特別研究報告

題目

再利用実績に基づくコード片検索における プログラム構造を考慮した検索結果の提示

指導教員
楠本 真二 教授

報告者
大谷 明央

平成 26 年 2 月 14 日

大阪大学 基礎工学部 情報科学科

平成 25 年度 特別研究報告

再利用実績に基づくコード片検索における
プログラム構造を考慮した検索結果の提示

大谷 明央

内容梗概

ソースコードの再利用を支援する手法の 1 つとしてコード片検索手法が広く知られている．コード片検索手法とは，ユーザが求める機能を表すクエリを入力として与えると，そのクエリに関連する機能を有するコード片を提示する手法である．特に，再利用実績に基づいたコード片検索手法では，過去に再利用されているコード片のみを提示することで，ユーザにとって必要なコード片のみを提示する．しかし，この手法はプログラム構造を考慮していないため，例えば if 文，while 文などの，複数の文から構成される文の途中までを提示することがある．このため，再利用の後に，コード片に残りの部分を追加する，あるいは文そのものを削除するなどの修正を加える必要が生じる可能性がある．本研究では，再利用実績を用いたコード片検索手法において，提示するコード片にプログラム構造を考慮した調整を施す手法を提案する．提案手法では，ユーザに提示するコード片について，提示する範囲を拡大あるいは縮小することで，プログラム構造を考慮した調整を実現している．これによって，再利用においてコード片に修正を加える手間を削減することができる．本研究では，9 人の被験者の協力のもと，既存手法との比較を行った．実験の結果，提案手法を使うことで既存手法に比べて効率的にソフトウェアの開発を行えることが確認された．

主な用語

コードクローン
コード片検索
ソースコードの再利用

目 次

1　はじめに

ソフトウェア開発において，ソースコードの再利用を行うことは有益である．既存のソースコードを適切に再利用することで，開発者が新しい機能を実装する必要がなくなり，開発効率が向上する．また，十分にテストされているソースコードを再利用すれば，信頼性の高い機能を簡単に利用することができる．このように，ソースコードの再利用には様々なメリットがあり，現在までにソースコードの再利用を支援する研究が多く行われている．

ソースコードの再利用を支援するシステムの１つにコード片検索システムがある [1][2][3]．コード片検索システムは，ユーザが求める機能を表現したクエリを入力すると，そのクエリに関連する機能が実装されているソースコードを出力する．また，コード片検索システムは独自の指標をもっており，その指標に基づいて出力するソースコードの順番を決定する．コード片検索システムの利点の１つはユーザに対して複雑な操作を要求しない点である．ユーザが行う操作はシステムに対してクエリを入力するだけである．

しかし，既存のコード片検索システムには問題点も存在する．既存システムはユーザが必要としていない機能を含むソースコードを提示することがある．この問題の原因は，既存のコード片検索システムがプログラム構造のみを基にソースコードを提示することにある．既存の多くのコード片検索システムはファイルやクラス単位でソースコードを提示するため，特にユーザが単一の機能や規模の小さい機能の再利用を行う際にこの問題が発生しやすい．ユーザは常に同じ規模の機能を求めるわけではない．そのため，コード片検索システムはユーザの要求に合った規模や抽象度のソースコードを提示する必要がある．この問題を解決する手法として，再利用実績に基づいたコード片検索手法が提案されている [4]．この手法を用いたコード検索システムは過去に再利用されたコード片を検出し，その中からユーザの入力するクエリと関連する機能を持つコード片を提示する．過去に再利用されたコード片は今後再び再利用される可能性が高く，このようなコード片を提示することで効率的な再利用支援が行える．

しかし，この手法には，コード片を提示する際に過去に行われた再利用のみを考慮し，プログラム構造については考慮しないという問題がある．過去に行われた再利用のみを考慮することはソースコードの再利用の観点において，必ずしも有益ではない．その理由は，提示されたコード片がコピーアンドペーストによって再利用された場合，コード片を貼り付けたプログラムにおいて構文エラーが発生する可能性があり，ユーザが構文エラーを修正する作業が必要となるためである．このことはソースコードの効率的な再利用を妨げている．

そこで本研究では，より効率的な再利用の支援を行うために，再利用実績を持つコード片にプログラムの構造を考慮した調整を加えて提示する方法を提案する．つまり，提案手法はプログラムの構造と再利用実績の両方を考慮した提示を行う．プログラムの構造を考慮した

調整を加えることで，提示した文がコピーアンドペーストされた際に構文エラーを修正するための作業が必要となることを防ぐことができる．また，9 人の被験者の協力のもと実験を行った．実験では，与えられたタスクを被験者が完成させるまでの経過を記録した．被験者はタスクを実装する際に提案手法を含む 3 つのツールを用いている．実験で得た記録を基に，ツールによって提示したコード片と実装したコード片を比較した．その結果，提案手法は既存手法と比べ効率的な再利用の支援を行えることが確認された．

以降，2 章ではコードクローンの定義と，コード片検索システムについて述べる．3 章では研究動機について述べる．4 章では提案手法について説明し，その実装について述べる．5 章では，被験者を募って行った実験について述べる．6 章では，実験結果の考察について述べる．7 章では，実験の妥当性について述べる 8 章で関連研究について述べる．最後に 9 章で本研究のまとめと今後の課題について述べる．

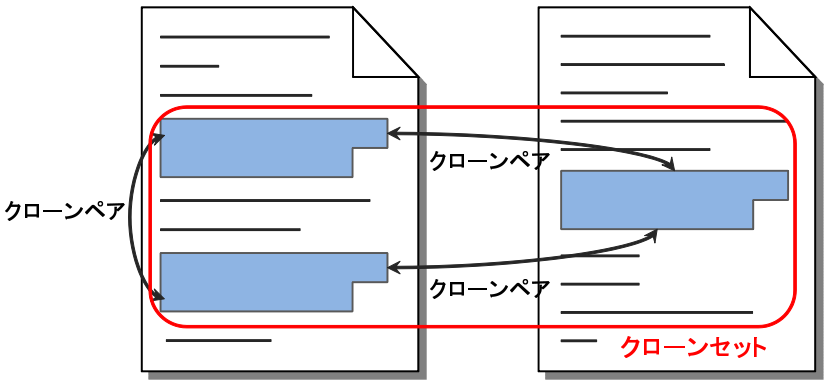


図 1: クローンペアとクローンセット

2 準備

2.1 コードクローン

2.1.1 定義

コードクローンとはソースコード中に存在する同一，あるいは類似するコード片のことである．図 1 に示すように，ソースコード中に存在する 2 つのコード片 α，β が類似しているとき，α と β は互いにクローンであるという．またペア（α，β）をクローンペアと呼ぶ．α，β それぞれを真に包含する如何なるコード片も類似していないとき，α，β を極大クローンと呼ぶ．また，互いにクローンであるコード片を同値としたときの同値類をクローンセットと呼ぶ [5]．ただし，どのような基準で類似していると判断するかは検出手法や検出ツールによって異なる．

また，コードクローン間の類似の度合に基づきコードクローンを次の 3 つのタイプに分類することができる [6][7]．

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に依存する箇所を除いて，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

Type-2 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 発生の原因

コードクローンがソフトウェアの中に作りこまれる，もしくは発生する原因として次のようなものが挙げられる [8][9][10]．

既存コードのコピーアンドペーストによる再利用

近年のソフトウェア設計手法を利用することにより構造化や再利用可能な設計が可能である．しかし，コードの再利用が容易になったために，現実にはコピーアンドペーストによる場当たり的な既存コードの再利用が多く行われるようになった．コピーアンドペーストによって生成されたコード片は，コピー元のコード片とコードクローン関係になる．

コーディングスタイル

規則的に必要なコードはスタイルとして同じように記述される場合がある．例えば，ユーザインターフェース処理を記述するコードなどである．

定型処理

定義上簡単で頻繁に用いられる処理．例えば，所得税の計算や，キューの挿入処理，データ構造アクセス処理などである．

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プログラミング言語を開発に用いている場合，同じようなアルゴリズムを用い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よって生成されるコードは，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．このため，類似した処理を目的としたコードを生成した場合，識別子名等の違いを除き，類似したコードが生成される．

複数のプラットフォームに対応したコード

複数の OS や CPU に対応したソフトウェアは，各プラットフォームを対象に生成されたコード部分に重複した処理が存在する傾向が強い．

偶然

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

2.2 コードクローン検出手法

コードクローンを検出する手法はこれまでに多数提案されている．またそれらを実装した，コードクローンを自動的に検出するツールも多数開発されている．これらの検出技術はその検出単位によって，大まかに以下の 5 つに分類することができる [11][12]．

行単位の検出

行単位の検出は，ソースコードを行単位で比較してコードクローンを検出する手法であり，閾値以上連続して一致する行をコードクローンとして検出する．他の手法と異なり，ソースコードに対する事前処理を必要としない．このため，他の手法と比べて高速にコードクローンを検出可能である．しかし，同じ処理を行っているコードであっても，例えば長い行を複数行に分割した場合と分割しなかった場合など，コーディングスタイルが違う場合はコードクローンとして検出できないという弱点を持つ．

字句単位の検出

字句単位の検出は，ソースコードを字句単位に分割し，閾値以上連続して一致する字句の部分列をコードクローンとして検出する手法である．行単位の検出と異なり，コーディングスタイルのみ違う場合などもコードクローンとして検出することが可能である．ソースコードを検出用の中間表現に変換する必要がないため，高速にコードクローン検出を行うことが

```
int num = k + size * 4;
```

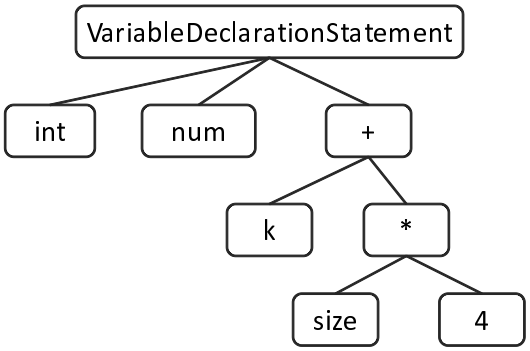


図 2: 抽象構文木の例

できるという利点もある．また，字句に事前処理を行うことで変数名などのユーザ定義名のみ異なるコードクローンなども検出可能となる．

抽象構文木を用いた検出

抽象構文木 (図 2) を用いた検出は，ソースコードに対して構文解析を行い，抽象構文木を構築した後，その抽象構文木を用いてコードクローンを検出する手法であり，抽象構文木上の同形の部分木がコードクローンとして検出される．抽象構文木を構築するという事前処理を要するため，行単位の検出や字句単位の検出と比べ，時間的，空間的コストが高くなるという欠点がある．ある関数定義の終わりから次の関数定義の先頭までの類似部分など，プログラムの構造を無視したコードクローンを検出しないという特徴を持つ．

プログラム依存グラフを用いた検出

プログラム依存グラフ (図 3，[13]) を用いた検出は，ソースコードに対して意味解析を行い，ソースコードの要素間の依存関係を表すプログラム依存グラフを構築した後，そのプログラム依存グラフを用いてコードクローン検出を行う手法である．プログラム依存グラフ上の同形部分がコードクローンとして検出される．抽象構文木を用いた検出と同様に事前処理を必要とするため，時間的，空間的コストが高くなるという欠点を持つ．ソースコードの順番が入れ替わっていても意味的に同一であるコードクローン (順序入れ替わりコードクローン) などは意味的な処理を考慮しなければ検出できないが，この手法はこれらのコードクローンを検出することができるという点が特徴として挙げられる．

```
1: void sample() {
2:   for ( int i = 0 ; i < 10 ; i++ ) {
3:     System.out.println(i);
4:   }
5: }
```

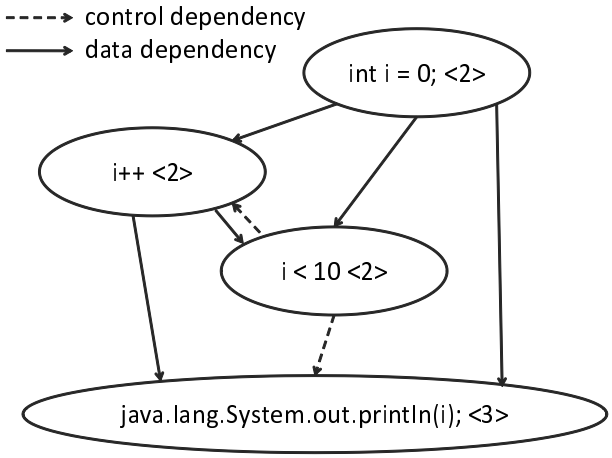
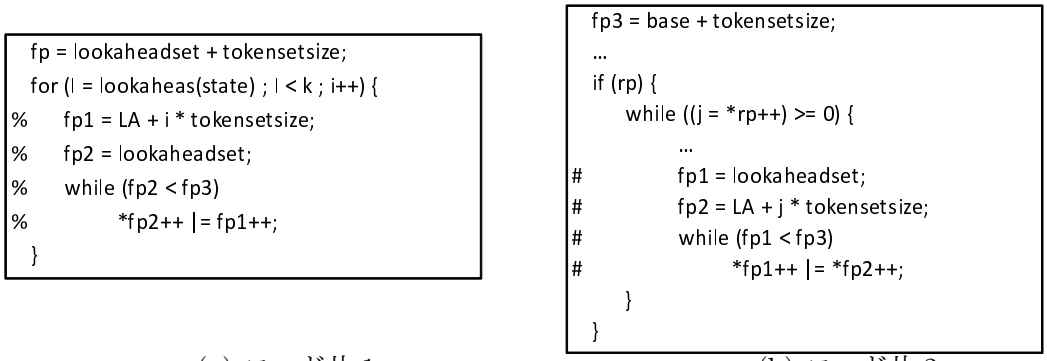


図 3: プログラム依存グラフの例



(a) コード片 1　　(b) コード片 2

図 4: 順序入れ替わりコードクローン

順序入れ替わりコードクローンの例を図 4 に示す．この例の場合，%で表されているコード片と，#で表されているコード片が順序入れ替わりコードクローンとなる．

その他の技術を用いた検出

その他の技術を用いた検出手法として，プログラムのモジュール (ファイル，クラス，メソッドなど) に対してメトリクスを計測し，その値の一致または近似の度合いを検査することによって，そのモジュール単位でのコードクローンを検出する手法であるメトリクスを用いた検出や，プログラムの盗用の検出やプログラムの作者を特定することを目的とした，フィンガープリントやバースマークを用いた検出手法などがある [11]．

2.3　コード片検索

ソースコードの再利用において，機能の実装に適したコード片を既存のソースコードから探さなければならない場合がある．このとき，機能の実装に適したコード片を既存のソースコードから発見するための手段として，コード片検索 [1][2][3] が用いられる．コード片検索は，ユーザが求める機能を表現したクエリを入力すると，そのクエリに関連する機能が実装されているソースコードを出力する．ここで機能とは，1 つあるいは複数の命令によって実現される，ひとまとまりの処理のことを指す．コード片検索では多くの場合，検索結果として複数のソースコードが出力されるので，ユーザはその中から機能の実装に適したものを探し再利用する．コード片検索システムの利点として，ユーザに対して複雑な操作を要求しない点が挙げられる．また，コード片検索は独自の指標により，出力するソースコードの重要度を判定し，ユーザに提示する順番を決定する．既存研究では，関数の呼び出し関係を基に計算する *Component Rank Score(CRS)*[1] や，ユーザの入力したクエリとコード片の関連

性の強さから計算する *Word Occurence Score(WOS)*[3] などが指標として用いられている.

```
…
220:public void form(String id) {
…
238:  StringBuffer buffer
            = new StringBuffer();
239:  buffer.append("<form>");
240:  if (items == null) {
…
244:  } else {
245:    if (items.size() > 0) {
…
261:      buffer.append("</p>");
262:    }
263:  }
…
266:}
…
```

(a) 従来手法

```
…
220:public void form(String id) {
…
238:  StringBuffer buffer
            = new StringBuffer();
239:  buffer.append("<form>");
240:  if (items == null) {
…
244:  } else {
245:    if (items.size() > 0) {
…
261:      buffer.append("</p>");
262:    }
263:  }
…
266:}
…
```

(b) 石原らの手法

図 5: 手法による提示例

3 再利用実績に基づくコード片検索

3.1 概要

石原らは，過去に再利用されたコード片を検出し，その中からユーザの入力するクエリと関連する機能を持つコード片を提示する手法を提案した [4]．図 5 に，従来手法および石原らの手法によって提示されるコード片の範囲をハイライトして示す．従来手法は図 5(a) に示すようにメソッドやクラス等のプログラミング言語単位でコード片を提示するのに対し，石原らの手法では図 5(b) のように過去に行われた再利用を単位としてコード片を提示する．石原らの手法では過去の再利用を検出するためにコードクローン検出を利用している [14]．コードクローンの発生原因の 1 つにコードの再利用があるため，コードクローンを検出することで過去の再利用を特定している．石原らの手法はソースコード解析部とソースコード提示部の 2 つから構成される．ソースコード解析部では，提示対象となるコード片集合を構築するために，対象となるソースコード集合におけるコードクローンの検出と，検出したコー

ドクローンに含まれるキーワードの抽出を行い，これらの情報を基にデータベースを作成している．ソースコード提示部では，ソースコード解析部で作成したデータベースの情報を基に，ユーザが入力したクエリに関連するコード片を探し，いくつかの指標を基にコード片に順位をつけ，順位にしたがってコード片をユーザに提示している．

3.2 実装

コードクローン検出

石原らの手法では，様々な規模や抽象度のコード片をユーザに提示するため，規模の小さいコードクローンを見つける必要がある．しかし，検出対象とするコードクローンの規模を小さくすると，大量のコードクローンが検出されることになる．したがって，スケーラブルなコードクローン検出が求められる．以上の点から，石原らは，文単位かつインデックスを用いたコードクローン検出手法を利用している [14]．この手法では，ソースコードを正規化し，正規化したソースコード中の文を一定数ごとにグループ化する．そして，グループごとに計算したハッシュ値からインデックスを作成し，作成したインデックスをもとにコードクローン検出を行う．

キーワード抽出

石原らの手法では，クエリとのマッチングに使用するためのキーワードを，コードクローンとして検出されたコード片から抽出する．キーワードとして抽出するのは，変数名，メソッド名，クラス名やメソッドやクラスに付随する Javadoc コメントに存在する単語である．抽出したキーワードの情報からデータベースを作成する．ユーザによってクエリが入力されると，データベースに登録したキーワードの情報をもとにクエリとのマッチングを行う．

順位の計算

石原らの手法では，クエリとのマッチングにより得られた各コード片について 3 つの指標を用いて順位を決定する．石原らの手法で用いられる指標を以下に挙げる．

- そのコード片を含むメソッドの重要度
- クエリとそのコード片の関連性の強さ
- 入力として与えられたソースコード集合において，そのコード片が再利用された回数

そのコード片を含むメソッドの重要度は，コンポーネントランク法 [1] により，メソッドの呼び出し関係を基に計算する．ここでは，多くのメソッドに呼ばれているメソッドや，重要

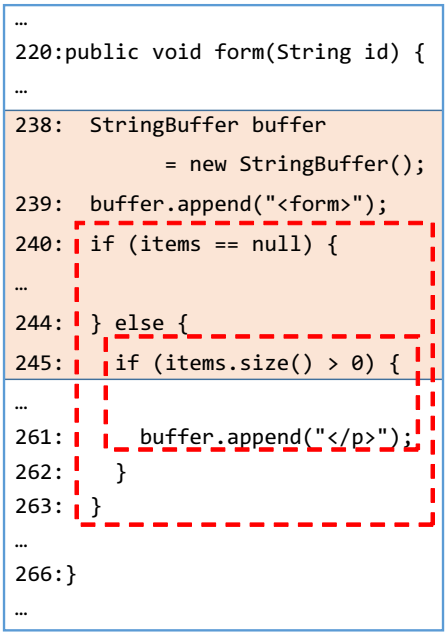


図 6: 有益ではないコード片の例

なメソッドから呼ばれているメソッドが重要であると判断される．クエリとそのコード片の関連性の強さは，TF-IDF 法 [15] により計算する．ここでは，入力されたクエリが，少数のコード片からのみ抽出されるキーワードと一致する場合，関連性が強いと判断される．入力として与えられたソースコード集合において，そのコード片が再利用された回数は，3.2 節において検出されたクローンセットの要素数から計算する．これらの指標それぞれについてコード片の順位を決定し，3 つの順位をもとにして計算した順位を，コード片の提示に用いている．

3.3 問題点

石原らの手法では過去に再利用されたコード片を検出することで，従来手法と比較してユーザに再利用される可能性が高いコード片を提示している．しかし，石原らの手法はコード片の提示の際にプログラム構造については考慮しておらず，例えば Java において中括弧でくくられるような，構文的なまとまりを無視してしまうという問題がある．再利用のみを考慮し，プログラム構造を考慮していないコード片の提示はソースコードの再利用の観点において，必ずしも有益ではない．その理由は，プログラム構造を考慮していないことで，構文的なまとまりに従っていないコード片が提示され，構文の途中で切れているものをコピーアンドペーストしてしまう可能性が生じるからである．その場合，コード片を貼り付けたプ

ログラムにおいて構文エラーが発生するため，構文エラーを修正するための作業が必要となるなどの問題も考えられる．図 6 は石原らの手法によって提示された，再利用の観点において有益ではないコード片の例を示す．図中のハイライトされた部分は石原らの手法によって提示されたコード片を表し，赤い枠線で囲まれた部分は構文的なまとまりを示す．この提示において 240 行目から始まる if 文と 245 行目から始まる if 文は途中までしか提示範囲に含まれていない．よって，提示されたコード片のみをコピーアンドペーストして再利用する場合，ユーザは自分で 246 行目から 263 行目を追加しなければならない．これにより，ソースコードの効率的な再利用が妨げられる可能性がある．

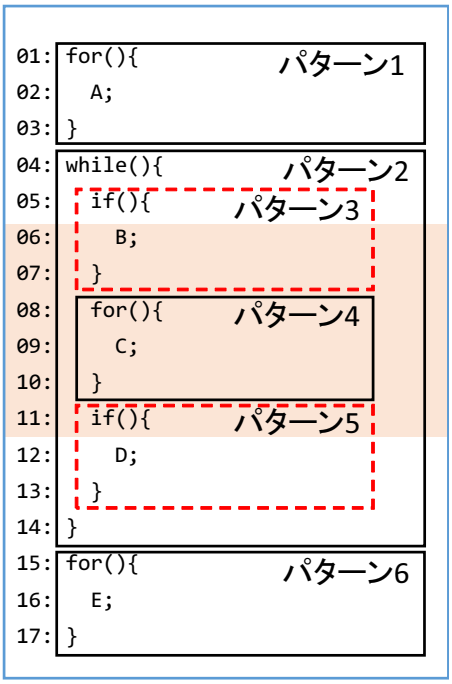
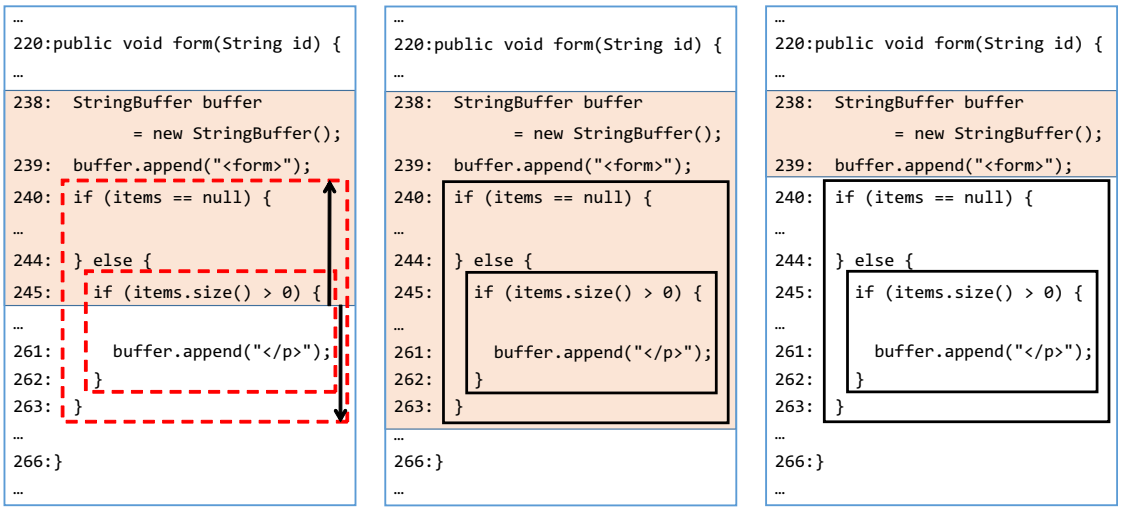


図 7: 石原らの手法による提示範囲とブロックの範囲の位置関係

表 1: 図 7 におけるパターンとブロックの範囲の対応

パターン	ブロックの範囲
パターン 1	提示範囲の前から提示範囲の前まで
パターン 2	提示範囲の前から提示範囲の後まで
パターン 3	提示範囲の前から提示範囲内まで
パターン 4	提示範囲内から提示範囲内まで
パターン 5	提示範囲内から提示範囲の後まで
パターン 6	提示範囲の後から提示範囲の後まで



(a) 提案手法による調整を行う前

(b) 分断されていたブロックを全て含む場合

(c) 分断されていたブロックを全て除く場合

図 8: 提案手法を用いた提示範囲の調整の例

4 提案手法

4.1 概要

この節では，本研究の提案手法である，再利用実績に基づくコード片検索におけるプログラム構造を考慮した検索結果の提示について説明する．以降，本論文では，Java を対象として説明する．Java において if 文，while 文，for 文などに存在する複数の文が中括弧で囲まれた箇所を本論文では**ブロック**と呼ぶ．

石原らの手法では，ブロックを考慮していないため，ブロックを無視した範囲の提示が行われてしまう．これにより，提示された範囲のみをコピーして再利用することで，貼り付けたプログラムにおいてバグが発生する危険がある．以降，バグが発生する場合について例を用いて説明する．図 7 に，石原らの手法による提示範囲とブロックの範囲の位置関係を示す．図中のハイライトされた部分は石原らの手法によって提示された範囲を表し，赤い点線枠あるいは黒い線で囲まれた部分はブロックの範囲を表す．表 1 に，図 7 におけるパターンとブロックの範囲の対応を示す．このうち，赤い点線枠で囲まれているパターン 3 とパターン 5 の位置にブロックが存在していた場合，3.3 節で挙げたように，ブロックを無視したコード片の提示が行われてしまう．パターン 3 では，ブロックの先頭が提示範囲外に位置する，パ

ターン 5 では，ブロックの最後が提示範囲外に位置する．このため，それらをコピーした場合に，構文エラーが発生する可能性がある．本研究では，そういった欠点を克服するために，石原らの手法で提示される範囲を，ブロックを考慮した範囲に調整する手法を提案する．以降，本論文では，パターン 3 またはパターン 5 のように，ブロックの一部が提示範囲外に位置することを**ブロックが分断されている**と呼ぶ．ここまでで述べたように，ブロックの分断が発生する範囲を提示する場合，提示範囲を調整する必要がある．提示範囲の調整方法として，以下の 2 つを設ける．

1. 提示する範囲を拡大し，分断されているブロックを提示範囲に含む
2. 提示する範囲を縮小し，分断されているブロックを提示範囲から除く

ブロックの分断は，パターン 3 あるいはパターン 5 のように，提示範囲の上端あるいは下端において発生する可能性があり，上端あるいは下端のそれぞれについて上記の 2 つの方法のいずれかが選択され，提示範囲の調整が行われる．ここでは，提示範囲の下端にブロックの分断が発生した場合について例を用いて説明する．図 8 は，石原らの手法によって提示されたコード片のうち，ブロックが分断されているものについて，本研究の提案手法を用いることで，その提示範囲を調整する例を示す．図 8(a) におけるハイライトされた部分は石原らのツールによって提示される範囲を示し，図 8(b)(c) におけるハイライトされた部分は提案手法によって提示される範囲を示す．また，赤い点線枠で囲まれた部分は分断されているブロックを示し，黒い実線枠で囲まれた部分は分断されていないブロックを示す．矢印は範囲の調整の方向を示す．

図 8(a) では 240 行目から 263 行目までのブロックと 245 行目から 262 行目までのブロックが分断されている．このとき分断されたブロックを提示範囲に含むように提示する範囲を拡大すると，図 8(b) のようにブロックが分断されることのない範囲を提示できる．また，分断されたブロックを提示範囲から除くように提示する範囲を縮小すると，図 8(c) のようにブロックが分断されることのない範囲を提示できる．

4.2 提案手法の詳細

4.1 節で述べた提案手法を実装し，コード片検索を行うツールを開発した．本研究では石原らの実装したツールに機能を追加することで提案手法を実装している．図 9 に実装したツールの概要を示す．このツールによるコード検索は以下の 8 ステップで行われる．

STEP1: コードクローン検出

石原らのツールを用いて，ソースコード集合に含まれるコードクローンを検出する．

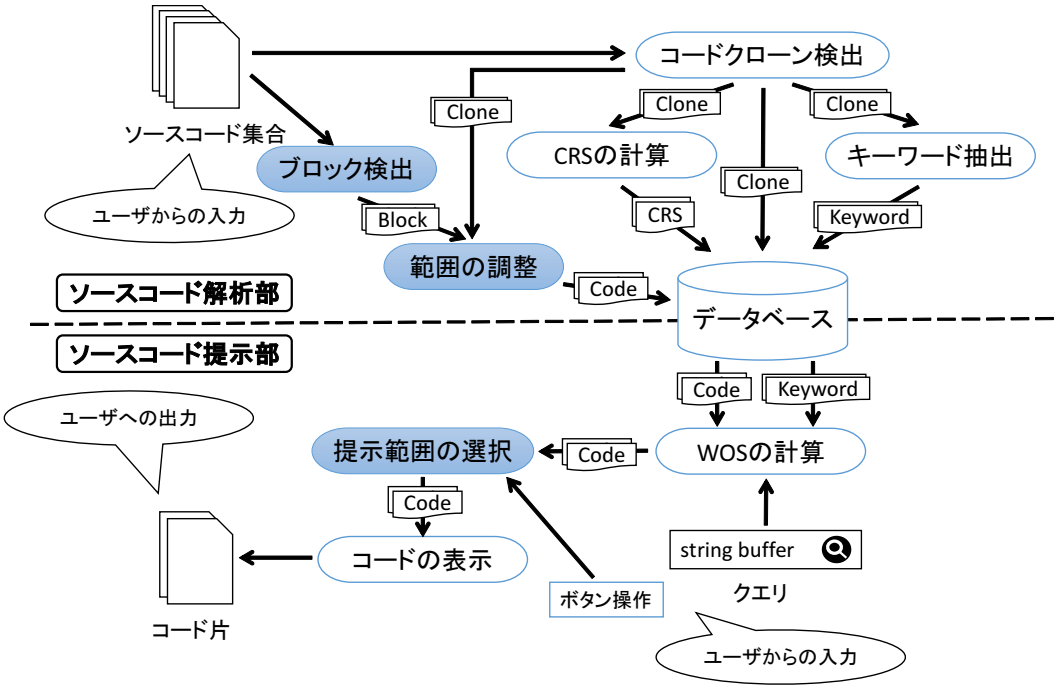

ソースコード集合
ユーザからの入力
ブロック検出
Block
Clone
コードクローン検出
Clone
Clone
CRSの計算
Clone
キーワード抽出
CRS
Keyword
範囲の調整
Code
データベース
ソースコード解析部
ソースコード提示部
Code
Keyword
ユーザへの出力
提示範囲の選択
Code
WOSの計算
Code
コードの表示
string buffer
クエリ
ボタン操作
コード片
ユーザからの入力

図 9: 提案手法の実装

STEP2: キーワード抽出

石原らのツールを用いて，検出されたコードクローンについて，その中に含まれるキーワードを検出する．

STEP3: CRS の計算

石原らのツールを用いて，検出されたコードクローンについて，そのコードクローンを含むメソッドの重要度や，再利用された回数をもとに順位を計算する．

STEP4: ブロック検出

ソースコード集合に含まれるブロックを検出する．

STEP5: 範囲の調整

検出されたコードクローンとブロックの位置情報をもとに，ブロックが分断されている箇所について提示範囲を調整し，データベースに保存する．

STEP6: WOS の計算

石原らのツールを用いて，検出されたコードクローンと入力されたクエリの関連性の強さをもとに順位を計算する．

STEP7: 提示範囲の選択

ユーザからのボタン操作を受けつけ，調整した範囲のうちから提示するものを決定する．

STEP8: コードの表示

石原らの実装したツールのインターフェースを用いて，提示範囲にハイライトを加えて画面に表示する．

図 9 に示された処理のうち，色付けされていない処理は石原らの研究での実装を引き継いだ処理である．本研究では，図 9 に示された処理のうち，青色に色付けした処理を新たに実装した．以降，本研究において実装を行った処理について，その実装方法を述べる．

4.2.1 ブロック検出

実装したツールでは，ソースファイルから抽象構文木を作成する．抽象構文木の作成には Java Development Tools[16]（以降 JDT と呼ぶ）を使用している．JDT で定義された文のうち，ブロックを持つことのできる文を以下に挙げる．

- do while 文
- 拡張 For 文
- for 文
- if 文
- label 文
- switch 文
- synchronized 文
- try 文
- while 文

これらのノード以下の部分木を取り出すことによってブロック検出を実現している．

4.2.2 範囲の調整

実装したツールでは，検出したコードクローンを順に取り出し，そのコードクローンが存在するファイル中に含まれる全てのブロックと範囲を比較する．範囲の比較によって分断されていると判定されたブロックについて，それぞれ 4.1 節で挙げた 2 つの調整パターンにあわせて範囲の調整を行う．範囲の調整はコードクローンの上端あるいは下端のいずれに対しても行われるが，ここでは図 10 に示す例を用い，コードクローンの下端における，ブロックに着目した範囲の調整を説明する．図 10 において，ハイライトされた部分は調整される前のコードクローンの範囲を示す．赤い点線枠はコードクローンによって分断されているブロックの範囲を示す．青い実線は，分断されているブロックを含んだ場合の提示範囲の下端を示し，緑の実線は，分断されているブロックを除いた場合の提示範囲の下端を示す．範囲の調整が行われる前には，図 10(a) に示すように，青い実線および緑の実線は，調整前のコードクローンの下端と同じ位置にある．図 10(b) に示すように，240 行目から 256 行目までのブロックに着目して 2 つの調整方法により範囲の調整を行うと，青い実線は 256 行目の直後に移動する．つまり，ブロックを含むように拡大した場合の提示範囲は 256 行目までを

(a) 範囲の調整を行う前の状態

(b) 240 行目からのブロックに着目した場合

(c) 245 行目からのブロックに着目した場合

図 10: あるブロックに着目した範囲の調整

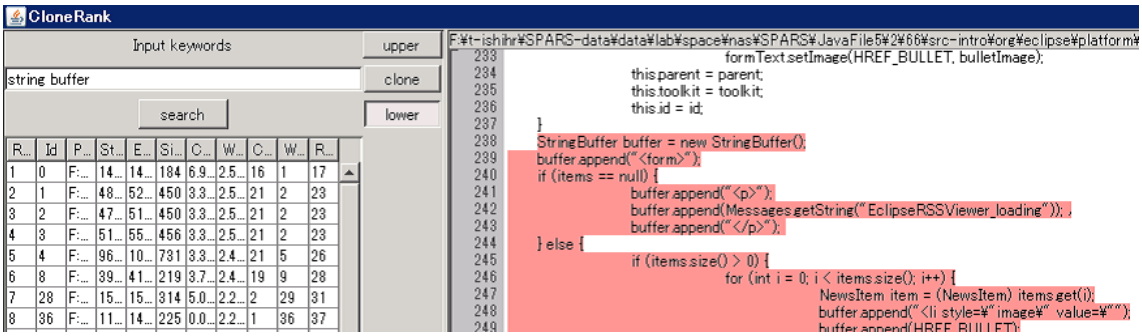


図 11: 実装したツールのインターフェース

含むことになる．また緑の実線は 240 行目の直前に移動する．つまり，ブロックを除くように縮小した場合の提示範囲は 239 行目までを含むことになる．次に，図 10(c) に示すように，245 行目から 255 行目のブロックに着目する．このとき，すでに拡大された範囲においてこのブロックは含まれており，また，すでに縮小された範囲においてこのブロックは除かれている．よって，このブロックに着目した範囲の調整は必要ない．こうした範囲の調整を全てのコードクローンに対して行い，結果として得られた，図 10 において青い実線あるいは緑の実線で示した，拡大後あるいは縮小後の範囲の情報を付与してデータベースに保存する．

4.2.3 提示範囲の選択

実装したツールは，4.2.2 節における範囲の調整で得られた範囲のうち，元となったコードクローンの範囲に近い範囲を最初にユーザに提示する．図 10 の場合，青い実線まで範囲を拡大すると，元のコードクローンとの行数の差は 11 行となるが，緑の実線まで範囲を縮小すると，元のコードクローンとの行数の差は 6 行となる．よって，この例においてツールは，縮小した範囲の方が元のコードクローンとの行数の差が小さいため，元のコードクローンに近い範囲として，縮小した範囲，つまり 238 行目から 239 行目にハイライトを加えてユーザに提示する．また，最初に提示された範囲に対してユーザはボタン操作を行うことによって提示範囲を再調整することが可能である．図 11 に実装したツールのインターフェースを示す．画面中央の 3 つのボタンによって提示範囲の再調整を行うことが可能である．それぞれの機能を以下にまとめる．

clone 提案手法による提示範囲と，その元となったコードクローンの範囲を重ねて表示する．これによって，提示範囲のうち，どの部分が提案手法による調整によって新たに含まれた，あるいは除かれたかを知ることができる．

upper，lower 提示範囲の上端あるいは下端について，範囲の拡大および縮小を選択することができる．すなわち，提示範囲の上端あるいは下端が，ブロックを含むように拡

大されていた場合，ボタンを押すと，ブロックを除くように縮小した範囲に再調整され提示される．提示範囲の上端あるいは下端が，ブロックを除くように縮小されていた場合，ボタンを押すと，ブロックを含むように拡大した範囲に再調整され提示される．upper が上端の調整に対応し，lower が下端の調整に対応する．

5　実験

提案手法の有効性を評価するために，4 章で実装したツールを用いて比較実験を行った．

5.1　準備

実験では，比較のために以下の 3 つのツールを使用した．

- コードクローン検出を行わず，プログラム言語の構造のみを使ってコード片を提示するもの．このツールはメソッド単位でコード片を提示する．また，メソッドの重要度と，クエリとそのコード片の関連性の強さを順位付けに反映させて提示させる．これは既存のコード片検索に対応する．以降このツールを A と呼ぶ．

- コードクローン検出を行い，コードクローン単位でコード片を提示するもの．このツールでは，メソッドの重要度，クエリとそのコード片の関連性の強さ，また，コードクローンから判断した過去に行われた再利用の回数を順位付けに反映させて提示する．これは石原らの手法に対応する．以降このツールを B と呼ぶ．

- ツール B に手を加え，コード片の提示において，ブロックを考慮した範囲の調整を行うように改造したもの．これは 4 章で実装したツールであり，提案手法に対応する．以降このツールを C と呼ぶ．

これらのツールの比較を表 2 にまとめた．B と C を比較することで，検索結果の提示においてブロックを考慮することが効率的な再利用の支援に有効であるかを評価できる．また，A と B または C を比較することで再利用単位でのコード片の提示が有効であるかを評価できる．本研究では，実験対象のソースコード集合として既存研究である *SPARS* で使用されているソースコード集合を使用した [1]．このソースコード集合は約 400 のプロジェクトで構成されており，約 19 万の Java ファイルが存在する．このソースコード集合に対して各ツールを適用し，作成したデータベースを実験に使用した．

5.2　実験方法

本研究では，被験者に複数のタスクを与え，被験者は 3 つのツールのいずれかを使用して与えられたタスクを完成させる．初めに，被験者に対して著者がツールの使い方と実験の手順を説明する．説明の後，被験者はそれぞれ独立にタスクを開始する．各タスクにおいて，被験者は自分で入力するクエリを決める．入力したクエリに合うコード片が存在しない場合は，被験者は入力するクエリを自由に変更できる．与えられた終了条件をみたすことを被験者が確認した段階でタスクが終了する．タスク開始から終了までの被験者の作業状況が画面

キャプチャによって録画される．本実験では，実際の再利用の状況を考慮して，クエリの入力回数を制限しなかった．また，各タスクの終了条件は以下の 4 つである．

1. 用意されたユニットテストを通過する．
2. GUI の実装を行うためユニットテストが用意されていないタスクについては，用意された完成例を確認し，実装が完了したと判断した時点で実験を監督している者からチェックを受け，合格する．
3. 5 分以上検索を行っても再利用が可能なコード片が提示されない．
4. 20 分以上実装を行ってもタスクを完了できる見込みがない．

1 と 2 の場合はタスク完成としてタスク開始からタスク終了までの時間を記録する．3 と 4 の場合はタスク失敗としてその旨を記録する．また，タスクを完成させるにあたり，被験者に下記の制限が課される．

- 指定したツール以外の手段でのコード検索を使用しない．
- タスクの完成において，かならず 1 度は指定したツールを使用したソースコードの再利用を行う．

5.2.1 被験者

本実験の被験者は，大阪大学基礎工学部情報科学科に所属する学生 2 人，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に所属する修士課程の学生 5 人と博士課程の学生 2 人の計 9 人である．すべての被験者は，Java の経験が半年以上あり，また過去に実装した Java プログラムの総行数が 5,000 行以上ある．また，被験者の Java の使用目的として，全員が研究での使用を挙げている．本実験では，被験者は，学年を考慮した上で 3 つのグループに分けられる．以下に，各グループの人数構成を示す．

- グループ 1: 博士課程 1 人，修士課程 1 人，学士課程 1 人
- グループ 2: 博士課程 1 人，修士課程 1 人，学士課程 1 人
- グループ 3: 修士課程 3 人

被験者は，与えられたタスクに対し，表 3 で示されるツールを使用してタスクを完成させる．

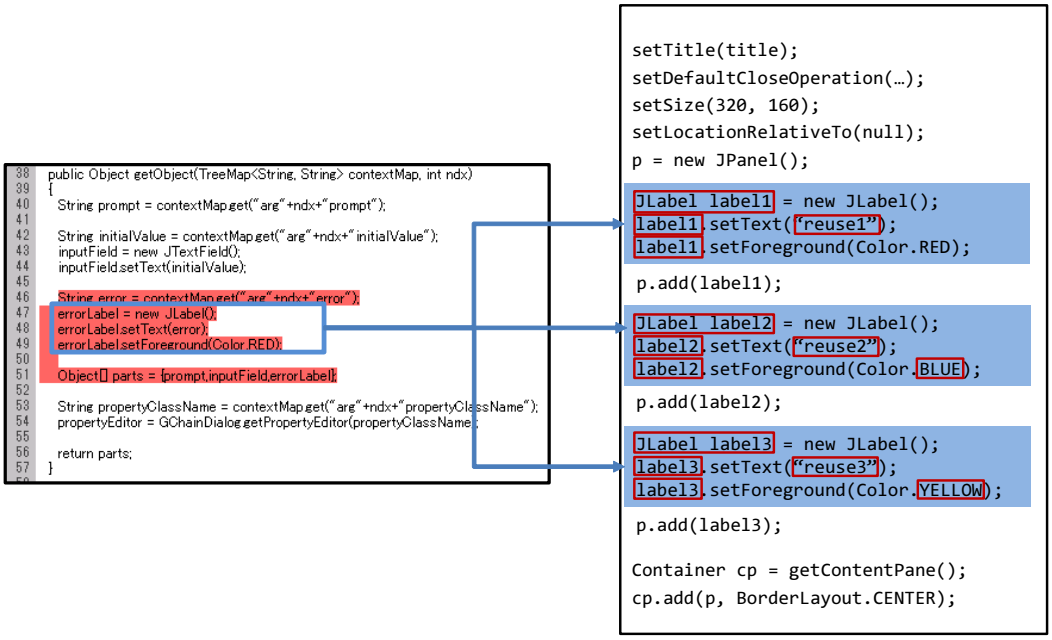


図 12: ツールを用いた実装例

5.2.2 タスク

本実験で被験者に与えられるタスクの数は 9 であり，その全てが与えられた仕様を満たすメソッドを実装するというものである．被験者には宣言部 (修飾子，返り値，名前，引数) だけ記述されたメソッドが与えられ，そのメソッドの満たすべき仕様がコメント部分に記述されている．被験者はそのメソッドの本文を実装する．タスクにはユニットテストあるいは完成例が用意されている．ユニットテストが用意されているタスクについては，ユニットテストを通過した時点でタスクが終了したと判断する．GUI の実装を行うためユニットテストが用意されていないタスクについては，用意された完成例を確認し，実装が完了したと判断した時点で実験を監督している者からチェックを受け，合格した場合にタスクが終了したと判断する．タスクを終了した時点での実装のうち，宣言部とコメント部分を除いた全ての実装を，仕様を満たすために被験者が行った実装として記録する．表 4 は本実験で使用したタスクの概要を示している．

5.2.3 評価基準

本実験では，ソースコードの再利用が，ツールによってどれだけ効率的に支援されているかを，以下に挙げる 3 つの指標を用いて判断した．

利用率 必要なコード片のみを提示しているか

貢献率 実装の助けになっているか

所要時間 タスクの完成にどれだけかかったか

利用率はツールによって提示された文の数のうち，コピーして使用された文の数の割合で計算する．貢献率は被験者が実装した文の数のうち，提示したコード片からのコピーによって実装された文の数の割合で計算する．所要時間は被験者がクエリの 1 文字目あるいは実装の 1 文字目を入力した時点から，テストに通った時点までにかかった時間から計算する．テストに通らなかった場合については，被験者が完成させたタスクのうち最も時間のかかったものと等しい時間がかかったものとみなす．利用率と貢献率の計算式を式 1，式 2 に示す．

$$\text{利用率} = \frac{\text{コピーして使用した文の数}}{\text{ツールが提示した文の数}} \tag{1}$$

$$\text{貢献率} = \frac{\text{コピーによって実装した文の数}}{\text{被験者が実装した文の数}} \tag{2}$$

図 12 はツールを用いた実装の例である．図 12 を用いて利用率と貢献率の算出例を示す．図 12 において，左のソースコードのうち，赤くハイライトされた部分はツールによって提示されたコード片であり，ここでは 5 文ある．青い枠線で囲まれた部分は，被験者がコピーしたコード片を示し，ここでは 3 文ある．コピーしたコード片は青い矢印によって示された箇所に貼り付けられている．右のソースコードは被験者が実装したソースコードを示し，ここでは 19 文ある．右のソースコードのうち青くハイライトされた部分は，コピーしたコード片に被験者が修正を加えて実装したコード片を示し，ここでは 9 文ある．このうち，赤い枠線で囲まれた部分が，被験者が修正を加えた部分である．このとき利用率は，

$$\text{利用率} = \frac{3}{5} \tag{3}$$

となり，貢献率は，

$$\text{貢献率} = \frac{9}{19} \tag{4}$$

となる．利用率及び貢献率が高い，あるいは所要時間が短いほど，ソースコードの再利用を効率的に支援していると考えられる．そこで，利用率及び貢献率が高い，あるいは所要時間が短いことを，指標が良い値であると定義し，その逆であれば，指標が悪い値であると定義する．

5.3 実験結果

図 13 は利用率についてのグラフを示し，縦軸は利用率，横軸は被験者がタスクにおいて使用したツールをそれぞれ表している．各タスクにおいて，ツールごとに利用率の平均を求

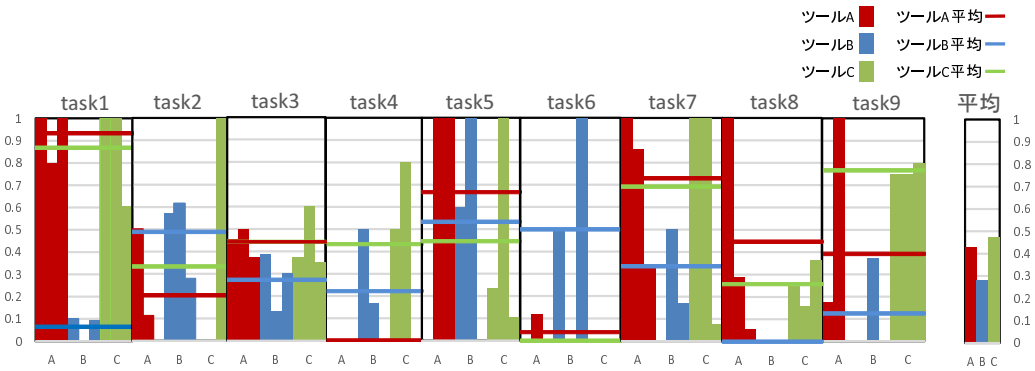


図 13: 利用率

めており，グラフ中に横線として示している．また，ツールごとの利用率について全体の平均を図右端に示している．図 13 より，利用率において全体の平均ではツール C とツール A が比較的良い．タスクごとの平均ではツール A が最良となるタスクが多い．図 14 は貢献率についてのグラフを示し，縦軸は貢献率，横軸は被験者がタスクにおいて使用したツールをそれぞれ表している．各タスクにおいて，ツールごとに貢献率の平均を求めており，グラフ中に横線として示している．また，ツールごとの貢献率について全体の平均を図右端に示している．図 14 より，貢献率において全体の平均ではツール C が比較的良い．タスクごとの平均ではツール B とツール C が最良となるタスクが多い．図 15 は所要時間についてのグラフを示し，縦軸は被験者がタスクにおいて使用したツール，横軸は所要時間をそれぞれ表している．テストに通らなかった場合については，被験者が完成させたタスクのうち最も時間のかかったものと等しい時間がかかったものとみなしており，ここではその値は 1706 である．図 15 より，所要時間において全体の平均ではツール B が比較的良い．タスクごとの平均でもツール B が最良となるタスクが多い．なお，実験においてツールの選択が正しく行われなかったミスがあり，task7 と task8 においてツール B のデータが 1 つ足りておらず，いずれのグラフにおいても空欄としてある．

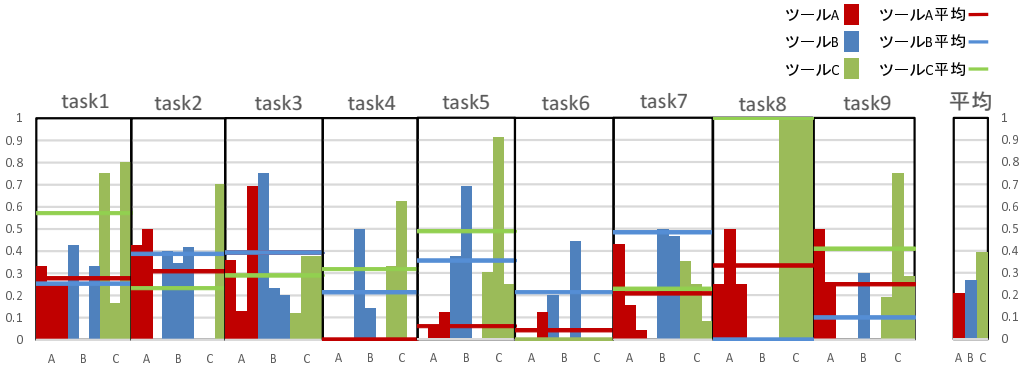


図 14: 貢献率

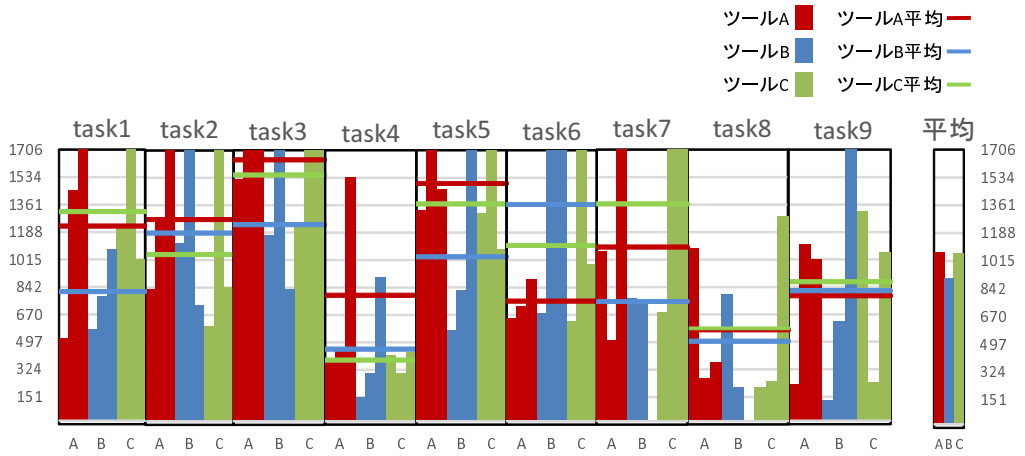


図 15: 所要時間

6　考察

実験結果において，データの数が少ないため，統計的な手法は用いない．そこで，各タスクにおいて，それぞれの指標を最も良い値にするツールを調査した．表 5 に，各タスクにおいて，各指標が最良となるツールを示した．

6.1　ツール A について

表 5 を見ると，5 つのタスクにおいてツール A の利用率が最良だが，ツール A の貢献率が最良となるタスクは存在しない．このような結果が得られた要因を調査したところ，ツール A の利用率が良いタスクの実装において，特徴的な再利用が行われていた．図 16 に，ツール A による再利用の例を示す．図 16 において，赤くハイライトされた範囲は，ツール A による提示範囲を示す．青い枠線で囲まれた範囲は，被験者によってコピーされた範囲を示す．また，提示範囲のうち緑の枠線で囲まれた箇所において，提示範囲外のメソッドを呼び出しており，ここでは橙の枠線で囲まれたメソッドを呼び出している．このとき，提示範囲の全てをコピーしているので，利用率は高くなるが，提示範囲外からも多くコピーしているため，貢献率は低くなる．このように，ツール A は再利用することのできるコード片を提示しているが，再利用の結果としてそれ以外の実装を多く必要とする，という場面が見られた．これは，ツール A による再利用を行うことで，より多くの実装が必要になっているということを意味する．ここから，ツール A すなわち既存のコード片検索は，再利用実績に基づくコード片検索と比べて，効率的な再利用を支援できていないといえる．

6.2　ツール B とツール C の比較

表 6 に，ツール B とツール C を比較して，各指標をより良い値にするツールを示した．表 6 を見ると，利用率においてツール C が最良となっているタスクは 6 つあり，貢献率においてツール C が最良となっているタスクは 5 つある．ここから，再利用実績に基づくコード片検索において，提案手法によって提示範囲の調整を行うことで，実装において有効なコード片を提示できているといえる．

6.3　ツール C がうまく働かない例

図 13 と図 14 を見ると，どちらの指標においてもツール C が悪くなっているタスクがある．このような結果が得られた要因を調査したところ，再利用できるコード片が提示されているにもかかわらず被験者が見逃している場面が見られた．図 17 に，被験者が見逃したコード片の例を示す．図 17 において，赤くハイライトされた範囲はツール C によって最初に提示される範囲を示し，青い実線で囲まれた範囲は元となったコードクローンの範囲を示

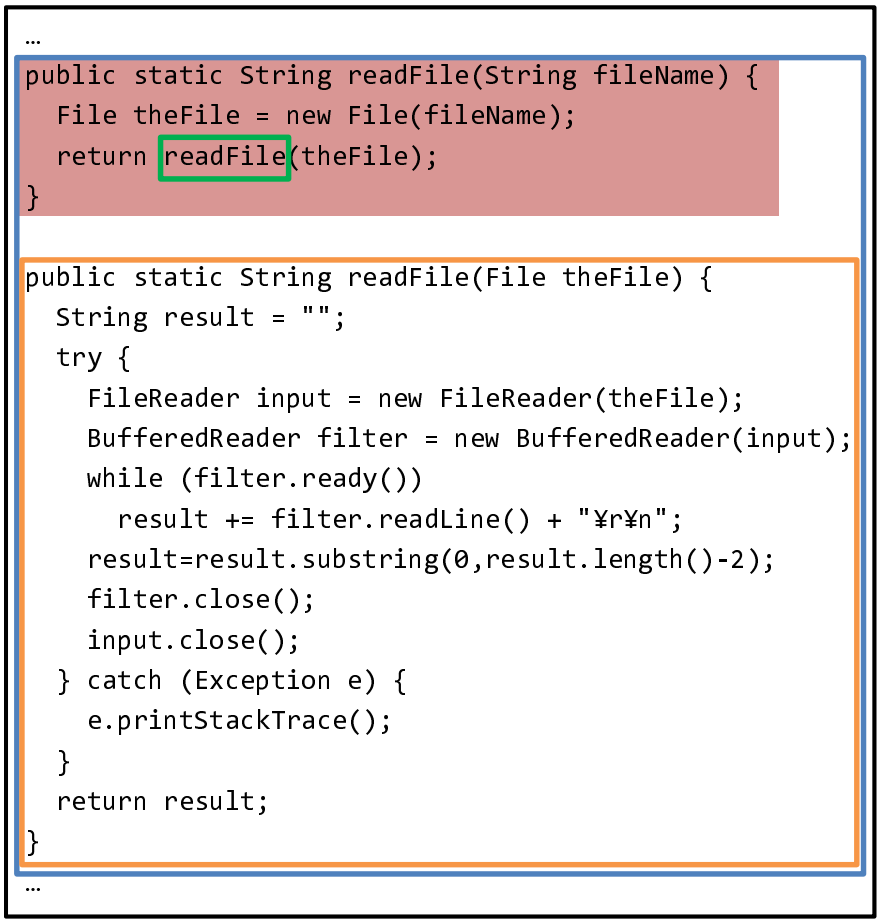


図 16: ツール A による再利用の例

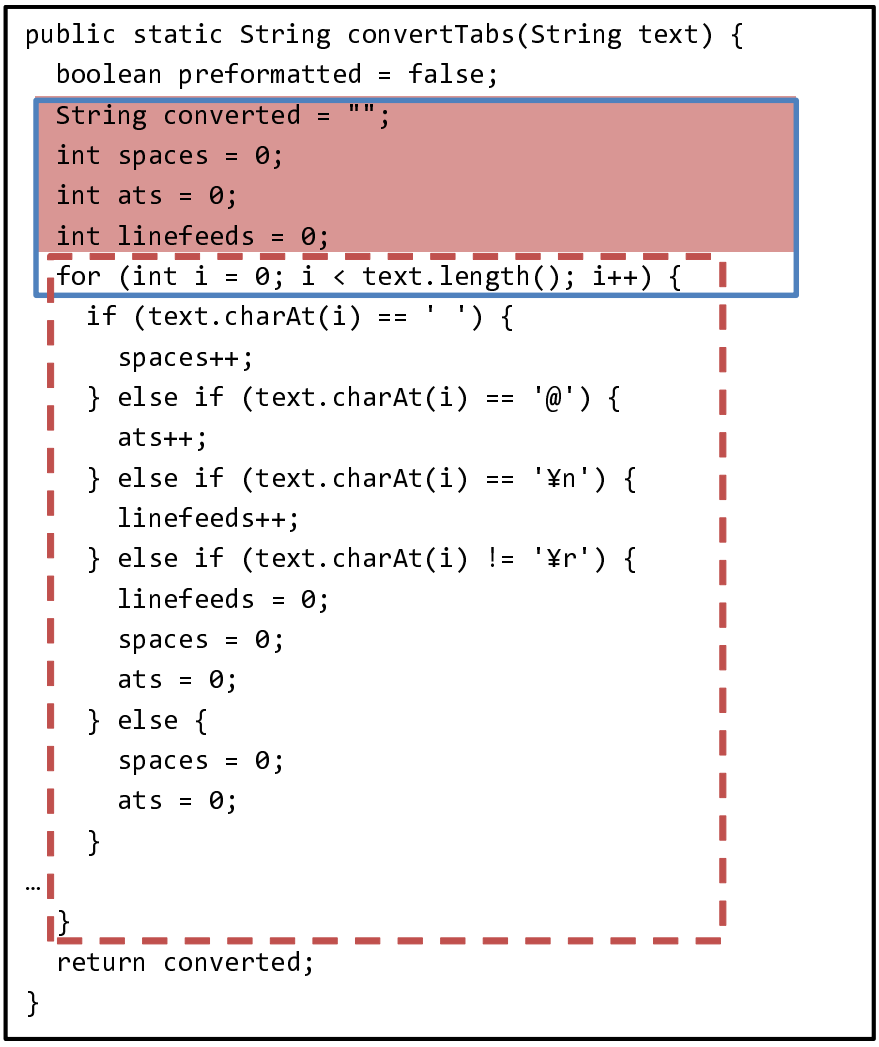


図 17: 被験者が見逃したコード片の例

図 18: コード片の提示方法の改良案

す．また，赤い点線で囲まれた範囲は，元となったコードクローンによって分断されていたブロックの範囲を示す．この例において，ブロックの範囲内のコード片を再利用することが可能であった．このとき，ツール C はブロックを含む範囲と，ブロックを除く範囲を比較し，元のコードクローンに近い範囲として，ブロックを除いた範囲を最初に提示した．その結果，被験者は本来再利用できるはずのコード片を，再利用できないものとして見逃した．このように，再利用を支援できるはずの場合であっても，最初に提示される範囲が適切でなければ，被験者が見逃してしまい，再利用を支援することができなくなる．

この問題を解決するために，提示範囲の調整を行う際に，ブロックの分断だけでなく，変数の依存関係も考慮して調整を行うことが考えられる．図 17 では，元となったコードクローンの範囲内で宣言されている変数 converted，spaces，ats や linefeeds がブロックの範囲内において何度も利用されているため，変数の依存関係を考慮して，ブロックに含まれるコード片が再利用に有効であると判断することができる．ここから，ブロックを含む範囲とブロックを除く場合の 2 通りの調整方法のうち，ブロックを含む範囲をユーザに最初に提示することができる．こういった改良によって，再利用の支援をより効率的に行えると考えられる．

6.4 所要時間について

利用率，貢献率が良く，効率的な再利用が行われていると考えられるタスクにおいて，所要時間が悪くなっている場合がある．このような結果が得られた要因を調査したところ，より多くのコード片を再利用することにより，再利用したコード片に対する修正に多くの時間を費やしている場面が見られた．再利用した結果，実装にかかる時間が長くなってしまったのでは，効率的な再利用を支援しているとは言えない．この問題を解決するために，ユーザにコード片を提示する際に，提示範囲外で宣言された変数が，提示範囲内で使用される箇所を空欄にするという手法が考えられる．図 18 に，コード片の提示の改良案を示す．図 18 において，赤くハイライトされた範囲は，ツールによって提示される範囲を示す．図 18 において，左の図は，提案手法による提示の例を示し，右の図が改良後の提示の例を示す．この例では，提示範囲外で宣言された変数 p に対応する箇所を青い枠で，label1 に対応する箇所を赤い枠で囲い，空欄としている．ユーザは再利用の際に，必要に応じて空欄を埋めて再利用することができる．

7　妥当性への脅威

本研究の結果の妥当性に関して，以下で挙げる点に留意する必要がある．

被験者

本実験において，被験者は全員 Java の経験が半年以上あり，Java によって過去に 5,000 行以上の開発を行っているが，被験者の間で Java プログラミングの能力に大きな隔たりがあった場合，実験結果に大きな影響を与えることになる．ただし，被験者の学年を考慮してグループ分けを行ったため，グループ間で能力の差がそれほど大きくないと考えられる．

タスク

本実験において，被験者に与えられるタスクの内容に重複があり，また，タスクの順番が固定されているため，タスクの順番によって実験結果に影響を与える可能性がある．

ツール

本実験において，被験者がツールを使用する順番を 3 パターン設定しているが，全てのパターンを網羅しているわけではないため，全てのパターンを網羅した実験を行った場合に実験結果が変わる可能性がある．

データベース

本実験において，比較するツールは全て，実験対象となるソースコード集合の解析によって得られたデータベースを使用する．よって，ツールがどのようなコード片を提示するかは，実験対象とするソースコード集合に依存する．そのため，実験対象のソースコード集合を変更することで結果が変わる可能性がある．

実装

石原らの手法では，データベースの作成を高速化するために，いくつかの近似計算を実装している．そのため，高性能マシンを使用する等で厳密な計算を行った場合，本実験と異なる結果が得られる可能性がある．また，6 章で説明したように，提案手法ではユーザに最初に提示するコード片の範囲を自動で計算している．この計算方法を変更した場合，本研究と異なる結果が得られる可能性がある．

8 関連研究

ソースコードの再利用を目的としたソースコード検索システムとして，Inoue らによる SPARS[1] や，McMillan らによる Portfolio[3] が挙げられる．Inoue らは，関数の呼び出し関係によりそれぞれの関数の重要度を計算する Component Rank 法と，ソースコードにおいてキーワードが存在する位置によってキーワードの重要度を計算する Keyword Rank 法を提案している．また，それらを実装したソースコード検索システムである SPARS を開発した．Component Rank 法では，多くの関数から呼び出される関数や，重要な関数から呼び出される関数は重要度が高くなる．加えて，Component Rank 法では，類似する関数をグループ化し，グループそれぞれに対して重要度を与える．グループの重要度は，そのグループの要素の重要度の合計で計算される．Keyword Rank 法では，ソースコードから抽出したキーワードが，そのソースコードの内容を表す上でどれだけの重要度を持っているかを計算する．ここでは，抽出したキーワードについて，そのトークンの種類に応じて異なる重要度を与えている．トークンの種類が重要であるほどキーワードの重要度が高くなる．例えば，クラス定義名やメソッド定義名から抽出されたキーワードは重要度が高くなる．

McMillan らは，関連する関数及びその使用法を知るために，開発者が関数をたどる際のふるまいを表したモデルである Navigation Model とキーワードの関連性を表したモデルである Association Model を提案した．また，それらを実装したソースコード検索システムである Portfolio を開発した．Navigation Model では，関数の呼び出し関係を基にそれぞれの関数の重要度を計算しており，ウェブページの重要度を決定する PageRank 法を応用したものとなっている [17]．Association Model では Spreading Activation 法によってキーワード間の関連性を計算している [18][19]．また，Portfolio では，ソースコードを提示する際に関数のコールグラフも提示する．ユーザはグラフをたどることで，関数の使用法を知ることができる．

上記の 2 つの既存手法は関数の呼び出し関係から検索結果の重要度を計算するという点で提案手法と類似しており，特に SPARS は類似関数をグループ化する点でも共通している．しかし，これらの手法はプログラム構造のみを基にソースコードを提示している．一方で，提案手法ではコードクローンを基にソースコードを提示している．そのため，ユーザが規模の小さい機能を再利用する際に，ユーザに必要な部分だけを提示するという点で，提案手法は既存手法と比較して，より効率的に再利用を支援しているといえる．

実用的な再利用を支援するシステムとして，Holmes らによる Gilligan が挙げられる [20]．Holmes らは，開発者が再利用したいと考えている機能を見つけ出し，保存されている実用的な再利用情報を基に半自動的にその機能を補完する手法を提案し，これを実装した Gilligan を開発した．Holmes らの手法では，ユーザが再利用したいと考えているソースコードを，

ユーザ自身でがステムに入力する必要があるのに対し，提案手法では，あらかじめ多くのソースコードからコード片の情報を抽出し構築したデータベースを使用するため，ユーザが把握していないシステムからの再利用が可能である．

9　おわりに

本研究では，再利用実績に基づいたコード片検索において発生していた再利用後の修正のコストを削減するため，プログラム構造を考慮した検索結果の提示手法を提案した．また，その手法の有効性を調べるために，9 人の被験者にタスクを与え，被験者がタスクの完成において実装したソースコードと，ツールによって提示したコード片を比較するという実験を行った．被験者は提案手法を実装したツールを含んだ 3 つのツールを用いて与えられたタスクを完成させた．

実験の結果，提案手法は既存手法と比べ効率的な再利用支援を行うことができることが確認された．本研究の今後の課題は以下のとおりである．

- 提案手法をウェブシステムとして実装し，多くの人に使用してもらえる環境を構築する．これによって多くの被験者を募って大規模な実験を行うことができ，提案手法の有効性を確認することができる．
- 提示するコード片の範囲を調整する際に，プログラム構造だけでなく変数の依存関係を考慮する．
- コード片の提示において，再利用が容易となるようにコード片を変形して提示する．

謝辞

本研究を行うにあたり，理解あるご指導を賜り，常に励まして頂きました 楠本 真二 教授に心から感謝申し上げます.

本研究に関して，有益かつ的確なご助言を頂きました 岡野 浩三 准教授に深く感謝申し上げます.

本研究に多大なるご助言およびご指導を頂きました 井垣 宏 特任准教授に深く感謝致します.

本研究の全過程を通し，終始熱心かつ丁寧なご指導を頂きました 肥後 芳樹 助教に深く感謝申し上げます.

本研究を行うにあたり, 多大なるご助言, ご助力を頂きました, 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博士前期課程 2 年の 石原 知也 氏に深く感謝申し上げます.

本研究を行うにあたり, 適切なご助言, ご助力を頂きました, 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博士前期課程 1 年の 今里 文香 氏に深く感謝申し上げます.

その他，楠本研究室の皆様のご助言，ご協力に心より感謝致します.

また，本研究に至るまでに，講義，演習，実験等でお世話になりました大阪大学基礎工学部情報科学科の諸先生方に，この場を借りて心から御礼申し上げます.

参考文献

[1] K. Inoue, R. Yokomori, T. Yamamoto, M. Matsushita, and S. Kusumoto. Ranking significance of software components based on use relations. *IEEE Transactions Software Engineering*, Vol. 31, No. 3, pp. 213–225, 2005.

[2] M. Grechanik, C. Fu, Q. Xie, C. McMillan, D. Poshyvanyk, and C. Cumby. A search engine for finding highly relevant applications. In *Proc. of the 32nd ACM/IEEE International Conference on Software Engineering*, pp. 475–484, 2010.

[3] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu. Portfolio: Finding relevant functions and their usage. In *Proc. of the 33rd International Conference on Software Engineering*, pp. 111–120, 2011.

[4] T. Ishihara, K. Hotta, Y. Higo, and S. Kusumoto. Reusing reused code. In *Proc. of the 20th Working Conference on Reverse Engineering*, pp. 457–461, 10 2013.

[5] 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.

[6] S. Bellon. Detection of software clones. *Technical Report, Institute for Software Technology, University of Stuttugart*, 2003. available at http://www.bauhaus-stuttgart.de/clones/.

[7] S. Bellon, R. Koschke, G. Antniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. *IEEE Transactions Software Engineering*, Vol. 31, No. 10, pp. 804–818, Oct. 2007.

[8] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multi-linguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, Vol. 28, No. 7, pp. 654–670, July 2002.

[9] I. Baxter, A. Yahin, M. Anna L. Moura, and L. Bier. Clone detection using abstract syntax trees. *Proc. of International Conference on Software Maintenance 98*, pp. 368–377, Mar. 1998.

[10] S. Uchida, A. Monden, N. Ohsugi, T. Kamiya, K. Matsumoto, and H. Kudo. Software analysis by code clones in open source software. *Journal of Computer Information Systems*, Vol. XLV, No. 3, pp. 1–11, Apr. 2005.

[11] 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.

[12] 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. 28–42, 8 2011.

[13] 肥後芳樹, 楠本真二. 実規模ソフトウェアへの適用を目的としたプログラム依存グラフに基づくコードクローン検出法. ソフトウェアエンジニアリング最前線 2008, 2009.

[14] B. Hummel, E. Jurgens, L. Heinemann, and M. Conradt. Index-based code clone detection: incremental, distributed, scalable. In *Proc. of the 26th International Conference on Software Maintenance*, pp. 1–9, 2010.

[15] K. Jones. A statistical interpretation of term specificity and its application in retrieval. *Journal of Documentation*, Vol. 28, No. 1, pp. 11–21, 1972.

[16] Java development tools. http://eclipse.org/jdt/.

[17] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN Systems*, Vol. 30, No. 1-7, pp. 107–117, Apr. 1998.

[18] A. M. Collins and E.F. Loftus. A spreading-activation theory of semantic processing. *Psychological Review*, Vol. 82, No. 6, pp. 407–428, 1975.

[19] F. Crestani. Application of spreading activation techniques in information retrieval. *Artificial Intelligence Review*, Vol. 11, pp. 453–482, 1997.

[20] R. Holmes and R.J. Walker. Systematizing pragmatic software reuse. *ACM Transactions Software Engineering Methodology*, Vol. 21, No. 4, pp. 1–44, 2012.

表 2: 実験で使用したツール

	ツール A	ツール B	ツール C
提示するコード片	メソッド単位	コードクローン単位	コードクローンにブロックを考慮した調整を加えたコード片
メソッドの重要度による順位付け	する	する	する
クエリとの関連性の強さによる順位付け	する	する	する
再利用された回数による順位付け	しない	する	する

表 3: 各グループがタスクを完了させるために使用したツール

	タスク 1,2,3	タスク 4,5,6	タスク 7,8,9
グループ 1	A	B	C
グループ 2	C	A	B
グループ 3	B	C	A

表 4: 実験で使用したタスク

タスク	概要
task1	与えられた文字列を分割し整数に変換しソートする
task2	与えられた文字列から指定した文字を抜き出し指定した書式で出力する
task3	swing によりボタンを実装する
task4	ファイルの作成をし与えられた文字列をファイルへ書き込む
task5	与えられた文字列を分割しソートする
task6	行列の掛け算を行う
task7	与えられたテキストファイルを読み込み単語数を数える
task8	swing によりラベルを実装する
task9	排他的論理和の計算を行う

表 5: 各タスクにおいて良い値となるツール

	task1	task2	task3	task4	task5	task6	task7	task8	task9
利用率	A	B	A	C	A	B	A	A	C
貢献率	C	B	B	C	C	B	B	C	C
所要時間	B	C	B	C	B	A	B	B	A

表 6: 各タスクにおけるツール B とツール C の比較

	task1	task2	task3	task4	task5	task6	task7	task8	task9
利用率	C	B	C	C	B	B	C	C	C
貢献率	C	B	B	C	C	B	B	C	C
所要時間	B	C	B	C	B	C	B	B	B