Vol.2011-CE-109 No.1
2011/3/18

プログラミング教育のためのプログラム開発過程分析手法の検討—プログラムを読まずにプログラミング状況を知る試み—

須藤克彦[†1,†2] 孫 一[†1] 大月一弘[†1]

本研究では，教員が学生のコンピュータ・プログラム開発過程における課題を発見し効果的な指導をすることを目的として，プログラムの開発過程を分析する手法を検討した．この方法によれば，ソースコードを精査することなく，ソースコードならびに診断情報（エラー・メッセージ）の量の変化を分析することで，学習者の開発状況をある程度知ることができる．

An analysis of program development process in programming education—An attempt to know the situation of student's programming process without reading programs—

Katsuhiko Sudoh,[†1,†2] Yi Sun[†1]
and Kazuhiro Ohtsuki[†1]

In this study, we examined the technique for analyzing the development process of the program in order to discover the problem in the situation of students and to do effective guidance. Using the amount of the source code and the diagnostic information(error message) without examining the source code closely, our method can let the teacher know learner's development situation roughly.

†1 神戸大学大学院国際文化研究科
Graduate School of Intercultural Studies, Kobe University
†2 神戸情報大学院大学
Kobe Institude of Computing, Graduate School of Infomation Technology

1. はじめに

プログラミング教育—特に初学者に対する—の現場では，プログラミング中に学習者が陥っている問題をいち早く見出し，適切なアドバイスをすることが求められる．

プログラミング教育に関しては，教授方法の提案[3] やオンライン利用による教育内容の改善[4]，初学者のための自習教材の提案[8] などさまざまな研究がなされているが，我々は学習者のプログラミングの過程をトレースすることにより，プログラム開発中の状況を把握することを行った．

また，学習者の達成度を評価するための多くの調査・研究が行われている．[10)6)] プログラミング教育に限らず，学習者の達成度を評価することは容易ではないが，プログラミング教育では一般に成果物としてのプログラムを見ることで評価がなされる．しかしながら，作成されたものを見るだけでは学習者の開発状況を把握することはできないため，プログラムの作成過程を分析することが行われるようになった．そのひとつにプログラムの実行過程を追跡するもの[11] があるほか，プログラムの差分を見てどのような改変を行ったかを調査するものがある．

プログラムの差分を見るための手法として，ソースコードそのものを比較するもののほかに，構文レベルで差分を取得する方法[5] や構文解析した結果である構文木を比較する技法が研究された[1)2)9)]．さらにプログラム作成過程の変遷を可視化する研究も現れた．[7)] プログラムの差分を抽出・比較するこれらの方法により，開発過程でのプログラムの変遷を把握することが可能となった．

これらの手法を用いることにより，プログラムの設計方針やプログラムをどのように段階的に作成しているかを把握することが可能になった．しかしながら、プログラム作成者が開発中にプログラムのエラーなどで立ち往生している，あるいはより基本的な問題により作業が滞っているなどの開発中の進捗度や困難を把握することはできない．

本研究は，開発中のプログラムの開発内容そのものに対してではなく，開発中に出くわす各種のトラブルやエラーなどの問題に対して，順調に問題を解決しているのか，あるいは停滞しているのかといった，開発の状況を見ようとするものである．この状況を把握することにより，学習者に対する助言と開発中のプログラムをチェックするタイミングを見つけるなど，より適切な学習指導に結びつけることができるのではないかと考える．

本研究においては，従来の研究とは異なり，作成されたプログラムの内容を見ることをせず，プログラム開発過程でのソースコード量および誤診情報（エラー・メッセージ）の量の

変化のみに着目する．量の変化という比較的解析可能なデータのみで，どの程度，状況把握が可能であるかどうかについて学生実験のデータをもとに検討する．

2. 基本的な考え方

本研究における基本的な考え方は，プログラム開発途中のソースコードの量（行数）やソースコードの書き換え量およびエラーの量などの数量的な変化から学習者の状況を読み取れるのではないか，ということである．例えば，これまでのシステム開発および教育現場での経験から，一般的に次のようなことが想定される．

- 順調な開発においてはソースコード量は増加する．その過程でエラーも増減するが，ささいなミスによる構文エラーなどの本質的ではないエラーは無視してプログラミングを継続する
- 一方，経験が浅くプログラミングに自信を持てない時期には，エラーのひとつひとつを解消することに注力するため，コード量は増えず徒に手を加えることを繰り返す
- 稀に，開発方針を大きく変更することがある．そのような場合にはソースコードの大半を削除するなど，大きな変更を加えることになるが，一定した方針を持たない改変はあまり有効でない場合が多い

このように，プログラミングの状況がソースコード等の量に現れると考えられ，これらの量を分析することで，学習者の状況を把握し，特に初学者が陥っている問題を発見できるのではないかと考える．

3. 実　　験

上記の想定を検証するために実験を行った．その内容について述べる．

3.1 実験方法

プログラミングの過程ではプログラムは何度も改訂されるが，改訂される度ごとのソースコード量とコンパイル時の診断情報（エラー・メッセージ）の量の変化を分析する．このために，今回はコンパイル時点でのソースコードとコンパイル時にコンパイラが出力する診断情報（エラー・メッセージ）を保存し，後にそれらを分析する．

3.2 実験の実施

実験は筆者のひとりが勤務する神戸情報大学院大学における担当科目での期末試験で実施した．被験者は当該科目の期末試験受験者（学生）23 名であるが，学生の中にはプログラミングの実務経験のある者から，プログラミング経験のまったくないものまで様々な経歴を持つものがいる．また，受験者に出題された課題は Linux で C 言語のプログラムを開発することである．プログラム編集にはテキスト・エディタ `vi` を用い，`gcc` によりコンパイル・リンクを行い，作成されたプログラムを実行・テストする．このプロセスを課題の指示に従って実行する．開発対象であるプログラムの詳細には触れないが，以下のような前提のもとで 90 分の時間内に開発することが求められる．

(1) 基本になるサンプル・プログラムは教員（筆者）が提供する
(2) 受験者はこのサンプル・プログラムから出発して，課題の機能を段階的に追加開発する
(3) 課題は 4 段階に別れており，受験者はこの 4 つの段階に従って機能を追加しなければならない．ただし，はじめの 3 段階は規定の課題であり，最後の 4 段階目は自由に機能を追加してよい
(4) ログの収集
受験者は，筆者が開発したツール[⋆1]を通常のコンパイル・コマンド (`gcc`) の替わりに使用する．このツールを使うことでログが自動的に収集される．収集されたログを本課題の成果として提出する

3.3 実験結果

3.3.1 集　　計

実験結果として得られたデータは 23 人分であり，そのうち有効なデータは 19 名分であった．19 例の集計結果を**表 1** に示す．各欄の意味は次の通りである．

- 学生 ID
学生を識別するための識別番号である．
- ソースコード
学生が開発したプログラムの途中（または最終）の最大の行数を表す．初めに教員（筆者のひとり）がサンプルを提供したがそれは 11 行であった．なお，ここではコンパイルの時点のソースコードをひとつの「版」（バージョン）とする．
- 変化量
前の版との差分の最大値を表す．ここで「差分」とは UNIX の `diff` コマンドによる出力の行数である．

⋆1 コンパイルのための専用スクリプトであり，コンパイルのためのコマンド（`gcc`）の替わりに利用してもらった．このスクリプトはコンパイルを実行するとともに，そのときのソースコードと診断情報（エラー・メッセージ）をログ・ファイルに保存する．

表 1 集計情報一覧
Table 1 List of log infomation

学生 ID	ソースコード	変化量	診断情報	版数
001	70	40	6	21
002	25	38	3	10
003	75	115	7	17
004	22	11	0	7
005	49	25	11	31
006	46	55	9	54
007	33	42	6	23
008	22	21	4	15
009	53	57	7	24
010	80	45	6	28
011	47	38	8	24
012	49	67	3	16
013	43	27	1	14
014	41	36	3	7
015	42	60	13	31
016	35	35	9	20
017	87	34	6	22
018	48	33	4	24
019	62	54	3	15
平均	48.9	43.8	5.7	21.2

- 診断情報
  各版をコンパイルした際の診断情報（エラー・メッセージ）の行数を表す．これには致命的なエラーから警告レベルまですべてを含む．
- 版数
  最大の版数（=コンパイル回数）である．

表 1 について説明する．

（1） ソースコード
最終的なソースコード量は平均で 48.9 行である．成績評価[*1]が高い者はこの数値が概ね大きい．機能的には，50〜70 行程度が妥当と思われるが実装方法によってはそれよりも大きくなる場合もあり，また，独自の機能追加も許しているためその内容によって変動する．

*1 5.3.2 で述べる．

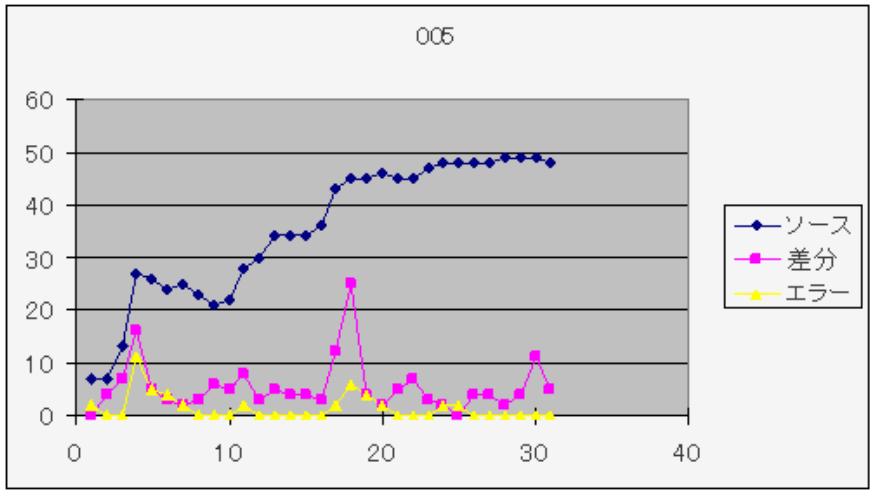

図 1 学生 005
Fig. 1 ex. Student 005

（2） 変化量
概ね，機能追加の際に変化量が大きくなる．これは当然と思われるが，大きな機能追加でない場合でも変化量が大きい場合がある．これは，プログラム構造上は若干の変更であっても，その影響から字下げ（インデント）が大きな範囲で変更されるなど，ソースコード上の変更が大きい場合などである．従って，変化量が大きいことが機能追加を意味しないことがあるため，ソースコード量の実際の増加量と併せて見ることが重要であると言える．

（3） 診断情報
ソースコードの増加量が大きいときに診断情報（エラー・メッセージ）が増える傾向がある．これは新たにコードを追加した場合に新しくエラーが発生することが多いため想像に難くない．

3.3.2 グラフ化

収集データをグラフにしたが，例を**図 1** に示しこれを説明する．

（1） グラフは 3 つの折れ線グラフから構成される．それぞれ以下の量を表す
（a） S：ソースコードの行数

コメントや空行も含む．一般には版が進むにつれて（つまりグラフの右方向に行くにつれて）増える傾向にある．

(b) C：ソースコードの変化量（前の版との差分量）
diff コマンドの出力結果の行数である．従って，追加と削除が多い場合は数値が大きくなる．また，多くの行に修正を加えた場合でも数値が大きくなる．

(c) E：コンパイル時の診断情報（エラーメッセージ）の行数
致命的なエラー（ERROR）や警告（WARNING）をすべて含む．

(2) 縦軸はそれぞれの量（行数）を表す．

(3) 横軸は版（コンパイルの回数）を表す．

このグラフから読み取れるものとそうでないものを整理する．

3.3.2.1 グラフから分かるもの

(1) プログラムの進捗状況
ソースコード量（S）は版ごとに増加するのが一般であるが，このことは，プログラム開発が「進展」していると見てよい．しかしながら，併せて変更量（C）にも注意しなければならない．すなわち，

(a) 増加量と同程度の変化量があった場合は純粋に追加と考えられる

(b) 増加量の倍程度の変化量があった場合は，増加量に相当する行をそっくり書き換えたものと想像できる

このことから，変化量が増加量に比べて多い場合は，それまでの開発方針を変更した場合であり，開発がなんらかの困難に直面していると推測することができる．

(2) コンパイルエラーへの対応状況
エラーに対して適切に対応できているかどうかがわかる．

(a) エラーをなくすために単にソースコードを「削除」する場合があるが，コード量の増減とともに観察することで適切な修正かどうかを推測することができる．

(b) ただし，コード量が増えることが真にコードを増やしているとは限らないことがある（コメントアウトするような場合）．その場合は，ソース（S）の増加分と変化量（C）とを勘案する必要がある．

(c) 続くバージョンでエラーが同様に出ているにも関わらずコードの追加を行っている場合（Eが一定の高い値を示しながらSが増えている場合），エラーを修正することは気にせずにコードの追加を行っていると見做すことができる．これは，開発にある程度の慣れあるいは自信があって，修正をまとめて行おうとしていると推測することができる．

(3) 開発への積極性
上の (1) とも関係するが，全体の版数およびソースコード量を平均値と比べることで，学生が開発を積極的に行っているか，あるいは躊躇せずに実施しているかについて，ある程度の傾向を知ることができる．自信のないまま開発を行った場合，明らかに版数とソースコード量が少ない傾向がある．

(4) 精査すべきソースコード
学生が作ったプログラムを評価するためには，最終版を精査することが基本であるが，最終版以外にも点検すべきソースコードがある．

(a) ソースコード量が目立って増えた版

(b) エラーが大量に出た後のもので，エラーが解消または減少した版

前者は，機能追加の際に学生がどのように振舞ったかの要点をつかむために有効である．後者は，エラー対応の適切さを見るとともの，プログラミング言語の文法などの基礎についての問題は解決した版であるので，プログラミングの基礎的な知識と技術を確認することができる．

3.3.2.2 グラフからは読み取れないもの

上ではグラフのパターンから推測できることを述べたが，その一方で，このパターンからでは読み取れない内容がある．

(1) コーディングの書式
コーディング・スタイルは知る由もない．

(2) リファクタリングの成果（完成度）
どのように関数定義を行い，あるいはコードの書き換えを行ったなど，開発過程におけるリファクタリングについては，ソースコードを精査しないと判断しようがない．

(3) エラーの真の原因を見誤っている状況
コンパイルエラーがあったときに，真の原因を検出できずに徒にコードに手を加えてエラーを拡大してしまう場合がある．このような状況では，具体的なエラーの内容と改訂の内容とを見なければ，学習者が陥っている問題を知ることは困難である．

(4) コード追加ができない原因
コードの追加そのものに躊躇していることが伺えるような場合に，何が原因であるかはこのグラフからは読み取れない．一般的には，プログラミング言語の基本的な文法を理解していない場合やプログラミングそのものに慣れていないと考えることはでき

るが，それ以上の情報をもたらすものではない．

4. 分　　析

上の実験で得られたグラフを以下の手順により分析した．

(1) 開発の過程をいくつかのフェーズに分割する．ここで「フェーズ」とは作業がソースコードを追加する段階にあるのか，あるいはそれを修正している段階にあるのかなど，作業がどのような状況にあるかを表す．

(2) 次に，結果のグラフに，上の「フェーズ」がどのように現れているかを分析した．

(3) このフェーズの現れ方から，受験者を開発成果によるグループに分類した．

4.1 開発のフェーズ

グラフを見ると，グラフの増減にいくつかの傾向があることがわかる．グラフの局所的な変化に着目すると，開発の過程は概ね以下の五つのフェーズから成り立つと考えることができる．

A　ソースコードの追加フェーズ
新規にコードを追加する，あるいは新規機能を追加する過程である．ソースコード量が増えるに伴ってエラー量も増えるが，ソースコードを削除することは多くないため変化量がソースの増加量を越えることはない．

B　チェック・修正フェーズ
コードをチェックし，コンパイル・エラーの修正などの微調整を行っているフェーズと考えられる．ソースコードの増減はさほど多くはない．エラーは減少する．

C　安定・停滞フェーズ
ソースコードおよび変化量おおびエラーが横ばいである．エラーは少ないのが一般である．これはプログラミングとしては安定していると見ることができる．一定の機能が完成したところで，ロジック（アルゴリズム）を再検討するなどの検討を行っていると推定できる．ただし，次のステップに進むことを躊躇しているとも見られるので，このようなフェーズとした．

D　改造フェーズ
初めの「追加」と似た傾向を示すが，変化量が大きいことと，その割にはエラーが大きくは増えないことが特徴である．新規機能の追加というよりは，既存の機能の範囲でプログラミングを変更しているものと想像できる．

E　見直しフェーズ
ソースコードが著しく減少するフェーズである．一般には，それまでのプログラムを全面的に書き換えるなどの場合である．方針転換をして書き換えるのか，あるいは致命的な問題を発見したためか，その理由は知るよしもないが開発過程になんらかの問題があったことが推察される．

表 2　開発フェーズ
Table 2　Phases of development

記号	フェーズ	ソース	変化量	エラー	備考
A	追加	↗	↗	↗	
B	修正	→	↗	↘	
C	安定	→	→	→	または停滞
D	改造	↗	↗	↗	変化量がソース増よりも大きい
E	見直し	↘	↗	↗	前版の廃棄

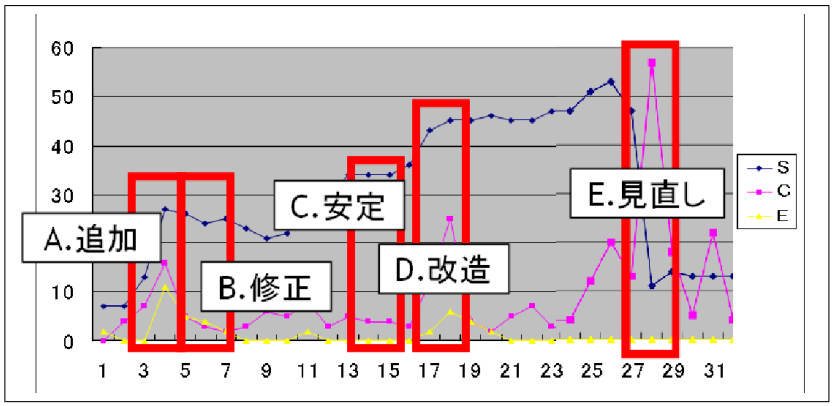

図 2　開発のフェーズ
Fig. 2　Phases of development

これらをまとめると **表 2** のようになる．

これらのうちはじめの四つの開発過程は一般的に現れるが，5 番目は稀に現れる．グラフ上で示せば**図 2** のようになる．なお，図 2 は，上の五つのフェーズがわかるようにグラフ上の典型的な箇所を示した．図上の他の部分もいずれかのフェーズに分類されるが，図示する上で煩雑になるのですべてのフェーズを示すことはしていない．

4.2 フェーズの現れ方

これらのフェーズが，実験結果の 19 例にどのように現れるかを見た．すなわち，以下の

表 3 開発フェーズの出現
Table 3 List of development phases

ID	開発のフェーズの現れ方										
001	A	B	C	D	C	E	C	D	B	C	
002	A	B	D	B							
003	A	B	E	A	E	B	C				
004	A	B	C	A	C						
005	A	B	D	A	B	D	B	C			
006	C	D	C	D	C	C	D	E			
007	A	B	D	B	A	E	D	C			
008	C	D	B	D	B	C					
009	A	D	B	D	B	C	A	E	B	C	
010	A	B	A	B	D	B	C	D	B	D	
011	A	B	A	B	D	B	D	B			
012	A	B	C	D	B	C	E	D			
013	A	B	D	B	A	B	A				
014	C	A	D	B							
015	A	B	D	B	E	D	E	D	B		
016	A	D	B	D	E	D	B	C	A		
017	A	B	A	B	C	A	B	A	B		
018	A	B	C	A	B	C	D	B	D	B	D
019	A	B	A	B	D	B	C				

手順に従ってグラフの変化を追って局所ごとにフェーズに分けた．

- グラフの部分部分の変動がどのフェーズに該当するかを 4.1 と表 2 を参照しながら目視によって判断した
- 主にフェーズ C（安定）の場合であるが，同様のフェーズが連続して現れる場合にはひとつのフェーズにまとめた

この作業の結果，19 例のグラフに上記のフェーズがどのように出現するかを**表 3** に示す．

次に，これら開発フェーズの現れ方に特徴がないかを検討した．一見してわかるのは横方向の長さで違いである．これは版の数，つまりコンパイルした回数の違いから来るものである．また，多くの学生はフェーズ A（追加）から始まっているが一部学生が C（安定または停滞）となっている．E（見直し）は先にも述べたように稀に現れている．このようにフェーズの現れ方の傾向から学生を分類する．

表 3 と表 1 を基礎として，開発フェーズの現れ方を集約した結果，開発全体を表す型として学生を以下の四つのタイプに分けた．

(1) 標準 1
開発が順調であったと想定されるタイプ．フェーズには A（または D）と B が交互に現れる傾向が見られる．ところどころに C がある．表 1 のコード量および版数が平均よりも大きい．

(2) 標準 2
開発は順調であったが，最終的な進捗がおもわしくなかったと想定されるタイプ．フェーズの現れ方は順調型と同等にであるが，最終的なコード量および版数が多くない点を加味した．

(3) 停滞型
開発作業が捗らず，進捗が思わしくないと想定されるタイプ．作業の進め方に困難がある．フェーズの現れかたは一見順調なものに見える場合でも，コード量および版数が著しく小さい．

(4) 混乱型
途中で方針転換をするなど，結果的には順調とはいえないと想定されるタイプ．典型的に E フェーズが現れる場合をこの型とする．

表 4 開発全体の傾向によるグルーピング
Table 4 Grouping of development process type

タイプ	学生
標準 1	005, 010, 011, 017, 018, 019
標準 2	008, 009, 012, 013
停滞型	002, 004, 014
混乱型	001, 003, 006, 007, 015, 016

各学生がどの開発タイプに属するかを分類した．この分類には表 3 からフェーズの現れ方を見ると同時に表 1 のソースコード量および版数を見て決定した．この結果を**表 4** に示す．

4.3 分析の評価

3.3.2 で述べたように，グラフの形状からプログラミングの状況を一定程度知ることができる．しかしそれは目視によるものであった．表 3 を作成する過程でも目視による方法を採ったが，この表は収集した数値と表 2 の判定規則を使うことにより，機械的に作成することも可能である．さらに，表 3 から学習者の開発成果のタイプを知ることができる．従って，グラフを目視するのではなく実験結果から自動的にプログラミング状況を見ることが可能になると言える．

5. 結　　論

今回の実験から得られた結果を評価するとともに，今後の課題と拡張性について述べる．

5.1 有 効 性

グラフにおけるフェーズの現れ方から，学習者が以下のような問題に直面している場合に，その状況を知ることができる．

(1) コンパイルエラーがあったとき，適切な改修をせず徒にコードを改変している
(2) どこから手をつけてよいかわからないまま時間が経過する
(3) 大きな「見直し」を行っている場合，とくにそれを繰り返している場合には，概ね開発方針が定められずに困っていることが多い

5.2 課　　題

既述したが，グラフからだけでは読み取れないものも多々ある．それらは，実際のソースコードをも読むなどして，具体的に学生が直面していた問題を分析するほかはない．また，今回は同じテーマについて同様の作業をすることでその結果を見たが，これが学生ごとに異なった課題に取り組んだ場合は，結果を比較して相対的に評価することはできない．

また，開発成果のタイプでは標準 1 と想定されたものの実際の評点が低かった者があった．このことは，開発のタイプだけでは問題を抱えている学習者を見逃す可能性があることを示している．

5.3 拡張と応用

5.3.1 時間軸での分析

今回の実験ではコンパイルを行った時刻も記録している．図 1 のグラフを時間軸で表示すれば**図 3** のようになる．

時間軸を取り入れることにより，次のコンパイルまでにどれほどの時間を要していたかを把握することができ，作業中どこに時間をかけていたかを知ることが出来る．この情報を利用すればより詳しく精査すべきソースコードも特定できると思われる．しかしながら，これらの様子を見ることができるものの，版の間の変化が見えにくくなるため，今回は版数を横軸に取ったものを利用した．

また，本研究ではソースコードの量を重要な要素としたが，ソースコードにはコメントなど，本質的には意味を持たないものが多々含まれている．コメントを除いた純粋なコード量を使うことが必要であろう．さらには，物理的な行数ではなく，文（*statement*）の数など，セマンティクス（意味論）を考慮した「量」を対象にする必要があろう．初めに紹介した構文木による分析・比較などを活用することが有効と思われる．

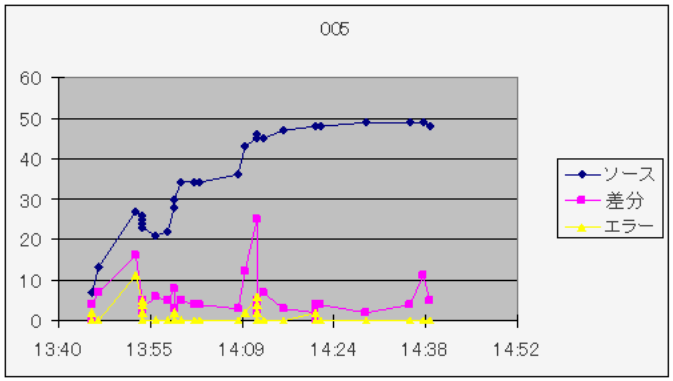

図 3 学生 005（時間軸）
Fig. 3 ex.005(time axes)

さらに，今回の分析手法は，一定の作業が終了した後にそのログを収集して行ったが，これを授業中のリアルタイムな学習支援に応用することが考えられる．すなわち，リアルタイムで学習者の状況をモニターするような仕組みと組み合わせれば，授業時間内での学生の進捗状況を把握することが可能となろう．

5.3.2 成果タイプと成績

本実験は，実際の試験問題として課せられたものでもある．試験の成績評価のためには作成されたプログラムを詳細に検討して評定を下した．その際には，作業結果の提出方法などの付帯的な課題も評価対象となっているが，ここでの評価は，開発プロセスを評価するために以下の項目について再評価して評点とした．すなわち，5 つの評価項目のそれぞれについて一定の成果が認められた場合，各 1 点を与え合計で 5 点満点とした．

評価項目は次の通りである．

(1) 課題に従って段階的にプログラムを発展的に改造しているか
課題は 4 段階の改造（機能追加）を課している．なお，最後の 4 段階目は任意の改造であるので必須ではないが，独自の工夫や完成度を評価する際に利用した．
(2) エラー対応が適切か
コンパイル・エラーがあったとき，どのような対応をしたかで評価した．エラーを解消するためにエラー箇所のコードを単に削除するなどは適切な対応とは看做さない．

表 5 評点
Table 5 Evaluation for students

学生 ID	1	2	3	4	5	合計
001	1	1	0	0	0	2
002	1	0	1	0	0	2
003	1	0	1	0	0	2
004	0	0	0	0	0	0
005	1	1	1	1	1	5
006	1	1	1	0	0	3
007	1	1	0	1	0	3
008	1	1	1	0	0	3
009	1	1	1	0	0	3
010	1	1	1	1	1	5
011	0	1	0	0	0	1
012	1	1	0	0	0	2
013	0	0	0	0	0	0
014	1	1	1	0	0	3
015	1	1	0	0	0	2
016	0	1	1	0	0	2
017	1	1	1	1	1	5
018	1	1	1	1	0	4
019	1	1	1	1	1	5
平均	0.8	0.8	0.6	0.3	0.2	2.7

また，構文エラーなどに対して合理的な変更をしているかを見た．

(3) コーディング書式が統一的か
コーディング規約は特に設けていないが，各自が統一的なコーディングをしているかを見た．字下げ（インデント）や構造（if 文など）の選択に一定の規律があれば良い．

(4) 独自の工夫が見られるか
ロジック（アルゴリズム）あるいは利用した関数（標準関数，システムコール）に工夫があるかを見た．ほとんどはライブラリ関数だけで解決できる課題であるが，どのような組み合わせを使うかを見た．

(5) 完成度が高いか
主にリファクタリングをしているかを見た．すなわち，関数の定義が合理的か，無駄なコードの除去をしたかなどを見た．

評点を**表 5** に示す．

さらに表 5 の結果を評点でグルーピングしてみると**表 6** のようになる．

表 6 評点によるグルーピング
Table 6 Grouping by evaluations

評価点	学生 ID
5	005, 010, 017, 019
4	018
3	006, 007, 008, 009, 014
2	001, 002, 003, 012, 015, 016
1	011
0	004, 013

表 6 で下線付きのものは先の表 4 で「停滞型」または「混乱型」にグルーピングされた者を表す．

これらのことから，成績についてもある傾向を読み取ることもできる．

(1) 成績優秀者ではあくまで成績の低いものとの相対的な評価ではあるが，概ね以下のような傾向が見られる．
- (a) バージョン数（コンパイル回数）が多い
- (b) 最終的なソースコードの量が多い
- (c) 段階的な改造が見て取れる（変化のピークが何回かに分かれて現れる）

(2) 作業が進まなかった者には上記の優秀者とは逆の傾向が見られる．
- (a) バージョン数（コンパイル数）が少ない
- (b) 最終的なソースコード量が少ない
- (c) コードの増加が少なく，また段階的な発展が見られない

このように，ひとつの課題を与えた環境では，相対的ではあるが学生の進捗状況の多くを，グラフだけから把握することができる．

6. おわりに

本研究により，ソースコードを精査することなく，ソースコードならびに診断情報の量の変化を分析することで，学習者の開発状況をある程度知ることができるがわかった．

参考文献

1) 会沢実，練林，井上克郎，鳥居宏次，構文の比較に基づいたプログラム差分の表示方法，情報処理学会 全国大会，1994.6
2) 会沢実，練林，荻原剛志，井上克郎，鳥居宏次，構文木の比較によるプログラム開発

Vol.2011-CE-109 No.1
2011/3/18

履歴分析ツールの試作，ソフトウェア工学 100-8，1994.9
3) Susan M.Eitelman，Computer tutoring for programming education，Proceedings of the 44th annual Southeast regional，2006
4) Gregor Fischer，Improving the quality of programming education by online assessment，Proceedings of the 4th international symposium on Principles and practice of programming in Java，2006
5) 福安直樹，吉田敦，プログラムの構文要素に基づく版管理システムのための差分取得手法，電子情報通信学会技術研究報告.SS，ソフトウェアサイエンス，105(490)，pp43-48，2005.12
6) 匂坂智子，渡辺成良，プログラミング初学者の学習方略と段階的理解度に関する調査および支援ルールの作成について（特集 次世代情報教育の構築に向けて）–（プログラミング教育），教育システム情報学会誌 26(1)，pp5-15，2009
7) 水穂良平，土田康太，浜名隆広，佐藤匡正，PSF 手法に基づくプログラム作成過程における変遷の可視化，電子情報通信学会技報，2005.10
8) 岡本雅子，寺川佳代子，喜多一，初学者を対象とした自習中心のプログラミング教材について，情報教育研究集会講演論文集，2009 年度，pp179-182，2009
9) 練林，井上克郎，鳥居宏次，構文木間の対応に基づくプログラムテキスト比較ツールの試作，電子情報通信学会技術研究報告，SS93-18，1993.6
10) 武内亮，佐藤匡正，プログラミング教育における合理的評価法，情報処理学会研究報告，2004.9
11) 内村隆聖，瀬川典久，杉野栄二，澤本潤，実行履歴差分を用いたプログラム学習ツールの提案，情報処理学会 70 回全国大会，2007.10