社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

# コードクローン分析ツールGeminiを用いたコードクローン分析手法

肥後　芳樹† 　楠本　真二† 　井上　克郎†

† 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　〒 560-8531 豊中市待兼山町 1-3
E-mail: †{y-higo,kusumoto,inoue}@ist.osaka-u.ac.jp

**あらまし**　ソフトウェアの保守作業を困難にしている原因としてコードクローンが挙げられる．コードクローンとは，ソースコード中の同一，または類似した部分を表す．あるコード片にバグが含まれていた場合，そのコード片のコードクローン全てについて修正の是非を検討する必要がある．我々の研究グループでは，コードクローンに対する保守支援を行うため，分析環境 Gemini を開発してきている．本稿では，Gemini を用いてどのようなコードクローンが検出されるのか具体的な事例を紹介する．また，Gemini を用いた効果的な分析手順について考察を行う．

**キーワード**　コードクローン，ソフトウェア保守，ソフトウェアメトリクス

# Code Clone Analysis Using a Code Clone Analysis Tool Gemini

Yoshiki HIGO†, Shinji KUSUMOTO†, and Katsuro INOUE†

† Graduate School of Information Science and Technology, Osaka University
1-3, Machikaneyama-cho, Toyonaka, Osaka, 560-8531, Japan
E-mail: †{y-higo,kusumoto,inoue}@ist.osaka-u.ac.jp

**Abstract**　Recently, code clone has been regarded as one of factors that make software maintenance more difficult. A code clone is a code fragment in a source code that is identical or similar to another. For example, if we modify a code fragment which has code clones, it is necessary to consider whether we have to modify each of its code clones. Up to now, we have been developing a maintenance support environment **Gemini** against code clones. In this paper, we describe a case study using Gemini. Also, we conduct an effective code clone analysis methodologies from the results of the case study.

**Key words**　Code Clone, Software Maintenance, Software Metrics

## 1. まえがき

ソフトウェアの保守作業を困難にする要因の 1 つとしてコードクローンが指摘されている [2]．コードクローンとは，ソースコード中に含まれる同一または類似したコード片のことであり，「重複コード」とも呼ばれる．コードクローンがソフトウェア中に作りこまれる原因として，既存コードのコピーとペーストによる再利用，頻繁に用いられる定型処理，パフォーマンス改善のための意図的な繰り返し，コード生成ツールによって生成されたコードなどがある [1], [4]．コードクローンの存在が保守作業を困難にするのは，修正されるコード片のコードクローンが存在すれば，その全てのコードクローンに対して修正の是非を検討する必要があるからである．これまでに多くのコードクローン検出・保守支援手法が開発されている [1], [3], [7], [9].

我々の研究グループでもこれまでにコードクローン検出ツール **CCFinder** [4] と分析ツール **Gemini** [10] を開発してきている．CCFinder は大規模なソフトウェアから実用的な時間でコードクローンを検出することが可能である．CCFinder の出力はテキストベースであり，コードクローンの位置情報はファイル名・行番号・列番号で表される．しかし，実システムに対してコードクローン検出を行うと，何万何十万というコードクローンが検出されてしまい，テキスト情報のみでは検出されたコードクローンの把握，理解は難しい．そこで，検出したコードクローンの分析には Gemini を用いる．Gemini はクローン散布図やメトリクスグラフなどさまざまなビューを用いてコードクローン情報の可視化を行う．Gemini を用いることで，ユーザは検出されたコードクローンの量や分布状態をより直感的に把握することができると期待される．

本稿では，Gemini を用いたコードクローン分析の適用実験について述べる．コードクローン分析を行う目的としては，例えば以下のものが考えられる．

（1） 設計とコードの一貫性確認：コードレビューの一つの目的は，ソースコードと設計情報の間の一貫性確認である．一貫性確認の一つの項目として，コードクローンの利用が考えられる．具体的には，設計情報で重複している部分はソースコードでも重複しており，それ以外の部分には重複をしている部分がないかを確認する．

（2） 信頼性の改善：長期間保守をされているシステムのソースコードには多くのコードクローンが存在する可能性が高い．新たなソフトウェア保守の際に，保守対象のコード片に対するコードクローンを調査することで，変更漏れを防ぐことが可能となる．

（3） 保守性の改善：複数のバージョンのソースコードが与えられたときに，バージョン間でのコードクローンの遷移を調

査し，保守作業の効率化を図る．例えば，繰り返し修正が加えられているコードクローンは，集約などを行い将来的な修正コストを抑えるべきであるかもしれない．一方で，安定しているコードクローンはシステムの保守コストを悪化させてはいないので，リファクタリングは不要であると思われる．

本稿で述べるケーススタディは，このうち，1. 設計とコードの一貫性の確認を目的としている．もちろん，設計とコードの一貫性確認にコードクローン分析だけが有効なわけではない．ソフトウェアのドメインに応じて，より優先度の高い分析方法は多く存在する．コードクローン分析は，対象のドメインにあまり左右されない，二次的な解析として有効ではないかと思われる．適用実験では，筆者らが開発した Gemini 自体を対象とした．また，適用実験の結果から，Gemini を用いた効果的と思われる分析方法をいくつか提案する．

## 2. コードクローン分析ツール: Gemini

本節では，コードクローン分析ツール **Gemini** [10] の紹介を行う．全ての機能を紹介することは紙面の都合上無理があるので，本稿で用いているコードクローンの定義，Gemini 内部で用いているコードクローン検出ツール CCFinder，そして本適用実験で用いたクローン散布図，メトリクスグラフ, ファイルリスト，フィルタリングメトリクス $RNR(S)$ について説明する．

### 2.1 コードクローンの定義 [5]

あるトークン列中に存在する 2 つの部分トークン列 $\alpha$，$\beta$ が等価であるとき，$\alpha$ と $\beta$ は互いにクローンであるという．またペア $\alpha$，$\beta$）をクローンペアと呼ぶ．$\alpha$，$\beta$ それぞれを真に包含する如何なるトークン列も等価でないとき，$\alpha$，$\beta$ を極大クローンと呼ぶ．また，クローンの同値類をクローンセットと呼ぶ．ソースコード中でのクローンを特にコードクローンという．

### 2.2 コードクローン検出ツール: CCFinder

**CCFinder** [4] はプログラムのソースコード中に存在する極大クローンを検出し，その位置をクローンペアのリストとして出力する．検出されるコードクローンの最小トークン数はユーザが前もって設定できる．

CCFinder のコードクローン検出手順（ソースコードを読み込んで，クローンペア情報を出力する）は以下の 4 つの STEP からなる．

**STEP1（字句解析）**：ソースファイルを字句解析することによりトークン列に変換する．入力ファイルが複数の場合には，個々のファイルから得られたトークン列を連結し，単一のトークン列を生成する．

**STEP2（変換処理）**：実用上意味を持たないコードクローンを取り除くこと，及び，些細な表現上の違いを吸収することを目的とした変換ルールによりトークン列を変換する．例えば，この変換により変数名は同一のトークンに置換されるので，変数名が付け替えられたコード片もコードクローンであると判定することができる．

**STEP3（検出処理）**：トークン列の中から指定された長さ以上一致している部分をクローンペアとして全て検出する．

**STEP4（出力整形処理）**：検出されたクローンペアについて，ソースコード上での位置情報を出力する．

### 2.3 クローン散布図

クローン散布図の簡単なモデルを図 1 に示す．散布図の原点は左上隅にあり，水平軸，垂直軸は，それぞれソースファイルの並び (f1 から f6) に対応している．両軸上で，原点から順にそれぞれのソースファイルに含まれるトークンが並んでいる．座標平面内に点がプロットされている部分は，その両軸の対応するトークンが一致することを意味する．したがって，散布図の主対角線は，両軸の同じ位置のトークンを比較することになり，全ての点がプロットされることになる．一定の長さ (CCFinder で設定される最小一致トークン数) 以上の対角線分が，検出されたクローンペアである．図 1 では，f1 から f6 までのファイルが，それぞれ 3 つのトークンを含んでいる．ファイルの並びはファイル名の辞書順となっている．検出されるクローンペアは f1 と f6 に含まれる“ ab ”というトークン列と，f3 と f6 に含まれる“ bc ”というトークン列である．

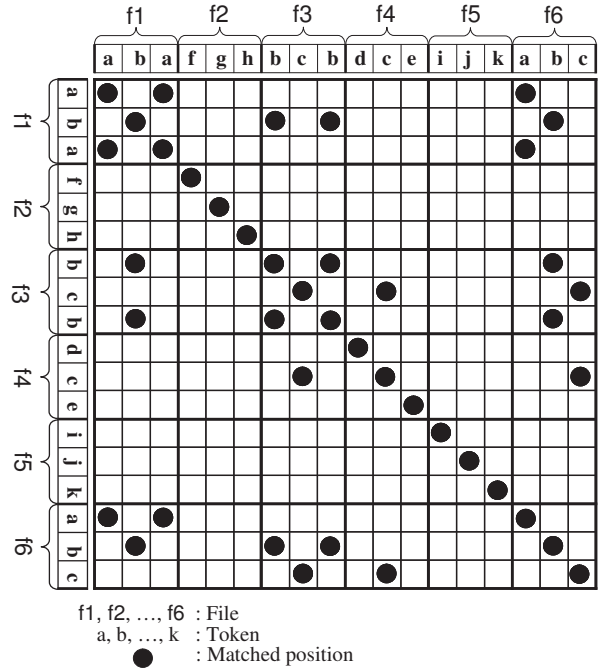
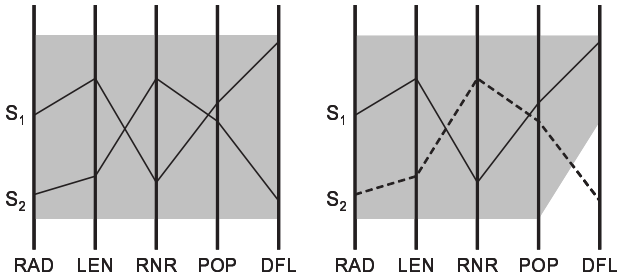



図 1　クローン散布図モデル




(a) 初期値　　(b) $DFL$ 軸の下限を変更

図 2　メトリクスグラフモデル

### 2.4 メトリクスグラフ

メトリクスグラフは検出されたコードクローンをメトリクスを用いて特徴づける．特徴づけされたコードクローンはクローンセット単位で表される．メトリクスグラフの簡単なモデルを図 2 に表す．メトリクスグラフは多次元並行座標表現 [6] を用いている．このグラフで用いられているメトリクスは，$RAD(S)$, $LEN(S)$, $RNR(S)$, $POP(S)$, $DFL(S)$ の 5 つである．ここでは，$RNR(S)$ を除く 4 つのメトリクスについて簡単に説明を行う．$RNR(S)$ については 2.6 節を参照されたい．

**RAD(S):** クローンセット $S$ 内のコード片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．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, $S$ 内でのその最大値を $RAD(S)$ として定義する．

**LEN(S):** クローンセット $S$ 内に含まれるコード片のトークン数の平均値を表す．

**POP(S):** クローンセット $S$ 内のコード片単位の要素数である．$POP(S)$ が高いということは，同形のコード片が多く存在

することになる．

**DFL(S):** クローンセット $S$ に含まれるコード片に共通するロジックを実装するサブルーチンを作り，各コード片をそのサブルーチンの呼び出しに置き換えた場合の減少が予測されるトークン数を表す．

5 つのメトリクスは図 2 の縦軸となっており，それぞれにメトリクス名のラベルがついている．このグラフではクローンセット毎に 5 つの縦軸上の点を結ぶ折れ線が引かれている．ユーザはこれら 5 つの座標の上限と下限を変更することで任意のクローンセットを選択することが可能である．例として図 2(b) は，$DFL$ 軸の下限値を変更した状態を表している．この変更によって，図 2(a) では選択状態であったクローンセット $S_2$ が非選択状態となっている．

### 2.5 ファイルリスト

ファイルリストでは，ユーザは定量的に特徴的なファイルを選択することができる．以下のメトリクスがファイルを定量的に特徴付けるために用いられている．

**NOL(F):** ファイル $F$ の行数を表す．

**NOT(F):** ファイル $F$ のトークン数を表す．

**NOC(F):** ファイル $F$ に含まれているコードクローンの数を表す．

**ROC(F):** ファイル $F$ がどの程度重複化しているかを表す．

**NOF(F):** ファイル $F$ がコードクローンを共有しているファイルの数を表す．

ファイルリストはテーブル形式で実装されており，各行に 1 つのファイルがそのメトリクス値と共に表示される．ファイルリストは行のソーティング機能があり，各メトリクス値の昇順，あるいは降順にファイルを並び替えることが可能である．また，ファイルリストはクローン散布図と連携しており，ファイルリストにおいて選択されたファイルがクローン散布図において強調表示される．

### 2.6 フィルタリングメトリクス: RNR(S)

ここではフィルタリングメトリクス $RNR(S)$ について述べる．$RNR(S)$ はクローンセット $S$ 中に含まれるコード片がどの程度非繰り返しであるかを表すメトリクスである．例えば，以下のトークン列を考える．

$$x\ a\ b\ c\ a\ b\ c^*\ a^*\ b^*\ c^*\ y.$$

* がついたトークンはそれが繰り返しトークン列に含まれていることを表している．このトークン列を入力として与えた場合，CCFinder は以下の 2 つのコード片をクローンとして検出する．

$$\text{F1. } x\ \underline{a\ b\ c\ a\ b\ c^*}\ a^*\ b^*\ c^*\ y,$$
$$\text{F2. } x\ a\ b\ c\ \underline{a\ b\ c^*\ a^*\ b^*\ c^*}\ y.$$

コード片 $F1$ は 6 トークンから成り，そのうち 1 トークンが繰り返しトークンである．一方コード片 $F2$ も 6 トークンから成り，そのうち 4 トークンが繰り返しトークンである．この場合，これらのコード片からなるクローンセット $S_1$ の $RNR(S_1)$ は，

$$RNR(S_1) = \frac{5+2}{6+6} = \frac{7}{12} = 0.58\dot{3}$$

となる．これまでの経験から $RNR(S)$ の値が 0.5 未満の場合は，C 言語であれば連続した `printf` や `scanf` であったり，Java 言語であれば連続した `import` 文であったりと，目で確認を行ってもあまり意味のないコードクローンであることがわかっている．メトリクス $RNR(S)$ を用いることでこのようなコードクローンを取り除くことが可能となる．また，メトリクス $RNR(S)$ はメトリクスグラフで用いられいているだけでなく，クローン散布図においても用いられている．クローン散布図では $RNR(S)$ が 0.5 未満のコードクローンは青く描画され，他のクローンと区別がつくようになっている．

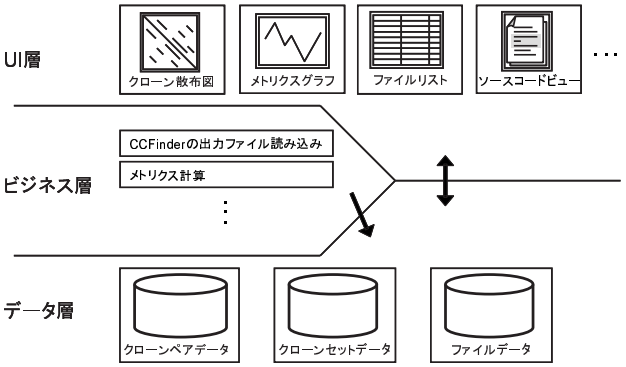



図 3　Gemini の論理アーキテクチャ

## 3. 適 用 実 験

本実験の目的は，対象のソースコードに不必要な (設計情報に含まれない) コードクローンが存在するかを調査することである．本実験の対象は Gemini のソースコードである．筆者らが作成したソフトウェアを対象とすることで，検出されたコードクローンが設計情報に含まれるものかどうかを判断することができる．Gemini のソースコードの総ファイル数は 126 個，総行数は約 26,000 行である．図 3 は Gemini の論理アーキテクチャを表した図である．Gemini は分析の開始時に CCFinder の出力ファイルを読み込み，メトリクス各種データベースを構築する．分析時は，各 UI はデータベースから必要は情報を入手し，さまざまな形でユーザに提示する．ユーザは，各 UI を用いてインタラクティブにコードクローンの分析を進めることができる．

Gemini には多くの UI が存在するが，本実験ではそのうちクローン散布図，メトリクスグラフ，ファイルリストの 3 つを用いて分析を行った．

### 3.1 クローン散布図

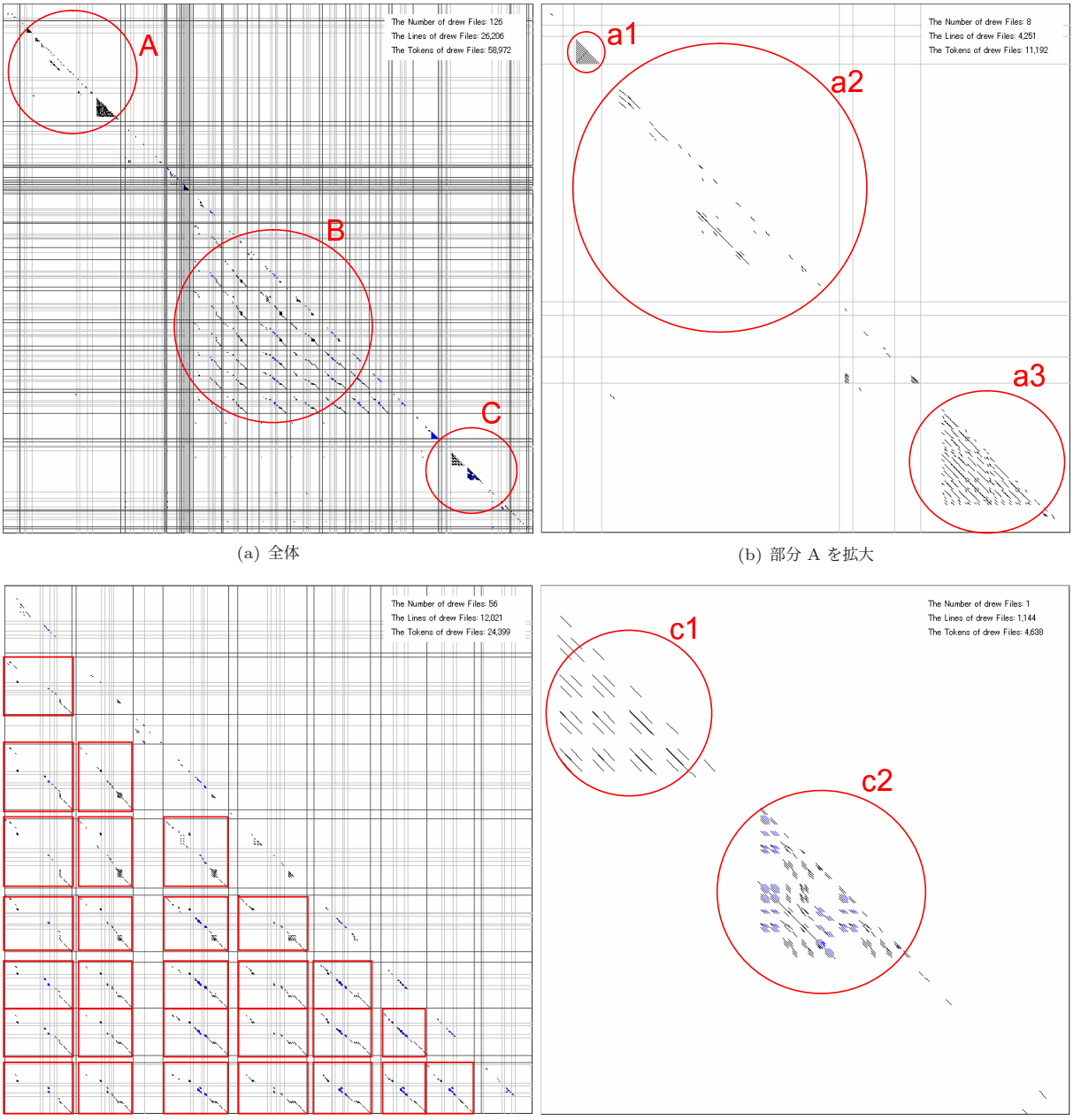
図 4(a) は Gemini のソースコード全体のクローン散布図である．本実験ではこのクローン散布図上の目立つ部分，A，B，C がどのようなコードクローンであるかの調査を行った．

#### 3.1.1 部分 A: クローンペアデータ

図 4(b) は図 4(a) の A の部分を拡大したものである．a1，a2，a3 の 3 つのファイル (クラス) に多くのコードクローンが保有されている．これら 3 つのクラスは各々クローンペアデータを構成するためのクラスである．a1 は 1 つのファイル内に存在するクローンペアデータを表すクラスであり，a2 の部分はいくつかのファイル内のデータを表すクラスであり，a3 は対象ソースコード全体のデータを表すクラスである．つまり a3 は複数の a2 を要素として持ち，さらに a2 は複数の a3 を要素として持つ構造となっている．消費メモリ量を抑えるためにこのような構造となっている．ソースコードを閲覧したところ，実際にクローンとなっていたのは，クローンペアの情報をクローン散布図に書き込むメソッド群であった．高いスケーラビリティを実現するために，検出されたクローンの量によって異なった粒度でクローン散布図に書き込む実装としているため，似たような処理の流れを持ったメソッドが対象に定義されている．これは設計情報に含まれるクローンであった．

#### 3.1.2 部分 B: Table ビュー

図 4(c) は図 4(a) の B の部分を拡大したものである．この部分はディレクトリ単位で類似している部分が多数存在している．図 4(c) の太枠 (赤) の部分は類似度の高いディレクトリを表している．この部分は，1 つのディレクトリにつき，1 つの Table ビューを作成している．Table ビューとは，J2SDK の標準ライブラリに含まれる javax.swing.JTable を継承して作成したビューを意味する．1 つの Table ビューは以下 6 つのクラス




(a) 全体　(b) 部分 A を拡大

(c) 部分 B を拡大　(d) 部分 C を拡大

図 4　クローン散布図

から構成される．

- テーブルの実体となるクラス
- テーブルのデータを表すクラス
- テーブルの行の並び替え機能を実装するクラス (2 つ)
- テーブルの見た目 (色，セルの幅など) を制御するクラス
- テーブルのポップアップメニューを構成するクラス

Gemini には Table として実装されているビューが複数存在する．例えば，対象ソースコードの Table ビュー (ファイルリスト)，検出されたクローンセットの Table ビュー，1 つのクローンセットに含まれるコード片の Table ビューなどがそうである．これらの Table ビューでは扱うデータは異なるが，その機能は非常に類似している．実際に新しい Table を作成する場合，既存のリストの実装部分をコピーし，細かい変更を加えていくという方針で実装を行ったため，非常に類似したディレクトリとなっている．これも設計情報に含まれるクローンであった．

#### 3.1.3　部分 C: ScatterPlotPanel

図 4(d) は図 4(a) の C の部分を拡大したものである．この部分は，クローン散布図を描画する Panel を実装しているファイル (クラス) である．このファイルにはコードクローンが集中している部分が二箇所存在した．c1 の部分は，マウスイベント (クリックやドラッグなど) が発生した時に呼び出されるメソッドの定義部分であった．これらのメソッドの中では，イベントが発生した座標がどのファイルに該当するのかを求めるために位置計算を行っており，その部分がコードクローンとなっていた．c2 の部分はコードクローンの描画の位置計算を行っている部分である．クローン散布図は拡大表示機能などがあるために，位置計算を行う部分が複数箇所に存在してしまっている．これらは 1 つのメソッドにまとめられることなく実装されてしまっていたため，コードクローンとなっていた．これらは，設計情報に含まれないクローンであった．

### 3.2　メトリクスグラフ

ここではメトリクスグラフを用いて特徴的なコードクローンを絞り込み，それらがどのようなものであったかを述べる．なお，予めメトリクス $RNR(S)$ を用いて，その値が 0.5 未満の

クローンセットは除いてある．

**3.2.1** 最もトークン数の多いコードクローン

クローン散布図を実装しているファイル (クラス) には，マウスによるドラック部分を拡大する機能と，選択をする機能がある．いずれの操作中もどこからどこまでをドラッグしているのかをユーザに伝えるために，開始部分から現在位置までを対角線とする長方形を描く実装となっている．しかし，ズーム時と選択時で，描かれる長方形が塗りつぶされるか，塗りつぶされないかの違いがあるため，異なるメソッドで実装を行っている．各機能は 1 つのメソッドとして実装されており，そのメソッドが最もトークン数を多いコードクローンとして検出された．なお，1 つのコード片のトークン数は 568(97 行) であった．このメソッドはコピーとペーストにより作成されたものである．

**3.2.2** 最も同形のコード片が多いコードクローン

同形のコード片が最も多かった 2 つのクローンセットがどのようなものであったかを述べる．

**(1) 位置計算を行っている部分:** 対象システム内の位置から，クローン散布図上での位置を求める計算部分がクローンとなっていた．ここで絞込まれたコードクローンは，3.1.3 節で述べたコードクローンと同一のものであった．この座標変換は，クローン散布図にデータを描画する上で不可欠なものである．クローン散布図にはさまざまな機能があり，随所でデータの変換を行う必要がある．そのため，複数箇所で作業変換を行ってしまっており，コードクローンとなっている．また，このクローンセットの $RAD(S)$ の値は 0，つまり全てのクローンが 1 つのファイル内に含まれていることを表しており，集約を行うことは可能である．しかしクローンの長さが短いここと，外部に強く依存していることから集約をするべきかは疑わしい．

**(2)Table ビューの行の並び替え部分:** Table として実装されているビューの行の並び替えを行っているメソッドの一部がクローンとなっていた．ここで絞りこまれたコードクローンは，3.1.2 節で述べたコードクローンの一部であった．前述のように，Gemini にはさまざまな情報を一覧表示するための Table ビューが存在する．全ての Table ビューは 2 つのクラスを用いて行単位での並び替え機能を実装している．この中で定義されていたメソッドがコードクローンとなっていた．

### 3.3 ファイルリスト

本節では，ファイルリストを用いて，特徴的なファイルがどのような機能を実装していたかを述べる．

**3.3.1** 最も多くのファイルとコードクローンを共有しているファイル

Table ビューを実装しているクラスがそれぞれ他の 10 個のクラスとコードクローンを共有していた．共有していたコードクローンは，スクロールバーを生成している部分，リスナーを登録している部分，マウスの右ボタンがクリックされた時にポップアップメニューを表示するメソッドなど，Java の GUI の特徴的な部分が多くのファイルに共有されているコードクローンであった．これらのコードクローンは，このソフトウェア独自の処理部分を実装しているというよりは，Java での GUI を実装する場合の定型的な処理部分であった．

## 4. 各ビューの特徴と効果的な分析法

クローン散布図，メトリクスグラフ，ファイルリストの特徴をまとめ，それらを用いた効果的な分析法を論ずる．

### 4.1 クローン散布図

- 対象ソフトウェアのどの部分にどの程度のコードクローンが存在するのかを俯瞰的に知ることができる．
- ファイルよりも大きな単位での類似部分を知ることができる．例えば，図 4(c) のコードクローンは複数のディレクトリが類似していることを表している．
- クローン散布図において目立つコードクローンはある程度の領域内に密集している部分 (図 4(b)，図 4(d)) や，繰り返し同じパターンが出現している部分 (図 4(c)) である．
- クローン散布図において目立つ部分は，必ずしもその部分に存在するコードクローンが特徴的であることを示しているわけではない．なぜなら，目立つ部分に存在するコードクローンは 1 種類ではなく，複数の種類のコードクローンが混在している場合があるからである．
- 目立ちやすさとコードクローンの位置が関係している．要素数の多いコードクローンであっても，複数のファイル内に点在してしまうような場合は，クローン散布図上でそれらは互いに離れた部分に描画されてしまい，目立ちにくい．

### 4.2 メトリクスグラフ

- 対象ソフトウェアに存在する特徴的なコードクローンを見つけることができる．クローン散布図では，コードクローンの位置が目立ちやすさに影響を与えてしまうが，メトリクスグラフではそのようなことはない．
- メトリクス $RAD(S)$ を用いることによってクローンセット内の要素がどの程度ファイルシステム上で散らばっているか (クローン散布図上ではなれた位置に描画されているのか) を知ることができる．

### 4.3 ファイルリスト

- 定量的な情報に基づいてファイルを選択することができる．クローン散布図のように，コードクローンの位置に影響されることはない．

前述したようにクローン散布図では，目立ちやすさとコードクローンの位置が関係している．例えば，100 個のコードクローンを含んでいるファイル (ファイル A とする) があるとする．もしこの 100 個のコードクローンが全て，他の 1 つとのファイル (ファイル B とする) と共有していた場合は，クローン散布図上では，水平方向がファイル A，垂直方向がファイル B の領域内にコードクローンが密集しているのが目立つと考えられる．しかし，100 個の異なるファイルと共有していた場合は，特定の領域にコードクローンが密集することはなく，クローン散布図上でそのようなファイルを見つけることは困難である．しかし，ファイルリストを用いれば，$NOC(S)$ の値を確認するだけで，そのファイルにどの程度コードクローンが含まれているのかを定量的に知ることができる．

3.3 節では述べなかったが，メトリクス $NOC(F)$ や $ROC(F)$ の値の高いファイルはクローン散布図において目立つ部分とほぼ一致した．しかし，$NOF(F)$ 値の高いファイル内には，クローン散布図でもメトリクスグラフでも目立たないコードクローンが存在した．それらは対象ソフトウェアではなく，プログラミング言語に依存したコードクローンであった．

### 4.4 効果的な分析法

上記の特徴から，Gemini を用いた効果的であると思われる分析方法を提案する．なお，STEP2~4 は順序不同である．

効果的な分析法

**STEP1**: 大まかな把握
**STEP2**: 要素数の多いクローンセットの特定
**STEP3**: トークン数の多いクローンセットの特定
**STEP4**: 多くのファイルとクローンを共有しているファイルの特定

以降本節では，各 STEP についての説明を行う．

**STEP1**: 大まかな把握

新規でコードクローン分析を行う場合は，まずクローン散布図を用いてコードクローンの分布状態を大まかに把握するとよい．クローン散布図では，マウスカーソル位置の垂直方向のファイル，水平方向のファイルのパスがリアルタイムで表示されるため，目立つ部分にマウスカーソルを移動させるだけで，その部分がどのファイルであるのかを知ることができる．

以下の 2 つの部分が目立ちやすい部分である．

- 特定の領域内にコードクローンが密集している部分
- 同じようなパターンが繰り返し現れている部分

また，クローン散布図上では，メトリクス $RNR(S)$ の値が 0.5 未満のコードクローンは青色，0.5 以上のコードクローンは黒色で描画される．目立つ部分のコードクローンの多くが青色の場合は，その部分に注意を払う必要は少ないと思われる．

**STEP2**: 要素数の多いクローンセットの特定

要素が多いということは，その機能がソフトウェアの多くの箇所で実装されていることを表しており，ソフトウェアの象徴的な処理部分であると考えられる．本稿での適用実験により検出されたコードクローンもまさしくそのようなものであった．また，要素が多いということは，その部分にバグが検出された場合，多くの箇所で同様の修正を行わなければならないことを示しており，このようなコードクローンは望ましくないと捉えることができる．このようなことから，要素数の多いクローンセットはリファクタリングの対象となるのではないのかとも考えられる．

要素数の多いクローンセットの特定にはメトリクスグラフを用いる．要素数を表すメトリクスは $POP(S)$ であるが，フィルタリングメトリクス $RNR(S)$ も同時に用いた方が好ましい．なぜなら，繰り返し部分は要素数の多いクローンセットになりがちであるからである．以下の部分を変更することで，要素数の多いコードクローンを特定することができる．

- RNR 軸の下限を上げて，繰り返し部分の多いコード片から成るクローンセットを取り除く
- POP 軸の下限を徐々に上げていき，該当クローンセット数が少なくなるようにする

メトリクスグラフの絞込みの結果，該当したクローンセット一覧は，クローンセットリストと呼ばれるビューに表示される．ユーザはこのリストから任意のクローンセットを選択し，そのソースコードを閲覧することができる．

**STEP3**: トークン数の多いクローンセットの特定

トークン数が小さいコードクローンは，偶然の一致によるものが存在するが，トークン数の大きいコードクローンはそのほとんどがコピーとペーストにより生成されたものであると考えられる．通常コピーとペーストの後には，変数名の付け替えや呼び出すメソッドの変更など細かな修正が行われる．もしこの修正漏れがあった場合はバグが発生してしまう．トークン数の大きいコードクローンを特定し，そのような確認作業を行うことは有効ではないかと思われる．

トークン数の多いクローンセットの特定にもメトリクスグラフを用いる．要素数の多いクローンセットの絞込みと似ており，$POP(S)$ の代わりに $LEN(S)$ を用いればよい．

**STEP4**: 多くのファイルとコードクローンを共有しているファイルの特定

多くのファイルとコードクローンを共有しているファイル内に存在するコードクローンは，現在の設計ではまとめることが困難なものであるかもしれない．プログラミング言語に適切な抽象化機構が存在していないのが原因であるかもしれないが，このようなコードクローンが多く存在する場合は，設計をみなしたほうが良いのかもしれない．またこのようなコードクローンは，アスペクト思考プログラミングの「横断的関心事」であるかもしれない．

多くのファイルとコードクローンを共有しているファイルの特定にはファイルリストを用いる．ファイルリストには，コードクローンの検出対象となっているファイルの一覧が表示されている．これらのファイルをメトリクス $NOF(F)$ の値の降順にソーティングをすることによって，多くのファイルをクローンを共有しているファイルを特定することができる．また，ファイルリストでファイルを選択すると，クローン散布図の該当ファイルの部分が強調表示される．

## 5. 関 連 研 究

門田ら [8] は COBOL で記述されたレガシーコードに対してコードクローン検出を行い，バグとの関係を調査している．この研究では，各ファイルにおいてそのファイルのクローンとなっている行の割合，ファイル中の最大長クローンとそのファイルの改版数を定量的に比較し，一定以上の割合でクローンが含まれている場合や，非常に長いコードクローンが含まれていた場合，そのファイルの改版数が多くなることが示されいる．

また，Kim ら [3] は，複数のバージョンに対してコードクローン検出を行い，その情報がソフトウェア保守に利用できると提唱している．例えば，同様の修正が各コード片に施されているコード片は，将来の修正コストを抑えるためにリファクタリングをするべきではないかと考えられる．

## 6. ま と め

本稿ではコードクローン分析ツール Gemini を用いた適用実験について述べた．適用実験では設計情報に含まれるもの，含まれないもの，プログラミング言語に依存したものなど，さまざまなコードクローンが検出された．また，適用実験の結果から，有益と思われるコードクローンの分析方法について検出した．今後は，より大きなシステムに対して Gemini の適用を行う予定である．

**謝辞** 本研究は一部，日本学術振興会 科学研究費補助金 基盤研究 (A)(課題番号：17200001)，文部科学省 科学研究費補助金 特別研究員奨励費 (課題番号：16・8351) の助成を得た．

## 文 献

[1] I.D. Baxter, A. Yahin, L. Moura, M.S. Anna, and L. Bier, *Clone Detection Using Abstract Syntax Trees*, Proc. 14th International Conference on Software Maintenance, pp.368-377, Bethesda, Maryland, Mar. 1998.

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

[3] M. Kim, and D. Notkin, *Using a Clone Genealogy Extractor for Understanding and Supporting Evolution of Code Clones* newblock Proc. The 2nd International Workshop on Mining Software Repositories, pp.17-21, May 2005.

[4] 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, Jul. 2002.

[5] 神谷年洋，“コードクローンとは，コードクローンが引き起こす問題，その対策の現状”，電子情報通信学会誌 Vol.87, No.9 pp.791-797, Sep. 2004.

[6] 加藤 博己，“データベースのビジュアルな検索と分析 (OLAP)”，情報処理学会会誌，Vol.41 No.4 pp.363 - 368, Apr. 2000.

[7] J. Mayland, C. Leblanc, and E.M. Merlo *Experiment on the automatic detection of function clones in a software system using metrics*, Proc. 12th International Conference on Software Maintenance, pp.244-253, Monterey, California, Nov. 1996.

[8] 門田暁人, 佐藤慎一, 神谷年洋, 松本健一，“コードクローンに基づくレガシーソフトウェアの品質の分析”，情報処理学会論文誌, Vol.44, No.8, pp.2178-2187, Aug. 2003.

[9] M. Toomim, A. Begel, and S.L. Graham, *Managing Duplicated Code with Linked Editing* Proc. 2004 IEEE Symposium on Visual Languages and Human-Centric Computing(VL/HCC'04), Rome, Italy, Sep. 2004

[10] 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, Vol.86-D-I, No.12, pp.863-871, Dec. 2003.