修士学位論文

題目

開発作業のモニタリングによる
メソッド抽出リファクタリング支援環境の構築

指導教員
井上 克郎 教授

報告者
沼田 聖也

平成 30 年 2 月 7 日

大阪大学 大学院情報科学研究科
コンピュータサイエンス専攻 ソフトウェア工学講座

平成 29 年度 修士学位論文

開発作業のモニタリングによるメソッド抽出リファクタリング支援環境の構築

沼田 聖也

内容梗概

ソフトウェア開発にかかるコストを増大させている要因の 1 つとして，ソースコード中のコードクローンが挙げられる．コードクローンとは，ソースコード中に存在する同一，あるいは類似したコード片を意味し，主に既存のコード片のコピーアンドペーストによって生成される．

コードクローンに関する保守作業の 1 つとしてソースコードの同時修正が挙げられる．例えば，あるコード片に欠陥が見つかった場合，そのコード片のコードクローンにも同様の欠陥が含まれる可能性が高い．そのため，開発者はコードクローンに欠陥が見つかった場合，同一クローンセット (互いにコードクローンとなっている集合) 内に含まれる全てのコードクローンに対して，同一の修正をするか検討する必要がある．しかし，大規模プロジェクトに対して，開発者がすべてのコードクローンを手作業で見つけて保守するのは困難である．

コードクローンに対する保守コストを削減する方法の 1 つとして，コードクローンの集約が挙げられる．コードクローンの集約とは，ソースコード中に存在する同一クローンセット内のコードクローンを，メソッド抽出リファクタリングを用いて単一のメソッドにまとめることである．コードクローンの集約を行うことによって，コードクローンの保守にかかるコストを削減することができる．

コードクローンの集約支援をするためには，優先して集約すべきコードクローンを提示するだけではなく，開発者の作業内容に応じて適切な時期に集約を支援することが求められる．その理由は，開発者の作業内容に関連した集約候補を推薦すると，開発者は作業内容に関する記憶をたどりながら効率的にコードクローンの集約を行えるからである．また，開発者はコードクローンの存在を意識せずに，メソッド抽出リファクタリングを行うことがある．そのような場合，同時にメソッド抽出リファクタリングを行うべきコードクローンを見落としてしまうという問題点も挙げられる．

そこで本研究では，メソッド抽出リファクタリングに着目し，統合開発環境 Eclipse 上で行われた開発作業を分析することで，コードクローンの集約を支援する環境を構築する．具体的には，開発作業からメソッド抽出リファクタリングの編集パターンを見つけ，抽出さ

れたコード片がコードクローンの場合，そのコード片のコードクローンを開発者に提示し，同時に集約を検討することを促す．これにより，開発者はメソッド抽出リファクタリングを行ったその場でコードクローンの存在を知り，提示されたコードクローンに対して同時集約作業を検討することができる．

また，本研究では，提案支援環境のコードクローンの集約支援における有効性を調査するために評価実験を行った．具体的には，提案支援環境のコードクローン検出部分に用いている，CCFinderX の GUI である GemX との比較実験を行い，コードクローンの集約にかかる時間とその精度を比較したところ，その有意差を確認することができた．

主な用語

コードクローン
メソッド抽出リファクタリング
コードクローンの集約

目 次

1　まえがき

ソフトウェア開発にかかるコストを増大させている要因の 1 つとして，ソースコード中のコードクローンが挙げられる．コードクローンとは，ソースコード中に存在する同一，あるいは類似した部分を持つコード片を意味し，主に既存のコード片のコピーアンドペーストによって生成される [1][2][8]．一般的に，互いにコードクローンになるコード片のことをクローンペアと呼び，互いにコードクローンになるコード片の集合のことをクローンセットと呼ぶ．

コードクローンに関する保守作業の 1 つとしてソースコードの同時修正が挙げられる．例えば，あるコード片に欠陥が見つかった場合，同一クローンセット内に含まれる全てのコードクローンにも同様の欠陥が含まれる可能性が高い [21]．そこで，開発者はコードクローンに欠陥が見つかった場合，同一クローンセット内に含まれる全てのコードクローン対して同様の修正を検討する必要があるが，すべてのコードクローンを開発者が手作業で見つけて保守するのは困難である．そのため，開発者はコードクローンの保守作業のためにコードクローン検出ツールを利用する．現在までに，ソースコードからコードクローンを自動検出する手法は数多く提案されている [10]．

コードクローンに対する保守コストを削減する方法の 1 つとして，コードクローンのリファクタリングが挙げられる [9][14]．コードクローンに対するリファクタリングとして，コードクローンの集約がある．コードクローンの集約とは，メソッド抽出リファクタリングを用いて同一クローンセット内のコードクローンを単一のメソッドにまとめることである [9]．コードクローンの集約を適切に行うことで，コードクローンの保守にかかるコストを予防することができる．

これまでに，ソースコード中からメソッド抽出リファクタリング対象のコードクローンを自動抽出することで，コードクローンを対象とした集約支援を行う手法が提案されている [3][26][27]．コードクローンの集約支援手法には，優先して集約すべきコードクローンを提示できるだけでなく，開発者の作業内容に応じて適切な時期に集約を支援することが求められる．その理由は，開発者の作業内容に関連した集約候補を推薦すると，開発者は作業内容に関する記憶をたどりながら効率的にコードクローンの集約を行えるからである．また，テストまで完了し，異なる作業を開始したコード片を集約候補として推薦しても，そのコード片に関する開発者の記憶が曖昧になっている．また，そのコード片に対して集約を行った場合，その集約の妥当性を確認するためのテストをやり直す必要があるため，コードクローンの集約にかかるコストが大きくなってしまうという問題点が挙げられる [23][25]．さらに，開発者はコードクローンの存在を意識せずに，メソッド抽出リファクタリングを行うことがある．そのような場合，同時にメソッド抽出リファクタリングを行うべきコードクローンを

見落としてしまうという問題点も挙げられる．しかし，我々の知る限り，今まで提案されたコードクローンの集約支援では，開発者の作業内容を考慮せずに，コードクローンの集約支援を行っている．

この問題を解決するためには，開発者の作業内容に応じて適切な時期にコードクローンの集約を支援することが必要である．本研究では，メソッド抽出リファクタリングに着目し，開発者の作業内容に応じて，適切な時期にコードクローンのリファクタリングを支援する環境を構築する．具体的には，統合開発環境 Eclipse 上での開発者のソースコードの編集作業をモニタリングする．そして，開発者がメソッド抽出リファクタリングを行ったことを検知すると，そのメソッド抽出リファクタリングを行ったコード片のコードクローンを開発者に提示し，同時に集約を検討することを促す．これにより，開発者はその場でコードクローンの存在を知り，提示されたコードクローンに対して集約作業を検討することができる．

本研究では，提案支援環境のコードクローンの集約支援における有効性を調査するために評価実験を行った．具体的には，提案支援環境のコードクローン検出部分に用いている，CCFinderX の GUI である GemX との比較実験を行った [7][15]．被験者実験を行い，両者でコードクローンの集約にかかる時間とその精度を比較したところ，提案支援環境のコードクローンの集約支援における有効性を確認することができた．

以降，2 章では，本研究に関わるコードクローンおよびリファクタリングの関連研究について説明する．3 章では，メソッド抽出リファクタリングの集約パターンを検出し，そのコードクローンを開発者に提示する提案環境を提案する．4 章では，提案環境の利用シナリオを利用例を用いて説明する．そして，5 章では，評価実験について説明し，6 章では，まとめについて述べる．

2　背景

2.1　コードクローン

コードクローンとは，ソースコード中に存在する同一，あるいは類似した部分を持つコード片を意味する．一般的に，このコードクローンの存在は，ソフトウェアの保守を困難にすると言われている [1][2][8]．また，互いにコードクローンになるコード片の対のことをクローンペアと呼び，クローンペアにおいて推移関係が成り立つコードクローンの集合のことをクローンセットと呼ぶ．

コードクローンに関する保守作業の 1 つとしてソースコードの同時修正が挙げられる．あるコード片に欠陥が見つかった場合，そのコード片のコードクローンにも同様の欠陥が含まれる可能性が高い [21]．そこで，開発者はコードクローンに欠陥が見つかった場合，同一クローンセット内に含まれる全てのコードクローン対して同様の修正を検討する必要があるが，すべてのコードクローンを開発者が手作業で見つけて保守するのは困難である．そのため，開発者はコードクローンの保守作業のためにコードクローン検出ツールを利用する．現在までに，ソースコードからコードクローンを自動検出する手法は数多く提案されている [10]．

2.1.1　コードクローンの発生要因

コードクローンがソフトウェア中に作りこまれる，あるいは発生する要因として次のようなものが挙げられる [2][8]．

既存コード片のコピーアンドペーストによる再利用

一からソースコードを書くよりも既存コード片を再利用して部分的な変更を加えることが多くなったために，コピーアンドペーストによる既存コード片の再利用が存在する．

定型処理

キューの挿入処理や，データ構造アクセス等，定義上簡単で頻繁に用いられるような処理がコードクローンとなることが多い．

プログラミング言語における適切な機能の欠如

抽象データ型や，ローカル変数を用いることができない場合に，似たようなアルゴリズムを持つ処理を繰り返し書かなくてはならなくなり，コードクローンとなる可能性がある．

パフォーマンスの改善

リアルタイムシステム等の時間制約のあるシステムにおいて，インライン展開等の機

能が提供されていない場合に，特定のコード片を意図的に繰り返し記述することで，パフォーマンスの改善を図ることにより，コードクローンとなる可能性がある．

コード生成ツールの生成コード

コード生成ツールにおいて，類似した処理を目的としたコードの生成には，識別子等の違いはあるが，あらかじめ決められたコードをベースにして自動生成されるため，コードクローンが生成される．

偶然の一致

偶然に，開発者が同一のコード片を書くことでコードクローンとなる場合がある．

2.1.2 コードクローンのタイプ分類

コードクローンには普遍的な定義は存在しない．本論文では，コードクローンの定義として以下の 2 つのタイプの分類を用いる [20]．

タイプ 1

空白の有無，レイアウト，コメントの有無などの違いを除き完全に一致する．

タイプ 2

タイプ 1 の違いに加えて，変数名などのユーザ定義名，関数の型などが異なる．

2.1.3 コードクローンの検出

コードクローン検出における粒度はいくつかある [10]．以下にその例を挙げる．

文字

プログラムテキストの構成要素を文字列とみなして，文字列のパターンマッチングを行い，コードクローンを発見する．この方法は，空白等も含めて厳密に同形のコード片しか検出されず，また，文字列レベルでの等価判定は，手間がかかり，大規模なプログラムの解析には向かないため，文字列レベルの比較を行う検出ツールは無い．

字句 (トークン)

プログラムの字句解析を行った後，その字句を要素とした系列に対してコードクローンを発見する．字句解析を行うことにより，空白やコメントを無視することができ，識別子や定数等の特定の種類の字句を特殊な 1 つの字句に固定することで，変数名や関数名の変更されたコード片もコードクローンとして検出することができる．

行

字句よりも粗い粒度にして，プログラムのテキストの各行をハッシュ関数を用いてハッ

シュ値に変換し，そのハッシュ値の列を対象として，コードクローンを発見する [4]．大きなプログラムテキストを比較的小さな要素列に圧縮できるため，効率良くコードクローンを見つけることができるが，空行の削除や挿入，改行位置の変更によって，検出できなくなる場合があるので，あらかじめ空行の除去等が必要である．

文

プログラムテキスト中の文を取り出して，それをハッシュ関数で 1 つの要素にし，その系列に対してコードクローンを発見する [18]．この方法では，文の認識のために簡単な字句解析と構文解析が必要だが，空白やコメント等の影響は受けない．変数名等の変更に対応するためには，識別子等をパラメータ化する必要がある．

関数

プログラムテキストの関数を 1 つの要素とし，等価な要素対を見つけることでコードクローンを発見する [24]．この方式では，関数全体ではなく，一部のみがコードクローンになっているものを発見することはできない．コードクローン検出のために，プログラムの性質を計測した特徴メトリクスを用いる．

2.1.4 コードクローン検出ツール CCFinderX

CCFinder は，字句単位のコードクローンを検出するツールであり，CCFinderX はその CCFinder を拡張したものである [15]．字句単位のコードクローン検出では，入力されたプログラムの字句解析を行った後，その字句を要素とした系列に対してコードクローンを検出する．CCFinderX は，字句解析を行うことにより，空白やコメントを無視することができる．また，識別子や定数等の特定の種類の字句を特殊な 1 つの字句に固定することで，変数名や関数名の変更されたコード片もコードクローンとして検出することができる．以下に CCFinderX のコードクローン検出手順を示す．CCFinderX は大きく 4 つのステップを経てコードクローンを検出する．

ステップ 1(字句解析)

ソースファイルを字句解析することでトークン列に変換する．入力ファイルが複数ある場合には，個々のファイルから得られたトークン列を連結し，単一のトークン列を生成する．

ステップ 2(変換処理)

実用上で意味を持たないコードクローンを取り除くこと，および，些細な表現上の違いを無視することを目的とした変換ルールによりトークン列を変換する．この変換に

より，変数名は同一のトークンに置換されるため，変数名が違うコード片もコードクローンとして判定することができる．

ステップ 3(検出処理)

トークン列の中から指定された長さ以上一致している部分をクローンペアとしてすべて検出する．

ステップ 4(出力整形処理)

検出されたクローンペアについて，元のソースコード上での位置情報を出力する．

CCFinderX は上記の 4 ステップにより，タイプ 1 とタイプ 2 のコードクローンを検出する．CCFinderX はコードクローンを高速で検出することができるため，多くの企業や研究で使用されている．

2.1.5 コードクローン分析環境 GemX

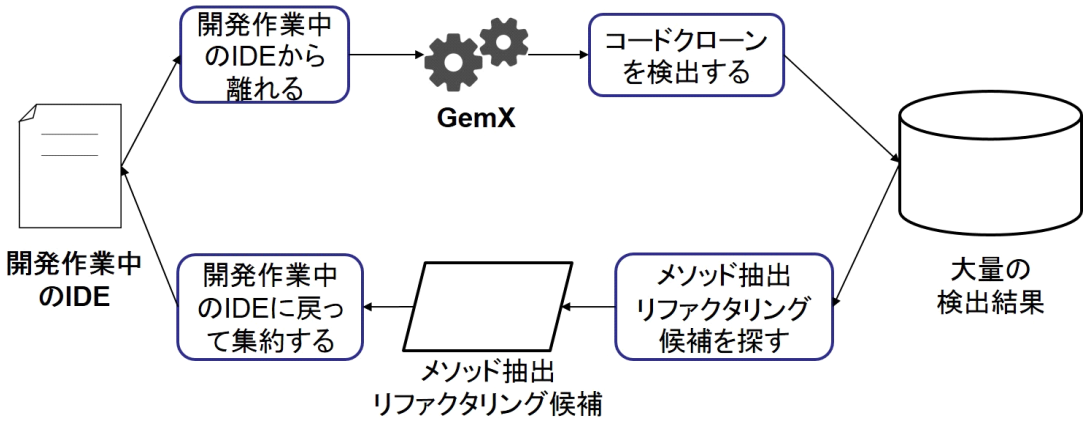

図 1: GemX によるコードクローンの集約支援

コードクローン分析環境 GemX は CCFinderX の GUI であり，CCFinderX が検出したコードクローンを分析することができる [7]．GemX を用いてコードクローンの集約支援を行うと，図 1 のような流れになる．開発者は開発中の統合開発環境 (IDE) から離れて，GemX を用いて対象プロジェクトからコードクローンの検出を行う．そして，コードクローンの検出結果からメソッド抽出リファクタリング候補を探し，再び開発作業中の IDE に戻ってコードクローンの集約を検討するという流れになる．

しかし，GemX は非常に多くのコードクローンを検出するため，開発者が検出されたコードクローンの中から同時集約候補を探すのは容易ではない．また，GemX は多くの開発者が

ソースコード開発を行うために使用する，IDE 上で使用できないという問題点も挙げられる．そのため，開発者は開発作業と並行してコードクローンの集約を行うために，現在の開発作業を中止して，GemX を開いて実行する必要があり，非常に手間がかかる作業となる．

2.1.6　コードクローン変更管理システム Clone Notifier

Clone Notifier は，版管理システムの 2 つのバージョン間のコードクローンの変更情報を得ることができるツールである [23][25]．図 2 にその処理の流れを示す．コードクローンの変更情報とは，前のバージョンではコードクローンでなかったものが，新しいバージョンではコードクローンになったり，逆に，前のバージョンではコードクローンであったものが，新しいバージョンではコードクローンでは無くなった等の，2 つのバージョン間のコードクローンがどう変更されたかに関する情報である．このコードクローンの変更情報を得ることにより、開発者はコードクローンに対して集約を検討することができ，ソフトウェア開発の保守性を高めることができる．

しかし，図 2 で表すように Clone Notifier は開発者が版管理システムにコミットしたソースコードに対して，コードクローンの変更情報を分析する．一般的に，開発者はテストまで終えたソースコードを版管理システムへコミットすると考えられる．そのため，版管理システムにコミットされたソースコードから集約対象のコードクローンを見つけた場合，作業の後戻りが必要になり，また，そのソースコードに関する記憶も薄れていることが多いため，Clone Notifier の変更情報から得たコードクローンを集約するコストが大きいという問題点が挙げられる．

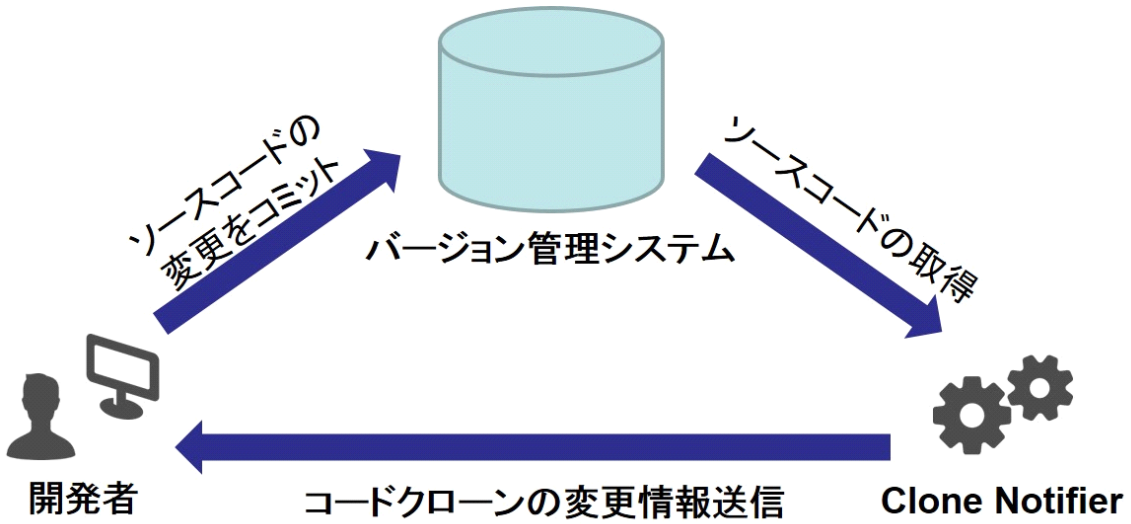

図 2: Clone Notifier の処理の流れ

2.2 リファクタリング

リファクタリングとは，ソフトウェアの外部的ふるまいを保ったままで，内部の構造を改善していく作業を指す [6]．リファクタリングを適切に行うことで，ソフトウェアの保守性を高めることができる．Fowler はリファクタリングを実施すべきソースコードの特徴をまとめており，その中の 1 つにコードクローンが挙げられる．本研究では，リファクタリングパターンの内，メソッド抽出リファクタリングに着目し，コードクローンの同時集約支援を目指している．

2.2.1 メソッド抽出リファクタリング

メソッド抽出リファクタリングとは，ひとまとめにできるコード片を新たなメソッドとして定義し，抽出されたコード片を抽出先のメソッドへの呼び出し文に置き換えるリファクタリングである．メソッド抽出リファクタリングの主な目的は，長すぎるメソッドの分割や，複数の機能が実装されたメソッドの分割である．メソッドを適切なサイズに分割することで可読性を向上させることができ，さらに，機能追加やバグ修正が容易になるという利点が挙げられる．メソッド抽出リファクタリングは，開発者が頻繁に行うリファクタリングの 1 つであることが分かっている [16]．メソッド抽出リファクタリングの例を図 3 に示す．図 3 では，printOwing メソッドの明細表示部分のコード片を，新たに printDetails というメソッドとして定義し，メソッド呼び出ししている．メソッド抽出リファクタリングは Fowler の定義に従い，一般的に以下の手順で行われる [6]．

メソッド抽出リファクタリングの手順 (Fowler の定義)

1. 新たなメソッドを作成し，そのメソッドが何をするのか，メソッドの意図に合わせて命名する．
2. 抽出されるコードを元のメソッドから新たな抽出先のメソッドにコピーする．
3. 抽出元のメソッドにローカルスコープ変数が存在した場合，抽出後に振る舞いが変わらないか確認し，新たなメソッドにパラメータとして渡す．
4. 元のメソッドにおいて，抽出されたコード片を抽出先メソッドへのメソッド呼び出し文に置き換える．
5. コンパイルしてテストする．

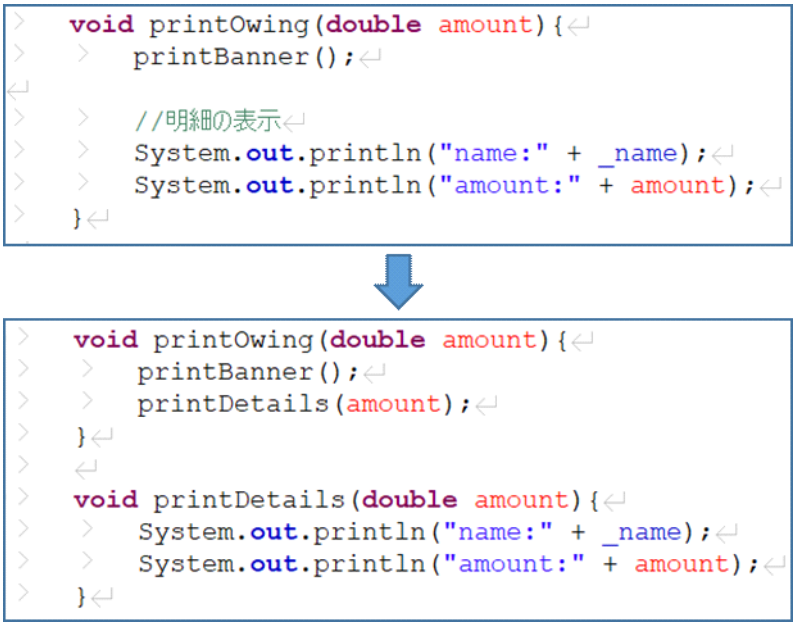

図 3: メソッド抽出リファクタリングの例 [6]

2.2.2 メソッドのインライン化

メソッドのインライン化とは，メソッドの本体が，名前を付けて呼ぶまでもなく明らかであるような場合に，メソッドの呼び出し元をメソッドの本体に置き換え，そのメソッドを除去するリファクタリングである [6]．メソッドのインライン化は，間接化しすぎた結果，すべてのメソッドが別のメソッドへと単純に委譲しているようにしか見えないようなときに適用される．本研究では，評価実験のデータセットの作成の際に，メソッドのインライン化を適用した．

2.2.3 リファクタリング支援ツール WitchDoctor

WitchDoctor とは，手動で行われているリファクタリングを進行中に検出し，自動でリファクタリングを完遂するツールである [5]．WitchDoctor はすべてのキー入力を追い，キー入力が起こるたびにプログラム行差分を取得する．そして，差分片と AST(抽象構文木) を対応付けし，差分の AST ノードを取得する．差分の AST ノードを蓄積していき，開発者の進行中の操作が特定のリファクタリング操作パターンと一致しているかどうかを検知する．最終的に，リファクタリング操作を検知すると，進行中の操作を元に戻し，自動でリファクタリングを適用する．しかし，このように自動でリファクタリングを行うことで，開発者が思いもしないソースコードになることもあるため，自動リファクタリングツールを好まない

開発者も多い．そこで，本研究では，自動でリファクタリングを行うことはせず，リファクタリングの候補を開発者に提示し，リファクタリングの実施は開発者の意思に任せるという方針を取った．本研究では，メソッド抽出リファクタリングの集約パターンを検出する際に，WitchDoctor の手法を参考にした．

3　提案環境

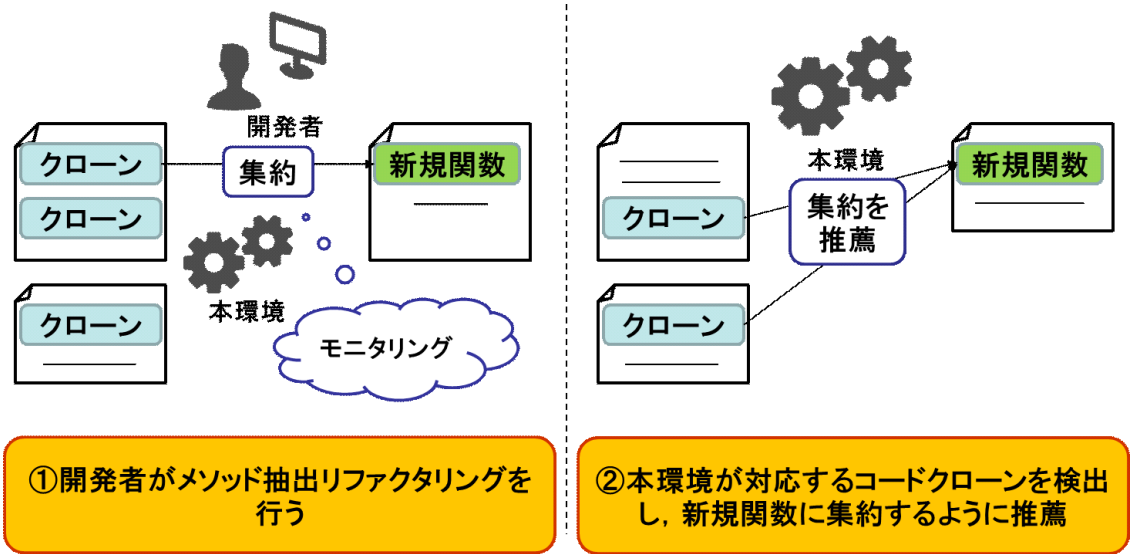

図 4: 提案環境の概要

3.1　研究動機

ソースコードからコードクローンを自動検出する手法は既に数多く提案されてきた [12][15][24]．しかし，現在まで提案された多くのコードクローンの保守作業支援手法は，開発者の現在の作業から離れてソースコードに対してコードクローンを検出し，保守支援を行うものが多い．Clone Notifier のように版管理システムとの連携を行う手法はいくつか存在するが，作業内容を分析することで作業内容に応じたコードクローンの保守作業を支援する手法は我々が知る限り存在しない [22]．版管理システムのソースコードのように，テストが完了したソースコード中のコードクローンの存在を知り，コードクローンの集約作業を行うと，再びテストを行う必要があり，開発コストが増大してしまうことがある．また，開発者が全てのコードクローンの存在を把握せずに開発作業を行うことによって，同時に集約できるコード片があることに気づくことができないという可能性も考えられる．そこで本研究では図 4 のように，開発者が全てのコードクローンの存在を把握する必要なく，開発者の開発作業をモニタリングすることによって，メソッド抽出リファクタリングの集約パターンを検知して，その抽出したコード片のコードクローンを開発者に集約の対象として推薦する手法を提案する．これにより，開発者の現在の作業から離れてコードクローン検出を行う必要なく，効率的にソフトウェアの保守作業が行われることが期待できる．

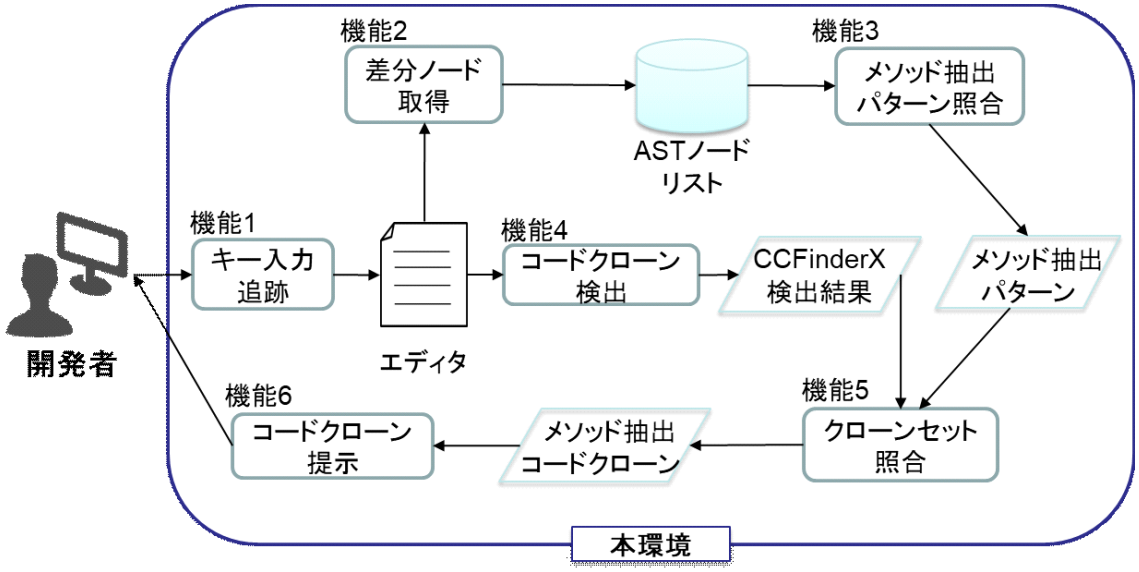

図 5: 提案環境の全体図

3.2 提案環境が実現する支援手法

開発作業のモニタリングによりメソッド抽出リファクタリングの集約パターンを検知し，その抽出したコード片のコードクローンを開発者に提示する提案環境について説明する．提案環境は Eclipse の Plugin として実装した．提案環境の全体図を図 5 に示す．図 5 に示す通り，提案環境は機能 1 から 6 までを有している．それぞれの機能の詳細は，3.2.1 節から 3.2.6 節で述べる．

3.2.1 キー入力追跡

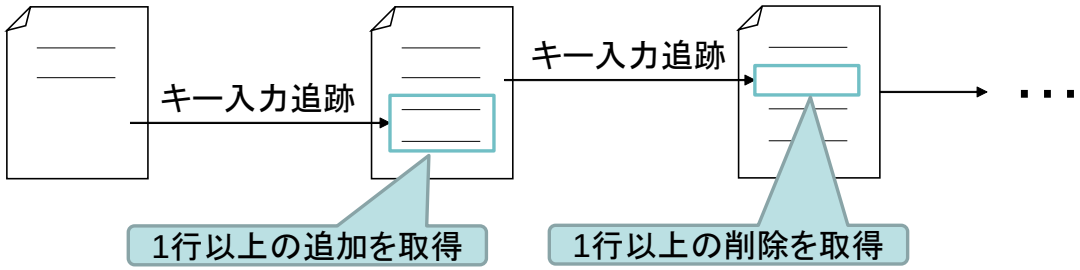

図 6: キー入力追跡

開発者の開発作業のモニタリングを行うために，提案環境では，開発者のキー入力の追跡

を行う．メソッド抽出リファクタリングを行う際に必要な情報は，1 行以上の行差分情報である．そこで，図 6 のようにキー入力を追跡し，開発者のキー入力によって 1 行以上の変更が起こるたびに行差分を取り，1 行以上の変更を検知できるようにした．行差分を取る際には，Myers の差分検出アルゴリズムを適用した [17]．

3.2.2 差分ノード取得

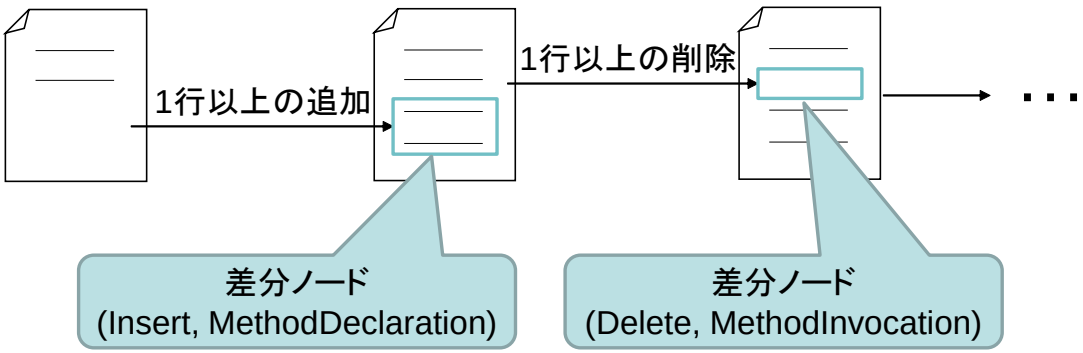

図 7: 差分ノード取得

ソースコードに対して 1 行以上の変更があった場合，その差分を AST のノードとマッピングし，取得する．AST の構築には Eclipse の ASTParser[19] を使用する．差分は組 (delta_type, ast_node) の集合から構成される．delta_type が取る値は Insert あるいは Delete のいずれかである．Insert は 1 行以上の行が挿入したことを表し，Delete は 1 行以上の行が削除されたことを表す．ASTParser を利用して作られた AST は，ASTVisitor クラスを実装することでたどることができ，これにより，AST からノードを取得することができる．そこで，図 7 のように差分検出アルゴリズムによって取得した行差分と一致する AST のノードを AST から取得する．ast_node はその取得した AST のノードの情報から成り，ノードの名前，ノードの種類，ノードの位置，ノードの中身等の情報が含まれている．ここでノードの種類とは，MethodDeclaration(メソッド宣言) や MethodInvocation(メソッド呼び出し)，VariableDeclarationStatement(変数宣言文) 等の情報のことである．メソッド抽出リファクタリングの集約パターンを検出するためには，(Insert, MethodDeclaration) と (Insert, MethodInvocation)，(Delete，code_block) の 3 つの情報が必要であるため，本研究では，この 3 種類の AST ノードのみを取得して，AST ノードリストとして保存する．

3.2.3 メソッド抽出リファクタリングの集約パターン照合

差分ノードを取得した後，蓄積した差分ノードがメソッド抽出リファクタリングの集約パターンと一致するかを識別する必要がある．提案環境では，2.2.1 節の Fowler の定義に従い，以下を満たす差分ノードが検出された時，メソッド抽出リファクタリングが行われたと判断する．

メソッド抽出リファクタリングの集約パターン

$$
\begin{gathered}
(Insert, MethodDeclaration) \\
\wedge (Delete, code\_block) \\
\wedge (Insert, MethodInvocation) \\
where \\
position(code\_block) = position(MethodInvocation) \\
\wedge\ name(MethodDeclaration) = name(MethodInvocation)
\end{gathered}
$$

つまり，提案手法では新しいメソッドが宣言され，あるコード片が削除され，そしてコード片が削除された位置に新しいメソッドの呼び出し文が追加された際にメソッド抽出が行われたと判断する．上式を見てみると，1 行目の $(Insert, MethodDeclaration)$ が新しいメソッド宣言が行われたことを表し，2 行目の $(Delete, code\_block)$ があるコード片が削除されたことを表し，3 行目の $(Insert, MethodInvocation)$ が新しくメソッド呼び出し文が挿入されたことを表す．しかし，それだけではメソッド抽出リファクタリングが行われたと判断することはできないため，さらに条件を設け，5 行目の $position(code\_block) = position(MethodInvocation)$ は削除されたコード片の位置とメソッド呼び出し文が挿入された位置が同じであるか判定を行い，6 行目の $name(MethodDeclaration) = name(MethodInvocation)$ は新しく宣言されたメソッド名と，新しく挿入されたメソッド呼び出し文のメソッド名が同じであるか判定を行う．以上，5 つの条件が満たされた時，メソッド抽出リファクタリングの集約パターンとして検出される．

3.2.4 コードクローン検出

メソッド抽出リファクタリングを行ったコード片のコードクローンを検出するために提案環境では CCFinderX を利用する．CCFinderX は，2.1.4 節で説明したように，多くの研究や企業で使用実績があり，高速にコードクローンを高い精度で検出するため，本研究でも利用した．メソッド抽出リファクタリングを行ったコード片のコードクローンを検出するためには，メソッド抽出リファクタリングが行われる直前のソースコードに対してコードクローンを検出する必要があるため，メソッド抽出リファクタリングが開始されたことを検知する

必要がある．そこで，提案環境では，開発者が新たなメソッド作成を行ったタイミングを，この後メソッド抽出リファクタリングが行われる可能性があると考え，そのタイミングでコードクローン検出を行う．

3.2.5　クローンセット照合

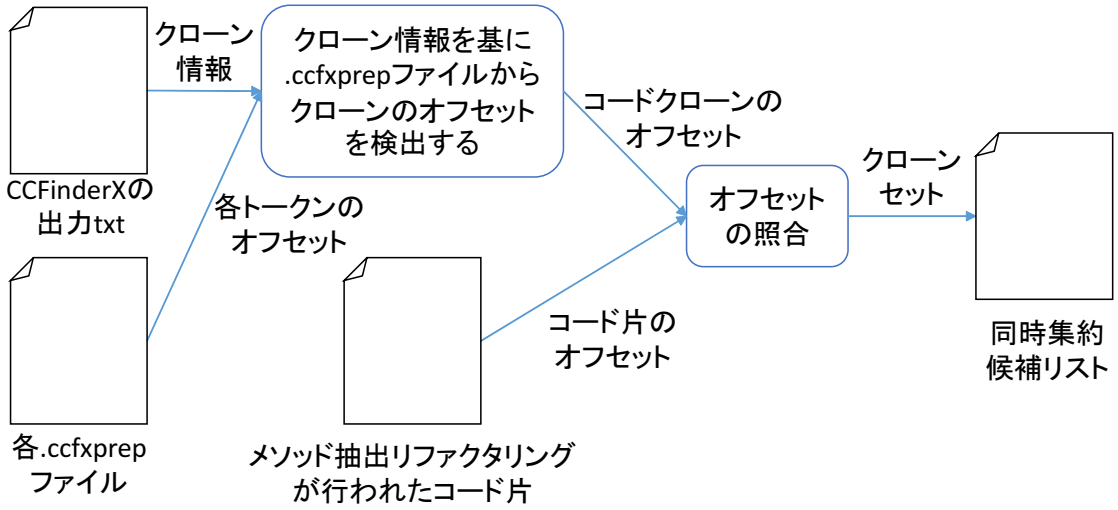

図 8: クローンセット照合

メソッド抽出リファクタリングの集約パターンが検出されると，次に，メソッド抽出リファクタリングで抽出されたコード片のクローンセットを検出する必要がある．そこで，3.2.1 節で CCFinderX が検出したコードクローンから，メソッド抽出されたコード片のコードクローンに一致するものを探す．図 8 はクローンセット照合の概要図となる．図 9 は CCFinderX のコードクローン検出結果の例である．図 9 の clone_pairs{} で囲われた部分の 1 行が 1 組のクローンペア情報を表す．1 組のクローンペア情報は，タブ文字で区切られており，一番左はクローンセット ID，右 2 つはクローンペアである各コードクローンのクローン情報を表している．クローン情報は，“(コードクローンが属するファイル ID).(.ccfxprep ファイルの開始行)-(.ccfxprep ファイルの終了行)” という形式で書かれている．ファイル ID は，図 9 の source_files{} で囲われた部分の ID を表しており，このファイル ID を用いてクローンが所属するファイルを識別することができる．.ccfxprep ファイルとは，CCFinderX の出力ファイルの 1 つであり，図 10 にその例を掲載する．.ccfxprep ファイルは，各 java ファイルごとに 1 つずつ作られ，各 java ファイルの全トークンについての位置情報が記されている．1 行に 1 トークンの位置情報が記されており，それは，(何行目).(行内でのオフセット).(ファイルの最初からのオフセット) という形式である．クローン情報から，.ccfxprep ファイルの

```
source_files {↓
1       C:¥UserSoft¥pleiades-e4.5-java-jre_20160312¥pleiades¥workspace¥JFreeChart↓
         ¥src¥main¥java¥org¥jfree¥chart¥LegendItem.java 1814↓
2       C:¥UserSoft¥pleiades-e4.5-java-jre_20160312¥pleiades¥workspace¥JFreeChart↓
         ¥src¥main¥java¥org¥jfree¥chart¥util¥SerialUtils.java   2933↓
}↓
source_file_remarks {↓
}↓
clone_pairs {↓
7       1.929-979       1.1079-1129↓
7       1.1079-1129     1.929-979↓
20      1.1228-1285     1.1259-1316↓
20      1.1259-1316     1.1228-1285↓
17      2.82-132        2.886-936↓
17      2.886-936       2.82-132↓
24      2.1170-1256     2.1243-1329↓
24      2.1170-1256     2.1316-1402↓
6       2.1825-1912     2.1905-1992↓
6       2.1825-1912     2.1985-2072↓
}↓
```

図 9: CCFinderX のコードクローン検出結果例

```
929 |20a.6.5215      +0      )def_block↓
930 |31e.c.6923      +0      (def_block↓
931 |31e.c.6923      +4      r_void↓
932 |31e.11.6928     +8      id|setShape↓
933 |31e.19.6930     +0      c_func↓
```

図 10: .ccfxprep ファイルの出力例

何行目から何行目のトークンがコードクローンであるか分かるため，コードクローンの開始オフセットと終了オフセットを取得することができる．そして，メソッド抽出リファクタリングによって抽出されたコード片のオフセットと，上記の要領で取得できる CCFinderX の出力結果のコードクローンのオフセットが一致するコードクローンを検出する．最後に，そのコードクローンが属するクローンセットのコードクローンをすべて検出し，そのクローンセットを同時にメソッド抽出リファクタリングを行うべきコード片として，開発者に提示する．

3.2.6 コードクローン提示

同時にメソッド抽出リファクタリングを行うべきコードクローンを検出した後，その情報を開発者に対して提示する必要がある．提案環境では Eclipse のビューを用いて，メソッド抽出リファクタリングを行うべきコードクローン一覧を提示する．同時集約候補の表示のために，最近修正されたファイル内のコードクローン (Modified File Clone) と，修正された

ファイル以外のコードクローン (Between Group Clone) に分けて表示する．その理由としては，ファイル内のコードクローンの方が，ファイル間のコードクローンよりも集約の優先度が高く，さらに，容易に集約を行えると考えるからである．各コードクローンにはクローンセット ID と，クローン情報，行情報を持たせている．クローンセット ID はそのコードクローンが属するクローンセットを識別するための ID であり，クローン情報は属するファイル情報とコードクローン部分のオフセットを表しており，行情報はコードクローン部分の行を表している．

さらに，開発者がコードクローン一覧から 1 つのコードクローンを選択すると，そのコードクローンが含まれるファイルを開き，そのコードクローン部分をハイライトすることにより，開発者がコードクローンを把握しやすくなっている．このハイライトされたコードクローンを見て，開発者は同時にメソッド抽出リファクタリングを行うかどうかを検討することができる．

4　提案環境の利用シナリオ

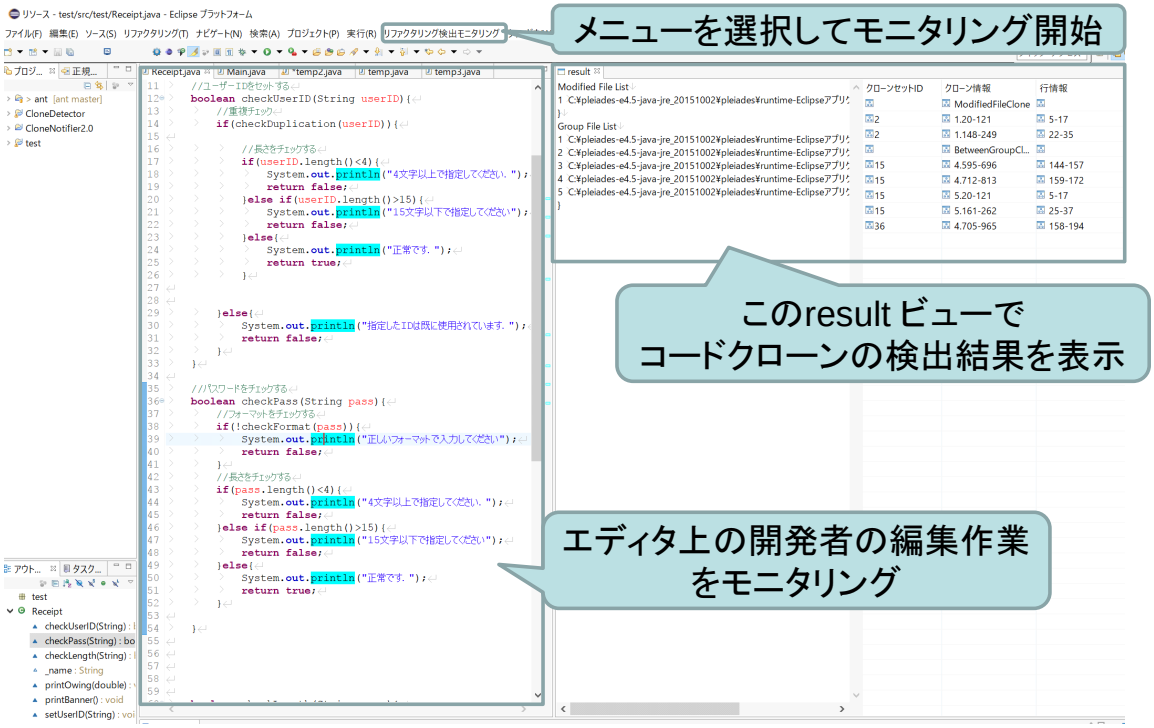

図 11: 提案環境の画面イメージ

提案環境の画面イメージを図 11 に示す．提案環境では，図 11 の “リファクタリング検出モニタリング” メニューを選択することで，開発者のエディタ上での編集作業のモニタリングを開始する．そして，開発者の編集作業からメソッド抽出リファクタリングの集約パターンを検知すると，メソッド抽出リファクタリングが行われたコード片のコードクローンの検出結果を，図 11 の右側の result ビューを用いて表示する．

図 12 にあるコードクローンの例を用いて，提案環境の利用シナリオを説明する．図 12 のソースコードは，ユーザー ID を設定する際，すでに登録されているユーザー ID ではなく，4 文字以上 15 文字以下という制限があり，さらにパスワードを設定する際，アルファベットと数字を両方必ず含み，4 文字以上 15 文字以下であるという制限があるシステムを想定している．それぞれの認証は，14 行目～32 行目の checkUserID メソッドと，35 行目～53 行目の checkPass メソッドによって行われている．これら 2 つのメソッドは，それぞれ重複ユーザー ID の確認とフォーマットの確認は別処理であるが，18 行目～27 行目と 42 行目～52 行目の長さに関する制限を確認する処理はコードクローンとなっている．

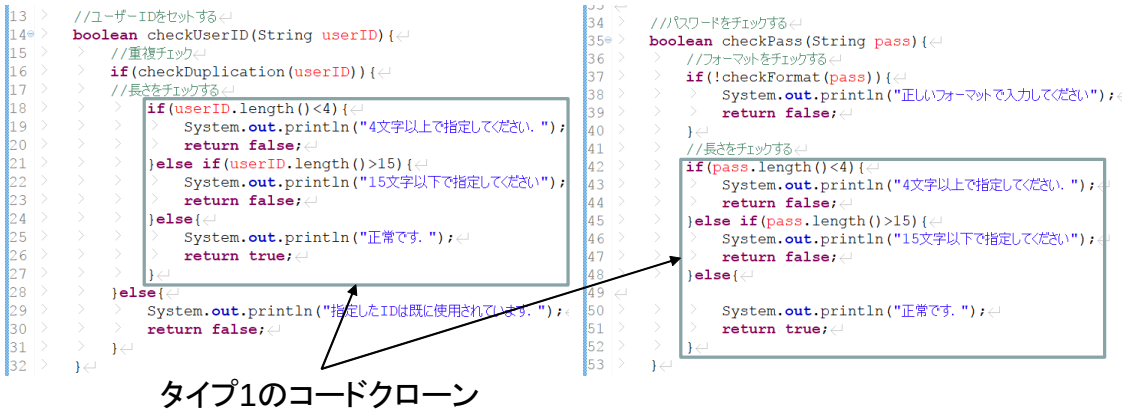

図 12: コードクローンの例

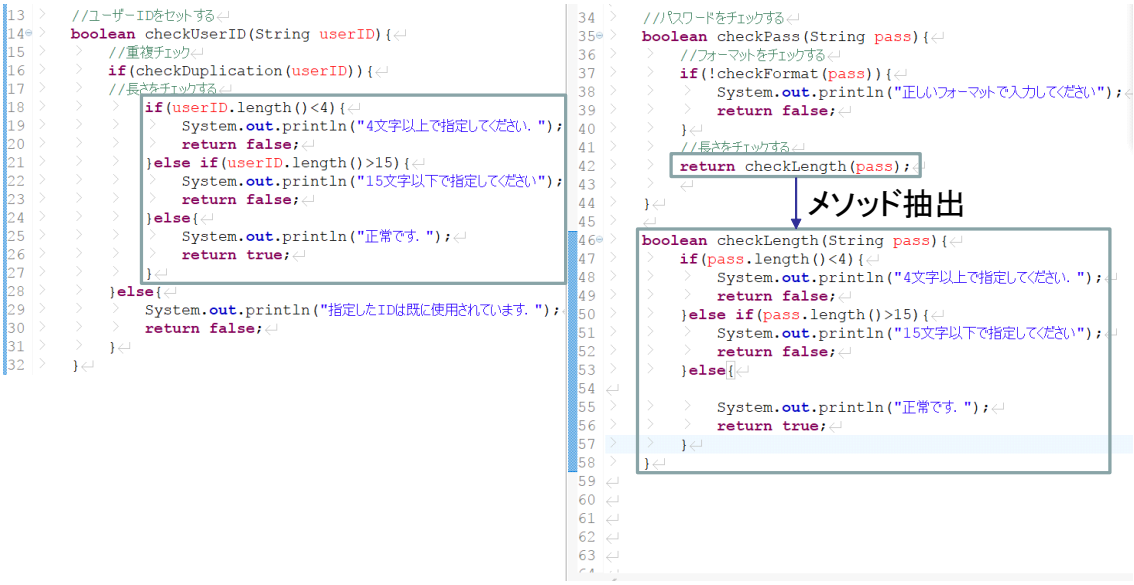

図 13: 開発者によるメソッド抽出リファクタリング

開発者は図 12 のソースコードに対して，編集作業を行う．この開発者の編集作業を提案環境がモニタリングする．モニタリングの際，提案環境は開発者によるキー入力を追い，ソースコードから行差分を取得して，その中からメソッド抽出リファクタリングの集約パターンに関係する AST ノードを探す．そして，開発者は図 12 の 42 行目〜52 行目の長さに関する制限を確認する処理部分を，図 13 のように checkLength メソッドとして抽出する．開発者はその際，(1) 新しい checkLength メソッドの宣言，(2) 図 12 の 42 行目〜52 行目のコード片の削除，(3)42 行目に新しい checkLength メソッドの呼び出し文の挿入，という 3 つの操作

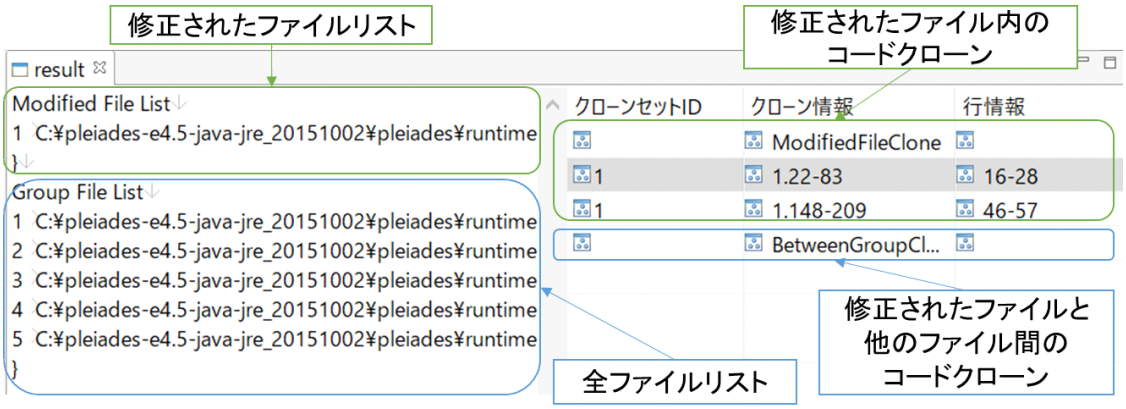

図 14: メソッド抽出リファクタリング候補のコードクローン

を行っているはずである．提案環境は，その操作列を追い，各操作を (delta_type, ast_node) の組として蓄積し，3 つの操作列が検知された際に，メソッド抽出リファクタリングの集約パターンとして検出し，“メソッド抽出リファクタリングを検出しました．”というメッセージダイアログを表示し，開発者に通知する．

メソッド抽出リファクタリングの集約パターンを検知すると，提案環境は図 13 でメソッド抽出リファクタリングによって抽出されたコード片のコードクローンを検出し，図 14 のように Eclipse のビュー機能を用いて，メソッド抽出リファクタリング候補のコードクローンを開発者に対して提示する．メソッド抽出リファクタリング候補のコードクローンを提示するビューでは，左半分で，プロジェクトに含まれるファイルリストを表示し，右半分で実際に検出されたコードクローン情報を表示する．実際に検出されたコードクローン情報の見方を説明すると，まず，クローンセット ID は同じクローンセットに含まれるコードクローンに一意に与えられるものである．クローン情報はピリオドの前の数字が左半分のファイルリストに対応する数字で，ファイル名を示し，ピリオドの後の数字は，それぞれコードクローンの開始オフセットと，終了オフセットを示している．そして，行情報はコードクローン部分の行の開始位置と終了位置を示している．コードクローンの検出結果は 2 種類に分けて表示するようにしており，上の ModifiedFileClone が修正されたファイル内のコードクローンを表示し，下の BetweenGroupClone は修正されたファイルとそれ以外のファイルをまたぐコードクローンを表示する．

最後に，図 14 の result ビューから開発者が 1 つのコードクローンを選択すると，エディタでそのファイルを開き，図 15 のように，そのコードクローン部分を緑色にハイライトして表示する．これにより，開発者は，コードクローン部分を確認することができ，メソッド抽出リファクタリングを検討できると考えられる．

*Receipt.java

```
10
11    //ユーザーIDをセットする
12    boolean checkUserID(String userID){
13        //重複チェック
14        if(checkDuplication(userID)){
15
16            //長さをチェックする
17            if(userID.length()<4){
18                System.out.println("4文字以上で指定してください. ");
19                return false;
20            }else if(userID.length()>15){
21                System.out.println("15文字以下で指定してください");
22                return false;
23            }else{
24                System.out.println("正常です. ");
25                return true;
26            }
27
28
29        }else{
30            System.out.println("指定したIDは既に使用されています. ");
31            return false;
32        }
33    }
34
```

図 15: コードクローン部分のハイライト表示

5　評価実験

提案環境のメソッド抽出リファクタリング支援における有効性を調査するために，評価実験を行った．具体的には，情報科学について研究している博士前期課程 1 年の学生 8 人に対して，提案環境と CCFinderX の GUI である GemX を使用してもらい比較実験を行った．

提案環境は，開発者が保守作業を行っている最中に，メソッド抽出リファクタリングを行いたいコード片を見つけて，そのコード片に対してメソッド抽出リファクタリングを行った際に，他にも同様のメソッド抽出リファクタリングを行うべき箇所があることを開発者に伝えて，効率的な保守作業を行う環境を提供することを目的としている．そこで，評価実験では，メソッド抽出リファクタリングを行うべきコード片と，そのコード片のコードクローンが複数含まれるデータセットを用意した．そして，被験者には，用意したデータセットに対して提案環境を使う場合と，GemX を使う場合に分けて，コードクローンのメソッド抽出リファクタリング作業を行ってもらい，以下の 2 つの調査を行った．

- データセットの内，集約することができたコードクローンの数．
- データセット内のコードクローンを集約するのにかかった時間．

また実験後には，被験者に対してアンケートを実施した．

5.1　データセット作成

評価実験を行うにあたり，本研究ではデータセットを 2 種類用意した．対象ソフトウェアは以下の 2 つである．

- JFreeChart[11](26 万行，990 クラス)
- JUnit[13](4.3 万行，449 クラス)

対象ソフトウェアはどちらも Java 言語で記述されている．JFreeChart はグラフライブラリであり，各種統計図表や関数のグラフを描くことができる．JUnit は，Java で開発されたプログラムにおいて，単体テストの自動化を行うためのフレームワークである．

図 16 にデータセットの作成方法を示す．データセットには，メソッド抽出リファクタリングを行うべきコード片とそのコードクローンをいくつか含む必要がある．そのため，本研究ではまず，上記の 2 つのオープンソースから短すぎるメソッドではなく，かつ同じ回数呼び出されるメソッドを探した．そして，そのメソッドをメソッド抽出リファクタリングを行うべきコード片として定義するために，メソッドのインライン化を行った．本研究では，呼び出し先の中身が呼び出し元に直接記述された，このメソッドのインライン化が行われた

メソッドを，メソッド抽出リファクタリングを行うべきコードクローンとして定義した．提案環境は，開発者の保守作業で使用してもらうことを想定している．そこで，メソッド抽出リファクタリングを行うべきコード片の 1 つにデッドコードを加える等して，実験ではこのコード片に対して保守作業を最初に行ってもらい，保守作業の最中という設定を再現して，提案環境の有効性を確かめた．

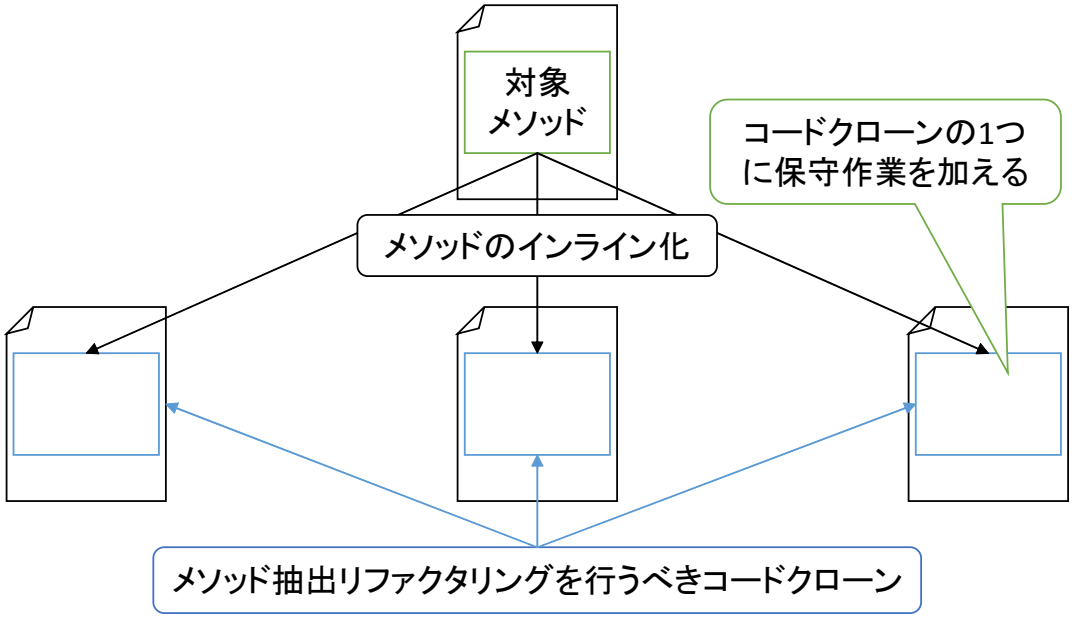

図 16: データセット作成方法

上記の方法でデータセットを作成することができる．本実験では，各被験者に提案環境と GemX の両方の環境を使用してもらうために，2 つのデータセットを作成した．それぞれのデータセットをデータセット 1，データセット 2 と定義する．各被験者には，2 つのデータセットに対して，提案環境と GemX のそれぞれを用いて，2 つのタスクに取り組んでもらった．そして，被験者を表 1 のような 4 つのグループに分けることによって，被験者間の能力差と，提案環境と GemX との使用順等による実験の結果の差異を極力小さくするように実験を行った．本実験は 8 人を，各グループに 2 人ずつ割り当てた．

5.2 実験手順

4.1 節で説明したデータセットを用いた評価実験の手順について説明する．まず，被験者間の事前知識の違いによる結果の相違を無くすために，本実験を行う前に，コードクローン，メソッド抽出リファクタリング，CCFinderX について，被験者に対して説明した．また，被験者に提案環境，GemX の両方の使い方を説明し，それぞれの環境を使用してもら

表 1: 実験のグループ分け

グループ	タスク 1	タスク 2
A	提案環境 + データセット 1	GemX + データセット 2
B	GemX + データセット 1	提案環境 + データセット 2
C	提案環境 + データセット 2	GemX + データセット 1
D	GemX + データセット 2	提案環境 + データセット 1

い，被験者に対して両環境の使い方を理解していることを確認した．

その後，用意した 2 つのデータセットに対して実験タスクを行ってもらった．実際に実験で用いた実験手順書を用いて，提案環境を用いた実験手順と，GemX を用いた実験手順について説明する．実験手順書の実験タスクは，提案環境+データセット 1，提案環境+データセット 2，GemX+データセット 1，GemX+データセット 2 の 4 種類の組み合わせが存在するが，提案環境の実験手順説明には，提案環境+データセット 1 の実験手順書，GemX の実験手順説明には GemX+データセット 2 の実験手順書を用いた．評価実験に用いた残りの 2 つの実験手順書については，付録として，本論文の最後に掲載する．

5.2.1 提案環境の実験手順

提案環境を用いる場合の実験手順について，以下の実験手順書 (提案環境+データセット 1) を用いて説明する．まず，手順 1 でメソッド抽出リファクタリングを行うべきコード片が含まれているメソッドを開いてもらう．提案環境は，保守作業を行っている開発者の開発作業をモニタリングする．そこで，手順 2 で被験者の開発作業のモニタリングを開始し，手順 3 では，開発作業中の保守作業によるコード片の編集を再現している．手順 3 以降はメソッド抽出リファクタリング作業となるため，手順 4 でストップウォッチを用いて時間を測り始める．手順 5 では，メソッド抽出リファクタリングを行うべきコード片を被験者に与えている．提案環境はメソッド抽出リファクタリングが行われたことにより，他のメソッド抽出リファクタリング候補を与える可能性があるため，このように最初にメソッド抽出リファクタリングを行うべきコード片を与えることとした．そして，手順 5 で指定したコード片に対してメソッド抽出リファクタリングが適切に行われると，手順 6 では，提案環境が被験者に対して提示した他のメソッド抽出リファクタリング候補に対して，手順 5 で新しく作ったメソッドに，同様にメソッド抽出リファクタリングを行ってもらった．最後に，提案環境が提示したすべての候補に対してメソッド抽出リファクタリングにかかった時間の記録を行った．

実験手順書 (提案環境+データセット 1)

手順 1. Junit/src/test/java/junit/tests/runner/BaseTestRunnerTest.java の 105 行目 testInvokeSuiteOnNonSubClassOfTestCase() メソッドを開く.

手順 2. メニューのリファクタリング検出モニタリングを押す.

手順 3. testInvokeSuiteOnNonSubClassOfTestCase() メソッドに対して，137 行目に

```
if (!Modifier.isStatic(suiteMethod.getModifiers())) {
    runner.runFailed(“ Suite() method must be static ”);
    suite = null;
}
```

という行が無いことが原因でバグが生じていることが判明したため，137 行目に上記コードを追加する.

手順 4. ストップウォッチで時間を測り始める.

手順 5. testInvokeSuiteOnNonSubClassOfTestCase() メソッドの/*from here*/と/* to here*/で囲われた部分に対してメソッド抽出リファクタリングを行う．メソッド抽出リファクタリングは，Junit/src/main/java/junit/runner/BaseTestRunner.java クラス内に返り値 Test の getTest() メソッドとして抽出すること．また，メソッド抽出リファクタリングはファウラーの定義の通りに行うこと.

手順 6. 手順 5 のメソッド抽出リファクタリングが終わったら，本環境が推奨するコードクローン対して，手順 5 で新しく宣言した getTest() メソッドに，同様にメソッド抽出リファクタリングを行う.

手順 7. 全コードクローンに対して，メソッド抽出リファクタリングが終わったら，ストップウォッチを止めて，時間を記録する.

5.2.2 コードクローン分析環境 GemX の実験手順

GemX を用いる場合の実験手順について，以下の実験手順書 (GemX+データセット 2) を用いて説明する．まず，手順 1 でメソッド抽出リファクタリングを行うべきコード片が含まれているメソッドを開いてもらう．手順 2 では，開発作業中の保守作業によるコード片の編集を再現している．そして，手順 3 でストップウォッチで時間を測り始めてもらい，手順 4 で GemX を用いてコードクローンの検出を行ってもらう．CCFinderX が出力するコードクローンは，直前の編集作業によってコードクローンの検出結果が異なることがある．提案

環境の場合は，保守作業を常にモニタリングし，適切なタイミングで CCFinderX を用いてコードクローンの検出を行うため，4.2.1 節の実験手順書 (提案環境+データセット 1) の手順 4 ではコードクローン検出を行う必要が無かったが，GemX を用いる場合には，メソッド抽出リファクタリングを行う直前にコードクローンの検出を行う必要がある．その後手順 5 では，手順 4 の GemX の出力結果と，手順 6 で指定しているメソッド抽出リファクタリングを行うべきコード片を照らし合わせて，同時にメソッド抽出リファクタリングを行うべきコード片を探し，すべてのコードクローンに対して，メソッド抽出リファクタリングにかかった時間を測ってもらった．

実験手順書 (GemX+データセット 2)

手順 1. JFreeChart/src/main/java/org/jfree/chart/axis/Axis.java の writeSmallObljject() メソッド (1779 行目) を開く．

手順 2. 1793 行目〜1795 行目にデッドコードがあるため，それを削除する．

手順 3. ストップウォッチで時間を測り始める．

手順 4. CCFinderX の GemX を用いてコードクローン検出を行う．(対象は C:/usersoft/pleiades-e4.5-java-jre_20160312/pleiades/workspace/JFreeChart)

手順 5. 手順 4 で CCFinderX が検出したコードクローンの内，手順 6 でメソッド抽出リファクタリングを行うコード片として指定したコード片にコードクローンがあるかどうか探す．

手順 6. writeSmallObject() メソッドの/*from here*/と/*to here*/で囲われた部分に対してメソッド抽出リファクタリングを行う．メソッド抽出リファクタリングは，JFreeChart/src/main/java/org/jfree/chart/util/SerialUtils.java クラス内に static 修飾子をつけ，writeAttributedString() メソッドとして抽出すること．メソッド抽出リファクタリングはファウラーの定義の通りに行うこと．

手順 7. 手順 6 のメソッド抽出リファクタリングが終わったら，手順 5 のコードクローンに対して，手順 6 で新しく宣言した writeAttributedString() メソッドに，同様にメソッド抽出リファクタリングを行う．

手順 8. 全コードクローンに対して，メソッド抽出リファクタリングが終わったら，ストップウォッチを止めて，時間を記録する．その後，その旨を伝える．

5.3 結果と考察

本実験において，データセットに対して被験者がメソッド抽出リファクタリングを行うことができたコードクローンの数を表 2 に示す．各データセットにはメソッド抽出リファクタリングを行うべきコード片を 3 つ用意している．表 2 を見ると，8 人中 6 人が提案環境を使った方が，多くのコードクローンに対してメソッド抽出リファクタリングを行うことができており，平均を見ても提案環境の方が多くできていることがわかる．したがって，提案環境を用いたほうがより多くのコードクローンに対して，メソッド抽出リファクタリングを行うことができる傾向にあると考えられる．t 検定を用いて, メソッド抽出リファクタリングを行えたコードクローンの数の統計的な有意差を判定した結果 $p(T <= t)$ 両側 $= 0.003 < 0.05$ となるため, 有意水準 0.05 のもとで有意差があることが分かった．したがって，提案環境と GemX のメソッド抽出リファクタリングを行えたコードクローンの数の差は，統計的な有意な差であると考えられる．

また，3 つすべてのコードクローンのメソッド抽出リファクタリングを行うことができた被験者数が，GemX が 2 人であるのに対して，提案環境は 7 人という結果になっている．このことから，GemX では同時にメソッド抽出リファクタリングを行うべきコードクローンを見落としてしまうことがあるが，提案環境ではほとんど見落としなく，メソッド抽出リファクタリング候補を提案していると考えられる．

表 2: メソッド抽出リファクタリングを行えたコードクローンの数

被験者	提案環境	GemX
A	3	3
B	3	2
C	1	0
D	3	2
E	3	3
F	3	2
G	3	2
H	3	2
平均	2.75	2

本実験において，データセットに対して被験者がメソッド抽出リファクタリングにかかった時間を表 3 に示す．本実験では，被験者が提案環境と GemX それぞれを用いて，メソッド抽出リファクタリングを行うべきだと考えられるすべてのコードクローンに対して，メソッ

ド抽出リファクタリングを完了するまでの時間を計測した．表 3 を見ると，対象のデータセットが異なるため，厳密ではないが，提案環境を利用したほうが早かった被験者が 6 名，GemX を利用したほうが早かった被験者が 2 名という結果になり，また，平均時間に関しても提案環境が 17 分 23 秒，GemX が 26 分 42 秒となり，提案環境の方が短いという結果であった．したがって，提案環境を利用したほうが，実験課題にかかった時間が短い傾向にあると考えられる．t 検定を用いて, 課題にかかった時間の統計的な有意差を判定した結果 $p(T <= t)$ 両側 $= 0.017 < 0.05$ となるため, 有意水準 0.05 のもとで有意差があることが分かった．したがって，提案環境と GemX のコードクローンのメソッド抽出リファクタリングにかかる時間の差は，統計的な有意な差であると考えられる．

表 3: 課題にかかった時間

被験者	提案環境	GemX
A	22 分 45 秒	43 分 30 秒
B	17 分 42 秒	33 分 00 秒
C	34 分 00 秒	30 分 00 秒
D	22 分 54 秒	21 分 39 秒
E	9 分 51 秒	21 分 46 秒
F	11 分 00 秒	18 分 00 秒
G	7 分 59 秒	19 分 03 秒
H	12 分 51 秒	26 分 39 秒
平均	17 分 23 秒	26 分 42 秒

また，GemX を利用したほうが早かった被験者 C，D の表 2 の結果を見てみると，どちらも提案環境の方がメソッド抽出リファクタリングを行えたコードクローンが多いという結果となっている．実際にメソッド抽出リファクタリングを行えたコードクローンの数の違いが，この時間差を生み出したと考えられる．さらに，被験者 C においては，実験後のアンケートで途中で気分が悪くなったという記述があったため，その影響も考えられる．被験者 D においては，実験の手順書を読み間違えたことが分かったため，それによる結果への影響も考えられるため，実験手順書をもっとわかりやすく作る必要があったと感じている．

5.4 実験後のアンケート

本研究では，評価実験の後，被験者に対してアンケートを実施した．実施したアンケートの内容とその結果を以下に示す．

Q1. Java の経験について

- 大学の講義等で習ったか．
- 研究で利用しているか．
- 趣味等でも利用しているか．(プログラミングコンテストやアプリ開発等)
- Java の経験年数，または開発行数はどの程度か．

Java の経験については，大学の講義等で習った程度の人が 4 人，研究や趣味等でも使用している人が 4 人という結果であった．また，Java の経験年数については，半年〜5 年という様々な結果となった．したがって，被験者の Java の経験年数には若干の差があったと考えられる．また，このことが実験結果に影響を与えている可能性も考えられる．

Q2. コードクローンの知識について

- 全く知らなかった．
- 本等で知っている程度．
- コードクローンに関する研究を行っている．
- CCFinderX を使ったことがある．

コードクローンの知識については，本等で知っている程度の人が 4 人，コードクローンに関する研究を行っている人が 4 人という結果になった．また，コードクローンに関する研究を行っている 4 人の内 2 人が CCFinderX を使ったことがあることが分かった．このことが実験結果に影響を与えている可能性が考えられる．

Q3. メソッド抽出リファクタリングについて

- 全く知らなかった．
- 本等で知っている程度．
- メソッド抽出リファクタリングを難なく行える．

- メソッド抽出リファクタリングに関する研究を行っている．または今後行う予定．
- 普段の開発作業から意識的に行っている．

メソッド抽出リファクタリングについては，本等で知っている程度の人が 6 人，普段の開発作業から意識的に行っている人が 2 人という結果になった．このことが実験結果に影響を与えている可能性が考えられる．

Q4．提案環境の集約候補の提示タイミングは適切だと感じたか．(1:早い〜5:遅い)

この質問に関しては，8 人中 6 人が 3 の適切，2 人が 4 のやや遅いという結果になった．その理由としては，提案環境はメソッド抽出リファクタリングを 1 つ終えてからでないと，他の集約候補が分からないため，抽出したメソッドのインターフェースや抽出先の検討がやりにくくなってしまうというものだった．これについては，メソッド抽出リファクタリングの最中に集約候補を見つけて知らせる等の工夫を検討する必要があると考えられる．

Q5．提案環境の良かった点・改善した方が良い点．

提案環境の良かった点としては，以下のような記述があった．

- Eclipse のプラグインであることによって，別のツールを参照することなく，Eclispe で作業が完結でき，直接コードと向き合えるのが使いやすいと感じた．
- 1 つのメソッド抽出リファクタリングを終えたタイミングで，他の集約候補が提示されるのが良いと思った．
- IDE で編集中に集約候補がハイライト表示されるのが良かった．

Eclipse のプラグインとして提供されることにより，ストレスなくメソッド抽出リファクタリングを行えたことが良かったという意見が多かった．また，改善した方が良い点としては以下のような記述があった．

- メソッド抽出リファクタリングが予期しない順番で行われた場合，提案環境が対応できないことがある点．
- メソッド抽出リファクタリング候補にたまに関係ないものが含まれている点．

提案環境は，メソッド抽出リファクタリングが Fowler の定義に従って行われることを前提としているため，その手順が違うと，メソッド抽出リファクタリングとして認識できない場合がある．また，提案環境は CCFinderX の出力結果を用いて，その位置情報から同時集約候補を探している．そのため，CCFinderX の出力結果にたまに含まれる関係ないコードクローンも出力してしまうという問題点も挙げられる．それらは，改善の必要があると考えられる．

Q6. その他自由記述.

以下のような記述が挙げられていた.

- Java の知識が少ないことがもあるが，画面と文字が小さかったために実験中に片頭痛を起こし，あまり実験に集中できなかった．結果に影響した可能性がある.

本実験は，環境構築等を済ませた PC を事前に用意し，評価実験中にはその PC を被験者に使ってもらった．それにより，上記のような被験者の意見があった．そのため，被験者の不慣れな開発環境が実験結果に影響を及ぼした可能性が考えられる．他には，以下のような意見もあった.

- 提案環境がメソッド抽出リファクタリングが行われたことを検知するためには，メソッド抽出リファクタリングが Fowler の定義の順番通りに行われなければならないという制約を無くし，柔軟な操作に対応して欲しい.

提案環境は上記の意見の通り，メソッド抽出リファクタリングが Fowler の定義の順番通りに行われる必要がある．この意見にあるように，Fowler の定義の順番を無視したメソッド抽出リファクタリングにも対応することを，今後の課題として検討する必要があると考えられる.

6　まとめ

本研究では，開発作業のモニタリングを行い，メソッド抽出リファクタリング支援を行う環境を構築した．具体的には，開発者のキー入力による編集作業をモニタリングし，編集に関係する AST ノードを蓄積していき，その中からメソッド抽出リファクタリングの集約パターンを検知して，メソッド抽出リファクタリングが行われたコード片のコードクローンを開発者に提示することで支援を行う．

提案環境のメソッド抽出リファクタリング支援における有効性を確認するために，評価実験を行った．評価実験では被験者を用意し，提案環境を使う場合と，CCFinderX の GUI である GemX を使う場合で，コードクローンの集約を行う比較実験を行った．両者でコードクローンの集約にかかる時間と，その精度を比較したところ，その有意差を確認することができた．

今後の課題としては，適応できるリファクタリングのパターンを増やすことが挙げられる．現在，検出するリファクタリングのパターンはメソッド抽出の集約パターンだけであるが，コードクローンの削減に適用可能なリファクタリングパターンは他にもたくさん考えられる．例えば，メソッドの移動，メソッドの引き上げ，メソッドのパラメータ化，テンプレートメソッドの形成，親クラスの抽出等が挙げられる．これらのリファクタリングパターンにも適用できることが今後の目標である．

謝辞

大阪大学大学院情報科学研究科コンピュータサイエンス専攻 井上 克郎 教授には，研究に関する適切な御指導及び御助言を賜りました．井上 教授の御指導及び御助言のおかげで本論文を完成させることができました．井上 教授に心より深く感謝いたします．

大阪大学大学院情報科学研究科コンピュータサイエンス専攻 松下 誠 准教授には，研究の各段階において多くの御助言を賜りました．多くの御指導及び御助言を頂いた 松下 准教授に心より深く感謝いたします．

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，研究に関する直接の御指導を賜りました．常に適切な御指導及び御助言を頂いたことにより，本論文を完成することができました．吉田 准教授に心より深く感謝いたします．

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，研究に関する多くの貴重な御助言を賜りました．多くの御助言を頂いたおかげで本論文を完成させることができました．崔 恩瀞 助教には心より深く感謝いたします．

大阪大学大学院情報科学研究科 春名修介 特任教授には，常に適切な御指導及び御助言を賜りました．多くの御助言を頂いた 春名 特任教授に心より深く感謝いたします．

大阪大学大学院情報科学研究科コンピュータサイエンス専攻 神田 哲也 特任助教には，研究においてたくさんの貴重な御意見を賜りました．多くの御助言を頂いた 神田 特任助教に心より深く感謝いたします．

大阪大学大学院情報科学研究科コンピュータサイエンス専攻 堤 祥吾 氏には，研究に関する相談に乗って頂きました．心より深く感謝いたします．

また, 本研究における被験者実験において, 長時間の実験にも関わらず被験者を快く引き受けてくださった大阪大学大学院情報科学研究科コンピュータサイエンス専攻井上研究室の博士前期課程 1 年の皆様，並びに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博士前期課程 1 年の皆様に心より深く感謝いたします．

最後に，御指導，御助言を頂き私を支えてくださった，大阪大学大学院情報科学研究科コンピュータサイエンス専攻井上研究室の皆様に，心より深く感謝いたします．

付録

付録として，4.2 節で紹介できなかった，評価実験に用いた残りの 2 つの実験手順書を掲載する．

実験手順書 (提案環境+データセット 2)

手順 1. JFreeChart/src/main/java/org/jfree/chart/axis/Axis.java の writeSmallObljject() メソッド (1779 行目) を開く．

手順 2. メニューのリファクタリング検出モニタリングを押す．

手順 3. 1793 行目〜1795 行目にデッドコードがあるため，それを削除する．

手順 4. ストップウォッチで時間を測り始める．

手順 5. writeSmallObject() メソッドの/* from here */と/* to here */で囲われた部分に対してメソッド抽出リファクタリングを行う．メソッド抽出リファクタリングは，JFreeChart/src/main/java/org/jfree/chart/util/SerialUtils.java クラス内に static 修飾子をつけ，writeAttributedString() メソッドとして抽出すること．メソッド抽出リファクタリングはファウラーの定義の通りに行うこと．

手順 6. 手順 5 のメソッド抽出リファクタリングが終わったら，本環境が推奨するコードクローンに対して，手順 5 で新しく宣言した writeAttributedString() メソッドに，同様にメソッド抽出リファクタリングを行う．

手順 7. 全コードクローンに対して，メソッド抽出リファクタリングが終わったら，ストップウォッチを止めて，時間を記録する．その後，その旨を伝える．

実験手順書 (GemX+データセット 1)

手順 1. Junit/src/test/java/junit/tests/runner/BaseTestRunnerTest.java の 105 行目 testInvokeSuiteOnNonSubClassOfTestCase() メソッドを開く.

手順 2. testInvokeSuiteOnNonSubClassOfTestCase() メソッドに対して，137 行目に

```
if (!Modifier.isStatic(suiteMethod.getModifiers())) {
    runner.runFailed(“ Suite() method must be static ”);
    suite = null;
}
```

という行が無いことが原因でバグが生じていることが判明したため，137 行目に上記コードを追加する.

手順 3. ストップウォッチで時間を測り始める.

手順 4. CCFinderX の GemX を用いてコードクローン検出を行う.(対象は C:/usersoft/pleiades-e4.5-java-jre_20160312/pleiades/workspaceJunit)

手順 5. 手順 4 で CCFinderX が検出したコードクローンの内，手順 6 でメソッド抽出リファクタリングを行うコード片として指定したコード片に，コードクローンがあるかどうか探す.

手順 6. testInvokeSuiteOnNonSubClassOfTestCase() メソッドの/*from here*/と/*to here*/で囲われた部分に対してメソッド抽出リファクタリングを行う．メソッド抽出リファクタリングは，Junit/src/main/java/junit/runner/BaseTestRunner.java クラス内に返り値 Test の getTest() メソッドとして抽出すること．また，メソッド抽出リファクタリングはファウラーの定義の通りに行うこと.

手順 7. 手順 6 のメソッド抽出リファクタリングが終わったら，手順 4 のコードクローン対して，手順 6 で新しく宣言した getTest() メソッドに，同様にメソッド抽出リファクタリングを行う.

手順 8. 全コードクローンに対して，メソッド抽出リファクタリングが終わったら，ストップウォッチを止めて，時間を記録する.

参考文献

[1] Brenda S. Baker. Finding clones with dup: Analysis of an experiment. *IEEE Trans. Softw. Eng.*, Vol. 33, No. 9, pp. 608–621, 2007.

[2] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *Proc. of ICSM*, pp. 368–377, 1998.

[3] Eunjong Choi, Norihiro Yoshida, Takashi Ishio, Katsuro Inoue, and Tateki Sano. Extracting code clones for refactoring using combinations of clone metrics. In *Proc. of IWSC*, pp. 7–13, 2011.

[4] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. A language independent approach for detecting duplicated code. In *Proc. of ICSM*, pp. 109–118, 1999.

[5] S. R. Foster, W. G. Griswold, and Sorin Leaner. Witchdoctor: Ide support for real-time auto-completion of refactorings. In *Proc. of ICSE*, pp. 222–232, 2012.

[6] Martin Fowler. リファクタリング-プログラミングの体質改善テクニック. ピアソン・エデュケーション, 2009.

[7] GemX. GemX. `http://www.ccfinder.net/doc/10.2/ja/tutorial-gemx.html`.

[8] 肥後芳樹, 楠本真二, 井上克郎. コードクローン検出とその関連技術. 電子情報通信学会論文誌, Vol. J91-D, No. 6, pp. 1465–1481, 2008.

[9] 肥後芳樹, 吉田則裕. コードクローンを対象としたリファクタリング. コンピュータソフトウェア, Vol. 28, No. 4, pp. 43–56, 2011.

[10] 井上克郎, 神谷年洋, 楠本真二. コードクローン検出法. コンピュータソフトウェア, Vol. 18, No. 5, pp. 47–54, 2001.

[11] JFreeChart. JFreeChart. `http://www.jfree.org/jfreechart/`.

[12] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proc. of ICSE*, pp. 96–105, 2007.

[13] JUnit. JUnit. `https://junit.org/junit4/`.

[14] 神谷年洋, 肥後芳樹, 吉田則裕. コードクローン検出技術の展開. コンピュータソフトウェア, Vol. 28, No. 3, pp. 29–42, 2011.

[15] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.*, Vol. 28, No. 7, pp. 654–670, 2002.

[16] Emerson Murphy-Hill, Chris Parnin, and P. Black Andrew. How we refactor, and how we know it. In *Proc. of ICPC*, pp. 199–206, 2013.

[17] Eugene W Myers. AnO (ND) difference algorithm and its variations. *Algorithmica*, Vol. 1, No. 1-4, pp. 251–266, 1986.

[18] Lutz Prechelt, Guido Malpohl, and Michael Philippsen. Finding plagiarisms among a set of programs with jplag. *J. UCS*, Vol. 8, No. 11, p. 1016, 2002.

[19] Program Creek. Use JDT ASTParser to parse Single Java files. http://www.programcreek.com/2011/11/use-jdt-astparser-to-parse-java-file/.

[20] Chanchal K. Roy, James R. Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, Vol. 74, No. 7, pp. 470–495, 2009.

[21] Chanchal K. Roy, Minhaz F. Zibran, and Rainer Koschke. The vision of software clone management: Past, present, and future (keynote paper). In *Proc. of CSMR-WCRE*, pp. 18–33, 2014.

[22] 佐野真夢, 吉田則裕, 春名修介, 井上克郎. 情報検索技術に基づく関数クローン検出を用いた変更管理システムの開発. 情報処理学会研究報告, Vol. 2015-SE-190, No. 4, pp. 1–8, 2015.

[23] 山中裕樹, 崔恩瀞, 吉田則裕, 井上克郎, 佐野建樹. コードクローン変更管理システムの開発と実プロジェクトへの適用. 情報処理学会論文誌, Vol. 54, No. 2, pp. 883–893, feb 2013.

[24] 山中裕樹, 崔恩瀞, 吉田則裕, 井上克郎. 情報検索技術に基づく高速な関数クローン検出. 情報処理学会論文誌, Vol. 55, No. 10, pp. 2245–2255, 2014.

[25] Yuki Yamanaka, Eunjong Choi, Norihiro Yoshida, Katsuro Inoue, and Tateki Sano. Applying clone change notification system into an industrial development process. In *Proc. of ICPC*, pp. 199–206, 2013.

[26] 吉田則裕, 肥後芳樹, 神谷年洋, 楠本真二, 井上克郎. コードクローン間の依存関係に基づくリファクタリング支援. 情報処理学会論文誌, Vol. 48, No. 3, pp. 1241–1442, 2007.

[27] Norihiro Yoshida, Yoshiki Higo, Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. On refactoring support based on code clone dependency relation. In *Proc. of METRICS*, pp. 16:1–16:10, 2005.