽されたままである．Strategy パターンの適用例は，検査モジュールは抽象構文木と記号表の両方と結合するので，Proxy パターンの適用時と比べて結合が密である．

## 6 考察

データ構造の設計およびデザインパターンの適用を反映させたアーキテクチャでは，検査に使用するデータ構造をそれぞれ独立に定義するようにし，それらのデータは，記号表の Proxy を介して結合する．変更後のアーキテクチャを図 10 に示す．

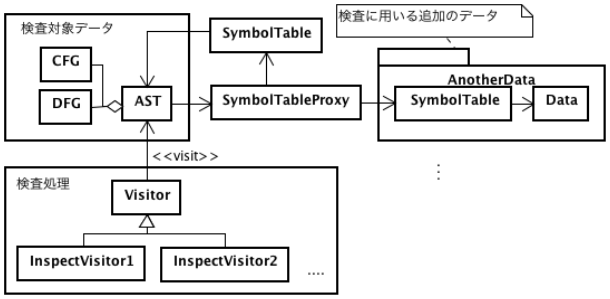


図 10 変更後の CDI ツールのアーキテクチャ

### 6.1 検査に使用するデータ構造の拡張性に関する考察

新たに検査データを追加するには，抽象構文木と記号表を用意する．変更後のアーキテクチャにおいて，複数の検査データは記号表を介して結合する．新たなデータ構造を導入する場合は，その記号表を Proxy に集約させることで導入が可能である．このように，変更後のアーキテクチャでは，検査対象のデータに対する拡張性が導入できたと言える．

### 6.2 検査処理の柔軟性に関する考察

検査処理に対する柔軟性は，ひとつの抽象構文木に対して導入された柔軟性である．検査処理は検査対象の抽象構文木のみを対象とする．これまで述べてきたアーキテクチャの変更では，検査対象の抽象構文木に対して変更はおこなっていないので，検査処理に対する柔軟性は損なわれていない．

検査処理に対する柔軟性が損なわれていないことを確認するために，変更後のアーキテクチャを用いて，実際にライブラリコードの情報を用いる検査処理を開発した．その結果，既存の検査処理をライブラリコードの情報を用いた検査処理へ変更することに成功した．検査処理に対する柔軟性は保持をしたまま，ライブラリコードの情報を用いた検査処理を開発可能であることが確認できた．

### 6.3 核資産に関する考察

変更後のアーキテクチャに基づいて，ライブラリの情報を用いた検査が実現できたことから，ライブラリコードの抽象構文木は核資産として再利用が可能である．さらに，本開発によって検査項目に対する柔軟性は損なわれておらず．構文解析や検査対象のソースコードの抽象構文木などの既存の核資産は，変更後のアーキテクチャにおいても核資産として再利用が可能である．

## 7 おわりに

本 OJL の成果は，CDI ツールのアーキテクチャに，検査処理に用いるデータ構造に対する拡張性を導入したことである．そのうえで，検査処理の高機能化を目的として，ライブラリコードを抽象化したデータ構造を導入した．

本 OJL のメタ技術は，アーキテクチャの拡張方法としてのデザインパターンである．ベース技術は，静的解析技術，オブジェクト指向である．専用手法は，アーキテクチャの拡張による検査対象データに対する拡張性の導入である．

## 参考文献

[1] B. Joy, G. Steele, G. Bracha, and J. Gosling, *The Java(TM) Language Specification*, Addison-Wesley, 2005.

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

[3] L. M. Northrop, “SEI’s Software Product Line Tenets,” *IEEE Software*, vol. 19, no. 4, pp. 32-40, 2002.

[4] 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.