特別研究報告

題目

プログラム変更支援を目的とした
コードクローン情報付加手法の提案と実装

指導教官
井上 克郎 教授

報告者
佐々木 亨

平成 16 年 2 月 19 日

大阪大学 基礎工学部 情報科学科

平成 15 年度 特別研究報告

プログラム変更支援を目的とした
コードクローン情報付加手法の提案と実装

佐々木 亨

内容梗概

ソフトウェアの保守を困難にするものとしてコードクローンがあげられる．コードクローンとは，ソースコード中に存在するコード片で同形のコード片が他に存在するもののことで，既存コードの「コピーとペースト」による再利用や，頻繁に用いられる同一処理によりプログラム中に作りこまれる．もし，複数のコードクローンを持つコード片にバグが見つかったなら，そのコードクローンすべてに同じ修正を行うかどうかを確認する必要がある．しかし，すべてのコードクローンに対して，確認・修正を行うことはきわめて困難である．

我々の研究グループは，ソースコード中のコードクローンを検出するツール (CCFinder) を開発してきている．CCFinder は，大規模ソフトウェアの保守・デバッグ作業の支援あるいは教育環境におけるプログラム評価支援を目的としている．しかし，コードクローンの検出結果はソースコードの行番号で提示されており，当然ながら，機能追加等のためにソースコードを修正すると，コードクローン情報とソースコードとの間に行番号のずれが生じる．したがって，複数箇所の修正（機能追加）を一度に行う場合には，その行番号のずれを意識しながら作業しなければならず，作業効率が悪くなる．

本研究では，コードクローン検出ツール CCFinder で検出されたコードクローン情報を，コメントとしてソースコード中に付加する手法の提案とツールの試作を行う．本ツールに CCFinder の出力を与えることで，ソースコードにコードクローン情報が付加される．付加されたコメントを利用することで，開発者は行番号のずれを意識せずに作業をすることができる．実際に本ツールをあるソフトウェアプロジェクトの修正事例に対して適用した結果，行番号のずれを意識せずに作業ができ，その有用性を確認した．

主な用語

コードクローン
ソフトウェア保守
デバッグ
コメント

目 次

1 まえがき

ソフトウェア保守は,「納入後，ソフトウェア・プロダクトに対して加えられる，フォールトの修正，性能またはその他の性質改善，変更された環境に対するプロダクトの適応のための改訂」と定義されている [14]．近年，ソフトウェアシステムの大規模化，複雑化に伴い，プログラムの保守・デバッグ作業に要するコストが増加してきている．このため大企業では，既存システムの保守に多くのコストを費やすようになり，保守作業の効率を高めることが，ソフトウェア工学の重要な課題のひとつとなっている．ソフトウェア保守を困難にしている要因の一つとしてコードクローンが指摘されている．

コードクローンとは，ソースコード中に存在するコード片で，同形のコード片が他に存在するものである．それらは，既存システムに対する機能変更や拡張時における「コピーとペースト」による安易な機能的再利用の際に発生する．もし，あるコード片にバグが含まれていた場合，そのコード片に対するコードクローンに対してもれなく，修正を行うか確認しなければならない．また，保守性を高めるため，同値類を一つのサブルーチン等にまとめるのがよい場合もある．そのためには，コードクローンをすべて検出することが必要となる．

これまでさまざまなコードクローン検出法が提案されている．我々の研究グループもトークン単位でのコードクローンを検出するツール (CCFinder[17]) を開発してきており，これまでにさまざまなソフトウェアに対する適用を行った．これらの適用の中で，CCFinder は大規模なソースコードも，細粒度でのコードクローン解析を非常に高速に行うことが可能であると評価されてきた．

一方，CCFinder を利用することで，ソースコード中からコードクローンを探す手間がなくなったが，CCFinder のコードクローン情報は，ソフトウェアの規模が大きくなればなるほど，膨大なものとなり，必要な情報が埋没してしまうおそれがある．そこで我々は，有効なクローン情報を抽出するために，CCFinder の解析結果の参照支援システムを試作し，コードクローンの位置情報の視覚化と，コードクローンに対する単純的な評価尺度を用いてソースコードの参照支援を試みた．ところが，複数箇所の修正を一度に行う場合，機能追加等でソースコードに修正を加える度に，コードクローン情報とソースコードとの間で行番号のずれが生じる．開発者は，この行番号のずれを意識しながら修正箇所を特定するか，あるいは一つの修正を終えるたびにコードクローンの位置情報を更新して修正箇所を見つけ出す必要があった．これは作業効率の低下の原因となり，また修正箇所を間違える可能性を引き起こしかねない．

そこで本研究では，ソフトウェア開発・保守の観点から，デバッグ・機能追加など，複数箇所の変更を一度に行う際に，開発者が行番号のずれを意識せず他の修正箇所を発見するための利用を目的とした，コードクローン情報をコメントとしてソースコードに付加する手法

の提案とツールの試作を行った．

実際には，まず，ソースコード中にコードクローンの位置情報をコメントとして付加する．このコメントの付加はクローンごとに一意に振った ID を利用し，クローンに含まれるすべての行に付加する．結果として，ソースコード上でのクローンの位置情報が行番号以外の方法で表されているので，行番号のずれが生じても開発者はこれを意識することなく，クローンの正しい位置を特定することができる．次に，ソースコード中で修正箇所を特定したときに，そこに付加されているコメントを利用して他の修正箇所を調べ修正を施す．この修正を終えたら不要となったコメントを除去する．これらの作業を支援するため，試作したツールには，コメント付加機能，一時ファイル編集機能，クローンクラス検索機能，コメント除去機能，一時ファイル書き戻し機能を実装した．また，実際のソフトウェアを対象に適用実験を行った結果，利用者が，行番号のずれを意識することなく，デバッグ作業を行うことができ，本手法の有用性を確認することができた．

以降，2 章では，ソフトウェア保守における問題やコードクローン検出に関連した研究について述べ，3 章で，コードクローン情報付加手法の提案を行う．4 章では，本ツールの利用方針とその利点，そしてその実装についての説明を行う．5 章では，実際にソフトウェアを対象にして適用実験を行い，その結果と考察を行う．最後に 6 章で，本研究のまとめと今後の課題について述べる．

2　コードクローン

2.1　コードクローンとソフトウェア保守

コードクローンとは，ソースコード中に存在するコード片で同形のコード片が他に存在するもののことで，いわゆる “重複したコード”のことである．

コードクローンがソフトウェア中に作りこまれる，若しくは発生する原因には次のようなものがある．

既存コードのコピーとペーストによる再利用

近年のソフトウェアの設計手法を利用すれば，構造化や再利用可能な設計が可能である．しかし，ゼロからコードを書くよりも既存のコードをコピーして部分的な変更を加えるほうが信頼性が高いということもあり，実際には，コピーと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存在する．

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，給与税の計算や，キューの挿入処理，データ構造アクセス処理などである．

プログラミング言語の適切な機能の欠如

抽象データ型や，ローカル変数を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ければ，意図的に繰り返し書くことでパフォーマンスの改善を図る場合がある．

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

単純に偶然一致し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．

もし，プログラムコード中にコードクローンが存在した場合には，一般的にコードの変更等が困難であるといわれ，保守容易性低下の一因となっている．このようなコードクローンによる問題に対処する方法としては，以下の 2 つが考えられる [15].

コードクローン情報の文書化

コードクローンに関する情報が文書化され，断続的に保守されている場合には，コードクローンに対する変更は幾分やさしくなる．しかし，すべてのコードクローンに対する情報を常に最新に保つ作業は非常に手間がかかるため，現実的に困難である場合が多い.

コードクローンの自動検出

コードクローンを形式的に定義し，コードクローンをプログラムテキスト中から自動的に検出するためのさまざまな手法が提案され，検出システムが開発されてきている [1][2][3][4][5][6][7][8][11][17][18][19][20][21].
それぞれの手法やツールの特徴を次節で述べる（我々の開発した CCFinder は 2.3 節参照）.

2.2 既存のコードクローン検出法

Covet

文献 [20] で定義された種々の特徴メトリクスのいくつかのメトリクス値を比較することによって，コードクローン検出を行う．検出対象言語は Java である.

CloneDR[8]

抽象構文木 (AST) の節点を比較することによって，コードクローン（類似部分木）の検出を行う．部分的に異なっているコードクローンも検出することが可能であり，検出したコードクローンを自動的に等価なサブルーチンやマクロに置き換えることも可能である．検出対象言語は，C/C++,COBOL,Java,Progress である.

Dup[2][3][4]

前処理として，ユーザ定義名のパラメータ化を行った後，行単位の比較によりコードクローンを検出する．マッチングアルゴリズムには，サフィックス木探索 [13] を用いている.

Duploc[11]

前処理として，空白やコメント等を取り除いた後，行単位（のハッシュ値）での表検索を用いた比較によってコードクローンを検出する．特徴は，言語に依存する処理をほ

とんど行わないため，多くのプログラミング言語に対応することである．また，コードクローンの散布図等の GUI を備えたツールであり，ソースコード参照支援を行う．検出対象言語は，C,COBOL,Python,Smalltalk である．

JPlag[21]

ソースコードを字句解析し，トークン単位での比較を行う．プログラム盗用の検出を目的として開発され，プログラム間の類似率を検出する．検出対象言語は，C/C++,Java である．

Komondoor らの手法 [18]

関数等にまとめるのに適したコードクローンの抽出を目的として，プログラム依存グラフ (PDG) 上での各節点の比較を行うことでコードクローン（同型 (isomorphic) 部分グラフ）を検出する．文字列比較や抽象構文木等を用いた検出法では発見できなかった非連続コードクローンや，対応行の番号が異なるクローン，互いに絡み合ったクローン等を検出可能である．[18] で作成されたツールの検出対象言語は，C である．

Krinke の手法 [19]

AST や TraditionalPDG に似た Fine-grainedPDG というグラフ上での類似 (similar) 部分グラフ（同型部分グラフではない）を検出することで，コードクローンが存在すると思しき場所を検出する．試作ツールの検出対象言語は，C である．

SMC[5][6][7]

まず特徴メトリクスによってソースコードをコードクローンと思しきメソッドに絞り込む．次に絞り込まれたメソッドのペアに対し，表検索を用いることでメソッド単位のコードクローンを検出する．また検出されたペアのメソッドは，特徴により 18 種類に分類される．更にそれぞれの分類については共通のメソッドへの書き換え指針が示されている．

MOSS[1]

検出アルゴリズムは公開されていない．JPlag 同様，プログラム盗用の検出を目的として開発された．検出対象言語は，Ada,C/C++,Java,Lisp,ML,Pascal,Seheme である．

いずれの手法，ツールにおいても提案者によってコードクローンの定義が微妙に異なっており，検出されるコードクローンが異なっている．つまり，コードクローンの定義とは検出アルゴリズムそのものによって定義される．Burd ら [9] も CloneDR,Covet,JPlag,Moss，そして我々の開発した CCFinder を含めた 5 つのツールを用いて，それぞれ検出されるコード

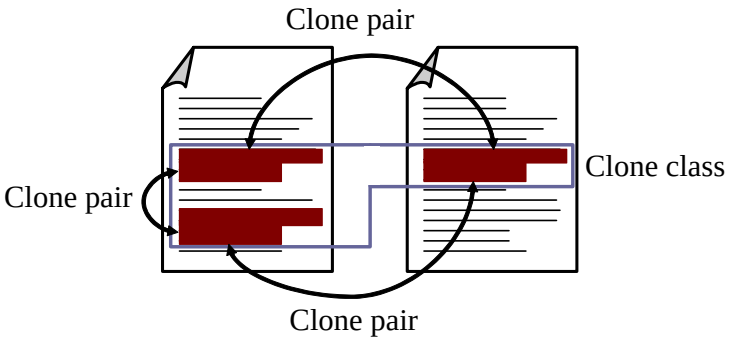

図 1: クローンペアとクローンクラス

クローンの比較を行っているが，すべての面において他のツールより優れているツールはなく，使う場面に応じて，適切なツールを選ぶことが必要となってくると述べている．

2.3 コードクローン検出ツール CCFinder

2.3.1 概要

あるトークン列中に存在する 2 つの部分トークン列 α，β が等価であるとき，α は β のクローンであると定義する（その逆もクローンであるという）．また，(α，β) をクローンペア (図 1 参照) と呼ぶ．α，β それぞれを真に包含するどのようなトークン列も等価でないとき α，β を極大クローンという．また，クローンの同値類をクローンクラス (図 1 参照) と呼び，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．

CCFinder は，単一または複数のファイルのソースコード中からすべての極大クローンを検出し，それをクローンペアの位置情報として出力する．CCFinder のもつ主な特徴は次の通りである．

細かい粒度のコードクローンを検出

字句解析を行うことにより，トークン単位でのコードクローンを検出する．

大規模ソフトウェアを実用的な時間とメモリで解析可能

例えば，10MLOC のソースコードを 68 分（実行環境 Pentium3 650MHz RAM 1GB）で解析可能である [15]．

さまざまなプログラミング言語に対応可能

言語依存部分を取り替えることで，様々なプログラミング言語に対応できる．現在は，C/C++,Java,COBOL/COBOLS,Fortran,Emacs Lisp に対応している．またプレーン

テキストに対しても，分かち書きされた文章として解析可能となっており，未対応の言語に対しても完全一致判定によるコードクローンは検出することができる．

実用的に意味を持たないコードクローンを取り除く

- コードクローンは小さくなればなるほど偶然の一致の可能性が高くなるが，最小一致トークン数を指定することができるため，そのようなコードクローンを検出しないようにできる．
- モジュールの区切りを認識する．

ある程度の違いは吸収可能

- ソースコード中に含まれるユーザ定義名，定数をパラメータ化することで，その違いを吸収できる．
- クラススコープや名前空間による複雑な名前の正規化を行うことで，その違いを吸収できる．
- その他，テーブル初期化コード，可視性キーワード (protected,public,private 等)，コンパウンド・ブロックの中括弧表記等の違いも吸収することができる．

繰り返しコードに対して特別な処理を行う

例えば，コード列中で

X A B C A B C A B C A B C Y

のように 3 つの記号 A,B,C が繰り返していたとき，以下の “*”が付された部分を “繰り返し”であると定義する．

X A B C A B *C *A *B *C *A *B *C Y

これは一見,

X A B C *A *B *C *A *B *C *A *B *C Y

のように定義する方が合理的であるように考えられる．しかしながら，この場合，もしコードクローンとして検出されたコード片が B C A であった場合,

X A [B C *A] [*B *C *A] [*B *C *A] *B *C Y

のように，このコード列中から 3 のコード片 (“[”,“[”で囲まれた部分) が，B C A を各要素とするクローンクラスに含まれる．すると，これら 3 つの各コード片の非繰り返しコードの長さは，左から順に 2,0,0 と判定される．
一方，

X A B C A B *C *A *B *C *A *B *C Y

であれば，もしコードクローンとして検出されたコード片が B C A であっても，

X A [B C A] [B *C *A] [*B *C *A] *B *C Y

となり，非繰り返しコードの長さは，左から順に 3,1,0 と判定される．つまり，少なくとも先頭のコード片に関しては，長さ 3 と判定可能である．もちろんコードクローン片の意味的な先頭になりうる記号が区別可能であれば，こういった問題は起こらない．しかしながら，現実的には人間が見てもすぐにはコードクローン片の最初を判断しかねる場合もある．たとえば，

:
代入文
出力文
代入文
出力文
:

のようなコード列があった場合，2 つ目の代入文は出力文の後始末をしているのか，次の出力に備えた処理なのか判断は難しい．ゆえに，このような定義となっている．

2.3.2 コードクローン検出処理手順

CCFinder のコードクローン検出処理は，以下の 4 ステップで構成されている．

ステップ 1：字句解析

ソースコードをプログラミング言語の文法に沿ってトークン列に変換する．その際，空白とコメントは機能に影響しないので無視される．ファイルが複数の場合には，単一ファイルの解析と同じように処理できるよう，単一のトークン列に連結する．

ステップ 2：変換処理

実用的に意味を持たないコードクローンを取り除くため，若しくはある程度の違いは

吸収するためにトークン列を実用的に意味のあるコードクローンのみを検出するための変換を施す．例えば，変数名，関数名などはすべて同一のユニークなトークンに置換される．

ステップ 3：検出処理

トークン列を比較して，一致した部分トークン列をコードクローンとする．ただし，指定された最小一致トークン以上の長さを持つコードクローンのみが検出される．また，トークン列の比較にはサフィックス木探索 [13] を行うため，線形時間で解析可能となる．

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．

2.3.3 検出例

実際に，CCFinder によってどのようなコードクローンが検出されるのか例を示す．図 2 には，Java で互いに似通った 2 つのメソッドが書かれ，左端には行番号が付されている．ここで，最小一致トークン数を 5 トークンに定め，図 2 のソースコードに対しコードクローン検出を行うと，図 2 中の A1（4 行目-6 行目）と A2（16 行目-17 行目），B1（8 行目-10 行目）と B2（20 行目-22 行目），そして C1（12 行目）と C2（25 行目）がそれぞれクローンペアとして検出される．それぞれのクローンペアの長さは順に 7，18，6 トークンとなっている．見ての通り，A1 と A2 の間，B1 と B2 の間には次のようないくらかの違いが含まれているがコードクローンとして検出可能となっている．

- 名前空間の違い (e.g. “org.apache.regexp.RE” と “RE”).
- 変数名の違い (e.g. “pat” と “exp”).
- 改行とインデントの違い
- 中括弧表記の違い

これらの違いは，2.3.1 節で述べた目的のため，CCFinder のトークン変換処理によって吸収されている．

2.3.4 出力結果

本研究では CCFinder からの出力結果を利用するため，実際に出力されてくる情報がどういったものか簡単に説明する．CCFinder の出力としては，クローンペアの出力と，クローンクラスの出力があるが，本研究では後者を利用する．図 3 にその出力例を示す．

```
    1. static void foo() throws RESyntaxException
    2. {
    3.    String a[] = new String [] {"123,400", "abc"};
A1  4.    org.apache.regexp.RE pat =
A1  5.              new org.apache.regexp.RE("[0-9,]+");
A1  6.    int sum = 0;
    7.    for (int i = 0; i < a.length; ++i)
B1  8.    {
B1  9.       if (pat.match(a[i])){
B1 10.         sum += Sample.parseNumber(pat.getParen(0);}
   11.    }
C1 12.    System.out.println("sum = " + sum);
   13. }
   14. static void goo(String [] a) throws RESyntaxException
   15. {
A2 16.    RE  exp = new RE("[0-9,]+");
A2 17.    int sum = 0;
   18.    int i   = 0;
   19.    while (i< a.length)
B2 20.    {
B2 21.        if (exp.match(a[i]))
B2 22.           sum += parseNumber(exp.getParen(0));
   23.        i++;
   24.    }
C2 25.    System.out.println("sum = " + sum);
   26. }
```

図 2: コードクローン検出例

#begin{file description}...#end{file description}

解析対象となったファイルパスのリスト.

ファイルパスの手前に来る 3 つの数字は順に,「グループ・ファイル番号，行数，トークン数」である．入力の際，解析対象ファイル集合を 1 階層のグループに分類することを可能にするため，グループ番号が存在する.

#begin{syntax error}...#end{syntax error}

言語文法を用いてトークン列に変換している際に発生した構文エラーの位置情報.

ここに書き出された行は，クローン検出の対象外となっている.

#begin{set}...#end{set}

ひとつのクローンクラスがこの間に含まれる.

コードフラグメントの内容は順に「グループ・ファイル番号，開始行番号・カラム番号・トークン番号，終了行番号・カラム番号・トークン番号」である.

```
#begin{file description}
0.0 1475 3429 /home/a/example1.c
0.1 3213 9669 /home/b/example2.c
         :
#end{file description}
         :
#begin{syntax error}
0.1 243,30
         :
#end{syntax error}
         :
#begin{clone}
#begin{set}
0.0 246,1,235 268,2,294
0.5 178,1,213 200,2,272
#end{set}
#begin{set}
0.0 1140,3,2342 1146,2,2378
0.16 143,5,279 149,2,315
0.26 197,5,415 203,2,451
0.55 59,3,29 65,2,65
#end{set}
         :
#end{clone}
```

図 3: CCFinder の出力例　クローンクラス

3　コードクローン情報付加手法

3.1　コードクローン情報の利用方法と問題点

我々は，開発者，保守者，管理者の 3 者の観点から，次のようなコードクローンの分析・利用目的があると考える．ここでいう管理者とは，ソフトウェアの開発保守プロセス全体を管理，監視する者を指す．

開発者的観点

利用目的 1 プログラム記述ヘルプとしての利用

ある処理を記述している際，もしくはバグへの対処を行っている際に，その箇所をサンプルコードや既存ソースコードと比較することにより，類似コード片から処理方法のヒントやバグへの対応策を得られる可能性がある．

利用目的 2 デバッグへの利用

コードクローン検出をバグの位置の特定に直接利用するのは困難であると思われるが,，1 箇所バグが見つかり，修正を行ったあと，同様の修正を行うべき箇所を発見することには利用できる．

保守者的観点

利用目的 3 リファクタリングへの利用

リファクタリングとは “外部から見たときの振る舞いを保ちつつ，理解や修正が簡単になるようにソフトウェアの内部構造を変化させること”であると定義されている [12]．このように，コードクローンになっている部分を 1 つのメソッドや関数にまとめることに利用できる．

利用目的 4 プログラムテキスト修正時のチェック

デバッグや，機能追加などにおける修正を行う際，同様の修正を行うべき箇所の発見をすることができる．

管理者的観点

利用目的 5 設計品質評価への利用

対象ソフトウェアの中にどの程度類似しているものが存在するのかを全体的に確認することで，類似モジュールが数多く存在するかどうか，設計品質が劣化していないかどうかを把握することで，設計の見直しの必要性を知ることができる．

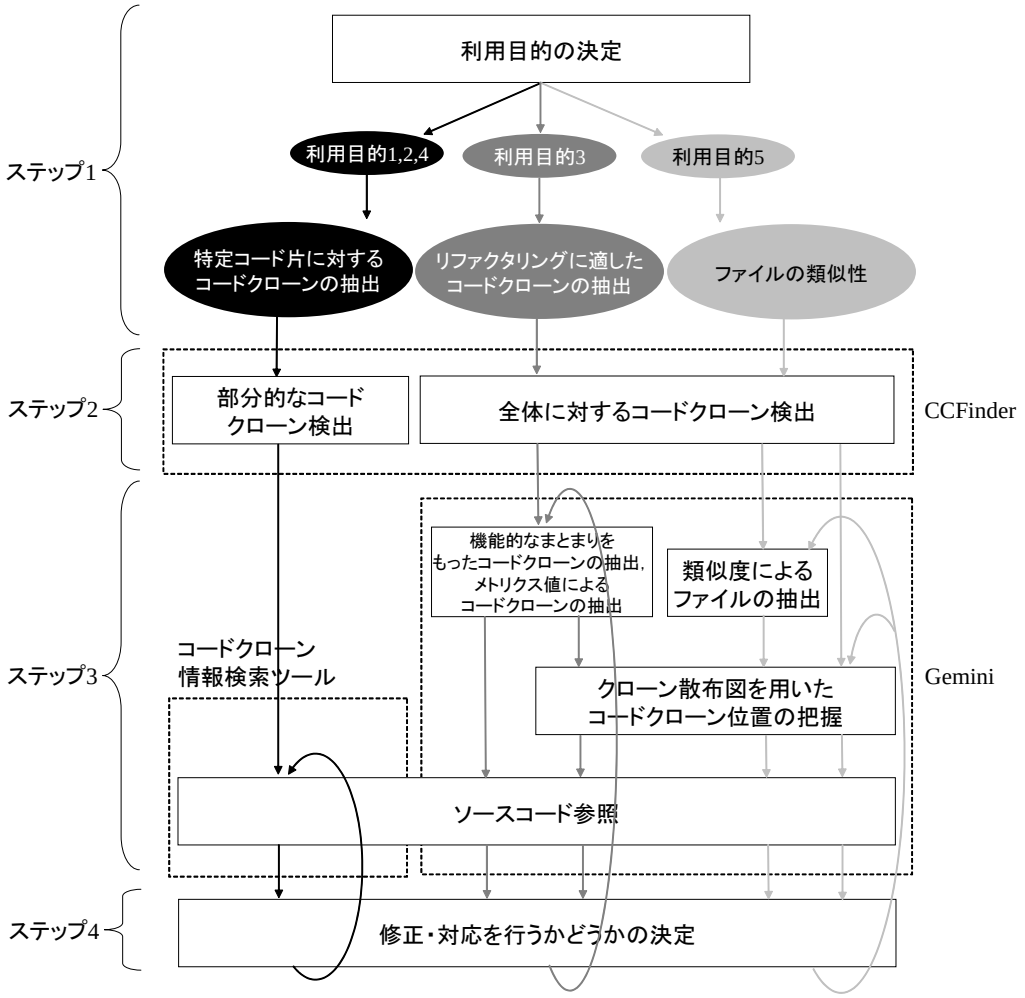

図 4: コードクローン情報利用プロセス

我々が考えるコードクローン情報利用プロセスは図 4 にも示すように，次の 4 つのステップで構成される．

ステップ 1　利用目的の決定

上述した 5 つの利用目的の中から選ぶ．これにより抽出すべきコードクローンが決定する．

ステップ 2　コードクローン検出

CCFinder を用い，コードクローンを検出する．特定コード片に対するコードクローンの抽出を行いたい場合は，そのコード片と分析対象ファイル群との間のコードクローンを検出する．それ以外の場合は，分析対象のファイル内，および分析対象のファイ

ル間のコードクローンを検出する．

ステップ 3　目的のコードクローンを探す

特定のコード片に対するコードクローンの抽出を行う場合は，これまでに “コードクローン情報検出ツール”が開発されており（[16] 参照），利用者が指定したコード片に対するコードクローン情報を得ることができる．それ以外の場合，リファクタリングに適したコードクローンの抽出を行う場合に関しては，これまでに “Gemini”が開発されており（[23][24] 参照），機能的なまとまりをもったコードクローンの抽出および，メトリクスを用いたコードクローンの抽出ができ，クローン散布図などを用いて抽出されたコードクローン位置情報を確認し，ソースコードの参照を行える．

ステップ 4　修正・対応を行うかどうかの判断

ステップ 3 で見つかったコードクローンが利用目的にあったものであるかを判断し，修正等を行う．

問題点として，CCFinder で検出されたコードクローン情報は，ソースコード中での行番号として提示されているため，上述のステップ 4 の修正・対応で，ソースコードに修正を加えるたびに，コードクローン情報とソースコードとの間に行番号のずれが生じてしまう．そのため，複数箇所の修正を続けて行う場合，開発者は次の対処法を行う必要がある．

対処法

検出されたコードクローン情報とソースコードの間で，行番号がずれていることを意識しながら修正箇所を特定し，確認・修正を行う．

ところが，この対処法ではコードの追加・削除など，行数の増減を計算する必要などもあり，作業効率の低下が考えられ，更には，修正箇所を間違えるといった可能性もあると考えられる．

3.2　コードクローン情報付加手法

ここでは，3.1 で述べた，行番号のずれを意識しながら作業することによって修正箇所を間違えたり，作業効率が低下する問題点を解決するために，コードクローンの位置情報をコメントとしてソースコード中に付加する手法を提案する．こうすることで，ソースコード中の行番号として示されたコードクローンの位置情報を，行番号以外の手段で提示することが可能となり，開発者はプログラムの変更時に行番号のずれを意識せずに作業を行うことができる．

手法の詳細は以下のようになる．

手順 1　コメントに利用する ID の決定

コードクローン検出ツール CCFinder から得られた解析結果をもとに，クローンクラス毎に一意に ID を振る．ID は 1 から始まる整数とする．

手順 2　コメントの付加

ソースコード中でクローンクラスに含まれるすべての行に，そのクローンクラスの ID をコメントとして付加する．また，そのクローンクラスの先頭行のコメントに関しては，ID の前後をソースコード中に現れないであろう文字列で挟んでおく．

本手法により，図 4 に示したプロセスのステップ 4 以降は以下のように修正される．

ステップ 4’　修正・機能追加

修正・機能追加すべき箇所を特定し，修正・機能追加を行う．

ステップ 5　コメント部のチェック

修正した箇所の前後の行に付加されているコメント（クローンクラスの ID）を調べる．それが修正箇所が属しているクローンクラスであるので，そのクローンクラスに対して修正の検討を行うべきことがわかる．クローンクラスの ID をもとに，他の修正・機能追加対象がどこに存在するかを調べる．エディタの検索機能でそのクローンクラスの先頭位置を探し出す．

ステップ 6　コードクローンの修正・機能追加

修正・機能追加対象すべてに対して，変更が必要であるか検討し，必要であれば変更する．

ステップ 7　不要コメントの除去

すべての箇所に対して，検討・変更を行った後，不要となったコメントを除去する．

4　コードクローン情報付加ツール

4.1　概要

3.2 で提案した手法をもとに，コードクローン情報をソースコード中にコメントとして付加するツールの試作を行う．

システム構成の概略図を図 5 に示す．ユーザは，GUI 上から CCFinder の出力ファイルを指定する．ツールは CCFinder の出力ファイルから，クローンの位置情報とファイル情報を解析し，それをもとにファイルにコードクローン情報を付加していく．

4.2　機能設計

以降では，図 5 で示した各機能について説明する．

4.2.1　コメント追加機能

コメントを追加する際のコードクローン情報を得るため，GUI 上から，ユーザに対して入力として CCFinder の出力ファイル（クローンクラスの出力）を求める．そのファイルから，コードクローンの位置情報とファイル情報を取得し，それらをリストに登録する．

コメントの付加は，オリジナルのファイルに対してではなく，編集用に作成する一時的なファイルに対して行う．まず，カレントディレクトリ内に新しくディレクトリを作成し，その中に，得られたファイル情報をもとに，ディレクトリ・ファイル階層を再現する．ファイルの内容は,「オリジナルファイルのソースコード」＋「コードクローン情報のコメント」となっている．

コメントにはクローンクラスごとに一意に定めた ID を利用し，コメントはそのクラスの先頭行から末尾行まですべての行に挿入する．こうすることによって，ソースコード上でのクローンの位置を行番号以外の方法で表すことになるため，コードクローン情報とソースコードの間に行番号のずれが生じても正しい位置を示すことができる．実際にコメントを付加した後のソースコードの様子は 4.4 節の図 12 を参照.

4.2.2　一時ファイル編集機能

コメントを追加した後の一時的なファイルを使ってプログラムを確認・編集するため，ファイル名を指定して，エディタを開く．エディタは，ユーザの使い慣れたものを利用できるように emacs,vi,gedit などから選択することができる．デフォルトの設定では emacs となっている．

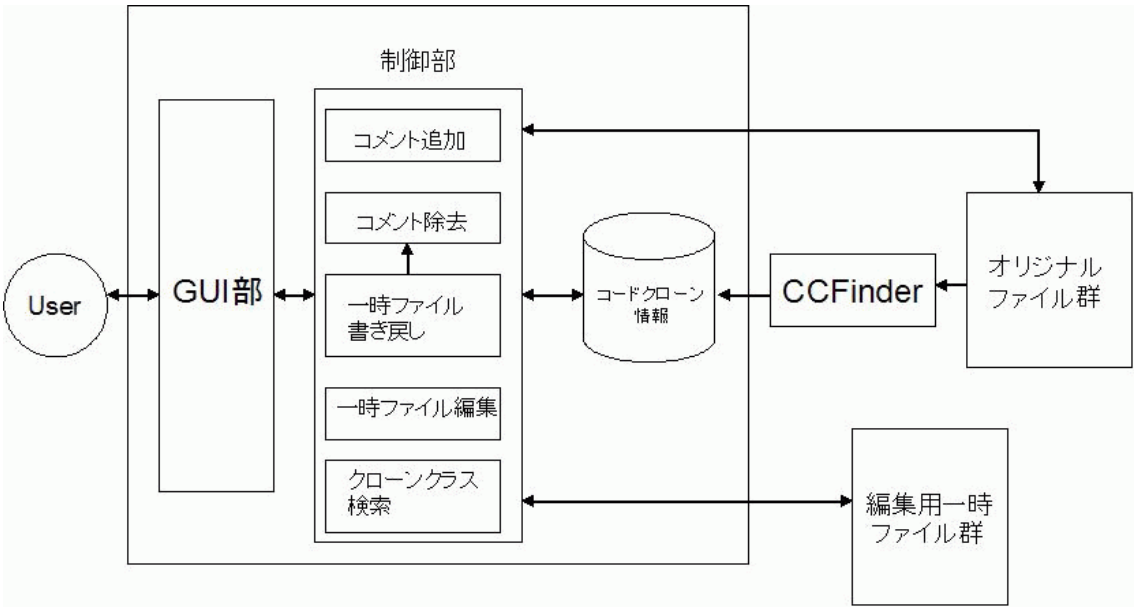

図 5: システム概略図

4.2.3 コメント除去機能

あるクローンクラスに関する検討・修正を終えたとき，そのクローンクラスのコメントは不要になる．このように不要となったコメントを除去する．除去する際には，クローンクラスごとに一意に定められた ID を指定するが，不要になったクローンクラスのコメントが複数個ある場合も考えられるので，その場合は複数個の ID を指定する．こうすることで，コメントを付加したファイル群から，指定した ID を含むコードクローンに関するコメントを，すべて除去する．

また，クローン情報に関するすべてのコメントが不要になった場合は，一括して除去する．

4.2.4 一時ファイル書き戻し機能

修正・機能追加のための編集用の一時的ファイル上で行った変更を，オリジナルのファイル群に反映させるために，付加されたコメントをすべて除去したのち，オリジナルのファイル群に上書きする．この際，上書きしたいファイルをユーザが指定する必要はなく，コメント付加時にリストに登録しておいたファイル情報を利用する．

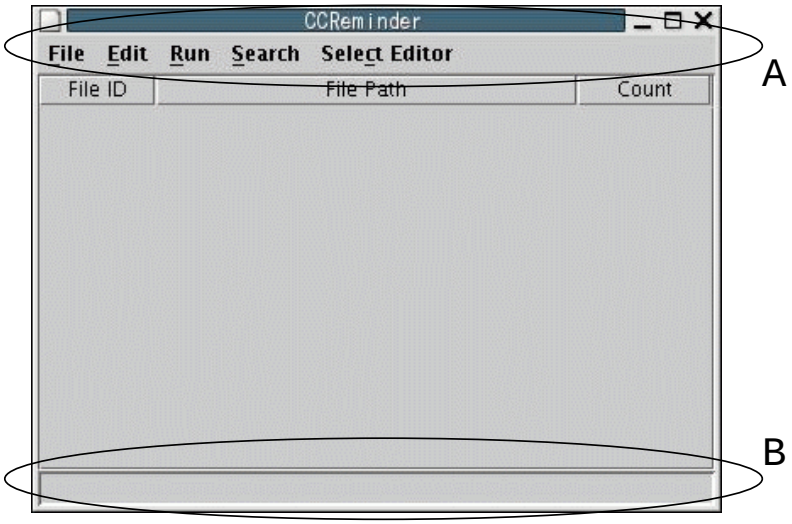

図 6: ツール起動直後

4.2.5 クローンクラス検索機能

修正箇所を特定したとき，修正したいクローンが複数のファイルに分散していることもある．この場合，コードクローン情報を参照する必要がある．この手間を省くため，修正しているクローンクラスがどのファイルに，何箇所含まれているかを知りたい．そこで，ユーザが指定したクローンクラスが含まれるファイル ID，ファイルパス，数を示す．ユーザは示されたファイル，数すべてについて修正の検討を行う必要がある．

4.3 実装

本ツールは Java を実装言語とし，JDK1.4 を用いて実装を行った．コードサイズは約 1500 行となった．本ツールは C,Java などの言語に対応している．

本ツールを起動すると，図 6 のような画面が表示される．図 6 の A で示されるのはメニューバーで，ユーザはここから利用したい機能を選択することができる．また，図 6 の B で示されるステータスバーには，その状況に応じたメッセージが出力される．

以降，本システムの実装，及び使用方法について説明する．

4.3.1 コメント追加機能

メニューから Run → Append Comment を選択する．すると，図 7 のような FileChooser のフォームが表示されるので，CCFinder の出力ファイルを選択し，Run ボタンを押すと，選択した CCFinder の出力ファイルをもとにコメント追加が実行される．実行が終了する

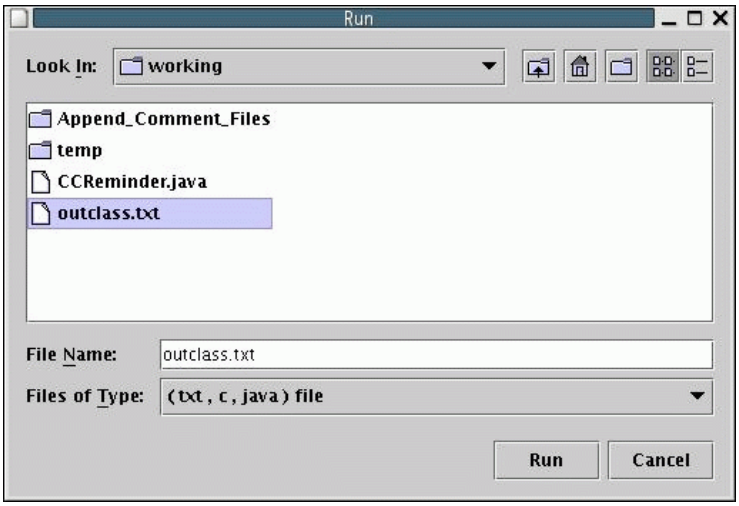

図 7: コメント追加機能選択時

と，ステータスバーに “Append Comment Complete”というメッセージが表示される．また，追加した総ファイル数，総クローンクラス数も表示される．実行に失敗した場合には，“fail”と表示される．

4.3.2 一時ファイル編集機能

メニューから File → Open を選択する．すると，図 7 と同様の FileChooser のフォームが表示されるので，エディタで開いて編集したいファイルを選択し，Open ボタンを押すと，エディタが起動する．エディタの選択は，メニューから Select Editor を選択し，利用したいエディタのラジオボタンにチェックを入れる．デフォルトは emacs とする．

4.3.3 コメント除去機能

メニューから Edit → Delete Comment または，Remove All Comments を選択する．

前者を選択すると図 8 のようなフォームが表示されるので，除去したいコメントのクローンクラス ID を入力する．複数除去したい場合は，数字をカンマで区切って入力する．Delete ボタンを押すと，コメント除去が実行される．実行が終了すると，ステータスバーに “Delete Complete”というメッセージが表示される．失敗した場合には，ステータスバーに “fail”とメッセージが表示される．

後者を選択するとコードクローン情報に関係するすべてのコメントを除去する（もとから存在したコメントは除去されない）．実行が終了すると，ステータスバーに “Remove complete”というメッセージが表示される．失敗した場合には，“fail”と表示される．

図 8: コメント除去機能選択時

図 9: 一時ファイル書き戻し機能選択時

4.3.4 一時ファイル書き戻し機能

メニューから File → OverWrite Original File を選択すると，図 9 のように，本当に上書きをするかどうかの確認を求めるダイアログが表示される．キャンセルする場合には NO ボタンを押す．上書きする場合には OK ボタンを押すと，編集を行った一時的なファイル中のコードクローン情報に関係するコメントをすべて除去したのち，オリジナルのファイルに上書きする．この際，コメント追加時に登録しておいたファイル情報を元に上書きしているので，ユーザがファイルを指定する必要はない．実行が完了すると，ステータスバーに “OverWrite Complete” というメッセージが表示される．失敗した場合には，“fail” と表示される．

4.3.5 クローンクラス検索機能

メニューから Search → SearchID を選択すると，図 10 のようなダイアログが表示される．テキストフィールドに検索したいクローンクラスの ID を入力して，Search ボタンを押すと，コードクローンの位置情報・ファイル情報から解析し，その結果を表として提示する．その様子を図 11 に示す．表には，ユーザが指定したクローンクラスを含むファイル ID，ファイルパスと，そのファイルに指定したクラスのコード片が何箇所あるのかが表示される．

図 10: クローンクラス検索

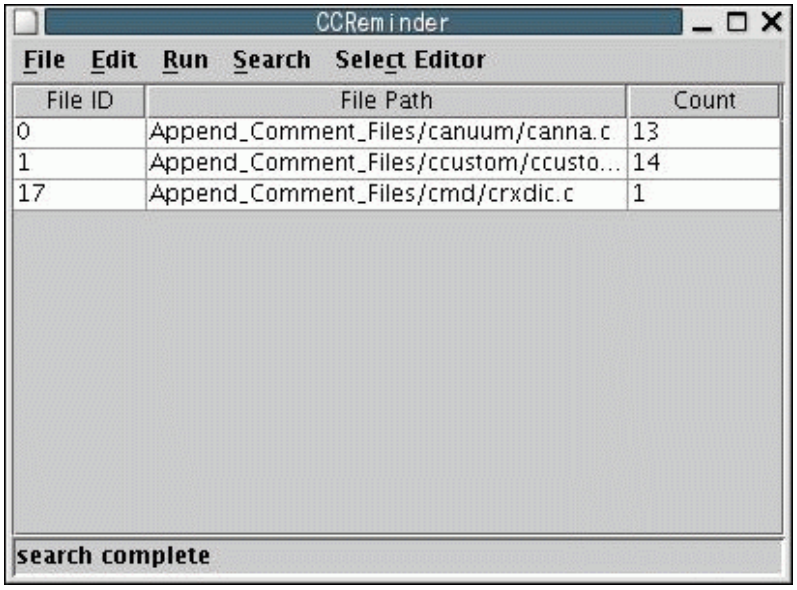

図 11: クローンクラス検索の結果

```
1:    static struct descpack *
2:    searchdesc(hin, tan, yom)
3:    Wchar *hin, **tan, **yom;
4:    { //@$544@
5:      struct descpack *p, **pp, *next = (struct descpack *)0; //@544
6      Wchar *s; //@$538@,@544
7:      int key = 0; //@538,@544
8:     //@538,@544
9:      for (s = hin ; *s ; s++) key += (int)*s; //@538,@544
10:     key = ((unsigned)key & HINSHIBUFINDEXMASK); //@538,@544
11:     for (pp = description + key ; p = *pp ; pp = &(p->next)) { //@538,@544
12:       if (!Wscmp(p->hinshi, hin)) { //@538,@544
13:         return p; //@538
14:       } //@538
15:     } //@538
16:     return (struct descpack *)0;
17:   }
```

図 12: コメント追加後のソースコードの一部

4.4 ツールの使用例

実際に本ツールを用いて，コメントを追加したソースコードを図 12 に示し，その説明をする．例えば，図 12 の 4 行目から 12 行目は，544 番目のクローンクラスに属しているということを示している．また，6 行目から 15 行目はクローンクラス 538 に属している．この例の場合，この関数中には 538，544 という 2 つのクローンクラスがあることになる．もし，この関数中に修正箇所を発見したなら，この 538，544 という 2 つのクローンクラスに対して修正の検討を行う必要があると考えられる．

5　適用実験

5.1　実験概要

オープンソース・ソフトウェア開発サイト SourceForge.jp において開発されている日本語入力システム「かんな」[10] で実際に行われたセキュリティ問題の修正履歴を用いた，擬似的なデバッグを本システムの “修正への適用例”として行う．なお,「かんな」の開発言語は C 言語である．この擬似的なデバッグ作業に対して，本ツールを利用する場合としない場合の作業過程を示し，本ツールの有用性を確認する．

5.2　修正データ

日本語入力システム「かんな」のバージョン 3.6 と 3.6p1 間でのセキュリティ問題の修正において，バッファのオーバフローを検出するコードが挿入された．ソースコードのファイル数及びサイズを表 1 に示す．この修正は，全部で 3 ファイル，30 箇所に対して施されているが，そのうちの 1 ファイル，20 箇所についてはバッファを仲介した処理を行っており，その際に発生する可能性のあるオーバフローの検出を行うコードがこの修正によって追加された．また，この 20 箇所では，変数名の一部が異なる変数に対してほぼ共通した処理を行っている．

具体的には，追加するコード片の一例が，図 13 に示すもので，このような追加処理を図 14 の 2405 行目の直前に挿入するものとする（コードの左にある数字が行番号）．このとき，この 1 箇所に対するコードクローン全てに関して，同様の修正を行うかどうか検討し，必要ならば修正を行うという作業をする．

5.3　作業過程の比較

本ツールを使わずに図 13 で示されるコード片を図 14 で示されるコード中 2405 行目の直前に挿入する場合を考える．他の修正箇所を見つける際ユーザはコードクローン情報を見て探すことになる，このとき，2405 行目に 2 行のコード片を挿入したため，それ以降の行番号は 2 行ずつずれるが，図 15 のようにコードクローン情報には 2 行のずれは反映されていない．

表 1: 処理対象データサイズ

コードサイズ	約 8 万 8 千行
ファイル数	92 ファイル

```
if (Request.type7.datalen != SIZEOFSHORT * 3)
        return( -1 );
```

図 13: 追加コード片の一例

```
2399    static
2400    ProcWideReq7(buf)
2401    BYTE *buf ;
2402    {
2403        ir_debug( Dmsg(10, "ProcWideReq7 start!!\n") );
2404
2405        buf += HEADER_SIZE; Request.type7.context = S2TOS(buf);
2406        buf += SIZEOFSHORT; Request.type7.number = S2TOS(buf);
2407        buf += SIZEOFSHORT; Request.type7.yomilen = (short)S2TOS(buf);
2408       ir_debug( Dmsg(10, "req->context =%d\n", Request.type7.context) );
2409       ir_debug( Dmsg(10, "req->number =%d\n", Request.type7.number) );
2410       ir_debug( Dmsg(10, "req->yomilen =%d\n", Request.type7.yomilen) );
2411
2412      return( 0 ) ;
2413   }
2414
2415    static
2416    ProcWideReq8(buf)
2417    BYTE *buf ;
2418    {
2419       ir_debug( Dmsg(10, "ProcWideReq8 start!!\n") );
2420
2421       buf += HEADER_SIZE; Request.type8.context = S2TOS(buf);
2422       buf += SIZEOFSHORT; Request.type8.curbun = S2TOS(buf);
2423       buf += SIZEOFSHORT; Request.type8.curkouho = S2TOS(buf);
2424       buf += SIZEOFSHORT; Request.type8.size = S2TOS(buf);
2425       ir_debug( Dmsg(10, "req->context =%d\n", Request.type8.context) );
2426       ir_debug( Dmsg(10, "req->curbun =%d\n", Request.type8.curbun) );
2427       ir_debug( Dmsg(10, "req->curkouho =%d\n", Request.type8.curkouho) );
2428       ir_debug( Dmsg(10, "req->size =%d\n", Request.type8.size) );
2429
2430       return( 0 ) ;
2431    }
```

図 14: 修正箇所の典型的なコード片

この行番号のずれへの対処法としては，次のものがある．

対処法

コードクローン情報とソースコードの間で，行番号がずれていることを意識しながら修正箇所を特定し，確認・修正を行う．

この場合，残り 19 箇所の修正を行っていくたびに，行番号のずれが生じる．そのような場合に開発者が行の増減を覚えながら修正箇所を特定する作業するのは効率が悪い．更には，修正箇所を間違えることも考えられる．実際にこの 20 箇所の修正において，合計で 101 行追加されていることが確認されている点から考えても，このことは明らかである．

これに比べて，本ツールを利用してコメントの追加を行った場合を考える．ソースコードは図 16 のようになっている．このとき，修正処理を追加した 2405，2406 行目をはさむ前後の数行には，クローンクラス 1311，1318 のコメントが書かれているのがわかる．つまり，この箇所はクローンクラス 1311 と 1318 に含まれていたこととなり，これらのクラスのコード片に対して同様の修正を行うかどうかの検討を行えばよいと判断できる．実際に図 16 中の 1311 のコメントが付加されている 2422 行目にも同様の修正を行う必要があることが確認できた．

この修正作業を行うには，クローンクラス検索機能を利用してクローンクラス 1311 と 1318 がどのファイルにいくつ含まれているかを調べる．そして，そのファイルをファイルオープン機能で開き，ファイルの検索機能，grep などの検索システムで，例えば，“@$1311@”というようにコメント中のクローンクラス ID を利用して，そのクローンクラスの先頭位置を探して修正を行っていけばよい．

以上の作業の間，行番号のずれを意識する必要はまったくなかったことがわかる．

このことより，複数箇所の修正を行う場合において，本ツールの利用は作業の効率化，正確化を考える上で非常に有用であることがわかる．

5.4 実行時性能の評価

このソースコードに対してコメント追加，全消去，一時ファイル書き戻し，及びクローンクラス検索にかかった処理時間を表 2 に示す．実行環境は，CPU は Pentium4 2.0GHz，主記憶容量は 512MB，OS は FreeBSD である．また，CCFinder の最小一致トークン数は 30 トークンに設定した．

5.5 クローン検出率

本ツールでは，ある特定のコード片に対するコードクローンを探して検討・修正するという情報検索的な利用を行う点から，f 値 [22] による評価も行う．f 値とは完全性と効率性か

```
#begin{set}                                    ← クローンクラス 1311
0.91 2367,1,7452 2375,24,7483
0.91 2383,1,7519 2391,24,7550
0.91 2399,1,7586 2407,24,7617
0.91 2415,1,7657 2423,24,7688      ←← これ以降の行番号は全て 2 行分ずれている
0.91 2433,1,7739 2441,24,7770
0.91 2587,1,8410 2595,24,8441
0.91 2640,1,8619 2648,22,8650
0.91 2657,1,8694 2665,24,8725
#end{set}
          :
#begin{set}                                    ← クローンクラス 1317
0.91 2376,5,7488 2381,2,7518
0.91 2392,5,7555 2397,2,7585
0.91 2408,5,7626 2413,2,7656      ←← これ以降の行番号は全て 2 行分ずれている
0.91 2426,5,7708 2431,2,7738
0.91 2444,5,7790 2449,2,7820
0.91 2554,5,8276 2559,2,8306
0.91 2650,5,8663 2655,2,8693
#end{set}
#begin{set}                                    ← クローンクラス 1318
0.91 2399,1,7586 2407,50,7619
0.91 2587,1,8410 2595,51,8443      ←← これ以降の行番号は全て 2 行分ずれている
0.91 2657,1,8694 2665,51,8727
#end{set}
```

図 15: コードクローン情報

```
2399    static //@$1311@,@$1318@
2400    ProcWideReq7(buf) //@1311,@1318
2401    BYTE *buf ; //@$1311,@1318
2402    { //@1311,@1318
2403        ir_debug( Dmsg(10, "ProcWideReq7 start!!\n") ); //@1311,@1318
2404     //@1311,@1318
2405       if (Request.type7.datalen != SIZEOFSHORT * 3)
2406          return( -1 );
2407        buf += HEADER_SIZE; Request.type7.context = S2TOS(buf); //@1311,@1318
2408        buf += SIZEOFSHORT; Request.type7.number = S2TOS(buf); //@1311,@1318
2409        buf += SIZEOFSHORT; Request.type7.yomilen = (short)S2TOS(buf); //@1311,@1318
2410       ir_debug( Dmsg(10, "req->context =%d\n", Request.type7.context) ); //@$1317@
2411       ir_debug( Dmsg(10, "req->number =%d\n", Request.type7.number) ); //@1317
2412       ir_debug( Dmsg(10, "req->yomilen =%d\n", Request.type7.yomilen) ); //@1317
2413    //@1317
2414      return( 0 ) ; //@1317
2415   } //@1317
2416
2417    static //@$1311@,@$1312@,@$1319@,@$1320@
2418    ProcWideReq8(buf) //@1311,@1312,@1319,@1320
2419    BYTE *buf ; //@1311,@1312,@1319,@1320
2420    { //@1311,@1312,@1319,@1320
2421       ir_debug( Dmsg(10, "ProcWideReq8 start!!\n") );
                                  //@$1307@,@1311,@1312,@1319,@1320
2422    //@1307,@1311,@1312,@1319,@1320
2423       buf += HEADER_SIZE; Request.type8.context = S2TOS(buf);
                                  //@1307,@1311,@1312,@1319,@1320
2424       buf += SIZEOFSHORT; Request.type8.curbun = S2TOS(buf);
                                  //@1307,@1311,@1312,@$1315@,@$1316@,@1319,@1320
2425       buf += SIZEOFSHORT; Request.type8.curkouho = S2TOS(buf);
                                  //@$1304@,@1307,@1311,@1312,@1315,@1316,@1319,@1320
2426       buf += SIZEOFSHORT; Request.type8.size = S2TOS(buf);
                                  //@$1304@,@1307,@1315,@1316,@1319,@1320
2427       ir_debug( Dmsg(10, "req->context =%d\n", Request.type8.context) );
                                  //@$1304@,@$1308@,@1315,@1316,@1320
2428       ir_debug( Dmsg(10, "req->curbun =%d\n", Request.type8.curbun) );
                                  //@$1304@,@1308,@1315,@1316,@$1317@,@1320
2429       ir_debug( Dmsg(10, "req->curkouho =%d\n", Request.type8.curkouho) );
                                  //@1308,@1315,@1316,@1317,@1320
2430       ir_debug( Dmsg(10, "req->size =%d\n", Request.type8.size) );
                                  //@1308,@1317,@1320
2431    //@1317,@1320
2432       return( 0 ) ; //@1317,@1320
2433    } //@1317,@1320
```

図 16: 修正後のコード片

ら検索結果の精度を評価するもので，

$$f\text{ 値} = \frac{2 \times \text{完全性} \times \text{効率性}}{\text{完全性} + \text{効率性}}$$

で求められ，その値が大きいほど情報検索の精度が高いとされる．また，ここで

完全性: 修正が必要な箇所のうち実際に検出された割合

効率性: 検出されたうち実際に修正箇所である割合

と定義する．ここでいう “検出された”とは，“ある修正箇所を特定したときに，その箇所にあるコメント内のクローンクラスの ID をたどって，たどり着ける箇所”とする．

これらの定義を用いて検出された修正箇所の数，f 値を計算した．比較対照として標準的な検索システムである grep をあげる．grep 検索の引数には，バッファ処理時に使われる変数名の共通部分である “Request.type”を用いた．結果は表 3 に示すとおりである．

結果から，本システムの検出した修正箇所を用いた場合，修正しない箇所を検出しないため，grep による検索よりも f 値は大きくなっており精度が高いといえる．また，grep などの検出システムでは，ユーザが明示的に検索対象のコード片を指定する必要があり，結果がユーザの能力に依存すると考えられるが，本システムでは指定する必要がないので，そのような心配は必要ないと期待される．本システムで，完全性を向上させるためには，CCFinder の最小一致トークン数を下げて検出するクローンの数を増やせばよいが，それに伴い，偶然の一致によるクローンなど，修正しない箇所も検出されるため効率性が低下することが考えられる．さらに，本ツールにより付加されるコメントの数が増加し，ソースコードが見づらくなることが考えられる．実際に本適用例において CCFinder の最小一致トークン数を変化させたときの結果を表 4 に示す．本実験では，CCFinder の最小一致トークン数はデフォルトの 30 に設定したが，この結果を見れば 20 トークンにしたときが f 値が最高になることがわかる．しかし，最小一致トークン数は CCFinder を実行する最初の段階で設定しなければならないため，実際には，対象とするソフトウェアの性質などから，よい結果が得られそうな最小一致トークン数を設定するしかない．

表 2: 処理時間

コメント追加	約 5 秒
コメント全消去	約 3 秒
ファイル上書き	約 3 秒
クローンクラス検索	1 秒未満

表 3: 検出結果と f 値

	grep	本ツール
総修正箇所	20 箇所	20 箇所
全検出箇所	61 箇所	20 箇所
検出された修正箇所	19 箇所	15 箇所
完全性	95 %	75 %
効率性	31 %	100 %
f 値	0.50	0.86

表 4: 最小一致トークン数の変化

トークン数	50	40	30	20	10
完全性	13 %	16.5 %	75 %	90 %	100 %
効率性	100 %	100 %	100 %	100 %	5 %未満
f 値	0.23	0.28	0.86	0.95	0.1 未満
コメント数	0.5 個	1 個	4 個	11 個	30 個

6　まとめと今後の課題

本研究では開発・保守工程において，複数箇所のプログラム修正作業を支援することを目的として，コードクローン検出ツール CCFinder を利用したコードクローン情報付加手法の提案とツールの試作を行った．

本研究は，あるコード片に対して修正を行ったあと，類似した他のコードに対する検討・修正を効率的に行うことを目的としている．コードクローン情報をコメントとしてソースコード中に付加することで，修正時に起こるコードクローン情報とソースコード間での行番号のずれを意識せずに作業することができる．

更に，本ツールを日本語入力システム「かんな」の開発において実際に行われたバージョンアップの修正を用いて擬似的なデバッグを行い，その有用性を確認した．

今後の課題としては，

- 利用者の使い慣れたエディタへの対応など，実用性の向上
- さまざまなプログラミング言語への対応
- 本ツールを利用したことによってプログラムの変更がどれだけ容易になったかの定量的，客観的な評価
- 実際のソフトウェア開発・保守現場での適用実験と評価

が考えられる．

謝辞

本研究を通して, 常に適切な御指導およ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致します.

本研究を通して, 常に適切な御指導およ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 楠本 真二 助教授に心から感謝致します.

本研究において, 常に適切な御指導およ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 助手に深く感謝致します.

本研究において, 常に適切な御指導および御助言を頂きました 科学技術振興機構　さきがけ　神谷 年洋 研究員に深く感謝致します.

本研究において, 適切な御指導およ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 博士前期課程 2 年 肥後 芳樹 氏に深く感謝致します.

最後に, その他様々な御指導, 御助言を頂いた大阪大学 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致します.

参考文献

[1] A. Aiken, "A System for Detecting Software Plagiarism (Moss Homepage)", `http://www.cs.berkeley.edu/~aiken/moss.html` [Last visited 1st Feb. 2003].

[2] B.S. Baker, "A Program for Identifying Duplicated Code", *Computing Science and Statistics*, 1992, 24:49-57.

[3] B.S. Baker, "On Finding Duplication and Near-Duplication in Large Software Systems", *Proceedings the 2nd Working Conference on Reverse Engineering*, 1995, 86-95.

[4] B.S. Baker, "Parameterized Duplication in Strings: Algorithms and an Application to Software Maintenance", *SIAM Journal on Computing*, 1997, 26(5):1343-1362.

[5] M. Balazinska , E. Merlo, M. Dagenais, B. Lagüe, and K. Kontogiannis, "Advanced Clone-Analysis to Support Object-Oriented System Refactoring", *Proceedings the 7th Working Conference on Reverse Engineering*, 2000, 98-107.

[6] M. Balazinska , E. Merlo, M. Dagenais, B. Lagüe, and K. Kontogiannis, "Measuring Clone Based Reengineering Opportunities", *Proceedings 6th IEEE International Symposium on Software Metrics*, 1999, 292-303.

[7] M. Balazinska , E. Merlo, M. Dagenais, B. Lagüe, and K. Kontogiannis, "Partial redesign of Java software systems based on clone analysis", *Proceedings 6th IEEE International Working Conference on Reverse Engineering*, 1999, 326-336.

[8] I.D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone Detection Using Abstract Syntax Trees", *Proceedings IEEE International Conference on Software Maintenance-1998*, 1998, 368-377.

[9] E. Burd, and J. Bailey, "Evaluating Clone Detection Tools for Use during Preventative Maintenance", *Proceedings 2nd IEEE International Workshop on Source Code Analysis and Manipulation*, 2002, 36-43.

[10] Source Forge, 日本語入力システム「かんな」`http://canna.sourceforge.jp/`

[11] S. Ducasse, M. Rieger, and S. Demeyer, "A Language Independent Approach for Detecting Duplicated Code", *Proceedings IEEE International Conference on Software Maintenance-1999*, 1999, 109-118.

[12] M.Fowler, *Refactoring: improving the design of existing code*, Addison-Wesley, 1999.

[13] D.Gusfield, *Algorithms on Strings, Trees, And Sequences, Cambridge University Press*, 1997.

[14] IEEE std 1219: "Standard for software maintenance", 1997.

[15] 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**, 2001, 18(5):47-54.

[16] 泉田聡介，植田泰士，神谷年洋，楠本真二，井上克郎, "ソフトウェア保守のための類似コード検索ツール", 電子情報通信学会論文誌，Vol.J86-D-I，No.12，pp.906-908，December 2003.

[17] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: A multilinguistic token-based code clone detection system for large scale source code", *IEEE Transactions on Software Engineering*, 2002, 28(7):654-670.

[18] R.Komondoor, and S.Horwitz, "Using slicing to identify duplication in source code", *Proceedings 8th International Symposium on Static Analysis*, 2001.

[19] J. Krinke, "Identifying Similar Code with Program Dependence Graphs", *Proceedings 8th Working Conference on Reverse Engineering*, 2001, 562-584.

[20] J. Mayland, C. Leblanc, and E. Merlo, "Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics", *Proceedings IEEE International Conference on Software Maintenance-1996*, 1996, 244-253.

[21] L. Prechelt, G. Malpohl, and M. Philippsen, "Finding plagiarisms among a set of programs with JPlag", *resubmitted to Journal of Universal Computer Science*, 2001. `http://www.ipd.uka.de/~prechelt/Biblio/#jplag` [Last visited 1 Feb. 2002].

[22] 徳永健伸, "情報検索と言語処理", 東京大学出版会, 2002.

[23] 植田泰士，神谷年洋，楠本真二，井上克郎, "開発保守支援を目指したコードクローン分析環境", 電子情報通信学会論文誌 D-I，Vol.86-D-I，No.12，pp863-871，December 2003.

[24] Y. Ueda, Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue, "Gemini: Code Clone Analysis Tool", *Proceedings 1st International Symposium on Empirical Software Engineering*, 2002, 2:31-32.