NAIST-IS-DD1561020

博士論文

ソフトウェア工学教育の改善を目的とした
初学者による探索的行動の分析

槇原 絵里奈

2018 年 3 月 15 日

奈良先端科学技術大学院大学
情報科学研究科 情報科学専攻

本論文は奈良先端科学技術大学院大学情報科学研究科に
博士 (工学) 授与の要件として提出した博士論文である。

槇原 絵里奈

審査委員：

飯田 元 教授	（主指導教員）
松本 健一 教授	（副指導教員）
市川 昊平 准教授	（副指導教員）
崔 恩瀞 助教	（副指導教員）
井垣 宏 准教授	（大阪工業大学）
吉田 則裕 准教授	（名古屋大学）

ソフトウェア工学教育の改善を目的とした初学者による探索的行動の分析*

槇原 絵里奈

内容梗概

第 4 次産業革命により社会におけるソフトウェアが果される役割はより一層重要なものとなった．そこで，情報系学部学科を有する殆どの教育機関において，プログラミング演習やソフトウェア開発演習といった，プログラムのコーディングやテスト等，ソフトウェア開発に必要な技術を学生が手を動かしながら学ぶ実践的な形式の演習が開講されている．

これら演習において学生のプログラミング行動の分析による支援が行われている．プログラミング行動とは，開発者がソフトウェアを開発するにあたって行ったコーディング，コンパイル，デバッグ，テスト等一連の過程における内容や成果物，及びそれらに関連する一連のメトリクスを指す．本研究ではプログラミング行動のうち，エラーや異常出力を修正するために開発者が行った試行錯誤に関わる振る舞いを探索的行動と定義し，プログラミング演習とソフトウェア開発演習において学生の探索的行動の分析を行った．探索的行動を分析することで，学生がソフトウェア開発を行うにあたっての理解度や躓きに応じた支援・教育が可能になると考える．

まず，プログラミング演習においてソースコードの編集履歴に着目し探索的行動を収集・分析した．さらに，探索的行動が，条件分岐や繰り返し処理など，プログラミングのどの要素に対して行われているか，すなわち学生が躓いているプログラミングの要素を自動検出する手法を提案した．提案手法により教員に負担を強いることなく教員サイドから各学生のプログラミングに対する試行錯誤や理解度を推測することが可能となった．

*奈良先端科学技術大学院大学 情報科学研究科 情報科学専攻 博士論文, NAIST-IS-DD1561020, 2018 年 3 月 15 日.

次に，教育プロジェクトである CloudSpiral のカリキュラムに組み込まれている，ソフトウェア開発演習においてビルドエラーに着目した探索的行動を収集・分析した．2013 年度から 2015 年度のビルドエラーのログを基に，エラーの分類や各種エラーが発生する原因等について調査を行った結果，個人で解決可能なエラーとチームメンバーと協力することで解決可能なエラーを特定した．この結果を基に CloudSpiral の教員が 2016 年度の同開発演習の改善を行った結果，エラー数の減少やエラー解決時間の減少に繋がった．本研究の結果はソフトウェア開発演習において教員が補填すべき内容を明らかにしただけでなく，学生がビルドエラーを引き起こしたときの詳細なアドバイスを可能にする．

キーワード

ソフトウェア開発支援，探索的行動，探索的プログラミング，ビルド活動，ソフトウェア工学教育

An investigation on exploratory behavior by novices aiming at improving software engineering education*

Erina Makihara

Abstract

As exemplified by Industry 4.0, software has become to a key role in our society. Therefore, many educational institutions which have an information/computer science department or faculty have been conducting practical exercises, named as programming exercise and software development exercise. In these exercise, students are required to learn software development techniques (e.g. coding, testing, debugging) by performing these techniques by themselves. Investigations on programming behaviors by students are conducted in these exercises. Programming behavior refers to a collection of metrics corresponding to coding, compiling, debugging, testing contents and products while a programmer develops software. In this research, I focused on the exploratory programming behavior and investigated it performed by students in introductory programming exercise and software development exercise. The exploratory behavior means the portions of programming behavior referring to trial-and-error cycle in order to solve the errors or incorrect outputs in software development. By collecting and investigating the exploratory behaviors by students, I assume that it is able to support and educate students based on the understanding and difficulties of software development by students. In my work, I focused on programming exercise and software development exercise.

*Doctoral Dissertation, Department of Information Science, Graduate School of Information Science, Nara Institute of Science and Technology, NAIST-IS-DD1561020, March 15, 2018.

Firstly, I collected and investigated the exploratory behavior, or named as exploratory programming behavior, by putting my focus on the editing histories by students. Moreover, I proposed an algorithm of automatically detecting which portion of source code is performed as exploratory programming by students. I collected the programming behavior logs from real introductory programming exercise, and detected exploratory programming behaviors by using my proposed algorithm. As the result, I am able to identify students who struggled for on the assignments or the programming factors (e.g. if, while, array). As well, I am also able to guess the causes of students' failure and detect the different approaches among the students. My proposed algorithm is able to automatically detect exploratory programming behavior by her/him without forcing workloads on educators. Educators are able to give more accurate and detailed advices for each student based on her/his progress, approach that she/he wants to perform and her/his difficult factors on programming.

Secondly, I collected and investigated the exploratory behavior by analyzing build errors from each team in software development exercise conducted as a part of CloudSpiral, which is one of the educational projects. In detail, I categorized the build errors and investigated the cause of each build error based on the build logs collected in CloudSpiral from 2013 to 2015. As the result, I identified the errors caused by technical factors and also poor communication among team members. In 2016, the instructors modified the software development exercise based on my analyzed results. As the result, in 2016, the number of build errors and the time required to solve the build errors decreased compared to previous years. My results are able to not only provide instructors the solution of each build error, but also give students detailed advices from educators when build errors were occurred.

Keywords:

Software development support, Exploratory behavior, Exploratory programming,

Build activity, Software engineering education

関連発表論文

学術論文誌

1. 槇原 絵里奈, 井垣 宏, 吉田 則裕, 藤原 賢二, 飯田 元, “プログラミング演習における探索的プログラミング行動の自動検出手法の提案”, コンピュータソフトウェア (採録決定). (第 2 章に関連する)
2. 槇原 絵里奈, 井垣 宏, 吉田 則裕, 藤原 賢二, 川島 尚己, 飯田 元, “ソフトウェア開発 PBL におけるビルドエラーの調査”, 情報処理学会論文誌, Vol. 58, No. 4, pp.871-884 , 2017 年 4 月. (第 3 章に関連する)
3. 槇原 絵里奈, 井垣 宏, 藤原 賢二, 吉田 則裕, 飯田 元, “初学者向けプログラミング演習のための探索的プログラミング支援環境 Pockets の提案”, 情報処理学会論文誌, Vol.57, No.1, pp.236-247, 2016 年 1 月. (第 2 章に関連する)

査読付き国際会議・ワークショップ

1. Erina Makihara, Hiroshi Igaki, Norihiro Yoshida, Kenji Fujiwara, Hajimu Iida, “Detecting exploratory programming behaviors for introductory programming exercises”, In Proceedings of the 24th IEEE International Conference on Program Comprehension (ICPC 2016), pp. 1-4, Austin, TX, USA, May 2016. (第 2 章に関連する)
2. Erina Makihara, “Pockets: a Tool to Support Exploratory Programming for Novices and Educators,” In Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015), pp. 1066-1068,

Bergamo, Italy, September 2015. (第 2 章に関連する)

国内研究集会

1. 槇原 絵里奈, 井垣 宏, 吉田 則裕, 藤原 賢二, 飯田 元 , “探索的プログラミング行動の自動検出によるモデル化の検討”, 情報処理学会研究報告, Vol. 2016-SE-191, No. 16, pp. 18, 2016 年. (第 2 章に関連する)
2. 槇原 絵里奈, 井垣 宏, 藤原 賢二, 上村 恭平, 吉田 則裕, 飯田 元, “初学者向けプログラミング演習における探索的プログラミングの実態調査と支援手法の提案”, 第 21 回ソフトウェア工学の基礎ワークショップ (FOSE 2014), pp. 123-128, 2014 年 12 月. (第 2 章に関連する)
3. 槇原 絵里奈, 藤原 賢二, Putchong Uthayopas, Chantana Chantrapornchai, Jittat Fakcharoenphol, 井垣 宏, 吉田 則裕, 飯田 元, “日本とタイにおけるプログラミング初学者のプログラミング行動の比較”, 電子情報通信学会技術研究報告, Vol. 114, No. 260, pp. 47-52, 2014 年 10 月. (第 2 章に関連する)

目次

図目次

表目次

第 1 章

序論

1.1. 研究の背景

近年，第 4 次産業革命に代表されるように，IoT やビッグデータ，AI などの技術が一般に広く普及している．それに伴い，社会におけるソフトウェアが担う役割の重要性も拡大し，高度な IT 活用能力を持つ技術者の育成は急務であるといえる [55].

これらを背景に，情報系学部学科を有する殆どの高等教育機関においてプログラミング演習やソフトウェア開発演習といった，実際に学生が手を動かしてプログラミングやテストなど様々なソフトウェア開発に関わる作業を実際に行う形式の科目が開講されている．学生はこれらの科目を通じて教員から適切な支援を受けることでプログラミングに対する理解を深め，実社会のソフトウェア開発において活用できる実践的な能力を培うことができると考えられる．しかしながら，プログラミングを学ぶことは初学者にとって非常に困難であることが知られている [25, 34]．Watson らが 15 の異なる国における 51 の機関において、161 件のプログラミング入門コース (CS1) のドロップアウト率を調査した結果，演習の受講を途中で諦めるあるいは単位を落としてしまう学生が，全体の受講生のうち平均して約 30% を占めることが分かった [59]．したがって，プログラミング演習やその応用にあたるソフトウェア開発演習における初学者の難所は，教員が適切に把握し，そして支援を行う必要がある．

プログラミング演習及びソフトウェア開発演習の支援を目的とした研究の一つにプログラミング行動の収集・分析があげられる [2, 7, 8, 18, 19, 23, 26, 52–54, 57]．こ

こで述べるプログラミング行動とは，学生がプログラミングを行うにあたって実施された一連のコーディング，コンパイル，デバッグ，テストの内容や対応した成果物，エラー内容，及びそれらに関連するメトリクスを指す [16]．学生のプログラミング行動を分析することで，教員は学生がプログラミングやソフトウェア開発を行うにあたっての難所や問題点を適切に発見でき，学生の演習に対するモチベーションの低下やドロップアウトを防ぐことが可能であると考えられる．

教育機関における学生のプログラミング行動を対象とした研究は，プログラミング演習における個人の開発を支援・分析対象としているものと，その発展にあたるソフトウェア開発演習において複数名の開発者が参加するチーム開発を支援・分析対象としているものに分けることができる．前者の例として，デバッグの過程に着目し，初学者がプログラムの実行動作を理解するために有効であるとみなし，デバッグすべき箇所を提案したり，ブレークポイントの設定やステップ実行を容易にするための研究が行われている [2, 4, 23, 26, 27, 32, 36, 53]．後者では，学生にチームでアジャイル型ソフトウェア開発を行ってもらい，ソフトウェア開発に不慣れな学生がアジャイル型ソフトウェア開発を行う上での難所や問題点を分析したり，効果的なフィードバックを行うための調査が実施されている [1, 5, 24, 31, 35, 42, 43, 46, 51, 58]．

本研究では，教育機関におけるプログラミング演習及びソフトウェア開発演習を支援するにあたり，March の述べた探索に着目する [30]．March は探索（Exploration）を，検索（search），変動（variation），冒険（risk taking），実験（experimentation），遊び（play），多様化（flexibility），発見（discovery），改新（innovation）によって得られるものを指し，探索により未知の領域に対する新たな可能性を発見でき，その領域に対する知識を広げることができると述べている．

本研究ではさらに，March の述べた探索の定義を拡張し，ソフトウェア工学教育における探索的行動を，「ソフトウェア開発におけるプログラムのコーディング，ビルド，テスト，デバッグなど一連の過程，あるいは特定の段階におけるプログラミング行動のうち，エラーや異常出力を修正するため，あるいは未完成のソフトウェアを完成させるために開発者が行った試行錯誤に関わる振る舞い」と定義する，

また，試行錯誤とは，トライ・アンド・エラーのような失敗を経験しながら開発をすすめるものだけではなく，思いついた複数の実装を試すためにソースコードを以前の状態に戻すような失敗を含まない行動も含む．

近年，プログラミング演習やソフトウェア開発演習を開講する教育機関は増えて続けており，プログラミング学習者の多様化が指摘されている [55]．プログラミング学習者の多様化とは，すなわち従来プログラミング演習は主に情報系に所属する学生を対象に開講されていたが，近年は非理系，また小学生など，様々な非情報系の教育機関においても開講されていることを指す．そのような環境の中，プログラミング行動の中でも特に探索的行動に着目することで，学習者がどのようなプログラミングの要素に躓いているのかといった，学習者の理解度をベースにした柔軟なプログラミング教育が可能になると考えた．学習者の理解度をベースにしたプログラミング教育を行うことで，非理系や小学生など，コンピュータの扱いかたそのものや英語に不慣れな学習者でも，モチベーションを下げることなく，適切なプログラミング教育の支援が可能になると考えた．

プログラミング演習やソフトウェア開発演習において各開発者，すなわち学生の探索的行動を分析することで，学生がプログラミングやソフトウェア開発を行う上での難所を見つけることが可能である．さらに，エラーメッセージやエラーを含まないが想定外の出力が行われたときに，学生がどのような修正を行い解決に取り組もうとしたのかといった，ソフトウェア開発に対する理解度も調査できる．各演習において学生のプログラミングやソフトウェア開発に対する難所及び理解度を分析・調査することで，各学生の理解度に応じたアドバイスや課題の配布など，より柔軟なソフトウェア開発教育を行うことが可能になると考える．

以下，プログラミング演習とソフトウェア開発演習における探索的行動の調査について詳細を述べる．

プログラミング演習における探索的行動の調査

通常プログラミング演習では，特定のプログラミング言語を扱うために必要な，基礎文法，条件分岐，繰り返しなど，プログラミングの要素のまとまりごとに段階的に教育を進めていく．演習中は，学生は教員によって与えられた複数の課題を，基本的には独力で，コーディング，コンパイル，実行，デバッグ及び修正していくことが求められる．

プログラミング演習における探索的行動の支援として，Matthew らは BlueJ[*1]を用いて学生が引き起こすコンパイルエラーの実態調査及び支援ツールを提案・開発した [17]．実際のプログラミング演習では初学者はコンパイルエラーだけではなく，エラーを出力しないものの望んだとおりの出力とも異なる異常出力の解決も求められる．すなわち，プログラミング演習において初学者の探索的プログラミングを支援するためには，エラー結果のみに着目せず，ソースコードをどのように修正した結果どのような出力を得たのか，出力に至るまでの過程に着目する必要があると考える．

そこで本研究ではプログラミング演習におけるソースコードの変更履歴に着目し，学生がプログラミングのどのような要素や課題に躓いているかを分析する．まず，各学生が実装したソースコードのスナップショット，コンパイル・実行時間及び対応する結果など，一連のプログラミング行動を収集する．そして，探索的行動が行われている箇所，すなわち特定の学生にとってソースコードの実装方法が不明確な箇所を特定することで，その学生がプログラミングのどの要素に対して躓き試行錯誤を行っているか検出及び支援を行うことが可能であると考える

ソフトウェア開発演習における探索的行動の調査

プログラミング演習に比べソフトウェア開発演習では，学生はソフトウェアの設計からコーディング，そしてテストの実施まで求められる場合がある．近年，複数名がチームを組みプロジェクト形式で実施される Project-Based Learning (PBL) 形式によるソフトウェア開発演習 (ソフトウェア開発 PBL [9,29]) や，開発形態も従来のウォーターフォール型だけではなくアジャイル型ソフトウェア開発を採用している機関も多く存在する [1,35].

本研究では特にビルドの過程に着目して調査・分析を行う．ビルドとはソースコードのコンパイルやライブラリのリンクなどを行い，最終的な実行ファイルを作成する工程のことを指す．多くの既存研究がビルドはソフトウェア開発において必要不可欠な工程であると述べており [11,40,49]，ビルドの支援のためにビルドエラーの調査や支援ツールの開発などが行われている．

特にソフトウェア開発 PBL においては，扱うファイルの多さや，複数人が同一の

[*1] https://www.bluej.org/

ソースコードを修正することから，ビルド時のエラー原因の特定が困難であることが考えられる．ビルドエラーに着目して調査を行うことで，教員は学生に対しビルドエラーの原因に関するアドバイスや，補足資料などで補填すべき要点が明らかになると考える．

1.2. 本研究の目的と貢献

本研究の目的は初学者が行う探索的行動を，個人がプログラミングを行う形式の演習であるプログラミング演習と，複数名の学生がチームを組みソフトウェア開発を行うソフトウェア開発演習の 2 つの粒度において分析し，その支援手法を提案することである．2 つの粒度の探索的行動を収集・分析することで，各学生のプログラミングやソフトウェア開発における躓きに応じた具体的なアドバイスが可能になると考える．

本研究ではまず，ソースコードの変更履歴に着目した初学者が行う探索的行動の調査について，東京工科大学の学生のプログラミング演習時のログを基に実態調査を行った．そして得られた結果を基に初学者向け探索的プログラミング支援環境を開発し，大阪府立大学工業高等専門学校及び奈良先端科学技術大学院大学の学生に実際に使用してもらった．また，提案ツールより得られたログをさらに可視化することで，各学生の特定の課題に対する取り組み方の違いを教員サイドから発見することが可能となった．ログの可視化に際してソースコードの編集履歴のうち探索が行われている箇所を，提案環境を用いて自動検出を行った．自動検出により教員に負担をかけることなく，学生がどのプログラミングの要素あるいは課題について躓いているか検出することが可能となった．

次に，ビルドエラーに着目した初学者の探索的行動の調査を，教育プロジェクトである CloudSpiral において実施した．CloudSpiral のカリキュラムの一端であるソフトウェア開発演習において，学生がビルドを行うときのログを基に調査を行った．まず，2013 年度から 2015 年度のビルドログを対象に実態調査を行った．具体的には，初学者が陥りやすいビルドエラーを調査するために，Seo らの Google 社におけるビルドエラーの報告を基に [49]，エラーログのカテゴライズ及び各エラー数，エラー継続時間を主に分析し，各エラーの原因，解決策について講じた．そし

て，2013 年度から 2015 年度の調査で得られたビルドエラーが生じる原因及び解決策を基に，CloudSpiral の教員によって 2016 年度の同ソフトウェア開発演習の改善がなされた．改善の結果，ビルドエラー数は最も多かった 2014 年度に比べ，2016 年度は 7 分の 1 にまで削減した．

本研究では教育機関におけるプログラミング演習及びソフトウェア開発演習を対象としているが，先述したように探索的行動は初学者がソフトウェアについて理解を深める過程において必ずとられる行動である．本研究で得られた分析結果や支援手法は学生の自学自習においても活用できるだけでなく，e-learning などオンラインのプログラミング環境と組み合わせることでプログラミングを行う場所に依存しないプログラミング学習支援が可能になると考えられる．

本研究の具体的な貢献は次のとおりである．

プログラミング演習における探索的プログラミング行動を自動検出するアルゴリズムの提案

本研究で提案したアルゴリズムは，ソースコードの編集履歴における差分に着目するため，初学者に多い文法エラーを含むソースコードでも探索的プログラミング行動が検出可能である．また，構文解析や特別なツールなども必要としないため，様々な教育機関だけでなく，e-learning，MOOCs を用いたオンラインのプログラミング教育にも活用可能である．

ソフトウェア開発演習におけるビルドエラーの分類の再検討

本研究では熟練の開発者向けのビルドエラーの分類を基に，ソフトウェア開発経験の浅い学生を対象にビルドエラーについて調査を行った．この際，熟練の開発者向けのビルドエラーの分析で行われていた手法に固執せず，リサーチクエスチョンによる定量的な分析と，付録 C，D に掲載しているような複数種類のグラフを用いて定性的な分析を行い，ビルドエラーの分類に関して再検討を行った．さらに，従来研究では述べられていなかった各種ビルドエラーを減らすための具体的な解決策についてまで講じた．

1.3. 本論文の構成

第 2 章ではソースコードの編集履歴に着目した初学者の探索的行動についてまとめる．以下，ソースコードの編集履歴に着目した初学者が行う探索的行動を探索的プログラミングと呼ぶ．まず初学者が行う探索的プログラミングについて実態調査を行ったため，その内容及び結果について述べる．次に，実態調査の結果を基に初学者向け探索的プログラミング支援環境について提案する．提案環境によって実際に初学者の探索的プログラミングの支援が可能か調査するために，大阪府立工業高等専門学校及び奈良先端科学技術大学院大学の学生へ提案環境を使用してもらった．さらに，探索的プログラミングを自動検出することで，教員に負担を強いることなく特定の課題やプログラミングに躓いている学生を教員は発見できると考えた．自動検出のためのアルゴリズムについて述べた後に，検出結果を可視化することで教員が得られる情報やどのようなアドバイスを学生へ与えることが可能化についても述べる．

次に第 3 章でビルドエラーに着目した初学者の探索的行動の調査についてまとめる．まず，調査対象となる CloudSpiral 及び CloudSpiral のカリキュラムの一部として開講されているソフトウェア開発演習について説明する．次に，2013 年度から 2015 年度の CloudSpiral におけるソフトウェア開発演習を対象に，ビルド活動を分析するためのリサーチクエスチョンについて述べた後に，各リサーチクエスチョンの結果を述べる．各リサーチクエスチョンでは，学生が引き起こしたビルドエラーを分類した上で，各ビルドエラーの解決策について講じる．さらに，2013 年度から 2015 年度の調査結果を基に，2016 年度の CloudSpiral の同ソフトウェア開発演習において，ビルドエラーを生じさせないための改善が行われたため，その結果についても述べる．

最後に 4 章を結論とし，まとめを述べる．

第 2 章

コード編集履歴に基づいた探索的行動に関する調査

本章ではソースコードの編集履歴に着目した初学者向けプログラミング演習における探索的行動の調査及び支援手法について講じる．本研究では特に，開発者が実装に躓いた際に行われる，修正，コンパイル，プログラムの実行，デバッグのサイクルである探索的プログラミング [48,50] に着目する．本章では探索的プログラミングを探索的行動の一部であるとみなし，初学者が行う探索的プログラミングの支援を目的とした．

まず，本研究の調査対象であるプログラミング行動や初学者向けプログラミング演習について関連研究を交え説明する．次に，探索的プログラミングを対象とした既存研究について述べた後，初学者が行う探索的プログラミングについて実態調査を行ったためその結果を述べる．

なお，従来研究において述べられていた探索的プログラミングの定義や特徴については 2.1.3 章で取り扱う．さらに，本研究において行った初学者が行う探索的プログラミングの実態調査の結果をふまえ，拡張した探索的プログラミングの定義を 2.2.1 章で述べる．

実態調査から得られた結果を基に，初学者向け探索的プログラミング支援環境について提案する．最後に提案環境を用いた探索的プログラミングの自動収集手法について提案し，提案手法によって得られた結果及び考察を述べる．

2.1. 関連研究・用語

2.1.1. プログラミング行動

プログラマがソフトウェアやシステムを開発するにあたって行ったコーティング，コンパイル，デバッグ，テストなどの成果物及び対応するメトリクスをプログラミング行動と呼ぶ [16]．Zimmerman は継続的にプログラミング行動を収集・モニタリングできる環境は教育現場において重要であると述べており，近年の教育機関ではプログラミング行動のログに課題や成績などの情報も加え分析する Educational Data Mining (EDM) が行われている [8].

Hermans らは小学生向けオンラインプログラミングコースにおいてプログラミング行動や受講生のプロフィール情報を収集・分析している [12]．さらに彼女らはオンラインで収集されるデータを機械学習で分析することで，コースをドロップアウトするおそれのある受講生を演習初期段階で発見が可能か調査を行った．彼女らの調査の結果，演習の最初の週の課題の成績が良い学生はコースを修了する可能性が高いことが分かった.

Arto らはプログラミング能力の向上には数学の知識が必要であると述べ，コンピュータ・サイエンス (CS) 学科において同時期に開講されているプログラミングの授業と数学の授業に着目した [57]．プログラミングにかかる時間やエラー数などのプログラミング行動のログと数学の授業の成績を比べたところ、プログラミングの授業において締め切り間際に課題を提出する学生は数学の授業に失敗する可能性が高いことを求めた.

2.1.2. プログラミング演習

近年，情報系学部学科を有する殆どの教育機関において，学生が教員により与えられた課題を実現するためにプログラミングを行う形式の演習が開講されている．本論文では学部初期において開講される，プログラミング初学者のプログラミング入門を目的とした演習をプログラミング演習と呼ぶ.

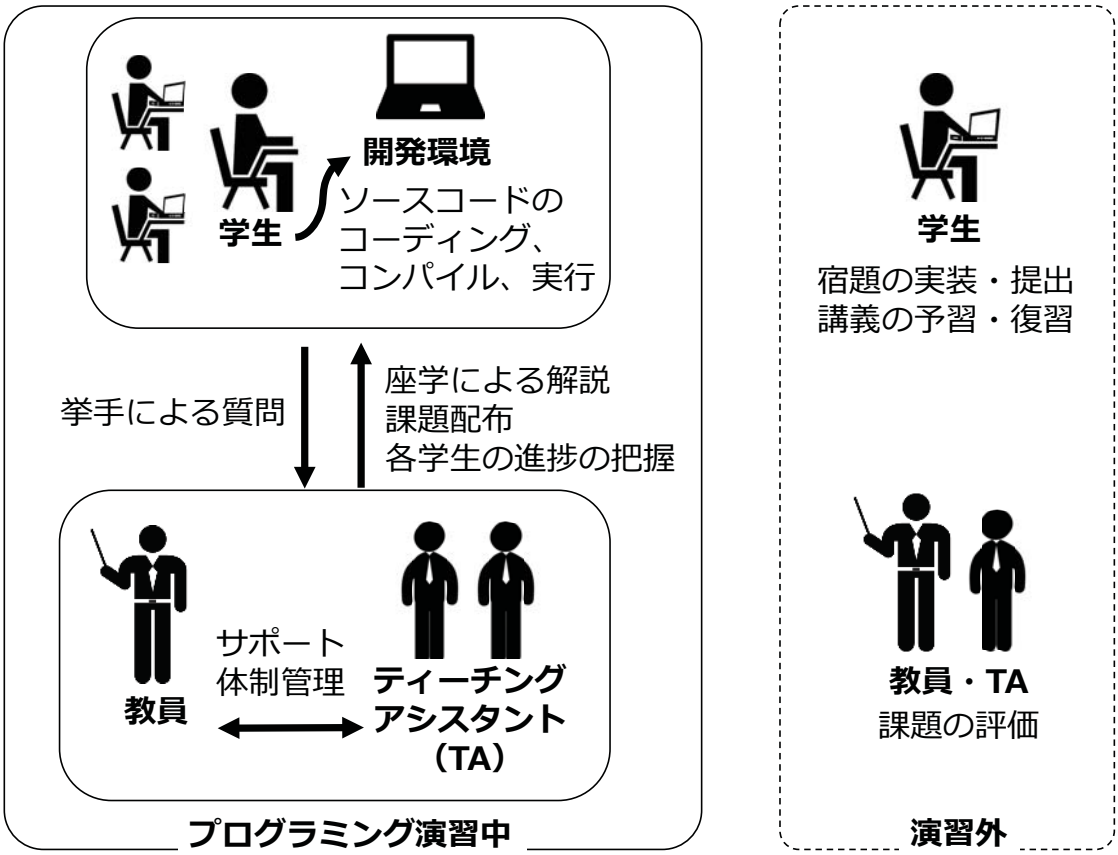


図 2.1 プログラミング演習中/演習外における学生・教員のタスク

図 2.1 に一般的なプログラミング演習における教員と学生のタスクを示す．プログラミング演習は数名の教員・ティーチングアシスタント (以下 TA と呼ぶ) と受講生である学生で構成されることが多く，特定のプログラミング言語を扱うために必要な基礎文法，繰り返し処理，条件分岐，配列など，プログラミングの要素ごとに各単元が分かれている．演習では，学生は教員による座学を受け，そして教員が準備した複数の課題を解くことで，各単元で扱うプログラミングの要素について学習することが求められる．課題について，教員による解説が行われる場合もあるが，基本的には受講生が各々の理解度に応じて独力で解き進めることが求められる．演習中，学生は課題の読解に躓いた場合，挙手により教員にアドバイスを求めることが可能である．教員や TA は，挙手を行った学生へアドバイスを与えるだけでなく，手が止まっていたり特定の課題に長時間躓いている学生を見つけるなど，各学生の進捗

を把握することが求められる．演習中に解くことができなかった課題は宿題となり，学生は決められた期日までに自学自習を行い提出することが求められる．また，各単元において各学生の理解度を計るためにレポートの提出が求められたり，小テストや中間テストが行われる場合もある．演習外におけるタスクとして，学生は先述した宿題や授業の復習予習，教員は各学生が提出した課題の評価を行うことが求められる．

2.1.3. 探索的プログラミング

ソフトウェア開発において，特に開発者が自身にとって不慣れな言語や API，アルゴリズムを使用する際，実行時あるいはテスト時に失敗してしまうことが考えられる．その際，開発者はエラーメッセージを新たな要求だとみなし，要求を満たすためにソフトウェアを再設計し，そして再び修正，実行，テストを繰り返していき，徐々にソフトウェアを完成形へと近づけていく [48, 60]．このようにインタラクティブにソフトウェアの実装と設計を繰り返す方法を探索的プログラミングと呼ぶ [50]．

Rosson ら [47] の実験では，開発者らの主要な振る舞いとして，プログラミングを行い，実行・テストし，プログラムが開発者の意図した状態であるかを評価し，意図にそぐわない場合はコーディング対象の全てあるいは一部分を削除してやり直すといったものが観測されている．また，Brandt ら [3] は，開発者らは複数の可能性を試行する目的で頻繁に自身のコードを以前の状態に戻す，と実験結果に基づいて述べている．

Sandberg [48] によると，開発者が探索的にプログラミングを進めることにより，対象に対する理解を深め，ソースコードの品質を改善することができるとされている．そのため，仕様が曖昧な場合や新しい言語，アルゴリズムを用いて開発を行う場合において，探索的にプログラミングを進めることが望ましい．同様に Myers ら [56] は，複数の変更パターンを反復的に評価することは高品質なソフトウェア設計を行う上でも重要であると述べている．さらにプログラミング初学者でも，新しい知識や技術について学ぶ際に，探索的プログラミングを行うことが望まれると述べている．

Kery ら [20] は，探索的プログラミングは特に途中で変更の可能性があるようなソフトウェア開発において効果的であると述べる一方で，近年探索的プログラミングの概念はソフトウェア工学だけでなく，プログラミング教育やデジタルアートなど，広い分野へ浸透しているため定義が曖昧であり，ツールを用いたサポートが困難で

あると指摘している．そのため，彼女らは従来の Sheil が述べていた探索的プログラミングの拡張した上で，複数の分野における探索的プログラミングの使われ方や特徴を調査し，探索的プログラミングの分析や支援ツールに必要な要件について考察している．

2.2. 初学者が行う探索的プログラミングの支援

2.2.1. 初学者が行う探索的プログラミングの実態調査

Myer らはプログラミングにおいて初学者が新たな技術を学ぶとき必ず探索的プログラミングを行うと述べている．一方で彼らは，探索的プログラミングはエラーを促進する危険性を含んでいるとも述べている [56]．例えば Sheil [50] が述べるようなコンパイルに成功するプロトタイプの作成は，初学者にとって困難であることが考えられ，プロトタイプ作成の過程で躓く可能性がある．同様に，探索を行う過程においてソースコードが冗長になったり，可読性が下がり新たなエラーを生じさせてしまうおそれもあると考えられる．したがって，初学者の探索的プログラミングを支援するにあたって，まず探索的プログラミングを行う上での難所を調査する必要があると考えた．

以上の結果をふまえ，実際の初学者向けプログラミング演習において，学生がどのように探索的プログラミングを行っているかを分析する．本分析では，初学者向けのプログラミング演習を前提とするため，一般的な探索的プログラミングの定義とは対象が異なる．まず，従来述べられていた探索的プログラミングが特に不慣れな言語や API，特定のアルゴリズムを扱うときを主に想定していたことに対し，プログラミング演習では各単元において，取り扱うプログラミング言語の基礎文法から条件分岐，繰り返し処理など，単一のプログラミングの要素について教育が行われる．さらに，従来述べられていた探索的プログラミングでは，ソフトウェアの設計，コーディング，実行，テスト，そして再設計を繰り返すことでソフトウェアを徐々に完成形へ近づけていくことを指していたが，プログラミング演習において学生が与えられる課題は基本的には単一のファイルで解くことが可能なものが多く，テストや設計のプロセスは明示的には行われないことが多い．

以上より，本調査では初学者が行う探索的プログラミングを「特定のプログラミ

ングの要素に対して，編集及びコンパイル，実行の一連のプログラミング行動が連続して複数回行われていること」とみなし，調査を進める．具体的には，まずプログラミング演習において課題を解くために初学者が行ったソースコードの修正履歴を，後述するツールを用いて収集する．収集したソースコードの修正履歴を基に，(1) 初学者が行う探索的プログラミングの実態調査，(2) 初学者向け探索的プログラミング支援ツールの提案・開発，(3) 探索的プログラミング自動検出のアルゴリズムの提案を行う．

初学者が行う探索的プログラミングの実態調査は，井垣らの開発した初学者向けプログラミング演習支援ツール C3PV の実験で得られたデータを使用した [15]．C3PV はブラウザ上で動作する web アプリケーションである．ユーザである学生らはブラウザを介して C3PV にアクセスすることで，ブラウザ上でソースコードのコーディング，保存，コンパイル，実行，コンパイルや実行結果の閲覧が可能となる．また，学生がブラウザ上で行った上記の振る舞いや対応する結果，時間は全てサーバ上のデータベースへ保存されていく．井垣らは C3PV によって得られた学生のプログラミング行動から，どの学生に優先的に指導を行えばいいのか教員が知るために複数のメトリクスを計測し可視化している．

本調査では学部一年生を対象とした Java のプログラミング演習において，C3PV が収集した学生のプログラミング行動のログを扱う．データの収集は，合計 15 回あるプログラミング演習のうち，12，13 回目の単元において実施された．2 時限の授業で取り扱う課題は合計 17 問で，このうち 8 問の提出が必須となっている．課題の内容は，今までの演習で学んだ条件分岐，Loop 文，簡単な String API の扱い方やメソッドの作成などを組み合わせたものである．また，新たにクラスを作成する 2 問を除き，15 問の課題の模範解答は 20 行以内で収まる．

C3PV によって収集された，各学生のソースコードの変更履歴のうち，先述した初学者が行う探索的プログラミングの定義にあてはまる変更を含むリビジョンを分析した．

分析の結果，初学者が行った探索的プログラミングとして，手戻り (Backtracking) を伴うものと伴わないものの 2 種類が観測された．なおここで手戻りとは，探索的プログラミングにおいて頻繁に観測される，開発者が自身のコードを以前と同一の状態に戻す振る舞いのことを指す [37]．以降，観測された 2 種類の探索的プログラ

ミングについて詳述する．

手戻りを伴う探索的プログラミング

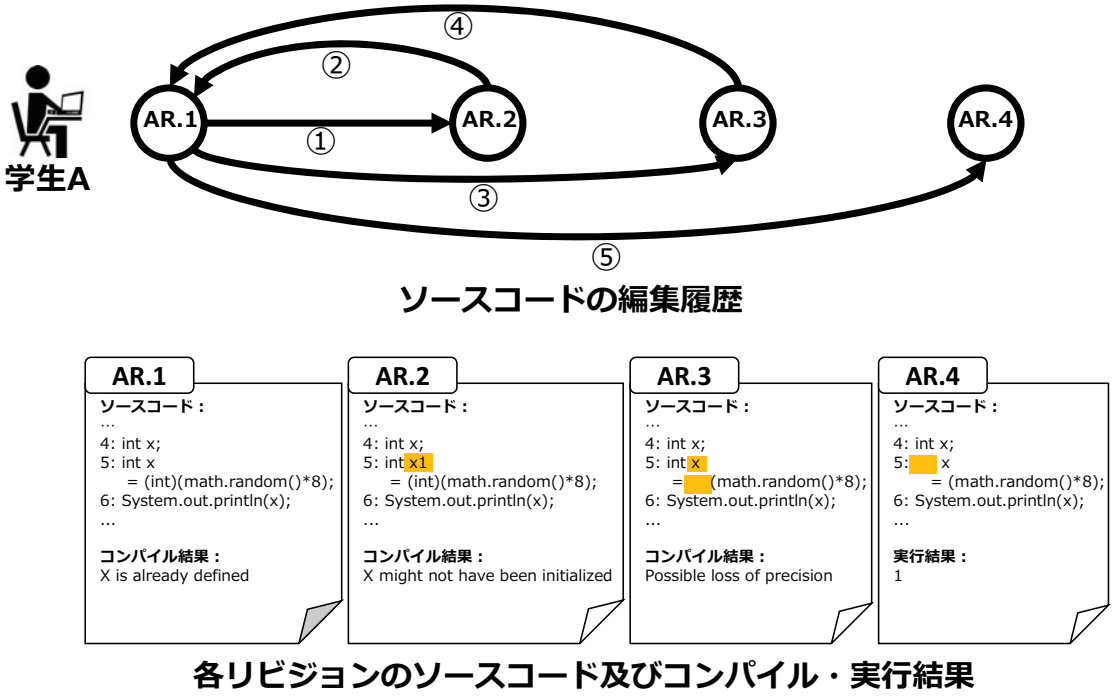


図 2.2　手戻りを伴う探索的プログラミングの例

図 2.2 に手戻りを伴う探索的プログラミングの変更例を示す．変更は AR.1→AR.2→AR.1→AR.3→AR.1→AR.4 の順序で行われた．この学生の場合，AR.1 でコンパイルを行った後，AR.2 から AR.4 にかけて 5 行目の変数 x の代入文を重点的に編集している．また，いずれのリビジョンでもコンパイルあるいは実行が行われており，かつ AR.1，AR.2，AR.3 のときはコンパイルエラーあるいは警告が生じている．

このような変更を行う理由として，過去と現在のソースコードの内容及び結果を見比べるためであると考えられる．例えば図 2.2 の場合，AR.1 では x の宣言が重複しているため宣言の重複を指す出力エラーが生じていた．この学生は変数 x に問題があると考え変数を x から x1 に変更している．しかし，次は x が初期化されていないため，変数の初期化を行っていないという警告が新たに生じた．ここでこの学生は

先ほど得たエラーと違うエラーが出現したことに気づき，先ほどのエラーは何だったのかを知るために再度 R1 の状態へ手戻りしコンパイルを行っている．AR.1 から AR.3 へ遷移した後も，double 型を int 型に代入していることを指摘する，AR.1 とは別の警告が生じている．その後，この学生は自身が変更すべき箇所は，変数名でもランダム関数の呼び出し前の int でもないことが分かり，x の宣言が重複していることに気づき，AR.4 で正しい変更を行った．

手戻りを伴わない探索的プログラミング

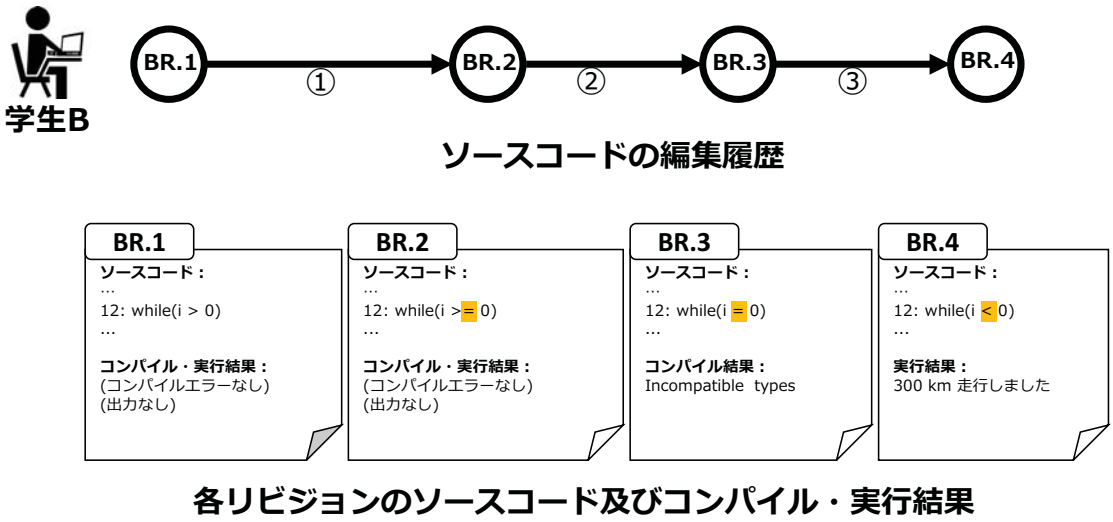


図 2.3　手戻りを伴わない探索的プログラミングの例

図 2.3 に手戻りを伴わない探索的プログラミングの例を示す．変更は BR.1→BR.2→BR.3→BR.4 の順で行われており，この学生は 12 行目にあたる while 文の条件を集中的に編集している．また，いずれのリビジョンでもコンパイルあるいは実行が行われており，かつ BR.1，BR.2，BR.3 のときはコンパイルエラーあるいは何も出力されないといった出力ミスが生じている．

図 2.3 の場合，while 文の条件式に何かしらのエラーの原因があることが分かっているが，文法について理解していないため演算子で考えられるものを順に代入，その都度コンパイルあるいは実行を行っていることが分かる．このような変更パターンは for 文，if 文の条件式でも同様に行われていることを確認した．

このように初学者が行う探索的プログラミングには手戻り，すなわち過去の同一状態にソースコードを戻すものと戻さないものがあることが実態調査により観測された．

初学者が行う探索的プログラミングを支援する上での課題

一般的な探索的プログラミングの課題として，探索の難しさがあげられる．手戻りを伴う，伴わないに関わらず，探索的にプログラミングを行うためには，一度記述したソースコードを編集し，再実行する必要がある．そのため，修正時のバグ混入や意図した状態に手戻りができなくなるといった問題が発生することがある [60]．前章で述べた初学者が行う探索的プログラミングの実態調査では，180 分で 17 問の課題を解くことが求められるプログラミング演習において，16 人の学生が探索的プログラミングを実施しており，探索には平均して 12.1 分程度費やしていることが確認できた．

これらの実態調査の結果や Sandberg [48] らの研究から，初学者の多くは探索的プログラミングを行っており，その際にソースコードが複雑になってしまい，混乱してしまうことがあるということが分かった．したがって，混乱を招くことなく探索的プログラミングを支援するツールがプログラミング初学者には必要であるといえる．

探索的プログラミングを行うにあたって，ソースコードのどの箇所を変更したか，またそれによって出力結果がどのように変化したかを一覧できることが望まれる．しかし，プログラミング演習ではテキストエディタとコマンドプロンプトを用いて演習を進める場合が多い．この場合，過去のソースコードやその実行結果を知るためには，テキストエディタの Undo 機能を特定のソースコードまで繰り返し，再びコンパイル及び実行をする必要がある．手戻りを行うときも同様である．Eclipse[*1] や Visual studio[*2] を使用している教育機関も存在するが，これらはプログラミング演習で使用する機能以外にも非常に多くの複雑な機能を持ち，初学者にとって望ましくないといえる [54]．

そこで，本研究では以下の要件を満たす探索的プログラミング支援環境を提案する．

[*1] https://eclipse.org/

[*2] http://www.microsoft.com/ja-jp/dev/default.aspx

R1　過去のソースコードの実行結果が一覧できる
R2　特定の過去のソースコードへ容易に手戻りが行える
R3　過去と現在のソースコード及び結果の比較が行える

特定のソースコードへ手戻りを行うためには，まず過去に自身がソースコードへ行った保存，コンパイル，実行といった動作，及びそれによって生じたエラーの有無が分かる必要がある（R1，R3）．さらにボタン一つで手戻りが行えることで（R2），例えば求めている出力結果とは違うがエラーが生じないソースコードや，コーディングを始めたばかりの複雑化していないソースコードへ容易に戻すことができる．これらの機能により，探索的プログラミングを行う上でのソースコードの冗長化，複雑化が緩和できると考えられる．また，手戻りの有無に関係なく，自身が加えたどの変更が結果にどのような影響を及ぼしたかを知ることができることが望ましい．

以上をふまえて，次章で本研究で提案する探索的プログラミング支援環境について詳述する．

2.2.2.　初学者向け探索的プログラミング支援ツール: Pockets

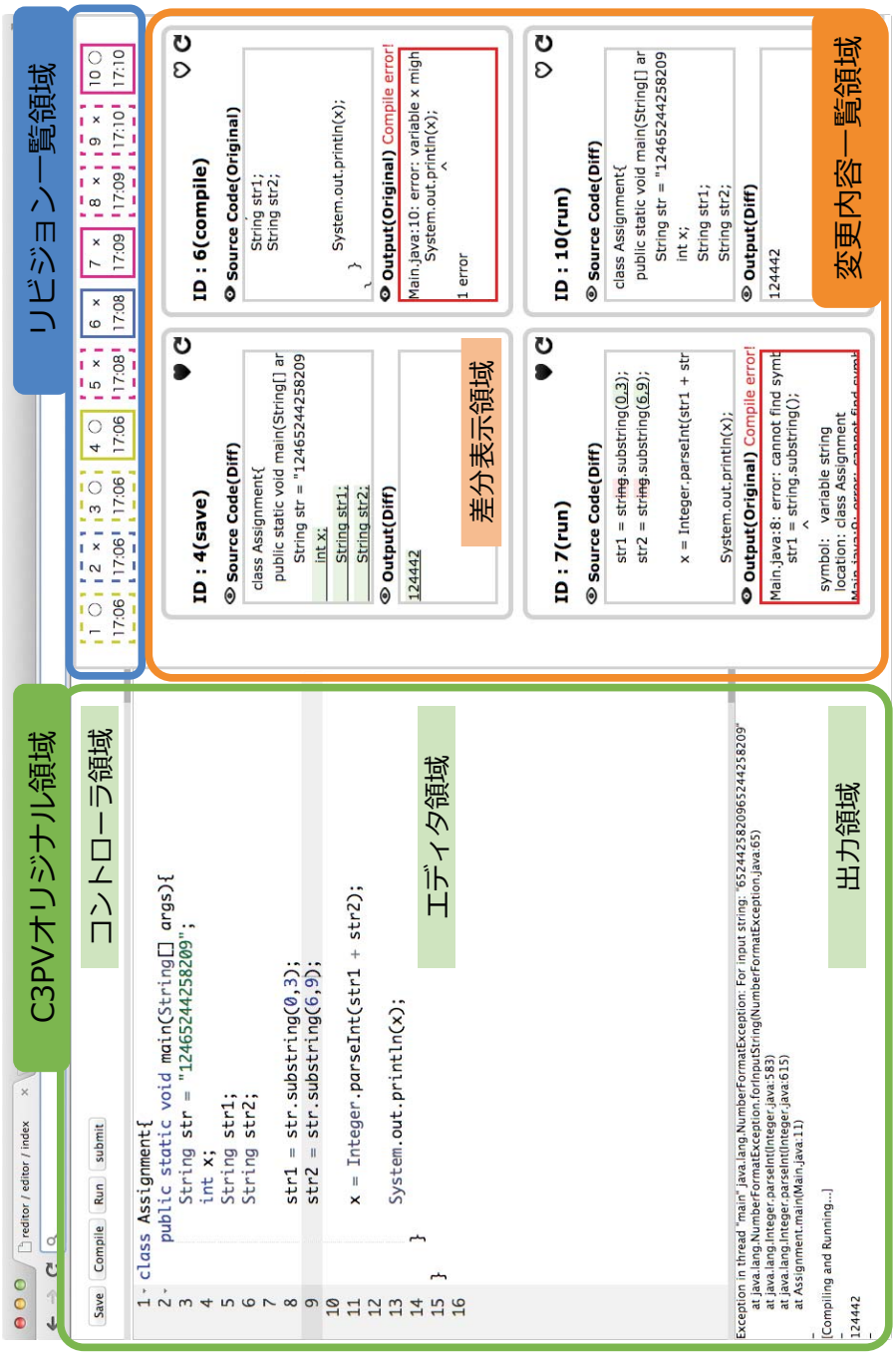


図 2.4　Pockets の UI

本研究では初学者が行う探索的プログラミング支援のため，以下の機能を持つプログラミング環境 Pockets を提案する．F1 から F3 のそれぞれの機能は 2.2.1 章の要件 R1 から R3 へそれぞれ対応している．

F1　リビジョン一覧表示機能
F2　手戻り機能
F3　差分表示機能

Pockets は，井垣らが開発した C3PV を基に拡張する形で実装している [15]．図 2.4 に Pockets の UI を示す．Pockets は「C3PV オリジナル領域」「リビジョン一覧領域」「変更内容一覧領域」の 3 つの領域によって構成されている．

なお，C3PV オリジナル領域はコントローラ領域，エディタ領域，出力領域の 3 つの小領域に分かれている．C3PV の機能により，学生が記述したソースコードは，保存（Save），コンパイル（Compile），実行（Run），提出（Submit）の各ボタンが押されたとき，あるいは 1 分毎に自動で，最新のソースコードのスナップショットがデータベースへ保存される．また，実行はコンパイルも兼ねている．

ソースコードのスナップショットがデータベースへ保存されると同時に，学生番号，時間，コンパイルや実行の成否，コンパイル結果，実行結果，出力内容なども自動でデータベースへ保存される．他にも C3PV の機能として，1 文字単位で修正された文字あるいは文も自動保存されており，ソースコードの総行数や課題ごとのコーディング時間なども表示することができるが，これらの機能は本研究においては使用しない．

Pockets を用いたコーディングの流れを以下に示す．

手順 1　エディタ領域に表示されているオンラインエディタ[*3]を用いて課題のコーディングを行う．
手順 2　コントローラ領域にある保存，コンパイル，実行ボタンのいずれかを押す．それに対応した動作がソースコードに対して行われる．
手順 3　コンパイルあるいは実行結果が出力領域へ，サムネイル（2.2.2 項にて説明）がリビジョン一覧領域へ，差分表示領域（2.2.2 項にて説明）が変更内容一覧

[*3] http://ace.c9.io/

領域へそれぞれ出力される．ユーザは差分表示領域を見て，自身が加えた変更が結果にどのような影響を及ぼしたかを確認する．

手順 4　必要があれば手戻り機能を用いて過去のリビジョンへ手戻りを行う．

ユーザは手順 1 から 4 を繰り返すことで，課題のコーディングを進める．

Pockets は C3PV のオリジナル領域に対し，新たに画面右上にリビジョン一覧領域が，画面右下に変更内容一覧領域がそれぞれ備わっている．リビジョン一覧領域ではリビジョン一覧表示機能を，変更内容一覧領域では差分表示機能と手戻り機能をそれぞれ提供する．以降，各領域が保持する機能について詳述する．

リビジョン一覧領域

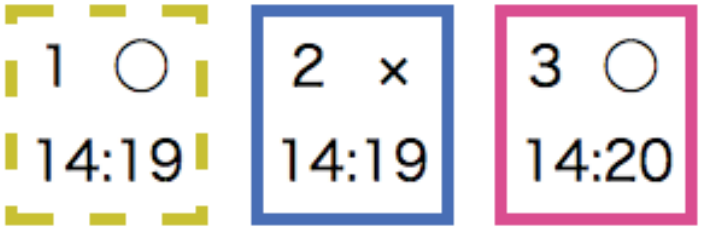


図 2.5　リビジョン一覧領域におけるサムネイルの例

コントローラ領域で Save，Compile，Run のいずれかのボタンが押されたとき，リビジョン一覧領域にそれぞれの動作に対応した色のサムネイルを自動で表示する（**リビジョン一覧表示機能**）．図 2.5 にリビジョン一覧表示機能の例を示す．この領域では，四角の枠線で囲まれたサムネイルを用いてユーザがいつ，どのような操作を行ったのかを表現する．サムネイルは左上，右上，下部の 3 つの領域に分かれている．左上にはリビジョン ID が表示され，下部にはそのリビジョンがデータベースへ保存されたときの時刻が表示される．右上には，コンパイルエラーや実行時エラーの有無が表示される．「○」の表示はプログラムが異常終了しなかったことのみを示しており，必ずしも実行時に正しい結果が出力されたとは限らない．

また，サムネイルの枠線には実線と点線があり，実線はそのリビジョンが差分表示領域に表示されていることを示している．枠線の色は Save，Compile，Run のい

ずれが実行されたのかを示しており，それぞれ黄色，青色，赤色の色に対応している．図 2.5 の例では，ユーザが順に Save，Compile，Run の動作を行い，Compile を行ったときにコンパイルエラーが生じたが，その後の修正により Run を行ったときはエラーが生じなくなったことを示している．

さらに，ユーザがサムネイルをクリックするとクリックした過去の特定のリビジョンのソースコードがエディタ領域に読み込まれる（**手戻り機能**）．これらの機能により，2.2.1 章で述べたような，過去の実行結果を知るために特定のリビジョンまで Undo 機能を繰り返しその都度コンパイルや実行を再び行う必要はなくなると考えられる．

変更内容一覧領域

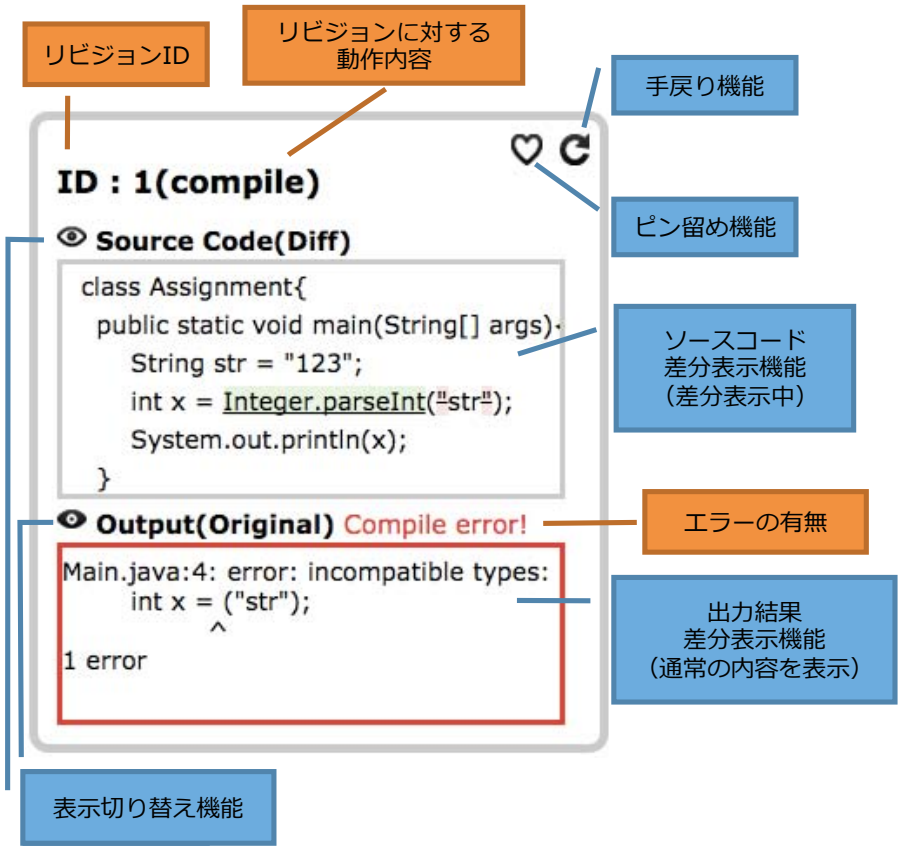


図 2.6　差分表示領域の表示例

変更内容一覧領域には 4 リビジョン分，過去と直近の，ソースコードと結果の差分を表示する領域がある（**差分表示機能**）．この各リビジョンのソースコードと出力の差分を表示する領域を差分表示領域と呼ぶ．ユーザが任意のリビジョンをリビジョン一覧領域から選択するか，ユーザの Save，Compile，Run の操作によって新たなリビジョンがデータベースへ保存されるたびに，変更内容一覧領域の右下に新たな差分表示領域が出現する．なお，差分表示領域は最大で 4 つまで表示され，古いものから順に左上，右上，左下，右下に配置されている．新しいリビジョンが右下へ追加されたとき，最も古い差分表示領域が削除され，再度整列される．

図 2.6 に差分表示領域の例を示す．この領域では，過去の特定の ID のリビジョンと最新のソースコードとの差分が表示される．特定の ID のリビジョンに対して，最新のソースコードで追加された文字は背景が緑色になり下線が引かれ，削除した文字は背景が赤色になり打ち消し線が引かれる．差分表示領域の Source code と Output という文字の左に表示されている目のアイコンをクリックすることで，元の出力にあたる Original，あるいは差分表示にあたる Diff に表示を切り替えることができる（**表示切り替え機能**）．

また，Pockets ではソースコードをコンパイルあるいは実行したときに得られる出力結果を，以下の 3 種類に分類している．

Output1 標準出力
Output2 コンパイルエラー出力
Output3 実行時エラー出力

差分出力領域に存在する元のリビジョンの出力と，最新のリビジョンの出力結果の種類が同じであった場合，初期状態として差分が表示される．差分を表示することでユーザによるソースコードへの変更とその変更に伴う出力結果の変移が明らかになり，ユーザが複数種類の実装を試行・評価する探索的プログラミングを容易に行うことが可能となると考えられる．一方，過去のリビジョンと最新のリビジョンで出力の種類が異なる場合，出力結果の差分は元の出力結果にあたる Output (Original) が表示され，目のアイコンを押しても差分は表示されなくなる．これは，出力結果の種類が違うもの同士の差分を表示して，ユーザが混乱することを防ぐためである．

各差分表示領域の右上にある矢印アイコンをクリックすることで，クリックした

過去の特定のリビジョンのソースコードがエディタ領域に読み込まれる（**手戻り機能**）．この機能により，探索的プログラミング中における過去のソースコードへの手戻りが容易となる．また，差分表示領域に表示されていないリビジョンに対しても，サムネイルをクリックすることで手戻りを行うことができる．

また，特定のリビジョンを差分表示領域に表示し続けておきたい場合は，各リビジョン表示領域右上のハートマークを選択する（**ピン留め機能**）．ピン留めされた状態のリビジョンはユーザが変更の基点にしているとし，新しいリビジョンとの入れ替えが行われない．

これらの機能及び UI により，ユーザは自身が加えたどのような変更によりどのように結果が変化したのか，エラーが生じたときと生じなかったときの違いは何かについて考えることができる．

Pockets の実装

後述する Pockets の各機能の実装以外，使用している環境は C3PV と同様である [14]．サーバの OS は Linux で，Web サーバ及びデータベースサーバはそれぞれ Apache，MySQL を使用している．実装言語及び実装フレームワークは，サーバサイドが PHP 及び ZendFramework である．

Pockets の各機能は，クライアントサイドにおいて Javascript 及び jQuery を使用して実装している．また，エディタ領域でコーディングされたソースコードのコンパイル及び実行は，C3PV と同様にオンラインコンパイラである Ideone[*4]の API を使用している．

2.3. 支援ツールの評価

初学者が探索的プログラミングを行う際に Pockets がどのように役に立つかを調査するために 2 つのケーススタディを行った．ケーススタディ 1 では，大阪府立大学工業高等専門学校における 41 名の本科 2 年生の学生を対象とした C 言語プログラミング演習授業に Pockets を演習環境として提供し，学生が探索的プログラミングを行う上で Pockets をどのように利用しているか調査した．ケーススタディ 2 では，

[*4] http://ideone.com/

奈良先端科学技術大学院大学の修士 10 名の学生を対象として，実験を行った．実験では，Pockets を使用した場合と使用していない場合において，探索的プログラミングの試行回数や課題の回答にかかった時間を比較することで，より詳細な Pockets を使用した探索的プログラミングの特徴に関して調査を行った．

2.3.1. ケーススタディ 1

ケーススタディ 1 の概要を述べる．ケーススタディ 1 は，大阪府立大学工業高等専門学校本科 2 年生 41 名が受講している，C 言語の習得を目的とした科目の第 13 回目に実施した．なお，本科目は合計 15 回の授業から構成されている．本実験の目的は，Pockets を用いて探索的プログラミングにおけるソースコードの複雑化，それによる混乱を防ぐことだけでなく，Pockets が実際の初学者向けプログラミング演習でも使用可能であるかを調査するためである．受講生らは，この回までに C 言語の基礎的な文法，条件分岐，繰り返し構文，配列について学習済みである．また，当該科目はプログラミングの導入科目であるため，独学により事前にプログラミングを学習していた学生を除いた多くの学生はプログラミング経験が半年未満である．このことから，今回対象とした学生は Pockets が対象としているプログラミング初学者に当てはまるといえる．

実験の手順としては，講義開始時に Pockets の操作方法について説明を行い，その後 45 分間で Pockets を使用して演習課題のコーディングを行ってもらった．演習課題として難易度が違う問題を 4 問与え，演習時間内で問題をできる限り解くように指示した．問題を解く順序に関しては特に指定していない．なお，これらの問題は全て，学生が既に学習済みのプログラミング知識を用いて解くことができる．

ケーススタディ 1 では，学生が Pockets の各機能をどのように使用したのかを調査するため，なお，演習中は教員 1 名，教員補助 1 名が課題の意味やエラー解消の方法が分からない学生の指導など，演習の補助を行ったが Pockets の操作を促すような指導は行っていない．Pockets により取得されたデータの分析と，実験終了後に Pockets のユーザビリティに関するアンケート調査を行った．

データの分析は，Pockets により取得されたデータのうち，2.2.1 章で述べた初学者の探索的プログラミングに当てはまる箇所を目視で抽出した．また「複数回の変更」については，より多くのデータを収集するために「2 回以上の連続した変更」で

表 2.1　グループ概要

項目	内容	
グループ	A	B
人数	5 名	5 名
使用ツール（課題 1）	Pockets	C3PV
使用ツール（課題 2）	C3PV	Pockets

ある場合，探索的プログラミングを行っているものと判断した．

今回観測された探索的プログラミングの事例は 16 件で，うち 8 件が手戻りを伴うものであった．また，手戻りを伴った探索的プログラミングのうち，6 件は Pockets の手戻り機能を使用していたことを確認した．また，手戻り機能が用いられていなかった 2 件についてどのように手戻りを行っていたのかを分析したところ，2 件とも 1 行程度の細かな手戻りであり，3 分以内に保存されたソースコードへと戻るものであった．このことから，Pockets の手戻り機能を使う必要がないと学生が判断したと考えられる．さらに，手戻り機能が利用された 6 件についても同様に分析を行ったところ，コンパイルエラーが連続した場合や本来不要な記述をソースコードに追加してしまった場合に利用されていた．また，手戻り先のソースコードの状態を確認したところ，必ずしも解答と一致した出力ではなくとも，最初に出力が成功したリビジョンや連続していたコンパイルエラーが解消されたリビジョンに戻っていることが観測された．以上より Pockets の手戻り機能は，学生による試行錯誤が行き詰まった際に，一度分かりやすい状態に戻る目的で利用されることが確認できた．これは，初学者は Pockets を使用することによって，探索的プログラミング中に行われる手戻りや細かな修正によるソースコードの複雑化を防ぐことが可能になると考えられる．

2.3.2. ケーススタディ 2

ケーススタディ 2 の概要を述べる．ケーススタディ 2 では，より詳細に Pockets 特有の機能が探索的プログラミングをどのように促進するかを調査することを目的とした．ケーススタディ 2 の被験者は 4 名が修士 1 年生，6 名が修士 2 年生である

が，全員 C 言語の授業を受けた経験は 2 年以下である．また実験後アンケートにおいて，10 名中 8 名は日常的に C 言語を使用していないと述べ，残りの 2 名についても授業外で使用することもあるが頻繁ではないと述べている．したがって，本論文では初学者を「高等教育機関における特定のプログラミング言語を対象をした学習経験が 2 年以下のもの」と定義し，ケーススタディ 2 の被験者らを C 言語に関する初学者として取り扱う．

Pockets 特有の機能が探索的プログラミングに及ぼす効果を調査するため，本実験では被験者 10 名を 5 人ずつのグループに分け，それぞれ Pockets を使用した場合と使用しなかった場合で課題を 1 問ずつ解いてもらい，そのデータを収集した．各グループの各課題における使用ツールは表 2.1 に示したとおりである．課題の内容を付録 A.1，A.2 へ示す．課題は両方とも条件分岐，Loop 文，配列のみで解くことが可能であり，使用言語は C 言語を指定した．また，実験時間の都合上，それぞれの課題の回答時間として 45 分の制限を設けた．課題 1 と課題 2 で使用ツール及びその順番を変更した理由は，ツールの慣れによる各機能の操作数に偏りが生じないようにするためである．

2.2 章で述べたように，C3PV は Pockets のベースとなったシステムであり，C3PV を用いてコーディングする際は，図 2.4 に示す「C3PV オリジナル領域」のみがブラウザに表示される．収集されるデータについては，Pockets で拡張した差分表示領域に表示されているリビジョン，手戻りを行ったリビジョン及びピン留めされたリビジョン以外のデータが C3PV では収集される．

本調査では Pockets を使用することによって得られる効果や利点について，以下の観点を基準として設けた．そして以下の項目において，Pockets の全ての機能を使用したときと，C3PV オリジナル領域のみでコーディングを行ったとき，どのような差が出るか調査を行った．

観点 1　手戻りを伴う探索的プログラミングの回数
観点 2　手戻りを伴わない探索的プログラミングの回数
観点 3　保存，コンパイル，実行，手戻り（Save，Compile，Run，Revert）の回数
観点 4　課題を解くのにかかった時間とその達成度

観点 4 について，課題 1 は被験者全員が指定した時間内に完答することができな

表 2.2　手戻りとツールの関係

	手戻りあり		手戻りなし	
	課題 1	課題 2	課題 1	課題 2
Pockets	2(2)	4(1)	4	1
C3PV	0	1	4	0

表 2.3　ケーススタディ 2 動作内訳

課題	課題 1		課題 2	
グループ	A	B	A	B
ツール	Pockets	C3PV	C3PV	Pockets
Save	33(2)	56	19	17
Compile	28	56	27	16
Run	40	46	56	58(4)
Revert	4	-	-	1
合計	105	158	102	92

表 2.4　達成度（課題 1）と解答時間（課題 2）

グループ	A					B				
被験者	A	B	C	D	E	F	G	H	I	J
課題 1 達成度（個）	5	3	4	2	4	4	5	3	3	5
課題 2 解答時間（分）	15	−	14	16	15	17	15	19	14	14

かったため，付録 A.1 のようなサブゴールを 7 つ設けた．そして，被験者らのコーディング過程から，どの程度各項目を達成しているかで達成度の評価を行った．

観点 1，観点 2 の結果を表 2.2 に，観点 3 の結果を表 2.3 に，観点 4 の結果を表 2.4 にそれぞれ示す．表 2.2 の括弧内の数値は，確認された手戻りのうち Pockets の手戻り機能を使用して手戻りを行っている回数である．表 2.3 の括弧内の数値は，Save あるいは Run により表示されたリビジョンのうち，ピン留めされたリビジョンの数

である．表 2.4 の達成度は，付録 A.1 で詳述した課題 1 におけるサブゴールを満たした個数である．また，被験者 B は時間内に課題 2 を回答することができなかったため，解答時間は評価の対象外とする．

表 2.2 より，手戻りを伴う探索的プログラミングの総回数は課題 1，2 ともに Pockets のほうが C3PV より増加したことが確認された．また，表 2.2, 2.3 の結果より，課題 2 では Pockets の手戻り機能が 4 回の探索的プログラミングのうち 1 回しか利用されていなかった．そこで手戻り機能を使用していない 3 回の手戻りについて詳細を確認したところ，3 回全てにおいて手戻り先が差分表示領域に表示されており，被験者がそれを見ながら手戻り先リビジョンへ，手入力により手戻りを行っていたことが推測される．以上より，差分表示領域によって過去のソースコードと出力結果を閲覧できるようになったことで，被験者が特定のソースコードに加えた変更や実行結果を振り返りやすくなったと考えられる．実際に，被験者によって利用された Pockets のピン留め機能の履歴を確認したところ，エラーが生じなかったリビジョンや，エラーが発生する直前のリビジョンがピン留めされており，被験者にとって分かりやすい過去の状態を把握する目的で差分表示領域が用いられていたことが分かった．以上より，今回の実験においては多くの被験者が，Pockets の差分表示領域や手戻り機能によって複数種類の実装の試行・評価が容易になり，探索的プログラミングをより積極的に行うようになったと考えられる．

2.4. ケーススタディの結果に関する考察

2.4.1. アンケートを基にした分析

本章では 2.2.1 章で述べた課題の改善を目的として Pockets を提案した．ケーススタディ 1 においては，コンパイルエラーや実行時エラーが解消されたリビジョンに戻る目的で Pockets の手戻り機能が利用されていたことが確認された．このときの手戻り先が必ずしも解答と一致するものではないことからも，探索的プログラミングにおけるソースコードの複雑化に伴う混乱を避け，学生にとって分かりやすい状態に手戻りをする手段として，Pockets が利用されていたと考えられる．ケーススタディ 2 では，Pockets の手戻り機能や差分表示機能を利用した探索的プログラミングがより積極的に行われるようになっていることが観測できた．以上より Pockets は

初学者による探索的プログラミング実施時における混乱を回避する手段として有用であるといえる．

ケーススタディ 1 において，学生らにとって Pockets の提供する機能やインターフェイスがどの程度適切であったかを定性的に評価するために，袴田ら [10] の手法に準じたアンケートを実施した．袴田らは，システムが適切な機能・インターフェイスであれば，ユーザは強制せずとも自然に利用するという仮説を立証している．アンケート項目とその結果を以下に示す．

質問 1　本日使用したツールに関して，良かったと思う点を選択してください．（選択式，複数選択可）

- **選択項目 1**　過去の Save，Compile，Run の結果がサムネイルとして一覧で見ることができること（リビジョン一覧表示機能）
- **選択項目 2**　過去に書いたソースコードに簡単に戻ることができること（手戻り機能）
- **選択項目 3**　過去のソースコードと現在のソースコードの差分を見ることができること（差分表示機能）
- **選択項目 4**　特になし
- **選択項目 5**　その他（自由記述）

質問 2　本日使用したツールに関して，使いにくかったと思う機能や改善点を教えてください．（自由記述）

質問 3　ツールを使用したことによって，普段プログラミング演習において使用しているプログラミング環境に比べ，課題は解きやすくなったと思いますか？（「思う」または「思わない」を選択）

質問 4　上記で「思う」もしくは「思わない」を選んだ理由を教えてください．（自由記述）

ケーススタディ 1 の質問 1 から 3 の結果をまとめた表を表 2.5 に示す．なお，質問 2 は自由記述であったため，分析者が各項目に分類した．ここで，分類項目 “その他” として，オンラインエディタのバグやツール全体の文字の小ささ，ウィンドウサイズを変更した際のレイアウトの崩れなどが指摘されていた．

質問 1「良かったと思う機能」において，“特になし” と回答した 3 名を除いた 35

表 2.5　ケーススタディ 1 結果（有効回答数 38 件）

質問 1：良かったと思う機能	
リビジョン一覧表示機能	26 件
手戻り機能	22 件
差分表示機能	20 件
特になし	3 件
質問 2：不便だと思った機能	
リビジョン一覧表示機能	1 件
手戻り機能	2 件
差分表示機能	10 件
その他	9 件
特になし	13 件
質問 3：普段のプログラミング環境に比べ課題は解きやすくなったと思うか	
思う	28 件
思わない	10 件

名から Pockets の機能のうち少なくとも 1 つは良かったという回答を得た．さらに，質問 3「ツールを使用したことにより普段のプログラミング環境より課題が解きやすくなったと感じるか」については，38 名中 28 名が「思う」と回答した．その理由として（質問 4),「差分表示機能により過去の実行結果が閲覧可能だったため」,「リビジョン一覧表示機能で表示されるのエラーの有無が ○ と × で簡単で分かりやすかったため」,「手戻り機能により初期状態へ容易に戻ることが可能だったため」と，Pockets 固有の機能に対する意見を多く得た．なお，質問 3 で課題が解きやすくなったと回答した 28 名に対して詳細な回答を記述してもらったところ，リビジョン一覧表示機能，手戻り機能等 Pockets 固有の機能について言及した学生は 17 名，オンラインエディタの機能について言及した学生が 7 名，両機能について言及した学生が 4 名となった．つまり，合計 21 名が Pockets 固有の機能により課題が解きやすくなったと回答している．実際に Pockets の手戻り機能を使用した事例は 6 件であったが，

22 名が手戻り機能を良かったと思うと回答している．この理由として，ケーススタディ 1 において実験前に行った事前講習が関係していると考えられる．全ての学生には事前講習において差分表示機能，リビジョン一覧表示機能，手戻り機能を使用したときの，Pockets の動作や表示内容について確認してもらっている．さらに質問 1 では「良かったと思う点を選択してください」と聞いているため，今回の実験では手戻り機能を使用していない学生でも，手戻り機能に関して良いと感じた場合投票していると考えられる．実際に，自由記述で得られた回答を確認したところ，「過去に書いたソースコードに戻れるのが間違えてしまったときに便利だなと思った」という回答が存在した．

一方で質問 2 において，38 名中 10 名が差分表示機能を不便だった機能としてあげた．不便に感じた理由について 10 名へ詳細な回答を記述してもらった結果，コンパイルエラーが生じたときに差分表示領域の出力覧が小さすぎること，エラー出力同士の差分の表示が分かりにくいためという回答が得られた．学生のプログラミング演習に対するモチベーション低下を防ぐためにも，今後，出力部分の拡大やレイアウト等の改善を図っていきたい．

また，ケーススタディ 2 でも実験後アンケートとして，ケーススタディ 1 のアンケートにおける質問 1 とプログラミングの経験年数を被験者に対して質問したところ，10 名中 8 名が差分表示機能，手戻り機能，サムネイル機能のいずれかについて使いやすいと思うと回答した．その理由として差分表示機能では「加えた変更と結果がまとまっていることで，手戻り先のリビジョンを容易に選ぶことができたから」，「前のリビジョンでなぜエラーが生じたのか，差分が表示されているのですぐ分かったため」，手戻り機能では「初期化してしまいたいときに容易に最初のリビジョンへ戻ることが可能であるため」，サムネイル一覧機能では「過去の成功したリビジョンがすぐ分かるため」という回答を得た．これらの意見からも， Pockets の差分表示機能は探索的プログラミングにおける手戻りの難しさを緩和し，手戻り失敗によるソースコードの複雑化を防ぐことができると考えられる．

これらのアンケート結果より，多くの初学者が Pockets の機能やインターフェイスを有用であると感じていること，実際に Pockets の機能を利用した探索的プログラミングが初学者によって行われていたことが分かった．以上より，不慣れな言語におけるプログラミングの課題において，Pockets が初学者による探索的プログラミ

ング，すなわち複数種類の実装を試行・評価しながら開発を進めていくことを支援する効果があることが確認できた．

一方で今回のケーススタディ 2 の結果から分かるように，Pockets の利用は必ずしも課題の達成度や解答時間の改善には繋がっていない．これは，プログラミング課題を解く際には，Pockets が支援する探索的プログラミングだけでなく，課題の内容やアルゴリズムを理解する能力，複数のライブラリやアルゴリズムを組み合わせて実際にソースコードを完成させる能力といった様々な能力が求められることが原因であると考えられる．今後，初学者のプログラミングにおける思考モデルの分析等を通じて，より広範囲にわたるプログラミング支援環境の構築を目指していきたい．

2.4.2. 関連技術・研究との比較

Scratch[*5]に代表されるビジュアルプログラミング環境は初学者が探索的にプログラミングを進めることを前提として開発されている．このような環境では．予め選択肢にあたるブロックが複数用意されている．ユーザはこれらのブロックから特定のブロックを選び，ブロック同士を繋いでいくことでプログラムが自身の意図した挙動を示すようにコーディングを行うことができる．これは Brandt ら [3] が確認した探索的プログラミングのプロセスと一致する．しかし，ビジュアルプログラミングは Java や C といった言語を対象とした初学者向けプログラミング演習には向いておらず，プログラミング未経験者がプログラミングに慣れるための教材として通常利用されている．

Myers [37] や Yoon [60] らは探索的プログラミングにおける手戻りに着目した研究を行っている．彼らは，ソフトウェア開発において開発者らはどの程度の頻度で手戻りを行っているか，また手戻りを行う理由はなぜなのかについて調査を行った．さらに，彼らは調査の結果から，既存の IDE は開発者が求めている手戻り機能を実装していないと指摘し，手戻りの種類（ソースコードの一箇所のみ手戻りを行う，あるいは複数箇所同時に手戻りを行うなど）が選択可能である Eclipse のプラグイン，AZURITE[*6]を開発している．AZURITE により，開発者は様々な種類の手戻りを

[*5] https://scratch.mit.edu/
[*6] http://www.cs.cmu.edu/∼azurite/

行うことが可能となる．AZURITE は初学者による利用を想定していないため，初学者向けプログラミング演習にそのまま適用することが可能であるかは判断できないが，手戻りの支援によって Pockets と同様探索的プログラミングが促進される可能性がある．一方で，第 2.2.1 章で述べた実態調査において，初学者は探索的プログラミングを行う際，こまめに修正を行い，その都度コンパイルや実行を行っていることが分かっている．また，初学者はエラーが生じるソースコードのまま探索を続けたり，一度エラーが生じたソースコードを再度コンパイルあるいは実行を行ったりしていたため，熟練の開発者に比べてエラーが生じる回数が多いと予測される．そのため手戻りを行う際には，ケーススタディ 1 におけるアンケート結果でも言及があったように，Pockets の機能の一部である過去のソースコードの実行結果や現在との差分提示がより有用であると思われる．

また，編集履歴を開発者に提示するものとして，Github[*7]などが存在する．しかし，Github は版管理システム（Git）を使用していることが前提であるため，初学者にとって操作することは容易ではない．さらに，これらのシステムの可視化はファイル単位であるため，探索的プログラミングのような特定の一行や変数に着目した変更履歴の可視化には適切ではない．

2.5. 探索的プログラミングの自動検出手法の提案

前章までの結果より，提案したプログラミング環境を用いることで，初学者が探索的プログラミングを行う上でのソースコードの冗長化や，手戻りにおける混乱を防ぐことが可能となった．

本章では差分情報に基づいた探索的プログラミングが行われている間のプログラミング行動（以下探索的プログラミング行動と呼ぶ）を自動検出する手法を提案する．探索的プログラミング行動を自動検出することで，教員や TA に負担を強いることなくリアルタイムに課題やプログラミングに躓いている学生を検出することが可能となる．

まず本研究では探索的プログラミングが行われているプログラムの箇所を抽出するために，既存研究 [50] [48] で述べられている探索的プログラミングの定義を初学

[*7] https://github.com/

者にも適用できるように拡張した．次に，学生が実装に躓いているプログラミングの要素やソースコードの箇所をより正確に特定するため，修正箇所の深度とブロックを定義し，それらを自動検出する手法を提案する．

本論文では，プログラミング演習の改善に向けて，ソースコードの修正履歴の中でも，探索的プログラミングが行われているプログラミングの要素に着目し，分析を行う．学生が探索的プログラミングを行っているプログラミングの要素を教員に提示することで，プログラミング演習の各単元において，特定のプログラミングの要素に対する教員からのより具体的なアドバイスや，課題の作成の支援などにも繋げることができると考える．

2.5.1. 初学者向け探索的プログラミング定義の再考察

第 2.1.3 章で述べたように，既存研究において探索的プログラミングとは，ソフトウェア開発において開発者がプログラミングに躓いた際，ソフトウェアの設計と実装をインタラクティブに繰り返すことで，ソフトウェアを徐々に完成形に近づけていく手法を指している．第 2.2.1 章で述べた実態調査の際，既存の探索的プログラミングの定義をプログラミング演習の形態に合わせ，特定のプログラミングの要素に対して，編集及びコンパイル，実行の一連のプログラミング行動が連続して複数回行われていることとして検出した．しかし，調査の結果，初学者は同一の課題でも様々な規模で探索を行うことが判明した．

例えば特定の課題において，ある学生は if 文の追加・削除を繰り返し，その都度結果がどのように変化するか確認，すなわち探索的プログラミングを行っていた．一方，他の学生は探索的プログラミングを if 文の条件式に対し行い，また別の学生は if 文の条件式の中でも比較演算子のみに対し行っていた．したがって，初学者の探索的プログラミングを支援する上では，修正がどのブロックをまたいであるいはブロックないで行われているか，また修正された文字数の大小など，修正の規模に応じる必要がある．このように修正が行われる規模を本研究では粒度と呼ぶ．

以上の結果より，本研究では初学者による探索的プログラミングの定義を「同一のブロック，行，あるいは行を構成する要素に対して修正及びコンパイル・実行結果の確認が連続で行われていること」と拡張した．本論文で述べる修正とは，文や文字の挿入，削除，入れ替えを指す．ブロックとは通常中括弧 “{}” で囲まれた部分を表す

が，ここではそのブロックを特徴づける前後の記述も含めるものとする．また，ブロックが入れ子になった場合を考慮するため，ブロックの深度を定義する．ブロックの深度とは，ブロックが入れ子になった場合，外側から数えて何番目のブロックであるかを表す．例えば，図 2.7 のソースコードの場合，各ブロックの深度は以下のようになる．

```
 1:int main(){
 2:int i;
 3:
 4:for(i=0;i<50;i++){
 5:  if(i%3==0){
 6:   printf(“bar”);
 7:  }
 8:  if(i%5==0){
 9:   printf(“foo”);
10:  }
11: }
12:}
```

図 2.7　複数のブロックの深度を含むソースコード

深度 0　main（1 行目）
深度 1　for（4 行目）
深度 2　if（5 行目），if（8 行目）

本論文では初学者向けプログラミング演習で対象となることが多い C 言語や Java を前提としている．これらの言語ではブロックを記述する際中括弧を用いるため，提案する定義によって探索的プログラミングの判定が可能である．一方，Python や Ruby のような中括弧を用いない言語においてもブロックの概念は存在する．したがって，提案した探索的プログラミングの定義をさらに拡張することで，ブロックの記述に中括弧を用いないプログラミング言語の探索的プログラミングも判定可能であると考える．

2.5.2.　探索的プログラミング行動の自動検出の手順

提案手法では，特に探索的プログラミングにおいて動作の確認が行われている，条件分岐や繰り返し処理など，プログラミングの要素に着目する．まず修正が行われた箇所を内包する，ブロックを特徴づける名前（main・for・if など）及びそれらの深度の情報を全て検出する．このブロックを特徴づける名前が，探索的プログラミングが行われている要素であるとみなす．次に，各深度において同一のブロックが修正されている場合，その深度において探索的プログラミングが行われているものと判断する．

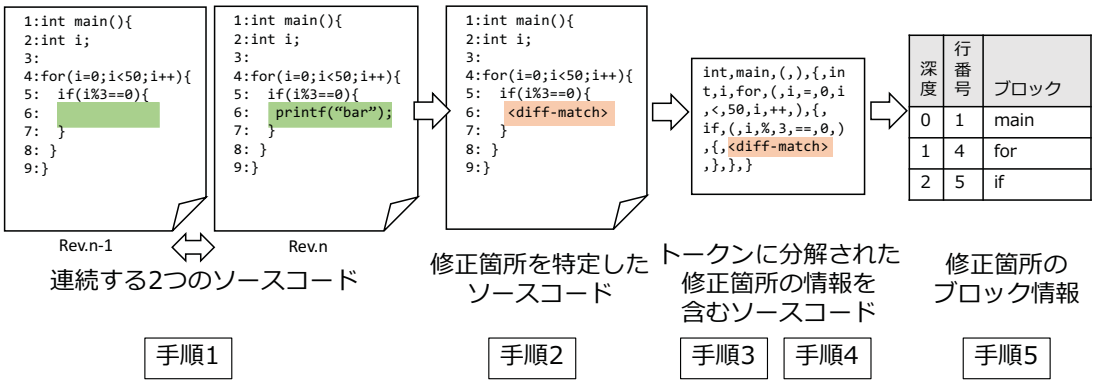


図 2.8　差分箇所の深度とブロック情報の検出

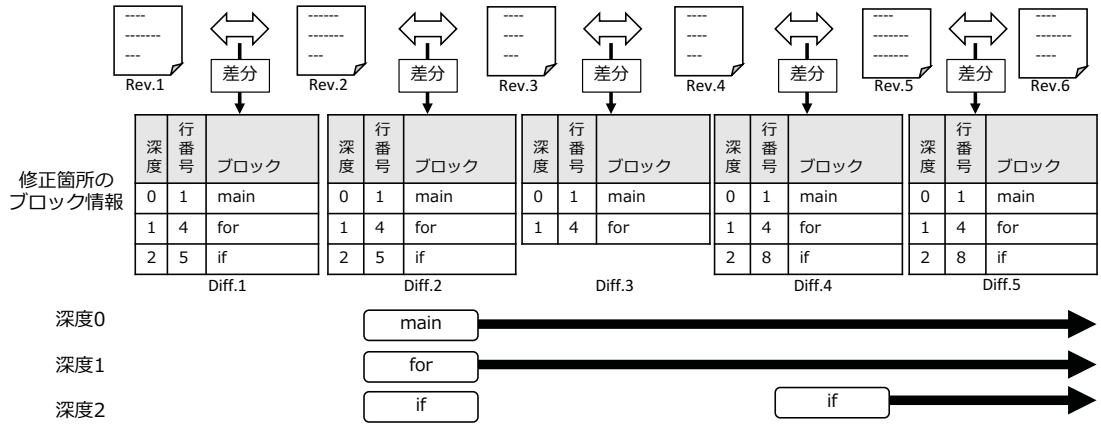


図 2.9　探索的プログラミング行動の検出

図 2.8 に修正が行われた箇所を内包する全ブロックの名前及び深度の検出の手順について示す．以下，各手順について詳述する．

手順 1　連続するソースコードの編集履歴から `Rev.n` と `Rev.n+1` を選択する

手順 2　手順 1 で選択されたソースコードの修正箇所（差分）を求める．差分の検出には Google-diff-match-patch[*8]を用いる

手順 3　手順 2 で求められた修正箇所を含むソースコードをトークンに分解し，空白を消す

手順 4　手順 2 で求めた修正箇所，すなわち探索が行われた箇所の粒度を手順 3 の結果を基に求める

粒度大　ブロック

粒度中　行

粒度小　行を構成する要素

手順 5　手順 2 で求められた修正箇所を内包する全てのブロックの深度，行番号，ブロックを特徴づける名前（以下，ブロック情報と呼ぶ）を，手順 3 の結果を基に特定する．

手順 6　手順 1 から手順 5 を次に連続するソースコードの編集履歴に対して行う

[*8] `https://code.google.com/p/google-diff-match-patch/`

以上の手順 1 から手順 6 の検出を全ての Rev.n と Rev.n+1 に対し繰り返し行うことによって，一連のソースコードの編集履歴における，修正箇所を内包する全てのブロック情報を得ることができる．

また，複数箇所が修正されていた場合，まず全ての修正箇所に対して手順 1 から手順 6 をそれぞれ行う．次に，検出された修正箇所を内包する全てのブロック情報を深度ごとに比べ，全てに共通するものをその修正におけるブロック情報として検出する．例えば，図 2.9 の 6 行目と 9 行目が同時修正されていた場合，これらの修正箇所に共通するブロック情報として，深度 0 の main ブロック及び深度 1 の for ブロックが検出される．

次に，上記の手順 1 から手順 6 によって得られた情報を基に，探索的プログラミングを検出する流れについて説明する．手順 1 から手順 6 を複数回繰り返した例を図 2.9 に示す．2.5.1 章に述べたとおり，本研究では探索的プログラミングを「同一のブロック，行，あるいは行を構成する要素に対して修正及びコンパイル・実行結果の確認が連続で行われていること」と定義している．したがって，図 2.8 の手順で検出された一連の差分情報において，ある区間に同深度で同じ名前のブロックが検出されていた場合探索的プログラミングが行われていたと判断できる．図 2.9 の場合，深度 0（main 関数内）では全修正において探索的プログラミングが行われている．深度 1 の場合，Diff.1 から Diff.5（Rev.1 から Rev.6 にかけての修正）で 4 行目から開始する for 文のブロック内で探索的プログラミングが行われている．深度 2 の場合，Diff.1 から Diff.2（Rev.1 から Rev.3）では 5 行目から始まる if 文内で，Diff.4 から Diff.5（Rev.4 から Rev.6）では 8 行目から始まる if 文内で探索的プログラミングが行われており，この場合は if 文の条件式の書き方や，if 文による条件分岐で躓いていることが考えられる．

2.5.3. 探索的プログラミング行動を自動検出した結果の一部

第 2.3 章で述べた，大阪府立大学工業高等専門学校における初学者が行う探索的プログラミングの実態調査より得られたデータを用いて，2.5.2 章で述べた手順にしたがい探索的プログラミング行動の自動検出に関する調査実験を行う．本実験の目的は，探索的プログラミング行動の自動検出によって得られるデータから，学生のプログラミングに対する躓きが確認できるか調査を行うためである．今回の実験では，

表 2.6　探索が行われた箇所の粒度と深度の検出結果（○ はコンパイル・実行時にエラーが生じなかったもの，× はエラーが生じたものを示す）

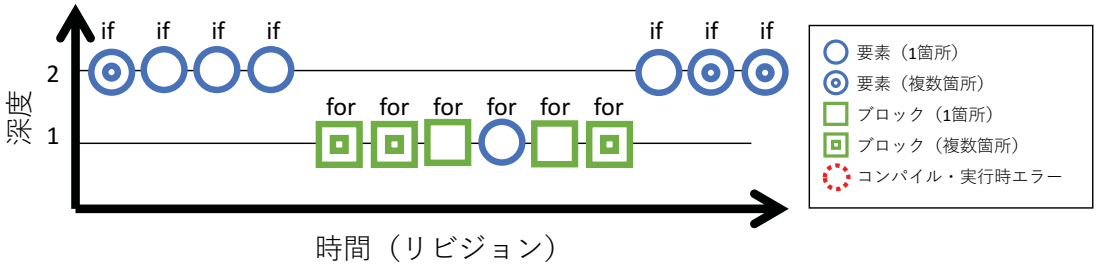
	深度ごとの検出数														
探索の粒度	深度 0		深度 1		深度 2		深度 3		深度 4		深度 5		深度 6		
エラーの有無	○	×	○	×	○	×	○	×	○	×	○	×	○	×	合計
ブロック（1 箇所）	12	18	8	5	2	3	2	1	0	6	0	0	0	0	57
ブロック（複数箇所）	17	20	10	15	4	7	2	1	0	2	0	0	0	0	78
行（1 箇所）	6	3	3	3	1	3	1	1	2	0	0	0	0	0	23
行（複数箇所）	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
要素（1 箇所）	24	20	30	22	11	14	6	9	3	4	0	0	0	0	143
要素（複数箇所）	17	16	26	13	17	5	3	4	2	1	0	0	0	1	105
合計	76	77	77	58	35	32	14	16	7	13	0	0	0	1	406

2.5.1 章に示した初学者向け探索的プログラミングの定義に当てはまった全てのコンパイル，実行時のログを対象に探索的プログラミング行動を調査した．調査対象にあたるコンパイル時のログは 359 件，実行時のログは 287 件である．実際の調査時には，学生がソースコードを保存，提出したときのログも収集していたが，本調査においてはコンパイル，実行時のログのみ使用する．ログには，学生 ID，コンパイル・実行が行われた時間，コンパイル・実行時エラーの有無，エラーメッセージ，出力結果などが含まれる．

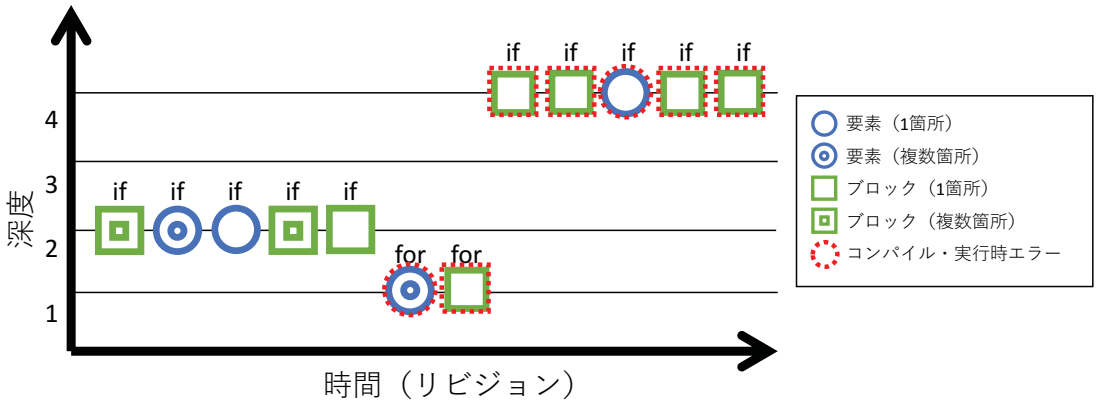
表 2.6 に，図 2.8 の手順 4 にあたる，探索が行われた粒度と深度について検出結果をまとめる．検出にあたって，図 2.8 で得られた修正箇所のブロック情報のうち，最も深度が深いものを検出対象とした．例えば，図 2.9 の `Diff.2` では深度 2，4 行目の if 文，`Diff.3` では深度 1 の for 文，`Diff.4`，`Diff.5` では深度 2，8 行目の if 文が該当する．最も深度が深いものを検出対象とすることで，修正箇所に最も近いブロック，すなわち修正箇所に最も関係があると思われるブロック情報に着目できる．

表 2.6 より，今回の実験では殆どの学生が深度 4 までのブロックで探索的プログラミングを行っていることが分かる．すなわち，深度 6 で探索を行っている学生は，ブロックを過度に入れ子にするといったアルゴリズム上の問題もしくは，文法上の誤りで入れ子が深くなっていることが考えられる．実際に深度 6 で探索している学生のソースコードを確認したところ，if 文の文法を間違えているためエラーが続いてお

り，エラーを消すために試行錯誤する過程で if 文が入れ子になっていた．また，探索が行われる粒度として要素単位の修正が 1 箇所，複数箇所共に多かった．要素単位での探索が行われたソースコードの修正内容を確認したところ，if 文や for 文の条件式部分の修正や比較演算子の修正などが存在した．



(a) 検出結果 A



(b) 検出結果 B

図 2.10　探索的プログラミング行動の可視化

次に，探索的プログラミングが行われた粒度と深度及びブロックの名前を学生の課題ごとに可視化を行った例を図 2.10 に示す．図中の丸は要素単位の探索，四角はブロック単位の探索，丸と四角の上の文字は探索的プログラミングが行われたブロック名を指す．点線で囲まれた丸や四角はコンパイル・実行時にエラーが生じたものを示す．

この課題は，与えられた配列の数値がうるう年であるか否かを判定する問題である．うるう年の判定には 3 つの条件があり，学生は for 文で配列の要素を走査し，if 文を組み合わせることで配列の要素がうるう年であるかを判定する．図 2.10(a) の学

生の場合，主に深度 2 の if 文の要素を編集しながら動作を確認しているため，if 文を入れ子にせずに論理演算子を修正しながら動作確認をしていると予測される．また，エラーも発生していない．この学生のソースコードを確認したところ，実際に論理演算子を使用して 3 つの条件式を組み合わせており，コンパイルエラーは生じないものの求めている答えと違う出力が行われていたため，条件式や if 文の中で修正を繰り返していた．一方，図 2.10(b) の学生は途中までエラーが生じていなかったものの，深度 1 の for 文の要素を修正した際にエラーが生じ，そのまま深度 4 の if 文の修正で躓いていることが分かる．さらに，この図では示していないが，図 2.10(b) の学生のソースコードには深度 3 の if 文も存在していた．以上より，この学生が if 文を過度に入れ子にしようとして行き詰まっていることが予測される．この学生のソースコードを確認したところ，論理演算子で組み合わせていた条件を 3 つの if 文に分解した際に文法エラーが生じ，その状態で修正を続けていた．

2.5.4. 探索的プログラミング行動の自動検出に関する考察

図 2.10 より，探索的プログラミングが行われている深度とコンパイル・実行時エラーがいつ生じたかの情報を組み合わせ教員に提示することで，教員はその学生がプログラミングのどのような要素で躓いているか把握が容易になる．例えば，if 文の特定の要素の探索が続いている場合，条件式の文法や条件そのものについての記述について試行錯誤していると考えられる．さらに複数箇所の要素の探索が続いている学生はデバッグを，複数箇所のブロックの探索が続いている学生はアルゴリズムそのものに試行錯誤していることが考えられる．以上のように，プログラムのどの特定の箇所の探索を行っているか，あるいは複数箇所の探索を行っているか，エラーが生じているかといった情報がブロック単位で得られることで，学生がどのような探索的プログラミングを行っているかや，行き詰まっているかを判定することが容易になる．

さらに，従来のプログラミング演習の環境では，学生が挙手により教員へアドバイスを求めたとき，教員は学生が挙手を行ったときのソースコードを基にアドバイスを与えるしか方法がなかった．しかし，ソースコードの編集履歴に着目し探索的プログラミングを可視化することで，教員は学生の試行錯誤に応じたアドバイスも可能になると考える．例えば，2.5.3 章の図 2.10 の検出結果 B の学生が挙手を行った際，従来ならば教員は学生最新のソースコードを見て，if 文の文法ミスを指摘すると

考えられる．しかし，図 2.10 を提示することで，教員は学生がエラーを引き起こす前は深度 2 の if 文，すなわち論理演算子を用いて if 文を繋げようとしていたことや，その間エラーが生じていなかったことから if 文の文法に対しては理解していることが伺える．よって，教員は単純に if 文の文法エラーに対するアドバイスを与えるだけでなく，論理演算子の使い方について触れたり，フローチャートを書かせることで条件分岐について整理させるなど，より具体的なアドバイスに繋がる．

また，提案手法では構文解析を行わずにソースコードの差分を利用して修正箇所を特定している．差分を用いることで初学者に多いセミコロンや括弧の閉じ忘れなど，文法エラーが含まれたコンパイルが成功しないソースコードでも，リアルタイムに修正箇所の検出やブロックや深度などの情報の検出が可能である．

本論文では 2 名の学生の可視化された探索的プログラミング行動のログ基に議論を行った．しかし，今回検出した探索が行われているプログラミングの要素やその深度の情報は，プログラミング演習において学生全員の統計を取ることでも，学生指導への利用が可能であると考える．例えば，模範解答で使用されるプログラミングの要素や最大深度を登録することで，使用されるはずのないプログラミングの要素や極端に深い深度での探索を行っている学生が多い場合，その旨を学生へフィードバックすることができる．2.5.3 章の課題を例に取ると，論理演算子を用いて 3 つの条件を繋げる場合は深度 2（深度 0 : main，深度 1 : for 文，深度 2 : if 文），論理演算子を用いずに if 文を入れ子にした場合は深度 4（深度 0 : main，深度 1 : for 文，深度 2 : if 文，深度 3 : if 文，深度 4 : if 文）より深い深度での探索は基本的に行われないはずである．今回の調査では深度 4 より深い深度で探索を行っていた学生は 1 人のみであったが，もしより多くの学生が教員が想定するより極端に深い深度で探索を行っていた場合，教員は詳細な個人の探索の履歴を見ることなく，クラス全体に対してアドバイスや課題の解説を行うことが可能である．

一方，図 2.10 で対象とした課題において，コンパイル・実行は行っているものの，main 関数内の探索的プログラミングばかりを行っている学生も見受けられた．これらの学生のソースコードを確認したところ，全てのブロックにおいて使用しているカウンタ変数や printf によるデバッグ文を一斉に修正していた．学生がプログラミングのどのような要素に対し躓き，探索的プログラミングを行っているのかを教員に提示するためには，探索的プログラミングの粒度や深度の情報に加え，学生が躓い

た箇所の原因や状況まで分かりやすくフィードバックするような UI を構築する必要がある．

探索的プログラミング行動の検出の他のアプローチとして，エラーが継続する時間を組み合わせたり，情報採餌理論のような動物行動学や心理学と組み合わせる手法も考えられる [22, 38, 41, 45]．しかし，これらの手法で得られた結果を教員へ伝える際，情報過多により指導すべき学生の特定や指導内容に混乱をきたすおそれがある．したがって，教員や学生へ提示する探索的プログラミング行動に関する情報の選択や UI の設計については，議論や評価を重ね慎重に検討する必要がある．

2.6. 本章のまとめ

本研究ではソースコードの編集履歴に着目した初学者が行う探索的行動，すなわちプログラミング行動について分析，及び支援を行った．

まず，探索的プログラミングの促進を目的としたプログラミング環境 Pockets を提案し，その実装を行った．本研究において実施した 2 回のケーススタディにより，初学者は Pockets のリビジョン一覧表示機能や手戻り機能を使用することで，探索的プログラミングにおけるソースコードの複雑化による混乱を回避することが可能であることを確認した．また，Pockets の差分表示機能は，探索的プログラミングにおける複数種類の実装の試行・評価を容易にすることが分かった．

次に，プログラミング演習における学生の探索的プログラミング行動を自動検出することで，学生が課題にどのように取り組んでおり，どのような箇所で行き詰まっているかをリアルタイムに特定する手法を提案した．提案手法によって得られた探索的プログラミング行動のログより，同一の課題に対する各学生の異なるアプローチが検出でき，教員は学生のアプローチに則ったアドバイスが可能であると考えられる．

また，通年を通し学生らの探索的プログラミング行動を収集・分析することで探索的プログラミング行動についてモデル化を検討することが可能になると考える．モデル化された探索的プログラミング行動のログと，演習の成績や課題の正答率のデータを組み合わせることで，正解のソースコードに近づいていくような望ましい探索を学生が行っているか否かの判断が可能となる．これらの情報は，教員が学生へアドバイスを与えに行くタイミングや，演習を中断しクラス全体へフィードバックを

行うタイミングの基準として使うことが可能であると考える．

さらに，本研究では探索的行動を基にしたプログラミング演習中における学生・教員の支援を主に行ってきたが，上述したように探索的プログラミングをモデル化することで，プログラミング演習における評価のフェーズにおいても学生や教員の支援は可能である．例えば，提案環境を用いて課題を作成，提出してもらうことで，単純に提出時のソースコードのみで評価を行うのではなく，その過程についても加味した評価を行うことができる．

本研究で提案したプログラミング行動の自動検出のアルゴリズムは，特別なツールやライブラリを使用せず，単純なソースコードの差分に着目し修正箇所を特定している．したがって，プログラミング演習以外にも e-learning や MOOCs (Massive Open Online Course) などオンラインのプログラミング環境と組み合わせることで，学生の自学自習の支援などリモート環境におけるプログラミング支援も可能になると考える．

また，通年を通し学生らの探索的プログラミング行動の粒度や深度，修正箇所の情報を収集・分析することで探索的プログラミング行動についてモデル化を検討することが可能になると考える．さらに，モデル化された探索的プログラミング行動のログと，演習の成績や課題の正答率のデータを組み合わせることで，正解のソースコードに近づいていくような望ましい探索を学生が行っているか否かの判断が可能となる．これらの情報は，教員が学生へアドバイスを与えに行くタイミングや，演習を中断しクラス全体へフィードバックを行うタイミングの基準として使うことが可能であると考える．

第 3 章

ビルドエラーに基づいた探索的行動に関する調査

前章では主に学部の初年度に行われる，初学者のプログラミング入門を目的としたプログラミング演習における探索的行動の分析・調査を行った．本章ではプログラミング演習の応用にあたる，ソフトウェア開発に関してより実践的な技術を学ぶソフトウェア開発演習における探索的行動の調査を行う．

近年のソフトウェア需要拡大・産業連携を背景に，IT 教育の分野では実践的なソフトウェア開発能力を有する人材育成が注目されている [55]．ここで述べる実践的なソフトウェア開発能力とは，プログラミングの知識だけではなく，多様な人材と交流しそれらをマネジメントする力，コミュニケーション能力，課題を発見・提案する能力などを含む．そこで，近年情報系学部・学科を有する高等教育機関において特定のアプリケーションやシステムをチームで開発するソフトウェア開発教育が重要視されている．特に，複数人の学生がチームを組みプロジェクト形式で実施されるソフトウェア開発演習 (ソフトウェア開発 PBL [9, 29]) が盛んになりつつあり，最近では従来のウォーターフォール型の開発だけでなく，アジャイル開発を取り入れたソフトウェア開発 PBL も行われている [1, 5, 24, 31, 42, 43, 46, 51]．アジャイル開発では，ウォーターフォール型開発と比較し，実装，テスト，ビルド，デプロイといった工程を短い期間で継続的に何回も繰り返すことが求められる．アジャイル開発を取り入れたソフトウェア開発 PBL においても，正常に動作するソフトウェアを継続的

に構築するため，実装やテストだけでなく，ビルドやデプロイといった工程の重要性を学生に教育することが必要となる．

ビルドとは，ソースコードのコンパイルやライブラリのリンクなどを行い，最終的な実行可能ファイルを作成する工程を指し [44, 61]，デプロイとはシステムをサーバへアップロードし，他のユーザやプログラムがシステムを利用可能な状態に展開する工程である [6]．多くの既存研究がビルドはソフトウェア開発に欠かせない工程であると述べており，ビルドの工程や結果がソフトウェア品質へもたらす影響 [28, 33, 44, 49] やビルドの支援ツールに関する様々な研究 [21, 40] が行われている．IBM や Google 社など企業でもビルド及びビルドエラーに関する分析がなされており [11, 49]，ビルドエラーが生じる原因を明らかにする取り組みが行われている．

ソフトウェア開発演習においてもビルドは重要な工程である．プログラミング演習が主に 1 つのファイルを実装することに対し，ソフトウェア開発演習では複数ファイルを Subversion や Git など版管理システムを用いて，チームで扱うことが求められることも多い．すなわち，ソフトウェア開発演習において生じたビルドエラーはチームメンバーの修正によりバグが混入した可能性があり，原因の特定は困難であることが考えられる．学生がビルドエラーの原因やついて正しく理解していない場合，新たなエラーが混入し混乱してしまうおそれがあるため，ビルドエラーの原因及びその解決策の調査は重要である．

本研究ではこのような，チーム共有のリポジトリにおいて，ビルドエラーを引き起こすようなバグが混入されることの削減を目的として，ビルドエラーの分析を行った．本研究ではソフトウェア開発 PBL を改善するために，ビルドエラーの観点から教員が支援・補填すべき点を明らかにする．

また，ソフトウェア開発 PBL は 2 章で述べたプログラミング演習と異なり，チーム開発であること，複数ファイルを扱うこと，自分以外の学生が実装したソースコードを扱うことなどが特徴としてあげられる．特に，扱うファイルが増えることや，自分以外の学生が実装したソースコードを扱うということは，チーム開発に不慣れな学生にとって，ビルドエラーが生じた際のエラーの原因を困難にするおそれがあると考えられる．

そこで，本調査ではチーム開発に不慣れな学生を支援するにあたり，特に個人ではエラーの原因の特定が困難であるリモートビルドにおけるエラーを防ぐための支援

を行う．学生のローカルビルドとリモートビルドを分析することで，リモートビルドにおけるエラーを防ぐために，各学生が個人で注意すべき点と，チームメンバーと協力して注意すべき点が明らかになると考える．

まず，2013 年度から 2016 年度にかけて，関西圏の修士 1 年生を対象としたソフトウェア開発 PBL である CloudSpiral におけるビルドエラーの分析を行った．具体的には，2013 年度から 2015 年度にかけて，CloudSpiral の演習科目として行われたウェブアプリケーション開発におけるローカル/リモートのビルドログを，以下のリサーチクエスチョンに沿って分析した．

RQ1 どのような種類のビルドエラーが存在するか
RQ2 どれぐらいの頻度で各種ビルドエラーは生じるか
RQ3 各種ビルドエラーの解決にはどれぐらいの時間がかかるか

これらのリサーチクエスチョンを通し，各種ビルドエラーが生じる原因やその解決策について調査を行った．

そして，2016 年度の同演習において，2013 年度から 2015 年度のリサーチクエスチョンの結果を基に，CloudSpiral の教員により同開発演習の改善が行われた．改善の結果，2016 年度にチームリポジトリにおいて生じたビルドエラー数は，最も多かった 2014 年度に比べ 7 分の 1 にまで減少した．さらに，エラー数が減っただけではなくエラーを解決するために要する時間も削減することができた．本研究の結果より，ビルドエラーに着目した探索的行動の収集・分析は学生のビルドエラーに関する理解を促し，正しくフィードバックを行うことでビルドエラーを削減することが可能であることが判明した．

3.1. 関連研究・用語

3.1.1. アジャイル型ソフトウェア開発

従来のソフトウェア開発において頻繁に取り入れられていたウォーターフォール型開発では，要件定義，設計，コーディング，テストなどの工程を逐次的に実施することが求められる．また，ウォーターフォールモデルにおいては，全工程は問題なく完了していることが想定されている．すなわち，前の工程に戻ることは推奨されて

いないため，コーディングやテスト段階で不具合が見つかった際に，要件定義や設計からやり直す必要が生まれ，納期の遅れや予算超過に繋がるおそれがある．

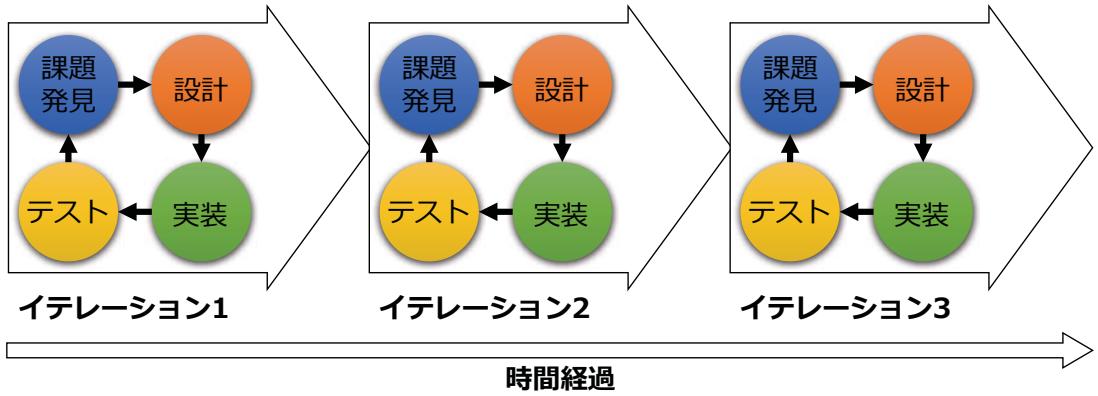


図 3.1　アジャイル型ソフトウェア開発の概要

アジャイル型ソフトウェア開発は，2001 年に Kent らによって提唱されたソフトウェア開発の総称である [13]．アジャイル型ソフトウェア開発の概要を図 3.1 に示す．アジャイル型ソフトウェア開発では，顧客の要求や開発計画の変更は予測できないものであり，変化することが前提に考えられている．したがって，開発者は厳密に開発について取り決めを行うのではなく，イテレーションと呼ばれる短い期間において，要件定義，設計，コーディング，テストを行い，イテレーションを繰り返していくことでインクリメンタルに開発を進めていく．

3.1.2.　ソフトウェア開発におけるビルド

ビルドとは，ソースコードのコンパイルやライブラリのリンクなどを行い，最終的な実行可能ファイルを作成する工程を指す．多くの既存研究がビルドはソフトウェア開発に欠かせない工程であると述べている [11, 49]．一方，Kerzazi ら [21] は，ビルドは重要な工程であるにもかかわらず，既存の研究はビルドが失敗する原因や，失敗したことによる影響の調査が不十分であると指摘している．彼らはビルドエラーが生じる原因の 1 つに，役割が違う開発者間でのコミュニケーション不足があると報告している．同様に，Phillips ら [44] も，ビルドチームの課題は技術的な要素ではなく技術者間のコミュニケーションや知識共有など，非技術的なものであると述

べている．Seo ら [49] は Google の開発履歴を基に，ビルドエラーの分類やどの種類のエラーが頻出するかなどについて調査を行った．彼らの調査の結果，必要なパッケージや変数の型名が見つからなかったときに生じる Dependency(依存関係の不整合) のエラーが最も多いことが明らかとなった．

これらの調査は Microsoft や Google, IBM など企業において行われていることが多い [11, 44, 49]．既存研究によると熟練の開発者でもビルドエラーの修正には大幅なコストがかかるとされている [21]．

以上の既存研究より，ソフトウェア開発プロジェクトにおいてビルドは重要な工程であり，ビルド時に発生するエラーには技術的な要素だけでなく，チームメンバー間のコミュニケーションや知識共有といった人的な要素にも起因することが分かっている．そのため，学生がどのようなビルドエラーを起こしやすいのか，またその原因や解決にどれだけのエフォートを必要とするのかを明らかにすることは，ソフトウェア開発教育において非常に重要な課題である．

3.1.3. ソフトウェア開発 PBL

Project-Based Learning(以下 PBL と呼ぶ) とは実際のプロジェクトを通して学生の課題発見・課題解決を実現する科目であり，近年教育機関において注目されている授業形態の一つである [55]．PBL を通して参加者は，参加者同士のコミュニケーションだけではなく，自分で課題を発見する力，学生間の能力の差を解決するための力などが養われると考えられる [1]．

実際に教育機関において開講されているソフトウェア開発 PBL として CloudSpiral があげられる．CloudSpiral[*1] とは分野・地域を超えた実践的情報教育協同ネットワーク (Education Network for Practical Information Technologies，略称 enPiT)[*2] の取り組みの一つとして，大阪大学と神戸大学を中心に 2013 年度から行われている教育プログラムである [39]．主な対象は関西の大学院に所属する修士 1 年生であり，学生はそれぞれ 5，6 人のチームに分けられ，1 年間の座学・演習と複数回のソフトウェア開発 PBL を通してアジャイル開発手法やクラウドコンピューティ

[*1] http://cloud-spiral.enpit.jp/
[*2] http://www.enpit.jp/

ングの基礎から応用を学習する．具体的な座学・演習の内容として，プロジェクトを円滑に進めるためのファシリテーションスキルや，LEGO を用いた Scrum・チケット駆動開発演習などがある．さらに，座学・演習の内容をふまえ，1〜3 週間程度の期間でアジャイル開発手法に基づいて，チームでソフトウェアを開発するソフトウェア開発 PBL が複数回実施されている．

他にもアジャイル開発を取り入れたソフトウェア開発 PBL は様々な教育機関で実施されている．国内では，産業技術大学院大学[*3]が革新的な教育の枠組み[*4]として，ビジネスアプリケーションなどソフトウェア・システム開発において PBL を利用している．九州大学[*5]でも，修士を対象に情報通信技術に関する高い実践力を持つ学生の育成を試みている．

3.2. 調査対象の教育プロジェクトについて

本研究では 3.1.3 章で説明した CloudSpiral の年間カリキュラムのうち，アジャイル開発手法に基づいてチームでウェブアプリケーション開発を主に行う，クラウド基礎 PBL 科目におけるビルドエラーに着目し，調査する．学生はクラウド基礎 PBL が行われる前に，開発に必要なプログラミングの基礎技術，設計書や仕様書の読み方，開発環境や使用ツールの操作方法，レビューやテストの手法について一通り受講済みである．以下，クラウド基礎 PBL について詳述する．

クラウド基礎 PBL は 8 月に 5 日間の短期集中合宿として実施されるソフトウェア開発 PBL である．なお，開発を主に行うのは合宿最終日を除く 4 日間 (それぞれ Sprint1 から 4 と呼称する) で，最終日は成果物や開発における工夫点の報告会が行われる．また，合宿が行われる前に，開発の流れを理解するための練習日 (preSprint に対応) が用意されており，学生は preSprint から Sprint4 までの期間，与えられた仕様書と開発環境を利用して，Java，JavaScript，mongodb といった技術を利用してウェブアプリケーションを開発する．開発対象はチケット販売システムと呼ばれるウェブアプリケーションであり，ログイン，アカウント作成，チケット購入といっ

[*3] http://aiit.ac.jp/
[*4] http://aiit.ac.jp/education/
[*5] http://www.qito.kyushu-u.ac.jp/

た 13 種類の機能について仕様書が与えられる．仕様書が共通なため，各チームによって開発されるものも共通であるが，チケット販売システムのどの機能をどういった順序で開発するか，開発中のメンバーの作業分担といった開発プロセスの詳細については学生らに任されている．なお，本調査対象のプロジェクト終了時点における平均行数は約 6000 行，ファイル数は約 100 ファイルに及ぶ．

また，各機能においてユーザの入力を検証するための仕組みとして Bean Validation と呼ばれるバリデータを用いる．バリデータの作成において，以下の枠で囲んだソースコードのように Java 言語の Annotation 機能を利用して行う．Annotation については，各機能の全テストにおいても利用される．

```
10: @Length(min=4, max=12,
    message="パスワードには 4 から 12 文字の英数字のみが利用できます")
11: @Pattern(regexp="^[a-zA-Z0-9]*$",
    message="パスワードには 4 から 12 文字の英数字のみが利用できます")
12: private String pass;
```

本研究ではこのウェブアプリケーションを開発する際のビルドログを分析する．対象期間は 2013 年から 2016 年の 4 年間である．各年度の概要を表 3.1 に，開発の流れを図 3.2 に示す．なお，2013 年度から 2016 年度のソフトウェア開発 PBL は，学生間・学生教員間のコミュニケーションツールなどは変更されたものの，開発に直接関わるツールやシステム，また演習時間等は大きく変更されなかったため，以降各年度で行われたプロジェクトは同一のものとして扱う．

以下，開発環境及び開発の流れ，そして今回調査したログ種別について説明する．

表 3.1　チーム概要

開講年度 (年)	2013	2014	2015	2016
参加人数 (人)	49	54	46	54
チーム数	9	9	8	9
ログ収集期間	7 月 26 日	7 月 25 日	7 月 24 日	7 月 29 日
	～8 月 22 日	～8 月 21 日	～8 月 20 日	8 月 18 日

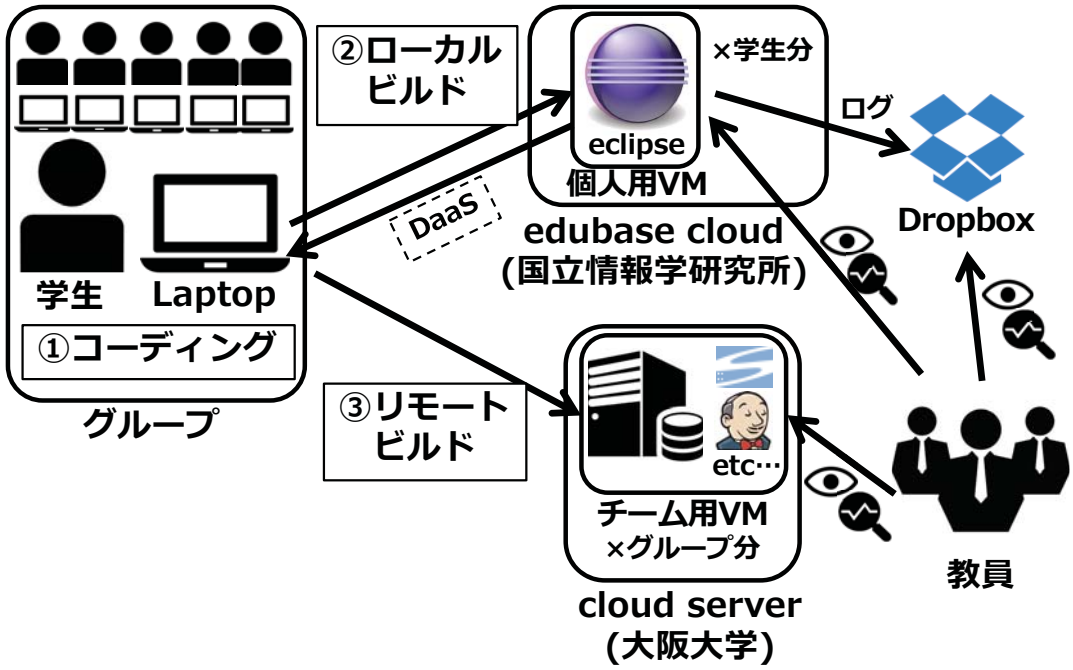


図 3.2　CloudSpiral における開発の流れ

3.2.1.　開発環境

各学生は国立情報学研究所で運用されているサーバ上の個人用 Virtual Machine(以下 VM と呼ぶ) と大阪大学のサーバ上で運用されているチーム用 VM を利用し，ウェブアプリケーションの開発を行う．個人用 VM には Windows がインストールされており，開発環境として JDK，Eclipse(ビルドツールの Apache Ant 含む各種プラグイン導入済み)[*6]，Apache Tomcat(アプリケーションサーバ) などが教員によって導入されている．学生は一人一つの個人用 VM にアクセスし，開発を行うことができる．

チーム用 VM には，Linux の Distribution の一つである CentOS がインストー

[*6]https://eclipse.org/

ルされており，版管理システムである Subversion[*7](以下 SVN と呼ぶ) や Ant[*8]，Tomcat[*9]，サーバでのビルドを支援する Jenkins[*10]など，開発に必要な様々なツールが事前に教員によって準備されている．学生はチームで一つのチーム用 VM を利用し，ソースコードの共有や完成したウェブアプリケーションの動作確認等を行うことができる．

これらの開発環境を学生がどのような手順で利用するかを以下に述べる．

手順 1：環境へのログイン及び実装

各学生は個人用 VM へとログインする．個人用 VM で Eclipse を起動し，仕様書及びチームで決定したタスク分担にしたがい，各種ソースコード (テストコードや設定ファイル等含む) の実装を行う．

手順 2：ローカルビルド及び SVN へのコミット

チームでソフトウェア開発を行う場合，実社会におけるチーム開発では版管理システムでソースコードを共有することが多い．ここで，版管理システムの一つである Subversion を用いてソースコードを共有する流れを図 3.3 に示す．

[*7] https://subversion.apache.org/
[*8] http://ant.apache.org/
[*9] http://tomcat.apache.org/
[*10] https://jenkins.io/

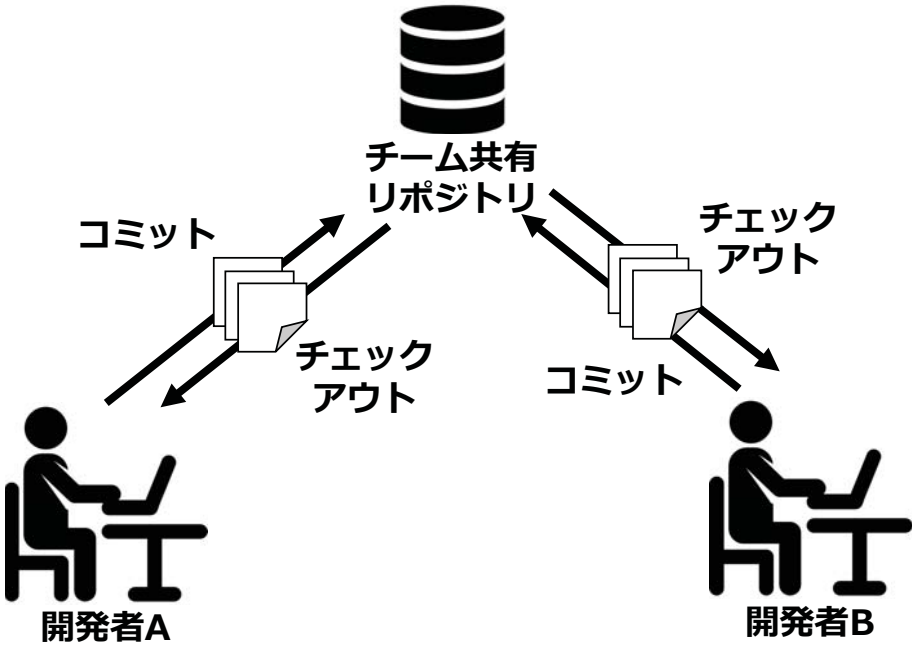


図 3.3　Subversion を用いたソースコード共有の流れ

チームメンバーはそれぞれ個人の開発環境と，チームメンバーでソースコードを共有するための共通リポジトリを持つ．各メンバーはチェックアウトを行うことで共有リポジトリに登録されているソースコードの最新版を個人の開発環境へダウンロードすることができる．また，ダウンロードしたソースコードへ何らかの変更を加えた場合は，コミットを行うことでチーム共有リポジトリへ反映することが可能である．版管理システムを使用することで，チームメンバー間で最新版のソースコードを共有しやすくなるだけではなく，誰がどの編集を行ったかのようなレビューを行いやすくするという利点がある．

本 PBL においても学生が手順 1 における実装終了後に他のメンバーとソースコード等を共有するため，版管理システム Subversion(以下 SVN と記述する) へのコミットを行う．このとき，もし学生が実装した部分に誤りがあった場合，誤りを含んだソースコードが SVN リポジトリにコミットされてしまう．本 PBL では，各学生が SVN リポジトリのチェックアウトを行うタイミングや回数は特に制限を設けておらず，いずれかのチームメンバーが SVN リポジトリへ対しコミットを行ったら，その都度チェックアウトを行い自身のローカルの環境を最新版へ更新するように指導し

ていた．したがって，SVN リポジトリへエラーを含んだソースコードをコミットすることは，結果として他のチームメンバーへ，自身がソースコードへ加えたエラーを拡散させてしまうことに繋がる．このような現象を本論文ではエラーの伝播と定義する．

エラーの伝播を起こさないためにも，本 PBL では実装が完了したことを確認するために，コミットを行う前に個人用 VM においてビルドを行うことを推奨し，事前学習においてその重要性についての教育を行っている．ここでは，個人用 VM において，学生自身が Apache Ant と呼ばれるツールを用いて実施するビルド作業のことをローカルビルドと呼ぶ．本 PBL におけるローカルビルドでは，コンパイル・テスト・コンパイル済みファイルやライブラリのリンク及びアーカイブ・アプリケーションサーバへの配備，といった一連のプロセスが実施される．そのため，理想的な流れとしては，手順 1 で実装を行った後にローカルビルドを行い，ビルドが正常に実行されることやデプロイによってウェブアプリケーションが正常に動作することを各学生が各自の環境で確認した後に，SVN にコミットすることになる．ここでもしローカルビルドが失敗した場合，再度手順 1 へ戻り，誤りの修正を行った後に SVN へコミットする，

手順 3：リモートビルド

手順 2 が完了し，SVN にコミットが行われると，実装の完了した正常に動作するソースコードがチーム内で共有されることになる．コミットされたソースコードに問題がない場合，その後のタスク分担にしたがい，他の学生が手順 1 から開発を継続することになる．一方で，手順 2 でローカルビルドを怠ったままコミットが行われていたり，ローカルビルドが正しく実施されていても，他の学生によるコミットとの兼ね合いで，誤りを含んだソースコードが SVN リポジトリにコミットされてしまうことがある．そこでアジャイル開発を取り入れたソフトウェア開発プロジェクトの多くでは，本 PBL でも利用する Jenkins のようなツールを利用し，版管理システムにコミットされた内容を取得し，ビルドが正常に実行できるかを検証する仕組みを導入している．本 PBL においても，学生によるコミットが行われると，チーム用 VM に導入された Jenkins が自動的に SVN にアクセスし，コミットされた内容を取得し，ビルドを行い，結果を記録する．このように，学生のリポジトリへのコミット

に連動して，Jenkins がチーム用 VM 上で実施するビルド作業のことをリモートビルドと呼ぶ．なお，リモートビルドとして行われるプロセスの内容はローカルビルドと同様である．学生は Jenkins によるリモートビルドの結果を確認し，成功したら次のタスクへとりかかり，失敗したら手順 1 へ戻りエラーの修正に注力することになる．

3.2.2. 開発ルール

一般に，ソフトウェア開発プロジェクトを進めるにあたって，開発チームは様々なルールを守らなければならない．例えば，原則としてチームによって開発された全てのソースコードは適切にレビューされなければならない，といったルールが実際に守られている．クラウド基礎 PBL においても，一般のソフトウェア開発プロジェクトに倣い，複数のルールが学生らのチームに提示されている．以下に教員側から定めたルールについて本論文に関係のあるものを抜粋して提示する．

- Sprint1 から 4 の開発可能時間は原則 10 時から 17 時のみで，それ以外の時間に個人用 VM での実装やレビューといった開発作業を行ってはならない (ただし，振り返りや次 Sprint の準備は認められている)．ただし，準備期間である preSprint の開発時間は指定された日の 13 時から 18 時まで及び Sprint1 が開始される直前までとする．
- SVN リポジトリにコミットを行う際，ビルドが失敗せず，原則としてその時点で存在する全てのテストケースが正常に動作していることを確認しなければならない．
- チームメンバー間で，ソースコード及びテストコードの実装回数，レビューの回数がなるべく均等になるようにタスクを分配すること．具体的には，各メンバーのファイル単位での実装・レビュー回数がチームの平均の 0.75 倍〜1.25 倍以内に収まるようにタスクを分配すること．このルールはソフトウェア開発 PBL でありがちな，特定の学生のみに実装・テスト等の負荷が集中し，経験の偏りが生じるのを防ぐことを目的としている [14].

3.2.3. 調査対象ログ

本論文で調査する対象は，ローカルビルド，リモートビルド及び SVN リポジトリの各ログである．ローカルビルドは学生の個人用 VM で Apache Ant を利用して行われる．そのため，ローカルビルドは Ant によりビルド定義ファイルが読み込まれてから，ビルドが終了するまでを 1 回のローカルビルドとし，その間に記録されるビルド開始・終了時刻，ビルド成功/失敗，ビルド失敗時のエラーメッセージ，ローカルビルド実施学生，ビルド実施内容詳細をまとめて，そのローカルビルドのビルドログとして，特にビルド失敗時のエラーメッセージをビルドエラーとして調査した．

リモートビルドは学生がチーム用 VM の SVN にコミットを行った際に，Jenkins が Ant を利用して自動的に行うものである．そのため学生がコミットを行い，Jenkins が連動して SVN から更新内容を取得し始めたときから，リモートビルドが終了して成功/失敗が Jenkins によって表示されるまでを 1 回のリモートビルドとする．リモートビルドのビルドログには，ローカルビルドと同様のビルド開始・終了時刻，ビルド成功/失敗，ビルド失敗時のエラーメッセージ，ビルド実施内容詳細と SVN リポジトリに対して行われたどのコミットに連動してリモートビルドが実施されたかといった情報が含まれる．特に，ビルド失敗時のエラーメッセージをリモートビルドエラーとして扱う．

SVN リポジトリには学生の全てのコミットが記録されており，誰がいつどのようなコミットを行ったかを確認することができる．本論文では，ローカル/リモートビルドにおいて失敗が記録されたとき，ビルドエラーが発生したものとする．リモートビルドにおいて，どのようなコミットが行われたときにビルドが失敗し，エラーが起きたのかを調査する．

以降ではこれらのログを利用し，何を調査するのかについてリサーチクエスチョンとして示す．

3.3. リサーチクエスチョン

本研究の目的は，ソフトウェア開発 PBL における学生のビルドの特徴及び難所を明らかにすることである．本研究で得られた結果は，学生がソフトウェア開発 PBL

のようなチーム開発を行う際，教員が何を留意して指導を行うべきかの指針として活用できると考える．

本研究の目的を達成するために，我々は以下のリサーチクエスチョンを設定する．

RQ1　どのような種類のビルドエラーが存在するか
RQ2　どれぐらいの頻度で各種ビルドエラーは生じるか
RQ3　各種ビルドエラーの解決にはどれぐらいの時間がかかるか

以下の章で各リサーチクエスチョンについて詳述する．

3.3.1. RQ1: どのような種類のビルドエラーが存在するか

まず，学生が陥りやすいビルドエラーの原因を調べるために，ビルドエラーの分類分けを行う．このリサーチクエスチョンでは既存研究 [49] で行われたビルドエラーの分類を参考に，我々が収集した全てのビルドエラーの分類分けを行う．

以下に Seo ら [49] の作成したビルドエラーの分類を示す．

C1: Dependency

特定のソースコード間における，変数やクラス，メソッド，ライブラリなどの依存関係に起因するエラーを Dependency として分類する．具体的には，クラス間やファイル間において必要なシンボルやパッケージが存在しなかったときに生じるエラーを指す．例として “シンボルを見つけられません (`cannot find symbol`)” というエラーメッセージがあげられる．これは変数名やメソッド名，クラス名など，本来別ファイルや別箇所で宣言されているべき要素が見つからなかったときに生じるエラーである．

C2: Syntax

文法ミスが原因のエラーを Syntax として分類する．例として中括弧の閉じ忘れの際に出力される “式の開始が不正です (`illegal start of expression`)” や，セミコロン抜けによる “‘;’ がありません (`’;’ expected`)” というエラーメッセージがあげられる．

C3: Type Mismatch

変数や引数の型の違いによって生じるエラーメッセージを Type Mismatch として分類する．例として違う型の変数を比較した際に生じる “型 xxx と型 yyy は比較できません (`incomparable types: xxx and yyy`)” や，異なる型の変数が引数で与えられたときの “xxx は yyy に適用できません (`xxx cannot be applied to yyy`)” などがあげられる．

C4: Semantic

文法は正しいがビルド時にコンパイルエラーが生じるソースコードのエラー文を Semantic として分類する．例えば，アクセス権のない変数・クラスの使用など修飾子に関するエラーや，実行されない文がソースコード中に含まれていることで生じるエラーが含まれる．エラーメッセージの例として “xxx はパッケージ外からはアクセスできません (`xxx is not public; cannot be accessed from outside package`)” や，“この文に制御が移ることはありません (`Statement not reached`)” などがあげられる．

以上が Seo らの論文に沿った分類である．

3.3.2. RQ2: どれぐらいの頻度で各種ビルドエラーは生じるか

RQ2 では，まず RQ1 で分類したローカル/リモートビルドエラーの分類ごとのエラー数を求める．調査対象のプロジェクトでは，学生は予め教員から，リモートでエラーを発生させないようにローカルでビルド結果の確認を済ませてからコミット (リモートビルド) を行うこと，と指導されていた．よって，どのようなエラーが頻出するかだけでなく，ローカルとリモートで分けてビルド発生の傾向を調査することで，ローカルで解決可能・不可能なエラーも調査できると考える．

次にチーム内における各学生のエラー数及びエラー種類内訳，エラー率を求める．チームごとに頻出するエラーや，エラー率の高いチームのエラーの種類を調査することで，教員が優先的に指導すべきエラーが明らかになると考える．

```
 1: 1440051303:Buildfile: C:Users\userID\...\path_of_build.xml
 2: 1440051304:clean:
 3: 1440051304:   [delete] Deleting directory:\Users\userID\...\dest
...(中略)...
20: 1440051307:    [javac]
C:\Users\userID\...\BuyTicketControllerTest.java:76: エラー: 式の開始
が不正です
21: 1440051307:    [javac]              query.put("totalSeats", );
22: 1440051307:    [javac]                                       ^
23: 1440051307:    [javac] エラー1個
24: 1440051307:BUILD FAILED
25: 1440051307:C:\Users\userID\...\build.xml:55: Compile failed; see
the compiler error output for details.
26: 1440051307:Total time: 4 seconds
```

図 3.4　ビルドログの例

図 3.4 にローカルビルドによって生じたビルドエラーのログの例を示す．ログの各行は UNIX 時間による実行時間と実行内容が記述されている．3.2.3 章で述べたように，1 つのビルドログには，1 行目に示すビルド定義ファイル（build.xml）が呼び込まれてから，24 行目に示すビルドの成功/失敗の結果及びビルドにかかった時間全てが表示されるまで記録されている．ビルドエラーを分析するにあたり，20 行目に記述されているエラーメッセージを基に分類する．

なお，複数種類のエラーが 1 回のビルドで発生した場合の分類は Seo らの分類方法に準拠する [49]．例えば “型の開始が不正です (Syntax)” が 10 件，“シンボルが見つかりません (TypeMismatch)” が 3 件，“クラス名が重複しています (Semantic)” が 1 件生じていた場合，それぞれのエラー数を 1 増加する．この理由として Seo らは，例えば特定のファイルのインポートを忘れた場合，類似したエラーが大量に生成され，特定のエラーの種類に不正な重みが追加されることを防ぐためと述べている．

なお，図 3.4 はローカルビルドにおけるビルドエラーだが，3.2.3 章に述べたとおり，リモートビルドにおいては学生が SVN にコミットを行った際，Jenkins が連動して SVN から更新内容を取得し始めたときからビルドの結果が全て表示されるまでが 1 つのビルドログとして保存されている．

3.3.3. RQ3: 各種ビルドエラーの解決にはどれぐらいの時間がかかるか

RQ3 では RQ1 のエラー種類ごとのエラーの解決時間を求める．エラー解決時間の定義は Seo らの定めたものを参考にして，エラーが生じたビルドのビルド終了時間から，次にエラーが解決した (エラーが生じなくなった) ビルドが開始した時間までとする．解決に長く時間がかかるエラーを調べることで，RQ2 同様教員が優先して指導したり，また講義資料等で補足すべきエラーの種類が明らかになると考える．

3.4. 調査結果

本章では 3.3 章で述べた各リサーチクエスチョンに対する結果と考察を述べる．なお，先述したとおり本調査は 2013 年度から 2016 年度の CloudSpiral におけるソフトウェア開発 PBL のビルドログを対象に行った．具体的には，まず 2013 年度から 2015 年度のビルドログを各リサーチクエスチョンに沿って分析した．そして，分析結果を基に CloudSpiral の教員によって 2016 年度の同ソフトウェア開発 PBL の改善が行われた．さらに，2016 年度の同ソフトウェア開発演習においても同様に各リサーチクエスチョンに沿った分析を行い，分析の結果，リモートビルドエラー数の削減やエラー解決時間の短縮に繋がった．

本章では 2013 年度から 2015 年度のビルド活動に対するリサーチクエスチョンの結果について主に取り扱う．図表には 2016 年度の結果もまとめているが，具体的な改善の結果については次章で扱うものとする．

3.4.1. RQ1: どのような種類のビルドエラーが存在するか

表 3.2　調査対象データの概要

	LOCAL				REMOTE			
開講年度	2013	2014	2015	2016	2013	2014	2015	2016
成功ビルド数	2395	1506	1670	2655	2309	1610	1764	2065
失敗ビルド数	340	142	297	331	149	219	94	30
合計	2735	1648	1967	2986	2458	1829	1858	2095

調査対象である各年度のビルド数の総計とビルド成功・失敗回数の内訳を表 3.2 に示す．表 3.2 のエラーについて，エラーの分類を行った結果を表 3.3 へ示す．

表 3.3 ビルドエラーの分類

	LOCAL				REMOTE			
エラー分類	2013	2014	2015	2016	2013	2014	2015	2016
C1: Dependency	34	14	18	34	86	118	64	18
C2: Syntax	12	6	12	7	0	10	0	0
C3: Type Mismatch	2	3	1	16	28	27	18	2
C4: Semantic	6	16	2	10	19	5	10	2
C5: Annotation	6	22	11	15	16	59	2	8
C6: Environment	280	81	253	249	0	0	0	0
C1〜C5 の合計	60	61	44	82	149	219	94	30

RQ1 の調査の結果，我々は新たにエラー分類の項目へ Annotation と Environment を加えた．Annotation とは Java プログラムのクラスやメソッド，パッケージへ対し付加情報を記入する機能を指す．Annotation はメソッドの直前に`@xxx` の形で書かれ，例えば直後に記述するメソッドがスーパークラスのメソッドをオーバーライドしていることを示す`@Override` や，テストメソッドであることを示す`@Test` などが存在する．

Seo らは Override の Annotation に関するエラーを Semantic として分類していた．しかし，3.2.1 章で述べたとおり，本研究で対象とした PBL では，多くの Annotation に関するエラーが生じていた．そこで，Annotation に関するエラーの特徴を調査するため，従来の分類である Semantic から分離させ，新たに Annotation の分類を設定した．なお，本研究における Annotation 及び Semantic のエラー数の合計値が，Seo らの Senmantic エラー数に対応しているため，対比させることが可能である．

また，Environment はチームで開発しているプロジェクト外が原因で発生するビルドエラーを示している．今回行ったビルドは，ソースコードのコンパイルだけでなく，テストの実行やテスト結果の表示，デプロイといった複数の処理を含んでいる．そのため，それらの処理を実行する際に必要な Ant 等の外部ツールが正常に実行しないことによるビルドの失敗が特にローカルビルドにおいて多数発生した．

そこで，Tomcat や Ant など開発ツールが原因で生じたビルドエラーをまとめて Environment と分類し，以下のとおり定義を行った．

C5: Annotation

今回の調査対象であるプロジェクトは Bean Validation[*11]を一部利用しており，誤った型の Annotation を使用しようとした際に生じるエラーを Annotation として分類する．エラーメッセージの例として，“`The annotation @NotEmpty is disallowed for this data type`” をあげる．これは，`@NotEmpy` は String 型にしか使用できないが，Date 型の変数に対し宣言しようとした際に生じるエラーメッセージである．

C6: Environment

調査対象のプロジェクトで使用しているツールに起因するエラーは全て Environment として分類した．例えば，Tomcat を起動せずに Eclipse でビルドを行ったときに生じる “`java.net.ConnectException: Connection refused`” や，データベースエンジンの不具合によるエラー，テスト結果ファイルの出力に失敗した等のエラーが含まれる．

本プロジェクトでは開発環境はほぼ教員が用意している．また，本論文の目的はソフトウェア開発 PBL におけるビルドエラーの特徴や学生にとっての難所を探し教育へ活かすことであり，開発環境に由来する Environment のビルドエラーは今回の我々の調査目的には当てはまらないため以降結果から省くこととする．

3.4.2. RQ2: どれぐらいの頻度で各種ビルドエラーは生じるか

本章ではまず得られた結果の概要図・表について説明する．次にエラー各種の知見について述べる．

[*11] http://beanvalidation.org/

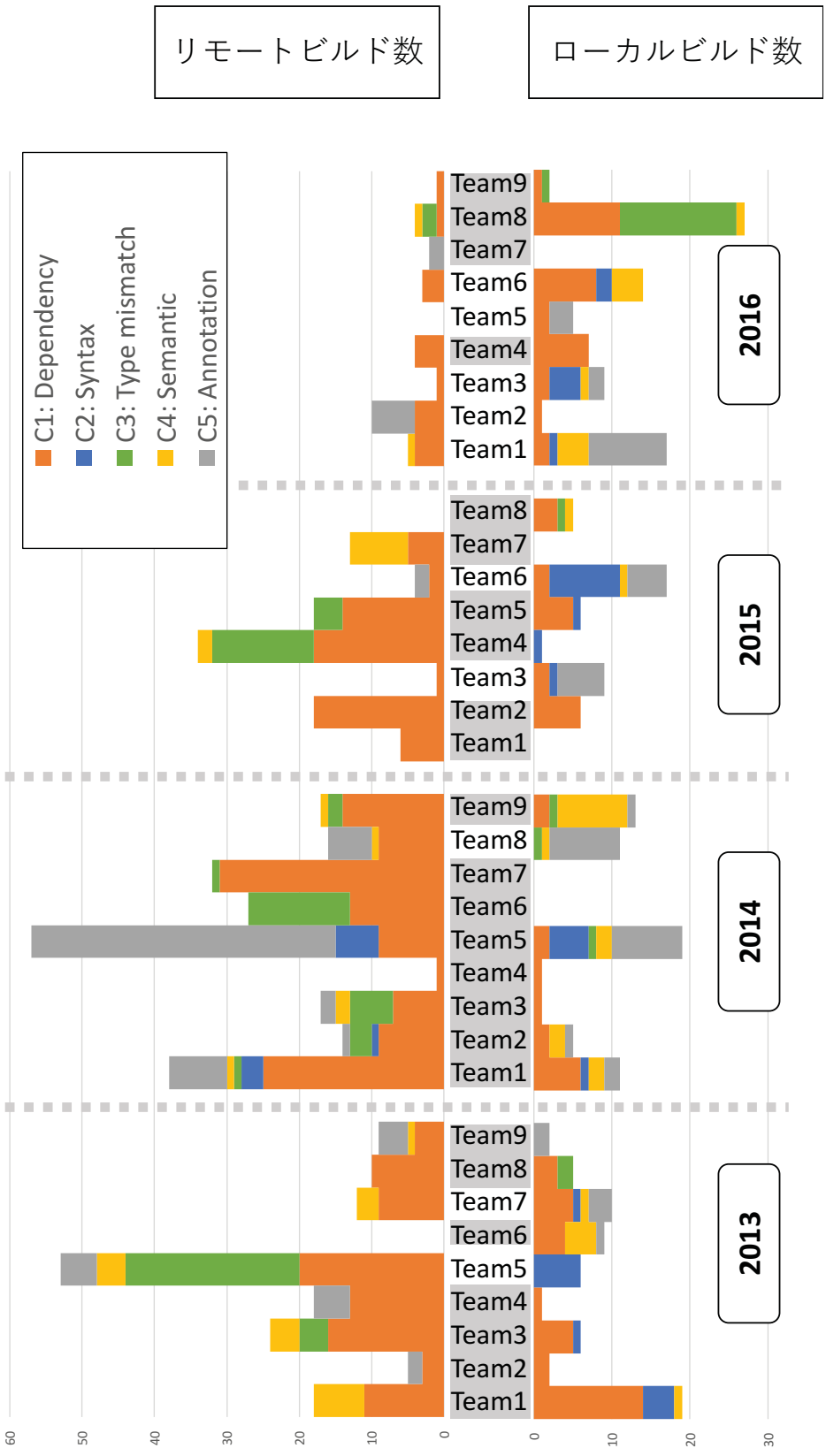


図 3.5　年度別チームごとのビルドエラー内訳 (グラフ上部：リモートビルドのエラー内訳 グラフ下部：ローカルビルドのエラー内訳)

表 3.4　ローカルビルドのほうが多い学生とリモートビルドのほうが多い学生の比較

		LOCAL		REMOTE	
	エラー分類	エラー数	E/T(%)	エラー数	E/T(%)
ローカルビルドの回数が多い人数 (56 人)	C1: Dependency	44	0.77	79	1.29
	C2: Syntax	22	0.38	6	0.10
	C3: Type Mismatch	4	0.07	13	0.21
	C4: Semantic	21	0.37	10	0.16
	C5: Annotation	30	0.52	36	0.59
リモートビルドの回数が多い人数 (92 人)	C1: Dependency	22	0.38	189	3.08
	C2: Syntax	8	0.14	4	0.07
	C3: Type Mismatch	2	0.03	60	0.97
	C4: Semantic	3	0.05	24	0.39
	C5: Annotation	9	0.16	41	0.67

エラー分類ごとの件数は先述した表 3.3 のとおりである．チーム内における各学生のビルドエラーの分類のグラフを図 3.5 に示す．横軸は年度ごとのチーム ID に対応し，各年度は灰色の破線で区切られている．エラーの有無に関係なく，ローカルビルドよりリモートビルドのビルド回数が多かった学生が半数以上いたチーム ID の背景色は灰色で塗りつぶされている．すなわち，ID が灰色の背景のチームは，リモートビルドを行う前にローカルでビルドエラーが生じないか確認することを怠っていた学生が多い可能性が高いことを示す．

また，2013 年度から 2015 年度にかけて，ローカルビルドよりリモートビルドが少ない・多い学生のエラー内訳について，3 年分合計したものを表 3.4 に詳細を示す．なお，ローカル/リモート両方のビルドエラー数が同じ学生は，ローカルのほうがビルド数が多い学生群へ分類し，0 の学生は表 4 へ含めていない．

また，ビルド数を比較するにあたって，ローカルビルドよりリモートビルドのほうが多い学生と少ない学生で人数が異なったため，エラー回数のみで比較することは困難であると考えた．そこで正規化のため，エラー分類ごとのエラー数を，ローカルはローカルの 3 年間の総ビルド数で，リモートはリモートの 3 年間の総ビルド数で割った値に 100 をかけた値を E/T として表 3.4 へ記述した．すなわち，E/T

の値は 3 年間のローカルビルド/リモートビルドにおける各エラーの占める割合にあたる．なお，3 年間のローカルビルド/リモートビルドの総数は表 3.2 から 3.3 の Environment を引いた値になり，ローカルビルドが 5736 回，リモートビルドが 6145 回である．

表 3.3 の調査の結果，全年度にわたってリモートビルドでは Dependency のエラーが最も多かった．これは Seo らの Google におけるビルドエラーの調査報告とも一致している．つまり，Dependency に関するビルドエラーは学生・実務家問わず発生しやすいエラーであることが分かる．さらに，ローカルビルドよりリモートビルドにおいて Dependency のエラーが多いことからも，Dependency はチーム開発における特徴的なビルドエラーであることが分かる．

リモートにおける Dependency のエラーの中でも特に “シンボルを見つけられません (`cannot find symbol`)” というエラーが多く，リモートの Dependency エラーの 88%，リモート全体におけるエラーメッセージの 51% を占めていた．このエラーメッセージが生じる原因として，テスト対象のファイルが完成していないにも関わらず，単体テストのテストファイル (以下，テストファイルと呼ぶ) をコミットしている事例を複数発見した．

例えば，ユーザ登録機能を実現するためには，Account.java ファイルなどの複数のファイルを実装し，ファイルごとに AccountTest.java などのテストファイルを実装し，テストやレビューを実施する必要がある．ここで，Account.java とそのテストファイルにあたる AccountTest.java を別々に異なる学生が実装するようなケースにおいて，テストファイルを実装した学生がそのテスト対象をまだ実装・コミットしていないにも関わらず，AccountTest.java のみをリポジトリにコミットしてしまうということが頻繁にあった．この場合，リポジトリには Account.java が存在しないため，リモートビルド時に，Account クラスが見つからない，つまりシンボルやパッケージが見つからないという Dependency エラーが生じてしまうことを防げない．Dependency エラーを防ぐためには，メンバー間のコミュニケーションが重要であるといえる．

具体的にはまず各チームメンバーが現在どのようなタスクを担当しているかを全員が把握している，あるいは即座に把握可能であるといった情報を共有することが必要であると考えられる．3.2 章で述べたとおり，今回の調査対象プロジェクトにおい

て学生らはチケット販売システムにおける 13 種類の機能 (ユーザ登録機能，チケット販売機能，チケット登録機能など) の実装を行う．各機能は複数のファイルで構成されており，学生らは特定の機能を構成する全てのファイルの実装，テスト，及びレビューを手分けして行う必要がある．このとき，どのファイルを誰がどのような順番で実装するかについては学生らが自由に決めることができる．単純な Syntax エラーのように，1 ファイル内の特定の箇所が原因で生じるエラーであれば，そのファイルを修正するのみで解決することが可能である．しかし，上記で述べたようなケースでは学生間で連携し，今からどのようなファイルを実装し，コミットするのか，そのコミットによってリポジトリの内容に不具合が生じないかどうか常に注意を払う必要がある．上記事例では，チームメンバーへ AccountTest.java の実装が完了し，コミットすることを周知し，Account.java の実装・コミットが完了しているかを確認することが必要となる．

今回の調査対象プロジェクトでは，予め教員からリモートでビルドエラーを発生させないようにローカルでビルドを行ってからコミットを行うこと，と指導されていた．しかし，表 3.4 に示したように，実際には多くの学生がローカルビルドをあまりせずにリモートビルド (すなわちコミット) を行っていた．結果として，表 3.4 より，Syntax を除く全てのエラー分類について，ローカルでビルドを頻繁に行っている学生群のほうがローカルビルドでのエラー率は高いが，リモートビルドにおけるエラー率は低くなっている．

類似するケースとして，特定のエラーを解決するために複数のソースコードを修正する必要があったものの，修正を行った学生が修正したファイルの全てをコミットせず，一部ファイルのみをチームリポジトリへコミットしたと思われる事例を発見した．この場合，ローカル環境においてはビルドエラーは生じないが，対象のエラーを解決するために操作する必要があったソースコードは，チーム共有リポジトリにおいては修正前の状態のままであるため，リモートビルド時にビルドエラーが生じた．このようなエラーを回避する場合，エラーを解決するために操作が必要なファイルが複数あると分かった時点で，チームメンバーとコミュニケーションを取り注意喚起をし，速やかにバグ修正のチケットを作成する必要がある．

また，図 3.5 のビルドエラーを，具体的に各学生がどの種類のビルドエラーをいくつ確認したのかについてまで調査を行った．各学生のビルドエラーの分類につい

て，付録 B.1 に示す．調査の結果，ローカルビルドで Syntax エラーを確認した学生はリモートビルドで Syntax エラーを引き起こしておらず，かつリモートビルドで Syntax エラーを引き起こした学生は全員ローカルビルドで Syntax エラーを確認していないことが確認された．調査対象プロジェクトで用いた Eclipse は基本機能として文法チェックや依存関係チェックを自動的に行うことが可能である．しかし，実際にローカル・リモートのビルドログを調査したところ，リモートビルドにおいて Syntax エラーが発生した全ての事例において，明らかに Eclipse の文法チェックで検出されるような Syntax エラーであるにも関わらずそれを無視し，ローカルビルドも実行せずに学生がリポジトリにコミットしていたことが判明した．

さらに詳細にエラー分類ごとのローカル/リモートのエラーを確認したところ，リモートビルドにおいて発生している Syntax エラー，Semantic エラー，Annotation エラーに分類されるエラーの中に，学生が各自ローカルビルドを行っていれば解決可能であったエラーが複数存在することが判明した．我々の調査では，Syntax エラー，Semantic エラー，Annotation エラーに属するエラーには Syntax エラーに代表される，特定の単一ファイル内での記述が原因で発生するものと，本章でも述べたテストファイルとテスト対象を別々の学生が実装し，コミットした結果発生しているような複数ファイルにまたがる記述が原因で発生するものの 2 種類が存在する．この中で前者に分類されるエラーは，その原因が単一ファイルに閉じているため，ローカルビルドによってエラーを確認し，修正したのちに対象ファイルをコミットすれば，リモートビルドにおいて同一のエラーが発生することはない．以上より，ローカルビルドがリモートでのビルドエラーの削減に効果があり，教員はコミット前にローカルビルドを行うことの重要性を学生により一層意識づける必要があると考えられる．

図 3.5 より，3 年間のうち Annotation のエラーは 2014 年度のチーム 5 で顕著に発生している．2014 年度のチーム 5 のビルドログを詳細に分析したところ，この Annotation エラーは全て preSprint の日に生じており，preSprint 以降は一度も生じていなかった．このチーム以外にも，リモートで Annotation が生じているチームは 3 年間を通して 9 チームあるが，9 チーム中 7 チームは Annotation のエラーは preSprint から Sprint1 までの期間に生じており，残りの 2 チームも Sprint2 以降 Annotation のエラーは生じていない．つまり，Annotation のエラーは主に技術的な問題によるところが大きく，利用方法について習熟することで，ビルドエラーを

削減できるといえる．

また，Dependency，Annotation に続き Type Mismatch に関するエラーがリモートで頻繁に生じていた．Type Mismatch のエラーが生じたソースコードを確認したところ，別のファイルでリストに格納する変数の型を宣言しているにも関わらず，型の宣言を行っているファイルが完成していない状態でコミットを行っていることが判明した．

3.4.3. RQ3: 各種ビルドエラーの解決にはどれぐらいの時間がかかるか

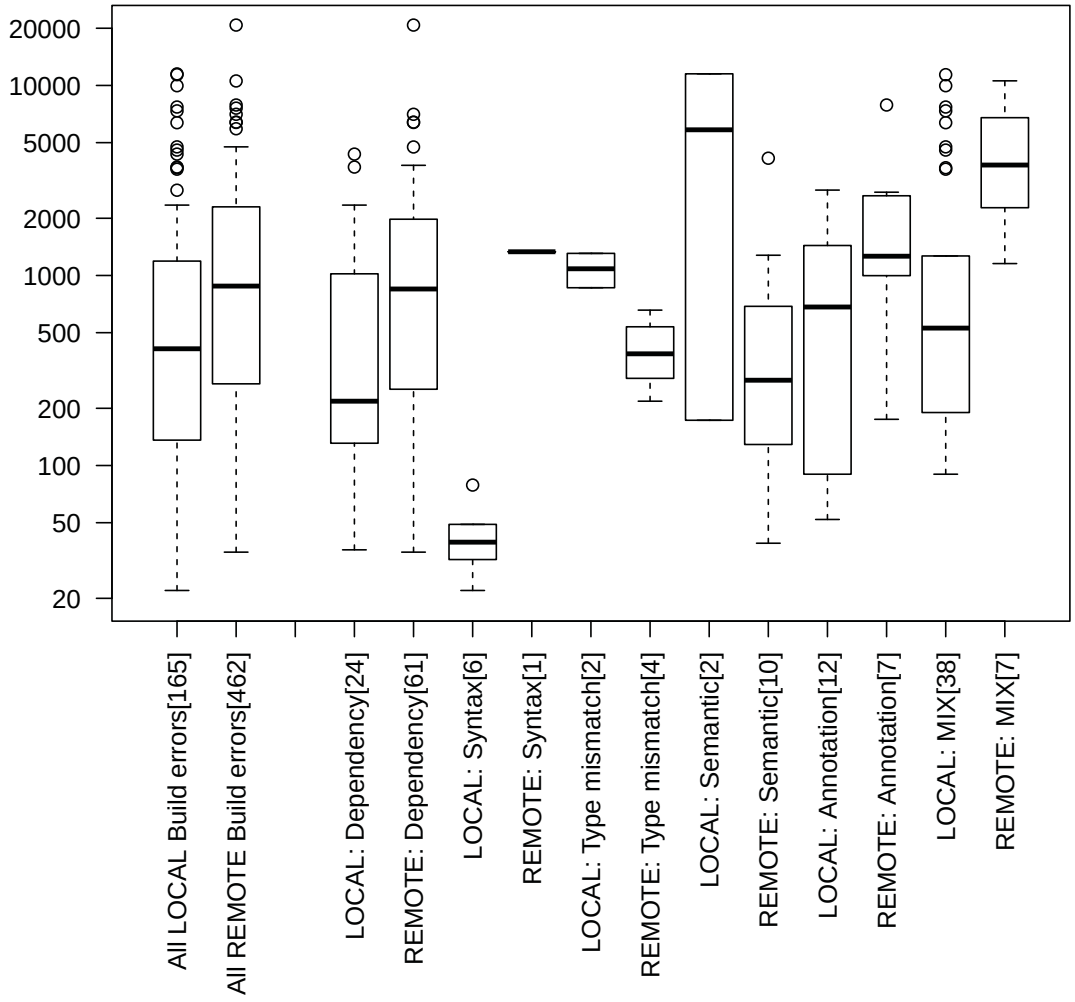


図 3.6 エラー分類ごとのエラー解決時間 (2013 年度から 2015 年度)

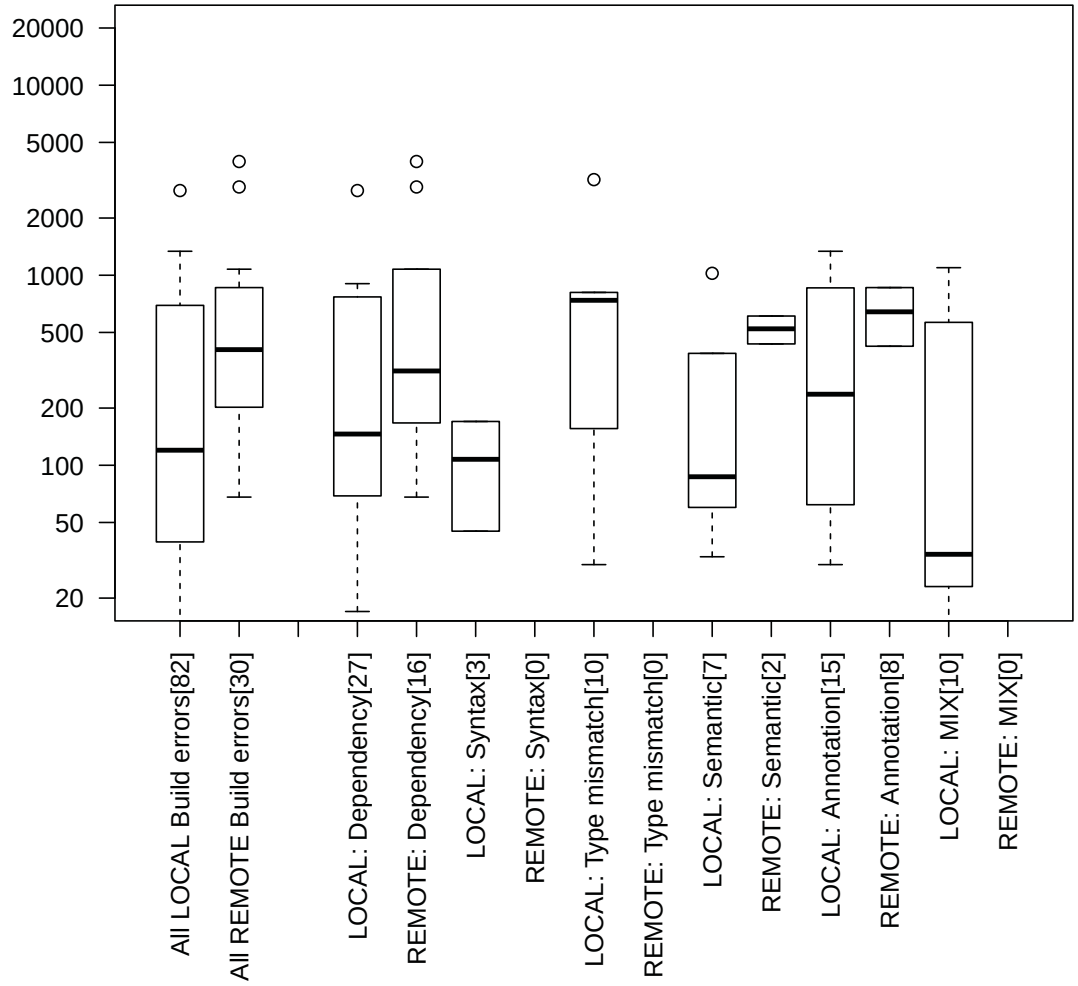


図 3.7　エラー分類ごとのエラー解決時間 (2016 年度)

図 3.6 に 2013 年度から 2015 年度の，図 3.7 に 2016 年度のローカルビルドとリモートビルドそれぞれについての各種エラー解決時間 (秒) を示す．以下，図 3.6 について結果を述べる．2016 年度の結果については，第 3.5.1 章で述べる．

LOCAL，REMOTE の文字の横に書かれている数字はそれぞれのビルドエラー数を示している．エラー解決時間は 3.3.3 章で先述したとおり，特定のエラーが生じたビルドのビルド終了時間から，次にエラーが解決した (エラーが生じなくなった) ビ

ルドの開始時刻までの時間である．

箱ひげ図の中央値に着目したところ，全体的にリモートビルドのほうがエラー解決に時間がかかっていることが分かる．一方で，一部のローカルビルドにおけるエラーはリモートビルドにおけるエラーよりも解決に時間がかかっていることも見て取れる．また，解決までに複数種類のビルドエラーが含まれていた場合は，MIX と定義して一番右へ結果を示している．分類名横の括弧中の数値は，対象となったビルドエラー数にあたる．それぞれのエラー分類のうち，左がローカルでのエラー解決時間で，右がリモートでのエラー解決時間を示す．図 3.6 より，エラー数の少ない Syntax と Type mismatch，Semantic を除くと，残りの Dependency，Annotation，MIX はローカルよりリモートのほうがエラー解決に時間がかかっていることが分かる．さらに，リモートビルドのエラーの中でも MIX，すなわち，複数の要因によるビルドエラーが最も解決に時間がかかっていることが分かる．

3.4.4. 2013 年度から 2015 年度の結果の考察

本研究では CloudSpiral で行われているソフトウェア開発 PBL におけるビルド活動に着目し，分析を行った．RQ1 でリモート及びローカルのビルドエラーの分類を行い，その結果をふまえて RQ2，RQ3 について調査を行った．本章では，2013 年度から 2015 年度のビルド活動に関して，RQ2，RQ3 の調査結果から得られた知見について考察を行う．

リモートビルドにおけるエラーの原因

RQ2 の結果をふまえ，リモートビルドにおけるエラーの原因について考察する．

リモートビルドにおいて発生するエラーには技術的な要因によるものと，学生間のコミュニケーションに関するものがあることが分かった．調査では，Syntax エラー，Semantic エラー，Annotation エラーのうち，単一ファイル内での記述が原因で発生しているエラーの多くは技術的な要因で発生していることが分かっている．例えば，3.4.2 章で述べたとおり Semantic のエラーの一部や Syntax のエラーは，各個人がローカルでのビルドを怠っていたため，ローカルで確認可能なエラーがリモートにおいて生じていた．さらに，Annotation に関するエラーは全て開発が始まってから 2 日以内に生じていた．このことから Annotation に関するエラーは学生が調

査対象プロジェクトで使用していたバリデータの実装方法を理解していなかったため生じたと考えられる．

一方，Dependency や Type Mismatch はコミュニケーションに起因することが多いという結果が得られた．3.4.2 章より，リモートにおける Dependency のエラーはテスト対象のファイルがコミットされていないにも関わらずテストファイルをコミットして生じることが多かった．さらに Type Mismatch で頻出していたエラーとして cant.apply.symbol があり，このエラーログを確認したところ Dependency の cant.resolve と同時に生じていることが多かった．これらのエラーが共起したコミットログを確認したところ，リストへ格納するデータの型を他のファイルで宣言しているにも関わらず，宣言元のファイルが存在しない状態でコミットを行っている事例を確認した．

類似した事例として，Dependency の cant.resolve というエラーが生じるとき，テスト対象のファイルがコミットされていないにも関わらず，テストファイルをコミットしている事例が確認された．これらは 3.4.2 章で述べたとおり，チームメンバー間で誰がどのコンポーネントを担当しているかを確認し合うことで，エラーを防ぐことが可能である．以上より，Dependency や Type Mismatch に関するエラーは学生同士が口頭でコミュニケーションを取ることで防ぐことが可能であったと考える．

発生したエラー数については，学生は複数ソースコード間におけるシンボルやパッケージの宣言ミス，すなわち Dependency(依存関係の不整合) によるエラーが最も多い．さらに Dependency によるエラーログを詳細に調べたところ，テストファイルのコミット時に Dependency のエラーが頻繁に生じており，コミットすべきファイルについての学生間のコミュニケーションに問題が発生していたといえる．この結果は Seo ら [49] や Phillips [44] らといった企業での報告とも一致しており，ビルドの過程における Dependency やコミュニケーションの問題はプログラミング能力の有無に関係なく発生するといえる．

ローカルビルドの重要性

RQ2 の結果をふまえ，ローカルビルドの重要性について考察する．

3.3.2 章で述べたとおり，本プロジェクトでは事前講義において，版管理システムを利用したチームソフトウェア開発におけるローカルビルドの重要性についての指

導を行っている．しかし，3.4.2 章の表 3.4 で取り上げた，ローカルビルド数よりリモートビルド数が多い学生は，148 名のうち 92 名にのぼり全体の約 62% を占めた．教員の指導どおりに，リモートへコミット (リモートビルド) を行う前にローカルでソースコードの動作確認 (ローカルビルド) を行っていた場合，リモートビルド数よりローカルビルド数が多くなるため，92 名については教員の指導どおりにビルドを行っていなかったことが分かる．

調査の結果においても，ローカルビルドを積極的に行っている学生のほうがリモートビルドでのエラー発生率が低くなっている．特に Syntax に関するビルドエラーのような技術的な要因によるエラーについて，学生がローカル環境でのビルドを行うことで見つけることが容易である．しかし，実際にはローカルビルドを行わない学生が，エラーが残ったままのソースコードを SVN リポジトリへコミットを行い，リモートビルドでエラーが生じてしまった事例が確認された．

一方で 2013 年度の Team2，3 のように，リモートビルドでのエラーが非常に少なく，3.2.1 章の手順 2 で定義したエラーの伝播も殆ど発生していないチームが各年度 2〜3 チーム存在する．これらのチームにヒアリングを行ったところ，早期段階でリモートビルドでのエラーが発生しないよう，メンバー間での意識共有を行っていたことが分かった．具体的には，ローカルビルドの徹底やコミット内容の事前確認，コミット時のセルフレビューといった様々な対策をチーム内で決定し，順守するよう相互に声を掛け合うといった対応を取っていた．結果として，これらの対策がリモートビルドにおけるエラーの発生やエラーの伝播を少なくしたと考えられる．

ソフトウェア開発 PBL を進める上では，事前講義においてローカルビルドの重要性を伝えるだけでなく，学生チームが自発的に重要性を理解し，積極的にローカルビルドを実施するような働きかけを行うことが重要であると考えられる．

ローカル/リモートのビルドエラーの違い

RQ3 の結果をふまえ，ローカルとリモートビルドにおけるエラーの違いについて考察する．

版管理システムを用いたソフトウェア開発では，コミットを行う前にローカルビルドによってビルドエラーを洗い出しておき，リモートビルドでのビルドエラーを発生させないことが重要になる．ソフトウェア開発 PBL においても同様であり，調

査において解決時間や解決を目的とした作業内容について，差異があることが調査により分かった．以降の段落において，解決時間及び解決を目的として学生が行った作業内容に関する考察を述べる．

図 3.6 において，解決に 1 時間以上要しているローカルビルドにおけるエラーをより詳細に調査したところ，実装に不慣れな特定の学生が問題の特定や解決方法の探索に時間がかかり，悩んでいるケースが多いことが分かった．このようなケースでは，周囲の学生がフォローすることで，解決時間を早めることができる．そのためには，解決に時間がかかりそうだと分かった時点で，その学生自身がサポートを積極的に求めるといった対応を取らせることが重要である．

一方，リモートビルドにおいて，ビルドエラーの解決に 1 時間以上を要している 11 件を分析したところ，技術的な問題だけでなく，開発の進め方の問題で解決時間が長くなっている事例が見られた．例えば，11 件中 7 件は 3.5.1 章で述べたような依存関係のあるファイル (テスト対象ファイル，リストに格納する要素の型を宣言しているファイル，インポート先のファイル等) の実装が完了しコミットされるまで待っていたため，結果としてエラー解決時間が長引いていた．他にも，技術的に解決可能な学生の手が空いてなかった事例やバグが埋め込まれてからしばらくしてエラーが発生したため，即座に対応可能な学生がいなかった事例などを確認した．特に，2013 年度のチーム 5，2014 年度のチーム 1，7，9，2015 年度のチーム 4，5 では，学生の一人がリモートにおいてビルドエラーを発生させた後，解決を保留する等した結果，他の学生がビルドエラーを発生させるソースコードを SVN からチェックアウトし，エラーの伝播が発生している．エラー解決時間については，図 3.6 からも分かるように，複数種類のエラーを含むことで，解決に時間を要してしまう．そのため，リモートビルドにおけるエラーについては，可及的速やかにエラー解決を行えるよう，開発の進め方をコントロールする方法を指導する必要があると考えられる．

PBL 教育への適用

本章では，RQ1 により得られた分類を基に，RQ2，RQ3 を調査した結果どのように PBL 教育へ適用可能かについて考察を述べる．

RQ2 より，技術的な要因に起因するエラーとコミュニケーション不足に起因するエラーの両者があることが分かった．技術的な要因により発生したエラーとして

Syntax，Semantic，Annotation エラーがあげられる．これらは，単一ファイルに閉じたエラーであるため，ローカルビルドで発見が可能である．このことから，教員は演習前にこれらのエラーについて解決方法を提示し，ローカルビルドを徹底するよう指示必要があると考えられる．一方，チーム内のコミュニケーション不足に起因するエラーでは，エラーを引き起こすコードの特定や解決に時間がかかっていた．このようなエラーはテスト対象ファイルが完成していないのにもかかわらず，テストファイルを SVN リポジトリへコミットするなど，複数ファイルに起因して生じることが多かった．RQ3 の調査結果より，エラーの修正を怠った結果，別のエラーが追加されてしまうと，エラー解決に時間がかかってしまうことが分かった．したがって，教員は学生に対して，依存関係のあるファイル（例：テストファイルや import 文を含むファイル）を SVN リポジトリへコミットするときは，エラーが生じないかメンバーへ確認することを勧めるべきである．

さらに RQ2，RQ3 からローカルビルドを怠るとエラーの伝播が生じ，先述したように複数種類のエラーが混ざりエラー解決に時間を要することが判明した．よって，教員はローカルで生じたビルドエラーが，他のチームメンバーへ伝搬する流れを学生へ説明し，演習前へローカルビルドでエラーを確認しなくすことの重要性を伝えるべきである．

3.5. 対象プロジェクトの改善

2015 年までの RQ2，RQ3 の結果からローカルビルドを怠るとエラーの伝播が生じ，先述したように複数種類のエラーが混ざりエラー解決に時間を要することが判明した．また，ローカルビルドを実施していないことによるエラーと学生間コミュニケーション不足に起因するエラーがあることが分かった．

そのため，2016 年度の同ソフトウェア開発 PBL において，CloudSpiral の教員により以下の改善が行われた．

学生によるエラーの原因の特定 毎スプリント後の学生の振り返りにおいて，学生にリモートビルドにおいて生じたエラーの原因を調査するように義務付けた．

エラーを放置するリスクの通知 教員がプロジェクト前の講義において，リモートビルドで過去の PBL で発生したエラーの種類とその原因を説明した．また，エ

ラーの伝搬について説明し，Mix の場合にエラー解決に時間がかかったことを述べた．

ビルド活動に着目したフィードバック　毎振り返り後，教員が各チームを訪れ，学生によって調査されたエラー原因がローカルビルドの不足によるものか，学生間コミュニケーションの不足のどちらであるかを確認した．

次章で 2016 年度の結果について述べる．

3.5.1. 改善内容

2016 年度の CloudSpiral ソフトウェア開発 PBL の改善について，ビルドエラー数の観点とエラー解決時間の観点より述べる．

ビルド数・ビルドエラー数の変化

表 3.2 や表 3.3 から分かるように，2016 年度のリモートビルドにおけるエラー数は 2013 年度から 2015 年度と比べ最も少なくなった．特に，過去 3 年間でもっともリモートビルドエラーが生じた 2014 年度に比べ，7 分の 1 にまで減らすことに成功した．なお，2016 年度は CloudSpiral の教員によりローカルビルドによるエラーの洗い出しが徹底されていたため，ローカルビルド数が増えたことは自明である．

ローカルビルドにおいては全ての種類のビルドエラーが生じている一方で，リモートビルドにおいては Syntax エラーが生じていない．また，Type Mismatch や Semantic のローカル/リモートビルドエラーについて確認したところ，章で述べたような，ローカルビルドを行うことで単一のファイル内で解決可能なエラーの洗い出しが行われていた．

図 3.5 より，2016 年度のチーム 1 については，ローカルビルドにおいて Syntax エラーと Annotation エラーを確認しているが，リモートビルドにおいてこれらエラーは生じていない．すなわち，学生が自らローカル環境において自身のソースコードに含まれていたこれらエラーを洗い出し，チームリポジトリにコミットを行う際はエラーの伝播を防ぐように動いていたことが分かる．

チーム 2 やチーム 7 においてはローカルビルドで確認されたエラーが少なく，かつリモートビルドにおいて Annotation エラーが生じている．これらのログの詳細

を確認したところ，合宿初日や 2 日目など，まだチーム開発に慣れていないと考えられる時期にリモートビルドにおいて Annotation エラーが生じていた．これは 2013 年度から 2015 年度と同様の結果であり，3 日目以降，システムに対する習熟度があがるに伴い Annotation エラーは発生しなくなった．

以上より，特定の種類のビルドエラーが生じる原因や，その解決策について学生らに講じさせることは，ビルドエラーに対する理解を促進し，エラーの削減に繋がることが確認された．

エラー解決時間の変化

2016 年度のローカル/リモートビルドにおける各種ビルドエラーの解決時間について，図 3.7 に示す．2016 年度は，CloudSpiral の教員によりリモート環境においてエラーを放置するリスクについて教育された結果，リモートビルドにおける MIX エラーは生じなかった．すなわち，ソフトウェア開発に不慣れな学生でも適切にチームメンバーとコミュニケーションを取り，また教員から適切なフィードバックを受けることで，リモートビルドにおけるエラーの削減や，エラーの伝播を防ぐことが可能であることが分かる．

3.6. 本章のまとめ

本研究ではソフトウェア開発におけるビルド工程に着目し，CloudSpiral で開講されているソフトウェア開発 PBL における学生のビルド活動の調査を行った．

調査は 2 段階に分かれており，まず 2013 年度から 2015 年度のビルドログを分析し，ビルドエラーの分類や各種エラーの解決策について講じた．調査の結果，我々は得られたビルドエラーを Seo らの分類を参考に，PBL におけるローカル環境，リモート環境のビルドログを 6 つの項目へ分類した．そして，それぞれの分類に対しエラーの回数，エラー継続時間等を測定し，ローカルとリモートでビルドの傾向を比べることで，ローカルで解決可能なエラーや，リモートで教員が優先して留意すべきエラーを調査した．調査の結果，既存研究において述べられているソースコード間のシンボルやパッケージや宣言といった Dependency(依存関係の不整合) によるエラーや，チーム内におけるコミュニケーション不足は，学生同士のチーム開発におい

ても影響を及ぼすことが判明した．

次に，2016 年度の同演習において，2013 年度から 2015 年度の結果を基に CloudSpiral の教員により改善が行われた．改善の内容として，ローカルビルドの徹底や，生じたビルドエラーに関して学生らにその原因について調査を行ってもらい，その結果をふまえて教員からビルドエラーに関して適切なフィードバックを行った．改善の結果，2016 年度のリモートビルドにおけるエラー数の削減やエラー解決時間の削減に繋がった．

ビルドはソフトウェア開発において必要不可欠な工程である．本研究により得られた結果は，CloudSpiral におけるソフトウェア開発 PBL のみならず，他の教育機関において開講されているソフトウェア開発演習などにおいても活用可能であると考える．

第 4 章

結論

本研究では，プログラミング演習及びソフトウェア開発演習において，学生が行う探索的行動に着目した分析・支援を行った．

まずプログラミング演習において，ソースコードの変更履歴に着目した探索的行動である，探索的プログラミングについて分析・支援を行った．従来，探索的プログラミングがソースコードの実装が不明確な箇所に対して行われる特徴に着目し，探索的プログラミングが行われている箇所を自動検出することで，学生が実装に躓いているプログラミングの機能や課題について，教員側から特定できると考えた．始めに，初学者が行う探索的プログラミングの実態調査を行い，探索的プログラミングを行う上での初学者にとっての難所や支援すべき観点について調査を行った．調査結果を基に，初学者向け探索的プログラミング支援環境を提案し，次に探索的プログラミングが行われている箇所を自動検出するアルゴリズムを提案した．提案アルゴリズムを用いて，実際のプログラミング演習における学生の探索的プログラミング行動を分析したところ，検出結果から同一の課題に対する各学生の異なるアプローチの検出や，エラーが混入した原因となる変更の特定及びその学生がどのようにして課題を解こうとしていたのか，学生のプログラミングにおける試行錯誤まで教員から推測することが可能であった．

次にソフトウェア開発演習において，ビルドエラーに着目した初学者の探索的行動の調査を，教育プロジェクトである CloudSpiral において実施した．CloudSpiral のカリキュラムの一端であるソフトウェア開発演習において，学生がビルドを行う

ときのログを基に調査を行った．まず，2013 年度から 2015 年度のビルドログを対象に実態調査を行った．具体的には，初学者が陥りやすいビルドエラーを調査するために，Seo らの Google におけるビルドエラーの報告を基に，エラーログのカテゴライズ及び各エラー数，エラー継続時間を主に分析し，各エラーの原因，解決策について講じた．そして，2013 年度から 2015 年度の調査で得られたビルドエラーが生じる原因及び解決策を基に，CloudSpiral の教員によって 2016 年度の同ソフトウェア開発演習の改善がなされた．改善の結果，ビルドエラー数は最も多かった 2014 年度に比べ，2016 年度は 7 分の 1 にまで削減した．

本研究の主な貢献は次のとおりである．

本研究の具体的な貢献は次のとおりである．

プログラミング演習における探索的プログラミング行動を自動検出するアルゴリズムの提案

本研究で提案したアルゴリズムは，ソースコードの編集履歴における差分に着目するため，初学者に多い文法エラーを含むソースコードでも探索的プログラミング行動が検出可能である．また，構文解析や特別なツールなども必要としないため，様々な教育機関だけでなく，e-learning，MOOCs を用いたオンラインのプログラミング教育にも活用可能である．

ソフトウェア開発演習におけるビルドエラーの分類の再検討

本研究では熟練の開発者向けのビルドエラーの分類を基に，ソフトウェア開発経験の浅い学生を対象にビルドエラーについて調査を行った．この際，熟練の開発者向けのビルドエラーの分析で行われていた手法に固執せず，リサーチクエスチョンによる定量的な分析と，付録 C，D に掲載しているような複数種類のグラフを用いて定性的な分析を行い，ビルドエラーの分類に関して再検討を行った．さらに，従来研究では述べられていなかった各種ビルドエラーを減らすための具体的な解決策についてまで講じた．

これらの本研究の結果より，プログラミング演習やソフトウェア開発演習において，教員は各学生やチームのソフトウェアに対する理解度や試行錯誤に応じたアドバイスが可能になる．教員が学生へ適切にソフトウェア開発に関するアドバイスを

与えることで，学生は限られた演習環境においてもソフトウェア開発に対する理解を深め，より実践的な技術を身につけることが可能となる．

実践的な技術を持つ若手技術者の育成は，近年産業界においても強く求められている．本研究で提案したプログラミング演習における探索的プログラミングの支援や，ソフトウェア開発演習におけるビルドエラーを削減するための対策は，エラーが長時間継続してしまうことを防ぎ，初学者の演習に対するドロップアウトやモチベーションの低下を防ぐことに繋がると考える．また，提案手法は教育機関など場所に依存しないため，e-learning や MOOCs などオンラインを用いたプログラミング教育と組み合わせることで，場所に依存しない初学者のプログラミングやソフトウェア開発教育の支援に貢献できると考える．

謝辞

本研究を進めるにあたり多くの方々に御指導，御協力，御支援を頂きました，ここに誠意を添えて御名前を記させていただきます．

奈良先端科学技術大学院大学 情報科学研究科 ソフトウェア設計学研究室 飯田 元 教授には，本学における本研究の全過程において熱心な御指導を賜りました．研究方針だけではなく，研究に対する姿勢，論文執筆，発表方法についても多くの御助言を頂きました．また，CloudSpiral を始めインターンシップ，その他課外活動など，幅広くソフトウェア工学教育に携わる機会を賜りました．5 年間充実した大学院生活を達成することができたのは先生の御指導と御人柄によるものであると強く確信しております．心より厚く御礼を申し上げます．

奈良先端科学技術大学院大学 情報科学研究科 ソフトウェア工学研究室 松本 健一 教授には，様々な場面で本研究に対し貴重な御指導，御助言を賜りました．先生から頂いた鋭い御指摘により，改めて自分の研究の学術的価値や展望について見つめ直すことができました．また，研究室の異なる学生であるにも関わらず，韓国ハンバット大学へのインターンシップへの参加の御機会を賜りました．心より厚く御礼申し上げます．

奈良先端科学技術大学院大学 情報科学研究科 ソフトウェア設計学研究室 市川 昊平 准教授には，様々な場面で本研究に対し貴重な御指導，御助言を賜りました．また，日々の研究室での行事・生活を通し，研究に対する姿勢，発表方法についても多くの御助言を頂きました．心より感謝申し上げます．

奈良先端科学技術大学院大学 情報科学研究科 ソフトウェア設計学研究室 崔 恩瀞 助教には，様々な場面で本研究に対し貴重な御指導，御助言を賜りました．先生には常日頃より研究者としての心構えや取り組み方を学ばせていただきました．また，

学生生活においても常日頃より御気遣いを賜り，研究者としてだけではなく一人の学生として多大な御支援を賜りました．心より感謝申し上げます．

大阪工業大学 情報科学部 情報システム学科 Team Software Development 研究室 井垣 宏 准教授には，研究に対する御指導のみでなく，研究者として，また教育者としてのありかたについてまで幅広く熱心な御指導を賜りました．私がソフトウェア工学教育について博士前期課程から後期課程の約 5 年間一貫して続けることができたのは，先生からの温かい御助言によるものだと強く確信しております．また，CloudSpiral で得た経験や知識は CloudSpiral 修了後も研究活動及び大学生活において非常に助けとなりました．心より厚く御礼を申し上げます．

名古屋大学 大学院情報学研究科 附属組込みシステム研究センター 吉田 則裕 准教授には，本研究を進めるにあたり，広範囲かつ多大な御助力を頂きました．特に，学会発表や論文投稿時に貴重な御助言を頂戴いたしました．また，私が本学へ入学する前のスプリングセミナー，入学後の研究テーマの設定では様々なアドバイスを頂きました．心より感謝申し上げます．

豊田工業高等専門学校 情報工学科 藤原 賢二 助教には，本研究を進めるにあたり，広範囲かつ多大な御助力を頂きました．また，大学院や研究室での活動全般にわたって様々な御助言・御厚意を頂きました．心より感謝申し上げます．

大阪大学 サイバーメディアセンター 先進高性能計算機システムアーキテクチャ共同研究部門 渡場 康弘 特任講師には，様々な場面で本研究に対し貴重な御指導，御助言を賜りました．心より感謝申し上げます．

大阪工業大学 情報科学部 情報システム学科 Database & Software Engineering 研究室 深海 悟 教授には研究に対して貴重な御指導，御助言を賜りました．また，私が本学へ入学するきっかけを与えていただき，卒業後もお会いする度に様々な御助言・御厚意を頂きました．心より感謝申し上げます．

奈良先端科学技術大学院大学 小川 暁子 氏，森本 恭子 氏，吉川 弥甫 氏，荒井 智子 氏，仲田 綾奈 氏，高岸 詔子 氏には，研究の遂行に必要な事務処理など，多岐にわたり御助力頂きました．心より感謝申し上げます．

奈良先端科学技術大学院大学 情報科学研究科 ソフトウェア工学研究室 畑 秀明 助教，伊原 彰紀 助教には，特に学会活動において本研究に対するアドバイスや御助力を頂きました．心より感謝申し上げます．

I would like to express my great appreciation to Professor. Putchong and Professor. Chantana for all the helpful advices and comments during my internship in Kasetsart University.

I greatly appreciate Professor. Jittat for all his helps to gather Thai students and his afterward supports for my experiment. Also I am also greatful thanks Kasetsart University students who participate in my experiment.

大阪府立大学工業高等専門学校 本科 2 年生 情報処理 I 受講生の皆様には，貴重な 2 時限の演習の時間を頂き本研究におけるツールの評価実験へ参加して頂きました．心より感謝申し上げます．

奈良先端科学技術大学院大学 情報科学研究科 ディペンダブルシステム学研究室 井上 美智子 教授は韓国ハンバット大学へのインターンシップと言う貴重な御機会をくださいました．ハンバット大学では，英語の発音からスライドの構成まで，英語での論文発表に関する広い御指導を賜ることができました．心より感謝申し上げます．

My deep appreciation are extended to Professor. Hyunbean and his laboratory students, especially to Jeong Seok, Taegeun Kang and Sunyoung Kim. Thanks to their support, I had a great time in Deajeon as internship student in Hanbat National University.

奈良先端科学技術大学院大学 情報科学研究科 ソフトウェア設計学研究室の OB，先輩，同期，後輩の皆様には，日頃から多大な御協力と御助言を頂き，公私ともに支えて頂きました．特に博士後期課程の学生として立場を同じくした Chawanat Nakasan 氏，Pongsakorn U-chupala 氏，Chan Kar Long 氏，黄 掣 氏，上村 恭平 氏には，研究だけではなく日々の学校生活においても広く御支援を頂き，精神的な支えとなってくださいました．皆様のおかげで非常に有意義な大学院での生活を送ることができました．本当にありがとうございました．

奈良先端科学技術大学院大学 情報科学研究科 ソフトウェア工学研究室 尾上 紗野 氏，ユビキタスコンピューティングシステム研究室 平部 裕子 氏は，大学のみならず私生活の場においても非常に大きな精神的支えとなってくださいました．氏らのおかげでたとえ忙しい時期でも楽しい学生生活を過ごすことができました．心より深く感謝いたします．

2013 年度 CloudSpiral 受講生の同期の方々には，CloudSpiral 受講時における活

動だけでなく，修了後も私生活において楽しい時間を過ごさせて頂きました．本当にありがとうございました．

最後に，常に温かく見守りそして辛抱強い支援と応援を続けてくださった両親，優しく見守ってくださった祖母，相談に乗ってくれた弟へ，心より深く感謝いたします．

参考文献

[1] Lukas Alperowitz, Dora Dzvonyar, and Bernd Bruegge. Metrics in agile project courses. In *Proceedings of the 38th IEEE International Conference on Software Engineering Companion*, pp. 323–326, 2016.

[2] Pansy Arafa, Daniel Solomon, Samaneh Navabpour, and Sebastian Fischmeister. Debugging behaviour of embedded-software developers: An exploratory study. In *Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computing*, pp. 89–93, 2017.

[3] Joel Brandt, Philip J. Guo, Joel Lewenstein, Mira Dontcheva, and Scott R. Klemmer. Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In *the SIGCHI Conference on Human Factors in Computing Systems*, pp. 1589–1598, 2009.

[4] Joel Brandt, Philip J. Guo, Joel Lewenstein, and Scott R. Klemmer. Opportunistic programming: How rapid ideation and prototyping occur in practice. In *Proceedings of the 4th international workshop on End-user software engineering*, pp. 1–5, 2008.

[5] Jane Cleland-Huang and Mona Rahimi. A case study: Injecting safety-critical thinking into graduate software engineering projects. In *Proceedings of the 39th International Conference on Software Engineering: Software Engineering and Education*, pp. 67–76, 2017.

[6] Alan Dearle. Software deployment, past, present and future. In *Proceedings of Future of Software Engineering*, pp. 269–284, 2007.

[7] Paul Denny, Andrew Luxton-Reilly, and Ewan Tempero. All syntax errors are not equal. In *Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education*, pp. 75–80, 2012.

[8] Ashish Dutt, Maizatul Akmar Ismail, and Tutut Herawan. A systematic review on educational data mining. In *Proceedings of IEEE Access*, pp. 15991–16005, 2017.

[9] 福安直樹, 井垣宏, 佐伯幸郎, 水谷泰治, 柗本真佑, 楠本真二. チーム内の役割分担を考慮したソフトウェア開発 PBL の評価基準と状況把握支援. 電子情報通信学会論文誌 D, Vol. J98-D, No. 1, pp. 117–129, 2015.

[10] 袴田大貴, 松澤芳昭, 太田剛. 初学者向けデバッガ DENO の利用実態の分析. 研究報告コンピュータと教育 (CE), 第 123 巻, pp. 1–8, 2014.

[11] Ahmed E. Hassan and Ken Zhang. Using decision trees to predict the certification result of a build. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pp. 189–198, 2006.

[12] Felienne Hermans and Efthimia Aivaloglou. Teaching software engineering principles to k-12 students- a mooc on scratch. In *Proceedings of the 39th International Conference on Software Engineering: Software Engineering and Education*, pp. 13–22, 2017.

[13] Jim Highsmith and Martin Fowler. The agile manifesto. In *Software Development Magazine*, Vol. 9, pp. 29–30, 2001.

[14] Hiroshi Igaki, Naoki Fukuyasu, Sachio Saiki, Shinsuke Matsumoto, and Shinji Kusumoto. Companion proceedings of the 36th international conference on software engineering. In *Proceedings of the 36th International Conference on Software Engineering*, pp. 372–381, 2014.

[15] 井垣宏, 齊藤俊, 井上亮文, 中村亮太, 楠本真二. プログラミング演習における進捗状況把握のためのコーディング過程可視化システム C3PV の提案. 情報処理学会論文誌, Vol. 54, No. 1, pp. 330–339, 2013.

[16] Petri Ihantola, Arto Vihavainen, Alireza Ahadi, Matthew Butler, Jürgen Börstler, Stephen H. Edwards, Essi Isohanni, Ari Korhonen, Andrew Petersen, Kelly Rivers, Miguel Ángel Rubio, Judy Sheard, Bronius Skupas, Jaime Spacco, Claudia Szabo, and Daniel Toll. Educational data mining and learning analytics in programming: Literature review and case studies. In *Proceedings of the 2015 ITiCSE on Working Group Reports*, pp. 41–63, 2015.

[17] Matthew C. Jadud. Methods and tools for exploring novice compilation behaviour. In *Proceedings of the second international workshop on Computing*

education research, pp. 73–84, 2006.

[18] Arne Jansen, Maarten Van Mechelen, and Karin Slegers. Personas and behavioral theories: A case study using self-determination theory to construct overweight personas. In *Proceedings of the Conference on Human Factors in Computing Systems*, pp. 2127–2136, 2017.

[19] James B. Fenwick Jr., Cindy Norris, Frank Barry, Josh Rountree, Cole Spicer, and Scott Cheek. Another look at the behaviors of novice programmers. In *Proceedings of the 40th ACM technical symposium on Computer science education*, pp. 296–300, 2009.

[20] Mary Beth Kery and Brad A. Myers. Exploring exploratory programming. In *Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computing*, pp. 25–29, 2017.

[21] Noureddine Kerzazi, Foutse Khomh, and Bram Adams. Why do automated builds break? an empirical study. In *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution*, pp. 41–50, 2014.

[22] Andrew J. Ko and Brad A. Myers. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Transactions on Software Engineering*, Vol. 32, No. 12, pp. 971–987, 2006.

[23] Andrew J. Ko and Brad A. Myers. Debugging reinvented: Asking and answering why and why not questions about program behavior. In *Proceedings of the 30th international conference on Software engineering*, pp. 301–310, 2008.

[24] Stephan Krusche, Bernd Bruegge, Irina Camilleri, Kirill Krinkin, Andreas Seitz, and Cecil Wöbker. Chaordic learning: A case study. In *Proceedings of the 39th International Conference on Software Engineering: Software Engineering and Education Track*, pp. 87–96, 2017.

[25] Essi Lahtinen, Kirsti Ala-Mutka, and Hannu-Matti Järvinen. A study of the difficulties of novice programmers. In *Proceedings of the 10th conference on*

Innovation and technology in computer science education, pp. 14–18, 2005.

[26] Henry Lieberman and Christopher Fry. Bridging the gulf between code and behavior in programming. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pp. 480–486, 1665.

[27] Yun Lin, Jun Sun, Yinxing Xue, Yang Liu, , and Jinsong Dong. Feedback-based debugging. In *Proceedings of the 39th International Conference on Software Engineering*, pp. 393–403, 2017.

[28] Christian Macho, Shane McIntosh, and Martin Pinzger. Predicting build co-changes with source code change and commit categories. In *Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering*, pp. 541–551, 2016.

[29] 眞鍋雄貴, 井垣宏, 福安直樹, 佐伯幸郎, 楠本真二, 井上克郎. 細粒度プロジェクトモニタリングのための DaaS を利用したソフトウェア開発 PBL 支援環境の提案. 電子情報通信学会ソフトウェアサイエンス研究会, 第 112 巻, pp. 73–78, 2012.

[30] James G. March. Exploration and exploitation in organizational learning. *Organization Science*, Vol. 2, No. 1, pp. 71–87, 1991.

[31] Christoph Matthies, Thomas Kowark, Keven Richly, Matthias Uflacker, and Hasso Plattner. How surveys, tutors, and software help to assess scrum adoption in a classroom software engineering project. In *Proceedings of the 38th IEEE International Conference on Software Engineering Companion*, pp. 313–322, 2016.

[32] Sean McGregor, Hailey Buckingham, Thomas G. Dietterich, Rachel Houtman, Claire Montgomery, and Ronald Metoyer. Facilitating testing and debugging of markov decision processes with interactive visualization. In *Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computing*, pp. 281–282, 2015.

[33] Shane McIntosh, Bram Adams, and Thanh H. D. Nguyen. An empirical study of build maintenance effort. In *Proceedings of the 33rd International Conference on Software Engineering*, pp. 141–150, 2011.

[34] Iain Milne and Glenn Rowe. Difficulties in learning and teaching programming—views of students and tutors. *Education and Information Technologies*, Vol. 7, No. 1, pp. 55–66, 2002.

[35] Marcello Missiroli, Daniel Russo, and Paolo Ciancarini. Learning agile software development in high school: an investigation. In *Proceedings of the 38th IEEE International Conference on Software Engineering Companion*, pp. 293–302, 2016.

[36] Laurie Murphy, Gary Lewandowski, Renée McCauley, Beth Simon, Lynda Thomas, and Carol Zander. Debugging: the good, the bad, and the quirky – a qualitative analysis of novices' strategies. In *Proceedings of the 39th SIGCSE technical symposium on Computer science education*, pp. 163–167, 2008.

[37] Brad A. Myers, Stephen Oney, Youngseok Yoon, and Joel Brandt. Creativity support in authoring and backtracking. In *Proceedings of Workshop on Evaluation Methods for Creativity Support Environments*, pp. 1–4, 2013.

[38] Tahmid Nabi, Kyle MD Sweeney, Sam Lichlyter, David Piorkowski, Chris Scaffidi, Margaret Burnett, and Scott D Fleming. Putting information foraging theory to work: Community-based design patterns for programming tools. In *Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computing*, pp. 129–133, 2016.

[39] 中村匡秀, 井垣宏, 佐伯幸郎, 柗本真佑, 楠本真二, 上原邦昭, 井上克郎. Cloud spiral の取り組み. 日本ソフトウェア科学会第 30 回大会, 2013.

[40] Andrew Neitsch, Kenny Wong, and Michael W. Godfrey. Build system issues in multilanguage software. In *Proceedings of the 28th International Conference on Software Engineering*, pp. 140–149, 2012.

[41] Kevin Ong. Using information foraging theory to understand search behavior in different environments. In *Proceedings of the Conference on Conference Human Information Interaction and Retrieval*, pp. 411–413, 2017.

[42] Maria Paasivaara, Ville Heikkilä, Casper Lassenius, and Towo Toivola. Teaching students scrum using lego blocks. In *Proceedings of the 36th In-*

ternational Conference on Software Engineering, pp. 382–391, 2014.

[43] Maria Paasivaara, Jari Vanhanen, Ville T. Heikkilä, Casper Lassenius, Juha Itkonen, and Eero Laukkanen. Do high and low performing student teams use scrum differently in capstone projects. In *Proceedings of the 39th International Conference on Software Engineering: Software Engineering and Education Track*, pp. 146–149, 2017.

[44] Shaun Phillips, Thomas Zimmermann, Christian Bird, and Tom Zimmermann. Understanding and improving software build teams. In *Proceedings of the 36th International Conference on Software Engineering*, pp. 735–744, 2014.

[45] Sruti Srinivasa Ragavan, Bhargav Pandya, David Piorkowski, Charles Hill, Sandeep Kaur Kuttal, Anita Sarma, and Margaret Burnett. Pfis-v: Modeling foraging behavior in the presence of variants. In *Proceedings of the Conference on Human Factors in Computing Systems*, pp. 6232–6244, 2017.

[46] Jan Oliver Ringert, Bernhard Rumpe, Christoph Schulze, and Andreas Wortmann. Teaching agile model-driven engineering for cyber-physical systems. In *Proceedings of the 39th International Conference on Software Engineering: Software Engineering and Education*, pp. 127–136, 2017.

[47] Mary Beth Rosson and John M. Carroll. The reuse of uses in smalltalk programming. *ACM Transactions on Computer-Human Interaction*, Vol. 3, No. 3, pp. 219–253, 1996.

[48] David W. Sandberg. Smalltalk and exploratory programming. *ACM SIGPLAN Notices*, Vol. 23, No. 10, pp. 85–92, 1988.

[49] Hyunmin Seo, Caitlin Sadowski, Sebastian Elbaum, Edward Aftandilian, and Robert Bowdidge. Programmers' build errors: A case study (at google). In *Proceedings of the 36th International Conference on Software Engineering*, pp. 724–734, 2014.

[50] B. Sheil. *Power tools for programmers.* Readings in artificial intelligence and software engineering, 1986.

[51] Jan-Philipp Steghöfer, Eric Knauss, Emil Alégroth, Imed Hammouda,

Håkan Burden, and Morgan Ericsson. Teaching agile – addressing the conflict between project delivery and application of agile methods. In *Proceedings of the 38th IEEE International Conference on Software Engineering Companion*, pp. 303–312, 2016.

[52] Sheela Surisetti, Catherine Law, and Chris Scaffidi. Behavior-based clustering of visual code. In *Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computing*, pp. 261–269, 2015.

[53] Ryo Suzuki, Gustavo Soares, Andrew Head, Elena Glassman, Ruan Reis, Melina Mongiovi, Loris D'Antoni, and Björn Hartmann. Tracediff: Debugging unexpected code behavior using trace divergences. In *Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computing*, pp. 107–115, 2017.

[54] Haruaki Tamada, Akihiro Ogino, and Hirotada Ueda. A framework for programming process measurement and compiling error interpretation for novice programmers. In *Proceedings of the 21st International Workshop on Software Measurement and the 6th International Conference on Software Process and Product Measurement*, pp. 233–238, 2011.

[55] 独立行政法人情報処理推進機構. IT 人材白書 2017. 独立行政法人情報処理推進機構, 2017.

[56] Carnegie Mellon University. Variations to support exploratory programming. `http://www.exploratoryprogramming.org/`.

[57] Arto Vihavainen, Matti Luukkainen, and Jaakko Kurhila. Using students' programming behavior to predict success in an introductory mathematics course. In *Proceedings of the 6th International Conference on Educational Dada Mining*, pp. 300–303, 2013.

[58] Rebecca Vivian, Hamid Tarmazdi, Katrina Falkner, Nickolas Falkner, and Claudia Szabo. The development of a dashboard tool for visualising online teamwork discussions. In *Proceedings of the 37th International Conference on Software Engineering*, pp. 380–388, 2015.

[59] Christopher Watson and Frederick W.B. Li. Failure rates in introductory

programming revisited. In *Proceedings of the conference on Innovation & technology in computer science education*, pp. 39–44, 2014.

[60] YoungSeok Yoon and Brad A. Myers. An exploratory study of backtracking strategies used by developers. In *Proceedings of the 5th International Workshop on Co-operative and Human Aspects of Software Engineering*, pp. 138–144, 2012.

[61] Xiaoqiong Zhao, Xin Xia, Pavneet Singh Kochhar, David Lo, and Shanping Li. An empirical study of bugs in build process. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, pp. 1187–1189, 2014.

付録

A. 2 章に関連する付録

A.1. 課題 1 出題内容

課題 1：数字の螺旋

入力された値を N(N は 2 以上 9 以下の自然数) とした時，N×N 個の数値を，1 から順番に螺旋状に表示するプログラムを作成せよ．数値は左上端を基点に，下，右，上，左…と，外側から内側に進むにつれ増えていく．

出力例 (N ＝ 9 の時)：
1 32 31 30 29 28 27 26 25
2 33 56 55 54 53 52 51 24
3 34 57 72 71 70 69 50 23
4 35 58 73 80 79 68 49 22
5 36 59 74 81 78 67 48 21
6 37 60 75 76 77 66 47 20
7 38 61 62 63 64 65 46 19
8 39 40 41 42 43 44 45 18
9 10 11 12 13 14 15 16 17

出力例 (N ＝ 2 の時)：
1 4
2 3

課題 1 サブゴール

SG1 表示する内容を記録するための配列を作っている

SG2 表示する内容を記録するための配列を初期化している

SG3 表示する内容を記録するための配列を使っている

SG4 ループ中に数値を埋める進行方向を変更するようなプログラムを作成しようとしている

SG5 行単位で出力する値を決定しようとしている

SG6 多重ループを使用して配列に数値を格納している

SG7 配列の出力を行っている

サブゴールの設定方法

課題 1 を解くにあたって，我々は以下のモデル解法を設定した．

手順 1 数値を格納するための配列を作成する (SG1，SG2 が対応)．

手順 2 数値を格納するためのアルゴリズムを作成する

手順 3 出力用の二次元配列へ数値を格納する (SG3 が対応)

手順 4 手順 3 で格納した配列の中身を出力する (SG7 が対応)

ここで，手順 2 における数値を格納するためのアルゴリズムは様々なものが考えられるため，第二著者と相談し，初学者が行う可能性のあるアルゴリズムとして以下の 3 つを設けた．

- ループと二次元配列を用いて，下，右，上，左…のように，螺旋状に数値をインクリメント，あるいはデクリメントしながら二次元配列へ格納するアルゴリズム (SG4 が対応)
- 行単位で格納する数値を計算．出力していくアルゴリズム (SG5 が対応)
- 多重ループと条件分岐を用いて数値を格納するアルゴリズム (SG6 が対応)」

本実験においては，望ましい出力が得られなかった場合でも上記アルゴリズムを使用した時点でサブゴールを達成したものと判断している．

A.2.　課題 2 出題内容

課題 2 : Sum of 4 Integers

50 以下の正の整数 n を入力し，0〜9 の範囲の整数 a，b，c，d の組みで

$a + b + c + d = n$

を満たすものの組み合わせ数を出力するプログラムを作成せよ.

例えば，n が 35 のとき，(a，b，c，d) は (8，9，9，9)，(9，8，9，9)，(9，9，8，9)，(9，9，9，8) の 4 通りなので，答えは 4 となる.

B. 3 章に関連する付録

B.1. 年度別学生ごとのビルドエラー内訳

各グラフの上部がリモートビルドのエラー数，下部がローカルビルドのエラー数を示す.

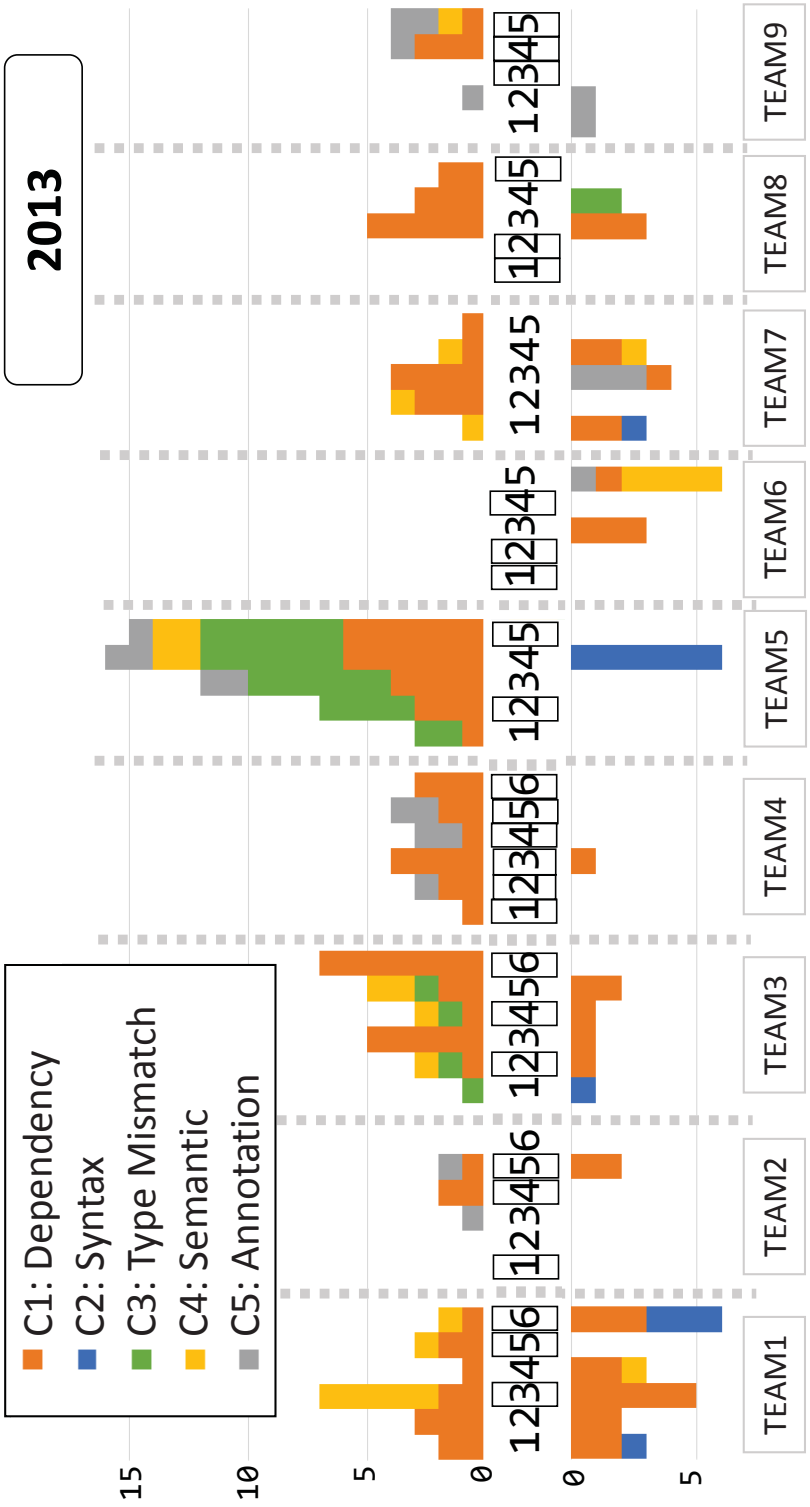

2013
C1: Dependency
C2: Syntax
C3: Type Mismatch
C4: Semantic
C5: Annotation
TEAM1
TEAM2
TEAM3
TEAM4
TEAM5
TEAM6
TEAM7
TEAM8
TEAM9
15
10
5
0
0
5

図 A.1　2013 年度

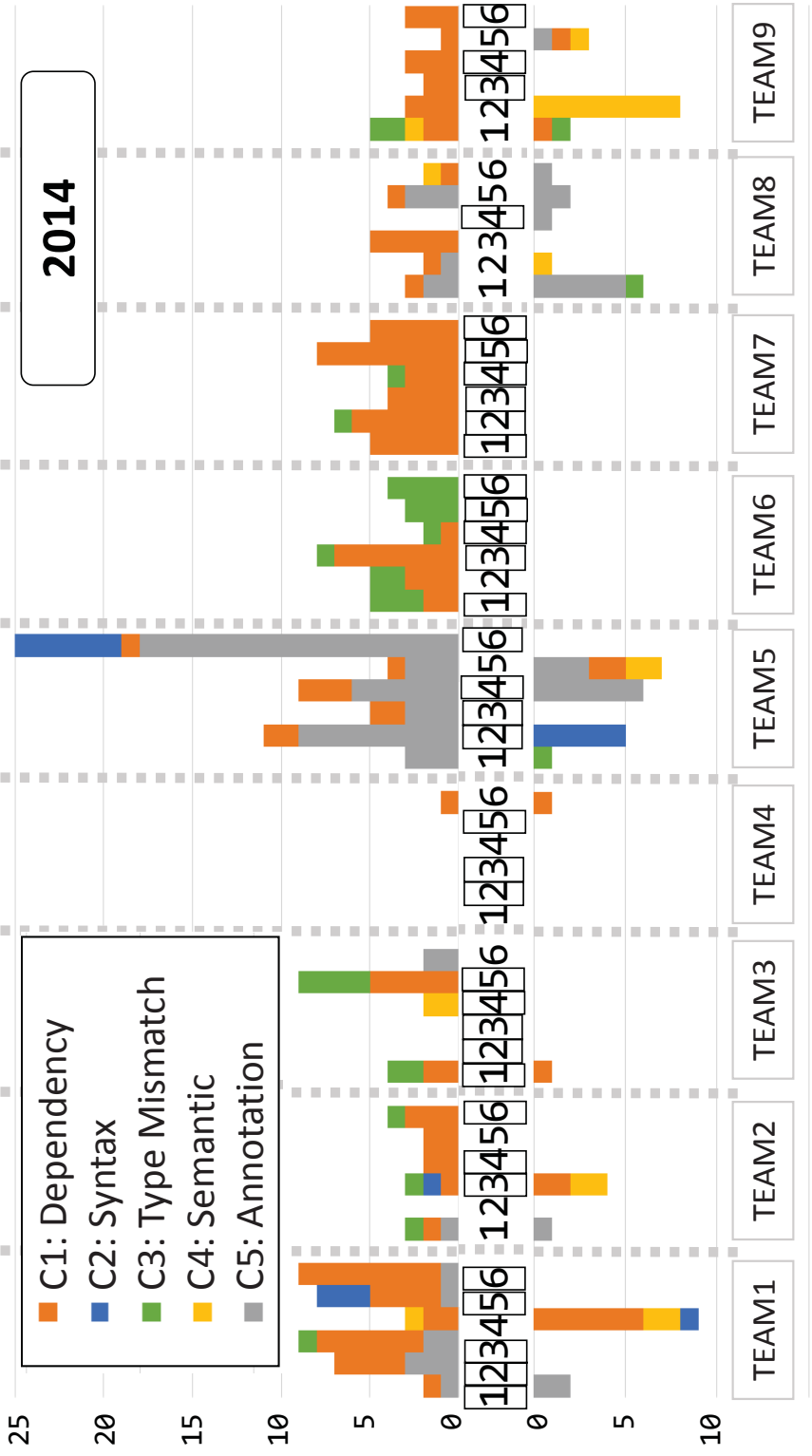

2014
C1: Dependency
C2: Syntax
C3: Type Mismatch
C4: Semantic
C5: Annotation
TEAM1
TEAM2
TEAM3
TEAM4
TEAM5
TEAM6
TEAM7
TEAM8
TEAM9
1 2 3 4 5 6
25
20
15
10
5
0

図 A.2　2014 年度

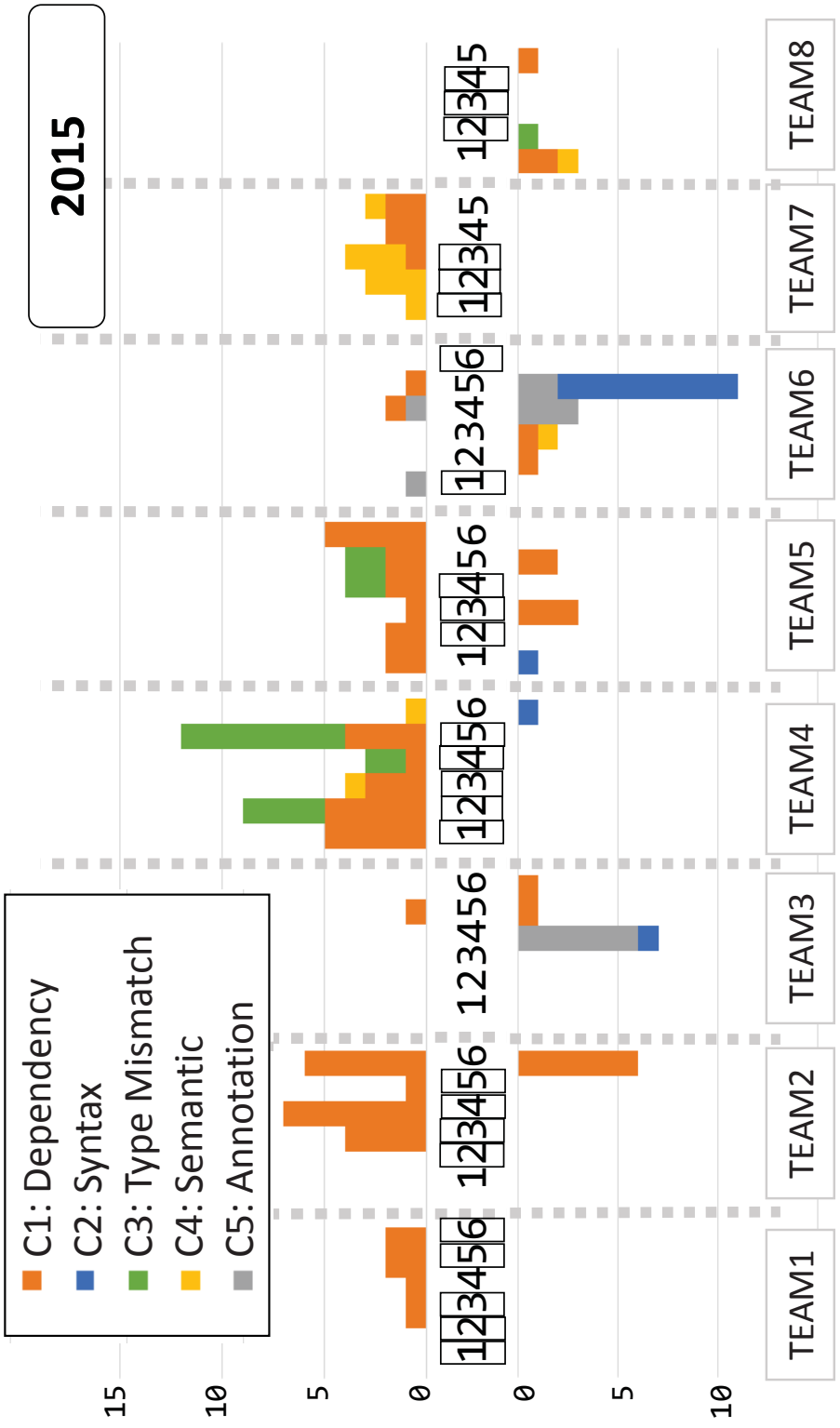

2015
C1: Dependency
C2: Syntax
C3: Type Mismatch
C4: Semantic
C5: Annotation
TEAM1
TEAM2
TEAM3
TEAM4
TEAM5
TEAM6
TEAM7
TEAM8
15
10
5
0
0
5
10

図 A.3　2015 年度

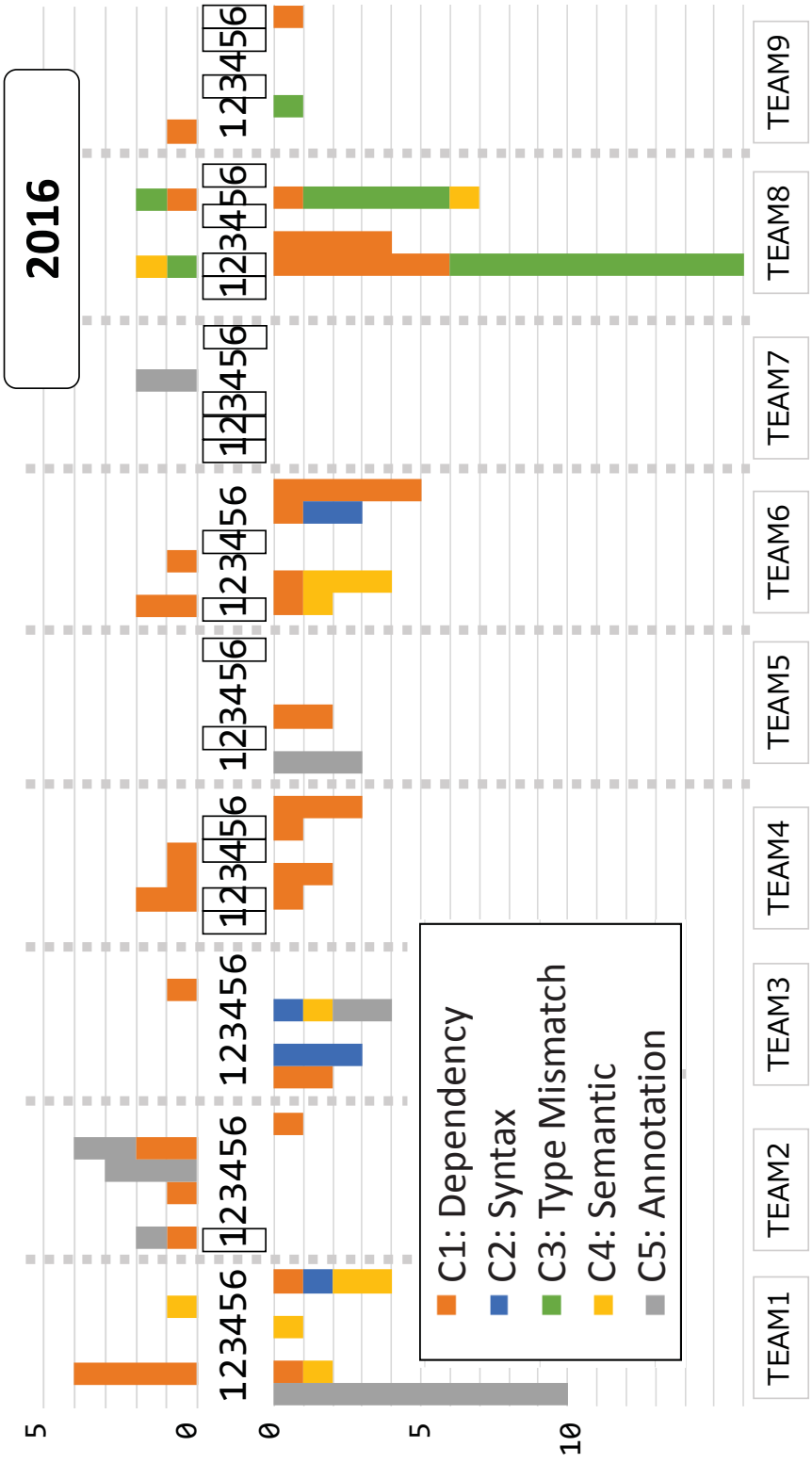

2016
TEAM1
TEAM2
TEAM3
TEAM4
TEAM5
TEAM6
TEAM7
TEAM8
TEAM9
C1: Dependency
C2: Syntax
C3: Type Mismatch
C4: Semantic
C5: Annotation
1 2 3 4 5 6
5
0
0
5
10

図 A.4　2016 年度

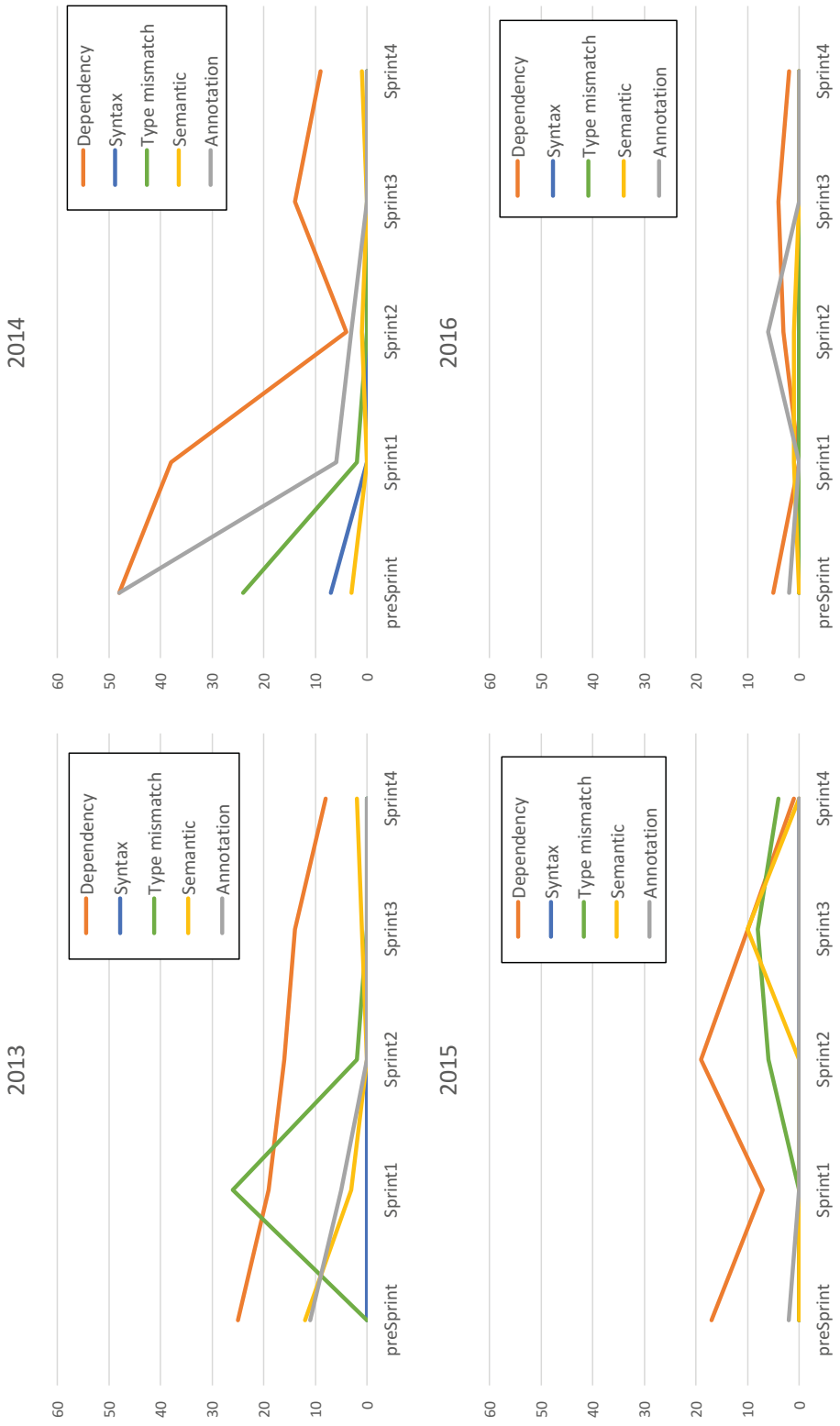


図 A.5　各年度の合宿におけるビルドエラーの遷移

B.2. 各種エラーの詳細な分類

表 A.1: ビルドエラーの詳細な分類

分類	略称	エラーメッセージの例
Dependency	cant.resolve	シンボルを解決できません
	doesnt.exist	XXX は存在しません
Syntax	expected	識別子がありません
	not.stmt	文がありません
	missing.ret.stmt	return がありません
	illegal.start.of.expr	式の開始が不明です
	illegal.end	文字リテラルの行末が不正です
	illegal.start.of.type	型の開始が不正です
	illegal.character	XXX は不正な文字です
	missing.semi.stmt	セミコロンがありません
TypeMismatch	cant.apply.symbol	シンボルが見つかりません
	incompatible	この型は XXX には不適合です
	type.List.does.not.take.para	型 XXX はパラメータをとりません
	no.suitable.constructor	適切なコンストラクタが存在しません
	unchecked	XXX から YYY への代入は未確認です
	no.suitable.method	適切なメソッドが存在しません
Semantic	exception.doesnot.throw	例外 XXX をスローできません
	not.def.public.cant.access	パッケージ外からはアクセスできません
	unreported.exception	報告されない例外 XXX が存在します
	should.be.declared	ファイル XXX で宣言しなくてはいけません
	already.defined	定義済みです
	class.is.public	ファイル XXX は public です
	not.initialized	初期化されていません
	static.imp	static インポートを使用できません
	duplicate.class	クラス名が重複しています
	statement.not.reached	この文に制御が移ることはありません
	non-static.cant.be.ref	static でない XXX を static コンテキストから参照することはできません
	cannot.be.accessed	アクセスできません
	catch.without.try	catch への try がありません
	no.suitable.identifier	適切な識別子が存在しません
	identifier.expected	識別子がありません
Annotation	disallowed.annotation	The annotation @XXX is disallowed for this data type
	may.not.be.annotated	Void methods may not be annotated with constraint annotations.

表 A.2　ビルドエラーの詳細な分類

エラーメッセージ	2013	2014	2015	2016	2013	2014	2015	2016
[*DE*]cant.resolve	13	13	10	26	67	114	55	16
[*DE*]doesnt.exist	21	1	8	8	19	4	9	2
[*SY*]expected	6	2	6	2	0	8	0	0
[*SY*]not.stmt	1	1	1	0	0	1	0	0
[*SY*]missing.ret.stmt	0	0	0	1	0	1	0	0
[*SY*]illegal.start.of.expr	3	1	2	3	0	0	0	0
[*SY*]illegal.end	1	1	0	0	0	0	0	0
[*SY*]illegal.start.of.type	1	0	3	0	0	0	0	0
[*SY*]illegal.character	0	1	0	0	0	0	0	0
[*SY*]missing.semi.stmt	0	0	0	1	0	0	0	0
[*TM*]cant.apply.symbol	2	1	1	4	28	17	18	0
[*TM*]incompatible	0	1	0	3	0	4	0	2
[*TM*]type.List.does.not.take.para	0	1	0	0	0	3	0	0
[*TM*]no.suitable.constructor	0	0	0	0	0	3	0	0
[*TM*]unchecked	0	0	0	8	0	0	0	0
[*TM*]no.suitable.method	0	0	0	1	0	0	0	0
[*SE*]exception.doesnot.throw	4	0	0	0	5	0	7	0
[*SE*]not.def.public.cant.access	0	0	0	1	9	2	0	0
[*SE*]unreported.exception	0	7	0	0	5	0	3	0
[*SE*]should.be.declared	0	0	0	0	0	0	0	2
[*SE*]already.defined	1	2	0	1	0	1	0	0
[*SE*]class.is.public	0	3	1	0	0	1	0	0
[*SE*]not.initialized	0	1	0	1	0	1	0	0
[*SE*]static.imp	1	0	1	0	0	0	0	0
[*SE*]duplicate.class	0	1	0	0	0	0	0	0
[*SE*]statement.not.reached	0	1	0	0	0	0	0	0
[*SE*]non-static.cant.be.ref	0	1	0	0	0	0	0	0
[*SE*]cannot.be.accessed	0	0	0	4	0	0	0	0
[*SE*]catch.without.try	0	0	0	1	0	0	0	0
[*SE*]no.suitable.identifier	0	0	0	1	0	0	0	0
[*SE*]identifier.expected	0	0	0	1	0	0	0	0
[*AN*]disallowed.annotation	6	22	11	14	11	59	2	8
[*AN*]may.not.be.annotated	0	0	0	1	5	0	0	0

B.3. ローカルビルドの詳細な分類

図 A.6 各年度の合宿におけるビルドエラーの遷移

B.4.　リモートビルドの詳細な分類

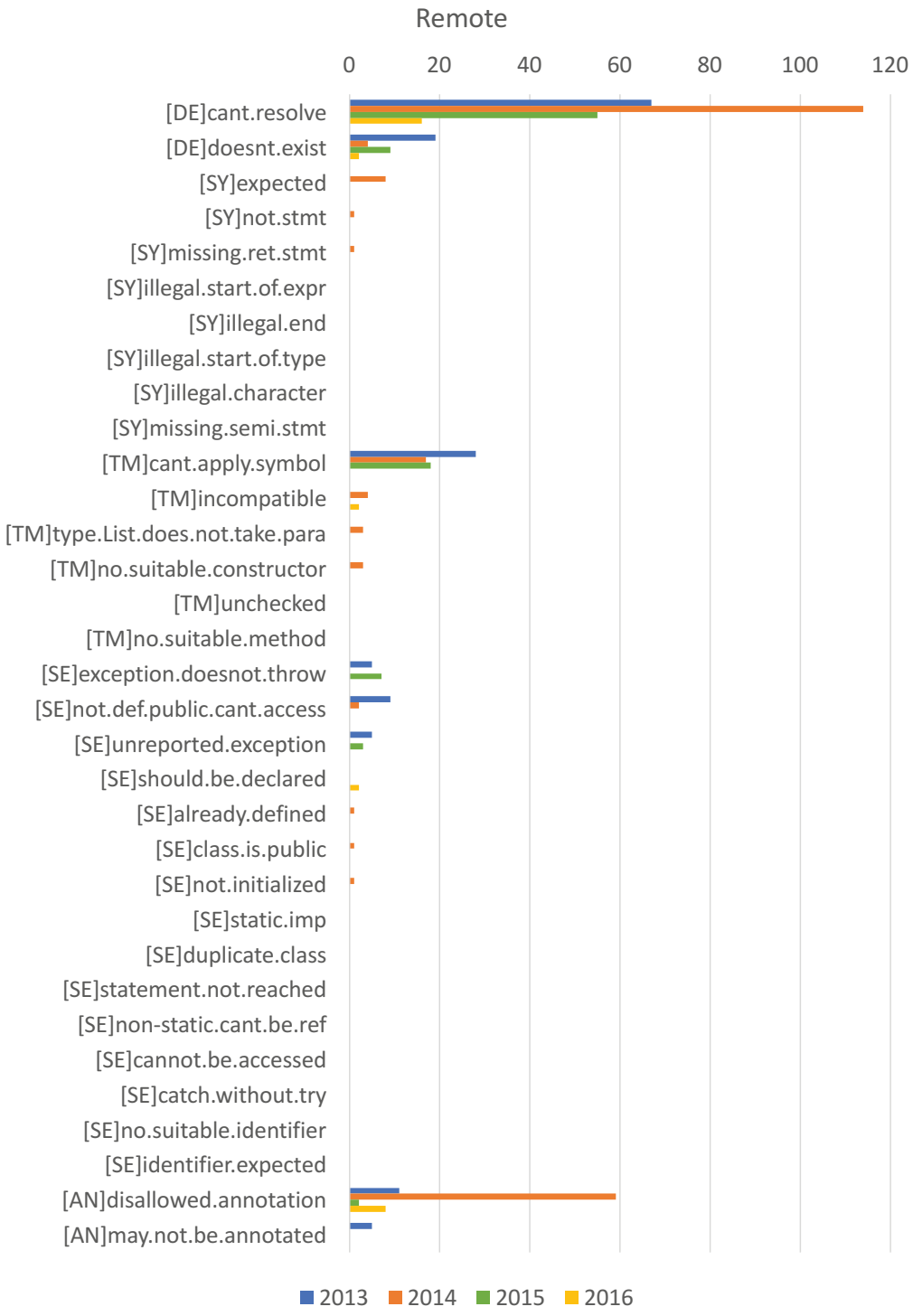


図 A.7　各年度の合宿におけるビルドエラーの遷移