ソースコード縮退によるソースコード理解

神谷 年洋†, 石尾 隆‡, 植田 泰士*, 楠本 真二‡, 井上 克郎‡

本稿では, ソースコードを理解するための手法の一つとして, ソースコードの規模を擬似的に小さくすることでソースコードのレビューを容易にする手法を提案する. 提案する手法は, ソースコードに相互に依存する部品群が含まれる場合でも, その一部を取り出して調べることが可能である. さらに, 本手法を利用してインクリメンタルにソースコード理解を行うことができる. ケーススタディにより具体的な適用例を示すとともに, 本手法の有効性を評価する.

Source Code Degeneracy for Understanding Source Code

Toshihiro Kamiya†, Takashi Ishio‡, Yasushi Ueda*, Shinji Kusumoto‡, Katsuro Inoue‡

In this paper, we propose a method for understanding a source code, which makes the review task easier by reducing the scale of the source code. The method can extract the small and self-contained part form the source code, even when the entities in the source code depend on each other. Furthermore, this method can be used in an incremental process to understand of the source code by a review. A case study is shown as an evaluation of the proposed method and the incremental review process.

1. はじめに

ソフトウェアの保守作業においては, 不具合の原因を特定してソースコードを修正する, 機能拡張のための修正を行う, 処理上のボトルネックやセキュリティホールといった潜在的な欠陥を調べるなどのために, 保守作業者がソースコードを理解することが必要とされる.

ソースコードを理解する方法の一つに, ソフトウェアに関する文書を参照することがある. 仕様や設計を理解することで, ソースコードを理解する手がかりを得ることができる. もう一つの方法としては, オブジェクトコードやソースコードから情報を抽出する手法がある.

ソースコードの理解を目的として, ソースコードを分析する手法としては, (1)リバースエンジニアリング, すなわち, ソースコードから抽象度の高い情報を生成する手法(ソースコードから Unified Modeling Language のクラス図やコラボレーション図を生成するツール), (2)ソースコードから, 何らかの意味的なまとまりを抽出する手法(プログラムスライシング[6] [8]や Concept Lattice[1]), (3)ソースコードには明示的に記述されない情報を解析する手法がある(クロスリファレンス, ソフトウェア部品の分類[9], エイリアス解析[5], コードクローン分析[2]など).

この中でも, ソースコードから何らかの意味的なまとまりを抽出するプログラムスライシングや Concept Lattice といった手法では, ソフトウェアの適切な構成要素(手法に依存して, クラスやソースコード中の文)の依存関係を追跡することにより, ある要素に関係のあるすべての要素を抜き出すアプローチを取っている. 要素間に相互依存関係がある場合には, それらの要素は併合されるか, 依存関係のあるすべての要素が取り出されることになる.

本稿では, ソースコードを理解するための分析手法の一つとして, ソースコードの規模を擬似的に小さくすることでソースコードのレビューを助ける手法である「ソースコード縮退」を提案する. この手法は, 構成要素の間に依存関係がある場合に, 一定の基準によってそのような依存関係を切り離すことで, ソフトウェアの規模を見かけ上小さくする. これにより, 相互依存関係を持った要素群が含まれる場合でも, 少数の要素を取り出してレビューすることが可能になり, ソースコードの理解が容易になる.

†科学技術振興事業団 さきがけ研究 21
Presto, Japan Science and Technology Corporation
‡大阪大学 大学院情報科学研究科 コンピュータサイエンス専攻
Department of Computer Science, Graduate School of Information Science and Technology, Osaka University
*宇宙開発事業団
NASDA

2. ソースコード縮退とその利用

2.1. ソースコード縮退とは

ソースコード縮退(degeneracy)とは，あるソースコードに何らかの依存解析を行う際の前処理として，ソフトウェアを構成する要素(後述のケーススタディでは，クラスやコンポーネントが要素となる)群のうち，一部の要素を副作用のない擬似的な定数(以下，「擬似定数要素」)と見なすことである．これにより，(1)要素間に依存関係の循環があった場合には，その一部の要素を擬似定数要素に指定することにより，その循環を断ち切ることができる，(2)ソースコード中の擬似定数要素に関連する部分(あるいはその部分による影響)取り除くことができる，2 つの効果が得られる．

効果(1)は，ソースコード縮退を依存関係に基づく他の分析手法と併用するときに重要である．これにより，依存関係が循環しているような要素群に対して適用した場合に，その要素群の一部を取り出すことができるようになる．

効果(2)に関して，具体的には，以下のようなソースコードを取り除くことができる．取り除かれる部分を特定するアルゴリズムはコンパイラの定数伝播と同じである(「擬似定数要素」という名称は定数伝播に由来する)．

- 擬似定数要素のソースコード
- 擬似定数要素への参照
  例えば，擬似定数要素による条件分岐(if 文や throw 文)は，定数による分岐であるため，分岐の一方のみに制御が移ると仮定し，それ以外の部分のソースコードを取り除くことができる．
- 擬似定数要素を計算結果とするルーチン
  例えば，擬似定数要素を戻り値とし，副作用を持たない手続き(メソッド)は取り除くことができる．
- 擬似定数要素を格納するデータ構造
  例えば，擬似定数要素のみを格納する配列は取り除くことができる．

ソースコード縮退では，手作業により擬似定数要素を指定する必要がある．上述の理由により，依存関係の循環が発生している要素がその候補となる．

2.2. インクリメンタルなソースコード理解

ソースコード縮退(および他のソースコードの一部を抜き出す手法)を用いて，インクリメンタルにソースコードを理解することが可能になる．すなわち，(1)まず，他に依存しない少数の要素を抽出し，それらの要素をレビューによって理解する．(2)次に，すでにレビューした要素のみに依存する要素を抽出し，レビューにより理解する．これを，すべての要素をレビューし尽くすまで繰り返す．次章では，ケーススタディを用いてこのプロセスを具体的に説明する．

3. ケーススタディ

提案する手法を，大阪大学井上研究室で開発されたアプリケーション Gemini[7]に適用する．Gemini は GUI を備えたソースコード分析ツールである．Java で記述され，規模はソースコード約 1 万 5 千行，クラス 115 個，パッケージ 13 個である．Gemini のソースコードを理解するために，ソースコード縮退を用いながら，これらのクラスを順にレビューするプロセスを示す．

Gemini の開発者は，115 個のクラスをパッケージ(Java が提供するクラスの名前空間)によって適切に分類していたので，レビューもパッケージごとに行うこととする．まず，パッケージ間の依存関係を図 1 に示す．この図で，例えば，clonedata から utility に向かう矢印は，clonedata に含まれるあるクラスが utility に含まれるあるクラスに依存していることを意味する．図に示される依存関係によって，これらのパッケージのうち，filefilter(7

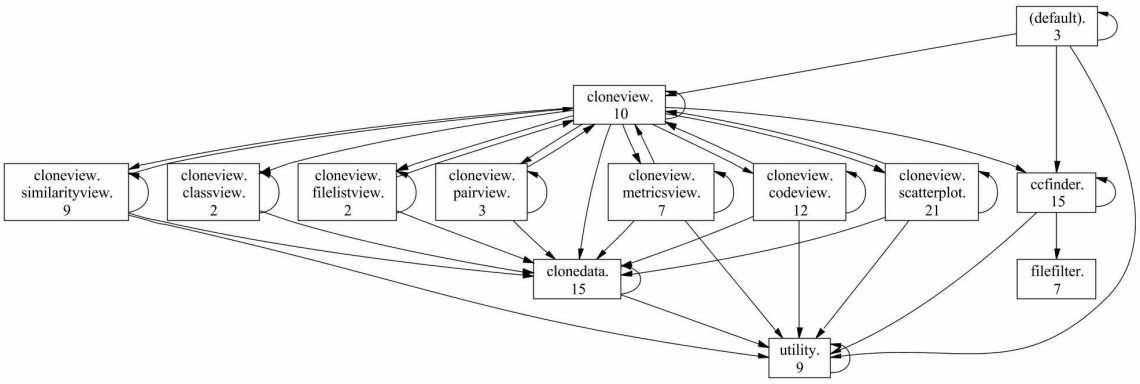

図 1 パッケージの依存関係と個々のパッケージに含まれるクラスの数

Fig. 2. The packages, the dependencies, and the number of the classes in each package

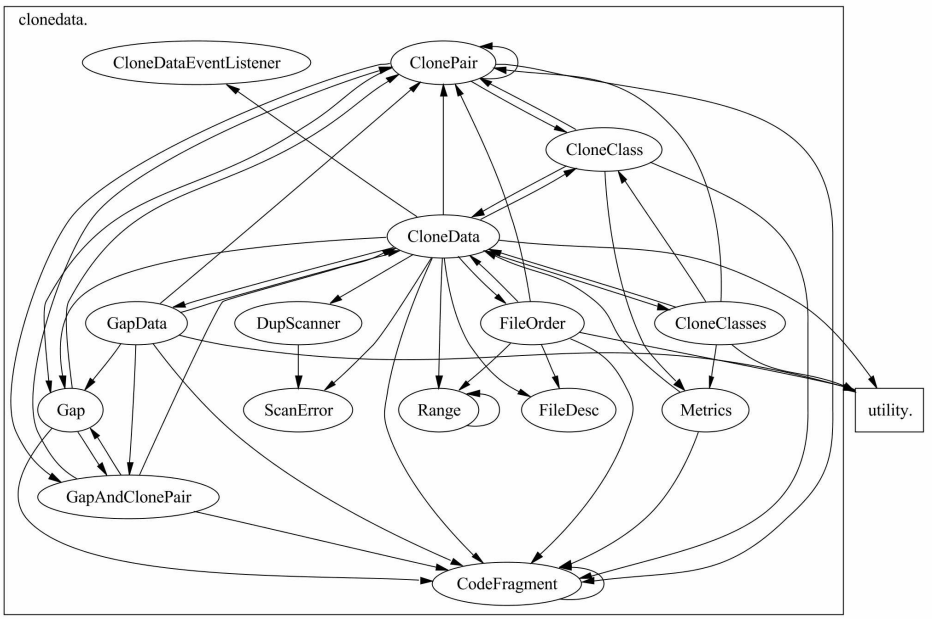

図 2 パッケージ“clonedata”に含まれるクラスの依存関係
Fig. 4. The dependencies of the classes in the package “clonedata”

個のクラスを含む), utility(同 9 個)は他のパッケージに依存していないことがわかる. 従って, これらのパッケージに含まれるクラスが最初にレビューされる.

次の段階として, パッケージ clonedata をレビューするために, 同パッケージに含まれる 15 個のクラス(およびインターフェイス)の依存関係を調べる(図 2). 半数以上のクラスが依存関係の循環(例えば, ClonePair → GapAndClonePair → CloneData → ClonePair)を起こしていることがわかる. これら依存関係の循環を起こしているクラス群{ CloneClass, CloneClasses, CloneData, ClonePair, FileOrder, Gap, GapAndClonePair, GapData, Metrics }の中でもっとも多くの依存を保持しているクラス CloneData を擬定数要素に指定してコード縮退を行うことにする. 適用結果を図 3 に示す. 依存関係の循環を起こすクラスは{ CloneClass, ClonePair, Gap, GapAndClonePair }に減少している. ソースコードの行数による評価として, コード縮退前後のソースファイルの行数を表 1 に示す. コード縮退によって, 依存関係が循環を引き起こしているクラス群の総行数は, 3265 行から 542 行へと減少していることがわかる. また, 行数が最大のソースファイル FileOrder.java は, 約 4 割の行が取り除かれることがわかる.

レビューの手順としては, 依存関係の循環を起こしているクラス群の 4 つのクラスを同時にレビューする必要があることを除けば, 疑似定数要素 CloneData 以外の 14 個のクラスのコード縮退後のソースコードを, 依存関係に基づいて一つずつ順にレビューすることになる. これら 14 個のクラスのレビューが終わった時点で, コード縮退によって取り除かれたていたソースコードをレ

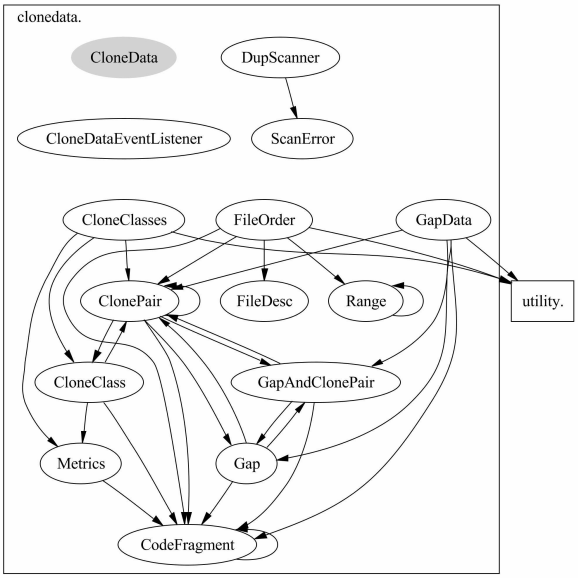

図 3 コード縮退を行った後の依存関係
Fig. 5. The dependencies after code degeneracy

ビューする. これによりパッケージ clonedata のレビューが完了する.

残ったパッケージに関しても, 同様の手順を行うことでレビューを行うことができる

4. まとめ

本稿では, ソースコードを理解するための分析手法の一つである「ソースコード縮退」を提案した. 本手法により, 他の要素への依存関係を持たない要素から順に, 一度に少数の要素をレビューしていくことで, インクリメンタルにソースコード全体をレビューすることが可能になる. ケーススタディでは, ある Java プログラムに本手法を適用することによって, そのようなレビューの具体例を示した.

今後の課題としては, 本手法を統合開発環境に組み込むことでより手軽に適用できるようにすること, 他の依存関係に基づく分析手法と組み合わること, あるいは, ソフトウェア分析のための枠組みである Reflection model[4]等において本手法を利用することがある. また, アスペクト指向技術によって開発されたプログラムでは, アスペクトを導入することにより, クラスとアスペクトの間に新たな依存関係の循環が生じる事例が知られており[3], そのようなソフトウェアをレビューする際にも本手法を用いることが可能であろう.

表 1. コード縮退前後のソースファイルの行数

Table 1. The LOC of each source file before/after the code degeneracy

Source file	LOC	LOC after Degeneracy
CloneClass.java	*109	*109
CloneClasses.java	*465	462
CloneData.java	*596	0
CloneDataEventListener.java	6	6
ClonePair.java	*129	*129
CodeFragment.java	79	79
DupScanner.java	195	195
FileDesc.java	19	19
FileOrder.java	*1217	741
Gap.java	*104	*104
GapAndClone.java	*202	*200
GapData.java	*228	127
Metrics.java	*215	208
Range.java	25	25
ScanError.java	7	7
Total	3596	2411
LOC in the cyclic dependency	*3265	*542

The figures with "*" mean that the source files include classes in a cyclic dependency.

参考文献

[1] Eisenbarth, T., Koschke, R., Simon, D., "Locating Features in Source Code," *IEEE Trans. Software Engineering*, vol. 29, no. 3, pp. 210-224 (2003).

[2] 井上克郎, 神谷年洋, 楠本真二: コードクローン検出法, コンピュータソフトウェア, Vol.18, No.5, pp.47-54, (2001).

[3] 石尾隆, 楠本真二, 井上克郎: アスペクト指向プログラミングのプログラムスライス計算への応用, 情報処理学会 論文誌 Vol. 44, No. 7, (採録決定).

[4] Murphy, G. C., Notkin, D., Sullivan, K., "Software Reflexion Models: Bridging the Gap between Source and High-Level Models," *ACM SIGSOFT '95: Proc. the 3rd Symposium on the Foundations of Software Engineering (FSE)*, pp. 18-28, Washington D. C. (Oct. 1995).

[5] 大畑文明, 近藤和弘, 井上克郎: エイリアスフローグラフを用いたオブジェクト指向プログラムのエイリアス解析手法, 電子情報通信学会論文誌 D-I, Vol. J84-D-I, No.5, pp. 443-453 (2001).

[6] 高田智規, 井上克郎, 芦田佳行, 大畑文明: "制限された動的情報を用いたプログラムスライシング手法の提案", 電子情報通信学会論文誌 D-I, Vol. J85-D-I, No. 2, pp. 228-235, (2002).

[7] Ueda, Y., Higo, Y., Kamiya, T., Kusumoto, S., and Inoue, K. "Gemini: Code Clone Analysis Tool," *Proc. 2002 International Symposium on Empirical Software Engineering (ISESE2002)*, vol.2, pp.31-32, Nara, Japan, (Oct. 2002).

[8] Weiser, M.: "Program Slicing", *Proc. the Fifth International Conference on Software Engineering*, pp. 439-449, (1981).

[9] 山本哲男, 横森励士, 松下誠, 楠本真二, 井上克郎: 利用頻度に基づくソフトウェア部品の解析・検索システムの提案, 電子情報通信学会技術研究報告, SS2002-17, Vol.102, No.329, pp.13-18 (2002).