

ソフトウェアオーバホール手法の実験的評価

内田 眞 司^{†1} 島 和 之^{‡2}
 武村 泰 宏^{‡3} 松本 健 一^{‡4}

ソフトウェアの保守の効率を高めるうえで、ソフトウェアの理解容易性を向上させることは有効な手段の1つである。本稿では、我々が提案したソフトウェアオーバホール手法を適用することでソフトウェアの理解容易性に問題がある箇所を発見できることを確認するためにに行った実験について報告する。提案手法は、作業者がソフトウェアを理解するプロセスを計測する手法で、ソフトウェアの分解と再統合から構成される。ソフトウェアの分解では、ソフトウェアをコンポーネントに分解する。再統合では、分解されたコンポーネントを作業者が元どりのソフトウェアへ再統合する。このソフトウェアを再統合する過程を分析することによりソフトウェアに含まれる理解容易性に問題がある箇所を発見することが可能になる。提案手法により発見した問題が理解容易性の問題であったことを確認するために、発見した問題を修正した後のプログラムと修正する前のプログラムのデバッグに要する時間を比較する実験を行った。実験の結果、提案手法によって発見した問題を修正した後のプログラムの方が、デバッグに要した時間が短くなったことを確認した。これにより、デバッグ作業の効率を低下させる理解容易性の問題を提案手法により発見できることを示した。

The Experimental Evaluation of the Method of Software Overhaul

SHINJI UCHIDA,^{†1} KAZUYUKI SHIMA,^{‡2} YASUHIRO TAKEMURA^{‡3}
 and KEN-ICHI MATSUMOTO^{‡4}

Improving software understandability is one of effective methods to enhance efficiency of software maintenance. This paper describes debugging experiments to confirm that software overhaul we proposed can be used to improve software understandability. Software overhaul is a method for externalizing process of understanding software and consists of deconstruction and reconstruction. Deconstruction is to take a software system apart to components. Reconstruction is to reproduce the software system by putting the components together again. Analyzing the history of activities of reconstruction, we can find issues of understandability in software. In the experiment, subjects debugged software systems which applied and not applied software overhaul. We compared time for debugging software systems which applied and not applied software overhaul and confirmed time for debugging software systems which applied overhaul is shorter than the others. The results show overhaul can be used to find issues of understandability that reduce efficiency of debugging.

1. はじめに

ソフトウェアの保守工程では、欠陥（フォールト、バグ）の除去、ユーザの要求による機能追加、ハードウェアやOSなどの変化に対する適応のため、対象ソフトウェアを変更する必要があるが生じる。保守担当者が対

象ソフトウェアの開発に参加していた場合、対象ソフトウェアをある程度理解しているため、正しく変更することは比較的容易である。しかし、近年では企業間、あるいは、企業内の部署間での人員の流動化や、保守作業そのものを外注して他企業に任せるケースが増えている¹²⁾。このようなケースでは、対象ソフトウェアをあまり理解していない技術者が保守を担当するため、保守作業において対象ソフトウェアを理解する必要がある。

保守担当者が不十分な理解のまま、対象ソフトウェアの欠陥を回避するために対症療法的な変更をすると、設計ドキュメントとソースコードとの間の一貫性が失われ、ソフトウェアの理解が困難となる。また、

†1 奈良工業高等専門学校
Nara National College of Technology

‡2 広島市立大学
Hiroshima City University

‡3 大阪芸術大学
Osaka University of Arts

‡4 奈良先端科学技術大学院大学
Nara Institute of Science and Technology

ソフトウェアの理解が難しいために保守担当者が試行錯誤で変更を行うと、新たな欠陥が混入する可能性が高く、ソフトウェアの信頼性が低下する。よって、ソフトウェアの理解容易性（理解しやすさの品質特性）は保守性だけでなく信頼性を確保するためにも重要といえる。Porter らによるコード検査の実験では、実際の開発現場で5年以上の経験を持つレビュー担当者が指摘した問題のうち60%が理解容易性に関する問題であったことが報告されている¹¹⁾。この結果は、経験のあるレビュー担当者がソフトウェアの理解容易性を重視していることを示している。

ソフトウェアの理解容易性を向上させる方法として以下のようなものがあげられる。

- プログラミング言語を使用する。
- プログラムを構造化する。
- アルゴリズムを単純にする。
- 設計ドキュメントを残す。
- 標準的なコーディング規約に従う。
- ソースコード中にコメントを入れる。
- 変数や関数の名を分かりやすくする。

ただし、これらの方法を適用する必要があるか否か、適用した結果が十分かどうかを判断するためには、理解容易性を評価することが必要である。

我々は、ソフトウェアの理解容易性を評価することを目的として、ソフトウェアオーバホール手法を提案している^{13),16)}。提案手法は、作業者がソフトウェアを理解するプロセスを計測する手法で、ソフトウェアの分解と再統合から構成される。ソフトウェアの分解では、ソフトウェアをコンポーネントに分解する。その際、分解されたコンポーネントはその順序をランダム化する。再統合では、分解されたコンポーネントを元どおりのソフトウェアへ再統合する。ソフトウェアを再統合する過程を分析することによりソフトウェアに含まれる理解容易性の問題点を発見することが可能になる。

本稿では、提案したソフトウェアオーバホール手法を適用することでソフトウェアの理解容易性に問題がある箇所を発見できることを確認するためにに行った実験について報告する。実験では、提案手法により発見した理解容易性の問題を修正する前のプログラムと修正した後のプログラムのデバッグ作業に要する時間を比較する。

以下、2章では、ソフトウェアの理解容易性について述べ、3章では、ソフトウェアオーバホール手法について詳細に述べる。4章では、実際に行った評価実験について述べる。5章では実験結果とその分析につ

いて述べ、6章では考察を述べる。7章では関連研究について述べ、8章でまとめと今後の課題について述べる。

2. ソフトウェア理解容易性

本稿で述べるソフトウェアの理解容易性とは、ソフトウェアの分かりやすさの特性を示す。Boehm らの品質モデルでは、理解容易性の上位の品質特性には保守性があり、下位には以下の品質特性がある^{1),14)}。

- 無矛盾性 (Consistency): ソフトウェアの表記法、用語、記号などが統一されている度合い。
- 構造化 (Structuredness): ソフトウェアの相互に関連した部分の構成が、パターン化されている度合い。
- 自己記述性 (Self-descriptiveness): ソフトウェアがその目的、仮定、制約、入出力、要素、状態などに関して、十分な情報を含んでいる度合い。
- 簡潔性 (Conciseness): 不必要な情報を除いた、必要な情報だけが備わっている度合い。
- 明瞭性 (Legibility): コードを読むことによって、ソフトウェアの機能などを容易に認識できる度合い。

なお、ISO/IEC9126⁵⁾において理解性は、ソフトウェアの論理的概念などについての利用者側の理解のしやすさ、と定義されている。ソフトウェア開発者にとっての分かりやすさの特性は解析性として、故障などの原因診断、または改訂すべき部分の識別に必要な労力に関することと定義されている。本稿で述べる理解容易性はISO/IEC9126における解析性を意味する。

保守工程における問題の要因のうち、大部分を占めるのはソフトウェアの理解の困難さである。このことは、保守工程において保守作業者が、多くの時間をソフトウェアの理解に割いているといったことから確認できる。本稿では、これらの特性を悪化させる要因を理解容易性の問題と呼ぶ。具体的には、ドキュメントの問題として、記述の誤り、複雑な構成、あいまいな表現などがあげられる。また、ソースコードの問題として、複雑な構造、大きすぎるモジュール、マジックナンバーの使用などがあげられる。

ソフトウェアの開発において高い理解容易性を持つソフトウェアを開発することが保守コストを削減するための重要な手段の1つである。また、理解容易性が低いソフトウェアは、レビューにおいてはレビュー担当者の理解不足によりレビュー後にフォールトが残りやすく、デバッグにおいてはデバッグ担当者の理解不足によってフォールトが見つかりにくい、と考えら

れる .

3. ソフトウェアオーバホール手法

3.1 ソフトウェアオーバホールの概念

ハードウェアの分野において、オーバホールは保守作業の1つとして行われる。この作業では、機械がつねに効率の良い経済的な運転を維持できるように、機械を分解して点検することにより正常な状態かどうかを確認し、性能回復のために必要な処置を施す。

我々が提案するソフトウェアオーバホール手法とは、作業者がソフトウェアを理解するプロセスを計測する手法で、ソフトウェアの分解と再統合から構成される。ソフトウェアの分解では、ソフトウェアをコンポーネントに分解する。ここでコンポーネントとは、プログラム中の関数やステートメントを指す。分解されたコンポーネントの順序はランダム化される。再統合では、分解されたソフトウェアを作業者が元どおりに再統合する。再統合の作業履歴を分析することでソフトウェアに含まれる理解容易性の問題点を発見することができる。なお、ソフトウェアオーバホール手法自体は、ソフトウェアの修正や変更は含まない。

図1は、あるプログラムをオーバホールする様子を示している。元のプログラム(図1(a))を分解することにより、ステートメント単位でランダムに並べられる(図1(b))。作業者は、ドキュメントやそれぞれのステートメントが持つ役割を理解しながら、ステートメントどうしの関係を考慮して、ステートメントを並べ替えることで元のプログラムへと再統合する(図1(c),(d))。再統合作業において、1度に全体を理解することは困難である。実際には、段階的にソフトウェアを理解し、最終的に全体を理解することになる。この段階的な再統合の作業の履歴を分析することで、ソフトウェアの中で理解しにくい部分を明らかにする。たとえば、再統合するまでに長い時間を要した部分は、作業者にとって理解が困難であったことを意味する。このような理解容易性が低い部分を優先的に再レビュー、再設計、修正を施すことにより、効率的にソフトウェアを改善することが可能となる。また、ソフトウェアオーバホール手法におけるソフトウェアの再統合という具体的な目標が定められた作業により、他人の作成したソフトウェアを理解するときに陥るモチベーションの低下や、先入観からくる読み飛ばしなどがある程度避けることができる。

3.2 ソフトウェアオーバホール環境

図2にソフトウェアオーバホールを行う環境を示す。PC上に用意されたソフトウェアオーバホールツール、

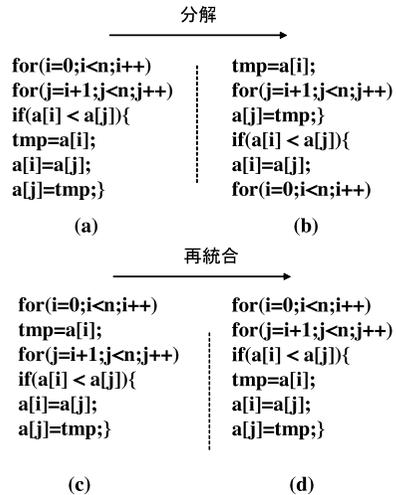


図1 ソフトウェアオーバホールの適用例
Fig. 1 The sample of software overhaul.



図2 ソフトウェアオーバホール環境
Fig. 2 The environment of software overhaul.

ソフトウェアオーバホールを行うソフトウェアの仕様を表したドキュメントや言語マニュアルが作業者に用意される。作業者はツールによってステートメント単位で分解されたソフトウェアの再統合を行う。ドキュメントを読みプログラムを理解しながら、分解されたプログラムのステートメントを元どおりに並べ替える。その際ツールでは並べ替えの過程を履歴として収集している。並べ替えが完了した時点で、プログラムのステートメントの位置が元どおりに組み立てられたかをツールにより判定する。すべてのプログラムのステートメントを正しく並べ替えることができたなら、再統合作業は終了する。

3.3 ソフトウェアオーバホールツール

我々は、ソフトウェアの分解を自動的に行い、再統合作業の履歴を自動的に収集するソフトウェアオーバホールツールを開発した¹⁵⁾。図3はソフトウェアオー



図 3 ソフトウェアオーバホールツールの画面
Fig. 3 The screen of software overhaul tool.

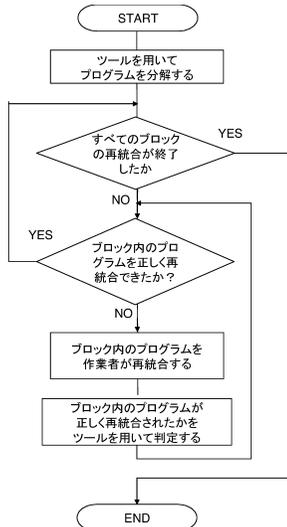


図 4 ソフトウェアオーバホール手法の手順
Fig. 4 The procedure of software overhaul.

パホールツールの画面の一例を示し、図 4 にソフトウェアオーバホール手法の手順を示す。プログラムの中で、背景が灰色のステートメントが再統合の対象となる部分である。灰色の濃さが異なる部分は、異なるブロックを意味する。ここでブロックとは 10 行未満のステートメントを意味する。再統合するステートメント数が多いほど再統合に要する時間が増加する¹⁵⁾。そこでオーバホールするステートメントが 10 行以上の場合、数個のブロックに分割する。ブロックごとにオーバホールを実施することで、作業者の負担を軽減させ理解容易性の問題以外の要因を排除することが期待できる。灰色の濃度が同じブロック内で、ステートメントの順番がツールによって自動的にランダムに並べ替えられている。作業者は、ブロックごとに行を並べ替える。作業者が判定ボタンをクリックすると、元のプログラムと同じステートメントは正しいと判定されて背景が白くなり、異なるステートメントは誤って

いと判定されて灰色のまま残る。作業者は正しくステートメントを並べ替えるまで作業を継続する。なお、再統合されたプログラムが正しいか否かはツールの機能上、ステートメントの順序そのものが分解前の順序と一致しているか否かを判断している。

ただし、現状のツールではインデントや括弧を手がかりにしてしまうと、プログラムを理解しないまま再統合できる可能性がある。解決策としては、再統合の作業者にインデントを入力させたり、ツールで作業者が並べ替えたソースコードに基づいたインデントを挿入したりする方法が考えられる。しかし、これらの解決方法は再統合をより困難にすることが予想される。なぜなら再統合の作業では、作業者はインデントをヒントにして全体の大まかな構成を把握してから細かな統合に移る様子が観察されたからである。ハードウェアのオーバホールにおいては、全体の構造と部品の機能はその「形」が有効であり、たとえば実際に直接はめ込んでみれば、ぴったりあてはまるかどうかは感触で分かることがある。したがってインデントや括弧を手がかりにして統合を行うことは、ハードウェアのオーバホールと対応していると考えられる。ただし、インデントや括弧によるヒントが、理解容易性を評価するうえでどの程度影響があるのか分析する必要がある。

4. 評価実験

評価実験の目的は、ソフトウェアオーバホール手法を適用することでソフトウェアの理解容易性に問題がある箇所を発見できることを確認することである。実験では、提案手法によって発見された問題点を修正した後のプログラムと、修正する前のプログラムのデバッグに要する時間を比較する。デバッグに要した時間がプログラムの理解容易性に影響されることから、問題点を修正した後のプログラムのデバッグに要する時間が短くなることにより、本手法によって発見された問題が理解容易性の問題であることを明らかにする。本稿では、問題を修正する前のプログラムを $M_j (j = 1, \dots, 4)$ 、修正した後のプログラムを M_j' ($j = 1, \dots, 4$)、被験者を $G_i (i = 1, \dots, 12)$ 、3 種類の実験を $EU1, EU2, EU3, 1 \sim 4$ 回目のデバッグ作業を $D1, D2, D3, D4$ と定義する。

4.1 プログラムとフォールト

プログラム M は、Lisp の S 式を構文解析しメモリ中にリストデータを生成する¹⁰⁾。プログラム M は、C 言語で記述され、全体のサイズが約 2,000 行で 40 個の関数から構成されている。プログラム M に異なるフォールトを 1 つずつ埋め込んだ 4 つのプログラム

M_j ($j = 1, \dots, 4$) を作成した。埋め込んだフォルトの種類は、条件式の誤り、ポインタ変数の誤り、ポインタの初期化の誤り、である。各フォルトは互いに独立である。

4.2 被験者

被験者 G_i ($i = 1, \dots, 12$) は、奈良先端科学技術大学院大学情報科学研究科博士前期課程の学生 9 名と社会人 3 名の合計 12 名である。すべての被験者は 4 年以上のプログラミング経験があった。

4.3 実験方法

実験は、予備実験と本実験から構成される。予備実験では、まず被験者はソフトウェアオーバホール作業を行う。作業終了後、ソフトウェアオーバホール作業により発見した理解容易性の問題の修正を行う。本実験では、予備実験で発見した理解容易性の問題を修正した後のプログラムと修正する前のプログラムのデバッグ作業を行う。本実験はデバッグを行う被験者によって EU1, EU2, EU3 に分類される。以下に予備実験と本実験 EU1, EU2, EU3 の手順を示す。

予備実験の手順

Step1: 4 人の被験者 (G_1, G_2, G_3, G_4) にソフトウェアオーバホールツールの説明を行う。

Step2: 被験者 G_i がプログラム M_i について、埋め込まれているフォルトが存在する関数のソフトウェアオーバホール作業を行う。

Step3: 被験者のソフトウェアオーバホール作業が終了した後、実験者はソフトウェアオーバホール作業における再統合の作業履歴を分析し、正しく統合するまでに要した判定回数が多い箇所を抽出する。抽出された箇所について、判定回数が多くなった理由を、被験者にコメントを求めることで理解容易性に問題がある箇所とその内容を特定する。特定した箇所に対して、関数の再構成やコメントの挿入を行い理解容易性の問題箇所を修正する。この作業によって修正した後のプログラムを M_i' とする。

本実験 EU1 の実験手順

被験者 (G_1, G_2, G_3, G_4) はソフトウェアオーバホール手法により発見した問題を修正した後のプログラム 1 つと修正する前のプログラム 1 つをデバッグする。デバッグにおいて、まず実験者は仕様書とテストデータを被験者に与える。次にテストデータをもとにエラーの状態を説明する。被験者がエラーの状態を理解した後、デバッグ作業を開始する。被験者はフォルトを特定しエラーの修復ができたことを実験者に確認されるまでデ

バッグを行う。実験者はデバッグ開始から終了までの時間を測定する。

本実験 EU2 の実験手順

被験者 (G_5, G_6, G_7, G_8) は、ソフトウェアオーバホールにより発見した問題を修正した後のプログラム 2 つと修正する前のプログラム 2 つをデバッグする。デバッグの方法は EU1 と同じである。

本実験 EU3 の実験手順

被験者 ($G_9, G_{10}, G_{11}, G_{12}$) はソフトウェアオーバホールによって発見した問題を修正した後のプログラム 2 つと修正する前のプログラム 2 つをデバッグする。デバッグの方法は EU1, EU2 と同じである。

表 1 は、予備実験における被験者へのプログラムの割当てを示している。表 2, 表 3, 表 4 は、本実験における各被験者のデバッグ作業の対象プログラムとその順序を示している。EU1 において、被験者は D_1, D_2 の順序でデバッグを行う。EU2, EU3 において、被験者は D_1, D_2, D_3, D_4 の順序でデバッグを行う。

デバッグ作業は、被験者の能力の差異に対処するため、ソフトウェアオーバホール手法により発見した問

表 1 予備実験のタスク割当て
Table 1 The task of preliminary experiment.

被験者	G1	G2	G3	G4
ソフトウェアオーバホールを実施するプログラム	M1	M2	M3	M4

表 2 EU1 のタスク割当て
Table 2 The task of EU1.

	G1	G2	G3	G4
D1	M3	M4	M1'	M2'
D2	M4'	M3'	M2	M1

表 3 EU2 のタスク割当て
Table 3 The task of EU2.

	G5	G6	G7	G8
D1	M3	M4	M1'	M2'
D2	M4'	M3'	M2	M1
D3	M1'	M2'	M3	M4
D4	M2	M1	M4'	M3'

表 4 EU3 のタスク割当て
Table 4 The task of EU3.

	G9	G10	G11	G12
D1	M4'	M4	M4	M4'
D2	M2	M2'	M2'	M2
D3	M1'	M1	M1	M1'
D4	M3	M3'	M3'	M3

題を修正した後のプログラムと、修正する前のプログラムについて同じ被験者が行う。ただし、1名の被験者が同じプログラムを2度デバッグすると2回目は1回目で見つけたフォールトを覚えている可能性があるため、1回目と2回目は異なるプログラムをデバッグする。たとえば EU1 において、各被験者がデバッグを行うプログラムは D1 と D2 で異なる(表 2)。なお本実験で使用したプログラムは、1つのプログラム M に4つのフォールトをそれぞれ埋め込んだ4種類のプログラム(M1~M4)である。4つのフォールトは独立しており、それぞれのフォールトを発見するためにはプログラムの異なる部分を読んで理解する必要があることから M1~M4 を異なるプログラムと見なした。

ソフトウェアが異なっても1回目と2回目では2回目の方が、デバッグ作業の効率が高くなることが考えられる。実験では、ソフトウェアオーバホール手法により発見した問題を修正した後のソフトウェアを1回目にデバッグする作業者と2回目にデバッグする作業者が同数になるように設定した。たとえば EU2 のデバッグ作業 D1 において、被験者 G5, G6 が問題を修正する前のプログラムをデバッグし、被験者 G7, G8 は問題を修正した後のプログラムをデバッグする(表 3)。以上の設定はすべての実験に対して行った。

EU3 では、フォールトの見つけにくさによるデバッグに要する平均時間に差が生じないように、デバッグするプログラムの順序を変更した。実験者の知見をもとに各プログラムにおけるフォールトの見つけやすさを、見つけやすい順に M4, M2, M1, M3 と評価した。フォールトの見つけやすいプログラム M4 からデバッグを始め徐々に難しいプログラムのデバッグを行い、プログラムに慣れることで、フォールトの見つけにくさの差異を小さくするようにした。また被験者4名を、同等の能力を有する2つのグループに分類するために、実験者の知見と経験をもとに被験者のプログラミング能力を高い順に G12, G11, G10, G9 と評価した。その後、評価を基準にグループ間の能力差が同等になるように被験者を G9 と G12, G10 と G11 のグループに分類した。

5. 実験結果と分析

5.1 実験結果

予備実験で行われたソフトウェアオーバホール作業では、すべての被験者が1時間以内で作業を終えた。図 5 は、ある被験者のソフトウェアオーバホールにおける統合の作業履歴の一部を示している。図 5 の例では、1つの行が1つの作業を示し、作業ごとに左から

```

.
.
.
Drag,983165062070,966,if (lisp_type(obj) != LISP_TYPE_CONS
Drop,983165063120,968,
Drag,983165159350,949,fputc(*p, out);
Drop,983165159950,950,
Drag,983165321820,982,fputs("#t", out);
Drop,983165325000,983,
Drag,983165662130,969,&& lisp_type(obj) != LISP_TYPE_PATTERN_CONS)
Drop,983165662790,966,
.
.
.

```

図 5 オーバホール作業の履歴

Fig. 5 The history of software overhaul.

表 5 EU1 におけるデバッグに要した時間

Table 5 The results of EU1.

	G1	G2	G3	G4	平均
修正前 プログラム	M3 365 分	M4 281 分	M2 149 分	M1 110 分	226 分
修正後 プログラム	M4' 171 分	M3' 132 分	M1' 112 分	M2' 50 分	116 分

表 6 EU2 におけるデバッグに要した時間

Table 6 The results of EU2.

	G5	G6	G7	G8	平均
修正前 プログラム	M2 16 分	M1 14 分	M2 32 分	M1 37 分	32 分
	M3 65 分	M4 30 分	M3 57 分	M4 6 分	
	M1' 23 分	M2' 13 分	M1' 7 分	M2' 6 分	
修正後 プログラム	M4' 7 分	M3' 55 分	M4' 3 分	M3' 26 分	18 分

順に、イベント、イベントが発生した時間、行番号、イベントを行った行となっている。イベントの「Drag」と「Drop」が対となってある行を並べ替えたことを示している。たとえば、図 5 に示す履歴の1行目と2行目の場合、ソースコードの966行目を Drag し、968行目に Drop することで2つの行を並べ替えたことになる。これらのデータをもとに分析した結果、発見された理解容易性の問題には、マジックナンバ、モジュールが大きすぎるといったものがあつた。それにともなって、ソースコード中へのコメントを追加する、規模の大きなモジュールを分割するといった修正を実験者が行った。

表 5, 表 6, 表 7 は、本実験において被験者がソフトウェアオーバホール手法により発見した問題を修正する前のプログラムと修正した後のプログラムのデバッグに要した時間を示している。修正後のプログラムのデバッグに要した平均時間が、修正前のプログラムの

表 7 EU3 におけるデバッグに要した時間
Table 7 The results of EU3.

	G9	G10	G11	G12	平均
修正前 プログラム	M2 80 分	M1 115 分	M1 105 分	M2 62 分	104 分
	M3 180 分	M4 115 分	M4 107 分	M3 70 分	
	M1' 70 分	M2' 25 分	M2' 32 分	M1' 10 分	
修正後 プログラム	M4' 60 分	M3' 115 分	M3' 105 分	M4' 24 分	55 分

デバッグに要した平均時間と比較して EU1 で 110 分、EU2 で 14 分、EU3 で 49 分短くなった。EU1、EU3 と比較して EU2 のデバッグに要した平均時間が短くなった理由としては、EU2 の被験者 (G5, G6, G7, G8) は全員 8 年以上のプログラミング経験があったこと、さらに被験者 G8 は LISP 処理系に関する知識を有していたことが考えられる。EU1 と EU3 の被験者のプログラミング経験年数は 4 年以上 9 年以下であった。

5.2 一元配置による分散分析

実験結果でデバッグに要した平均時間に差が生じた要因として、ソフトウェアオーバホールの要因以外に、被験者の要因およびフォルトの要因が考えられる。そこでデバッグに要した平均時間の差の要因を明らかにするために、以下の 3 つの仮説をたてて分散分析による検定を行った。

仮説 1: 被験者の要因によるデバッグに要した平均時間の差が小さい。

仮説 2: フォルトの要因によるデバッグに要した平均時間の差が小さい。

仮説 3: ソフトウェアオーバホールの要因によるデバッグに要した平均時間の差が小さい。

仮説 1 における被験者の要因とは被験者の能力を指す。被験者の能力がデバッグに要する平均時間に影響を与えると考えられる。仮説 2 におけるフォルトの要因とはフォルトの難しさ、いいかえればフォルトの見つけにくさを指す。フォルトの難しさがデバッグに要する平均時間に影響を与えると考えられる。仮説 3 におけるソフトウェアオーバホールの要因とはソフトウェアオーバホール手法により発見した問題を修正した場合、もしくは修正しなかった場合を指す。発見した問題を修正した場合と、修正していない場合によってデバッグに要する平均時間に影響を与えると考えられる。表 8 は 3 つの仮説について一元配置による分散分析を行った結果の p 値を示す¹⁸⁾。EU1 および EU3 については 3 つの仮説が有意水準 5% で採

表 8 3 つの仮説の分散分析 (一元配置)

Table 8 The results of one-way layout analysis of variance.

	EU1	EU2	EU3
仮説 1	0.298	0.924	0.275
仮説 2	0.364	0.006	0.194
仮説 3	0.115	0.399	0.058

表 9 仮説 1 のもとでの分散分析 (二元配置)

Table 9 The results of two-way layout analysis of variance based on hypothesis1.

	EU1	EU2	EU3
仮説 2	0.140	0.010	0.105
仮説 3	0.061	0.051	0.018

択された。EU1 および EU3 において、被験者の要因、フォルトの要因、ソフトウェアオーバホールの要因によるデバッグに要した平均時間に有意な差が認められなかった。EU2 については有意水準 5% で仮説 1、仮説 3 は採択され、仮説 2 は棄却された。EU2 においてフォルトの要因によるデバッグに要した平均時間に有意な差が認められた。

5.3 二元配置による分散分析

一元配置による分散分析において、EU1 と EU3 でデバッグに要した平均時間の差に影響する要因に有意な差が認められなかった理由として、デバッグに要した平均時間の差に影響する要因が 2 つ以上あったことが考えられる。そこで一元配置による分散分析で採択された仮説のもとで、他の仮説について二元配置による分散分析¹⁸⁾を行った。仮説 1 のもとで仮説 2 と仮説 3 について二元配置による分散分析を行った結果の p 値を表 9 に示す。仮説 1 のもとで EU1 は、仮説 2、仮説 3 とともに有意水準 5% で採択された。EU1 において、被験者の要因とソフトウェアオーバホールの要因によるデバッグに要した平均時間に有意な差が認められなかった。仮説 1 のもとで EU2 は、有意水準 5% で仮説 2 が棄却され、仮説 3 が採択された。EU2 において、フォルトの要因によるデバッグに要した平均時間に有意な差が認められた。また EU3 において有意水準 5% で仮説 2 が採択され、仮説 3 が棄却された。EU3 において、ソフトウェアオーバホールの要因によるデバッグに要した平均時間に有意な差が認められた。

次に、仮説 2 のもとで仮説 1 と仮説 3 について二元配置による分散分析を行った結果の p 値を表 10 に示す。ただし EU2 は、一元配置による分散分析において仮説 2 を棄却したので、二元配置による分散分析を行わない。仮説 2 のもとで EU1 および EU3 とともに、

表 10 仮説 2 のもとでの分散分析 (二元配置)
Table 10 The results of two-way layout analysis of variance based on hypothesis 2.

	EU1	EU2	EU3
仮説 1	0.064	-	0.227
仮説 3	0.031	-	0.034

有意水準 5% で仮説 1 が採択され、仮説 3 が棄却された。EU1 および EU3 において、ソフトウェアオーバホールの要因によるデバッグに要した平均時間に有意な差が認められた。分析の結果をまとめると以下のようになる。

- 仮説 1 のもとでは EU3 においてソフトウェアオーバホールの要因によるデバッグに要した平均時間に有意な差が認められた。
- 仮説 2 のもとでは EU1 と EU3 においてソフトウェアオーバホールの要因によるデバッグに要した平均時間に有意な差が認められた。

6. 考 察

6.1 ソースコードの修正と理解容易性

ソフトウェアオーバホール手法における統合作業を分析することにより明らかになった、理解容易性の問題を改善するために行われた修正が理解容易性を向上させたことを考察する。ソフトウェアオーバホール終了後にソースコードに対して行われた変更では、以下の 2 点があった。

- ソースコード中にコメントを入れる。
- 規模の大きな 1 つのモジュールを複数のモジュールに分割する。

ソースコードにコメントを追加することで、ソースコードの読み手を助けることが可能になる²⁾。コメントの書き方によってはその目的を果たさない場合もあるが、ソフトウェアオーバホールにより理解容易性に問題がある部分に対して、コメントを追加することでソースコードの読み手の理解を助長することが可能である。また、規模の大きなモジュールを分割することで、モジュールを構成する行数は少なくなる。人間が理解できるソースコードの長さには限界があり、長くなればなるほど複雑さを感じるといわれている³⁾。モジュールを分割することで、モジュールごとの行数が少なくなり、ソースコードの読み手の理解を助長することが可能である。以上より、実験で行われたソースコードの変更がソフトウェアの理解容易性の向上に貢献していると考えられる。

6.2 実験結果と保守性

ソフトウェアオーバホールによって発見された理解

容易性の問題を修正することにより保守性が向上したことを、実験結果をもとに考察する。保守性には以下の 3 つの特性が必要とされる⁸⁾。

- テスト容易性
- 理解容易性
- 更新性

ソフトウェアオーバホール後のプログラムに加えた修正によって、データ構造やアルゴリズム、プログラムの入出力は変わらなかった。また、デバッグに用いたテストケースも変わっていないため、テスト容易性は変化していない。

次に、プログラムに含まれているフォールトは 1 カ所だけであり、このことは実験前に被験者に伝えていた。テストデータも与えられていたために、被験者は 1 カ所を修正した後、他にも修正すべき箇所がないかを悩むことなく、テストを実行していた。テストの結果、正しい出力を得ればデバッグ作業は完了とし、その修正の影響で別の未知のフォールトが混入していないかを検証する作業は含めていない。したがって、更新性も変化していない。以上より、実験でデバッグに要する時間が短縮されたのは、プログラムの理解容易性が向上したためと考えられる。プログラムの理解容易性が向上したことにより、保守性が向上したといえる。

ソフトウェアの保守性を向上させることは、デバッグ時間の短縮だけではなく、ソフトウェア保守の様々な場面において有効である。ソフトウェア保守において、プログラムの理解に要する時間が最も多い。このためソフトウェアオーバホール作業によりソフトウェアの理解容易性を向上させることは、作業以降のソフトウェア保守におけるプログラム理解を容易にする。

7. 関連研究

ソフトウェアの理解容易性を評価する手法として、コードレビュー、ウォークスルー、Recall、Fill-in-the-blank、Maintenance task などがあげられる。コードレビューは、実際にプログラムをコーディングしたプログラマーを含めた数人によりプログラムのソースコードの読み合わせを行い、その過程で欠陥を見つけ出す手法である⁹⁾。コードレビューでチェックする項目としては以下のようなものがあげられる。

- プログラムは簡潔で分かりやすいか。
- コードはプロジェクトの規約に沿って書かれているか。
- モジュールは設計仕様書どおりに開発されているか。

これらの中では「プログラムは簡潔で分かりやすい

か」という項目が、理解容易性の評価に対応する部分となる。しかし、これを判断するためにはレビュー担当者は、どのようなコードがどの程度分かりにくいかを開発経験や訓練によりあらかじめ知っておく必要がある。

ウォークスルーは主に以下の2つの意味で使われている。

- 開発者が集まって、生産物をレビューし、討論すること¹⁷⁾。
- レビュー参加者がコンピュータの役割を演じ、テストケースを実行する¹⁴⁾。

前者は上述のコードレビューと同義であるので、ここでは後者の場合について述べる。ウォークスルーは、設定したテストケースそのものによって欠陥を見つけ出そうとするものではなく、プログラムの流れに沿って、開発者が論理を説明したり、参加者が質問をしたりする過程を通じて、要求仕様に関する開発者の誤解や論理の誤りなどを見つけることが目的である。生産物を最もよく理解している開発者本人が説明することにより、ウォークスルーの効率は良くなる。しかし、本研究では、開発に参加しなかった技術者が保守担当者になる将来に備えて、開発に参加した技術者がいる間に理解容易性の問題を修正しておく必要がある状況を想定している。そのような場合、保守担当者が生産物のみを読んで、どの程度理解できるかをウォークスルーの結果によって評価することはできない。

Recall⁴⁾ は決められた時間ソースコードについて学習をして、学習時間が過ぎると、作業者に学習したソースコードをできるだけ多く書くことを指示する手法である。本稿で述べた提案手法は、あらかじめソフトウェアに関して学習する必要がなく、仕様書とオーバホールツールを用いて、ソースコードを並べ替えるので、被験者の記憶力に依存する Recall とは異なる。

Fill-in-the-blank⁴⁾ は、特定の部分が空白になったソースコードを正しく修正するときの正確さと時間により、作業者の理解度を評価する手法である。提案手法では、オーバホールするソースコードの部分はランダムに並べられた状態で被験者に提示される。コードは、隠されることなくすべて提示されるので、特定の部分を空白にして修正を行う Fill-in-the-blank とは異なる。

Maintenance task⁴⁾ は、被験者にデバッグや機能追加などの保守作業を行わせて、その作業の完全性、正確性、所要時間などに基づいて作業者の理解度を評価する手法である。文献 6), 7) では、被験者の認識プロセスを調査するためにこの手法を用いている。

Maintenance task は、現実の保守作業に近い作業を被験者に行わせるため、現実の保守作業の難しさを予測する方法として有効と考えられる。ただし、デバッグや機能追加の難しさは、対象ソフトウェアの理解容易性だけでなく、検出しなければならないフォールトの種類や追加しなければならない機能の種類に大きく依存する。たとえば、C 言語のプログラミングにおいて初心者が混入しやすいフォールトとして、配列の添え字が 1 から始まるという誤解（正しくは 0 から始まる）に起因するものがある。この種のフォールトは、for 文の初期値設定において変数に 1 を代入しているなどのような単純なパターンマッチングによって検出可能である。このため、Maintenance task の実験において、この種のフォールトを被験者が短時間に検出できたとしても、対象ソフトウェアが理解しやすいとは限らない。機能の異なる複数のソフトウェア（あるいはソフトウェアモジュール）に、同程度の難しさのデバッグや機能追加の保守作業を設定することは非常に難しい。よって、Maintenance task は、同じ機能でドキュメントやコメントが異なるソフトウェアの理解容易性を比較する方法としては有効であるが、機能の異なるソフトウェアの理解容易性を比較する場合には向かない。

8. おわりに

本稿では、ソフトウェアオーバホール手法を適用することで理解容易性の問題が発見できることを確認するために行った実験について述べた。実験では、提案手法により発見した理解容易性の問題を修正した後のプログラムと修正する前のプログラムのデバッグに要する時間を比較した。実験の結果、理解容易性の問題を修正する前のプログラムのデバッグに要した時間と比較して、修正した後のプログラムのデバッグに要した時間が短くなった。すなわち、提案したソフトウェアオーバホール手法を適用することで、デバッグ作業の効率を低下させる理解容易性に問題がある箇所を発見できることを示した。

今後の課題として、異なるソフトウェア、特に実際のソフトウェア開発プロジェクトにおいて開発されたソフトウェアに対してオーバホール手法を適用し、その有効性を確認すること、オーバホールツールの改良などがあげられる。

参 考 文 献

- 1) Boehm, B.W., Brown, J.R., Kaspar, H., Lipow, M., MacLeod, G.J. and Merritt, M.J.:

- Characteristics of Software Quality*, North Holland Publishing Company (1978).
- 2) Brain, W.K. and Rob, P. (著), 福崎俊博 (訳): プログラミング作法, ASCII 出版 (2000).
 - 3) Dahl, O.J., Dijkstra, E.W. and Hoare, C.A.R. (著), 野下浩平, 川合 慧, 武市正人 (訳): 構造化プログラミング, サイエンス社 (1975).
 - 4) Dunsmore, A. and Roper, M.: A Comparative Evaluation of Program Comprehension Measures, *The Journal of Systems and Software*, Vol.52, Issue 3, pp.121-129 (2000).
 - 5) ISO/IEC 9126: Information Technology, Software Product Evaluation, Quality, Characteristics and Guidelines for their Use (2001).
 - 6) Letovsky, S.: Cognitive Processes in Program Comprehension, *Proc. Empirical Studies of Programmers, 1st Workshop*, pp.58-79 (1986).
 - 7) Littman, D.C., Pinto, J., Letovsky, S. and Soloway, E.: Mental Models and Software Maintenance, *Proc. Empirical Studies of Programmers, 1st Workshop*, pp.80-98 (1986).
 - 8) 宮本 勲: ソフトウェア・エンジニアリング: 現状と展望, TBS 出版 (1982).
 - 9) 大橋洋貴, 山本晋一郎, 阿草清滋: ハイパーテキストに基づいたソースプログラム・レビュー支援ツール, 電子情報通信学会技術報告, SS98-28, pp.15-22 (1998).
 - 10) Probst, M.: The lispreader library.
<http://www.complang.tuwien.ac.at/~schani/lispreader/>
 - 11) Porter, A.A., Siy, H., Toman, C.A. and Votta, L.G.: An experiment to assess the cost-benefits of code inspections in large scale software development, *IEEE Trans. Softw. Eng.*, Vol.23, No.6, pp.329-346 (1997).
 - 12) Regelson, E. and Anderson, A.: Debugging practices for complex legacy software systems, *Proc. International Conference on Software Maintenance*, pp.137-143 (1994).
 - 13) Shima, K., Takemura, Y. and Matsumoto, K.: An approach to experimental evaluation of software understandability, *Proc. International Symposium on Empirical Software Engineering*, pp.48-55 (2002).
 - 14) 玉井哲雄, 三嶋良武, 松田茂広: ソフトウェアのテスト技法, 共立出版株式会社 (1988).
 - 15) 寺井淳裕, 内田真司, 島 和之, 武村泰宏, 松本健一, 井上克郎, 鳥居宏次: 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, 電子情報通信学会技術報告, SS2000-52, pp.81-88 (2001).
 - 16) 内田真司, 島 和之, 阪井 誠, 松本健一: ソフトウェアオーバホール手法の評価実験, ソフトウェアシンポジウム 2002 論文集, pp.116-121 (2002).
 - 17) Yourdon, E. (著), 國友義久, 千田正彦 (訳): ソフトウェアの構造化ウォークスルー, 近代科学社 (1981).
 - 18) 涌井良幸, 涌井貞美: Excel で学ぶ統計解析, ナツメ社 (2006).
- (平成 19 年 5 月 15 日受付)
(平成 19 年 12 月 4 日採録)



内田 真司 (正会員)

平成 9 年奈良工業高等専門学校専攻科電子情報工学専攻修了。平成 11 年奈良先端科学技術大学院大学情報科学研究科博士前期課程修了。平成 17 年同大学博士後期課程修了。平成 12 年近畿大学工業高等専門学校電気情報工学科助手。平成 16 年同校講師。平成 18 年同校助教授。平成 19 年より奈良工業高等専門学校情報工学科助教。博士 (工学)。エンピリカルソフトウェア工学の研究に従事。電子情報通信学会, 日本産業技術教育学会各会員。



島 和之 (正会員)

平成 5 年大阪大学大学院博士前期課程修了。平成 6 年同大学院博士課程中退。同年奈良先端科学技術大学院大学助手。平成 16 年より広島市立大学情報科学部助教授。平成 19 年より同大学准教授。博士 (工学)。ソフトウェア工学の研究に従事。電子情報通信学会, IEEE Computer Society 各会員。



武村 泰宏 (正会員)

平成 14 年奈良先端科学技術大学院大学情報科学研究科博士後期課程修了。工学 (博士)。大阪芸術大学短期大学部助教授を経て大阪芸術大学教授。ソフトウェア工学教育の研究に従事。電子情報通信学会, IEEE Education Society 各会員。



松本 健一（正会員）

昭和 60 年大阪大学基礎工学部情報工学科卒業．平成元年同大学大学院博士課程中退．同年同大学基礎工学部情報工学科助手．平成 5 年奈良先端科学技術大学院大学助教授．平成 13 年同大学教授．工学博士．エンピリカルソフトウェア工学，特に，プロジェクトデータ収集/利用支援の研究に従事．電子情報通信学会，日本ソフトウェア科学会，ACM 各会員．IEEE Senior Member．
