Java ソースコードを対象とした影響波及解析ツールの開発
– 高精度化におけるアーキテクチャの再設計–

2007MI042 林 宏美　　2007MI057 星野 陽　　2007MI075 伊藤 彩乃
指導教員　野呂 昌満　沢田 篤史

1　はじめに

影響波及解析 [2] とは，ソフトウェアに加えられた変更に対して，その影響が及ぶ可能性のある箇所を識別し，影響が及ぶことで修正する必要がある箇所を推定する手法である．本研究では Java ソースコードを対象とした影響波及解析ツール JCIA(Java Change Impact Analysis) を扱う．JCIA は変更による影響箇所を解析し，テストケース選択を支援する．

影響波及解析の精度は，実行時間や何を変更とみなすかによってユーザごとに異なる．さまざまなユーザの要求を満たすために，JCIA では影響波及解析アルゴリズムの変更に対する柔軟性が求められている．変更に対する柔軟性などの性質が保証できるかは，アーキテクチャによって決まる．現在のアーキテクチャで担保されていない性質については，性質を保証できる新たなアーキテクチャを構築する必要がある．JCIA のような言語処理系は大規模かつ複雑であり，変更に対する柔軟性や保守性が必要となる．ソフトウェア開発で起こる問題を類型化したデザインパターン [1] は存在するが，言語処理系に特化したものはない．したがって，言語処理系に特化した，担保すべき性質を保証する系統的な変更方法は示されていない．担保すべき性質を保証するアーキテクチャの変更方法の確立が課題となる．

本研究の目的は，JCIA において，担保すべき性質とそれを保証するアーキテクチャの系統的な変更方法を示すことである．変更方法を示すことで，変更のたびに開発者にかかる，性質を保証する新たなアーキテクチャを模索する労力を軽減することができる．

われわれは目的を達成するために，変更で担保したい性質とデザインパターンを対応付ける．デザインパターンは，担保したい性質とそれを保証するアーキテクチャが対応付けられたカタログである．デザインパターンを用いることで，対応関係にある性質を担保しながらアーキテクチャを変更することができる．変更で担保すべき性質とデザインパターンの担保したい性質を対応付けることで，変更で担保すべき性質とそれを保証する変更方法を対応付けることができると考えた．

本研究では，まず，JCIA に求められている要求を抽出し，JCIA の変更で，処理の変更に対する柔軟性や，処理対象データの安定性を担保する必要があることが分かった．次に，JCIA と同系統のソフトウェアであるコードインスペクションツール JCI(Java Code Inspector)[3] や JCIA の今までの変更事例を再検討した．JCI は JCIA と同様に変更の頻度が低いデータ構造と変更の頻度が高い処理からなり，ともに処理の変更の柔軟性が求められる．共通の要求があるので，JCIA においても同様の変更が可能だと考え，JCI の変更事例を利用した．JCIA の高精度化に向けた変更で，過去の変更を基にした対応表を利用し，処理の変更の柔軟性や処理対象データの安定性を保証できるか検証した．対応表に無い種類の変更は，複数の変更方法を比較し変更した．

本研究の成果として，JCIA の高精度化の事例を考察した結果から，対応表を利用することで，処理の変更の柔軟性や処理対象データの安定性が保証できることを示せた．作成した対応表について，言語処理系一般に適用できることを示した．

2　研究背景

2.1　JCIA の概要

JCIA の解析処理の概要を**図 1** に示す．

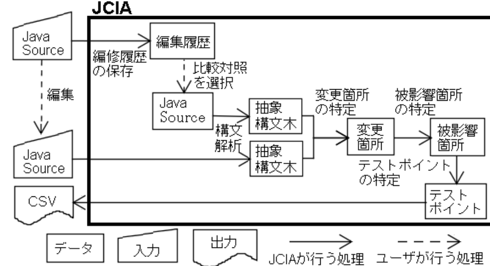

図 1　JCIA の解析処理の概要

JCIA は Java ソースコードに対して静的に影響波及解析をおこなうツールである．ソースコードの編集履歴を保存し，過去のソースコードと現在のソースコードを入力とする．変更前後の各ソースコードの抽象構文木を構築し，構文木に対してフロー解析をおこなう．構築された抽象構文木やフローグラフを基に解析し，変更箇所と変更による影響を受けた箇所 (被影響箇所)，テストすべきメソッド (テストポイント) を出力する．

JCIA のアーキテクチャの概略を**図 2** に示す．

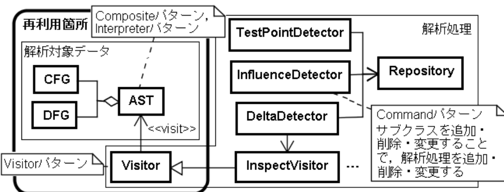

図 2　JCIA のアーキテクチャ

JCIA は，同系統のソフトウェアであるコードインスペクションツール JCI を再利用して開発した．JCIA のアーキテクチャは，JCI のものと同様，抽象構文木に Composite パターン，その走査順序に Interpreter パターン，処理とデータ構造の分離に Visitor パターン

を用いて設計した．JCIA では JCI と違い，変更箇所を検出するために変更前後の構文木を比較するので 2 つの構文木を扱う．JCIA ではさらに，重複する処理を Command パターンで定義している．Command パターンは，被影響箇所を特定する処理のアーキテクチャで用いた．変更内容に応じて 1 つ以上の被影響箇所の解析アルゴリズムを利用できるようになっている．

2.2 JCIA に対する要求

JCIA に求められる要求を**表 1** に示す．

表 1　JCIA に対する要求

分類	要求
機能	変更履歴の保存
	変更箇所の解析
	被影響箇所の解析
	テストポイントの解析
非機能	高い解析精度
	解析時間
	解析処理の変更が容易
	外部ツールとの相互運用が可能
	解析処理の変更が他の解析処理に影響しない

機能に関する要求は，実現する解析処理についてのものである．ソースコードを保存し，変更前後のソースコードに対して影響波及解析をおこなう．とくに被影響箇所の解析では，メソッドやクラスをまたがる複雑な解析が求められる．

非機能要求では，ツールの品質に対する要求と保守性に対する要求がある．品質では，解析の方法や空間効率が求められる．保守性では，ツールの拡張性や，機能を拡張するさいの試験性が求められる．

2.3 変更において担保すべき性質

JCIA のアーキテクチャは，解析対象データと解析処理によって構成されている．解析対象データは，検査を対象とするプログラミング言語の仕様や文法によって決定されるので，変更の頻度は低い．反面，解析処理は，JCIA の変更箇所，被影響箇所，テストポイントの検出に対するユーザの要求によって多種多様に変化するので，変更の頻度は高い．

JCIA に求められる保守性を満たすには，変更の頻度が低い解析対象データの安定性や，変更の頻度が高い解析処理の変更に対する柔軟性を担保しなければならない．処理の変更に対する柔軟性は，変更の頻度が高い処理ほど，変更に対して柔軟であることが求められる．

JCIA は今後 JCI との連携も考えられるので，相互運用できるかも重要である．JCIA と JCI でともに利用する既存の部品については，安定性が必要となる．

3 アーキテクチャの変更におけるデザインパターンの適用

表 2 は JCI と JCIA の過去の変更事例から，変更の指針をまとめたものである．表の作成にあたり，担保したい性質を保証できる方法を複数比較し，変更において担保すべき性質を保証できるデザインパターンの適用方法を考察した．

われわれは JCIA の高精度化の変更事例を通じて，過去の変更から得た対応表が妥当か，対応表に無い種類の変更指針をまとめた．対応表の使いかたとして，まず変更する箇所の特徴と担保したい性質の組合せを対応表から探す．対応に応じたデザインパターンを用いて変更する．高精度化で変更する箇所は以下の 3 つである．

- 変更箇所にあるフィールドの参照箇所の解析
- 変更箇所と代入互換性に関する箇所の解析
- 変更箇所の解析

表 2 に無い種類の変更では，複数の変更方法から性質を保証する方法として適した変更方法を考察した．JCIA の高精度化で新たにまとめた対応表を**表 3** に示す．

3.1 フィールド参照箇所の解析の変更

フィールド参照箇所の変更で以下の 3 つを変更する．

- 被影響箇所の解析の変更
- デッドコード検出機能の導入
- デッドコードを含まない定数伝播の作成

デッドコードを含まない定数伝播の作成は，JCI の定数伝播とデッドコード検出機能を利用して作成するので，複数の処理を組み合わせた処理の追加という特徴を持つ．過去の変更の対応表に無い変更である．既存の機能を変更すること無く，組み合わせて新たな機能を作成することから，変更に柔軟でない既存の処理の安定性や，それらを複数用いた処理の使用性を担保することが必要だと考えた．変更方法として以下の 3 つを比較し，Facade パターン適用が妥当だと考えた．

- Facade パターン適用
- Client で処理
- 新たな定数伝播の作成

Facade パターン適用時の特徴として，Facade にメッセージを送るのみで，デッドコードを含まない定数伝播を利用できる．既存の定数伝播を変更しなくても，Facade の作成のみで新たな機能である，デッドコードを含まない定数伝播が作成できる．Facade パターンを適用したクラス図を**図 3** に示す．

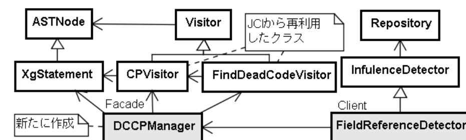

図 3　Facade パターン適用

Client で処理する場合の特徴として，今後，フィールド参照箇所以外でデッドコードを含まない定数伝播を利用する可能性が高い．そのさい，Client となるクラスに再度同じ操作を定義しなければならない．各 Client で処理した場合のクラス図を**図 4** に示す．

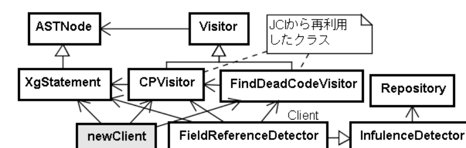

図 4　Client で処理

新たな定数伝播を追加する場合の特徴として，ノードは新たな伝播特有の情報をいくつも保持する必要があ

表 2　過去の変更事例から得られた対応表

変更の種類(変更事例)	担保すべき性質	変更の詳細	デザインパターン
一意に決まる，ある箇所の情報を保持する箇所への問い合わせ(ライブラリ情報の追加:JCI)	構文木の情報を保持するデータを処理から隠蔽，構文木の情報を利用する処理に対する安定性	複数の記号表の解決を処理から隠蔽，検査処理の変更なし	Proxy パターン
インタフェースが異なる，データの種類が同じ情報の追加(ライブラリ情報の追加:JCI)	既存の構文木の情報を保持するデータの安定性，データの種類が同じ情報の変更に対する柔軟性	異なるインタフェースを適合	Adapter パターン
構文木の生成のさい，重複する要素の生成を抑制(空間効率の改善:JCI)	インスタンスが共有できる要素の資源効率性	解析の実行のさいに余分なインスタンスの生成を抑制	Flyweight パターン
構文木に対するデータ走査処理の変更(検査処理の変更:JCI，伝播の導入:JCI，JCIA)	変更の頻度が高い処理の変更に対する柔軟性，処理対象データの安定性	処理をおこなう Visitor の変更による影響の抑制，データ構造の変更なし	Visitor パターン
いくつかの処理で重複する処理の変更(被影響箇所の解析の変更:JCIA)	変更の頻度が高い処理に対する柔軟性，変更の頻度が高い処理の再利用性	保守すべき箇所が明確,変更しても他の解析には影響なし，処理を再利用可能	Command パターン

表 3　JCIA の高精度化から得られた対応表

変更の種類(変更事例)	担保すべき性質	詳細	デザインパターン
既存の処理を利用した処理の変更(デッドコードを含まない定数伝播の導入:JCIA)	変更に柔軟でない既存の処理の安定性，変更に柔軟でない複数の処理からなる機能の使用性	定数伝播とデッドコードを含まない定数伝播を同じように利用可能，JCI への導入が可能	Facade パターン
既存の処理の拡張(インスタンスの型の伝播の導入:JCIA)	変更に柔軟でない既存の処理の拡張性，変更に柔軟でない既存の処理からなる機能の安定性	共通の処理がある null 伝播を利用して型の伝播を実現，既存の処理の変更なし	Decorator パターン
入出力の種類が同じ処理の変更(変更箇所の解析の変更:JCIA)	入出力の種類が同じ各処理の変更に対する柔軟性	保守すべき箇所が明確，変更しても他の解析に影響なし	Strategy パターン

る．多くのノードを対象とした複雑な処理を定義しなければならない．

3.2　代入互換性に関する箇所の解析の変更

代入互換性に関する箇所の解析の変更では，以下の 3 つの変更をおこなう．

- 被影響箇所の解析の変更
- インスタンスの型の伝播の導入
- インスタンスの型の伝播の作成

インスタンスの型の伝播の作成では，JCI の機能である null か null でないかを伝播する null 伝播を拡張するので，既存の処理の拡張という特徴を持つ．過去の変更の対応表には無い変更である．変更方法として以下の 2 つを比較し，Decorator パターン適用が妥当だと考えた．

- Decorator パターン適用
- 共通の処理を抽出

Decorator パターン適用時の特徴として，Decorator パターンでは，既存の null 伝播をそのまま利用できる．null 伝播に追加する処理のみを新たに定義すれば良い．Decorator パターンを適用したクラス図を**図 5** に示す．

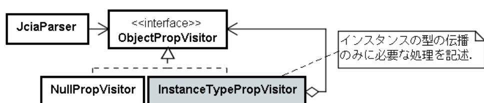

図 5　Decorator パターン適用

共通の処理を抽出する場合の特徴として既存の null 伝播の処理で，インスタンスの型の伝播と共通の処理を抽象化するために，変更する必要がある．共通の処理を抽象化することで，同じ処理が散在せず，保守しやすい．共通の処理を抽出したクラス図を**図 6** に示す．

null 伝播は JCI で利用されている機能であり，JCIA でも今後の null 伝播の利用を考慮すると，null 伝播の変更は安定性を損なってしまう．結果として，Decorator パターンを適用することが妥当だと考える．

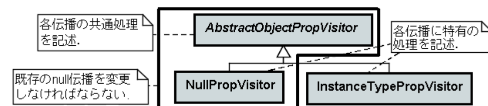

図 6　共通の処理を抽象化する場合

3.3　変更箇所の解析の変更

変更箇所の解析の変更では，以下の変更をおこなう．

- 変更箇所のリファクタリング

変更箇所の解析は現在，1 つのクラスで複数の変更箇所の解析をおこなっている．変更箇所毎に分離することで変更が柔軟になると考え，リファクタリングする．変更箇所の解析のリファクタリングでは，入出力の種類が同じ処理の変更という特徴を持つ．過去の変更の対応表には無い種類の変更である．変更方法として以下の 2 つを比較し，Strategy パターン適用が妥当だと考えた．

- Strategy パターン適用
- Strategy パターンと Command パターンの併用

Strategy パターンの特徴として，新たな変更箇所の解析を容易に追加できる．Command パターンとの併用の特徴としては，何度もおこなう処理を再利用可能である．

変更箇所の解析において再利用可能な処理は少ない．結果として Strategy パターンを適用するほうが妥当だと考えた．

4　考察

JCIA の高精度化の事例を通じて，対応表が妥当であったか考察する．4 つの項目について説明し，対応表の一般性についても考察する．

4.1　構文木に対するデータ走査処理の変更の考察

JCIA の高精度化のフィールド参照箇所の解析の変更におけるデッドコード検出機能の導入や，代入互換性に関する箇所の解析の変更におけるインスタンスの型の伝播の導入では，Visitor パターンを用いた．これら 2 つの変更では，抽象構文木の各ノードで値を伝播する処理を定義する必要があり，抽象構文木に対する処理の変更という特徴を持つ．過去の変更の対応表の Visitor パターンに相当する．

実際に対応表を基に変更した結果，Visitor パターンで構文木と処理を分離でき，変更の頻度が高い処理の変更に対する柔軟性，変更の頻度が低い処理対象の安定性を担保することができた．対応表の Visitor パターンは妥当だと考える．

4.2　既存の処理を利用した処理の変更の考察

フィールド参照箇所の解析の変更におけるデッドコードを含まない定数伝播の作成では，Facade パターンを用いた．Facade パターンを用いることで，既存の処理を変更すること無く，それらを複数組み合わせて新たな処理を定義することができた．また，今後，Facade パターンを用いて定義した複数の処理を組み合わせた処理を利用するさいは，Facade に問い合わせるのみで複雑な処理を容易に利用できる．**図 7** に新たな処理を追加するさいのクラス図を示す．

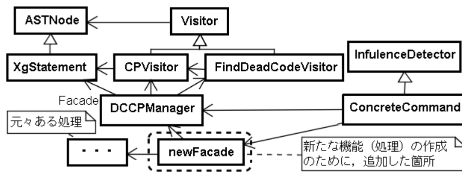

図 7　Facade パターン適用時に新たな処理を追加した場合

変更に柔軟でない既存の処理の安定性や，それらを複数用いた処理の使用性を担保できると考えた．対応表の Facade パターンは妥当だと考える．

4.3　既存の処理の拡張の考察

代入互換性に関する箇所の解析の変更におけるインスタンスの型の伝播の作成では，Decorator パターンを用いた．Decorator パターンを用いることで，既存の処理を変更せずに利用することができ，拡張する部分の処理のみを定義するだけで処理を拡張できた．更なる拡張をおこなうさいも，拡張する部分の処理を定義するだけで容易におこなえる．**図 8** に更なる拡張をおこなうさいのクラス図を示す．

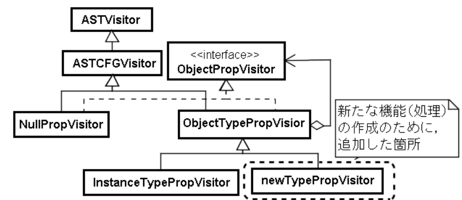

図 8　Decorator パターン適用時に更に拡張する場合

変更に対して柔軟でない処理の拡張性や安定性を担保できると考えた．結果として，対応表の Decorator パターンは妥当だと考える．

4.4　入出力の種類が同じ処理の変更の考察

変更箇所の解析の変更では，Strategy パターンを適用した．Strategy パターンを用いることで，変更箇所の解析のような，種類が同じ各処理を変更するさいに互いに影響を及ぼすこと無く変更することができた．

種類が同じ各処理の変更に対する柔軟性を担保することできると考えた．対応表の Strategy パターンは妥当だと考える．

4.5　対応表の一般性の考察

言語処理系は構文木のデータ構造とそれに対する処理で構成されており，JCIA と JCI でも同様の構造である．言語処理系は大別して，インタプリタとコンパイラの 2 つがある．JCIA と JCI はインタプリタにあたるが，コンパイラでも構文木を扱い，それに対して字句解析，構文解析，意味解析をする点は同じである．JCIA と JCI で共に対応表が適用できた理由として，検査対象データと処理の変更の頻度は異なり，検査対象データの安定性や処理の変更の柔軟性など保証すべき共通の要求があるからだと考える．これは言語処理系でも共通である．

われわれは JCIA と JCI の変更が，アーキテクチャのどの部分を対象とした変更であるかと，担保すべき性質を考えた．このことから一般的な言語処理系においても，データ構造と処理が分離された構造であり，担保すべき性質が同様であれば，対応表が適用できるといえる．

5　おわりに

本研究では，JCIA と JCI の過去の変更事例から，変更で担保すべき性質と変更方法を対応付けた．JCIA の高精度化を通じて，対応表の妥当性を検証し，対応表に無い種類の変更は複数の方法を比較して変更した．対応表について考察し，対応表の利用で性質を担保できることを示した．作成した対応表が他の言語処理系にも適用できるかを考察し，その結果，作成した対応表は言語処理系においても妥当であることを示せた．本研究では，言語処理系を変更するさいの指針を示せた．

今後の課題として，まだ示せていない変更の変更指針を示すことが挙げられる．今回 JCI と JCIA で示した変更以外についても検証していくことや，対応表の適用範囲についても考察することが挙げられる．

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] Robert S. Arnold and Shawn Bohner, *Software Change Impact Analysis*, Wiley-IEEE Computer Society Pr; 1st edition,1996.

[3] 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.