特別研究報告

題目

コードクローン検出ツールを用いた
ソースコード分析システムの試作とプログラミング演習への適用

指導教官

井上 克郎 教授

報告者

植田 泰士

平成 13 年 2 月 22 日

大阪大学 基礎工学部 情報科学科

平成 12 年度 特別研究報告

コードクローン検出ツールを用いた
ソースコード分析システムの試作とプログラミング演習への適用

植田 泰士

内容梗概

近年, プログラムの保守作業は, 開発されるシステムの大規模化に伴い複雑かつ困難な作業となってきている. 保守性を阻害する一つの要因として , Copy-and-Paste Programming(切貼プログラミング) が指摘されている . 切貼プログラミングとは , 適当なコード片をコピーし修正することで , 機能追加を行う手法である . コピーされたコード片にフォールトが含まれていた場合には , それに関連する箇所を全て修正する必要がある . しかし , 大規模なプログラムの場合 , 関連箇所を全て検出することは困難である .

我々の研究グループでは , ソースコード中の類似コード片 (コードクローン) を検出するツール (CCFinder) の開発を行ってきている . 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コードフラグメント対の位置情報の集合である為, 検出結果を直感的に理解することは困難であり, その利用にはコードクローン情報の視覚化, および即座にソースコードへフィードバックしていく手段が求められていた.

本研究では, コードクローンに基づいた効率的なソースコード分析手法を検討するため, CCFinder の検出結果を視覚化したソースコード分析システムの試作を行った. 本システムは, コードクローンの散布図や, クローンクラス (同値類) に関するメトリクスグラフから, 警戒すべきコードクローン集合を特定し, 対応したソースコードを対で表示できる. また, 本システムを, 本学科で実施されたプログラミング演習 (コンパイラ開発) で作成された 76 名の学生のソースコードに対して適用し, 有効性を確認した.

主な用語

プログラム保守
コードクローン
ソフトウェアメトリクス

目 次

1 まえがき **3**

2 準備 **5**

2.1 コードクローン検出ツール CCFinder 5

2.1.1 概要 5

2.1.2 処理概要および特徴 6

2.1.3 出力結果 8

2.2 クローンクラスに関するソフトウェアメトリクス 10

2.2.1 $RAD(C)$ 10

2.2.2 $LEN(C)$ 11

2.2.3 $POP(C)$ 11

2.2.4 $DFL(C)$ 11

3 ソースコード分析システム **12**

3.1 コンセプト 12

3.2 概要設計 14

3.3 機能設計 15

3.3.1 コードクローン検出部 15

3.3.2 クローンペア管理部 16

3.3.3 クローンクラス管理部 20

3.3.4 ソースコード管理部 22

3.4 実装 23

3.4.1 コードクローン検出 24

3.4.2 クローン散布図による分析 24

3.4.3 クローンメトリクスグラフによる分析 27

4 適用 **30**

4.1 実行時性能 30

4.2 プログラミング演習への適用 31

4.2.1 概要 31

4.2.2 結果とその分析 32

5 まとめと今後の課題 **39**

謝辞 40

参考文献 41

1 まえがき

ソフトウェア保守は，出荷後のソフトウェアに対して施す，欠陥の修正，性能などの属性の改善，または変更された環境に適合させるための変更と定義されている [11]．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．例えば，Lam は，ソフトウェア全ライフサイクルのコストの 66%が保守に費やされていると報告している [14]．保守作業の効率を高めることは，ソフトウェア工学における重要な課題の一つとなっている．

保守作業を難しいものとしている要因も幾つか指摘されている．その中の一つに，Copy-and-Paste Programming(切貼プログラミング) がある．切貼プログラミングとは，既存のプログラムに機能を追加する時に，そのプログラム中のコード片をコピーし修正することで，機能追加を行う手法である．この手法は，手軽に機能追加が行えるという利点があるが，コピーされたコード片にフォールトが含まれていた場合，フォールトがプログラム中に分散してしまうという問題点がある．ソフトウェア保守においては，このようなコード片に対する修正を行う際には，類似コード片を全て抽出し，漏れなく修正を施すことが必要となってくる．

我々の研究グループでは，一つあるいは複数のプログラムから，類似しているコード片 (コードクローン) を検出するツール CCFinder[8] を開発してきている．CCFinder では，処理対象プログラムをトークンの列に分解し，ユーザ定義名をあるユニークなトークンに置きかえてから比較し, 類似コード片を抽出する．全てのユーザ定義名を同じトークンに置きかえることで，単に変数名や関数名を変更しただけのコード片も抽出することが可能である．CCFinder は，大規模プログラム開発の保守支援や教育環境におけるプログラム評価支援を目的として開発されている．しかし，コードクローンの検出結果はコードフラグメント対の位置情報の集合である為, 特に大規模プログラム対する検出結果を直感的に理解することは困難であり, その利用にはコードクローン情報の視覚化, および即座にソースコードへフィードバックしていく手段が求められる.

本研究では,CCFinder の検出結果を利用したソースコード分析システムの試作を行うとともに, 効率的なソースコード分析手法を検討し, それを大学におけるプログラミング演習に適用していくなかで, その有効性を確認することを目的とする. 本システムは, コードクローンの散布図や, クローンクラス (同値類) に関するソフトウェアメトリクス値グラフから, 警戒すべきコードクローン集合を特定し, 対応したコードフラグメント集合をコードクローン対で表示できるため, コードクローンに基づくデバッグ支援, 再構築支援が可能となる.

以降，2 節では，コードクローン検出ツール CCFinder, およびクローンクラスに関するソフトウェアメトリクスについて説明する. 3 節では, 本システムの設計, 実装に関して説明す

る. さらに,4 節においては, 本システムの性能, および大学におけるプログラミング演習で開発されたプログラムに本システムを適用し, その結果についての分析と考察を行う . 最後に 5 節では , まとめと今後の課題について述べる .

2 準備

2.1 コードクローン検出ツール CCFinder

2.1.1 概要

コードクローンとは, 一つもしくは複数のソースコード中に含まれる類似したコード片のことである. その対応した 1 対のコード片のことをクローンペア, それらのコード片の同値類の集合をクローンクラスと呼ぶ [8]. そして CCFinder は, そのクローンペアの位置情報を取得するものである.

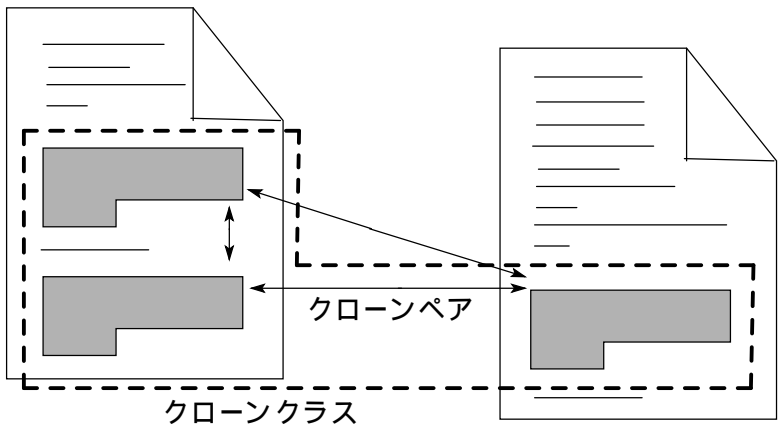


図 1: コードクローン

そもそもコードクローン検出の背景は, 新しい機能を追加したりバグを修正した際の変更によって , コードの一致箇所が生じるという観察結果に基づいている [1]. そのコピーされた箇所はより肥大に, 複雑になり, そして保守を困難にしていく事が長く知られてきている. 特に, あるコピー箇所でバグが見つかっている場合, 見つかったその箇所でのバグは修正されるが, 同じ複写元からコピーされた他の箇所では修正されない. 一般にコピー・修正による機能拡張は, 元のソースコードを変更しなくてするため, 既存のコードにバグを作り込む可能性はないという理由により, 特に多人数によるソフトウェア開発では多用されることがある. そこでバグの含まれたコードフラグメントへ漏れなく修正を施すことができるよう, コードクローンを全て抽出することのできるツール CCFinder を現在我々の研究グループで開発してきているわけである .

2.1.2 処理概要および特徴

CCFinderが行う処理の概要を説明する (図 2 参照). まず最初に, 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, 複数ソースも 1 つのトークン列に連結される. また, 空白とコメントは機能に影響しないので無視される. さらに, そのトークン列を実用的に意味のあるクローンのみを検出するための変換を施す. 例えば, パラメータ置き換え (名前が異なっても等価にする) などである. 次に, そのトークン列を比較し, ソースコードが一致しているかどうか調べる. トークン列の比較には suffix-trre という木構造を用いた事実上 $O(n)$ のアルゴリズム[5] を採用している[1]が, 詳細については文献 [3] を参照されたい. 最後に出力整形処理を行い, オリジナルソース上での行番号 (およびカラム番号) が出力される.

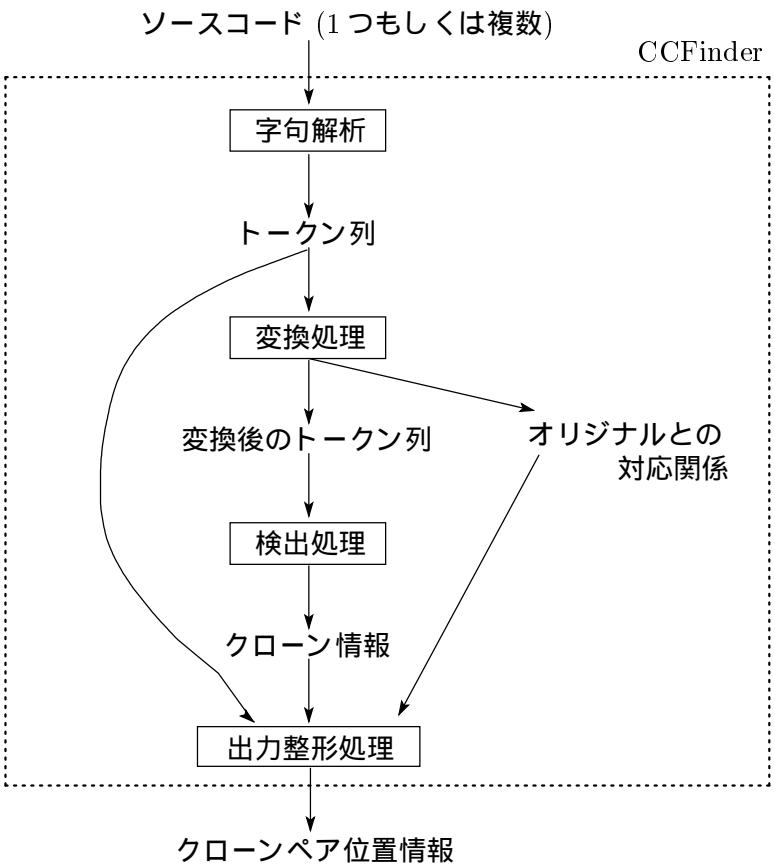


図 2: CCFinder の処理手順

また, 主な特徴として以下のことがあげられる.

- 数百万行規模のシステムにも実用時間で解析可能

[1] 実際の CCFinder では $O(n \log n)$

  - Linux(160 万行:2095 ファイル),FreeBSD(130 万行:2906 ファイル) の計 290 万行の処理に 40 分 [8]

- 言語依存部分を取り替えることで，さまざまなプログラミング言語に対応
  - 現在 C/C++,Java,COBOL, 平文テキストに対応.
    (但し, 同時には 1 言語のみでの解析)
- 主記憶容量に対し,1 つの suffix-tree で構成するにはソースコードが大きすぎる場合, 部分集合に分割して suffix-tree を構成する
- 実用的に意味のあるクローンのみを検出するための工夫が施されている
  - 最小一致トークン数が指定できる
  - 定数を区別しない
  - 共有ルーチンとして再構築するのが困難なクローンは検出しない
    * テーブル初期化部分を取り除く
    * モジュールの区切りを認識する
  - 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
    * パッケージネーム削除
    * 実引数削除
  - 繰り返しコードには特別なクローン認識方法

ここでいう繰り返しコードとは単純に代入文の連続や switch 構文により生成されるクローンが連続しているコードのことである. 上述の, 繰り返しコードに施されている特別なクローン認識方法に関して, 簡単な例を挙げて説明する (図 3 参照). 簡単のため,break 文は除いている.

仮に, 最小一致トークン数を 5 トークンに指定し, コードフラグメント A と B に対しクローン検出処理を行うと, 本来, 最長一致で検出されるので [3], コードフラグメント A の 3 行目から 6 行目 (以下「A:3-6」と表記) と A:5-8 のクローンペアおよび B:5-6 と A:3-4,A:5-6,A:7-8 の 3 クローンペア検出されるはずである. しかし経験的に, その検出方法では, このようなコードフラグメントが検出対象に複数分散し, 繰り返し回数が相当数になってくると, さほど興味の惹かれないクローンペア (自己クロスクローン等) が膨大に現れる. こうなると本来見つけなければならないクローンが, 繰り返しコードによるクローン集合に埋もれてしまう.

そこで CCFinder では, 予備検出として繰り返しコードをあらかじめ検出し, 先頭の 1 周期分より後ろの繰り返し部分は全て検出処理前に切り捨てられる (コードフラグメント A').

コードフラグメント A

```
1:switch(a)
2:{
3:    case CONST_A0:
4:        a = 0;
5:    case CONST_A1:
6:        a = 1;
7:    case CONST_A2:
8:        a = 2;
9:}
```

コードフラグメント B

```
1:switch(b)
2:{
3:    case CONST_B0:
4:        println("0");
5:    case CONST_B1:
6:        b = 1;
7:}
```

コードフラグメント A'

```
1:switch(b)
2:{
3:    case CONST_A0:
4:        a = 0;
9:}
```

図 3: 簡単な繰り返しコード例 (Java)

しかし, 単純に 1 周期目以降が切り捨てられるのではなく行番号を保持したまま切り捨てが行われるので, A:3-9 と B:5-7 として検出結果に現れる. これは, 直感的にコードフラグメント A と B の switch 構文後半がほぼ同じ形式であると感じられる感覚と一致している.

2.1.3 出力結果

ソースコード分析システムでは,CCFinder からの検出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 (図 4 参照).

```
#version: ccfinder 3.1
#langspec: JAVA
#option: -b 30,1
#option: -k +
#option: -r abcdfikmnprsv
#option: -c wfg
#begin{file description}
0.0 52  C:\Gemini.java
0.1 94  C:\GeneralManager.java
0.2 237 C:\MDI.java
1.0 7   C:\CCFEventListener.java
1.1 116 C:\CCFinderManager.java
1.2 695 C:\CCFinderOptionFrame.java
             :
             :
#end{file description}
#begin{syntax error}
1.1 56,13
#end{syntax error}
#begin{clone}
0.1 53,9   63,13   1.10 542,9  553,13 35
0.1 53,9   63,13   1.10 624,9  633,13 35
0.2 124,9  152,31  0.2  154,9  216,51 42
0.2 124,9  152,31  1.10 194,9  225,30 42
0.2 126,9  152,31  1.10 185,9  204,34 37
0.2 153,14 211,9   1.10 207,9  242,5  31
0.2 153,14 216,51  1.10 193,9  225,30 44
0.2 172,9  216,51  1.10 185,9  204,34 37
1.2 195,13 210,64  1.2  219,13 232,49 50
1.2 235,13 250,37  1.2  276,13 291,41 51
             :
             :
#end{clone}
```

図 4: CCFinder 出力結果例

- `#begin{file description}...#end{file description}`
  解析対象となったファイルパスのリストである. ファイルパスの手前にくる 2 つの数字は「グループ番号. グループ内ファイル番号, 行数」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が可能にするため, グループ番号というものが存在する.

- `#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[2]の位置情報であり, 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{clone}...#end{clone}`
  クローンペアの位置情報が, グループ番号, ファイル番号等を用いて出力される. 一行に一クローンペアで, そのクローンペアが「コードフラグメント 1, コードフラグメント 2」とコードフラグメントの対で並ぶ. 一コードフラグメントの内容は「グループ番号. グループ内ファイル番号, 開始行番号. カラム番号, 終了行番号. カラム番号」となっている. カラム番号とはその行の中での何文字目かということを表す.

[2]ソースコードが EUC 等の CCFidner の対応していない文字コードである場合にもここに構文エラーとして現れる.

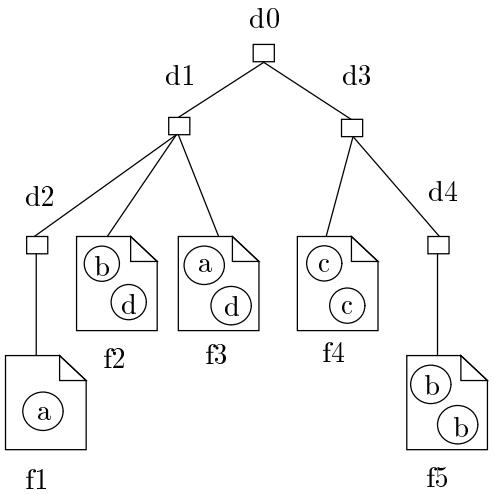
2.2 クローンクラスに関するソフトウェアメトリクス

では, ソースコード分析システムにおいて用いるソフトウェアメトリクスについて, あらかじめ定義を行う. 以下の 4 つのソフトウェアメトリクスはクローンクラスに関して定義されており, 文献 [8] で詳しく述べられているが, 多少定義が異なることに注意されたい.

2.2.1 $RAD(C)$

$RAD(C)$(Radius of clone-class) は, クローンクラス C 内のコードフラグメントが含まれるファイル集合 F が, ファイルシステムの中でディレクトリ構造的にどれだけ分散しているかを表すメトリクスである. ディレクトリ構造を表す木構造を考え,F 内の全てのファイルに共通の親ノードの中で最も下位層に存在するノードまでの距離を求め,C 内でのその最大値を $RAD(C)$ として定義する.

簡単な例を挙げる (図 5 参照). 各々葉となっているのがファイルであり, その他のノードはディレクトリを表す. ファイル中に含まれる a,b,c,d は各コードフラグメントの属するクローンクラスの識別子を表す. クローンクラス a について上の条件に当てはまるディレクトリノードは d1 であり,d1 までの距離が最も長いのは f1 であるから $RAD(a)$ は 2 である. クローンクラス b についても同様に 3 となる. しかし, クローンクラス c のように, 単一ファイル内で閉じているクラスについては 0 と定義される. また , クローンクラス d のように同一ディレクトリ内のファイル集合内で閉じている場合は 1 となる.



クローンクラス C	RAD(C)
a	2
b	3
c	0
d	1

図 5: クローンクラスの分散例

$RAD(C)$ の値が大きいならば, クローンクラス C 内のクローンは , システム内において, 広範に広がっているため, バグの作り込まれたクローンに修正を施すのはより困難であると考えられる.

2.2.2 $LEN(C)$

$LEN(C)$(Length) は, クローンクラス C 内の 1 コードフラグメントのサイズである. 文献 [8] では, クローンクラス C 内の最大 LOC [3]として定義されているが, 本論文においては, トークン数として再定義する.

2.2.3 $POP(C)$

$POP(C)$(Population of clone-class) は, クローンクラス C 内のコードフラグメント単位の要素数である. $POP(C)$ が高いということは, より多くの箇所にクローンが分散していることになる.

2.2.4 $DFL(C)$

$DFL(C)$(Deflation by clone-class) とは, クローンクラス C を再構築した場合に, どれだけコードサイズが減少するかというメトリクスである.$LEN(C)$ 同様, 文献 [8] では, 減少分の LOC として定義されているが, 本論文においてはトークン数として再定義する. ここでの再構築とは, クローンクラス C 内のコードフラグメント集合を新たな 1 サブルーチンに整理し, そのサブルーチンを各箇所で呼び出すという最も単純な場合を考えている. まず再構築対象となる総トークン数は $LEN(C) \times POP(C)$ で求めることができる. そして, 再構築対象が削除された後, 新たに追加されるコードはサブルーチン実装コードと, サブルーチンコール用コードであるが, ルーチンサイズは $LEN(C)$ トークン, コール用コードは全て併せて $5 \times POP(C)$ トークンとなる. 呼び出し回数の $POP(C)$ に 5 をかけているのは,1 回の呼び出しにかかるトークン数はサブルーチン名, (, 引数,), ; の 5 トークンとみなしているためである.

従って $DFL(C)$ は以下の式によって定義することができる.

$$
\begin{aligned}
DFL(C) &= (\text{再構築対象コードサイズ}) - (\text{再構築後コードサイズ}) \\
&= (LEN(C) \times POP(C)) - (LEN(C) + 5 \times POP(C)) \\
&= (LEN(C) - 5) \times (POP(C) - 1) - 5
\end{aligned}
$$

[3]LOC(Lines Of Code):ソースコードの行数

3 ソースコード分析システム

3.1 コンセプト

既にこれまで関連研究として, クローン位置情報を視覚化するツールがいくつか開発されている. 下で述べるクローン散布図というのは図 6 のような図のことである. 詳細は 3.3.2 節で述べる.

- DOTPLOT [6]
  単語の一致の分布図を作成するツール. しかし, 自動的なクローン検出は行わない.
- DUP [1]
  クローン散布図を生成できるが, それに対する操作を行うことはできない.
- DUPLOC [12][13]
  クローン散布図を生成する. その図上での拡大, スクロール機能を備える. ソースコードをブラウズする機能として, クローン散布図から指定したコードフラグメントを見ることができる. また, 指定したコードフラグメントを含む, 全てのクローンペアを見せないようにもできる.

図 6: クローン散布図の例

いずれのツールも, クローン位置情報の視覚化にクローン散布図を採用している. この視覚化手法はクローンがどこにあるのかが一目で分かり, DUPLOC にみられるように, ソースコードとのインタラクティブな操作への応用ができる非常に有効な手法であると考えられる.

しかしながら, クローン位置情報をそのままプロットした散布図とソースコードの間にリンクを貼ったとしても, 大規模プログラムにおいては, 膨大なクローン情報の中からどの部分に注目していいのか判断するのは困難な作業である. そして, 散布図からだけでは 1 節で

述べたような, 一箇所にフォールトが見つかった為, そのコードフラグメントとクローンの関係にあるコード片を抽出してこなければならない場合に非常に手間がかかる. 従って, 警戒すべき, もしくは探したいクローンを探す際の負担をできる限り減らす工夫が必要となる. また, 散布図の見た目とソースコードだけで分析を行うのには限界があるため, 統計的または多角的な分析を行う必要もある.

そこで本研究では, 次のようなコンセプトをもって本システムの試作を行った.

コンセプト 1: クローン分析を見据えたクローン位置情報の効果的な表示
コンセプト 2: クローン位置情報から得られる付加情報により分析に多角性をもたせる

コンセプト 1 については主に, クローン散布図 (3.3.2 節) に持たせ, コンセプト 2 についてはクローンクラスツリー (3.3.3 節), クローンメトリクスグラフ (3.3.3 節) に持たせている.

3.2 概要設計

システム構成の概略を図 7 に示す. 本システムは CCFinder を内部的に実行させ, そこから得られた結果を用いて分析を行うものである.

図上では基本的に, 処理は上から下へ流れていく. ユーザは, まず GUI 上から,CCFinder の実行に関する諸設定を行い,CCFinder を起動させ解析を行う. 解析が終了すると, クローンペア管理部, クローンクラス管理部の GUI に結果が表示されるので, ユーザはその上で分析を行うことになる[4]. 重要なのは, クローンペア管理部, クローンクラス管理部が持つ GUI を通してユーザがクローンを選択できる点であり, その部分に効果的な表示, 及び選択方法を工夫していく必要がある.

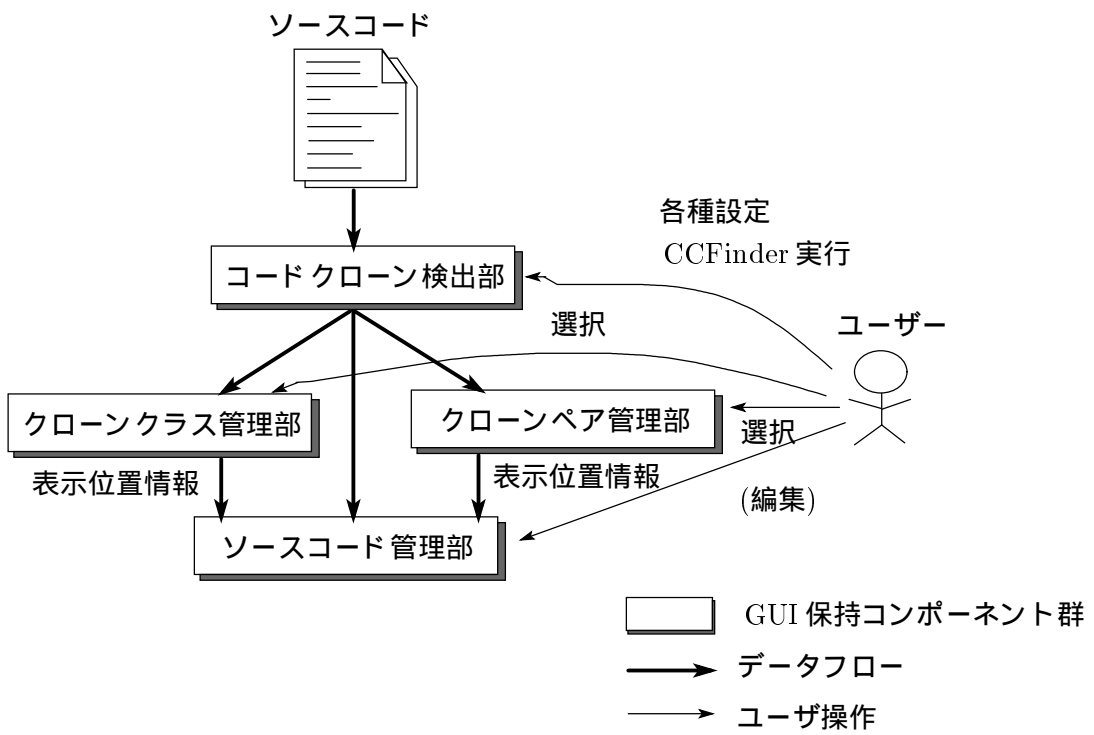


図 7: システム構成図 (概略)

[4]本研究ではクローンの分析自体に主眼を置くため, ソースコード編集時のインクリメンタルな動作は想定しないものとする.

3.3 機能設計

以降では図 7 上で GUI を保持しているコンポーネント群毎にその構成と機能について述べる.

3.3.1 コードクローン検出部

CCFinder を実行させるものとして必要なのは,CCFinder 実行時オプションの設定を行うコンポーネント, 解析対象ファイルリストを作成するコンポーネント, 実行制御コンポーネント, CCFinder からの出力状況の表示コンポーネントであるが, 中でも重要性の高い,「オプション設定フレーム」「解析対象設定フレーム」の 2 つの機能について述べる.

オプション設定フレーム

CCFinder が既に備えているオプション機能を利用するためのものであるが, CCFinder の主なオプションとして次のものがある.

- 解析対象言語の指定 (C/C++,Java,COBOL, 平文テキスト)
- 最小一致トークン数
- 検出するクローンペア間の関係
- 解析結果出力ファイルの指定
- 使用メモリ容量指定

検出するクローンペア間の関係で設定できる関係は, “同一ファイル内”, “同一グループ内の相異なるファイル間”, “相異なるグループ間”の 3 つのオンオフを各々設定できる. また, システム外部で CCFinder を実行させるため, これらの他に CCFinder の実行ファイルへのパスの指定が必要となる. 以上のオプションについて適正な設定を行える機能を提供する.

解析対象設定フレーム

CCFinder は,2.1.3 節で述べたように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が可能である. したがって, この分類分け操作を GUI としてユーザーに分かりやすく表示することが必要になるので 2 階層ツリー (グループ識別子, ファイルパスの 2 つ) として構成する. 機能としてはそのツリーに対するグループの追加, 削除, および指定グループへのファイルの追加, 削除となる. また, この解析対象が決定した時点で,CCFinder を実行することができるので, 本フレームに CCFinder 起動の機能を備える.

3.3.2 クローンペア管理部

ここでは, コードクローン検出部によって蓄えられたクローンペアの位置情報を取得し, クローン散布図, クローンペアリスト管理を行う.

クローン散布図

ここでいうクローン散布図というのは図 8 のような図のことである. 座標の原点は左上角にとってあり, 水平軸, 垂直軸には同じように解析対象ファイルが 1 行目から順に並ぶ. 図では, 簡単の為, 各ファイルを 3 トークンずつにしている. つまり各軸に並ぶ「a,b,a,... 」というのはトークンとして解釈されたい. 黒丸プロットは, その座標の水平軸, 垂直軸のトークン内容が一致していることを表す. 当然左上から右下への対角線上では自己比較が行われるため, 必ず一本の線が引かれ, その対角線に対して線対称な分布状態となる. (実際のシステムでの描画単位は行単位であり, かつプロットではなく直線として描画する.CCFinder からの出力結果にはカラム番号が入っているのでトークン単位にすることも可能であるが, 大規模プログラムを対象としているため大した違いもなく, 表示時間の兼ね合いを考え行単位となっている. 従って, 対角線に平行でない直線も存在する.)

図 8: プロット例

コンセプト 1 を踏まえ, クローンの表示以外, この散布図に持たせる機能として次のことが挙げられる.

- ソースコードブラウズ機能
  - 散布図上で選択を行えるようにし, その選択をソースコード管理部の GUI に表示させる. その際, マウスクリック選択であれば, 特定領域に含まれるもの全て選択したい場合に非常に時間がかかる. したがって, マウスドラッグによって描く選択枠というものを用いて領域選択方式にする.
- ズーム機能, スクロール機能
  - 解析対象ファイル総サイズが大きくなってくると, 一画面で表示するには限界がある. 詳細な分析が行えるよう, 指定領域の拡大ができる機能をもたせる.
    また, 分析を行う立場から考えると, システム内の特定のファイル (グループ) にのみ着目したい場合という要求がある. 従って, 単純ズーム機能に加え, 表示画面を指定ファイル (グループ) にフィットさせ, 後は水平方向もしくは, 垂直方向にスクロールするだけでそのファイル (グループ) について調べることができるフィットズーム機能をもたせる.
- ソート機能
  - 座標軸上に並ぶのファイルの配置順を効率よく分析が行えるよう並び換える機能. これについては後述.
- クローンを持たないファイルの表示, 非表示
  - 全くコードクローンを持たないファイルというのは, 基本的にはコードクローン分析の対象外であり, 散布図上に余分なスペースを生み出す. 必要な情報がより分かりやすく表示できるよう, そういったファイルを散布図上から見せなくする機能をもたせる.
- ファイル区切り, グループ区切りの表示, 非表示機能
  - 最も基本的な機能である. ソートが行われていない状態では, 座標軸上では, グループ番号の昇順に, かつグループ内ではグループ内番号の昇順に並ぶ. このファイルもしくはグループの区切れ位置を散布図上で認識できるための, マス目状の区切り線である. しかしながら, ディスプレイ解像度に対しファイル数が十分多い場合, この区切れ線でクローンの分布自体が見づらくなる為, 非表示機能を持たせる.

ソート機能は, コンセプト 1 を実現する本システムの最も重要な機能である. 本来, 座標軸上のファイルの並びはファイルを読み込んだ順番である, しかしながら, そのように並べただけでは場合によって図 8 のようにクローンが目立った偏りがなく広範囲に広がってしまう場合がある. そのような場合, どの部分に着目し分析を行えばいいのか判断が難しく. 全て分析し尽くすには相当の労力が必要となる. そこで, このソート機能を実現することにより, その負担はかなり軽減できると考える.

以下ソート実現方法のアイデアについて説明する.

できるだけクローンが分散しにくい散布図にするためには, 座標軸上でのファイルの並びに関して, できる限り類似しているものを近くに配置することで解決できる. 例えば, 図 8 においては, 仮にファイル $f1$ の位置が決定すれば, その次に配置するのは, まだ配置場所の決まっていないファイル集合の中から,$f1$ に対して最も類似率の高いファイル $f6$ を配置する. $f6$ に対しても同じことを繰り返していけば隣り合うもの同士が類似しているため, 類似率の高いファイル集合は自然と集中していく (図 9 参照).

図 9: 図 8 の再配置例

また類似率は, ファイル $f1$ に対するファイル $f2$ のコードクローンによるコードカバー率 (以下ファイル間クローンカバレッジと呼ぶ) として, 次のように定義する.$f1$ 内に含まれるコードクローンの中で,$f2$ とクローンペアとなっている $f1$ 側のコードフラグメント集合 CF を抽出し,CF 内のコードフラグメントが ト合計で何行あるか (重複部分は一度しか数えない) を求め,$f1$ のトータルサイズで割った値をファイル間クローンカバレッジとする.

次に, 先頭に配置するファイルを決めなければならない. 一概にどれが良いかという判断基準が難しいが, 本研究では経験的にクローンの散らばりは, 出来る限り左上角に集合させたいという方針を持つ. その場合,2 番目に来るファイルが先頭のファイルに対し高くはない類似率で並んでしまっては (2 番目のファイルの選び方から他のファイルはさらに低い類似率となっている), 左上角にクローンが集まりにくい. つまり, 先頭のファイルはできるだけ, 次のファイルが高い類似率を持ってこれるよう誘導する必要がある. 従って, ファイルの組み合わせでなく 1 ファイルに対し 1 つ定まる値を定義しなければならない. そこで, ファイル f に対する解析対象全ファイルのコードクローンによるコードカバー率 (以下, 単にクローンカバレッジと呼ぶ) を用いる. 簡単にいえばファイル f 内がどれだけクローンで埋まっているかという割合である. ファイル間クローンカバレッジ同様, ファイル f 内に存在するクローンのコードフラグメント (重複部分は一度しか数えない) 全てのトータルサイズをファイル f のトータルサイズで割ればよい. このクローンカバレッジが最も高いものを先頭のファイルとすれば,2 番目に来るファイルと高い類似率を持つ可能性が高い.

しかしながら, このようにして決まった先頭ファイルから順に配置を決定していっても, 途中で, 最も高いファイル間クローンカバレッジでも値が 0 という場合が当然でてくる. こうなると, ファイル間クローンカバレッジが高いファイル集合が次にいつ現れるか分からない. 従って,0 である場合は, 先頭のファイルを決めたのと同じ方法で, もう一度, 次のファイルが高い類似率を持ってこれるよう誘導する.

以上ファイルのソート方法について述べてきた. しかし, 実際は散布図上はグループ区切りが存在する. グループ区切りの意味合いによっては, グループ区切りを越えてソートしてしまうと対応関係が見えなくなり余計に分析しにくくなる (同一ソフトウェアのバージョン間比較等). 従って, このグループ区切りを越えてソートする場合と, グループ区切りを残したままソートしたい場合の 2 通りの機能を提供する. また, グループ自体の並びに対してもファイル同様の (グループ間) クローンカバレッジを用いてソートできる機能をもたせる.

クローンペアリストテーブル

本コンポーネントはクローン散布図の補助的な機能を果たす. 実際に, クローンペアの位置情報は全て散布図に反映され, 選択も可能であるが, その散布図を見るだけでは, 具体的に何行目から何行目までの何トークンのコードフラグメントなのかということが把握できない. わざわざソースコード上で確認するのも効率がよくない. 従って, クローンペアの位置情報をそのままリスト化し, 表示する必要がある. 本コポーネントは, クローンペア位置情報をテキスト表示のテーブルで構成する.

散布図にはソースコードブラウズ機能があると述べたが, 実際は散布図上での選択は, 常にこのクローンペアリストを介して行う. つまり, 本コンポーネントとクローン散布図の選択状態というのは常に同期をとる. ユーザーが散布図上でクローンペア集合を選択すれば, このテーブルも即座にそのクローンペア集合の先頭の位置までスクロールされる. 逆に, テーブル側から選択を行えば, 散布図上で選択されるのはもちろんのこと, ズーム状態にある場合はその選択位置が即座に表示される.

3.3.3 クローンクラス管理部

ここでは, コンセプト 2 に基づき, コードクローン検出部によって蓄えられたクローンペアの位置情報を取得し, それらをクローンクラス (2.1.1 節) に分類, および各クローンクラスについてクローンメトリクス (2.2 節) を算出し, クローン情報に付加情報を追加することで, 分析に多角性を持たせることを目的とする. これらの付加情報を表示し, 分析手段を提供するコンポーネントは, 以下の「クローンクラスツリー」,「クローンメトリクスグラフ」である.

クローンメトリクスグラフ

用いるクローンメトリクスは 2.2 節で述べた,$RAD(C)$,$LEN(C)$,$POP(C)$,$DFL(C)$ の 4 つである. この各々の値をグラフ化するにあたり, 本研究では, 多次元並行座標表現 [9] を用いる. 多次元並行座標表現は, 大量のデータから価値のある情報を導き出すデータマイニングを行う際に有効である. しかしながら, 本研究で用いる平行軸には $RAD(C)$,$LEN(C)$,$POP(C)$, $DFL(C)$ のわずか 4 種類であり, なおかつ $DFL(C)$ は定義より $LEN(C)$,$POP(C)$ に比例することが既に分かっている為, 実質 3 種類のデータしか用いない. これでは多次元並行座標の本来の力は発揮することはできないのだが, あくまで, この 4 つの値の分布を 2 次元の散布図で表すならば 6 通りの組み合わせができ, 分析に手間がかかるであろうこと, また今後のメトリクスの拡張性 (構文に基づいたクローンクラスの分類等) を考慮した上で, 多次元並行座標表現を選択した. また, 当然 1 クローンクラスにつき 1 本の折れ線が描画される.

クローンメトリクス値の表示以外, このグラフに持たせる機能として次のことが挙げられる.

- フィルタリング機能とソースコードブラウズ機能
  - グラフ上で選択 (フィルタリング) を行い, その選択されたコードフラグメントをソースコード管理部の GUI に表示させる. ここでの選択は, 後述のクローンクラスツリーからの選択が反映された場合, もしくは, フィルタリング機能により, フィルタ内に収まったデータが選択状態にある場合の 2 通りを指す. フィルタは GUI 上で各軸の値範囲の選択を行えるものである.3.4 節で具体的に説明を行う.
- グラデーション機能
  - 多次元並行座標では, データ (クローンクラス) が多くなってくると, 値の分布状態が今ひとつ分からなくなってくる. フィルタリング機能により選択領域を変化させることで, 調べることも可能ではあるが, その手間を省く機能として, 指定軸の値が大きくなっているものにはより赤く, 小さくなっているものにはより黄緑になるようグラデーションをかけることができる機能をもたせる. 但し, 選択状態にあるデータのみの中でのグラデーションとなるため, 値が小さなものばかりの場合であっても, その中で一番値の大きなデータが赤く染まる.
- 軸反転機能
  - グラデーション機能同様, データ (クローンクラス) が多くなり, 値の分布状態が今ひとつ分からなくなってきた場合のための補助機能である. 例えば, ある座標軸で値が大きめのクラス集合が, それと隣接する座標軸の値ではどれも小さめの値へ分布していた場合を考える. その場合, その集合の折れ線データは, どれもあまり変わらない角度で斜めへ伸びるため, それらの微妙な違いが分からなくなる. (やはりフィルタリング機能を用いれば特定できるが). これを解消する為, 軸の反転機能を備える. 軸が反転すれば, このような場合, 折れ線の伸びる角度に差がつきやすく区別が容易になる (図 10 参照).

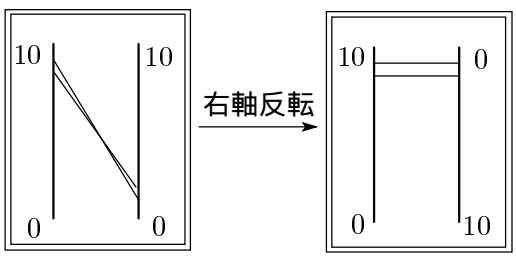


図 10: 軸反転例

クローンクラスツリー

クローンメトリクスグラフでは, そのクローンクラスの要素に関する情報が欠けている. つまり, クローンクラスにどのコードフラグメントが含まれているのかが分からない.

従って, 本コンポーネントはどのコードフラグメントが同値であるのかそのグループ分け内容を示す. クラス分類は 2 階層 (クラス識別子, コードフラグメント識別子) であるので, 解析対象設定フレームと同じようにツリーで構成する. クラス識別子とはいっても名前を決まっていないので, 内部的に割り振られたユニークな番号を用いる. 実際, 上では述べなかったが, クローンメトリクスグラフにもその識別子の軸が表示される. また, 内部的に割り振られた番号で上から順に並んでも意味がないので, $POP(C)$ を基準にソートする (他のメトリクスでソートすることも可能).

また, このツリー上での選択はクローンメトリクスグラフに反映され, グラフ上でのフィルタリング機能が停止する. 逆に, グラフ上での選択もこちら側に反映される. やはり散布図等と同様ソースコードブラウズ機能は, 常にこのクローンクラスツリーを介して行う.

3.3.4 ソースコード管理部

ここでは, 先に述べたクローンペアリストテーブル, クローンクラスツリーから表示すべきコードフラグメント集合の指定を受け, そのコードフラグメント集合を表示するのに必要なソースコード全体をファイルシステムからロードし, 指定コードフラグメントの強調表示を行う.

クローンペアリストテーブルからは, クローンペアとして指定されるので, その対応関係が一度に分かるよう上下に分けて表示する. クローンクラスツリーからは, 単にコードフラグメント集合として指定されるので, 上下には分けずに表示する.

複数ファイルにまたぐ, クローンペア集合が指定された場合は, 上の表示部分にはその全てのファイルがそれぞれ 1 つのタブとして用意され, 下の表示部分には, 上の表示部分で選択されたファイルに関連するクローンペアをもつファイル集合 (指定されたクローンペア集合が含まれるもののみのファイル集合) が表示される.

強調表示はコードフラグメントの背景色を強調する. この強調背景色は, 半透明にして, 複数強調部分が重なりあっても, その重なり具合が分かるようにする. つまり, 重なった部分の背景はより濃い色で表示される.

3.4 実装

本システムの実装言語は Java であり,JDK1.3[4] を用いて開発した. 全体の構成を図 7 に示す. 実装対象外となるのは,CCFider のみである.

図における矢印は, 基本的に矢印の根本のコンポーネントの内容が変化すれば (命令を送れば), 指されているコンポーネントにも影響が及ぶという意味である. また, 二重枠で囲まれたものが,GUI としてユーザが実際に操作することができるコンポーネントして描かれている.

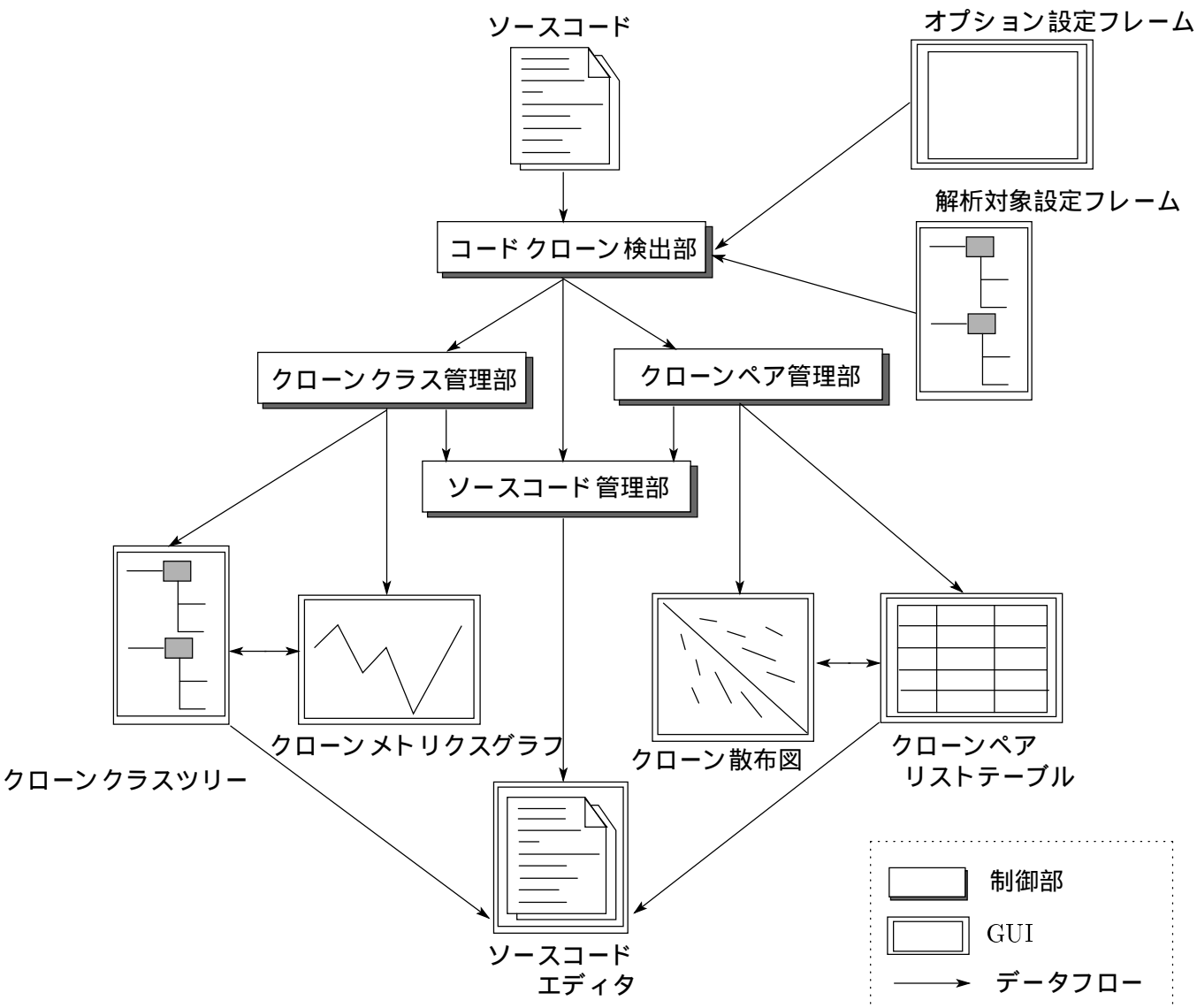


図 11: システム構成図全体

では前節で述べた機能を持たせたシステムの実装状態を実行画面と対応させながら説明する. 実行例では本システム自体のソースコード (Java, 約 1 万行, 52 ファイル) を用いる.

3.4.1 コードクローン検出

システムを起動させると, 図 12 のようなメニューだけの MDI フレームが表示される. コードクローンの検出を行うには, メニューの「File」の「New Detection」を選択する. そうすると, メインフレーム内に解析対象設定フレーム (図 14 左下) が表示される. 解析対象を指定するには「Add Target」ボタンを押すとファイルチューザーが現れるので, 解析したいファイル (複数選択可, ディレクトリ指定不可) 指定する. またグループ分けを行うには,「New Group」ボタンを押して新規グループを作成し, そのグループを選択した状態で「Add Target」を行う. また, その解析対象設定は「Save List」ボタンによりファイルに保存できる. 保存しておけば次の解析時には「Load List」ボタンでファイルの設定を読み込むことができる. 解析対象の設定が完了し,「OK」ボタンを押すことにより CCFinder での解析がスタートする (図 13).

また, オプションの設定は, 解析対象設定フレームの「OK」ボタンを押す前にメニューの「Settings」の「CCFinder Option」を選択するとオプション設定フレーム (図 14 右上) が現れるので, 各種オプション (解析対象言語等) を変更し,「OK」もしくは「Apply」を押して設定する.

3.4.2 クローン散布図による分析

次に, クローン検出処理が終了すると, 図 15 のような画面になる. 向かって左側に表示されているのが, クローンペアテーブル. 右側に表示されているのがクローン散布図全体図 (デフォルトではファイル, グループ区切り線は非表示) となる. ファイル区切りが薄い線 (灰色), グループ区切りが濃い線 (黒線) で描かれている. この散布図上で右クリックをすると, ポップアップメニューが現れる. このポップアップメニューから 3.3.2 節で述べた機能が全て実行可能となる. このクローン散布図を用いての分析手順を示す.

- 探索対象の定まっていない場合の手順
  1. 散布図に類似率に基づいたソートをかける
     (但し, グループ分けの意味合いに応じソート方法の種類の使い分けが必要)
  2. 類似したファイル集合が連続して並び, 比較的原点まわりに集中するので, クローンペア集合が密集している領域等の興味対象を見つける
  3. 領域指定ズームを繰り返し, その領域をより詳細に見る

図 12: 起動画面

図 13: コードクローン検出処理経過表示

図 14: コードクローン検出設定

4. ある程度クローンペア集合を絞り込むことが出来れば, 左ドラッグにより領域選択枠でそのクローンペアを囲い込む
5. 散布図上で選択クローンペアが赤く囲まれ, クローンペアリストテーブルでもその結果が反映される
6. その選択した状態で右側の「 SourceCode 」というタブを選択すると, 図 16 のように, 選択状態のクローンペア集合が強調表示されている.(タブ選択直後では, 選択状態にあるクローンペアの中で, クローンペアリスト上で一番上にあるクローンペアの位置が表示されている)
7. ソースコードがどういったものであるか順に確認し分析を行う

- 特定領域 (フォールトが含まれていることが発覚したクローンクラスなど) に対する分析手順
  1. 前節までは述べなかったが, ファイルリストテーブル (ファイルパスとそのグループ番号, グループ内ファイル番号の対応が参照することが出来る) というものがあるので, そのテーブルを参照し, 探したい特定領域のファイルの番号を知る
  2. 散布図上でマウスカーソルを動かすとそのカーソル位置のファイルパスとグループ番号, グループ内ファイル番号が散布図上部に表示されるので, ファイル区切り (グループ区切り) の表示を ON にして, 目的のファイル位置を探し出す
  3. 散布図上の対象ファイル (グループ) 領域の上で右クリックを行い, フィットズーム機能により表示サイズを調整
  4. スクロールバーを水平方向 (垂直方向に) に動かし, クローンペアの存在を確認する
  5. "探索対象の定まっていない場合"と同様に, クローンペア集合の選択 , ソースコードの参照を行う

その他の機能については詳細は割愛する. ソートの実際の性能については 4 節において確認を行う.

3.4.3 クローンメトリクスグラフによる分析

クローン検出処理の終了後, 左側で「Clone Class」, 右側で「Clone Metrics Graph」というタブを選択すると, 図 17 のようにクローンクラスツリーとクローンメトリクスグラフが現れる. このクローンクラスツリー上でコードフラグメント集合 (ツリーの各葉) を選択すると, そのコードフラグメントが含まれるクローンクラス集合の折れ線がクローンメトリクスグラフ上で赤く選択され, そのクローンクラスの各メトリクス値が分かる (グラフ上なので概数までであるが). そこで右側の「Source Code」タブを選択すると, 選択状態のコードフラグメント集合が全て強調表示されたソースコードが表示される (選択状態のコードフラグメント集合のクローンクラス集合内要素全てが表示されているのではない).

逆に, クローンメトリクスグラフからは, フィルタリングを行うことにより, クローンクラスツリー上でのクローンクラス集合の選択を行うことが可能となる. フィルタリングを行うには, クローンメトリクスグラフのすぐ下にある「Filtering Mode : OFF」と描かれた領域へマウスカーソルを移動すると矢印カーソルから手の形をしたカーソルへ変化するので, クリックする. そうすると,「Filtering Mode : ON」になり, その直前までのクローンクラスの選択は全てキャンセルされ, 薄紫のフィルタがグラフ全体に張られる. このフィルタ内に折れ線が完全に収まっているクローンクラスのみが選択状態になる. その選択状態はクローンクラスツリーにも反映される. 初期状態ではグラフ全体にフィルタが張られている為, 全クローンクラスが選択状態になる. この状態から各クローンメトリクス値による選択範囲を行うには, 各軸付近のフィルタの端へマウスカーソルを移動すると, やはり手の形をしたカーソルへ変化するので, 左ドラッグによりフィルタの大きさを部分的に狭くすることができ, その変化したフィルタの外部へ一部でも出ているクローンクラスの選択状態はキャンセルされる. 選択状態を元に戻したければ同じ操作を行い, フィルタの形を元に戻せばよい. このようにして, あるメトリクス値が高いクローンクラスだけの選択, あるメトリクス値は高いが, その他のメトリクス値は低いクローンクラスの選択などのような選択が可能となる.

また, フィルタリングを行っている状態でクローンクラスツリーの選択状態を変化させると, グラフ上のフィルタは消え自動的に「Filtering Mode : OFF」になる. しかし, もう一度フィルタを表示させるとそれまでの状態は保持されているため元に戻すことは可能である.

その他の 3.3.3 で述べた機能は, グラフ上での右クリックによるポップアップメニューにより全て実行可能であるが, クローン散布図同様, 詳細は割愛する.

図 15: クローン散布図とクローンペアリスト

図 16: クローン散布図 (クローンペアリスト) からソースコードの参照

図 17: クローンメトリクスグラフとクローンクラスツリー

図 18: クローンメトリクスグラフ (クローンクラスツリー) からソースコードの参照

4 適用

4.1 実行時性能

プログラミング演習へ適用を行う前に, 前節で実装したシステムの実行時性能を確認する. この確認には JDK1.3 のソースコードを用いた. ソースコードサイズ等は表 1 に示す.

表 1: 処理対象

コードサイズ	約 58 万行
ファイル数	約 1900 ファイル
クローンペア数	約 8000
クローンクラス数	約 2500

このソースコードを処理するのにかかった処理時間を表 2 に示す. 実行環境として,CPU は Celeron466MHz, 主記憶容量は 128MB,OS は Windows2000,VM は JDK1.3 付属 VM である. またコードクローン検出には最小一致トークン数として 30 トークンを設定した.

表 2: 処理時間

処理内容	処理時間
起動	約 20 秒
コードクローン解析	約 10 分
解析結果の読み込みと結果表示	約 1 分
散布図ソート	1 秒以内
散布図上クローンペア選択	1 秒以内
クローンメトリクスグラフのフィルタを OFF から ON に切り替え (全部選択)	約 3 秒

ソート処理は, 正確には, 解析結果を読み込んで結果の表示を行う前処理の段階であらかじめソート後の順番を全て計算している. また, 上の表に示していないこととして, 選択クローンペア集合のソースコード表示の時間が挙げられるが, これは表示すべきソースコード数 (サイズ) による. 現段階この環境下において, 同時にある程度以上の数のファイルを表示しようとすると JavaVM 上で Out Of Memmory Error が発生することが分かっている. ある程度以上のファイルが表示されようとした場合, 選択をキャンセルする, もしくは, メモリ状態を

管理した上での表示などの改善が望まれる. その他の結果は, 多少ストレスが感じられる部分もあるが, 実用上問題のない処理時間であると考えられる.

4.2 プログラミング演習への適用

GUI システムの定量的な有効性評価は困難である為, 本研究では, 以下の適用実験を行い, 本システムの有効性を確認した.

4.2.1 概要

大阪大学のカリキュラムである情報科学演習 D において作成されたプログラムに対し, 本システムを適用した. またソースコードは C 言語で書かれたものである.

課題内容は「Pascal 風言語 (Pascal 言語のサブセット) で記述されたプログラムをアセンブラ言語 CASL で記述されたプログラムに翻訳 (変換) するコンパイラを作成する」である. 作成にあたり, 学生には指導書が渡され, 指導書にはプログラムの仕様が掲載されている. また別の講義ではコンパイラの作成法について学習し, その講義で用いられる教科書には, サンプルプログラムが掲載されている. 開発手順は, 課題 1 として構文チェッカ (Parser), 課題 2 として意味チェッカ (Checker), 課題 3 としてコンパイラ (SPC) の順に行われる. 課題 2,3 においては, 課題 1,2 のプログラムを拡張, 再利用することで実現されることが望まれる. 他人のプログラムをコピーする事は禁止されているが, 学生同士で相談しあったり, 協力する事は禁止されていない.

収集したソースコードのサンプル数は計 76 人分 (約 38 万行) である. これは Parser,Checker,SPC の全てが揃っていたもののみを対象としている.

4.2.2 結果とその分析

以下に示す結果は, 最小一致トークン数を 30 トークンに設定してコードクローン検出を行ったものである.

結果 1:ソートの有効性確認

3.3.2 節で述べたソート機能が, どの程度有効であるか確認を行う.

ここでは,Parser プログラムのソースコードに対するコードクローン検出結果を例に用いて説明する. また, 収集したソースコードはある年度とある年度の 2 年分のものであるため, 図 19 に示した図は. グループ区切りを開発年度にし, 各グループ内では, 各個人の名前の辞書順にファイル集合を並べている (1 個人が単一ファイルに対応しているとは限らない).

図を見るとやはりこれでは, クローンペアの分布状態は散布図全体に広がり, 点在箇所それぞれを何回にも分けて分析しなければならないことが分かる. そこで, グループ (開発年度) の区切りを保ったままグループ内でソートを行った. すると 30 箇所近くあったクローンペア密集領域が, わずか 5,6 箇所に整理された (図 20 参照). 図 19 では分かり辛かったが, 類似ファイル集合が年度越しに派生していることが容易に認識できる.

図 19: ソート前の年度区切り散布図 (Parser)

また, これらの類似ファイル集合に含まれる人数は,Parser,Checker,SPC へと拡張されるにつれ, 次第に減少していくことが確認できた (12 人→ 6 人→ 3 人). つまり, 課題初期段階においては, 課題理解の困難さから, ある程度の協力, 相談等が行われたかもしれないが, 拡

張の際には各個人が独自の手法を採用していった様子が伺える.

　このようにソートは類似ファイル集合を区分, 判別していく上で非常に有効であることが確認できた.

図 20: ソート後の年度区切り散布図 (Parser)

結果 2 : 再利用性

1 グループを 1 個人とし, グループ内では全課題のソースコードを含めてコードクローンを検出した場合, 全員の再利用性を容易に知ることができる. 図 21 は, グループ内において課題順に Parser,Checker,SPC が配置され, グループソート完了後の散布図である (ファイル区切りは非表示).

図 21: 個人区切り散布図 (Parser+Checker+SPC)

対角線上に並ぶ正方形の各領域は各個人内での比較なので, その正方形内の課題間領域に

再利用性が現れてくることになる. 拡大図が, 図 22 と図 23 であるが, それぞれ, 課題 1,2 間での再利用性の低い例と高い例となっている.

全員の各個人内散布図を分析したところ, ほぼ 8 割が図 22 のように課題 1,2 間での再利用性がかなり低いことが確認された (平均約 7.5%). しかしながら, 課題 2,3 間ではほぼ全員にある程度の再利用性がみられた (平均約 30.3%). つまりは, 課題初期段階での課題全体の理解度を高める必要性があると言える.

また,1 グループを 1 バージョンに対応させれば, 同一ソフトウェアのバージョン間比較において, 再利用性を視覚的に判断可能 (グループ間比較により) であろう.

図 22: 課題 1,2 間での再利用性の低い例 (左 (上) から順に Parser,Checker,SPC)

結果 **3** : メトリクスグラフからの特徴のあるコードフラグメントの抽出
以下は, 再利用性を分析した際と同じ解析対象 (グループ分け) を設定している.

- **DFL** 値分析

  DFL 値が非常に高いクローンクラスを分析したところ, Parser で作り込まれた膨大なコードクローンを Checker 拡張時に的確に 1 モジュール内に再構築した例と, それを残したまま拡張している例を発見できた.

  図 24 を見れば,Parser 後半部分に大きな黒い長方形が存在し, コードクローンが何回も繰り返されている (間には異なるコードフラグメントが挟まれている) のが分かる. しかしながら,Parser と Checker の比較領域を見れば, その長方形が一本の線に変わっ

図 23: 課題 1,2 間での再利用性が高い例 (左 (上) から順に Parser,Checker,SPC)

ているのが分かる. 実際にソースコードを見ても, 一つのルーチンとして置き換えられており, 機能拡張であるのにも関わらず 100 行のコードサイズ減少が確認できた.

逆に, 図 24 では,Parser で作り込まれたコードクローンがほとんど再構築されず, コードクローン間の距離が拡大し, 後々のプログラム内に分散していく様子がみられる. この場合, 保守性が下がっているかと予測し, Parser から Checker,Checker から SPC への拡張時の平均増加コード量と比較したが, 大した差は現れてはいなかった. 正確に拡張に要した時間, コンパイル回数等を平均値と比較すれば何らかの違いが確認できるかもしれない. 本研究では既に, 開発が終了した時点での分析であるので未確認である.

- **RAD,POP** 値分析

  この場合,RAD 値が高いということは, 複数人物で使用されたコードフラグメントであり, POP 値が高いということは, 多数回使用されているコードフラグメントであるので, この値が共に高いクローンクラスは, 最も多くの人に最も多く使われたコードフラグメントである可能性がある.

  また実際のソースコードを参照したところ,Pascal 風言語における「変数宣言の並び」もしくは「仮パラメータの並び」の構文解析ルーチンであった.「変数宣言の並び」と

図 24: コードクローンに対し的確に再構築を行ってから拡張した例 (左 (上) から順に Parser,Checker,SPC)

図 25: コードクローンを残したまま拡張した例 (左 (上) から順に Parser,Checker,SPC)

「仮パラメータの並び」の定義は以下のようになり, 全く同じ定義である.

$$\begin{cases} \text{変数宣言の並び} & = \text{変数名の並び } ":" \text{ 型 } ";" \ \{ \text{ 変数名の並び } ":" \text{ 型 } ";" \ \} \\ \text{仮パラメータの並び} & = \text{変数名の並び } ":" \text{ 型 } ";" \ \{ \text{ 変数名の並び } ":" \text{ 型 } ";" \ \} \end{cases}$$

実装コードは定義からほぼ同じコードになる. しかしながら, 宣言変数と仮パラメータの変数では意味的に異なるので, 多くの人がそれぞれ別ルーチンとして処理している. ルーチンに引数を持たせ処理分岐する等の再構築を行えば同じルーチンとして多少の保守性改善が行われる.

この場合は, 大して長くはないコードフラグメントであり, それぞれのコンパイラは独立したものであるが, 単一大規模プログラム内で, このようにシステム内に分散した, もっと長いコードフラグメントが発見されれば, 保守性を大きく下げている可能性があるので, 早急に再構築を行うべきであろう.

5　まとめと今後の課題

本研究では, コードクローン検出ツール CCFinder の検出結果を有効に利用し, コードクローンに基づいたソースコード分析を行うシステムの試作を行った. 主な機能として二つ, 一つはクローンペアの位置情報をクローン散布図で表現し, 対応ソースコードを参照する機能, もう一つは, クローンクラスからソフトウェアメトリクスを算出するとともにグラフ化し, 対応ソースコードを参照する機能である.

またプログラミング演習に適用する中で, システムの有効性の確認を行った. クローン散布図においては, クローン分析における散布図ソート機能の有効性の確認等, クローンメトリクスグラフにおいては, 特徴あるコードフラグメント集合を容易に抽出できることを示した.

今後の課題としては,

- クローンペアテーブルと, クローンクラスツリー間に選択状態の同期をとり, クローン散布図とクローンメトリクスグラフを連携させる
- 検出, 分析, 再構築間のサイクルをインクリメンタルに行えるようにして実用性を向上させる
- 含まれる構文によりクローンペア, クローンクラスを分類表示する
- 不一致部分を含むクローンペアの表示
- 本システムの有効性検証

などが挙げられる.

謝辞

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大阪大学 基礎工学研究科 情報数理系専攻 ソフトウェア科学分野 井上 克郎 教授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 同 楠本 真二 助教授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 同 松下 誠 助手に深く感謝致します.

本研究, プログラミング演習適用実験において, 様々な御配慮を頂いた 大阪大学 基礎工学研究科 情報数理系専攻 計算機科学分野 土屋 達弘 講師に深く感謝致します.

本研究, プログラミング演習適用実験において, 様々な御配慮を頂いた 大阪大学 基礎工学研究科 情報数理系専攻 ソフトウェア科学分野 多田 知正 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 同 神谷 年洋 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 基礎工学研究科 情報数理系専攻 ソフトウェア科学分野 井上研究室の皆様に深く感謝いたします.

参考文献

[1] B. S. Baker, “A Program for Identifying Duplicated Code”, Computing Science and Statistics, 24:49-57, 1992.

[2] S. Ducasse, M. Rieger, and S. Demeyer,“A Language Independent Approach for Detecting Duplicated Code”, Proc. of IEEE Int′l Conf. on Software Maintenance(ICSM)′99, pp. 109-118. Oxford, England. Aug., 1999.

[3] 藤井 邦洋, “トークン単位のソースコード比較ツールの適用事例”, 大阪大学基礎工学部情報科学科特別研究報告, 2000.

[4] D. M. Geary, “Graphic Java2, Mastering the JFC 3rd Edition vol.2”, ASCII, 2000.

[5] D. Gusfield, “Algorithms on Strings, Trees, And Sequences”, Cambridge University Press, 1997.

[6] J. Helfman, “Dotplot Patterns: a Literal Look at Pattern Languages, TAPOS, 2(1):31-1,1995.

[7] S. Holzner, “Java Swing プログラミング Black Book”, 株式会社インプレスコミュニケーションズ, 2000.

[8] 神谷 年洋, 楠本 真二, 井上 克郎, “コードクローン検出における新手法の提案及び評価実験”, 電子情報通信学会技術研究報告 SS2000-42～52 Vol.100 No.570 pp.41-48, 2001.

[9] 加藤 博己, “データベースのビジュアルな検索と分析 (OLAP)”, IPSJ Magazine Vol.41 No.4 pp. 363 - 368, 2000.

[10] 近藤 和弘, “Java プログラムのエイリアス情報表示ツールの試作”, 大阪大学基礎工学部情報科学科特別研究報告, 2000.

[11] C. McClure，“The Three Rs of Software Automation Re-engineering, Repository, Reusability, Prentice-hall”,1992, (藤本 厚，藤堂 清，小野 誠，堀田 耕治，芦沢 真佐子[訳],“ソフトウェア開発と保守の戦略”，共立出版株式会社，1993).

[12] M. Rieger, S. Ducasse, “Visual Detection of Duplicated Code”, 1998.

[13] M. Rieger, “DUPLOC Tutorial”, 1999.

[14] Standard for Software Maintenance, 1219, IEEE Computer Society, 1993.

[15] 辻野 嘉宏, “コンパイラ”, 昭晃堂, 1996.

[16] Java(TM) 2 Platform Standard Edition, http://java.sun.com/j2se/1.3/.