修士学位論文

題目

コードクローン分類の詳細化に基づく集約パターンの提案と評価

指導教員
井上 克郎 教授

報告者
徳永 将之

平成 24 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 23 年度 修士学位論文

コードクローン分類の詳細化に基づく集約パターンの提案と評価

徳永 将之

内容梗概

コードクローンとは，コピーアンドペースト等によって作成される，ソースコード上の一致もしくは類似したコード断片のことである．あるコードにバグが含まれた場合，そのコードクローン全てに対して一貫した修正を検討する必要があるため，コードクローンはソフトウェアの保守を困難にする要因と言われている．また，ソースコードの内部構造を変更することで保守性を高める技術はリファクタリングとして知られている．リファクタリングの優れた手法は，リファクタリングパターンとして Fowler や Kerievsky らによってカタログ化されている．特に，コードクローンの集約に関して述べたリファクタリングパターンを集約パターンと呼び，開発者は集約パターンを参照してコードクローンを集約することで，コードクローンの問題を解決することができる．既存の集約パターンは，詳細なコードクローンの特徴を特に考慮していないため，コードクローンの特徴に対応した詳しい集約作業手順が記されていない．従って，コードクローンの集約作業の初級者が既存の集約パターンを参照して集約作業を行うことは困難である．

本研究では，コードクローンの詳細な分類に基づいたコードクローン集約パターンの作成，および集約パターンの有効性の評価を行った．

集約パターン作成のために，まず，コードクローンの集約に特化した分類をオープンソースソフトウェアのコードを利用して詳細化した．次に，詳細化された分類に基づいてコードクローンを分類するコードクローン自動分類ツールを作成した．さらに，作成したツールを用いて，パターン作成の対象となるコードクローンの特徴を持つソースコード上のコードクローンを収集した．最後に，収集されたコードクローンをそれぞれ集約していき，記録される集約作業手順の集合を基に集約パターンを作成した．

集約パターンの有効性の評価のために，既存の集約パターンと提案する集約パターンのいずれかを用いた場合のコードクローンの集約作業時間を，12 名の学生を対象に測定した．その結果として，提案する集約パターンを利用した場合の方が，既存の集約パターンを利用した場合よりも作業時間が短くなり，特にコードクローンの特徴の詳細化によって細分化された集約プロセスにおいて，作業時間の短縮が見られた．

主な用語

コードクローン
リファクタリング
パターン

目 次

1　はじめに

コードクローンとは，ソースコード内の同一または類似するコード断片のことであり，コピーアンドペースト等によってソフトウェアに作成される [3]．例えば，あるコードにバグが発見された場合，そのコードのコードクローン全てに対し，同様の修正作業を行うべきかどうか検討をする必要がある．このように，ソフトウエア上に多く存在するコードクローンはソフトウェアの保守作業を困難にする要因と言われている [15]．ある大規模ソフトウェアでは約半数のモジュールに何らかのコードクローンが存在していることが確認されており [24]，近年のソフトウェアの大規模化によるコードクローンの保守コストの増加は，ソフトウェア工学上の課題の 1 つとして挙げられる．コードクローンを 1 つに集約しソースコード上から除去することは，バグ発見時に修正の検討を必要とする箇所が減るため，保守コストの削減に繋がる．そのために，コードクローンに対するリファクタリングの適用が検討される．

リファクタリングとは，ソフトウェアの外部的振る舞いを変えずに内部構造を変更し，ソースコードの可視性や再利用性の高い設計へと改善する技術である [7, 17]．リファクタリングを適用して安全にコードクローンを除去することで，コードクローンの問題の多くを解決することができる．リファクタリングの優れた手法として，熟練した作業者のリファクタリング作業経験を参照可能にした，リファクタリングパターンが提供されている．リファクタリング作業初心者は，それを参照することで経験者の作業を学習し，自身のソースコードに対して有効な修正作業を適用することができる．リファクタリングパターンの代表的なものとして，Fowler や Kerievsky が提供しているものがある [6, 7, 12]．それら既存のリファクタリングパターンは，オブジェクト指向に従ったリファクタリング手法を幅広く提供している．本研究では，リファクタリングパターンの中で，コードクローンの集約について述べたものを集約パターンと呼んでおり，代表的なものとして Fowler が提案しているものを利用することができる [7]．しかし，既存の集約パターンは，コードクローンの集約を特に考慮して作成されたものではないため，コードクローンの詳細な特徴に対応する作業手順が曖昧に記述されている．そのため作業者は，自身の判断によって詳細な集約作業を行う必要がある．よって，集約作業初心者が既存のパターンを適用する場合には，自身で行う集約作業を判断するための経験および知識が不足するため，バグを含む修正を行う可能性がある．

そこで本研究では，コードクローンの分類を詳細化することによって，細分化された作業プロセスを持つ集約パターンを作成し，集約作業初心者の支援を行うことを目的とする．また，既存のパターンとの比較実験により，作成した集約パターンの有効性の評価を行った．

詳細な分類に基づいた集約パターンを作成するため，まず，コードクローンの集約に特化した分類を作成した．具体的には，8 つの分類基準を定義し，各分類基準に関する分類木を作成することで，コードクローンの分類の詳細化を実現した．次に，詳細化された分類に基

づいて対象コードクローンの分類情報を出力するコードクローン自動分類ツールを作成した．さらに，作成したツールを用いて指定した特徴を持つコードクローン群を集約し，集約作業の手順集を作成した．最後に，作成された集約作業の手順集を基に集約パターンを作成した．

次に，作成した集約パターンの有効性を既存パターンとの比較実験によって評価した．比較実験では，提案する集約パターンの有効性を評価するため，12 名の学生を対象に，既存のパターンを用いた集約作業に要した時間と提案する集約パターンを用いた集約作業に要した時間を計測し，比較を行った．結果として，既存のパターンを用いる場合よりも，提案する集約パターンを用いる場合の方が要する作業時間が短くなった．特にコードクローンの詳細な分類によって導かれた，細分化された作業プロセスにおいて大きな作業時間の短縮が見られたことにより，集約パターンの作成手法および提案する集約パターンの有効性が確認された．

以降，2 章では本研究に関わるコードクローンおよびコードクローンのリファクタリングパターンについて説明する．3 章では，コードクローンの集約パターンを作成するための手法，および本手法を用いて作成された集約パターンを提案する．4 章では，提案する集約パターンの既存パターンとの比較実験の結果を述べる．5 章では，関連研究について述べ，6 章でまとめと今後の課題について述べる．

2　背景

2.1　コードクローン

2.1.1　コードクローンの定義

コードクローンとは，ソースコード内の同一または類似するコード断片のことを意味する [3]．コードクローンを検出するために，様々なコードクローン検出ツールが提案されている [25]．それらの検出ツールは，それぞれに異なったコードクローンの定義を持つ．Bellon は，コードクローン間の違いの度合に基づいて，コードクローンを以下の 3 つに分類している [4]．

Type 1
空白やタブの有無，括弧の位置などのコーディングスタイルを除いて，完全に一致するコードクローン．

Type 2
変数名や関数名などのユーザ定義名，また変数の型などの一部の予約語のみが異なるコードクローン．

Type 3
Type2 における変更に加えて，文の挿入や削除，変更が行われたコードクローン．

2.1.2　コードクローンの発生原因

過去の文献によると，コードクローンの発生原因として以下のものが挙げられる [3, 13]．

既存コードのコピーアンドペーストによる再利用
最近のソフトウェア設計技術には，構造化設計プロセスや再利用技術が存在する．しかし，レガシーコードは，図 1 のような，コピーアンドペーストによる既存のコードの場当たり的な再利用によって作成されたものを多く含む．

定型処理
定義上簡単で頻繁に用いられる計算．例えば，給与税や，キューの挿入，データ構造へのアクセスなどの処理を指す．

プログラミング言語の仕様上の機能欠如
抽象データ型や，ローカル変数が用いられない場合には，同じようなアルゴリズムを持った処理を繰り返し書かなくてはならないことがある．

パフォーマンスの改善

リアルタイムシステムなど時間制約のあるシステムにおいて，インライン展開など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書くことによってパフォーマンスの改善を図ることがある．

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，識別子名等の違いはあろうとも，あらかじめ決められたコードをベースにして自動的に生成されるため，類似したコードが生成される．

偶然

偶然に，開発者が同一のコード片を書いてしまうことがあるが，大きなコードクローンになる可能性は低い．

2.1.3 コードクローン検出ツール CCFinder

本項では，コードクローン検出ツールの代表例として，CCFinder [10] に関して述べる．

はじめに，CCFinder におけるコードクローンの定義，およびコードクローンに関する語句について述べる．

ある系列中に存在する二つの部分系列 α，β が等価である時，C(α , β) と書き，α と β は互いにクローンであるという．そして，クローンである α と β の組をクローンペアと呼ぶ（図 2）．また，α を含む同値類を α のクローンセットと呼ぶ（図 2）．任意の α , β に対して C(α , β) ならば，それぞれの部分系列 α ’, β ’ に対して C(α ’, β ’) が成り立つ．また，任意の部分系列 α ”, β ”に対して C(α ”, β ”) ではなく，かつ C(α , β) ならば，α , β を極大クローンペアと呼ぶ．さらに，ソースコード中でのクローンを特にコードクローンと呼ぶ [21].

CCFinder は，単一または複数のファイルのソースコードの中に含まれる極大クローンペアを検出し，それらのソースコード上の位置情報を出力する．CCFinder の主な特徴を以下に述べる．

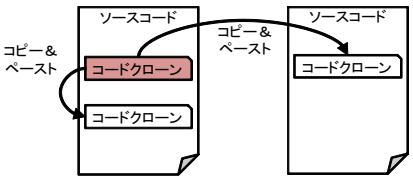


図 1: コードクローンの発生

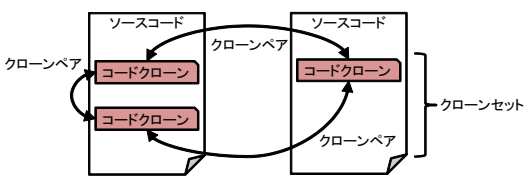


図 2: クローンペアとクローンセット

細粒度のコードクローンを検出

プログラムの字句解析を行うことにより，トークン単位のコードクローンを検出を行う．

大規模ソフトウェアを実用的な計算時間とメモリ使用量で解析

Pentium3 650MHz 1GB RAM の実行環境において，10MSLOC のソースコードに対して，68 分という計算時間でコードクローンの解析を行うことが可能である．

複数のプログラミング言語に対応

言語依存する部分を小さいサイズに制限するように実装されているため，多くのプログラミング言語に対応できる．

実用性の低いコードクローンを除去

単一行の短い文や初期化のための連続する文など，クローンではあるが，ユーザにとって有用ではない可能性の高いクローンは，ヒューリスティックな知識により除去する．

ある程度の違いを吸収

コピーアンドペーストの際に，コード断片にはわずかな修正が加えられることがある．そこで CCFinder は，ユーザ定義名や定数のパラメータ化，および名前の正規化などによってその違いを吸収する．

CCFinder は，Bellon のコードクローンの定義 [4] におけるタイプ 1 およびタイプ 2 のクローンを検出することが可能である．また，Ueda らの手法 [19] を用いることで，タイプ 3 のコードクローンも検出することも可能である．本研究では，CCFidner によって検出されるタイプ 2 のコードクローンを対象にしている．

2.1.4 コードクローンの問題点

プログラム中にコードクローンが存在すると，そのプログラムの保守は困難になる可能性がある．例えば，図 3 のように，あるコード断片にバグが含まれた場合，そのコード断片に行った修正作業を，対応するコードクローン全てに対して行うかどうかを検討する必要がある．特に大規模なソフトウェアを対象とする保守作業において，大量のソースコードの中からコードクローンを探し出し，その全てに修正の検討を行うことは，高い修正コストを必要とする．また，アウトソーシングなどによって，あるソースコードに対する開発者と保守者が異なる場合は，特にこの修正コストが高くなる．投資効果の見積もりが難しいソフトウェア保守工程において，このことは非常に重要な課題である [9, 11, 14, 15, 20]．

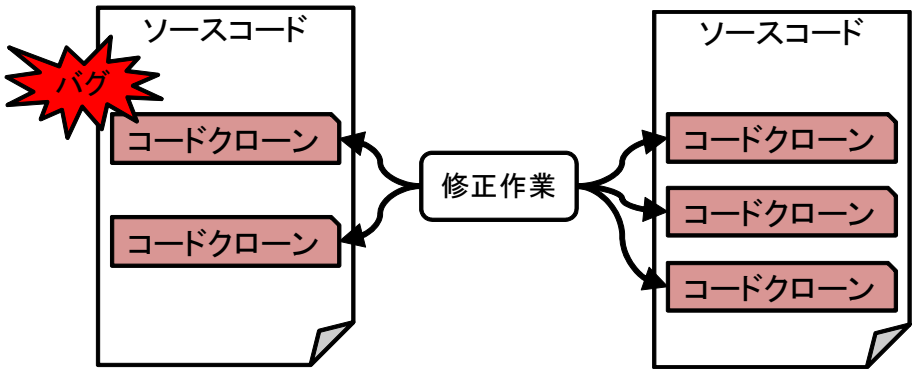


図 3: コードクローンの問題点

2.2 コードクローンのリファクタリング

コードクローンには，2.1.4 項で述べたソフトウェア保守上の問題点がある．そこで，コードクローンの問題の解決を支援するリファクタリングと呼ばれる技術を紹介する．

2.2.1 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な振る舞いを保ちつつ，ソフトウェアの内部構造を変化させる技術を指す [7, 17]．リファクタリングはソフトウェアの設計を改善することを目的としており，可読性や再利用性の向上を図ることができる．Fowler は文献 [7] の中で，コードクローンを重複するコード片として，リファクタリングすべき対象として挙げている．

2.2.2 コードクローンの分類とリファクタリングの関係

コードクローンの集約に関するリファクタリングにおいて，対象とするコードクローンの分類は，コードクローンの集約プロセスに大きな影響を与える．例えば，コードクローンが完全に一致している場合は，コードクローンを単純に 1 つのモジュールに移動すれば解決する (図 4)．しかし，コードクローンが完全に一致しない場合は，クローンペア間の差異を取り除く必要が出てくる．(図 5)．つまり，クローンペアのコード断片が一致するかどうかという特徴は集約プロセスに影響を与える．このように，集約プロセスは対象とするコードクローンの特徴に依存し，コードクローンの分類が詳細になれば，対応する集約プロセスも詳細に記述することができる．

2.2.3 Fowler によるコードクローンの分類

本稿ではコードクローンの集約に関するリファクタリングパターンのことを，集約パターンと呼ぶ．書籍 [7] の中で，Fowler はコードクローンを大きく 5 つに分類しており（表 1），それぞれの分類に対して適用可能な集約パターンを提案している．作業者は，自身が対象と

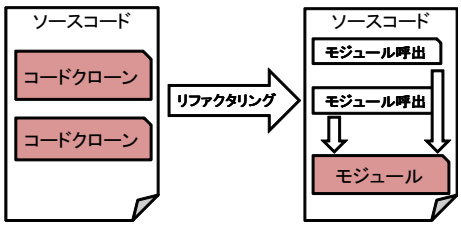


図 4: リファクタリングの例 1

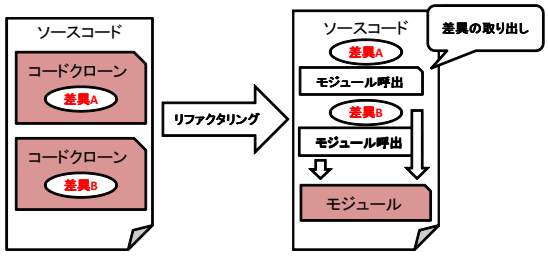


図 5: リファクタリングの例 2

表 1: コードクローンの集約に関する Fowler のリファクタリングパターン

クローンの特徴	対応する集約パターン
同一クラス内に重複したコードがある	「メソッドの抽出」
兄弟クラス間に重複したコードがあり，コードが完全に一致する	「メソッドの抽出」,「メソッドの引き上げ」
兄弟クラス間に重複したコードがあり，コードが類似する	「メソッドの抽出」,「Template Method の形成」
兄弟クラス間に重複したコードがあり，異なるアルゴリズムで同様の処理を行っている	「アルゴリズムの取り換え」
関係のないクラス間に重複したコードがある	「クラスの抽出」

しているコードクローンがどの分類に当てはまるのかを確認し，提案された集約パターンの作業手順に従ってコードクローンの集約作業を行う．

2.2.4 Fowler のリファクタリングパターン

Fowler の提案するリファクタリングパターンは，以下の 5 つの項目から構成されている．

- 名前

  リファクタリングの語彙を形成する上で重要な意味を成す．

- 要約

  リファクタリングの必要となる状況や行うことを短くまとめたもの．

- 動機

  リファクタリングを行うべき理由，および状況．

- 手順

  リファクタリングの実施方法．

- 例

  簡単なリファクタリングの使用例．

以下，書籍 [7] の中から，コードクローンの集約に関する Fowler のリファクタリングパターンの概要を紹介していく．

メソッドの抽出

ひとまとめにできるコード片がある場合に，コード断片を新たなメソッドとして定義し，抽出されたコード断片を抽出先のメソッドへの呼び出し文に置き換える．メソッドの抽出は，頻繁に行われるリファクタリングパターンの 1 つであり，様々な自動化手法やツールが提案されている [8, 22].

メソッドの引き上げ

同じ結果をもたらすメソッドが複数のサブクラスに存在した場合，それらを親クラスに引き上げる．

Template Method の形成

異なるサブクラスの 2 つのメソッドが、類似の処理を同じ順序で実行しているが，各処理が異なる場合，元のメソッドが同一になるように，各処理を同じシグニチャを持つメソッドとして，親クラスに引き上げる．

アルゴリズムの取り換え

複数のメソッドが同じ処理を異なるアルゴリズムで実装している場合，わかりやすい方を選択し，その他をそのわかりやすいものに置き換える．置き換えたメソッドに関してその他の集約パターンを適用することができる．

クラスの抽出

2 つのクラスでなされるべき作業を 1 つのクラスで行っている際に，新たにクラスを作って，適当なフィールドとメソッドを元のクラスからそこに移動する．

2.2.5 Fowler のリファクタリングパターンの利点と問題点

Fowler の提案するリファクタリングパターンは，系統立ったリファクタリング手法を記述しているため，コード中にバグを加えずにソフトウエアの構造を体系的に改善することができる．しかし，提案されているリファクタリングパターンは，コードクローンの集約を特に考慮して作られていないため，コードクローンの特徴は 5 つの分類にとどまっている（表 1）．集約プロセスの記述は対象となるコードクローンの特徴の分類の詳細度に依存するため，既存パターンの分類の詳細度では，作業者が参照する作業プロセスの記述が十分なものにならないと考えられる．そのため，既存のパターンを適用する場合，作業者が独自の判断で追加的な工夫をする必要が起こり得る．経験や知識の少ない作業初心者にとってこの状況は，集約作業の中断やバグを含むコードの導入などのプロジェクトの失敗を招く可能性を含む．そこで，より詳細なコードクローンの分類に基づいて作成されたパターンがあれば，細

分化された集約プロセスを提供することができるため，既存パターンにおける曖昧な集約プロセスが引き起こす問題を解決できると考えられる．

3　コードクローン集約パターン作成手法

本研究では，クローン分類の詳細化に基づく集約パターンの作成を行う．本章では，集約パターンの作成のための手法，および作成した集約パターンについて述べる．

図 6 に，集約パターン作成のための概略図を記す．まず，コードクローンを集約する上で考慮すべき分類基準および分類木によるコードクローンの分類を定義し，それら分類を用いてコードクローンの特徴を表現可能にした．次に，定義した分類を基に，クローンリストのコードクローンを特徴別に自動的に分類するコードクローン自動分類ツールを実装した．さらに，このツールを利用して頻出するコードクローンの特徴を調査し，頻出する特徴を持つコードクローンを取得した．最後に，取得されたコードクローンをそれぞれ集約していき，収集された集約作業手順から集約パターンを作成した．以降，集約パターン作成のために行う各手順について説明する．

3.1　詳細なコードクローン分類

本節では，細分化された集約作業プロセスを提供するために定義した，コードクローンの特徴の詳細な分類について述べる．

3.1.1　分類基準の定義と分類基準ごとの分類木の作成

まず，コードクローンの特徴を分類するために，分類基準を定義した．次に，定義した基準に対してそれぞれ分類木を作成した．それぞれの分類基準は，コードクローンを集約する際に特に考慮すべきコードクローンの特徴に関する項目であり，各分類基準の分類木における分類値は，その特徴を持つコードクローンの集約方法に影響する．本研究では，以下に示す 8 種類の分類基準と対応する分類木を定義した．

クローンセットのコード断片間の差異

コードクローン関係にある複数のコード断片間の，完全に一致しない部分である識別子の種類を分類する．差異を含むコードクローンは単純な集約ができない．そのため，差異である識別子の特徴を分類し，分類それぞれに対して適切な対応を行う必要がある．そこで，差異となる識別子がどのような役割を持つのか調査し，図 7 に示す分類木を作成した．差異に関する分類木は，差異がない場合，パッケージ名が異なる場合，各種宣言が異なる場合，各種参照が異なる場合，リテラルが異なる場合，演算子が異なる場合，および，差異が多数存在し，構造のみが一致する場合として大きく分類した．特に，宣言が異なる場合と参照が異なる場合については，さらに詳細な分類を作成した．このクローンペアの差異に関する分類は，同時に複数の該当分類を持つこと

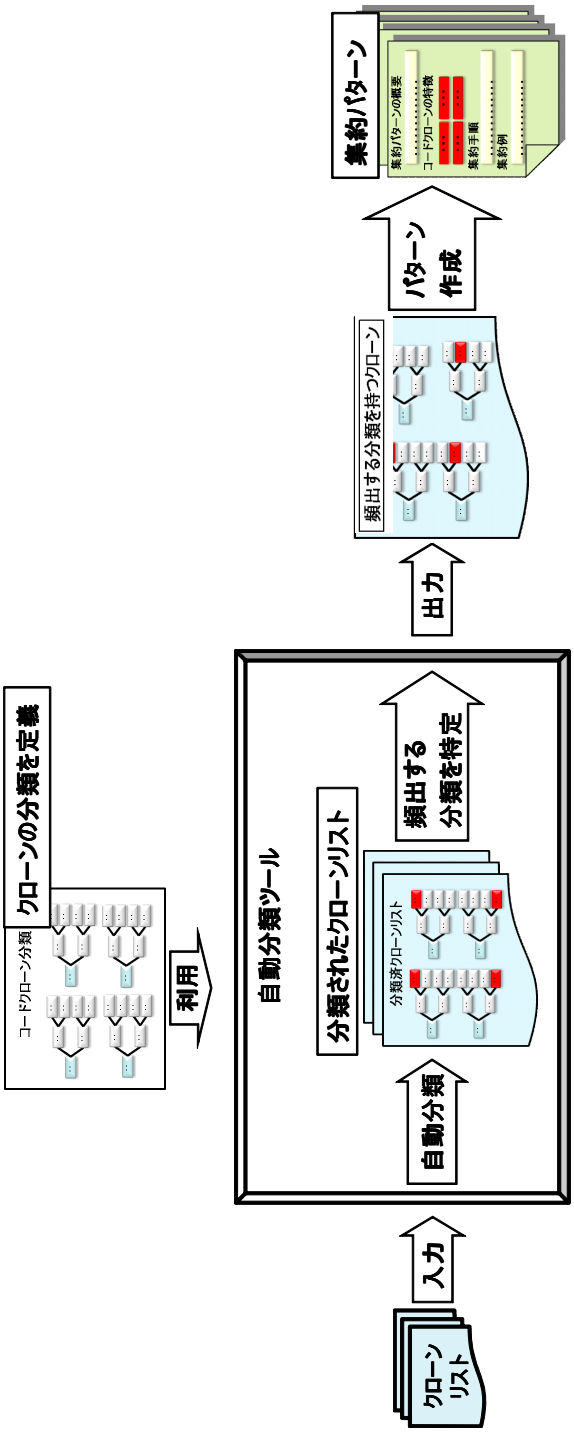


図 6: パターン作成のための概略図

を許容する．例えば，対象とするコードクローンの差異に関する特徴として，“メソッド宣言の戻り値の型” であり “メソッド名が異なる”，というように複数の特徴を同時に持つことがある．

クローンセットのコード断片間のクラス階層位置関係

コードクローン関係にある複数のコード断片間の，クラス階層上の位置関係を分類する．コードクローンを集約する際に，集約したコード断片を配置すべきクラス階層位置は，集約される元のコード断片間のクラス階層位置関係に依存する．そこで，図 8 のようにクラス階層位置関係に関する分類木を作成した．図 8 では，クラス階層位置関係を，パッケージの位置関係，およびクラスの継承関係で分類している．

コード断片の粒度

コードクローン関係にあるコード断片の粒度を分類する．集約するコード断片の粒度によって，集約方法およびモジュールの単位が変わってくるため，図 9 のように分類木を作成した．コードクローン部の粒度は，各プログラム上のまとまりごとに場合分けをしている．図 9 のクラス単位やメソッド単位とは，Java 言語における，1 つのクラスとしてのまとまり，1 つのメソッドとしてのまとまりを指す．また，ブロック単位とは，if 文や while 文などのブロックとしてのまとまりを指す．そして，単語・行単位とは，上記のようなまとまりをもたない表現の列挙を指す．

所属メソッドのローカルスコープ変数への参照関係の有無

集約対象のコード断片における，所属するメソッド内のローカル変数への参照関係の有無を分類する．この関係は，集約先モジュールに与えるべき引数の種類に依存する．

所属メソッドのローカル変数への変更関係の有無

集約対象のコード断片における，所属するメソッド内のローカル変数の変更関係の有無を分類する．この関係は，集約先モジュールが返すべき戻り値の種類に依存する．

制御構造要素の有無

集約対象のコード断片における，制御構造要素の有無を分類する．その有無によって，抽出する範囲を検討する必要がある．

instanceof 演算子の有無

集約対象のコード断片における，instanceof 演算子の有無を分類する．その有無によって，モジュール化の際の処理が異なる．

break 文の有無

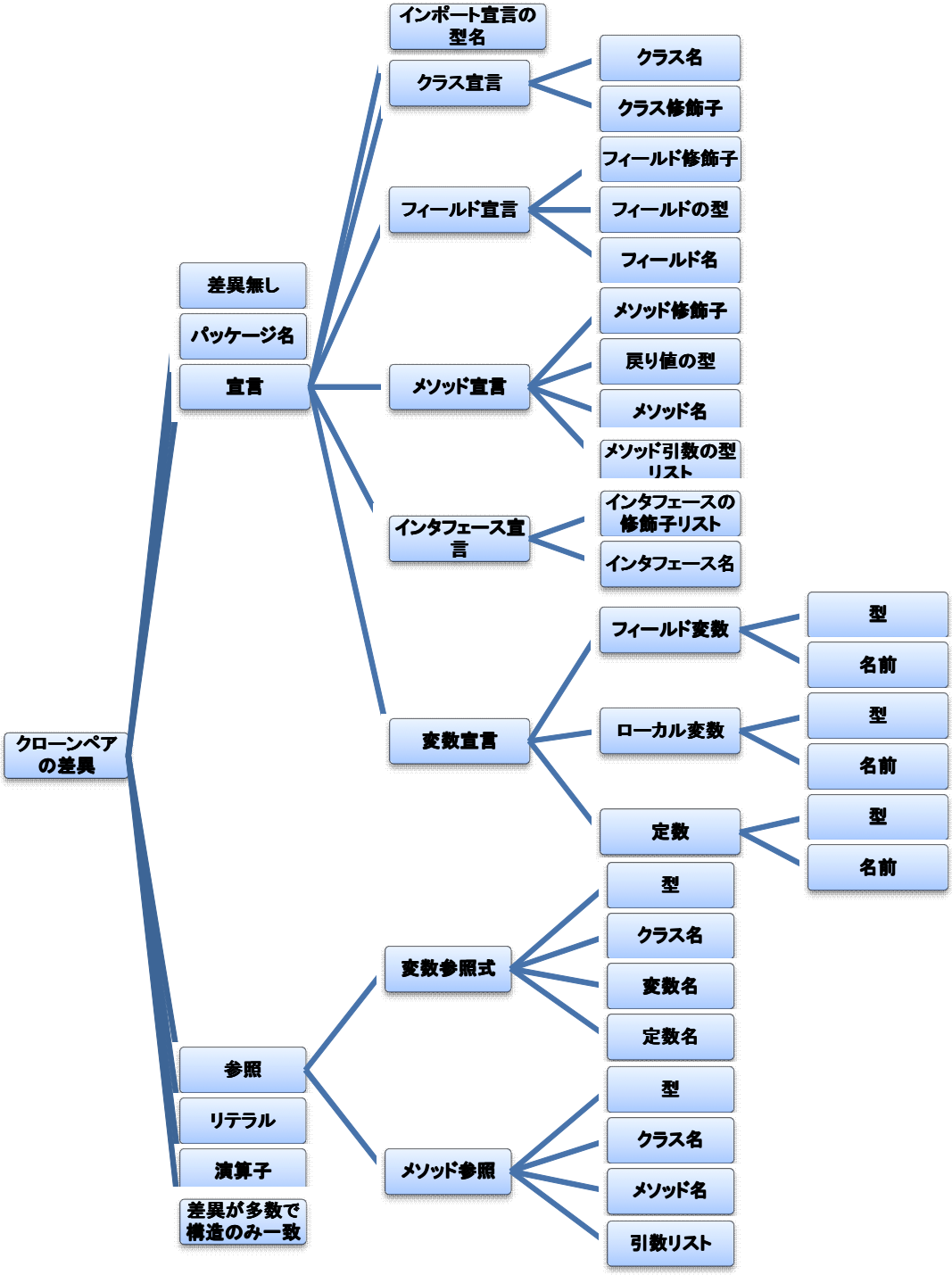


図 7: クローンセットのコード断片間の差異

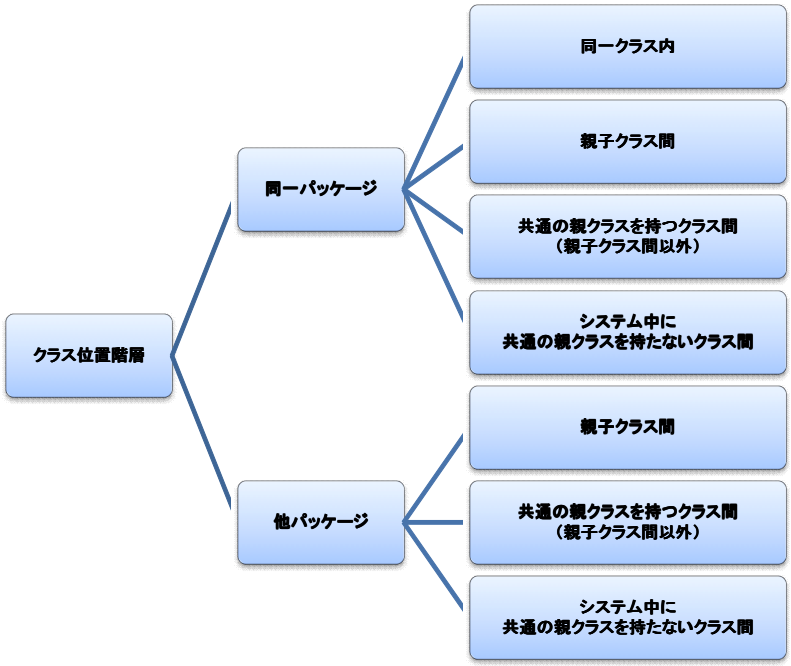


図 8: クローンセットのコード断片間のクラス階層位置関係

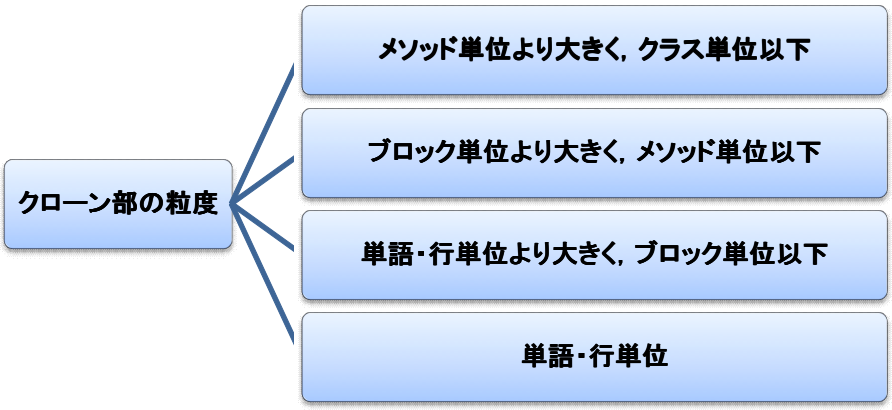


図 9: コード断片の粒度

集約対象のコード断片における，break 文の有無を分類する．その有無によって，モジュール化の際の処理が異なる．

3.1.2　分類を用いたコードクローンの特徴の表現

コードクローンの詳細な特徴を表現するために，作成した分類基準と基準ごとの分類木を用いてコードクローンの特徴を表現する．図 10 に示されるように，分類基準ごとの分類木の分類値の組み合わせを 1 つのコードクローンの特徴とする．図 10 の例で言えば，分類基準 “クラス位置階層関係” と，分類基準 “クローン断片の粒度” の分類木に対して，それぞれ該当する分類値を選択する．そして，分類基準 “クラス位置階層関係” の分類値が，“同じパッケージの同じクラス内にある” であり，分類基準 “クローン断片の粒度” の分類値が，“メソッド単位より小さくブロック単位以上” であるという分類値の組み合わせを記述することによって，このコードクローンの特徴を表現することができる．

3.2　コードクローン自動分類ツール

3.2.1　ツールの目的

コードクローンの集約パターンを作成するためには，特定の特徴を持つコードクローンを収集する必要がある．そこで，3.1.1 項で紹介したコードクローンの分類に従って，コードクローンを自動的に分類し，指定したコードクローンの特徴を持つコードクローンを表示するコードクローン自動分類ツールを作成した．

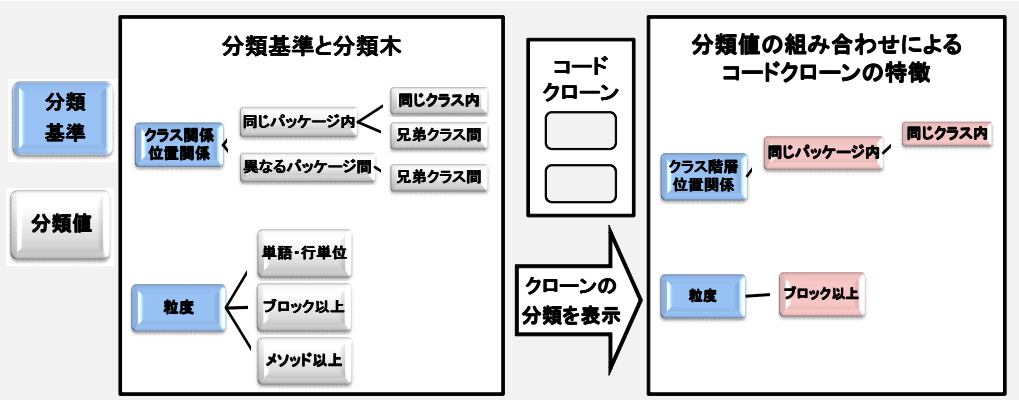


図 10: 分類を用いたコードクローンの特徴の表現

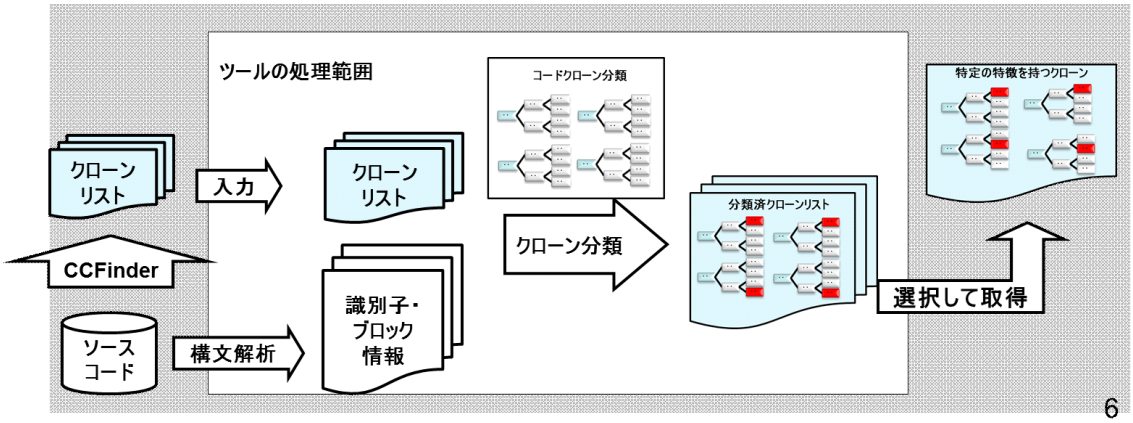


図 11: コードクローン自動分類ツールの入出力

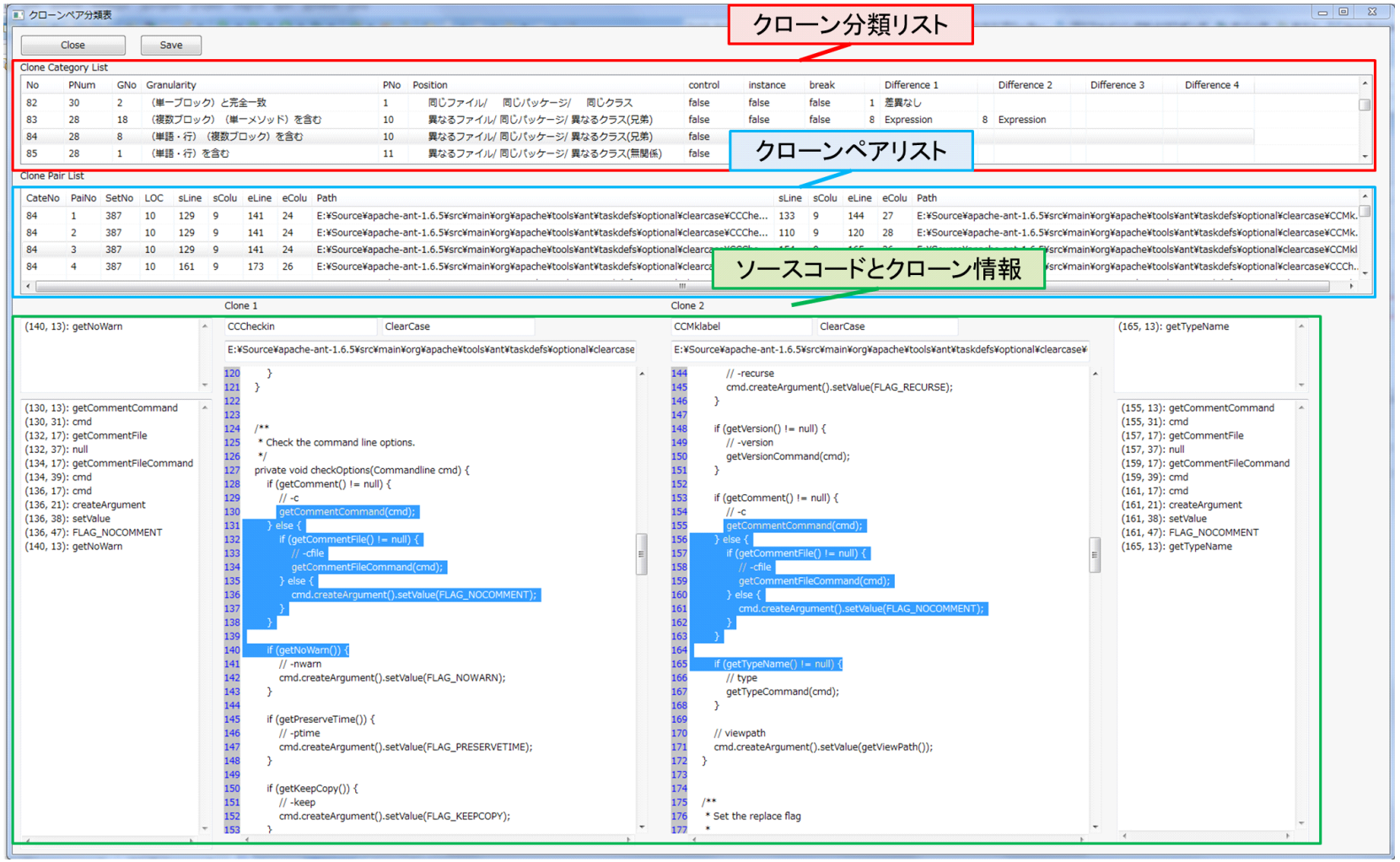


図 12: コードクローン自動分類ツールのインタフェース

3.2.2 ツールの動作

コードクローン自動分類ツールは，図 11 のように，コードクローン検出ツール CCFinder の出力ファイル情報を利用し，各クローンセットの特徴をそれぞれ分類する．分類の際は，3.1.1 項で定義したコードクローン分類に従う．そして，集計した分類結果を出力する．また，指定された分類を持つコードクローンを収集して出力する．

自動分類ツールの分類の動作として，まず，ソースコードの構文解析による識別子情報とブロック情報を取得を行う．識別子情報は，ソースコード内に含まれるすべての識別子の位置情報を表し，ソースコード上の行番号情報，所属パッケージ名，所属クラス名，および所属メソッド名などを保持する．また，ブロック情報は，ソースコード内に含まれるすべてのブロックの範囲とそのブロックの役割を表す．ブロックの役割とは，そのブロックが if 文，assert 文，switch 文，do 文，for 文，while 文などのいずれに該当するのかを指す．次に，コードクローン検出ツール CCFinder による出力情報によって得られたコードクローンの位置情報と，識別子情報およびブロック情報を比較し，各コードクローンがどのような特徴を持つコード断片であるのかを確認する．最後に，3.1.1 項で定義した分類基準ごとの分類木に対応する分類値を選出することで，各コードクローンの特徴を分類する．

図 12 に，コードクローン自動分類ツールのインターフェースを記す．画面上部のリストが，全ての分類情報を指すクローン分類リストであり，該当する特徴を持つクローンペアの数で昇順に整列されて表示される．1 つの行の中に，分類番号，および 3.1.1 項で定義した分類基準ごとの分類値の組み合わせが提示されており，1 つの行情報が 1 種類のコードクローンの特徴を表す．このリストを参照することで，頻出するコードクローンの特徴を調査する．

また，分類情報のリスト中の 1 つの行を選択すると，その行が指すコードクローンの特徴を持つクローンペアのリストが画面中部に表示される．クローンペアのリストの各要素は，それぞれ，分類番号，クローンクラス番号，クローンペア番号，クローン部の行数，および二つのクローン断片の位置情報を持つ．

次に，クローンペアのリストの中の一つの行を選択すると，その行が指すクローンペアのソースコードが画面下部に表示される．画面下部の左右に分かれて，クローンペアのそれぞれの位置情報，ソースコード情報，クローン部の識別子情報，およびクローン部間の差異情報が表示される．集約パターン作成者は，この下部の情報を見ながらコードクローンの集約手順を検討する．また，ツールが提供する機能として，画面上に提示されている分類情報リストを CSV 形式で出力することが可能であり，コードクローンの分類を集計するために利用する．

3.3　集約作業手順の収集

より高い信頼性と適用可能性を持つ集約パターンを作成するために，特定のコードクローンの特徴に対して複数の集約作業手順を収集する (図 13)．そのために 3.2 節で紹介したコードクローン自動分類ツールを利用して，頻出するコードクローン分類を持つソースコード上の複数のコードクローンを取得する．そして，それらのコードクローンに対して集約を試み，その集約作業手順を記録する．各集約作業において，テストケースで動作を確認することで，収集する集約作業手順がソースコードの一貫性を損なわないことを確認しながら作業を行う．結果として，本稿で提案する集約パターンが対象とするコードクローンの特徴に対して，10 種類の集約作業手順を収集した．

3.4　コードクローン集約パターンの作成と妥当性保証

3.4.1　パターンの作成

図 13 に記すように，3.3 節において収集したコードクローンの集約作業手順の共通部分を，集約パターンとして記述した．提案する集約パターンの場合では，収集した 10 種類のコードクローン集約作業手順から，共通する集約作業プロセスを記述した．特異な集約作業プロセスを含む場合は，注釈としてそれらプロセスを記述し，適宜本文中から参照することができるように集約パターンに記載した．

また，集約パターンが対象とする特徴を持つクローンペアを自動分類ツールを用いて別途収集し，集約パターンを適用した．その際，集約パターン適用前後のソフトウエアの挙動が

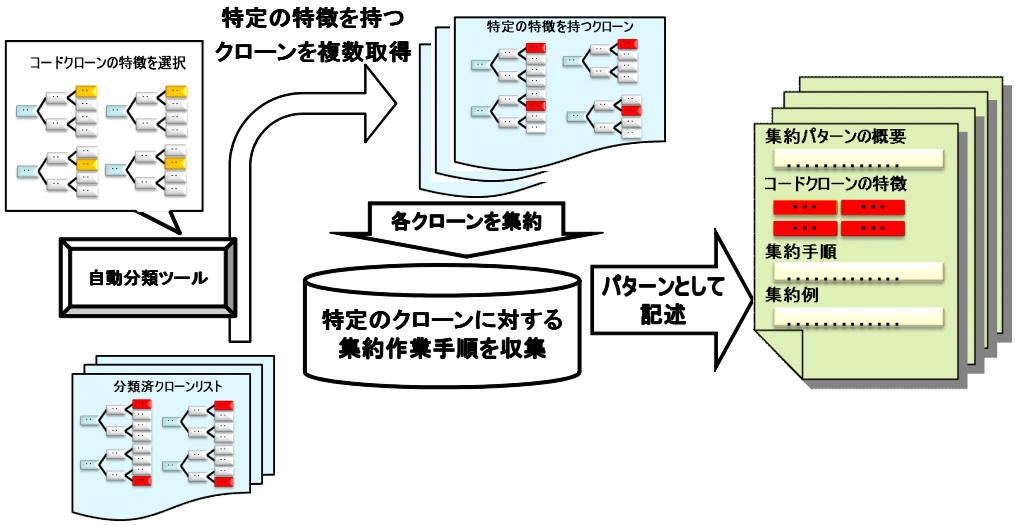


図 13: コードクローン集約パターン作成の流れ

変わらないことを，事前に準備したテストケースを用いて確認することで，集約パターンの妥当性を確認した．

3.4.2 パターンの記述形式

集約パターンの記述形式は，Fowler の用いているものに準拠する．しかし，対象とするコードクローンの分類については，3.1.2 項で示したコードクローンの特徴の表現方法を利用している．以下に，本稿で採用した記述形式について述べる．

- 名前

  本パターンの名前である．名前として集約パターンの対象及び作業内容を慨したものを用いる．

- 対象とするコードクローンの特徴

  本パターンを適用することのできるコードクローンの特徴を表す．コードクローンの特徴は，3.1.2 項のように，各分類基準に対応する分類値の組み合わせによって表現する．

- 動機

  本パターンを適用するべき状況について述べる．

- 集約手順

  作業者がコードクローン集約作業を行う際の具体的な手順を表す．集約手順の各プロセスの中には，詳細な対象とするコードクローンの特徴に対応した分岐や，細かい留意点を説明する注釈が含まれる．

- 具体例

  実際のソースコードに対して本パターンが提案する集約作業を行った様子を，集約手順の各プロセスごとに示している．具体例を示すことで，作業者が集約手順を理解する支援となる．

3.5 提案する集約パターン

集約パターンの作成手法に基づいて，頻出するコードクローンの分類に対して集約パターンの作成を行った．本節では，作成した集約パターンを以下に提案する．

- 名前
  差異がメソッド呼び出しであるコードクローンのメソッド抽出

表 2: 対象とするコードクローンの特徴

分類基準名	分類値
クローンセットのコード断片間の差異	メソッド参照のメソッド名
クローンセットのコード断片間のクラス階層位置関係	同一パッケージ・同一クラス内
コード断片の粒度	メソッド単位以下
所属メソッドのローカル変数への参照の有無	有
所属メソッドのローカル変数への変更関係の有無	無
制御構造要素の有無	無
instanceof 演算子の有無	無
break 文の有無	無

- 対象とするクローンの特徴
  各分類基準ごとの分類値を表 2 に示す.

- 動機

  クローン関係を持つコード断片は，必ずしもそのコード断片が完全に一致するとは限らない．コードクローンを除去する際に，クローンペア間に一致しない部分がある場合，コード断片の一致しない部分を処理し，メソッドの抽出を行うことによってコードクローンを除去する必要がある．そのような場合に，兄弟クラス間のクローンペアであればテンプレートメソッドの適用が考えられる．しかし，同じクラス内の場合，テンプレートメソッドの適用を行うことは難しくなる．このように，同じクラスに含まれるクローンペアが存在し，クローンペアが差異を含む場合に，本パターンを適用することができる.

- 集約手順

  本集約パターンの集約プロセスを図 14 に示す．各プロセス中の*印が参照を表し，図 15 における対応した番号の説明への参照を表す.

- 具体例
  集約プロセスを具体的にイメージするために，ソースコード上での集約パターンの適用例を作成した．詳細は付録 [A] に記載する.

集約プロセス	作業概要	作業内容
1	差異の調査	1-1. クローン部とクローンペア間の**差異を特定する**。 メソッド抽出を用いたクローン除去において弊害となる差異を除去するため。 1-2. クローン部の差異である『メソッド呼び出しの戻り値』もしくは差異である『変数』の**型が、クローンペア間で一致していること**を確認する。
2	差異の除去 （コード片の表現の統一）	【①差異の型が一致している場合】 2-1. 差異である『メソッド呼び出しの戻り値』の型 もしくは差異である『変数』の型を持つ新しい変数を、 コード片と同じメソッド内において、抽出されるコード片よりも前でそれぞれ同じ名前で宣言する。(*1) 2-2. (集約プロセス2-1)で宣言した新しい変数に、 差異である『メソッド呼び出しの戻り値』もしくは差異である『変数』を代入する。 2-3. 抽出するコード片の差異である『メソッド呼び出し部』もしくは差異である『変数』を、 (集約プロセス2-1)で宣言した**新しい変数と置き換える**。 これによってクローンペア間の差異を無くす。 --- 【②差異の型が一致していない場合】 2-1. (*2)に記される作業を行う。
3	メソッドの作成	3-1. 同じクラス内に**新たなメソッドを作成**する。(*3) 3-2. 元のクローン部のコード片を、(集約プロセス3-1)で作成した新たな抽出先の**メソッドの本体にコピー**する。
4	クローン部によって参照される 元メソッドの一時変数の特定	作成した新たな抽出先メソッド内における、元メソッドの**一時変数(*4)への参照の有無**を調査する。 これによって、新たに作成したメソッドの引数の候補を特定する。
5	特定した一時変数の利用の調査 （抽出先メソッドの引数の決定）	5-1. (集約プロセス4)で発見された元のメソッドの一時変数が, 抽出されるコード内だけで使用（参照, 代入）されていることをそれぞれについて調査する。 これによって、新たに作成したメソッドの引数の扱いを決定する。 【①抽出されるコード内だけで使われている場合】 5-2. その一時変数が、(集約プロセス2-1)にて、 差異を置き換えるために宣言した新しい変数であるかどうかを調べる。 【①新しく宣言した変数である場合】 5-3. (集約プロセス7)において、**抽出先メソッドの引数として渡す**。 --- 【②新しく宣言した変数でない場合】 5-3. その一時変数の宣言を**抽出先メソッド内にも記述する**。 --- 【②抽出されるコード以外の部分で使われている場合】 5-2. (集約プロセス7)において、**抽出先メソッドの引数として渡す**。
6	特定した一時変数の変更の調査 （抽出先メソッドの戻り値の決定）	6-1. (集約プロセス4)で発見された元のメソッドの一時変数が、抽出されるコード内で変更されるかどうか、 それぞれについて調査する。 これによって、メソッドの戻り値を決定する。 【①全ての一時変数が抽出されるコード内で変更されない場合】 6-2. 変更を加えずに次のプロセスを行う。 --- 【②変更されるコードが存在する場合】 6-2. (*5)に記される作業を行う。
7	変数の扱い	(集約プロセス5-2)で特定した変数の中で、引数で渡す必要があるものがあれば、抽出先メソッドの引数として渡す。
8	コンパイル	この時点でコンパイルが通ることを確認する。
9	抽出先メソッドの呼び出し	9-1. 元のメソッドにおいて、抽出される各コード片を**抽出先メソッドへのコールに置き換える。** これによって、クローンペアを除去することができる。 引数に対応する変数を代入することを忘れないようにする。(*6) コードの可読性を考慮して、必要があれば、以下のプロセスを行う。 9-2. (*7)に記される作業を行う。
10	コンパイル・テスト	10-1. この時点でコンパイルが通ることを確認する。 10-2. テストケースを用いてテストを行う。

図 14: 集約作業手順

参照番号	説明
(*1)	抽出対象のクローンペアのコード片がそれぞれ同一ブロック内にある場合は、 新たな変数を一つだけ定義して、各コード片でそれぞれ利用する。 変数名は、差異となるメソッドの戻り値のクローンペア間で共通する意図の名前を付与する。
(*2)	差異である識別子を含むコード片をメソッド抽出することは困難であるため、 差異を含むメソッド呼び出しを抽出対象外へ出して、共通部分を新たなメソッドとして抽出する。
(*3)	1. 修飾子はprivateにする。 2. 戻り値の型はvoid(戻り値を返す必要があれば適宜変更) にする。 3. 名前はクローン断片の共通する作業の意図に合わせて命名する。
(*4)	元のメソッドのローカル変数とメソッドのパラメータがこれに当たる。
(*5)	1. 抽出されるコード片よりも後で参照される変数の数を調査する。 【①変更される変数が一つの場合】 2. 抽出したコードを問合せメソッドとして扱い、結果を関係する変数に代入する。 1. (集約プロセス3-1)で作成した新しいメソッドの戻り値の型を、変更される変数の型名に変更する。 2. 新しいメソッドの本体の終端部において、変更される変数をreturn文で返す。 【②変更される変数が複数の場合】 2. メソッドをそのまま抽出することはできないので、以下のプロセスのいずれかを試す 1. 可能であれば, クローン部で宣言され、クローン部より後に参照される変数をクローン部外に移動する。 2. 「一時変数の分離(128) (Fowlerのパターン)」を適用し、再度検討する。 3. 「問い合わせによる一時変数の置き換え(120)(Fowlerのパターン)」を適用して、一時変数を除去する。
(*6)	(集約プロセス5-2) において、一時変数の宣言を抽出先メソッドに移した場合、 それらが抽出されるコードの外部での宣言を削除する。
(*7)	1. クローンペアの両コード片においてそれぞれ、 (集約プロセス2-3)で差異を含むメソッド呼び出しを置き換えるために新しく宣言していた一時変数の宣言部を除去し、 メソッド呼び出しの引数には、直接差異であるメソッド呼び出し部を記述する。 2. 同様に、クローンペアの両コード片においてそれぞれ、 (集約プロセス2-3)で差異を含む変数を置き換えるために新しく宣言していた一時変数の宣言部を除去し、 メソッド呼び出しの引数には直接差異である変数を記述する。

図 15: 集約作業注釈

4　比較実験

本章は，3.5 項で提案した集約パターンの有効性について行った評価について述べる．被験者には，特定のコードクローンに対して，提案する集約パターンを用いる場合と Fowler の提案するパターンを用いる場合の両方で集約作業を行ってもらい，要した作業時間を計測した．以降，比較実験の内容および評価基準について説明した後，実験結果について述べる．

4.1　実験内容

比較実験では，まず，被験者に対して集約作業の内容と利用する集約パターンに関する事前講義を行った．次に，被験者が実際に被験者に集約作業を行った．最後に，被験者が，行った集約作業に関するアンケートに回答した．また，被験者には前もって予備アンケートを行った．以降，事前に行った集約作業内容の説明，事前講義，集約作業，およびアンケートに関して述べる．

4.1.1　集約作業内容の説明および事前講義

比較実験では，集約作業内容の説明および集約に関する事前講義を行った．これにより，被験者の作業時間がパターン以外の要因からの影響を強く受けるのを防ぎ，また被験者の集約作業に対する知識のレベルを統一した．

まず，被験者に比較実験の概要を説明し，作業内容の理解を促した．次に，被験者の集約に関する理解度を一致させるために，事前に用意したスライド資料を説明することで，実験対象となる集約パターンの作業手順の理解を促した．その後，実験環境における作業の進め方の理解を促すために，事前に用意した実環境での集約作業動画を被験者にみせることで，集約作業の実演を行った．また，Editor 上の機能の利用による作業時間の影響を防ぐために，実験環境における Editor の使い方に関する被験者の知識を確認し，必要に応じて補足を行った．さらに，作業後に行うアンケートについて，質問する項目の簡単な説明を行った．最後に，利用する集約パターンを手渡し，被験者に集約パターンを理解する時間を 5 分間与えた．

4.1.2　集約作業

集約作業では，集約に要した作業時間を計測した．被験者が集約作業を始めてから，集約作業対象であるクローンペアを 1 つのモジュールに集約し，準備したテストケースが全て成功するまでの経過時間を計測した．また，集約パターンの有効なプロセスを特定するために，作業時間を図 14 の各集約プロセスごとに計測した．

実験環境として，14.1 型のディスプレイのノート型パソコンを利用した．また，コンパイルやテストケース実行の容易性，および編集機能が充実しているという理由で，統合開発環境 Eclipse [5] を利用した．

集約作業は，1 つのクローンペアに対して 5 分から 40 分の作業時間を想定し，40 分を超過する場合はそこで集約作業を中止してもらうようにした．

被験者からの質問は，適宜受け付けた．集約方法に関する質問には，回答の程度によって計測される作業時間やテストケースの失敗回数に影響を与えるため，配布する集約パターン上に記述されている範囲内で回答した．それ以外の環境や作業手順についての質問に関しては必要に応じて適宜回答を行った．

4.1.3 アンケート

- 予備アンケート
  集約作業レベルが均一なグループ分けを行うために，各被験者の集約に関する知識を把握する予備調査アンケートを行った．内容は，Java プログラミング作業経験，コードクローンの知識，およびリファクタリング作業経験に関するもので，詳細は付録 [B] に記載する．被験者は，配布したものを事前に回答する．

- 作業後アンケート
  既存パターンと提案するパターンの有効性の比較，および提案するパターンの定性的な評価を行うために，被験者は両方の集約パターンを用いたコードクローンの集約を行った後，作業後アンケートに回答する．

  調査内容は以下の 3 点である．

  - 既存パターンと提案するパターンとの優劣
    既存パターンと提案する集約パターンの比較を行うために，被験者は以下の 2 つの基準に対してパターン間の優劣をつける．
    * パターンを用いた作業の正確性
    * パターンの学習教材としての有効性
  - 提案するパターンの有効なプロセス
    集約作業を行う上で，被験者が有効だと考えた提案する集約パターンのプロセスがあれば列挙する．
  - 提案するパターンに関するその他のコメント
    提案するパターンの記述の詳細度，不明確な記述，および不足する記述に関する意見があれば回答する．

詳細な項目については，付録 [C] に記述する．

4.2 評価基準

4.2.1 計測値の評価基準

有効な集約パターンは，適用することで集約に要する作業時間を短縮につながると考えられる．そこで，既存パターンを用いる場合と提案する集約パターンを用いる場合に要した作業時間を比較していく．比較を行うために，被験者が各クローンペアに対して要した平均作業時間を，以下の分類に分けて集計し，各分類の分類値について優劣を確認する．

分類 1 クローンペア 1 におけるパターン間の比較

学習効果が無い状態での提案する集約パターンと Fowler のパターンの作業時間を比較する．

分類 2 クローンペア 1 に対して提案する集約パターンを用いたグループのクローンペア 2 におけるパターン間の比較

一度提案する集約パターンを学習しているという前提が，その後それぞれのパターンを利用する場合の作業時間にどのような影響をもたらすのか比較する．

分類 3 クローンペア 1 に対して Fowler のパターンを用いたグループのクローンペア 2 におけるパターン間の比較

一度 Fowler のパターンを学習しているという前提が，その後それぞれのパターンを利用する場合の作業時間にどのような影響をもたらすのか比較する．

分類 4 クローンペア 2 におけるパターン間の比較

クローンペア 1 による学習効果は無視し，クローンペア 2 に対する提案する集約パターンと Fowler のパターンの作業時間を比較する．

分類 5 クローンペア 1 と同じパターンを用いたグループのクローンペア 2 におけるパターン間の比較

どちらか一方のパターンを利用してクローンペア 1 およびクローンペア 2 を集約したという条件では，クローンペア 2 においてそれぞれのパターンを利用した時の作業時間にどのような影響がでるのか比較する．

4.2.2 アンケートの評価基準

提案するパターンと Fowler のパターンの有効性の比較を行うために，作業後アンケートにおける 2 つの基準 (集約作業の正確性および学習教材としての有効性) に対して，提案するパターンが優れたパターンとして選ばれていることを確認する．

4.3 実験対象とするクローンペア

CCFinder と作成したコードクローン自動分類ツールを利用して，提案するパターンを適用することが可能なソースコードを取得した．

まず，オープンソースソフトウエアの apache-ant-1.6.5 に対して CCFinder を動作させ，クローンペアの集合を検出した．次に，その集合を入力としてコードクローン自動分類ツールを動作させ，適用するパターンの想定しているコードクローンの特徴を選択することで，パターンを適用することが可能なソースコード群を取得した．最後に，出力されたソースコード群から以下に示す 2 つのソースコードを選択した．

- 課題クローンペア 1

  課題クローンペア 1 は，JavaDoc クラスの parsePackage() メソッドにおけるコード断片 A (2021 行目から 2026 行目) とコード断片 B(2031 行目〜2036 行目) である（図 16）．図 16 においてハイライトになっているコード断片がクローン部である．

- 課題クローンペア 2

  課題クローンペア 2 は，SOSGet クラスの buildCmdLine() メソッドにおけるコード断片 A (93 行目から 97 行目) とコード断片 B(105 行目〜108 行目) である（図 17）．図 17 においてハイライトになっているコード断片がクローン部である．

テストケース

被験者の集約作業によってソースコードの外部的動作が変更されないことを確認するため，対象とするソースコードに関連するモジュールに対する JUnit のテストケースを準備し，動作の一貫性を確認した．

4.4 被験者と被験者グループ

本実験では，被験者としてコンピュータサイエンス専攻に所属する大学院生 7 名および大学生 5 名の計 12 名を選出した．被験者は，以下の条件にあう人物であることを前提とした．

- オブジェクト指向や Java でのプログラミングについて，大学の講義によるコンピュータサイエンス専攻の一般的な知識を有する

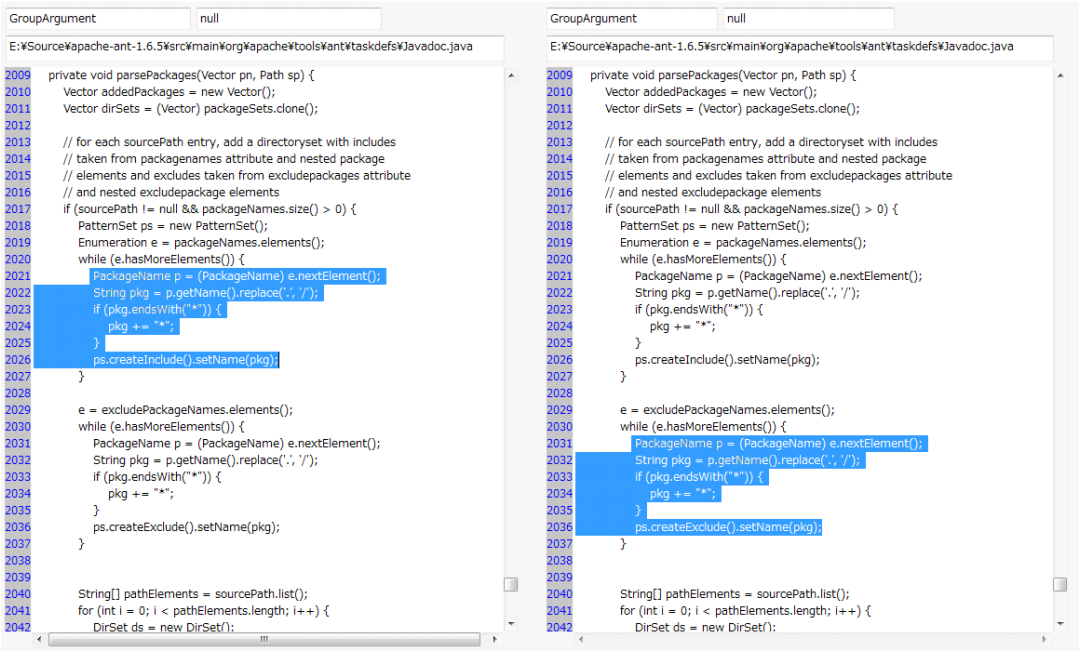


図 16: 課題クローンペア 1 のソースコード

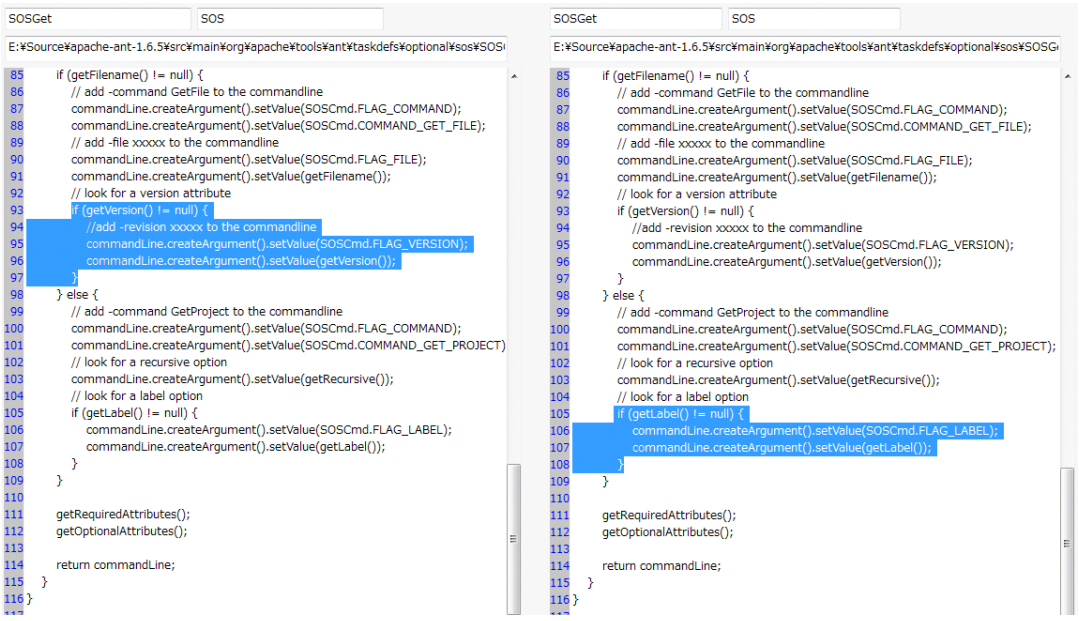


図 17: 課題クローンペア 2 のソースコード

表 3: グループ別適用パターン表

	課題クローンペア番号	
グループ名	クローンペア 1	クローンペア 2
グループ A	提案するパターン	提案するパターン
グループ B	提案するパターン	Fowler のパターン
グループ C	Fowler のパターン	提案するパターン
グループ D	Fowler のパターン	Fowler のパターン

- 企業での大規模開発の経験がない
- 業務上でリファクタリングの作業を行った経験がない

提案する集約パターンと Fowler のパターンの有効性を比較するために，被験者を適用するパターンと対象とする課題クローンペアによって 4 つのグループに分類した（表 3）．各グループの被験者の数は 3 名である．グループ間の集約作業時間を比較するため，各グループ間で想定される被験者の作業レベルが等しくなるように分類する必要がある．そこで分類は，予備アンケートで収集した被験者の集約作業に関する経験を基に行った．全ての被験者はクローンペア 1 に対する集約作業を行った後に，クローンペア 2 に対する集約作業を行った．

4.5 実験結果

4.5.1 計測結果

各プロセスの集約作業時間の平均値を計測し，分類ごとに対応するグループ間の作業時間の差を集計した．表 4 はその値を表している．各分類では，提案する集約パターンを用いた場合の平均作業時間，および Fowler のパターンを用いた場合の平均作業時間を示す．また，Fowler のパターンを用いた場合に対する提案する集約パターンを用いた場合の平均作業時間の差の値を示す．よって負の値をとっている項目は，提案する集約パターンのケースの方が，Fowler のパターンのケースよりも短い作業時間で作業を終えることができたことを表す．各行は 4.2.1 項で定義した評価のための分類に対応する計測値を示す．

表 4 の結果において，プロセス全体の集約作業時間の差を見ていくと，分類 1 および分類 5 の集約作業時間の差が特に大きい．分類 1 は，前提知識がない状態での各パターンを用いて集約作業を行ったグループである．分類 5 は，どちらか片方のパターンだけを利用して 2 つのクローンペアを集約したグループである．この 2 つの分類は提案する集約パターンと Fowler のパターンのどちらか一方を用いた場合の分類であり，これらの分類における差が

表 4: 分類別平均作業時間表 (集約作業全体)

分類名	提案パターン を適用した場合	平均作業時間 (秒)	Fowler のパターン を適用した場合	平均作業時間 (秒)	全プロセス 平均作業時間の差 (秒)
分類 1	クローンペア 1 の グループ A と B の 平均作業時間	1189	クローンペア 1 の グループ C と D の 平均作業時間	1649	-459
分類 2	クローンペア 2 の グループ A の 平均作業時間	749	クローンペア 2 の グループ B の 平均作業時間	880	-131
分類 3	クローンペア 2 の グループ C の 平均作業時間	1044	クローンペア 2 の グループ D の 平均作業時間	1363	-320
分類 4	クローンペア 2 の グループ A と C の 平均作業時間	814	クローンペア 2 の グループ B と D の 平均作業時間	804	-82
分類 5	クローンペア 2 の グループ A の 平均作業時間	749	クローンペア 2 の グループ D の 平均作業時間	1649	-615

表 5: プロセス別平均作業時間差表

	集約プロセス									
分類名	1 差異 調査	2 差異 除去	3 メソッド 作成	4 変数 特定	5 参照 調査	6 変更 調査	7 引数 設定	8 コンパイル	9 メソッド 呼出	10 テスト
分類 1	-26	-20	-24	-90	-67	-91	74	-81	31	-61
分類 2	-33	-81	-4	-16	-9	-1	-54	-4	-2	-74
分類 3	22	-127	55	-54	57	24	-102	-100	-27	-67
分類 4	2	24	-24	-31	-31	-3	17	-3	-4	-29
分類 5	-8	-160	3	-132	-15	18	-123	-109	-38	-49

表 6: 分類別平均作業時間表 (プロセス 2)

分類名	提案パターン を適用した場合	平均作業時間 (秒)	Fowler のパターン を適用した場合	平均作業時間 (秒)	プロセス 2 平均作業時間の差 (秒)
分類 1	クローンペア 1 の グループ A と B の 平均作業時間	188	クローンペア 1 の グループ C と D の 平均作業時間	208	-20
分類 2	クローンペア 2 の グループ A の 平均作業時間	180	クローンペア 2 の グループ B の 平均作業時間	261	-81
分類 3	クローンペア 2 の グループ C の 平均作業時間	213	クローンペア 2 の グループ D の 平均作業時間	340	-127
分類 4	クローンペア 2 の グループ A と C の 平均作業時間	220	クローンペア 2 の グループ B と D の 平均作業時間	197	-24
分類 5	クローンペア 2 の グループ A の 平均作業時間	180	クローンペア 2 の グループ D の 平均作業時間	340	-160

大きく開いていることから，Fowler のパターンよりも提案する集約パターンの方が有効性が高いと考えられる．

また，分類 2 と分類 3 を比較すると，分類 3 の差が分類 2 の差の 3 倍近い値を示している．分類 2 はクローンペア 1 に対して提案するパターンを既に利用していた場合であり，分類 3 はクローンペア 1 に対して Fowler のパターンを既に利用していた場合である．この結果から，クローンペア 1 において提案するパターンを先に用いて集約作業を行う場合は，1 回目の学習により，2 回目にいずれのパターンを提案されても，同様な特徴を持つクローンペアに対する集約処理を行うことが出来るということが分かるが，Fowler のパターンを先に用いた場合は，その後提案するパターンを用いた場合でも更に時間を短縮する余地があった．その余地は，Fowler のパターンには不足しているが，提案するパターンにおいて詳細化された集約プロセスが存在することによって生まれたものであり，それら集約プロセスが有効に作用しているためだと考えられる．そこで，表 5 の結果を集約プロセス別に比較していく．分類 5 に着目すると，集約プロセス 2，集約プロセス 4，および集約プロセス 7 について集約パターン間の集約作業時間に大きな差が見られた．これらの集約プロセスは，コードクローンの特徴を詳細に分類することによって記述された集約プロセスであった．

ここで，表 5 の分類 5 において大きな差が見られた集約プロセス 2 の集約作業時間について詳細に見ていく．表 6 は，集約プロセス 2 における計測結果を示す．表 6 中の各項目は，表 4 と同様の項目を集約プロセス 2 に関して示す．本研究において，集約プロセス 2 とは，クローンペア間の差異を扱う集約プロセスであり，コードクローンの特徴を詳細化することで，特に明確になった部分である．分類 5 における結果に着目すると，全体の集約作業時間を約半分に短縮することができている．このことから，特に集約プロセス 2 が集約作業時間の短縮に有効であることが確認できる．また，後述する作業後アンケートの項目 2（有効だと考えられるプロセス）の結果において，過半数の被験者がプロセス 2 は有効であると判断していた（表 8）．

以上の結果のように，集約作業全体を通して，提案する集約パターンを用いる場合の方が，Fowler を用いる場合よりも短い時間での集約作業を実現したため，提案するパターンの有効性が確認された．特に，本研究で用いる分類の詳細化によって作成される明確な集約プロセスがより集約作業の作業時間短縮に有効であることが示された．

一方，付録 [D] における図 31 における被験者 No.11 の例を見ると，被験者 No.11 は，被験者の中で唯一クローンペア 1 の作業時間よりもクローンペア 2 の作業時間の方が長くかかった．被験者 No.11 は，クローンペア 1，クローンペア 2 の両方に対して Fowler のパターンを適用するグループ D に所属する．被験者 No.11 は，クローンペア 1 において自身で考えた差異の除去方法を，クローンペア 2 でも同様に適用しようとしたが，その方法はクローンペア 2 では適用することが困難であった．つまり，Fowler のパターンを適用するには，作

業者の独自の判断が必要である場合があり，対象とするクローンペアと作業者の知識によっては作業できる場合と出来ない場合が起こり得ることが確認された．一方，グループ A（クローンペア 1，クローンペア 2 のどちらにも提案する集約パターンを利用するグループ）ではこのような状況は起こらなかったため，提案する集約パターンが既存のパターンよりも高い汎用性を持つことを示す事例が確認できた．

4.5.2 アンケート結果

作業後アンケートの結果を表 7，表 8，および表 9 に示す．各表には，集約作業に提案するパターンと Fowler のパターンの両方を利用したグループ B とグループ C の被験者である被験者 4 から 9 のアンケート結果について示している．また，表 7 の作業の正確性および学習教材の項目において，Mine とは提案するパターンを選択したということを指す．表 9 において，各コメントが肯定的であれば Positive，否定的であれば Negative と表現する．

表 7 におけるアンケート結果では，提案するパターンと Fowler のパターンについて，集約プロセスの正確性および学習教材としての有効性という観点で被験者に優劣を付けてもらったが，それぞれの観点に対して 6 人中 6 人が，提案する集約するパターンが優れているという回答をした．また，表 9 におけるアンケート結果では，作成したパターンについての肯定的な意見が多く，その有効性を被験者からのコメントから確認することができた．一方，否定的な意見としては，プロセスの記述が過多であるという意見が得られた．

4.5.3 課題の一般性

パターンの有効性の評価の為には，実用的なソースコードに対する実験を行うべきである．本実験では，オープンソースソフトウェア Apache Ant のソースコードを利用して集約パターンを作成した．Apache Ant は OS などの特定の環境に依存しにくいビルドツールとして，統合開発環境 Eclipse に標準内蔵されている [5]．また，10 年以上リリースを繰り返されており，クローン分析の対象としても頻繁に利用されている [1, 9]．これらの理由により，本実験の課題対象として妥当であると考える．

提案する詳細化の手法の一般性の為には，各分類に対応する集約パターンそれぞれについての有効性の評価実験を行うべきである．本研究では，単一のコードクローンの分類に対する集約パターンの有効性を示すにとどまっている．しかし，本実験で対象にしたクローン分類は，特に頻出するコードクローンの特徴に対して作成した集約パターンであるため，その有効性が示されたことは集約作業初心者に対して，利用性の高いパターンを提供できたと考える．

表 7: アンケート結果 (項目 1)

	グループ B クローンペア 1: 提案するパターン クローンペア 2: Fowler のパターン			グループ C クローンペア 1: Fowler のパターン クローンペア 2: 提案するパターン		
選択基準	被験者 4	被験者 5	被験者 6	被験者 7	被験者 8	被験者 9
集約作業の正確性	Mine	Mine	Mine	Mine	Mine	Mine
教材としての有効性	Mine	Mine	Mine	Mine	Mine	Mine

表 8: アンケート結果 (項目 2)

	グループ B クローンペア 1: 提案するパターン クローンペア 2: Fowler のパターン			グループ C クローンペア 1: Fowler のパターン クローンペア 2: 提案するパターン		
質問内容	被験者 4	被験者 5	被験者 6	被験者 7	被験者 8	被験者 9
評価の高いプロセス	2	1, 2, 5			2	2, 5

表 9: アンケート結果 (項目 3)

	グループ B クローンペア 1: 提案するパターン クローンペア 2: Fowler のパターン			グループ C クローンペア 1: Fowler のパターン クローンペア 2: 提案するパターン		
コメント基準	被験者 4	被験者 5	被験者 6	被験者 7	被験者 8	被験者 9
詳細度			Positive	Positive	Positive	Positive
記述方法		Negative				
不足する手順	Negative					

4.5.4 被験者の作業能力の差異

本実験では，12 名の被験者を 3 名ずつの 4 つのグループに分けた．実験結果を各グループの平均作業時間の差をとることでその有効性を判定しているので，グループ間の集約作業レベルを統一することが重要である．そのため本実験では，予備アンケートによる収集された被験者のプログラミング知識等を考慮したグループ分けを行った．この結果を利用して各グループの集約作業レベルの統一を行ったため，実験の結果はグループ間の作業レベルに依存しない有意な値であると考えられる．また，その他の解決策としては，多くの被験者を導入することによる計測値の分散の抑制が考えられる．

5　関連研究

Kapser らは，コードクローンをプログラム要素（関数や構造体など）との対応関係に基づいて分類している [11]．また，Balazinska らは，保守支援を目的として，コードのクローンの差異に基づく分類を行っている [2]．これに対して, 本研究はコードクローンを集約するという観点において，差異の分類を行った．

また，メソッド抽出に基づいてコードクローン集約を支援する手法が数多く提案されている [9][23]．これら手法を拡張し，本研究で定義した集約パターンの集約プロセスと対応させることで，特定の分類に関してはツールによる集約作業の支援が可能になると考えられる．

Ng らは，保守作業の効率向上の為のデザインパターンを用いたリファクタリングによる影響と業務経験による影響とを比較する実験を行っている [16]．この研究に対して，本研究ではコードクローンの集約に注目したリファクタリングの影響について調査を行った．

6　まとめと今後の課題

本稿では，コードクローンの分類の詳細化を利用して，コードクローンの集約パターンを作成し，その有効性を評価した．

集約パターンの作成のために，まず，コードクローンの集約に特化した分類をオープンソースソフトウェアのコードを利用して詳細化した．次に，その分類に基づき，作業者が選択したコードクローンの分類を持つソースコードを出力するコードクローン自動分類ツールを作成した．そして，作成したツールを利用して，収集した特定のコードクローンの特徴を持つソースコードを集約していき，収集される集約作業を基に集約パターンを作成した．

また，有効性の評価を行うために，12 名の学生を対象に，既存パターンもしくは提案する集約パターンを利用してコードクローンを集約してもらい，集約作業に要した時間を比較する比較実験を行った．実験の結果として，提案する集約パターンを利用した場合の方が，既存パターンを利用した場合よりも作業時間が短くなり，特にコードクローンの特徴の詳細化によって明確に記述された集約プロセスにおいて，作業時間の短縮が見られたため，提案パターンの有効性を確認できた．

今後の課題として，他のコードクローン分類に対する集約パターンの作成および評価を行うことが挙げられる．また，本研究ではオープンソースソフトウェアを利用して評価実験を行ったが，産業ソフトウェアにおける実践的なリファクタリング支援に効果があるということも検証していきたい．

また，コードクローン自動分類ツールは，定義した全ての分類木について自動的に分類するまでには至っていないため，全自動化を目指したい．また，クローン分類に対応したそれぞれの集約プロセスをアルゴリズムとして利用することで，集約作業自動支援ツールの実現を行いたいと考えている．

謝辞

本研究の全過程を通して，常に適切な御指導及び御助言を賜りまし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克郎教授に心より深く感謝いたします.

本研究において，随時適切な御指導及び御助言を賜りまし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松下誠准教授に深く感謝いたします.

本研究において，逐次適切な御指導及び御助言を頂きまし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石尾隆助教に深く感謝いたします.

本研究を通して，終始適切な御指導及び御助言を頂き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設計学講座吉田則裕助教に心より深く感謝いたします.

本研究において，適切な御指導及び御助言を頂きまし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眞鍋雄貴特任助教に深く感謝いたします.

本研究において，有益な御指導及び御助言を頂きまし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伊達浩典氏に深く感謝いたします.

最後に，その他様々な御指導，御助言等を頂い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深く感謝いたします.

参考文献

[1] Ant. `http://ant.apache.org/`.

[2] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis. Measuring Clone Based Reengineering Opportunities. In *Proc. of METRICS 1999*, pp. 292–303, 1999.

[3] I. D. Baxter, A. Yahin, L. Moura, M. S. Anna, and L. Bier. Clone Detection Using Abstract Syntax Trees. In *Proc. of ICSM 1998*, pp. 368–377, 1998.

[4] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and Evaluation of Clone Detection Tools. *IEEE Trans. Softw. Eng.*, Vol. 33, No. 9, pp. 577–591, 2007.

[5] Eclipse. `http://www.eclipse.org/`.

[6] M. Fowler. `http://refactoring.com/`.

[7] M. Fowler. *Refactoring: Improving the Design of Existing Code.* Addison Wesley, 1999.

[8] Y. Higo, S. Kusumoto, and K. Inoue. A metric-based approach to identifying refactoring opportunities for merging code clones in a java software system. *J. Softw. Maint. Evol.: Res. Pract.*, Vol. 20, No. 6, pp. 435–461, 2008.

[9] Y. Higo, Y. Ueda, S. Kusumoto, and K. Inoue. Simultaneous Modification Support based on Code Clone Analysis. In *Proc. of APSEC 2007*, pp. 262–269, 2007.

[10] T. Kamiya, S. Kusumoto, and K. 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.

[11] C. Kapser and M. W. Godfrey. “ Cloning Considered Harmful ”Considered Harmful. In *Proc. of WCRE 2006*, pp. 19–28, 2006.

[12] J. Kerievsky. *Refactoring to Patterns.* Addison Wesley, 2004.

[13] M. Kim, L. Bergman, T. Lan, and D. Notkin. An Ethnographic Study of Copy and Paste Programming Practices in OOPL. In *Proc. of ISESE 2004*, pp. 83–92, 2004.

[14] B. Lague, D. Proulx, J. Mayrand, E. M. Merlo, and J. Hudepohl. Assessing the Benefits of Incorporating Function Clone Detection in a Development Process. In *Proc. of ICSM 1997*, pp. 314–321, 1997.

[15] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code. *IEEE Trans. Softw. Eng.*, Vol. 32, No. 3, pp. 176–192, 2006.

[16] T. H. Ng, S. C. Cheung, W. K. Chan, and Y. T. Yu. Work Experience versus Refactoring to Design Patterns: A Controlled Experiment. In *Proc. of the FSE 2006*, pp. 12–22, 2006.

[17] W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[18] C. K. Roy and J. R. Cordy. A Mutation/Injection-based Automatic Framework for Evaluating Code Clone Detection Tools. In *Proc. of ICSTW 2009*, pp. 157–166, 2009.

[19] Y. Ueda, T. Kamiya, S. Kusumoto, and K. Inoue. On Detection of Gapped Code Clones Using Gap Locations. In *Proc. of APSEC 2002*, pp. 327–336, 2002.

[20] A. Zeller. *Why Programs Fail*. Morgan Kaufmann Pub., 2005.

[21] 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.

[22] 丸山勝久. 基本ブロックスライシングを用いたメソッド抽出リファクタリング. 情報処理学会論文誌, Vol. 43, No. 6, pp. 1625–1637, 2002.

[23] 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. 1431–1442, 2007.

[24] 中江大海, 神谷年洋, 門田暁人, 加藤裕史, 佐藤慎一, 井上克郎. レガシーソフトウェアを対象とするクローンコードの定量的分析. 電子情報通信学会技術研究報告, Vol. 100, No. 570, pp. 57–64, 2001.

[25] 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 D-I, Vol. J87-D-I, No. 12, pp. 1060–1068, 2004.

付録

A　提案する集約パターンの具体例

プロセス 1. **差異の調査（クローン断片の表現の統一）**

1. **クローン部とクローンペア間の差異を特定する．**
   この例では，図 18 に示すコードクローンを対象とする．265 行目から 271 行目までのコード片 A と，290 行目から 296 行目までのコード片 B とがクローンペアの関係になっている．
   また，図 19 のように，コード片 A の差異は 265 行目の sameDefinition( ) のメソッド呼び出し部であり，コード片 B の差異は 290 行目の similarDefinition( ) のメソッド呼び出し部である．
2. **差異であるメソッド呼び出しの戻り値の型が，クローンペア間で一致していることを確認する．**
   この例では，どちらのメソッド呼び出しの戻り値も boolean で一致している．
   （コード片 A）
   sameDefinition( ) の戻り値 → boolean

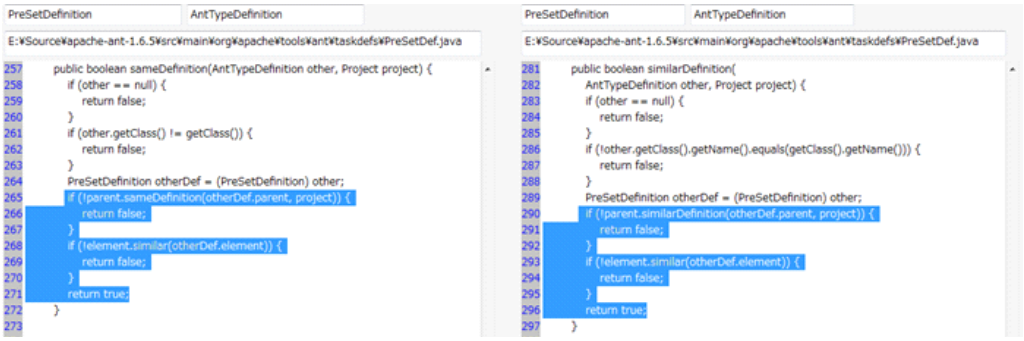


図 18: クローン部の調査

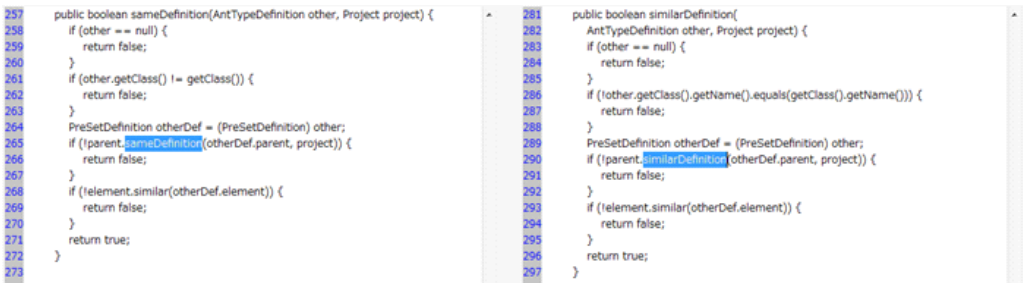


図 19: 差異の調査

（コード片 B）

similarDefinition( ) の戻り値 → boolean

プロセス 2. **差異の除去**

差異の型が一致しているため，対応する（集約プロセス 2-1）を行う．

1. **差異であるクローンペア間で共通するメソッド呼び出しの戻り値の型を持つ新しい変数をクローン部の前に宣言する．**

   この例では，boolean 型のローカル変数を宣言する（図 20）．

   （コード片 A）（コード片 B）

   ・boolean isTargetDefinition;

2. **（集約プロセス 1-3）で宣言した新しい変数に，差異であるメソッド呼び出しの戻り値を代入する．**

   この例では先ほど定義したローカル変数に，差異を持つメソッド呼び出しの戻り値をそれぞれ代入する（図 21）．

   （コード片 A）

   boolean isTargetDefinition = parent.sameDefinition(otherDef.parent, project);

   （コード片 B）

   boolean isTargetDefinition = parent.similarDefinition(otherDef.parent, project);

3. **クローン断片内の差異を持つメソッド呼び出し箇所を，(集約プロセス 1-3) で宣言した新しい変数と置き換える．**

   この例では，メソッド呼び出しを置き換えている（図 22）．

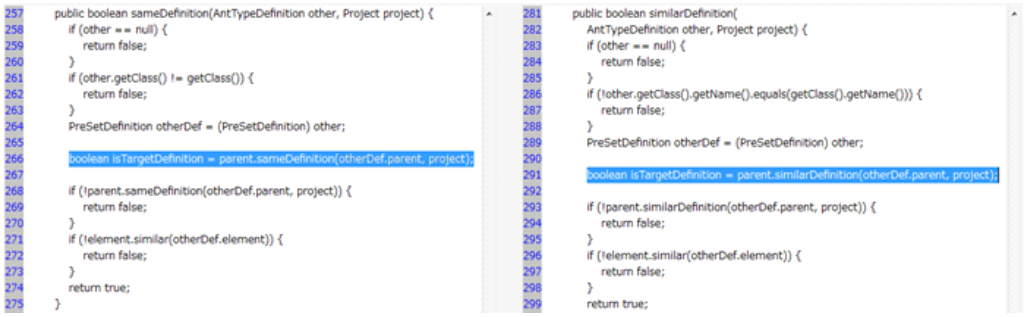
   （コード片 A）

   ・If(！ parent.sameDefinition(otherDef.parent, project)) → If(！ isTarget-

```
257     public boolean sameDefinition(AntTypeDefinition other, Project project) {
258         if (other == null) {
259             return false;
260         }
261         if (other.getClass() != getClass()) {
262             return false;
263         }
264         PreSetDefinition otherDef = (PreSetDefinition) other;
265
266         boolean isTargetDefinition;
267
268         if (!parent.sameDefinition(otherDef.parent, project)) {
269             return false;
270         }
271         if (!element.similar(otherDef.element)) {
272             return false;
273         }
274         return true;
275     }
276
277 parent.sameDefinition( )の戻り値 → boolean
278
```

```
281     public boolean similarDefinition(
282         AntTypeDefinition other, Project project) {
283         if (other == null) {
284             return false;
285         }
286         if (!other.getClass().getName().equals(getClass().getName())) {
287             return false;
288         }
289         PreSetDefinition otherDef = (PreSetDefinition) other;
290
291         boolean isTargetDefinition;
292
293         if (!parent.similarDefinition(otherDef.parent, project)) {
294             return false;
295         }
296         if (!element.similar(otherDef.element)) {
297             return false;
298         }
299         return true;
300     }
301
302 parent.similarDefinition( )の戻り値 → boolean
```

図 20: ローカル変数の宣言



```
257     public boolean sameDefinition(AntTypeDefinition other, Project project) {
258         if (other == null) {
259             return false;
260         }
261         if (other.getClass() != getClass()) {
262             return false;
263         }
264         PreSetDefinition otherDef = (PreSetDefinition) other;
265
266         boolean isTargetDefinition = parent.sameDefinition(otherDef.parent, project);
267
268         if (!parent.sameDefinition(otherDef.parent, project)) {
269             return false;
270         }
271         if (!element.similar(otherDef.element)) {
272             return false;
273         }
274         return true;
275     }
```

```
281     public boolean similarDefinition(
282         AntTypeDefinition other, Project project) {
283         if (other == null) {
284             return false;
285         }
286         if (!other.getClass().getName().equals(getClass().getName())) {
287             return false;
288         }
289         PreSetDefinition otherDef = (PreSetDefinition) other;
290
291         boolean isTargetDefinition = parent.similarDefinition(otherDef.parent, project);
292
293         if (!parent.similarDefinition(otherDef.parent, project)) {
294             return false;
295         }
296         if (!element.similar(otherDef.element)) {
297             return false;
298         }
299         return true;
```

図 21: ローカル変数への差異の代入

Definition)

（コード片 B)

・If(！parent.similarDefinition(otherDef.parent, project)) → If(！isTarget-Definition)

プロセス 3. **メソッドの作成**

1. **同じクラス内に新たなメソッドを作成する．**
   - 修飾子は private とする．
   - 戻り値の型は void とする．
   - 名前はクローン断片の共通する作業の意図に合わせて命名する．

   この例では，図 23 のように作成する．

   ・private void sameDefinitionElement( )

2. **抽出部のコピー**

   元のクローン部のコード片を，(集約プロセス 2）で作成した新たなメソッド

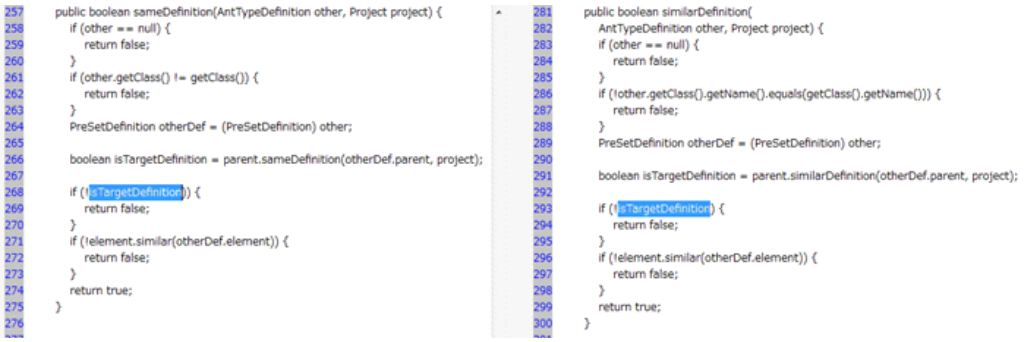


図 22: 差異をローカル変数で置き換え

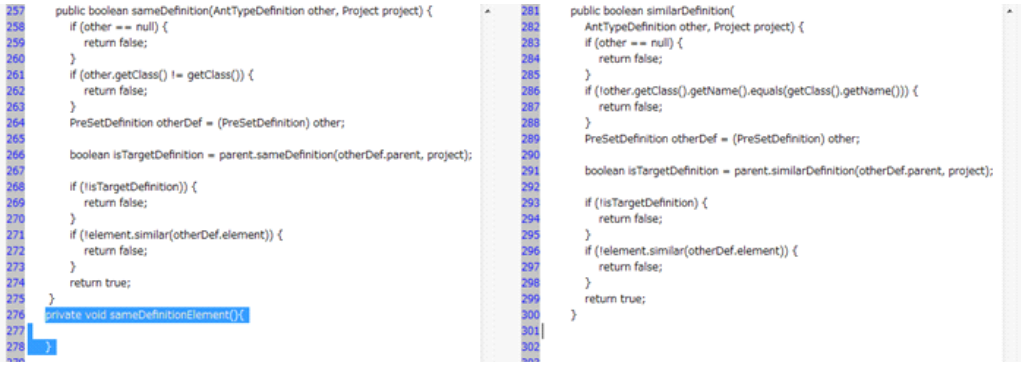


図 23: 新たなメソッドの作成

の本体にコピーする．
通常では，コード片をコピーアンドペーストを行うのみである．しかしこの例では，抽出するコードが return を含んでいるので，新たな抽出先メソッドの戻り値の型を boolean に変更している．また，条件式に当てはまらない場合は True を返すように，新たなメソッドの末尾に return 文で true を返すように記述している（図 24）．

プロセス 4. **クローン部によって参照される元のメソッドの一時変数の特定**
抽出されるコードのうち，元のメソッドにおいてスコープが局所的な（ローカルスコープ）変数への参照を調査する．この例では，以下の 2 つの変数への参照が存在する．

- isTargetDifinition (差異を置き換えるために新たに定義した変数)
- otherDef （元のメソッドのローカル変数）

プロセス 5. **特定した一時変数の利用の調査（抽出先メソッドの引数の決定）**

1. **（集約プロセス 4）で発見された元のメソッドの一時変数が，抽出されるコード内だけで使われていることをそれぞれについて調査する．**
・isTargetDifinition （差異を置き換える新たに定義した変数）
→ 抽出されるコード内だけで使われている
・otherDef （元のメソッドのローカル変数）
→ 抽出されるコード部以外で使用されている

```
257     public boolean sameDefinition(AntTypeDefinition other, Project project) {
258         if (other == null) {
259             return false;
260         }
261         if (other.getClass() != getClass()) {
262             return false;
263         }
264         PreSetDefinition otherDef = (PreSetDefinition) other;
265
266         boolean isTargetDefinition = parent.sameDefinition(otherDef.parent, project);
267
268         if (!isTargetDefinition)) {
269             return false;
270         }
271         if (!element.similar(otherDef.element)) {
272             return false;
273         }
274         return true;
275     }
276     private boolean sameDefinitionElement(){
277         if (!isTargetDefinition)) {
278             return false;
279         }
280         if (!element.similar(otherDef.element)) {
281             return false;
282         }
283         return true;
284     }
```

```
281     public boolean similarDefinition(
282         AntTypeDefinition other, Project project) {
283         if (other == null) {
284             return false;
285         }
286         if (!other.getClass().getName().equals(getClass().getName())) {
287             return false;
288         }
289         PreSetDefinition otherDef = (PreSetDefinition) other;
290
291         boolean isTargetDefinition = parent.similarDefinition(otherDef.parent, project);
292
293         if (!isTargetDefinition) {
294             return false;
295         }
296         if (!element.similar(otherDef.element)) {
297             return false;
298         }
299         return true;
300     }
301
302
303
304
305
306
307
308
```

図 24: クローン部のコピー

2. それぞれの参照について，一時変数の参照される範囲について場合分けされた作業を行う．

   各一時変数がコード内だけで使われている場合の処理を行う．isTargetDifinition がメソッド参照の差異を置き換えた変数であるので，対応した処理を行う．
   ・isTargetDifinition （差異を置き換えるために新たに定義した変数）
   → 抽出されるコード内だけで使われている
   → **（集約プロセス 7-1）にて抽出先メソッドの引数として扱う．**

   otherDef がクローン部外で使われている変数であるので，対応した処理を行う．
   ・otherDef （元のメソッドのローカル変数）
   → 抽出されるコード部以外で使用されている
   → **（集約プロセス 7-1）にて抽出先メソッドの引数として扱う．**

プロセス 6. **特定した一時変数の変更の調査（抽出先メソッドの戻り値の決定）**

1. **（集約プロセス 4）で発見された元のメソッドの一時変数が，抽出されるコード内で変更されるかどうか，それぞれについて調べる．**
   ・isTargetDifinition
   →抽出されるコード内で変更されない
   ・otherDef
   →抽出されるコード内で変更されない

2. **変更を加えずに次のプロセスに進む**
   全ての一時変数が抽出されるコード内で変更されないため，特に変更を加えずに次のプロセスに進む．

プロセス 7. **変数の扱い**

1. **（集約プロセス 5-2）で特定した変数を，抽出先メソッドの引数として渡す．**
   この例では
   ・isTargetDefinition
   ・otherDef が,（集約プロセス 5-2）の調査で引数で渡すべき変数であると確認できている．

よって，図 25 のように書き換える．

・private boolean sameDefinitionElement(boolean isTargetDefinition, PreSetDefinition otherDef)

プロセス 8. **コンパイル**

コンパイルする．

プロセス 9. **抽出先メソッドの呼び出し各クローン断片元のメソッドにおいて，抽出される各クローン断片を抽出先メソッドへのコールに置換する．**

この例では元のメソッドの戻り値を返す必要があるため，図 26 のように書き換える．

・retrun sameDefinitionElement(isTargetDefinition, otherDef)

プロセス 10. **コンパイル・テスト**

コンパイルしてテストする．

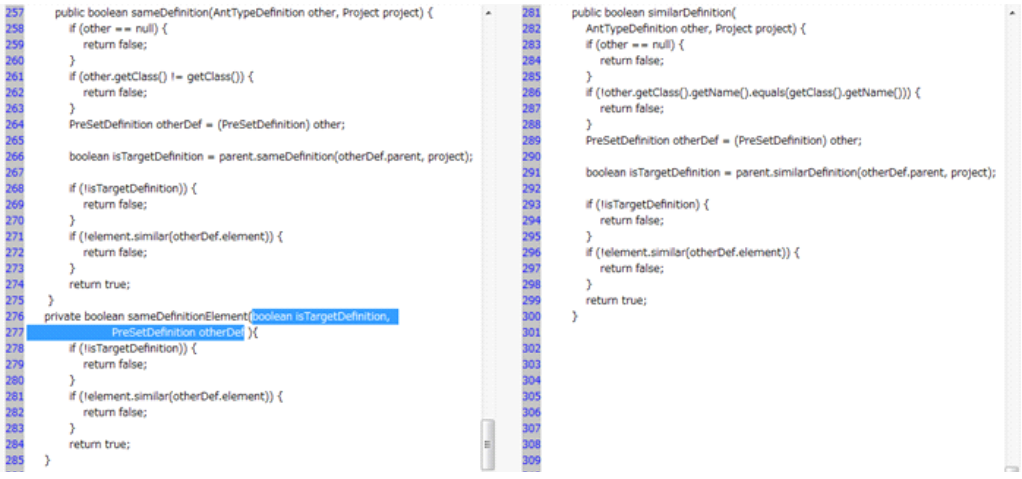
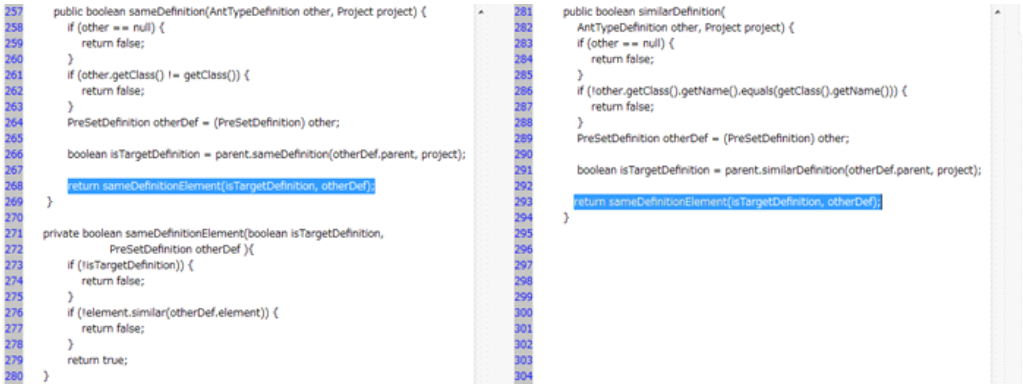


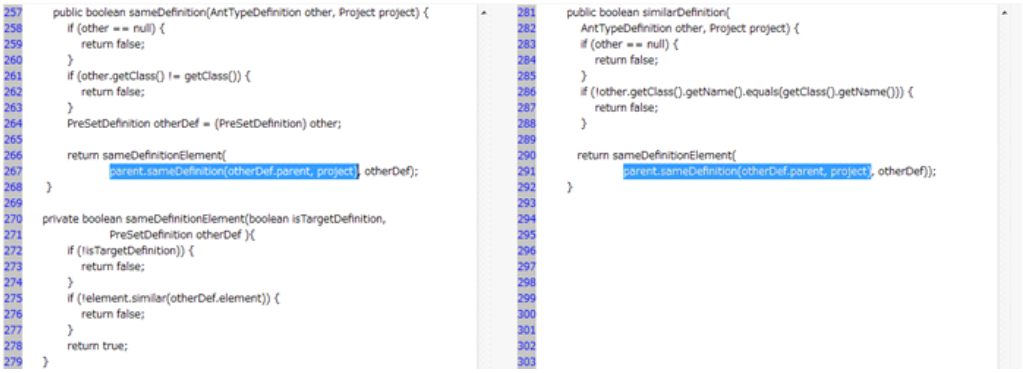
図 25: メソッドの引数の設定



```
257     public boolean sameDefinition(AntTypeDefinition other, Project project) {
258         if (other == null) {
259             return false;
260         }
261         if (other.getClass() != getClass()) {
262             return false;
263         }
264         PreSetDefinition otherDef = (PreSetDefinition) other;
265
266         boolean isTargetDefinition = parent.sameDefinition(otherDef.parent, project);
267
268         return sameDefinitionElement(isTargetDefinition, otherDef);
269     }
270
271     private boolean sameDefinitionElement(boolean isTargetDefinition,
272                 PreSetDefinition otherDef ){
273         if (!isTargetDefinition)) {
274             return false;
275         }
276         if (!element.similar(otherDef.element)) {
277             return false;
278         }
279         return true;
280     }
```

```
281     public boolean similarDefinition(
282         AntTypeDefinition other, Project project) {
283         if (other == null) {
284             return false;
285         }
286         if (!other.getClass().getName().equals(getClass().getName())) {
287             return false;
288         }
289         PreSetDefinition otherDef = (PreSetDefinition) other;
290
291         boolean isTargetDefinition = parent.similarDefinition(otherDef.parent, project);
292
293         return sameDefinitionElement(isTargetDefinition, otherDef);
294     }
295
296
297
298
299
300
301
302
303
304
```

図 26: メソッドの呼び出し



```
257     public boolean sameDefinition(AntTypeDefinition other, Project project) {
258         if (other == null) {
259             return false;
260         }
261         if (other.getClass() != getClass()) {
262             return false;
263         }
264         PreSetDefinition otherDef = (PreSetDefinition) other;
265
266         return sameDefinitionElement(
267                 parent.sameDefinition(otherDef.parent, project), otherDef);
268     }
269
270     private boolean sameDefinitionElement(boolean isTargetDefinition,
271                 PreSetDefinition otherDef ){
272         if (!isTargetDefinition)) {
273             return false;
274         }
275         if (!element.similar(otherDef.element)) {
276             return false;
277         }
278         return true;
279     }
```

```
281     public boolean similarDefinition(
282         AntTypeDefinition other, Project project) {
283         if (other == null) {
284             return false;
285         }
286         if (!other.getClass().getName().equals(getClass().getName())) {
287             return false;
288         }
289
290         return sameDefinitionElement(
291                 parent.sameDefinition(otherDef.parent, project), otherDef));
292     }
293
294
295
296
297
298
299
300
301
302
303
```

図 27: ローカル変数の除去

B 予備アンケート内容

1. Java プログラミングの経験

   (a) 言語を問わないプログラミング経験年数

   (b) これまでの Java プログラミング経験年数

   (c) 自身の Java 言語で作成したプログラムの最大コード行総数

   (d) 他人が開発したプログラムの機能追加，保守経験がある

2. コードクローンの知識

   コードクローンに関して

   (a) コードクローンを扱った経験がない

   (b) 自分が作成したプログラムのコードクローンを検出した経験がある

   (c) コードクローンに関する研究をした経験がある

   (d) その他

3. リファクタリング作業経験

   既読のリファクタリングに関する書籍に関して

   (a) 何も読んだことがない

   (b) 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

   (c) パターン指向リファクタリング入門

   (d) その他リファクタリング関連書籍

   実践的なリファクタリング作業経験に関して

   (a) リファクタリングの経験がない

   (b) 自分のプログラムをリファクタリングした経験がある

   (c) リファクタリングに関する研究をした経験がある

   (d) 業務上のプログラムをリファクタリングした経験がある

C アンケート内容

1. 二つのパターンの比較

   (a) どちらのパターンを利用した方が，正確な集約作業の実現に繋がるという観点で，より有効であると考えられるか．

   (b) どちらのパターンを利用した方が，集約作業を学習する上での学習教材という観点で，より有効であると考えられるか．

2. 有効な集約プロセス

   提案するパターンに含まれ，Fowler のパターンには存在しない記述の中で，集約作業の手助けになった記述を挙げてください．

3. コメント

   (a) 提案するパターンの集約プロセスの記述の詳細度に対してコメントがあれば記述してください．

   (b) 提案するパターンの集約プロセスの不明確な記述に対してコメントがあれば記述してください．

   (c) 提案するパターンの集約プロセスの不足する記述に対してコメントがあれば記述してください．

   (d) 提案するパターンの集約プロセスに対してその他コメントがあれば記述してください．

D 集約作業時間データ

本実験で計測した各被験者のグループ別の，作業時間に関するデータを図 28，図 29，図 30，図 31 に記載する．それぞれ，クローンペア 1 とクローンペア 2 に対する集約作業時間をプロセスごとに表示している．各表には，各グループの被験者 3 名の作業時間の表，および各グループの平均値を集計した作業時間の表を掲載している．全ての表は秒単位で表されている．表中の Mine とは，本稿で提案する集約パターンを用いて集約作業を行ったことを表し，Fowler は Fowler のパターンを用いた場合を表す．

グループ平均表

被験者No.	1～3
被験者Group	A

	時間
クローンペア1	1071
クローンペア2	749

対象クローンペア	1
利用パターン	Mine

プロセス番号	計測時間
1	125
2	112
3	149
4	66
5	131
6	155
7	124
8	12
9	124
10	73
計	1071

対象クローンペア	2
利用パターン	Mine

プロセス番号	計測時間
1	68
2	180
3	76
4	46
5	30
6	41
7	51
8	15
9	66
10	177
計	749

被験者別比較表

被験者No.	1
被験者Group	A

	時間
クローンペア1	732
クローンペア2	481

対象クローンペア	1
利用パターン	Mine

プロセス番号	計測時間
1	58
2	113
3	114
4	94
5	104
6	98
7	40
8	14
9	86
10	11
計	732

対象クローンペア	2
利用パターン	Mine

プロセス番号	計測時間
1	61
2	146
3	51
4	64
5	11
6	32
7	59
8	11
9	35
10	11
計	481

被験者別比較表

被験者No.	2
被験者Group	A

	時間
クローンペア1	1792
クローンペア2	1269

対象クローンペア	1
利用パターン	Mine

プロセス番号	計測時間
1	280
2	111
3	274
4	32
5	241
6	275
7	203
8	10
9	236
10	130
計	1792

対象クローンペア	2
利用パターン	Mine

プロセス番号	計測時間
1	97
2	208
3	133
4	43
5	56
6	68
7	43
8	18
9	114
10	489
計	1269

被験者別比較表

被験者No.	3
被験者Group	A

	時間
クローンペア1	688
クローンペア2	496

対象クローンペア	1
利用パターン	Mine

プロセス番号	計測時間
1	37
2	112
3	58
4	72
5	49
6	92
7	128
8	12
9	50
10	78
計	688

対象クローンペア	2
利用パターン	Mine

プロセス番号	計測時間
1	46
2	185
3	43
4	30
5	22
6	22
7	52
8	16
9	48
10	32
計	496

図 28: グループ A 集約作業時間計測データ

グループ平均表

被験者No.	4～6
被験者Group	B

	時間
クローンペア1	1308
クローンペア2	880

対象クローンペア	1
利用パターン	Mine

プロセス番号	計測時間
1	167
2	265
3	88
4	60
5	229
6	56
7	216
8	18
9	137
10	71
計	1308

対象クローンペア	2
利用パターン	Fowler

プロセス番号	計測時間
1	101
2	261
3	80
4	62
5	39
6	42
7	105
8	19
9	68
10	103
計	880

被験者別比較表

被験者No.	4
被験者Group	B

	時間
クローンペア1	936
クローンペア2	487

対象クローンペア	1
利用パターン	Mine

プロセス番号	計測時間
1	193
2	57
3	51
4	42
5	199
6	24
7	161
8	18
9	111
10	80
計	936

対象クローンペア	2
利用パターン	Fowler

プロセス番号	計測時間
1	90
2	114
3	58
4	21
5	5
6	10
7	27
8	10
9	39
10	113
計	487

被験者別比較表

被験者No.	5
被験者Group	B

	時間
クローンペア1	1916
クローンペア2	1200

対象クローンペア	1
利用パターン	Mine

プロセス番号	計測時間
1	167
2	358
3	129
4	101
5	344
6	103
7	423
8	25
9	230
10	36
計	1916

対象クローンペア	2
利用パターン	Fowler

プロセス番号	計測時間
1	88
2	250
3	128
4	135
5	92
6	104
7	84
8	24
9	116
10	179
計	1200

被験者別比較表

被験者No.	6
被験者Group	B

	時間
クローンペア1	1071
クローンペア2	952

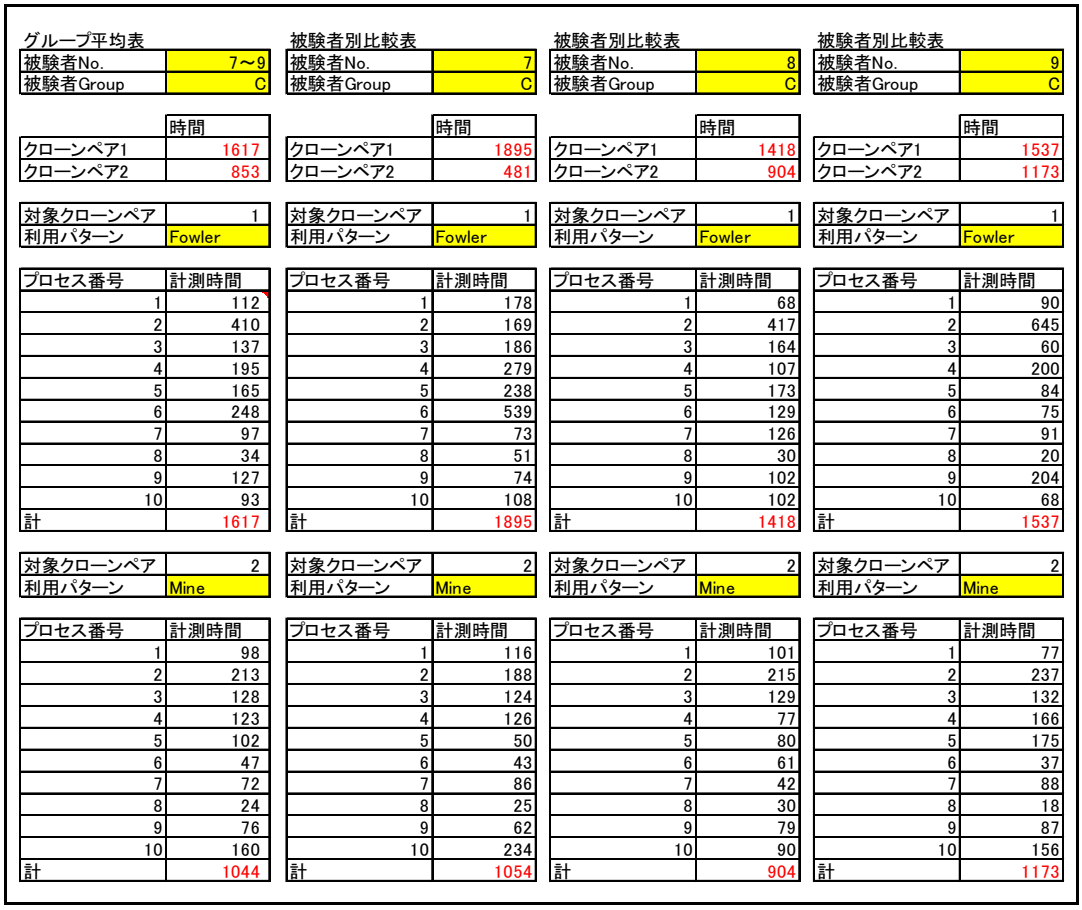
対象クローンペア	1
利用パターン	Mine

プロセス番号	計測時間
1	140
2	379
3	85
4	38
5	145
6	42
7	65
8	10
9	69
10	98
計	1071

対象クローンペア	2
利用パターン	Fowler

プロセス番号	計測時間
1	125
2	419
3	53
4	30
5	20
6	12
7	205
8	22
9	49
10	17
計	952

図 29: グループ B 集約作業時間計測データ



グループ平均表

被験者No.	7～9
被験者Group	C

	時間
クローンペア1	1617
クローンペア2	853

対象クローンペア	1
利用パターン	Fowler

プロセス番号	計測時間
1	112
2	410
3	137
4	195
5	165
6	248
7	97
8	34
9	127
10	93
計	1617

対象クローンペア	2
利用パターン	Mine

プロセス番号	計測時間
1	98
2	213
3	128
4	123
5	102
6	47
7	72
8	24
9	76
10	160
計	1044

被験者別比較表

被験者No.	7
被験者Group	C

	時間
クローンペア1	1895
クローンペア2	481

対象クローンペア	1
利用パターン	Fowler

プロセス番号	計測時間
1	178
2	169
3	186
4	279
5	238
6	539
7	73
8	51
9	74
10	108
計	1895

対象クローンペア	2
利用パターン	Mine

プロセス番号	計測時間
1	116
2	188
3	124
4	126
5	50
6	43
7	86
8	25
9	62
10	234
計	1054

被験者別比較表

被験者No.	8
被験者Group	C

	時間
クローンペア1	1418
クローンペア2	904

対象クローンペア	1
利用パターン	Fowler

プロセス番号	計測時間
1	68
2	417
3	164
4	107
5	173
6	129
7	126
8	30
9	102
10	102
計	1418

対象クローンペア	2
利用パターン	Mine

プロセス番号	計測時間
1	101
2	215
3	129
4	77
5	80
6	61
7	42
8	30
9	79
10	90
計	904

被験者別比較表

被験者No.	9
被験者Group	C

	時間
クローンペア1	1537
クローンペア2	1173

対象クローンペア	1
利用パターン	Fowler

プロセス番号	計測時間
1	90
2	645
3	60
4	200
5	84
6	75
7	91
8	20
9	204
10	68
計	1537

対象クローンペア	2
利用パターン	Mine

プロセス番号	計測時間
1	77
2	237
3	132
4	166
5	175
6	37
7	88
8	18
9	87
10	156
計	1173

図 30: グループ C 集約作業時間計測データ

グループ平均表

被験者No.	10～12
被験者Group	D

	時間
クローンペア1	1680
クローンペア2	1363

対象クローンペア	1
利用パターン	Fowler

プロセス番号	計測時間
1	232
2	216
3	148
4	112
5	329
6	145
7	96
8	157
9	71
10	174
計	1680

対象ソースコード	2
利用パターン	Fowler

プロセス番号	計測時間
1	76
2	340
3	73
4	177
5	45
6	23
7	174
8	124
9	103
10	227
計	1363

被験者別比較表

被験者No.	10
被験者Group	D

	時間
クローンペア1	1482
クローンペア2	1233

対象クローンペア	1
利用パターン	Fowler

プロセス番号	計測時間
1	185
2	170
3	112
4	70
5	498
6	77
7	145
8	44
9	10
10	171
計	1482

対象ソースコード	2
利用パターン	Fowler

プロセス番号	計測時間
1	52
2	164
3	59
4	36
5	69
6	27
7	198
8	48
9	36
10	544
計	1233

被験者別比較表

被験者No.	11
被験者Group	D

	時間
クローンペア1	1543
クローンペア2	1723

対象クローンペア	1
利用パターン	Fowler

プロセス番号	計測時間
1	257
2	293
3	180
4	174
5	115
6	67
7	74
8	183
9	81
10	119
計	1543

対象ソースコード	2
利用パターン	Fowler

プロセス番号	計測時間
1	112
2	743
3	94
4	256
5	10
6	10
7	80
8	211
9	125
10	82
計	1723

被験者別比較表

被験者No.	12
被験者Group	D

	時間
クローンペア1	2016
クローンペア2	1134

対象クローンペア	1
利用パターン	Fowler

プロセス番号	計測時間
1	253
2	186
3	151
4	91
5	375
6	291
7	69
8	244
9	123
10	233
計	2016

対象ソースコード	2
利用パターン	Fowler

プロセス番号	計測時間
1	65
2	113
3	66
4	240
5	56
6	32
7	245
8	114
9	149
10	54
計	1134

図 31: グループ D 集約作業時間計測データ