社団法人 電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.
SS2000-52 (2001-01)

ソースコードの並び替えによる ソフトウェアの問題発見手法

寺井 淳裕[†] 内田 眞司[†] 島 和之[†] 武村 泰宏[†*] 松本 健一[†] 井上 克郎[†‡] 鳥居 宏次[†]
[†]奈良先端科学技術大学院大学 情報科学研究科
[†*]大阪芸術大学短期大学部
[‡]大阪大学 大学院基礎工学研究科

あらまし 本研究では、再利用ソフトウェアなどに潜在する問題を見つけるための方法として、ソースコードのオーバーホール(分解検査)を提案する。一般にオーバーホールとは、分解と統合の過程を通じて対象をよく理解し、その結果として故障などの原因を突き止めることである。本稿ではソフトウェアについてこのような作業を支援する環境を構築した。

キーワード ソフトウェア品質、再利用ソフトウェア、レビュー

A Method of Finding Software Problems by Rearranging Source Codes

[†]Atsuhiro Terai [†]Shinji Uchida [†]Kazuyuki Shima [†*]Yasuhiro Takemura
[†]Kenichi Matsumoto [†‡]Katsuro Inoue [†]Kouji Torii
[†] Graduate School of Information Science, Nara Institute of Science and Technology
[*]Osaka University of Arts Junior College
[‡] Graduate School of Engineering Science, Osaka University

Abstract In this research, we propose the software overhaul of source code, which is suggested to find problem hidden in reused software. We can understand an object well through the process of dismantlement and integration of software, moreover reasons of hardware trouble can be found through the result of overhaul. In this paper, we construct the environment which supports overhaul operation in software field.

Key words software quality, reused software, review

1. はじめに

オブジェクト指向技術やコンポーネントウェアに代表されるソフトウェア再利用技術の普及により、ソフトウェア開発プロジェクトにおいてプロジェクト外で開発されたソフトウェアを再利用する機会が増えている。再利用されたソフトウェアは新規開発されたソフトウェアに比べて一般的には信頼性が高い。しかし、再利用部分に問題がある場合や新規開発部分との間に不整合がある場合などは、開発者が再利用部分を十分理解していないと問題の原因を突き止めることが困難になる。

また、開発者が不充分な理解のまま問題を回避するために対処療法的な修正をすると、その修正により別の問題が発生するという状況に成りかねない。さらに、レガシーソフトウェア(開発されてから非常に長い時間が経過した場合など、開発当時の担当者が組織からいなくなり、組織で受け継がれているソフトウェア)では、多数の開発者による度重なる修正によって大部分が理解困難となり、問題が頻発することになる。

本研究では、再利用ソフトウェアやレガシーソフトウェアに潜在する問題を見つけるための方法として、ソフトウェアのオーバーホールを提案する。

一般にオーバーホールとは、ハードウェア分野でよく用いられるアプローチである。例えば、故障車の修理において故障の症状からその原因部分を特定できない場合、オーバーホール(分解検査)が行われる。また、車が古くなり、全体的に調子が悪くなった場合にもオーバーホールが行われる。修理担当者はオーバーホールにおける分解と統合の過程を通じて、対象の故障車を理解し、理解の結果として故障の原因を突き止めていると考えられる。

ソフトウェアの分野では、チェックリストを用いた設計レビューやコードレビューなどのようにハードウェアにおける点検に相当する作業は行われているが、分解し統合するというオーバーホールに相当する作業は行われていない。また、従来の設計レビューやコードレビューなどの検査では問題個所を見つけることに主眼が置かれていたが、本研究では対象ソフトウェアを理解するということに特に注目している。なぜなら、統合に時間がかかった部分や統合できなかった部分は、その開発者にとって理解が難しかったことを意味するからである。よって、そのような部分を優先的に、より熟練した開発者が再オーバーホール、再レビュー、再設計することにより、効率的にソフトウェアを改善できると予想される。また、本手法の効果が確認されれば、学生や新入社員を対象としたソフトウェア教育にも有効と考えられる。

以下、まず第 2 章においてソースコードについてのオーバーホールの具体的な方法を提案する。第 3 章では第 2 章で述べたオーバーホールを支援するための環境について詳しく述べる。さらに第 4 章で、提案手法の適用例について説明する。これから、オーバーホールにかかる時間、オーバーホールの過程で見つけることのできる問題についての考察を行う。最後に、第 5 章でまとめと今後の課題について述べる。

2. 並び替えを用いたソースコードのオーバーホール

本研究で提案するソフトウェアのオーバーホールでは、ソースコードの分解、統合の具体的な方法として、ステートメントを一つの部品と考えたソースコードの並び替えを用いる。

2.1 概要

ソースコードの並び替えには、1.プログラムの仕様を表したドキュメント、2.コードの一部が抜き出されたプログラムソースコード、3.抜き出された部分を分解しランダムに表示した選択肢の三つが必要になる。

手続き型のプログラム言語では、ソースコードから、たとえばステートメントをひとつの単位(部品)として分解した場合でも元のソースコードを復元することができる。なぜなら、それらの分解されたステートメントの間には、制御の流れやさまざまな変数の関連などが存在するからである。

```
/******************************
name:ymd2rd2
In:  int yy,mm,dd 年,月,日を入力
Out: int 基準日から入力された
          年月日までの日数
******************************/
01: int ymd2rd2(int yy,int mm,int dd){
02:     register int  cnt;
03:     int ret;
04:     ret = 0;
05:     for (cnt = 1992; cnt > yy; --cnt) {
06:         ret -= getdofy(cnt);
07:     }
08:     for (cnt = 12; cnt >= mm; --cnt) {
09:         ret -= getdofm(yy,cnt);
10:     }
11:     ret -= dd + 1;
12:     return(ret);
13: }
```

図1 カレンダープログラムのあるモジュール

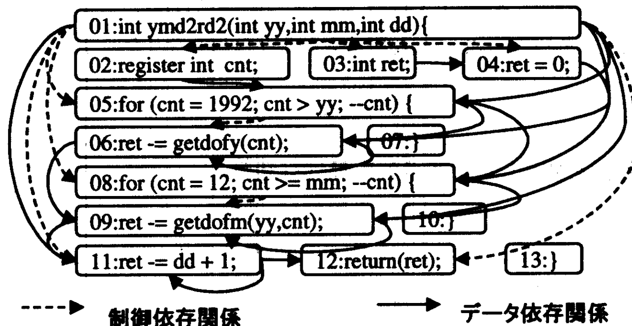

図2 図1のPDG

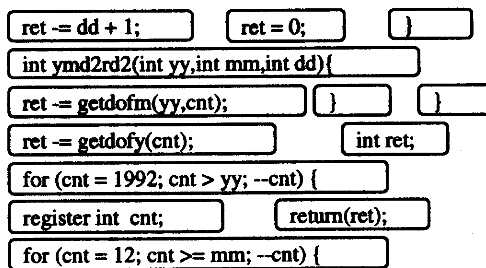

図3 並び替えられたPDGのノード

以下、実際のコード(図 1)を使ってそのステートメント間の関連や制御の流れについて説明する。

図 2 は図 1 のソースコードの PDG (Program Dependence Graph) である。

PDG とは、基本的にソースコードの各ステートメントをノードとし、その間を制御依存関係、データ依存関係といったアークを使って結んだ有向グラフであり、プログラムの内部表現を明示することができる[4][5]。このことから分かるように、ソースコードからは、ステートメント間に変数のデータ依存関係や、条件文からの制御依存関係などを読み取ることができる。そのため図 3 のようにステートメントを抜き出しバラバラに並び替えた場合でも、それらのステートメント(部品)を一つ一つ吟味し、残りのステートメントとの依存関係を見つけ出すことで、コード全体を理解し[3]統合を行うことが可能である。

よって、並び替えの過程を通じてステートメント間の依存関係を発見し、ソフトウェア全体への理解を深めることが、オーバーホールにおける統合の過程であると考えることができる。

2.2 作業者による問題点の発見

分解された状態のソースコードから、作業者が元の仕様にあるようにコードに統合するためには、ステートメント間の依存関係を見つけ、そのソフトウェアへの理解を深めなければならない。ソースコードの並び替えによるオーバーホール、すなわちステートメントを一つ一つ吟味して依存関係を理解していく過程が、自動車などのオーバーホールと同様に、検査されているソフトウェアの問題点を発見することに役立つと考えられる。言い換えれば、ソフトウェアのオーバーホールで、仕様ドキュメント、選択肢、残りのソースコードから、遮断されているデータや処理の流れを見つけ出して理解し、その発見の過程からソフトウェアの問題点の有無を確かめることになる。

また、元のソースコードを統合するという明確な目的が与えられることによって、先入観による読み飛ばしなどを少なくすることができると考えている。

2.3 履歴を利用した問題点の定量的な発見

作業者がオーバーホールする際には、抜き出されたステートメントの役割やステートメント間の因果関係を見出し、頭の中でその処理の過程をなぞる、または、ドキュメントにある仕様を満たす上でそれぞれのステートメントの果たしている役割を考えるなどの試行錯誤を伴う。作業者はこれらの問題解決の過程を通じてプログラム全体を段階的に把握、理解していると予想される。この理解の進捗状況が、並び替えの作業時間や、答え合わせをした際の誤りの数などに現れると考え、これらを履歴として収集する。ドキュメントやソースコードに問題点があった場合、作業者がソフトウェアを理解できない、あるいは理解に時間がかかり、並び替えが困難な状況に陥ったことがこの履歴を調べることで発見できると考えている。

3. オーバーホールツール

ソフトウェアのオーバーホールを並び替えで行う利点には、分解されたコードの生成、元のソースコードとの比較、作業履歴の収集などが容易であること、作業時間が比較的短くてすむことが上げられる。

ソフトウェアのオーバーホールには、この特徴を十分生かせるようなソースコードの分解ツールとソースコードの統合を行う並び替えツールが必要である。

3.1 ドキュメント

オーバーホールで使用するドキュメントは、基本的に元のソースコードを作成するために必要となった仕様書であるが、主に機能仕様書[3]を想定している。すなわち、モジュールの機能的、性能的な役割と、どんな入力によってどんな出力を生ずるかを明記したものである(図 4)。

```
name:ymd2rd2
In:  int yy,mm,dd 年,月,日を入力
Out: int 基準日から入力された年月日までの日数
-----------------------------------------------------
name:getdofm
In:  int yy,mm  年,月を入力
Out: int 閏年の 2 月ならば 29 を返す
         その他は月の日数を返す
```

図 4 機能仕様書の一部

3.2 コード分解ツール

ソースコードを分解するためには以下のような点を考慮しておかなければならない。

3.2.1 ソースコード

主に機能仕様書をドキュメントとして利用するため、ひとつのモジュールをオーバーホールのためのソースコードにしている。別のモジュールがコード中で参照されている場合は、そのモジュールの機能仕様書も与える。

3.2.2 単位

ソースコードから抜き出される内容はトークン単位、ステートメント単位、ブロック単位などが考えられる。抜き出されるサイズが大きくなれば並び替えはより簡単に、小さくなれば一からの再コーディングに近くなり並び替えの作業に手間が掛かる。また、オーバーホールにおける一つの部品として抜き出される単位にはそれ自身にある程度の意味のまとまりが必要となる。今回は使用するソースコードの大きさを一モジュール程度としていることから、ステートメントもしくは複合ステートメントを原則として単位に設定した。さらにこれは第 2 章で述べたようにステートメント間にデータ依存関係や制御依存関係が存在することからも都合がいい。

ただし一行に複数のステートメントがある場合や、引数などがうまく一行に収まらず、二行に分けて記述されて

いる場合、ステートメントの前後の行にコメントが併記されている場合なども多々あるので、任意のブロックを一つの単位とする抜き出しについても対応している。

3.3 並び替えツール

並び替えツール(図 5)では、オーバーホールにおける統合、すなわち、ドキュメントをみながら、分解されて並び替えられたソースコードを復元する。また、この統合の過程を履歴として自動収集し、理解性の定量的評価のために利用する。

3.3.1 ユーザーインターフェース

図はオーバーホールのための並び替えツールの外観である。これは下記の 3 つのフレームから構成されている。

A. コードフレーム

ステートメント、もしくは行を単位として複数の部分が抜き出されたソースコードを表示する。空行となっている部分にステートメントを挿入し、元のソースコードを統合する。

ステートメントが挿入される際、もとのソースコードとおなじになるようにインデントをつけてしまうと、ソースコードの内容を考えず、インデントに従って機械的に並び替えを完了させることができる場合がある。よって、インデントは選択肢の内容と挿入された場所によって決定し、並び替えが行われるたびに再描画される。

B. 選択肢フレーム

元のソースコードから抜き出しされたステートメントをランダムに並び替えて選択肢として表示する。このステートメントをマウスで選択し、コードフレームにドロップすることで並び替えを行う。

ここでもインデントをそのまま表示しておくと、コードフレームと同様の問題が起こるので、インデントはすべて削除されている。

C. チェックフレーム

作業者が並び替えを完了したと考え、チェックボタンを押したとき、元のソースコードが再現できているかを確認し、その結果(一致しなかったステートメントの数)を表示する。

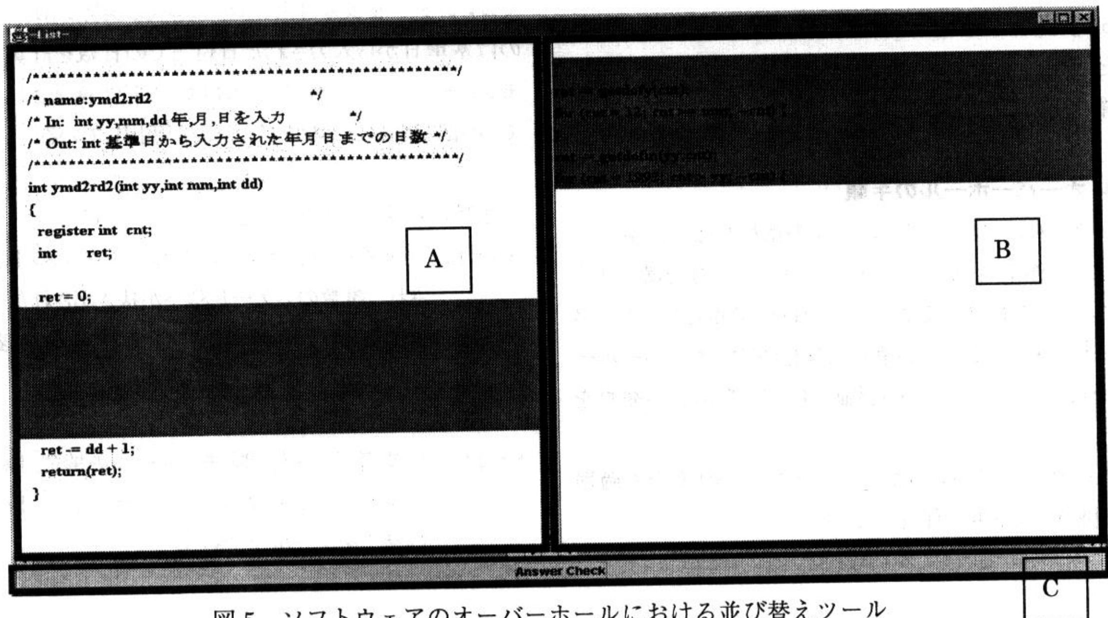

図 5　ソフトウェアのオーバーホールにおける並び替えツール

3.3.2 履歴収集

作業者が各選択肢やチェックボタンをマウスでクリックしたとき、以下の二種類がソースコードの統合過程の履歴として収集される。図 6 は履歴の例である。

- 一つ前の操作からの経過時間(秒)、ステートメントが選択されてから並び替えられるまでの時間(秒)、挿入された位置(行番号)、選択されたステートメントの内容
- 一つ前の操作からの経過時間(秒)、元のソースコードとの比較を行ったことを示すフラグ、比較の結果一致しなかったステートメントの数

```
01.97,01.38,13, }
21.75,01.54,14, ret -= getdofm(yy,cnt);
01.92,answer check, 0
```

図 6 収集する履歴情報の例

図 6 の二行目について説明すると、ステートメント" ret -= getdofm(yy,cnt);"が選択されたのはステートメント"}"が並び替えられてから 21.75 秒後で、この部分がコード中の 14 行目に挿入され並び替えが完了するまでには 1.54 秒掛かったことが分かる。三行目は比較が行われたことを表し、不一致が 0、よってすべて一致し、並び替えが終わったことを示している。

3.4 オーバーホールの手順

ソフトウェアのオーバーホールの過程を図 7 に示す。

まず、作業者は、ドキュメントを読み、仕様を満たすようにコードを統合するよう求められる。分解されたソースコードが表示され、作業者はこれを通じでオーバーホールを行い、ソフトウェアの評価、および問題点の発見を行う。

この時、システムではオーバーホールの統合の過程を履歴として収集し保存している。

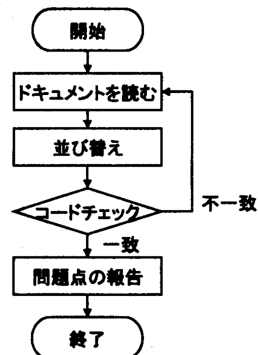

図7 オーバーホールのフローチャート

作業者が、並び替えが終了したと考えたとき、チェックボタンを押すことで、元のソースコードと統合されたソースコードの比較が行われ、一致しない場合は、その不一致の数が表示される。

4. 適用例

今回作成したツールについて適用例を述べる。作業者は情報系の大学院助手1名で、約7年のプログラミング経験がある。オーバーホールに使用したプログラムは、C 言語でかかれている。これは入力された年月に基づいてカレンダーを表示するプログラムで、実際に使用したのは基準日から入力された日付までの日数を計算するモジュールである (図 1)。全体は 14 行で構成されていて、内部で別の2つのモジュールを呼出している。

手順は以下のとおりである。

Step1: 作業者には、対象プログラムのドキュメントが渡され、複数のステートメントが抜き出されたモジュールのソースコードが PC 上のツールに表示される。

Step2: 作業者は、仕様書とモジュールの機能仕様書を参照しながら提示されたステートメントの並び替えをツール上で行う。

Step3: 並び替えが終了すると作業者は、ツールを用いて元のソースコードとの比較を行う。すべて一致すればStep4 へ、不一致ならば Step2に戻る。

Step4: オーバーホール終了後、どのような問題点が対象ソフトウェアに存在したかを作業者にインタビューすることで調べる。

並び替え作業そのもの(Step2、3)には 42 秒かかった。その際の履歴は図 8 のようになった。

```
00.00,01.53,12, for (cnt = 1992; cnt > yy; --cnt) {
20.44,01.37,13, ret -= getdofy(cnt);
04.56,02.96,14, }
01.87,01.16,15, for (cnt = 12; cnt >= mm; --cnt) {
01.97,01.38,16, ret -= getdofm(yy,cnt);
01.75,01.54,17, }
01.92, answer check, 0
```

図 8 オーバーホールの履歴

さらにこのカレンダープログラムプログラムについて、インタビューによって作業者が明らかにした問題点を述べる。

ドキュメントの問題

- モジュールが実行される条件(このモジュールは 1992 年以前の年月日についてのみ処理すること)が明記されていない。
- 返される日数の正常値が負の値であることを記していない。

ソースコードの問題

- 日数を計算する為の基準日がマジックナンバーとなっている (05 行目)。
- マイナス記号とプラス記号を間違えている(11 行目)。

つぎにこの結果をもとに提案手法とツールの評価と問題点について考える。

提案手法について

開発ツールを用いたソフトウェアオーバーホールの生産性は、およそ 20LOC/分であった。プログラマ人・月当りのコーディング行数を約 1KLOC とする[1]と、それをオーバーホールするには 50 分～204 分を要することになる。

さらに、ドキュメントとソースコードの両方について作業者は問題点を見つけることができた。

また、履歴によるとこの作業者は、コード中の空欄を上から順番に並び替えて埋めていく中で、図 4 の二行目の部分を並び替える前に、他と比べて多くの時間を費やしていることが分かる。このことを再度作業者にインタビューすると、この個所で時間を費やしたのは、モジュールの名前(getdofy()と getdofm())が似ているため、なにが違うのか戸惑ったこと、さらに仕様書でそれらの機能を確認することに時間がかかったためと分かった。

これは、モジュール名を短縮しすぎて、ソフトウェアの理解が難しくなっていることからくる問題であると考えられる。

ツールの問題点について

現状では、ステートメントの順序そのものが元のソースコードと一致しているかをチェックしている。しかし、この方法では、並び替えるステートメントをお互いに入れ替えてもモジュールの機能が同じ場合は判定できないことになる。

5 まとめと今後の課題

本研究では、再利用ソフトウェアなどに潜在する問題を見つけるための方法として、ソースコードのオーバーホール(分解検査)を考案し、その具体的な方法としてコード中のステートメントを並び替えることを提案した。またこ

の作業を支援する環境を構築した。

この手法を用いることで、ソフトウェアのチェックを行う作業者は、コードの統合という具体的な目標を持つことができ、ソフトウェアの理解が進むと考えている。またこの統合の過程を理解の過程と考え、履歴として収集している。この履歴からどのようなコードのチェックが行われたかを再確認することが可能であると思われる。

さらに提案手法と支援環境の適用例で、オーバーホールに掛かる時間について考察した。これにより、ソフトウェアのオーバーホールで作業者が理解しにくかった個所がそのソフトウェアの問題点である可能性を指摘している。

今後の課題としては、ツールの、並び替えるステートメントをお互いに入れ替えてもモジュールの機能が同じ場合は判定できないという問題点が上げられる。解決方法としては PDG による依存関係を比較する方法が考えられる。この場合、PDG の定義から上記のような問題は起こらない。

参考文献

[1] C.Jones, "システム開発の生産性", 井上義祐, 荒木淳三監訳, pp8-13, マグロウヒルブック, 1986.

[2] 柏原昭弘、寺井淳裕、豊田順一: "いかにプログラム空欄補充問題を作るか?"、信学技報、ET99-116 (1999-5)

[3] 管野文友、ソフトウェア・デザインレビュー、pp38-40、日科技連出版社、1982

[4] 下村隆夫、"Program Slicing 技術とテスト、デバッグ、保守への応用"、情報処理学会、Vol.33、No.9,pp.1078-1086 (1992)

[5] Horwitz,S.,Reps,T., and Binkley,D.:Interprocedural Slicing Using Dependence Graphs, ACM Transactions on Programming Languages and Systems, Vol.12,No.1,pp.26-60 (1990)

福山写真製版所