グループウェア 32－7
（1999．5．19）

二人によるソフトウェアデバッグにおける役割分担と情報交換

森崎 修司　伊藤 充男*　門田 暁人　松本 健一

奈良先端科学技術大学院大学 情報科学研究科

〒 630-0101 奈良県 生駒市 高山町 8916-5 Tel: 0743-72-5312

{shuuji-m, atsuo-i, akito-m, matumoto}@is.aist-nara.ac.jp

アブストラクト　本研究の目的は，二人によるソフトウェアデバッグの効率のよい方法を確立することである．本稿では，二人の作業者間の役割分担，および，情報交換の方法を分類し，実験を通してそれらの長所と短所を考察した．実験の結果，(1) 一人がデバッグに専念しもう一人がデバッグに役立つ情報を収集すること，(2) プログラムのソースコードを共有し，ソースコード中にコメントを書く事で情報を伝達することがデバッグ時間の短縮に有効であることが分かった．

キーワード　デバッグ時間の短縮，二人によるデバッギング，作業分担，情報交換

An Analysis of Task- and Information-Sharing in Collaborative Debugging by Two Person

Shuuji Morisaki　Atsuo Ito　Akito Monden　Ken-ichi Matsumoto

Graduate School of Information Science

Nara Institute of Science and Technology

8916-5 Takayama, Ikoma, Nara 630-0101　Tel: +81-743-72-5312

{shuuji-m, atsuo-i, akito-m, matumoto}@is.aist-nara.ac.jp

Abstract Software debugging is often performed by more than one programmer working together. The goal of this research is to identify a model of how two person collaboratively perform a debugging task on a single program. Based on empirical studies we have conducted, we have identified patterns in task- and information-sharing, and applied them to analyze the observed collaborative debugging processes. The following two collabration patterns were found to be effective in debugging (1) one concentrated on a debugging task while the other collected potentially useful information about the program, and (2) the two shared one program source file in their editors instead of using duplicates and wrote comments into the file.

Keywords collaborative debugging, task sharing, shared knowledge

*現 旭化成マイクロシステム 設計開発センター〒 243-0021 神奈川県厚木市岡田 3050

1　はじめに

稼働中のソフトウェアや出荷直前のソフトウェアに故障が発生した場合，たとえ多くの人員を投入してでも，故障の原因となるソフトウェア中のバグ（fault：ソフトウェア誤り）を速やかに発見し，除去することが望ましい．例えば，銀行のオンラインシステムが故障した場合，社会に与える影響を最小限に食い止める必要があり，バグの除去は一刻を争う．また，納期が間近に迫っているソフトウェアに新たな故障が発生した場合にも，納期を守る上では一刻も早くバグを取り除く必要がある．

しかし，たとえ多くの人員を投入できるとしても，複数人でデバッグすることは必ずしも容易でない．その理由の一つは，作業の詳細があらかじめ決まっていないため，作業分担が容易でないことである．複数人でプログラミングを行う場合は，あらかじめ作られた設計書に基づいて作業分担できるが，デバッグには設計書に当たるものが存在しない．もう一つの理由は，作業者間の情報交換に時間を要することである．プログラミング作業においてはソースコードが成果物となるが，デバッグ作業の過程では，バグに関する手がかりとなる情報が成果物となる．複数人のデバッグではそうした情報の交換が必須となるが，情報交換に時間がかかりすぎると，一人によるデバッグよりもかえって効率が悪くなる恐れがある．

本研究の目的は，多人数によるデバッグの第一歩として，二人によるデバッグの効率的な方法を明らかにすることである．特に，作業の切り分け（役割分担），および，作業者間のコミュニケーション（情報交換）について，効率のよい方法を実験を通して明らかにする．

以降，2章では関連する研究について述べる．3章では二人によるデバッグの作業分担の形態，および，情報交換の形態を整理する．それに基づいて行った実験と考察を4章で述べ，さらに改良した実験について5章で述べる．6章はまとめである．

2　関連研究

複数人が集まってソフトウェアプロダクト中のバグを発見する方法として，従来より，レビューが用いられている．レビューにおける役割分担を効率的に行う方法としては，Scenario-Based Reading[1], Perspective-Based Reading[4] などが提案されている．

しかし，それらのレビュー方法は，本研究が想定している状況（デバッグ）には必ずしも適用できない．レビューではプロダクト中の矛盾を網羅的に探すことに重点が置かれるが，デバッグではバグのありそうな箇所を迅速に絞り込むことに焦点があてられる．デバッグにおいてプロダクトを網羅的に探すと，むしろデバッグの効率を悪化させることになる．

以降では，簡単のため，故障の原因となるバグ（fault）は，ソフトウェア中に1個だけ含まれるものと仮定する．つまり，複数のバグが複合して1個の故障が発生しているような状況は想定しない．なお，ここでいう故障とは，仕様書に記述されている動作と異なるようなソフトウェアの動作のことである．

3　役割分担と情報交換

3.1　役割分担の形態

一般に二人で何らかの作業を分担して行えば，全体の作業時間を短縮できることが多い．デバッグにおいても，作業を適切に分担することでバグを発見するまでの時間を短縮できる可能性がある．ここでは，デバッグにおいてはどのような役割分担の形態が考えられるかを整理する．

(1) 作業分割型の役割分担

バグを探す範囲を二つに分割し，各人がそれぞれの担当箇所のバグを探す方法である．バグを探す範囲をうまく二つに分割して作業分担できれば，効率が良くなる可能性がある．

ただし，バグを探す範囲の分割には十分な注意が必要である．故障の症状やデバッガの実行結果を無視し，やみくもにプログラムを二つに分割して作業者を割り当てても，デバッグの効率が上がるとは限らない．バグがなさそうな箇所に作業者を割り当ててしまう恐れがある．

(2) 主従型の役割分担

バグを探すというメインの作業は一人（リーダー）がやり，それに付随する作業をもう一人が補助する．バグを発見するのはリーダーである．補助する人はリーダーが必要とする情報を集めることになる．

(3) リスク分散型の役割分担

役割分担を行わない方法である．各作業者に何らかの役割を与えてしまうと，各人のデバッグスタイルが阻害され，能力を発揮できない可能性がある．本方法では，役割分担を行わず，各人が自分のデバッグスタイルに基づいてデバッグを行う．この方法では，二人が時々会話を行うことで，リスク分散の効果が期待される．

一般に，デバッグ作業者は，あれこれ仮定を置きながら不確かな（不十分な）知識に基づいてバグを

探しているので，誤った行動を取ってしまうことがよくある．例えば，実際にはバグがないところをバグがありそうだと判断し，バグのないところばかりを探してしまうことがある．しかし，二人で時々会話することにより，お互い自分の間違いに気づきやすくなると期待される．

3.2 情報交換の形態

情報交換の形態は，デバッグ効率に大きく影響する [3]．ここでは，三つの情報交換の形態について述べる．

(1) 同期型の情報交換

同期型の情報交換を図 1（上）に示す．同期型の例としては，口頭による情報交換がある．同期型の情報交換では，交換したい知識を実時間かつ短時間でパートナーに伝えたり，パートナーから受け取ることができる．

(2) 非同期型の情報交換

非同期型の情報交換を図 1（下）に示す.「バッファ」を持つのが非同期型の特徴である．一方の作業者が相手に情報を伝達しようとすると，その情報は一旦共有バッファに書き込まれる．他方の作業者は，情報を受け取りたいときに共有バッファを参照する．そのため，情報の伝達と受け取りが異なる時間に行われる．

非同期型の例としては，計算機上でのチャットツールを利用する方法がある．この場合，チャットウィンドウが共有バッファに該当する．非同期型の情報交換の利点は，情報の投げかけによって相手の作業を中断しないことである．非同期型では，相手がバッファに書き込んでいる間は自分の作業をすることができ，情報を受け取りたいときにだけバッファを読めばよい．

(3) スイッチ付き同期型の情報交換

スイッチ付き同期型の情報交換を図 1（中）に示す．スイッチが ON になっている間だけ情報が交換できる．

例えば，1 時間おきに 10 分間だけ情報交換を行うことが考えられる．スイッチのない同期型と比べた利点の一つは，情報交換により作業が中断する時間帯が限られているため，作業の集中を妨げないことである．

4 実験

本章では，前章で述べた情報交換と役割分担の形態を用いて実際に二人によるデバッグを行った実験について述べる．実験の目的は，それぞれの情報交換および役割分担の形態の優れている点と問題点を明らかにし，二人が効率良くデバッグする方法についての知見を得ることである．

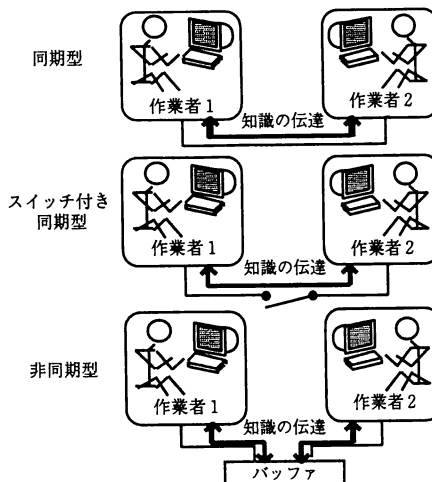

図 1: 三つの情報交換の型

4.1 実験の環境と手順

4.1.1 実験の環境

実験では，それぞれの被験者に UNIX マシンを割り当て，計算機上でデバッグしてもらった．2 台のビデオカメラを用いて，被験者の計算機のディスプレイと被験者らの発話を記録した．被験者は，5 ～ 6 年のデバッグ経験をもつ二人の大学院生を選んだ．二人のデバッグ技術に大きな差はない．

デバッグに使うツールとして，コンパイラに gcc-2.8，デバッガに gdb-4.1，エディタに emacs20.3 を選んだ．被験者は，これらのツールを普段から使用している．

デバッグに用いたプログラムは 200 ～ 400 行の規模であり，C 言語で記述されている．被験者はプログラムの作成に関与していない．各実験で用いたプログラムの仕様は同一であり，酒屋の在庫管理を行うプログラムである．ただし，プログラムごとにデータ構造などの設計が大きく異なっており，仕様が同一であることによる学習効果がデバッグ時間に与える影響は小さいと考えられる．

各プログラムにはそれぞれ 1 つのバグが混入しており，特定の入力に対してのみ，誤った実行結果を出力するものである．各プログラムのバグは，実験

者が故意に混入したものである．バグの種類は，条件分岐の条件式の誤り，ポインタの誤り，配列の添字の誤りである．

4.1.2 実験の手順

各実験では，被験者に以下の順に作業してもらった．

(1) 仕様書の理解: 被験者はプログラムの仕様書を読み，理解する．制限時間は設けない．

(2) 故障の症状の理解: 実験者はプログラムにテストデータを与えて実行し，故障が発生していることを被験者に示す．テストデータは被験者に与えられ，デバッグ中にいつでも参照できる．

(3) デバッグの開始: 二人の被験者がデバッグを同時に開始し，デバッグ時間の計測を開始する．デバッグ時間には，被験者が仕様書を読む時間，及び，バグの症状を見る時間を含めない．

(4) デバッグの終了判定: 被験者のうち 1 人がバグを取り除くことができたと判断したら，もう 1 人の被験者と実験者に知らせる．バグが除去できたと実験者が判断したら実験を終了する．

4.2 実験条件

実験 A(主従・同期)　役割分担の方法は主従型とした．一人がリーダーとなりデバッグ作業を進め，必要に応じてもう一人に作業を依頼する．各人の作業内容は，リーダーとなる作業者によってデバッグ中に決定される．

情報交換の方法は同期型とした．同期型にした理由は，リーダーが作業中に素早く指示を出せるようにするためである．二人は隣あった計算機でデバッグしてもらった．

実験 B(作業分割・非同期)　役割分担の方法は，作業分割型とした．一人は，プログラムの先頭から制御フローをたどることでバグを探し，もう一人は，故障発生箇所からデータフローを遡ることでバグを探す．この役割分担の方法は，Korel ら [2] のアルゴリズミック・デバッギングの方法を参考にした．

それぞれの作業がある程度独立しているため，情報交換は相手の作業を中断しない非同期型とした．共有バッファとしてチャットツールを使用し，情報交換はすべてチャットツールのウィンドウを通して行う．

実験 C(リスク分散・スイッチ付き同期)　役割分担の方法は，リスク分散型とした．

ある程度作業を進めてからでないと，リスク分散の効果があらわれないことが予想されるため，情報交換はスイッチつき同期型とした．20 分おきに会話ができる時間を設定した．デバッグ開始からはじめの 20 分はデバッグに集中してもらい，その後自由に会話してもらう．これを繰り返した．

表 1: デバッグ時間と情報交換に費やした時間

	時間	情報交換の時間	難易度	行数
A	85 分	50%	やや難	423 行
B	57 分	42%	ふつう	285 行
C	68 分	14%	やや難	193 行

4.3 実験結果と考察

三つの実験すべてにおいて，バグが正しく取り除かれた．デバッグに要した時間，情報交換に時間の割合，および，バグの難易度に関する被験者の主観的な評価を表 1に示す．

三つの実験全体にいえることは，二人によるデバッグの効果があまり見られなかったことである．個別の分析を以下に述べる．

実験 A(主従・同期)　情報交換に 50 %もの時間を費やした．リーダーは相手に意図を伝えたり，指示を出すために会話の時間が長くなった．また，同期型の情報交換では，容易に相手に話しかけることができるため，ついつい会話時間が長くなってしまいがちであった．これは同期型の情報交換の弱点であると言える．

指示した作業内容にも問題があった．リーダーが指示した内容は，緊急性を要するものが多かった．指示を受けた人がその作業を終えない限り，リーダーは次の作業に進めない場合があった．そのような場合，二人でデバッグしている意味がなく，リーダーは指示を出さずに自分で作業を進めるべきであった．デバッグ中に指示を出すことは容易ではないと言える．

実験 B(作業分割・非同期)　交換された情報量は小さかったが，情報交換に 42 %もの時間を費やした．その理由は，チャットウィンドウに文字を入力するのに時間がかかったためである．また，チャットでは相手に意図を伝えにくいことがあった．たとえば，プログラム中のある箇所を指定しながら説明したい場合，チャットでは困難であった．

役割分担（作業分割）には無理があったと言える．制御フローを追跡してバグを探す役割は順調であったが，故障発生箇所からデータフローを遡る役割は遂行が困難であった．結果として，二人によるデバッグの効果は見られなかった．

実験 C(リスク分散・スイッチ付き同期)

実験 A，B と比較すると会話時間は 14 %と小さいが，会話できる時間帯を制限したことはいくつかの問題を生んだ.

定期的な会話では，バグに関する詳細で具体的な会話は少なく，抽象度の高い会話が多く行われた．相手に何かを尋ねたり伝えたいと思った瞬間に会話できないと，その内容の詳細を忘れてしまったり，すでに必要なくなったりするためである．そのため，どちらの作業者も相手から有益な情報を得られなかった．これはスイッチ付き同期型の情報交換の弱点と言える.

また，二人のデバッグは順調に進んだため，リスク分散の効果は見られなかった.

5 改良した実験

前章の実験をふまえて，二つの実験を行った．二つの実験 1, 2 はそれぞれ実験 A, B, C の問題点を改良したものである.

5.1 実験条件

実験 1 役割分担をリスク分散型とし，情報交換は同期型を採用した．また，後で述べるように，特殊な機能を持ったエディタを使用した.

実験 C のスイッチ付き同期型の情報交換では，情報交換の時間は小さく押えられるが，相手に尋ねたいことがあってもすぐに尋ねられないという問題があった．一方，同期型の情報交換は，実験 A の結果から分かるように，情報交換の時間が非常に大きくなる.

同期型の情報交換において情報交換の時間を小さくするためには,「自分が尋ねたいことのうち，相手が知ってそうなことだけを尋ねる」ことが有効であると思われる．そこで，実験 1 では，相手が知らない内容に関する質問を減らすための制限を設けることにした.

設けた制限は相手の知らないことを質問しないことである．しかし，相手が何を知っていて何を知らないかを直接知ることはできないので，全体のソースコードのうち，パートナーが見ていない部分は，パートナーが知らない部分であると仮定し，パートナーが見ていない部分に関する質問をしないようお願いした.

相手がどの程度ソースコードを見ているかを表示する機能を持ったエディタを作成した（図 2）．エディタには，左右にソースファイルを表示する部分がある．左側で自分のファイルを編集することができる．右側は相手が編集しているファイルを表示している．また，それぞれのソースファイルを表示する部分の左端には，ソースファイルの各行にカーソルとスクロールバーの停留時間から求められる参照時間がグラフと数字で表示される．これにより，相手が知らなさそうな質問を制限できると期待される.

実験 2 役割分担を主従型とし，情報交換は非同期型を採用した.

実験 A では，実験中にリーダーが指示を出していたが，指示を伝えるために時間がかかっていた．そこで今回は，リーダーを補助する人が行うべき作業内容と，リーダーに伝えるべき情報をあらかじめ決定しておいた．具体的には，プログラム中のグローバル変数の意味を理解し，それをリーダに伝えることとした.

実験 B から分かるように，非同期型では，バッファへの書き込みに時間がかかった．そこで，バッファへの書き込みは，リーダーを補助する人のみが行い，リーダーはそれを読むだけにした.

また，実験 B では，チャットでは意図を伝えにくいことが分かったので，次のような方法を取った．ソースファイルを共有バッファとし，補助する人はリーダーに伝える情報をソースファイルにコメントとして直接書き込む．リーダーは，書き込まれたコメントを読むことで，情報を受け取る．emacs の機能を利用し，同一のファイルを異なる計算機上で編集することにしたため，補助の人の行った変更はリアルタイムにリーダーの見ているファイルに反映される.

5.2 実験結果と考察

実験 1,2 ともにバグが正しく取り除かれた．デバッグに要した時間と，バグの難易度についての被験者の主観的な評価を表 2に示す．それぞれの実験について，次のことがわかった.

実験 1 二人によるデバッグの効果はあまり見られなかった.

実験開始からしばらくの間，ほとんど会話がなかった．実験後の被験者へのインタビューから，相手が見ていないソースコードに関する質問をしなかったためであることがわかった．作成したエディタの効果があったと言える.

実験終了間際では，多くの会話が見られた，先行している被験者がもう片方の被験者に説明するために時間をとられていた．このとき，先行している方の被験者は有益な情報を得ることができなかった．二人のデバッグ作業の進捗に差がある場合には，リス

表 2: 実験 1, 2 の作業時間と情報交換に要した時間

実験	時間	情報交換の時間	難易度	行数
1	29 分	27%	ふつう	255 行
2	31 分	3%リーダー 60%補助の人	やや難	299 行

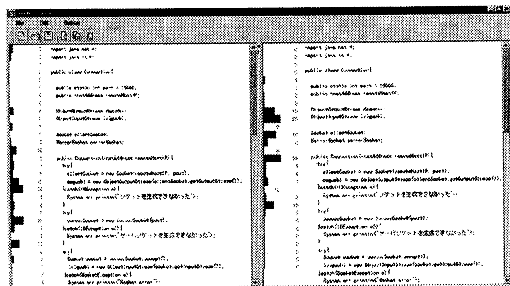

図 2: 実験 2 に用いたエディタ

ク分散型の役割分担は有効に働かないと言える.

実験 2 二人によるデバッグの効果が大いに見られた.

リーダーを補助する人はコメントの書き込みに多くの時間をさいていた(60 %)が, リーダーが情報の受け取りに要した時間は 3 %と小さく, 非同期型の情報交換におけるバッファの効果が見られた. リーダーは, グローバル変数についての知識が必要になったときのみ, 補助の人が書き込んだコメント(ソースコード上のグローバル変数のそばにある)を読めばよく, 情報の受け取りにほとんど時間がかからなかった.

また, ソースコードそのものをバッファとしたことの有効性も確認できた. チャットにおいては, プログラム中のある箇所を指定しながら説明することが困難であったが, ソースコード上に直接コメントを書くことで, 自分が伝えたい情報がソースコード上のどこにあるかを説明する必要がなくなった.

6 おわりに

二人によるソフトウェアデバッグにおける役割分担, 及び, 情報交換の形態を整理し, 実験を通して効率のよいデバッグ方法を模索した. 4 章と 5 章の実験結果からは, 以下のことが分かった.

役割分担について

(1) 主従型では, リーダーが補助の人にデバッグ中に指示を与えることは容易ではない. 補助の人の役割をあらかじめ決めておくことが有効である.

(2) 作業分割型は, 現状では適切な作業分割方法がない.

(3) リスク分散型は, 簡単なプログラムでは効果がない. また, 二人の間のデバッグ能力に差があり, デバッグの進捗に差がでる状況では効果がない.

情報交換について

(1) 同期型は, 情報交換の量が多くなりがちであるが, 相手の作業状況がリアルタイムに分かるような仕組みを設けておくことで, 無駄な情報交換を減らすことができる.

(2) 非同期型は, 双方向の情報交換には向かない. 補助の人からリーダーへの一方向の情報伝達では有効である. また, ソースコードをバッファとすることが有効であった.

(3) スイッチ付き同期型は, 有効性が見られなかった.

参考文献

[1] A. A. Porter, L. G. Votta, and V. R. Basili, “Comparing detection methods for software requirements inspection: A replicated experiment, ” IEEE Trans. on Software Engineering, 21, 6, pp.563-575, 1995.

[2] B. Korel, “PELAS – program error-locating assistant system, ” IEEE Trans. on Software Engineering, 14, 9, pp.1253- 1260, 1988.

[3] 伊藤 充男, 森崎 修司, 門田 暁人, 松本 健一, 鳥居 宏次, “デバッグ時間の短縮を目的とする二人によるデバッギングの実験的考察”, 信学技報, AI98-82, pp.45-52, 1999.

[4] V. R. Basili, S. Green, O. Laitenberger, F. Lanubile, F. Shull, S. Sorumgard, and M. V. Zelkowits, “The empirical investigation of perspective-based reading, ” Empirical Software Engineering, 1. pp. 165-188, 1996.