ソフトウェア工学 129−11
（2000．11．17）

# プログラムの静的解析と動的解析の相互補完による プログラム理解・開発支援プロセス

小野 展弘　本間 昭次　深谷 哲司

株式会社 東芝 研究開発センター システム技術ラボラトリー
〒212-8582 神奈川県川崎市幸区小向東芝町1番地
Phone: 044-549-2402　Fax: 044-549-2223
E-mail: { nobuhiro.ono, akitsugu.homma, tetsuji.fukaya }@toshiba.co.jp

## 概 要

ソフトウェア開発においては、新たに開発を行うことよりも既存プログラムを再利用して開発を行うことが多くなっている。既存プログラムの再利用は、開発コストの削減および開発期間の短縮に繋がると考えられているからである。しかしながら、現状では既存プログラムを理解する作業が大きなボトルネックとなっている。本稿では、プログラム理解をソフトウェア開発支援のポイントとして位置付けて、プログラム理解コストをプログラムの静的構造と動的構造の両視点から省力化する手法を述べる。それを基にプログラムの改善過程と、プログラムへの変更の確認過程を特徴として持つプログラム開発プロセスを提案する。

Proposal of a Program Understanding / Development Process
with Mutual Complement of the Static-Analysis and Dynamic-Analysis

Nobuhiro Ono, Akitsugu Homma, Tetsuji Fukaya

Corporate Research & Development Center, Toshiba Corporation
1 Toshiba-cho, Komukai, Saiwai-ku, Kawasaki-shi, Kanagawa 212-8582, Japan
Phone: +81-44-549-2402　Fax: +81-44-549-2223
E-mail: {nobuhiro.ono, akitsugu.homma, tetsuji.fukaya }@toshiba.co.jp

## Abstract

In this paper, we locate the understanding of existing programs as a significant point in supporting software development, and propose a software development process with two features. One is to reduce the cost for understanding programs by analyzing both static and dynamic structure. Moreover, to enhance the understandability of programs considering improvement processes.

## 1 はじめに

近年ソフトウェア開発においては、新たに開発を行うことよりも、既存プログラムを再利用して開発を行うことが多くなってきており、その開発コストも増大の一途を辿っている。このことは、既存プログラムの再利用性を向上させることができれば、開発コストの削減および開発期間の短縮が可能となると言い換えることができる。

しかしながら、既存プログラムの利用には既存プログラムを理解する作業がついてまわり、それがボトルネックとなって現状では開発コストの実に半分以上が理解作業に費やされている[1]。このような状況に加え、既存プログラム理解の手助けとなるべきドキュメント類が十分に存在するとは言えない。

そこで、既存プログラムを様々な視点から解析して理解の手助けとなる情報を抽出・提供するプログラム理解支援、さらには理解容易化、変更・拡張容易化のためのプログラム構造の改善支援による開発コストの削減が特に求められている。

このようなニーズに対して、効率よく既存プログラムを活用し、迅速なソフトウェア開発を行うため、「理解」、「変更」、「確認」の3つのメインステージ[2]と、「改善」の1つのサブステージを1サイクルの構成要素としたスパイラルなプログラム開発プロセスを提案する。本提案プロセスの特徴は、「確認」ステージによるサイクル移行の為の評価過程の設置による曖昧なプロセス進行の排除と、「改善」ステージの適時実施によるプロセスのスムーズな進行の保持にポイントを置いていることである。また、このプログラム開発プロセスに基づいた開発支援ツールを開発することで、本提案プロセスの有用性について検討する。

## 2 プログラム開発プロセスの提案

既存ソフトウェアを理解し、活用するという開発作業は、開発コストの大半が投入されているにもかかわらず、その作業プロセスは曖昧なままになっている。本稿ではこうした曖昧な作業プロセスに決別し、明確なプログラム開発プロセスを提案する。そしてそのプロセスに応じた的確な支援を可能にすることによって、開発工数を大幅に削減することを目的としている。

### 2.1 提案プロセス

ソフトウェア開発を計画的に進める場合には、実現する段階毎にフェーズとして区切って進められる。フェーズとは、1つ以上のプログラムの変更・機能拡張等といった実現しなければならないマイルストーンから構成されると定義する。そしてマイルストーンの解決ステップはサイクリックな形態（サイクル）によって進められていく。

1つのサイクルは開発者が行うべきいくつかの作業のステージ、およびリンクする他のサイクルへ移行するための評価基準がある。ここで、ステージとは以下に定義する3つのメインステージと1つのサブステージに分けられる。

メインステージ

- 理解ステージ
  プログラムおよびなされるべき変更の理解
- 変更ステージ
  システムの変更・拡張を実現するためのプログラムの変更・拡張、およびシステム全体における変更・拡張の影響理解
- 確認ステージ
  変更後のプログラムの正当性の確認

サブステージ

- 改善ステージ
  プログラム理解および変更容易性を向上させるためのプログラム構造の改善

メインステージはプログラムの外部的な振る舞いの変化を伴う変更の基本的な活動段階である。サブステージはプログラムの外部的な振る舞いを保持したまま、内部の構造を改善するというプログラム変更の過程がサイクルに含まれる場合に登場する活動段階である。

フェーズの進行は上記のステージによって構成されるサイクルを順次移行することで行われ、サイクル間の移行は、各サイクルに設定された評価基準をクリアすることで行われる。評価基準に達しない場合には、図1に示すように「理解」→「変更」→「確認」→「理解」→・・・ のように再度サイクルが繰り返される。このように評価基準を定めた「確認」ステージが明確に存在することが、本提案プロセスの1つの特徴である。

なお、サブステージである「改善」は、「理解」ステージの作業での、なされるべき変更の理解が困難であると判断された場合に実施されるもので、本提案プロセスの特徴のもう1つである。改善ステージによるプログラム構造の変更が完了したならば、その変更に対する、確認ステージへの移行か、もしくはそのまま変更ステージでのシステムの変更・拡張の実施へと移行することになる。

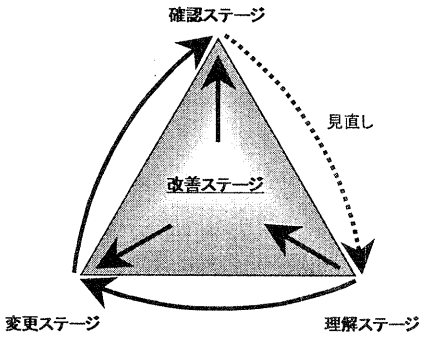



図1: 各ステージ間の移行関係

以上のような、3つのメインステージを1サイクルとする開発プロセスは図 2に示すようにスパイラルなプロセスで表現できる。このようなスパイラルの流れに沿って開発が進むことより、計画的で無駄のないソフトウェア開発が行える。

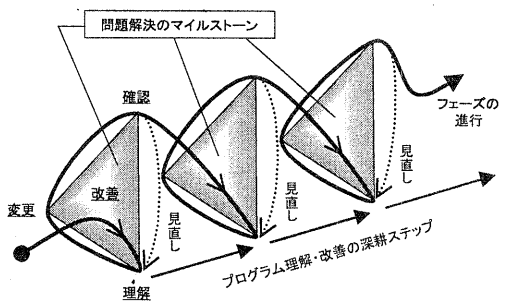



図2: 開発プロセスモデル

## 2.2 サイクル移行の為の評価基準

本稿で提案する開発プロセスにおいて開発者は次のサイクルへ進めるために、現サイクルの評価基準をクリアする必要がある。そのためには、そのサイクルで設定された対象プログラムに対する変更および機能拡張といった実装作業を完了し、その実装内容の動作確認と安全性を検証しなければならない。

ここで、それぞれのサイクルに設定されるべき実装作業とは、開発者が対象プログラムに対してコード編集作業を行い、プログラムをコンパイルおよびリンクする毎の単位を最小とするもので、変数名の変更という小さなものから、ある機能モジュールの追加といった明らかにプログラムの動作に変更が加わるものまでも含まれる。最小単位のサイクルを複数まとめて、大きな1つのサイクルとすることも可能ではあるが、プログラム理解の深耕や、安全性の向上の面からも、ビッグ・サイクルよりもスモール・サイクルを連続して設定し、確実にクリアすべきである。

次のサイクルへ移行するための評価基準は、以下の2つの確認作業をクリアする事と定義する。

- 対象プログラムに対して設定した実装によって生じるべき振る舞いの確認
- 実装作業に伴うシンボル変更の追加／削除／移動／名称変更等による副作用が存在しないことの確認

前者については、実際にプログラムを実行させて、その動的な振る舞いを確認するというような曖昧なものでは意味がなく、実装作業の前後における制御の流れとデータの流れの変化を比較して確認できるようにしなければならない。

また、後者の副作用の存在については、実際にプログラムを実行させて確認するのでは確実な検出が行えるとは限らず、静的に解析した結果によるシンボル間の依存関係を基に確認する必要がある。なお、本稿で扱うシンボルとは、プログラムを構成する関数や変数等を指す。

# 3 提案プロセス内ステージ

この章では、本稿で提案するプログラム開発プロセスの各ステージにおいて、開発者の行うべき作業と、その支援に必要となる技術について考察する。

## 3.1 理解ステージ

プログラムに限らず、人間があるものを理解しようとするならば、理解対象の性質を表す情報を試行錯誤しながら反復して参照する作業が必然的に生じる。本稿では、この単純化した人間の理解プロセスを基に、理解ステージについて検討を行う。

### 3.1.1 プログラム理解においてすべきこと

開発者は、既存ソフトウェア資産のプログラムを理解するために次に示すようなプログラムの性質を表す情報を、正確かつ詳細に既存プログラムから抽出し、把握しなければならない。

- 既存プログラムの構造情報
- 既存プログラムの振る舞いに関する情報

一般的にこれらの情報を得るためには、表 1に定義する静的解析と動的解析の2通りの手法がある。

| | |
|---|---|
| 静的解析 | プログラムを実行させずにシンボル間の静的な依存関係を解析する |
| 動的解析 | プログラムを実行させ、その結果からプログラムの振る舞いを解析する |

表1: 静的解析と動的解析の定義

また、静的解析、動的解析によって得られる情報は、プログラム理解に有効な試行錯誤の材料となるべきであり、反復して参照可能である必要が無ければならない。よって解析によって得られた情報を記録し、開発者が常に再利用可能とすべきである。

### 3.1.2 静的解析による抽出情報

静的解析によって得られるシンボル間の依存関係情報は、プログラムスライシング技術[3]を用いることによって、制御依存関係やデータ依存関係を考慮した解析が可能である。これは開発者の注目点に関連する依存関係情報に絞った出力を行うなどして、各シンボルの役割や、変更が行われた場合の影響などの把握のように、プログラムの構造理解を支援することに利用できる。

しかしながら静的解析のみからシンボル間の依存関係を把握するには、効率と正確性が十分であるとは言えず、対象プログラムの構造情報を把握するためにも以下の改善が必要である。

- 膨大な数のシンボルに対しての調査や、2次的、3次的な依存関係を辿る際に、理解対象が発散するのを防ぐ
- ポインタなどの使用によって静的な解析では不定となる依存関係情報を、実行時情報を利用して補完する

### 3.1.3 動的解析による抽出情報

動的解析では、実際に対象プログラムを実行させるため、プログラムに対する入力を含めた開発者の注目するプログラムの振る舞いについての情報を得ることが可能である。

また、動的解析結果を静的解析へのフィードバックすることで、膨大な情報から実行パスに関連したシンボル依存関係に調査の焦点を絞ったり、静的解析では不定であった情報の補完を行うことができる。

しかしながら、1回のプログラムの実行によって得られる情報は1サンプルに過ぎないため、対象プログラムの全体的な振る舞いを理解するための以下に示す工夫が必要となる。

- 1サンプルの動的解析情報を確実に記録し、複数回の実行結果を統合的に利用する
- 実行パス上のシンボルの依存関係情報を利用した仮想的な他の実行パスを予想する

## 3.2 変更ステージ

前節の理解ステージで得た対象プログラムの構造特性および動作特性の知見により、開発者は対象プログラムに施す変更・拡張箇所の特定とその影響範囲調査の段階へ移る。

このステージで開発者がすべきことは、対象プログラムに施す変更・拡張によって変更・追加されたシンボル群が影響し合うシンボル間依存関係情報を、慎重に見極めることである。これを怠るとバグ組込みを生じる恐れがある。

この時に開発者が利用できるのは、理解ステージにおいて静的解析と動的解析の両者の結果により構築されたシンボル間の依存関係情報である。

また、変更・拡張によって対象プログラムの振る舞いが予測できることが望ましい。この解決として、変更・拡張箇所を実行パスとして含むような動的解析情報の記録を再利用することが有効である。

以上より変更ステージでの必要となる要件は次のようになる。

- 変更・拡張によるプログラム構造上の影響範囲の事前把握が可能であること
- 変更・拡張による動的な振る舞いの変化の指標となる情報が参照可能であること

### 3.3 確認ステージ

確認ステージで開発者がすべきことは、対象プログラムに対しての変更・拡張作業を実行する前後の、対象プログラムの構造的変化と外部的な振る舞いの変化が期待通りであったかと、副作用がないかを確認することである。

よって、ここでも静的解析と動的解析を用いたシンボル間依存関係情報の構築と、実際に対象プログラムを動作させた動的解析情報の記録が必要であり、その差分情報を確認できる環境構築が重要となる。

この変更・拡張作業前後の差分情報を取得し、それだけを確認することで、開発者自身による、実施した変更・拡張の評価が可能であることが、本提案プロセスのポイントである。

プログラムの構造的変化、つまりシンボル間の依存関係の変化を確認するためには、差分を調査した上で膨大なシンボル間依存関係のネットワークから情報を絞らなければならない。また、外部的な振る舞いの変化の確認においても、変更・拡張前後について動的解析の記録情報を比較することになる。しかし、既存デバッガを使用したステップ実行による動作確認では、比較する複数のプログラムの実行シーケンスを同時に確認することは難しい。

以上より、確認ステージで求められる要件をまとめると表 2のようになる。

| 確認情報 | 要件 |
|---|---|
| 静的解析 | 変更・拡張前後に依存関係に変化が生じたシンボル群の抽出、およびそれらのシンボルを中心とした依存関係の参照が可能であること |
| 動的解析 | 動的解析情報を可視化し、開発者にプログラムの動作状況を把握し易くすること |
| | 可視化した動的解析情報は、同時に比較参照を可能にすること |
| | 可視化した動的解析情報は、ソースコード情報およびシンボル間依存関係情報と密接な関係を持たせること |

表2: 確認ステージでの支援要件

### 3.4 改善ステージ

改善ステージは、プログラムの変更・拡張における障害の有無によって実施の有無が変わるサブステージという位置付けではあるが、このステージを実践することによって、以降のサイクルのスムーズな進行に大きく寄与する意味では重要度は高く、本提案プロセスのポイントとなるステージである。

このステージを実施すると判断した場合、開発者が行うべき作業は、コードの再構築作業に集中することである。この時に守らなければならないことは、プログラムの機能拡張は一切行わないことである。そして、コードの再構築作業の際にプログラムの外部的な振る舞いを保つことである。

プログラムの変更・拡張における障害の有無は、対象プログラムの構造上の弱点、つまりプログラム設計上の欠陥が存在するかどうか、に対応する。

開発者は、このような改善ステージを実施するかどうかの判断を行わなければならないため、これを支援するため以下の要件が必要となる。

- プログラムの構造上の欠点を検出が可能であること
- プログラム構造の改善するための手順を提示可能であること

## 4 支援技術

本稿の提案するプロセスの各ステージでの作業支援として、前章で洗い出した様々な要件や問題を解決するための技術について注目し、表 3にまとめる。

| 支援技術 | 解決する要件 |
|---|---|
| シンボル間依存関係抽出技術 | 静的解析での理解対象の発散の防止<br>静的解析での不定となる依存関係情報の補完<br>動的解析結果のフィードバック<br>変更・拡張によるプログラム構造上の影響把握<br>変更・拡張前後のプログラム構造上の変化確認 |
| 仮想再実行技術 | 動的解析結果の記録<br>複数回の動的解析結果の統合的な利用<br>変更・拡張によるプログラム動作上の影響把握<br>変更・拡張前後のプログラム動作上の変化確認 |
| 解析情報の可視化技術 | 変更・拡張によるプログラム構造／動作上の影響把握<br>変更・拡張前後のプログラム構造／動作上の変化確認<br>ソースコード情報との連携<br>動的解析での1サンプルからのプログラム構造情報派生 |
| 静的解析と動的解析の連携技術 | 静的解析での理解対象の発散の防止<br>静的解析での不定となる依存関係情報の補完<br>動的解析での1サンプルでの情報量不足<br>動的解析での1サンプルからのプログラム構造情報派生 |
| 改善箇所検出用メトリクス測定技術 | プログラム構造上の欠点検出<br>プログラム構造の改善のための手順提示 |

表3: ソフトウェア開発支援技術と解決する要件

以下では、各支援技術についての説明を行っていく。なお、説明を行う各技術は、本稿で提案するプログラム開発プロセスでの駆動を基にして、提案および実現したプログラム理解支援ツールであるDEANSUITE[1]での例を用いる。なお、対象プログラム言語はC, C++言語である。

---

[1] DEANSUITE: Program Dependency Analysis Suite

### 4.1 シンボル間依存関係抽出技術

DEANSUITE では、静的解析のエンジンとして、プログラムを構成するシンボル間の依存関係を抽出にスタティックなプログラムスライシング技術を応用している[4][5]。プログラムスライシング技術を用いると、大規模なプログラムであっても調査に必要となる部分を切り出すことによって、調査範囲を絞り込むことが可能となる。ただし、静的解析の限界を考慮するとこれだけでは不十分であるので、動的解析結果のフィードバックを受け入れる。すなわち、動的解析によって得た実行パス上のシンボルを起点とした場合、静的解析では不定となっていた情報がプログラム実行時の情報を反映して依存関係情報の補完する。これによって理解対象の的をさらに絞ることができる。

また、シンボル間の依存関係情報をデータとしてデータベースとして蓄えることによって、開発者の注目シンボルを起点とする依存関係ツリーを出力し、さらにその葉に相当するシンボルを起点とした2次的な依存関係情報の参照というように、シンボル間の依存関係ネットワークを辿っていくことが可能である。

さらに、シンボル間の依存関係情報同士の差分情報出力により、変更・拡張前後のプログラムの構造的変化の確認が可能である。

なお、DEANSUITE ではシンボル間の依存関係としてに表 4に示すものを考える。

| | |
|---|---|
| シンボル A がシンボル B を使用する | 関数Aが関数Bを呼び出す |
| | 関数Aが変数Bの値を変更する |
| | 関数Aが変数Bの値を参照する |
| シンボル A がシンボル B に使用される | 関数Aが関数Bの中で呼び出される |
| | 変数Aが関数Bの中で値を変更される |
| | 変数Aが関数Bの中で値を参照される |

表4: シンボル間の依存関係

### 4.2 仮想再実行技術

動的解析によって得られた情報をプログラム理解に反映させるためには、その情報を記録し、再利用可能なものにしなければならない。既に述べたように理解とは情報参照の反復と試行錯誤によりなされ、何度も再実行して確認できるようにすべきである。

再利用可能となった動的解析情報は、ソースファイルと連動することで、仮想的にプログラムを実行し、その実行時の情報を従来のデバッガツールのようにソースコードに連動してトレース表示可能である。

DEANSUITE が提供する仮想再実行環境は図 3に示す形態で実現する。

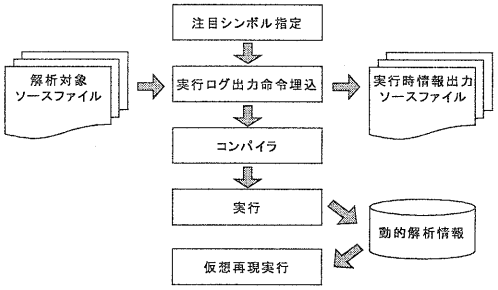



図3: 仮想再現実行の実現手順

ここで実行ログ出力命令は、シンボル間の依存関係情報を用いて、指定された注目シンボルに依存関係がある全てのシンボルを洗い出し、そのソースコードの位置に自動的に埋め込まれる。

DEANSUITE では、再利用可能な動的解析情報を参照することによって、従来のデバッガツールが標準的に備える機能に加え、プログラム実行の逆方向へのトレースと、プログラム動作状況の可視化(後述)が可能である。

### 4.3 解析情報の可視化技術

静的解析によって対象プログラムを構成するシンボル間の依存関係が抽出されたならば、開発者にとって理解し易く出力できなければ意味がない。

DEANSUITE では、開発者の注目するシンボルを基点とした依存関係をツリー状に可視化できるビューを備えている。図 4にシンボル間の依存関係ツリーを表示したビューの例を示す。

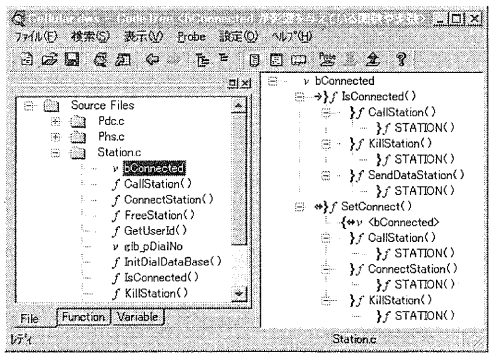



図4: シンボル間依存関係ツリー

一方、動的解析については、対象プログラムの制御の流れと、シンボル間で受け渡されるデータの流れの両者を理解することがポイントである。この二つの流れの変化をイベントしてとらえ、対象プログラムの実行時のシーケンスをチャートとして可視化することが、開発者の直感的な理解を可能とし、大量の情報に対しても対象プログラムの振る舞いをマクロ的に把握可能とする最良策と考える。発生する様々なイベントをトレースするチャートの例を図 5に示す。このチャートは縦がシーケンスの方向で、左に関数群のビュー、右に変数群のビューを持つ。左側ビューの太線により制御の流れを表現し、左右のビューを横断する矢印によりデータの流れを表現している。

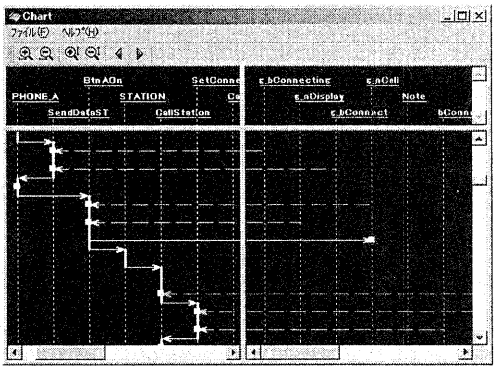



図5: イベントトレースチャート

## 4.4 静的解析と動的解析の連携技術

静的解析と動的解析は互いの短所を相互補完するために、対象プログラムの解析を連携して行う。

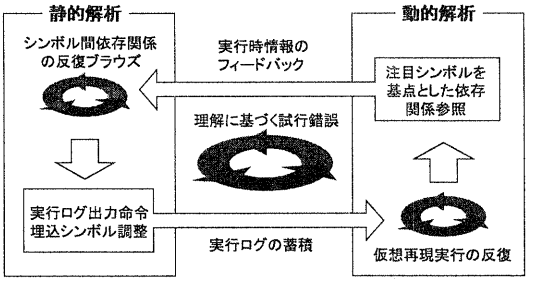



図6: 静的解析と動的解析の連携

静的解析においては、調査対象を絞るため動的解析によって実行ログが取得できるように、依存関係情報から注目箇所として実行ログ出力命令を埋め込むシンボル群を指定し、その情報を仮想再実行への入力として動的解析へ移行する。動的解析は仮想再現実行より得た知見より、詳細な調査を行いたいシンボルを発見し、実行時の情報と共にシンボル間依存関係抽出への入力として静的解析へ移行する。このような静的解析と動的解析のやり取りは図 6で示すようにサイクリックに行われ、人間の理解プロセスの試行錯誤として成り立つ。

## 4.5 改善箇所検出用メトリクス測定技術

本稿が提案するようなサイクリックな開発プロセスでは、各反復毎に以前のサイクルで得られたプログラム構造の知見を基に、変更・拡張作業が行われる。

変更・拡張作業に伴う設計変更が度々発生する場合には、以前のサイクルで実施された変更・拡張が再作業される恐れがある。再作業の発生するような設計変更は設計を悪くしている証拠であり、対象プログラムの構造上の欠陥を的確に検出し、検出箇所に対しての改善処理を施す必要がある。

こうした検出の手段として、プログラムの構造欠陥の兆候をメトリクスとして測定し、検出された兆候に対するプログラム改善処置として対応するリファクタリング技術の適用指針を示す。リファクタリングは Martin Fowler らによって研究が進められているプログラム改善技術であり、プログラムコード内の様々な問題に対する改善処理のカタログ化[6]が進められているため、適用指針として応用可能である。

簡単な例として、C++言語のプログラムなどで、30行を超えるような関数を検出した場合には、その関数の複雑性や複数の機能が混在しているという兆候と判断する。この兆候に対応するリファクタリングの適用として、その関数からの他の関数を抽出したり、その前段階として自動変数の様々な整理を行うことを指針として提示可能となる。本稿では指針提示のために、プログラムの構造欠陥の兆候を検出するメトリクスを表 5で示す3つのレベルで分けて評価し、対応する改善作業を表 6のように整理している。

| 評価メトリクス | 定義 |
|---|---|
| クラスの健全性 | クラス内のメンバ情報やクラス間の関連などのクラスプロパティについて評価 |
| 関数の明瞭性 | 関数構成やスコープ、そして他シンボルとの依存関係について評価 |
| コードブロックの明瞭性 | 自動変数の書込／参照やスコープ、そして他のシンボルとの依存関係について評価 |

表5: 評価メトリクス

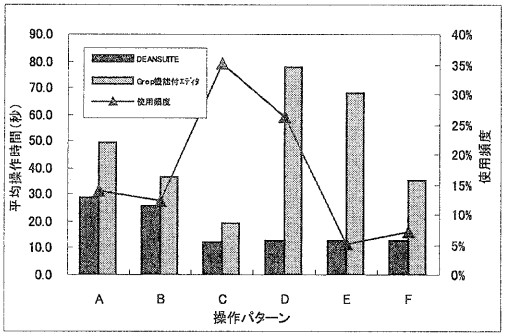
| 注目レベル | クラス | 関数 | コードブロック |
|---|---|---|---|
| 改善作業カテゴリ | クラスの抽出 | 引数の変更 | 関数抽出 |
| | メンバ編集 | 関数の移動 | 自動変数の追加 |
| | クラスの名称変更 | 関数の名称変更 | 自動変数の削除 |
| | クラスの削除 | 関数の削除 | 自動変数の名称変更 |
| | クラスのインライン化 | 関数のインライン化 | 自動変数の初期化 |
| | | | 自動変数のスコープ |

表6: 改善作業カテゴリ

また、単純に静的解析、動的解析を融合して得られた情報より、静的にも動的にも他のシンボルから利用されないことが保証されたシンボル群を調査することによるデッドコード削除を支援することなども可能である。

## 5 評価

本提案プロセスを実プロジェクトに適用した。その結果のうち、シンボル間依存関係抽出技術の適用による工数削減効果の測定について報告する。このプロジェクトの開発規模は、約 900K ステップであり、プログラミング言語にC言語を用いている。実験方法として、実プロジェクトでの変更ステージで行われたDEANSUITE を用いた作業の操作ログを記録し、これを基に同様の作業を再現し、その作業時間を測定した。DEANSUITE 適用以前は Grep 機能付エディタを用いた作業であったため、これと比較した。結果を図 7に示す。




| 操作パターン | 説明 |
|---|---|
| A | あるシンボル(関数や変数)に対し、そのシンボルを使用している個所のコードを見るためにエディタを開く |
| B | ある特定の関数の中で、あるシンボルが使用されている個所のコードを見るためにエディタを開く |
| C | ある関数に対し、その関数の中身のコードを見るためにエディタを開く |
| D | あるシンボルがどのシンボルから使用されているのか調査する |
| E | あるシンボルがどのシンボルを使用しているのか調査する |
| F | ある変数の構造体を調べる。 |

図7: 比較結果

各操作パターンについて平均操作時間に使用割合を掛け合わせた結果を比較すると、従来作業に比べ実に 64%もの作業時間の削減が行われた事が確認された。なお得られた結果は、調査結果を反映した2次的な調査への派生や、変更・拡張前後の差分による調査対象の限定化、といったメリットを考慮していないため、さらに作業時間削減が期待できる。

以上の結果は評価を行った複数の実プロジェクトの内の1つの例ではあるが、他の実プロジェクトでの評価も平均して作業時間を半分から3分の1にするという結果を得ている。

## 6 まとめと今後の課題

本研究では、既存プログラムの利用が中心となるソフトウェア開発において、プログラムの静的解析と動的解析によって得られる両情報を基に、「理解」、「変更」、「確認」の3つのメインステージと、「改善」の1つのサブステージを1サイクルの構成要素としたスパイラルなプログラム開発プロセスを提案した。

本プロセスを実践することにより、既存プログラムの理解向上、プログラムの修正・拡張作業の安全性向上が図られ、ソフトウェア開発工数を削減する効果があることを説明した。また、本提案プロセス内のプログラム構造の改善ステージにより、再作業による後戻り開発の除去、開発スピードの一定化が図れる。さらには開発ソフトウェアの品質向上が見込まれる。


[1] CASE 1988-89, Sentry Market Research, Westborough, MA, 1989, pp. 13-14.

[2] Carm McClure : ベスト CASE 研究グループ訳 : ソフトウェア 開発と保守の戦略, 共立出版, 1993.

[3] 下村隆夫 : プログラムスライシング技術と応用, 共立出版, 1995.7.

[4] Weiser.M. :Program-Slicing, Proceedings of the Fifth International Conference on Software Engineering, 1981.

[5] 本間昭次 他 : プログラムの構造と動作の理解支援, 情報処理学会 第58回全国大会.

[6] Martin Fowler : REFACTORING: Improving the Design of Existing Programs, Addison-Wesley, 1999.