産学連携に基づいたコードクローン可視化手法の改良と実装

肥後芳樹[†] 吉田則裕[†]
楠本真二[†] 井上克郎[†]

近年，ソフトウェアの大規模化・複雑化に伴い，保守作業に要するコストが増大している．ソフトウェアの保守を困難にしている要因の 1 つ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ある一部分（コード片）のうち，他のコード片と同一または類似しているものを指す．コードクローンはコピーアンドペーストなどのさまざまな理由によりソースコード中に作りこまれる．例えば，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考慮する必要がある．コードクローンを対象とした保守支援を行うために，著者らは検出ツール CCFinder・可視化ツール Gemini を開発し，産業界に配布している．また，著者らはツールの開発者（大学）と利用者（産業界）の意見交換の場としてコードクローンセミナーを開催している．セミナーを開くことによって出席者から現場の生の声を聞くことができる．セミナーの開催に加えてメーリングリストの運営も行っている．利用者はツールの利用法に関する質問や新機能の要望を行い，開発者はツールのバージョンアップや次期セミナーの開催日程などを告知するために用いている．その結果 Gemini に実装されているコードクローン可視化手法を大幅に改良することができた．さらにこの改良手法を用いて Gemini を再実装し，実際のソフトウェア保守作業で用いることのできる実用的なツールに発展させることに成功した．本稿では，著者らの産学連携の取り組みと，コードクローン可視化手法の改良について述べる．また，新 Gemini を日本のベンダー 5 社が共同開発したソフトウェアに対して適用した．適用の結果，対象ソフトウェア内に存在するさまざまなコードクローン情報を簡単に得ることができた．

Improvement and Implementation of Code Clone Visualization Method based on Academic-Industrial Collaboration

Yoshiki Higo,[†] Norihiro Yoshida,[†] Shinji Kusumoto[†]
and Katsuro Inoue[†]

Maintaining software systems becomes more difficult as the size and complexity of software increase. One of the factors that makes software maintenance more difficult is the presence of code clones. A code clone is a code fragment which has identical or similar code fragments to it in source code. Code clones are introduced by various reasons such as reusing code by 'copy-and-paste'. If we modify a code clone with many similar code fragments, it is necessary to consider whether or not we have to modify each of them. For supporting software maintenance against code cloning, we have developed a code clone detection tool, CCFinder and a code clone visualization tool, Gemini. These tool have been delivered to domestic or overseas organizations/individuals. Also, we have held code clone seminars that provide opportunities for discussions between developers and users of the tools. Through the seminars, we can get what industrial people really require. In addition to seminars, we are managing a mailing list. Users ask about how to use the tools and require new functionalities that they want, and developers announce a version upgrade of the tools and the date for next seminar. As a result, we were able to improve our visualization method and succeed to refine it as a practical one. We re-implemented Gemini based on the improvements as a tool which can be used in practice. In this paper, we describe how we are promoting academic-industrial collaboration and how the visualization method was improved. Moreover, We applied new Gemini to a system which was co-developed by 5 Japanese companies. Application results demonstrate the usefulness and capability of new Gemini.

1. はじめに

近年，ソフトウェアの大規模化・複雑化に伴い，保守作業に要するコストは増大している．例えば，Yip

† 大阪大学大学院情報科学研究科
Graduate School of Information and Science Technology, Osaka University

らはコスト全体の 66%が保守作業に費やされている，と報告している[16]．保守作業を困難にしている原因の１つとして，コードクローンが挙げられる．コードクローンとは，ソースコード中のある一部分（コード片）のうち，他のコード片と同一または類似しているものを指す．Fowler は，重複コード（コードクローン）は最も優先してリファクタリングを行う対象である，と述べている[4]．コードクローンは，コピーアンドペーストなどの理由によりソースコード中に生成される．あるコード片に対して修正を行う場合，もしそのコード片のコードクローンが存在する場合は，それらについても同様の修正の是非を検討する必要がある．このような作業は，システムが大きい場合は非常に煩雑であり，また修正漏れによる新たなバグの混入の危険もある．それゆえ，システム中に存在するコードクローンを効率的に検出することが必要である．

これまでにさまざまなコードクローン検出手法が提案されている[1)~3),10),13)]．著者らもコードクローン検出ツール CCFinder を開発してきている[7)]．CCFinder は，コードクローンを高速に検出することを目的として実装されており，数百万行規模のシステムであっても実用的な時間で検出を行うことができる．

しかし CCFinder が出力するコードクローン情報はテキスト形式であり，コードクローン分析作業でその情報をそのまま用いるのは効率的でない．分析作業を支援するために，コードクローン可視化ツール Gemini を開発してきている[15)]．そして CCFinder・Gemini を国内外の組織・個人に配布しており，現在では 100 社以上で使用されている．

また，コードクローンセミナーを開催し，ツールの開発者（大学）と利用者（産業界）との意見交換の場を設けている．メーリングリストも運用し，利用者から新機能の要望やツールの利用法に関する質問などのために用いている．このような取り組みを通じて，産業界のコードクローン検出・可視化技術に対するニーズを把握し，ツールの改良や技術移転を続けてきている．

本稿では，産業界の意見を基に改良したコードクローン可視化技術とその適用事例について述べる．本手法は，様々な開発組織から得られた知見を基に実現しており，汎用性が高い手法となっている．さらにこの手法を，新 Gemini として再実装した．新 Gemini の有用性を確認するために，情報処理推進機構（IPA）ソフトウェア・エンジニアリング・センター（SEC）の先進ソフトウェア開発プロジェクトにおいてベンダー 5 社が共同で開発したシステム[12)] に対して適用した．適用の結果，さまざまなコードクローンの状態を得ることができ，改良手法・新ツールの有用性が確認された．

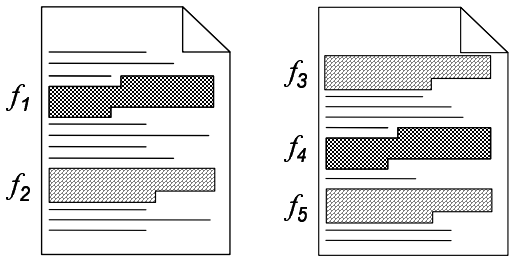


図 1 クローンペアとクローンセット
Fig. 1 Clone Pair and Clone Set

2. コードクローン

2.1 定　　義

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のうち，他のコード片と一致または類似しているものを指す．しかし，コードクローンの厳密で普遍的な定義は存在しない．これまでにさまざまなコードクローン検出手法が提案されているが，それらはどれも異なった定義を持つ．以降，本稿では，コードクローン検出ツール CCFinder の定義を用いる．

CCFinder ではコードクローンであるか否かはコード片の同値関係（反射律，推移律，対称律）で決定される[5),7)]．ここで，**コード片**とはソースファイルの一部分を指し，ID，$Line_{start}$，$Column_{start}$，$Line_{end}$，$Column_{end}$ の 5 つの属性を用いて表される．$ID(f)$ はコード片 f を含むファイルの ID を表す．CCFinder は全てのコードクローン検出対象ファイルに対してユニークな ID を割り当てる．$Line_{start}(f)$（$Line_{end}(f)$）はコード片 f の開始行（終了行）を表し，$Column_{start}(f)$ ($Column_{end}(f)$) はコード片 f の開始列（終了列）を表す．この定義では，コード片は部分的に重なり合う場合もありうる．

ある系列中 (ソースコード中) に存在する 2 つの部分系列（コード片）α，β が同一または類似しているとき，$C(\alpha, \beta)$ と書き，α は β とクローン関係をもつという．C は，反射律，推移律，対称律が成り立つ同値関係である．また，コードクローンの同値類を**クローンセット**という．

任意の α，β に対して $C(\alpha, \beta)$ ならば，α の任意の部分系列 $\dot{\alpha}$ に対し，$C(\dot{\alpha}, \dot{\beta})$ となる β の部分系列 $\dot{\beta}$ が存在する．また，α，β をそれぞれ真に含む任意の系列 $\ddot{\alpha}$，$\ddot{\beta}$（ただし $\ddot{\alpha} \neq \ddot{\beta}$）に対して $C(\alpha, \beta)$ かつ $\neg C(\ddot{\alpha}, \ddot{\beta})$ ならば，(α, β) を**クローンペア**という．

図 1 はクローンペアとクローンセットの例である．

この例では，5 つのコードクローンが存在している．コード片 f_1 はコード片 f_4 とクローン関係を持ち，またコード片 f_2，f_3，f_5 も互いにクローン関係を持つ．この場合，$(f_1, f_4), (f_2, f_3), (f_2, f_5), (f_3, f_5)$ の 4 つのクローンペアと $\{f_1, f_4\}, \{f_2, f_3, f_5\}$ の 2 つのクローンセットが存在する．

2.2 CCFinder

CCFinder[7] はプログラムのソースコード中に存在するコードクローンを検出し，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字句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 から成る．

STEP1（字句解析）：ソースファイルを字句解析することにより字句列に変換する．ファイルが複数の場合には，個々のファイルから得た字句列を連結し，単一の字句列を生成する．

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字句列を変換する．例えば，連続した複数の関数にまたがったコードクローンは分割され，各関数内で閉じたコードクローンとして検出される．また，変数名は同一の特別な字句に置換されるので，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）：字句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のソースコード上での位置情報を出力する．

3. 産学連携の取り組み

本節では，著者らが行ってきた産学連携の取り組みについて紹介する．産学連携を行うことによって，産業界のコードクローン分析に対するニーズを把握することができ，コードクローン可視化手法を実用的な技術として高めることができた．

3.1 コードクローンセミナー

著者らはツールの開発者（大学）と利用者（産業界）の意見交換の場としてコードクローンセミナーを開催している[17]．第 1 回セミナーは 2002 年 11 月に開催し，これまでに東京で 3 回，大阪で 3 回の計 6 回開催している．セミナーの内容は毎回異なるが，ツールのデモンストレーションと利用法講座や，コードクローン情報の利用法，また実際にツールを使用している企業の事例報告など，その内容は多岐に渡る．

3.2 メーリングリスト

コードクローンセミナーは現場の生の声が聞けるという利点があるが，頻繁に行うのは難しい．また，全てのユーザに出席して頂くことも現実的ではない．このことから，メーリングリストを運用している．利用者（産業界）はメーリングリストを用いてツールの利用法に関する質問や新機能の要望などを行い，また開発者（大学）は，ツールのバージョンアップや次期セミナーの開催日程などを利用者に告知している．

3.3 産業界からの要望

著者らは既にコードクローンの可視化手法を提案し，その手法を実装した可視化ツール Gemini を開発している[15]．この手法・ツールは，学術的に見れば新規性はあるものの，実際の開発現場の使用には向いていなかった．指摘された主な問題点を以下に示す．

- コードクローン情報の中に，調査の必要があるコードクローンとそうでないものが混在している．
- ファイルに着目した分析を行うための機能が存在しない．
- スケーラビリティが低い．

1 つ目の問題の原因は，検出したコードクローンをフィルタリングせずに可視化しているという点であった．CCFinder は調査しなければならないコードクローンの他に調査の必要がないコードクローンも検出してしまう．ここで，“調査の必要がないコードクローン”とは，ソフトウェア開発・保守を行う視点でコードクローン情報を扱う場合に特に対象とする必要が無いものである．ソースコード中には，プログラミング言語やドメイン，フレームワークに依存した定型処理部分が含まれており，これらは CCFinder によって頻繁に検出されてしまう．このようなコードクローンの存在は，調査を必要とするコードクローン情報を隠蔽し，分析作業の非効率化を招いてしまう．実用的な可視化手法にするには，調査の必要がないコードクローンのフィルタリングを行う必要がある．フィルタリングを行うことにより，調査の必要があるコードクローンの分布状態の把握や，調査の必要があるコードクローンのうち特定の特徴を持つものの抽出などが可能となる．

2 つ目の問題は，ファイルに着目した分析を行うことができない点であった．旧ツールは，コードクローンに対してはその特徴を基に選択を行うための機構が存在するが，検出対象ファイルについては選択機構が存在しなかった．しかし，利用者はコードクローンを多く含むファイルや重複度の高いファイルに興味を持

つため，それらを効率的に選択する機構が必要である．

3 つ目の問題は，スケーラビリティの低さであった．旧ツールが円滑に動作するのは，対象ソフトウェアが 10 万行程度の規模までであり，商用ソフトウェアを対象とした場合，このスケーラビリティでは不十分であった．実用的なツールであるためには，対象ソフトウェアが数百万行規模であっても円滑に動作する必要がある．

4. コードクローン可視化手法の改良

4.1 提 案 手 法

ここでは，改良を行ったコードクローン可視化手法について述べる．この改良手法は，3.3 節で述べた問題点を補うためのものである．計 4 つの手法を述べるが，各手法によってどのようにコードクローン情報が扱われるのかを表すために，以下の例を全ての手法の説明で共通して用いる．

コードクローンの例

ディレクトリ D_1，D_2 が存在し，D_1 の下にはファイル F_1 と F_2 が，D_2 の下にはファイル F_3 と F_4 が存在する．ファイル $F_1 \sim F_4$ は次の字句で構成されている．“*” の意味は 4.1.1 節で述べる．

F_1: $a\ b\ c\ a\ b$,
F_2: $c\ c^*\ c^*\ a\ b$,
F_3: $d\ e\ f\ a\ b$,
F_4: $c\ c^*\ d\ e\ f$

コード片を表すためにラベル $C(F_i, j, k)$ を用いる．$C(F_i, j, k)$ はファイル F_i の j 番目の字句から k 番目までの字句で構成されるコード片を表す．

ここでは，コードクローンとして検出されるために最低 2 字句が必要であるとすると，次の 3 つのクローンセットが検出される．

S_1: $\{C(F_1,1,2), C(F_1,4,5), C(F_2,4,5), C(F_3,4,5)\}$
S_2: $\{C(F_2,1,2), C(F_2,2,3), C(F_4,1,2)\}$
S_3: $\{C(F_3,1,3), C(F_4,3,5)\}$

4.1.1 調査の必要がないコードクローンのフィルタリング

CCFinder は調査しなければならないコードクローンのほかに，調査の必要がないコードクローンも検出してしまう．“調査の必要がないコードクローン” とは，ソフトウェア開発・保守を行う視点でコードクローン情報を扱う場合に特に対象とする必要が無いものである．調査の必要がないコードクローンは，そのソフトウェアが記述されているプログラミング言語に依存した，いわゆる言語依存のコードクローンと，そのソフトウェアのドメインや用いているライブラリ・フレームワークに依存した，アプリケーション依存のコードクローンに大別される．言語依存のコードクローンは，そのプログラミング言語を使っていれば，どのようなアプリケーションでも検出されてしまうのに対し，アプリケーション依存のコードクローンは，対象ソフトウェアによって全く異なる．このため，本稿では調査の必要がないコードクローンのフィルタリングの第一歩として，言語依存のコードクローンのフィルタリング手法を提案する．例えば，連続した変数宣言やメソッド呼び出し，switch 文の連続した case エントリなど，プログラミング言語の構造上どうしてもコードクローンになってしまうものが言語依存のコードクローンである．

効率的なコードクローン分析を実現するために，このようなコードクローンをフィルタリングするためのメトリクス **RNR(S)** を提案する．$RNR(S)$ はクローンセット S に含まれるコード片がどの程度繰り返し要素を含まないかを表す．

f をクローンセット S に含まれているコード片とする．$TOC(f)$ はコード片 f を構成している字句の数，$TOCrepeated(f)$ はコード片 f を構成している字句のうち，繰り返し要素の字句の数を表すとする．このとき $RNR(S)$ は次式で表される．

$$RNR(S) = 1 - \frac{\sum_{f \in S} TOC_{repeated}(f)}{\sum_{f \in S} TOC(f)}$$

また繰り返し要素の字句とは，直前の字句列の繰り返しである字句列中の字句を指す☆．例では，“*” 付きで表示されている字句が繰り返し要素の字句である．例の 3 つのクローンセットに対して RNR の値を算出すると，$RNR(S_1) = 1.0$, $RNR(S_2) = 0.3$, $RNR(S_3) = 1.0$ となり，S_2 を構成するコード片の大部分が繰り返し要素であることがわかる．このメトリクスを使うことによって，連続した変数やアクセサ宣言，メソッド呼び出しなどのコードクローンをフィルタリングすることが可能である．

一方，アプリケーション依存のコードクローンに対するフィルタリングは現在のところ行えていない．こ

☆ 各字句が繰り返しであるか否かは正規表現のメタ文字の 1 つである “+” を用いると理解しやすい．F_2 は “+” を用いることによって $c + ab$ と表すことができる．この表現で省略されている字句が繰り返し要素である．例えば，新たなファイル $F_5 : x\ a\ b\ a\ b\ a\ b\ y$ を考えた場合，このファイルは “+” を使って $x(ab) + y$ と表されるため 4 番目から 7 番目までの字句で構成される字句列 $a\ b\ a\ b$ が繰り返し要素となる．

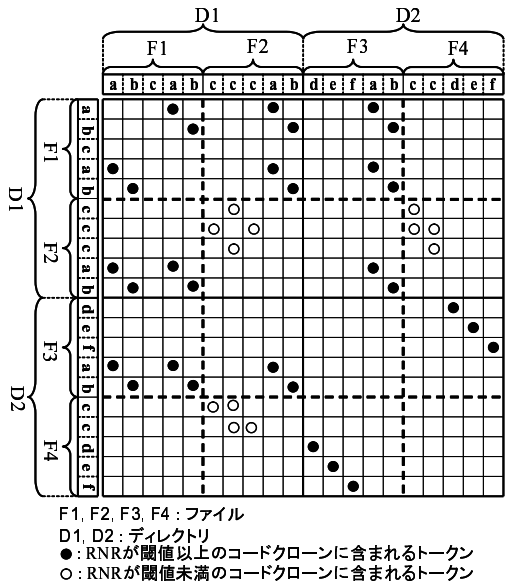


図 2 クローン散布図
Fig. 2 Model of Scatter Plot

のようなコードクローンは，対象ソフトウェアが用いているフレームワークやライブラリに依存しているため，その性質はさまざまであり，言語依存のコードクローンのように，コードクローンの特徴を定量化してフィルタリングを行うことは難しい．しかし，開発現場からのフィードバックとして，アプリケーション依存のコードクローンのフィルタリング機構を希望する声は大きく，現在検討中である．

4.1.2 クローン散布図

調査の必要があるコードクローンの分布状態を把握するために，クローン散布図に対して改良を行った．図 2 は本稿で提案するクローン散布図を例を用いて表現したモデルである．水平・垂直軸にはソースコード中の字句が出現順に配置される．各ソースファイルはそのパスのアルファベット順でソートされており，同じディレクトリに位置するファイルは近い位置に出現する．クローン散布図では，各クローンペアが線分として表現される（例ではコードクローンとして最低 2 字句が必要であるとした）．線分を構成している各字句は，その水平成分と垂直成分が等しいことを意味している．クローン散布図は左上隅から右下隅への対角線に対して常に線対称になる．クローン散布図を用いることによって，ユーザはコードクローンの量や分布状態を俯瞰的に把握することができる．

このクローン散布図はメトリクス RNR によるフィルタリングの結果を表示している．◯ で示された字句は，それを含むクローンセットがメトリクス RNR を用いることによって，調査の必要がない，と判断されたことを表している（ここでは，RNR の閾値として 0.5 を用いた）．調査の必要がないコードクローンを他のコードクローンと異なって表示することにより，ユーザは，それらを除外して分析作業を行うことが可能となる．これにより，より効率的にコードクローン分析作業を行うことができる．

また，このクローン散布図は，ファイル間の境界線とディレクトリ間のそれを区別して表示する．これにより，ユーザは，ファイル内クローン・ファイル間クローンだけでなく，ファイル内クローン・ディレクトリ内ファイル間クローン・ディレクトリ間クローンを区別することができ，どのディレクトリが多くコードクローンを所有しているのか，どのディレクトリ間にコードクローンが多く共有されているのか，を瞬時に把握することが可能である．

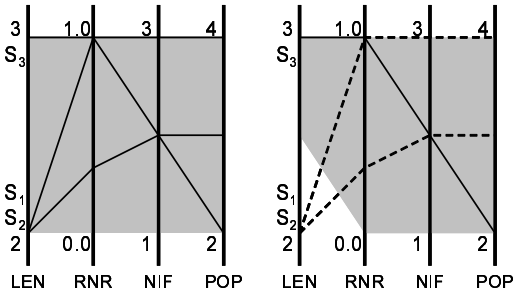
4.1.3 クローンセットメトリクス

ユーザが，興味のある特徴を持ったコードクローン情報に瞬時にアクセスできるように，メトリクスを用いてコードクローンを定量的に特徴付ける．調査の必要がないコードクローンをフィルタリングするために，メトリクス RNR も用いる．用いるクローンセットメトリクスを以下に示す．なお，LEN と POP は文献 15) で提案したものと同じである．

$LEN(S)$**[15]** クローンセット S に含まれるコード片の大きさ (字句数) の平均を表す．例の 3 つのクローンセットに対してこのメトリクスを計算すると，$LEN(S_1) = 2$, $LEN(S_2) = 2$, $LEN(S_3) = 3$, となる．このメトリクスを用いることによって，クローンセット S_3 のコード片が他のクローンセットのコード片に比べて長いことがわかる．

$POP(S)$**[15]** クローンセット S に含まれるコード片の数を表す．この値が大きいほど，同型のコード片の数が多いことを意味する．例の 3 つのクローンセットに対してこのメトリクスを計算すると，$POP(S_1) = 4$, $POP(S_2) = 3$, $POP(S_3) = 2$ となる．このメトリクスを用いることによって，クローンセット S_4 が他のクローンセットに比べて多くのコード片を所有していることがわかる．

$NIF(S)$ クローンセット S に含まれるコード片を所有しているファイルの数を表す．この値が高い場合，システムの設計が悪い，プログラミング言語に適切な抽象化機構が存在しない，横断的関心事であるなどの原因が考えられる．例の 3 つのクローンセットに対してこのメトリクスを計算すると，



(a) 選択前 (b) 選択後

図 3 メトリクスグラフを用いたクローンセットの選択
Fig. 3 Model of Metric Graph

$NIF(S_1) = 3$, $NIF(S_2) = 2$, $NIF(S_3) = 2$ となる．このメトリクスを用いることによって，クローンセット S_1 が他のクローンセットに比べて多くのファイルを巻き込んでいることがわかる．

$RNR(S)$ クローンセット S に含まれるコード片の繰り返し要素でない部分の割合を表す．調査する必要の無いコードクローンのフィルタリングを行うために用いられる．このメトリクスの詳細は 4.1.1 節を参照されたい．

次に上記のメトリクスを用いてどのようにクローンセットを選択するかを説明する．クローンセットの選択機構をメトリクスグラフと呼ぶ．メトリクスグラフのモデルを図 3 に示す．メトリクスグラフは多次元並行座標表現[9] を用いている．各メトリクスにつき 1 つの座標軸が用意される．また各クローンセットにき 1 つの折れ線がそのメトリクス値に基づいて描画される．ユーザは任意のメトリクスの上限または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 3(b) は，LEN 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 3(a) では選択状態であったクローンセット S_1 と S_2 が非選択状態となっている．

4.1.4 ファイルメトリクス

ファイルに着目した分析を実現するために，クローンセットだけでなく，対象ファイルもコードクローン情報を用いて特徴づけを行う．ファイルメトリクスはメトリクス RNR を用いたフィルタリングの結果，調査の必要がないと判断されたコードクローンを除外したコードクローン情報を用いて計算される．具体的には，メトリクス RNR の閾値を th とした場合，このメトリクスの値が th 以上のクローンセットのみがファイルメトリクスの計算に用いられる．ここでは，th を 0.5 とする．ファイルリストは以下に示す 3 つである．

$NOC_{th}(F)$ ファイル F に存在するコード片の数を表す．例の 4 つのファイルに対してこのメトリクスを計算すると，$NOC_{0.5}(F_1) = 2$, $NOC_{0.5}(F_2) = 1$, $NOC_{0.5}(F_3) = 2$, $NOC_{0.5}(F_4) = 1$ となる．このメトリクスを用いることによって，ファイル F_1, F_3 がファイル F_2, F_4 に比べて多くのコードクローンを含んでいることがわかる．

$ROC_{th}(F)$ ファイル F がどの程度重複化しているかを表す．F の字句数（コメントは除く）を $TOC(F)$，F の字句のうち RNR の値が th 以上のクローンセットに含まれているものの数を $TOC_{duplicated}(F, th)$ とした場合，$ROC_{th}(F)$ は次式で計算される．

$$ROC_{th}(F) = \frac{TOC_{duplicated}(F, th)}{TOC(F)}$$

例の 4 つのファイルに対してこのメトリクスを計算すると，$ROC_{0.5}(F_1) = 0.8$, $ROC_{0.5}(F_2) = 0.4$, $ROC_{0.5}(F_3) = 1.0$, $ROC_{0.5}(F_4) = 0.6$ となる．このメトリクスを用いることによって，ファイル F_3 が完全に重複化していることがわかる．

$NOF_{th}(F)$ 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．例の 4 つのファイルに対してこのメトリクスを計算すると，$NOF_{0.5}(F_1) = 2$, $NOF_{0.5}(F_2) = 2$, $NOF_{0.5}(F_3) = 3$, $NOF_{0.5}(F_4) = 1$ となる．このメトリクスを用いることによって，ファイル F_3 が他の全てのファイルとコードクローンを共有してることがわかる．

4.2 コードクローン可視化ツール: Gemini

4.1 節で提案した手法に基き，コードクローン可視化ツール Gemini を再実装した．新 Gemini は CCFinder が対応している全てのプログラミング言語☆に対して適用可能である．新 Gemini は以下のビューを，4.1.2 節 ～ 4.1.4 節の提案手法の実装として持つ．

- クローン散布図,
- メトリクスグラフ,
- ファイルリスト

これらのビューで選択したコードクローンやファイルは，簡単にそのソースコードを閲覧できる．

4.1.2 節で述べたように，クローン散布図を用いることによって，ユーザはコードクローンの分布状態を俯瞰的に把握することができる．これは特に分析の初期段階で有効であると考えられる．図 2 では，クローン散布図は左上隅から右下隅への対角線に対して線対称であるが，レンダリングコストを抑えるために，実

☆ C/C++, COBOL, FORTRAN, Emacs-Lisp, Java

File List

FILE ID	LOC(f)	TOC(f)	NOC(f)	ROC(f)(%)	NOF(f)	File Name
FILE 0.0	137	121	2(2)	82(82)	1(1)	ZipShort.java
FILE 0.1	203	408	2(2)	39(39)	1(1)	ExtraFieldUtils.java
FILE 0.2	113	5	0(0)	0(0)	0(0)	UnixStat.java
FILE 0.3	135	124	0(0)	0(0)	0(0)	UnrecognizedExtraField.java
FILE 0.4	496	861	10(12)	23(23)	14(17)	ZipEntry.java
FILE 0.5	119	6	0(0)	0(0)	0(0)	ZipExtraField.java
FILE 0.6	557	1018	3(7)	6(12)	1(1)	ZipFile.java
FILE 0.7	141	160	3(3)	83(83)	2(2)	ZipLong.java
FILE 0.8	871	1686	8(17)	24(29)	2(2)	ZipOutputStream.java
FILE 0.9						
FILE 0.10	442	668	4(4)	22(22)	1(1)	TarBuffer.java
FILE 0.11	191	5	0(0)	0(0)	0(0)	TarConstants.java
FILE 0.12	682	993	2(5)	3(13)	4(6)	TarEntry.java
FILE 0.13	417	628	1(1)	6(6)	1(1)	TarInputStream.java
FILE 0.14	348	462	1(1)	8(8)	1(1)	TarOutputStream.java
FILE 0.15	233	317	0(0)	0(0)	0(0)	TarUtils.java
FILE 0.16	81	22	0(0)	0(0)	0(0)	ErrorInQuitException.java
FILE 0.17	562	969	7(9)	15(19)	15(15)	MailMessage.java
FILE 0.18	131	157	0(0)	0(0)	0(0)	SmtpResponseReader.java
FILE 0.19	136	5	0(0)	0(0)	0(0)	BZip2Constants.java
FILE 0.20	873	2069	24(25)	38(39)	2(2)	CBZip2InputStream.java
FILE 0.21	1670	3940	29(47)	21(26)	2(2)	CBZip2OutputStream.java
FILE 0.22	167	81	0(0)	0(0)	0(0)	CRC.java
FILE 0.23	277	295	2(6)	15(39)	1(1)	AnsiColorLogger.java
FILE 0.24	339	716	3(4)	23(26)	1(1)	CommonsLoggingListener.java

Z:¥apache-ant-1.6.0¥src¥main¥org¥apache¥tools¥zip¥ZipLong.java

図 4 ファイルリストのスナップショット
Fig. 4 A Snapshot of File List

装では対角線より上方のコードクローンは描画しない．なお，この実装では，RNR の値が閾値以上のコードクローンを黒，閾値未満のコードクローンを青で表示する．

メトリクスグラフは定量的な特徴に基いてクローンセットを選択するための機構である．クローン散布図では，各クローンセットがどの程度目立つかは，それに含まれるコード片間の位置関係に大きく依存しており，定量的な特徴に基いて選択を行うことは難しい．例えば，コード片を 100 個所有するクローンセット $S_{example}$ があるとする．もし全てのコード片が同一ファイル内に存在する場合は，クローン散布図上でそれらは近い位置に描画され，把握しやすい．しかし，もしコード片が多くのファイル内に分散して存在する場合は，クローン散布図上では，各クローンペアは離れた位置に描画されてしまい，把握しづらくなる．一方，メトリクスグラフでは，“100 個のコード片を持つ” という特徴は，メトリクス $POP(S_{example})$ で表されるため，コード片の位置に関係なくユーザは容易に $S_{example}$ を選択できる．

ファイルリストは，ソースファイルを選択するための機構である．図 4 はファイルリストのスナップショットである．ファイルリストは，全てのソースファイルを 4.1.4 節で紹介した 3 つファイルメトリクスと 2 つのサイズメトリクス $LOC(F)$，$TOC(F)$ と共に表示する．$LOC(F)$ と $TOC(F)$ はそれぞれファイル F の行数と字句数を表す．

著者らはファイルの選択機構としてメトリクスグラフを用いなかった．ファイルを選択する場合は，メトリクス値だけでなく，ファイル名やパス名も用いたほうが利便性が高い．4.1.3 節で述べているように，メトリクスグラフは数値に基いた絞込みには非常に向いている．しかし，ファイル名などに基いた選択には向いていない．この理由から，ファイルを選択する機構としてメトリクスグラフでなく，ファイルリストを用いた．また，ファイルリストに表示されているファイルは，ファイル名のアルファベット順やメトリクス値の昇順・降順でソーティング可能である．

クローン散布図は，コードクローンの分布状態を俯瞰的に表示するので便利ではあるが，スケーラビリティが低く実用的ではないという指摘もある[8),14)]．しかし，著者らが実装したクローン散布図は，JDK1.5 のソースコード全体（約 1,880,000 行）から検出したコードクローン（クローンセット数：約 12,000 個，クローンペア数：2,500,000 個）に対して円滑に動作することを確認している．

5. 適 用 実 験

新 Gemini を情報処理推進機構（IPA）ソフトウェア・エンジニアリング・センター（SEC）の先進ソフトウェア開発プロジェクトにおいてベンダー 5 社が開発しているプローブ情報システム☆に対して適用した[12)]．各社は個別に開発を行っており，プロジェクトマネージャでもソースコード，開発工数，開発体制（外部委託先，要員数等）に関しては知りえない状況であった．プロジェクトマネージャが主催する進捗会議では，各社のマネージャから各工程の進捗割合と，予定日数のずれが報告されるのみであった．このようなブラインドマネジメントを支援するために，つまりブラックボックスとなっているソースコードの特徴を把握するために，コードクローン分析を行った☆☆．

新 Gemini の適用は単体テスト終了後，結合テスト終了後の 2 回行った．全社合計の開発規模は数十万ステップであり，結合テスト後は単体テスト後に比べ約 2 万ステップ増加していた．このシステムは C/C++ 言語を用いて開発されている．コードクローン分析は各社が開発したソースコードに対して個別に行った．この実験では，30 字句以上のコードクローンを検出した．なお，検出された全てのコードクローンは関数内に閉じたものである．紙面の都合上，全ての分析結果を示すのは困難であるので，一部分のみを示す．5.2

☆ プローブ情報システムとは，センサーなどの計測機器が収集した情報をネットワークを通じてセンターに転送し，その情報を分析・蓄積・加工などすることによってさまざまな有用な情報を提供するシステムである．

☆☆ この分析は共同開発各社合意の下に確保されたソフトウェア工学研究用の機密室内で行った．

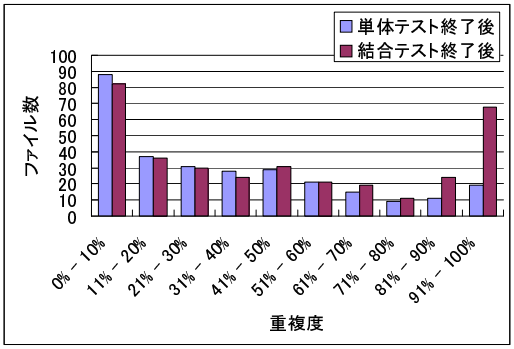


図 5 Y 社の重複度の変化
Fig. 5 Duplicated ratio of file in the sub-system developed by Y company

節 ～ 5.4 節の各分析は，結合テスト後のコードに対する適用結果である．なお，この適用実験では，メトリクス RNR の閾値は 0.5 とした☆．

5.1 履歴分析

単体テスト後と結合テスト後のクローン検出量の変化を調査した．表 1 は単体テスト後，結合テスト後の検出された重複コード片の数とシステムの重複度を表している．Y 社の開発部分に，単体テスト後に比べて結合テスト後に多くのコードクローンが含まれていた．通常，単体テスト後には新規で実装される機能は無いため，単体テスト後と結合テスト後のクローン検出量にあまり差はない，と予測していた．Y 社が開発したファイルの重複度分布状態のグラフを図 5 に示す．この図からわかるように，結合テスト後は重複度が非常に高いファイル (81% - 100%) の数が増えていた．Y 社の開発者にインタビューを行ったところ，これらのファイルは，結合テストを行う前に，新しい機能を実装するために社内で構築したライブラリコードを流用した部分であった．ライブラリのファイル間で多くのコードクローンを共有しているため，重複度は非常に高いが，これまでに数多くのソフトウェア内で運用されており，信頼できるとの回答を得た．

表 1 検出されたクローンの量
Table 1 Amount of code clones in sub-systems

	単体テスト後		結合テスト後	
	コード片数	重複度	コード片数	重複度
V 社	259	33.9%	259	33.4%
W 社	369	27.3%	379	26.2%
X 社	4,483	55.3%	4,768	50.8%
Y 社	6,747	42.6%	7,628	46.0%
Z 社	2,450	56.2%	2,505	56.3%

☆ 本来ならば，RNR の値が 0.5 以下であるクローンセット全てを調査し，フィルタリングの精度を算出すべきである．しかし，5 社全てのソースコードを 10 時間程度で分析しなければならないという時間的制約の都合上，そのような分析を行うことができなかったことをご理解頂きたい．

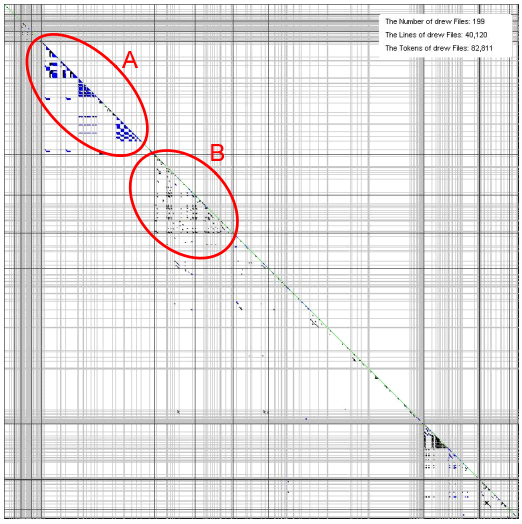


図 6 クローン散布図のスナップショット
Fig. 6 A snapshot of Scatter Plot

5.2 クローン散布図を用いた分析

図 6 は，ある社が作成したコードのクローン散布図である．A の部分にメトリクス RNR でフィルタリングされたコードクローンが多数存在していた．これらのソースコードを閲覧したところ，デバック用の情報出力コード（連続した printf 文）やデータの妥当性チェック（連続した if 文）であり，うまくフィルタリングを行えていることが確認できた．

B の部分にも多くのコードクローンが存在した．この部分のコードクローンは黒の格子をまたいで存在することからディレクトリ間クローンであることがわかる．この部分は車両の位置情報を扱う処理部分であり，車両種別によりディレクトリが分かれていた．ディレクトリ毎に扱う情報の種類は異なるものの，処理内容は非常に類似していた．

5.3 メトリクスグラフを用いた分析

この分析では，あらかじめ RNR の値が 0.5 以下のクローンセットは除いている．

5.3.1 LEN の大きいコードクローン

ある社のソースコードから検出された最長のコードクローンは長さが 441 字句 (154 行) であった．このクローンセットは 2 つのコード片からなり，それぞれ AAAXXXBBB.cpp と AAAYYYBBB.cpp というファイルに含まれていた．AAAYYYBBB.cpp 中のコードクローン内で呼び出されている関数の名前の一部が XXX であり，また該当箇所のコメントにも XXX が含まれていた．このことから，このコードクローン

は，AAAXXXBBB.cpp から AAAYYYBBB.cpp にコピーアンドペーストされ，その後修正を忘れたのではないかと思われる．

5.3.2 POP の大きいコードクローン

全社において POP の値が最も大きかったクローンセットは，データの整合性をチェックし，もし間違っていればエラーを出力する，という処理の部分であった．データの種類は社間で違いがあるものの，処理内容は全社で共通していた．

5.3.3 NIF の大きいコードクローン

ある社のソースコードから検出されたクローンセットの NIF の最大値は 8 であった (8 個のファイル内に存在する)．このコードクローンは，文字列の終端に NULL があるかをチェックし，もしなければ追加する処理を行っている関数であった．関数全体が重複していたため，ユーティリティパッケージへの移動などで，簡単に集約可能である．

5.4 ファイルリストを用いた分析

この分析では，RNR の閾値は 0.5 とした．

5.4.1 NOC の大きいファイル

ある社が開発したソースコードに，358 個のコードクローンを所有しているファイルが存在した．コードクローンは特定の処理をしている部分に集中しているわけではなく，さまざまな部分がファイル内クローンもしくはディレクトリ内ファイル間クローンになっていた．特に問題であると思われるコードクローンは特定されなかったが，このファイル自体が非常に多くの処理を行っており，その保守性が疑われた．

5.4.2 ROC の大きいファイル

ある社のソースコード中に重複度が 96%であるファイルが 2 つ検出された．一方はある処理をオンライン時に行い，他方は同様の処理をオフライン時に行うファイルであった．開発者にこのコードクローンの存在を報告したところ，設計時からオンライン時とオフライン時のコードは別々に実装すると決めており，存在が把握されているコードクローンであった．

5.4.3 NOF の大きいファイル

ある社が開発したソースコードに，他の 13 個のファイルとコードクローンを共有しているファイルが存在した．このファイルにはさまざまな入力処理が含まれており，それぞれの処理が別ファイルとコードクローンになっており，結果として NOF の値が大きくなっていた．コードクローンになっている部分は，対象ソフトウェアで汎用的な処理であり，処理内容も単純であることから，特に問題はないと思われた．

5.5 旧 Gemini との比較

ここでは，本実験を仮に旧 Gemini で行った場合，どのような不都合が生じるのかを述べる．

本実験の対象規模は 5 社合計で数十万ステップであり，それぞれの社が担当したサブシステムの規模にはかなりの差があった．ある社が担当していたサブシステムの規模は，他の 4 社のサブシステムの合計規模よりも大きかった．このため，もしこの社のサブシステムを旧 Gemini で分析した場合，クローン散布図のスケーラビリティの問題により，分析が困難であったことが考えられる．

特徴的なクローンセットを調査するためのメトリクスは旧 Gemini でも用いており，同様の分析を行うことは可能である．しかし，新 Gemini では，メトリクスグラフにフィルタリングメトリクス RNR を新たに導入していることにより，より効率的にクローンセットの調査を行うことができる．RNR の低いコードクローン，つまりプログラミング言語に依存したコードクローンは，システムの至る所に存在することから，メトリクス POP と NIF の値が高い傾向であることがわかっている．本実験では，POP と NIF が高いクローンセットを調査しており，旧 Gemini を用いた場合はこの分析により時間が必要であったことが想定される．

また，特徴的なファイルを調査するために本実験で用いたメトリクス NOC，ROC，NOF はいずれも新 Gemini において導入されたメトリクスであり，旧 Gemini において同様の分析を行うことはできない．

6. 産業ソフトウェアの分析

ここでは，これまでに筆者らが産業界のソフトウェアを分析したことによって得た知見を述べる．筆者らは，本稿で述べたソフトウェア以外にも，産業ソフトウェアの分析作業を行ってきている．

コードクローン分析の利用法として外部委託先が作成したソースコードのチェックが挙げられる．外部委託先へ支払う金額が規模ベースの場合，不必要にコードクローンを作成し，規模を大きくしている場合が多数見つかっている．全く同一内容のファイルが複数存在している場合もある．ソフトウェアの規模が大きい場合このような工作を手動によるチェックで見抜くのは難しいが，コードクローン分析を用いることにより簡単に見つけることができる．

コードクローンを多く含む箇所を “設計が失敗した部分” と捉え，次回以降のプロジェクトで同じ過ちを繰り返さないための利用法もある．特に注意しなけれ

ばならないのは，開発者が存在を知らなかったコードクローンである．このようなコードクローンの存在を確認し，なぜ発生したのかその原因を究明することによって，次回以降のプロジェクトにおいて，修正漏れなどを減らすことが可能となる．ここでは簡単に 2 つの利用法は述べたが，コードクローン分析はソースコードさえあれば行うことが可能であるのでその汎用性は非常に高く，他にもさまざまな利用法が考えられる．

産業界のソフトウェアのソースコードを企業の外部に持ち出すことは現実的ではないため，著者らが企業まで赴いて分析作業を行わなければならない．コードクローン分析はソースコードを対象としているので，プロジェクトのマネージャよりも実際にコーディングを行った開発者の意見を頂きたい場合が多い．しかし，コーディングを外部委託していたり，長期開発のソフトウェアにおいてコードクローンが検出された部分の開発者が現在は違うプロジェクトに移動になっていたりなど，開発者から意見を頂ける場合は少ない．また企業まで赴いて分析をするということから，分析作業に費やすことのできる時間は限られてくる．本稿で述べた実験も実際に分析作業に割り当てることができたのは 10 時間程度であった．

7. 関 連 研 究

Kapser らはコードクローンに対する理解支援を目的としたツール CLICS を開発している[8]．CLICS はソースファイルの構造やシステムのアーキテクチャをコードクローン情報を付与して表示する．CLICS はクエリ処理を実装しており，ユーザは興味のある特徴をクエリとして与えることにより，簡単にその条件を満たすコードクローン情報を得ることができる．Kapser らはクローン散布図はスケーラビリティが高くなく実用的ではないと述べており，CLICS はクローン散布図を実装していない．

筆者らは，コードクローンの状態を把握するためにはクローン散布図は必要不可欠であると考えている．新 Gemini のクローン散布図は過去に提案されたクローン散布図に比べ十分にスケーラビリティが高い．例えば，JDK1.5(1,881,140 行) を対象とした適用実験では，コード片の長さが 30 字句以上のクローンペアが 2,497,433 個，クローンセットが 12,522 個検出されたが，クローン散布図は円滑に動作した☆．また，著者らのクローン散布図は，調査の必要がないコードクローンのフィルタリング結果を表示する．これは他のクローン散布図に比べ改良されている点である[1),3),14)]．Kapser らも調査の必要がないコードクローンのフィルタリングは重要であると述べており，彼らのツールもフィルタリング機構を持つ[8)]．また，著者らのクローン散布図は，ディレクトリ境界とファイル境界を異なって表示する．これにより，ファイル内・ファイル間のコードクローンの分布情報だけでなく，ディレクトリ間・ディレクトリ内ファイル間・ファイル内のより詳しいコードクローンの分布状態を知ることができる．これらの 2 つの機能を持ったクローン散布図を使うことにより，ユーザは調査する必要のあるコードクローンの，ディレクトリ階層中での分布状態を知ることができる．

Rieger らはコードクローン情報の視覚的な表示方法を提案・実装している[14)]．彼らの提案は Polymetric View[11)] の原理に基いており，ユーザにさまざまな粒度で抽象化されたコードクローン情報を提供する．彼らもまた，コードクローン検出対象が非常に大きいシステムである場合は，非常に多くのコードクローンが検出されてしまうため，フィルタリング機構が必要不可欠であると述べている．

Johnson は HTML を用いたコードクローン情報の巡回手法を提案している[6)]．HTML のハイパーリンクを用いることによって，コードクローンを共有しているソースファイル間を自由に巡回することを可能にしている．しかし，Johnson の手法には，コードクローンの状態を俯瞰的に把握する機構・コードクローンを定量的に特徴付けする機構が無く，どのコードクローンから分析すべきかユーザは判断することができない．

8. おわりに

8.1 ま と め

本稿では，コードクローン可視化手法の改良について述べた．これは著者らの産学連携の結果得られた知見に基づいており，実用性に長けている．また，この改良手法を用いてコードクローン可視化ツール Gemini を再実装した．新 Gemini は以下の機能を持つ．

- 調査の必要がないコードクローンのフィルタリングを行う機能
- 対象ソフトウェア全体のコードクローンの分布状態を俯瞰的に提示する機能
- 特定の特徴を持ったコードクローン・ファイルの情報を簡単に得ることができる機能

新 Gemini を用いて企業が開発したソフトウェアに対して適用実験を行った．実験の結果，さまざまなコードクローンの状態・発生理由を把握することができた．

☆ CPU:Pentium4 3.0GHz，メモリ:2GB，OS:WindowsXP

8.2 今後の課題

現在の Gemini ではプログラミング言語依存のコードクローンのフィルタリングしか行うことができないため，アプリケーション依存のコードクローンへの対策を行わなければならない．アプリケーション依存のコードクローンは，その特徴はさまざまであり，言語依存のコードクローンのようにメトリクスを用いてフィルタリングを行うことは難しい．このため，ユーザが検出して欲しくないコードクローンのパターンを入力し，そのパターンにマッチするコードクローンをフィルタリングするなどの方法を用いることが考えられる．

ツールのスケーラビリティをさらに高くすることも課題の 1 つである．Gemini を再実装することによって，分析可能なソースコードの規模は 10 万行単位から 100 万行単位まで上昇した．100 万行規模まで分析可能になったことで，1 つのソフトウェアが対象の場合は円滑に分析作業を行うことができるようになった．しかし，その部署で過去に開発したソフトウェアからまとめてコードクローン分析を行い，ソフトウェアにまたがって存在するコードクローンを社内ライブラリ化したい，との声も寄せられている．現在のスケーラビリティではこのような分析を行うことは難しく，更なるスケーラビリティ面での改良が望まれている．

現在，著者らは関数内で閉じたコードクローンのみを対象としている．このため，検出されるコードクローンはあまり大きくない．しかし，ソフトウェア内部にはより大きい単位での重複関係が存在している可能性がある．例えば，CCFinder が検出したコードクローン情報をアーキテクチャ情報にマッピングをさせることによって，より大きい単位での重複部分を抽出することが考えられる．

コードクローンの検出精度そのものも高めていく必要もある．現在 CCFinder が検出できるコードクローンは，全く同一もしくは識別子名が異なるコードクローンのみであり，コピーアンドペースト後に文が挿入・削除されたコード片はコードクローンとして検出できていない．このようなコードクローンを検出できれば，クローン分析はより実用的になると考えられる．

また，コードクローン情報と他の情報の相関を調べることにより，なにか有益なことがわかるかもしれない．例えば，コードクローン情報とバグ情報を付き合わせることにより，コードクローンとバグ発生頻度どの程度関連しているか，つまりコードクローンがどの程度保守作業に影響を与えているかが特定できるのではないかと思われる．

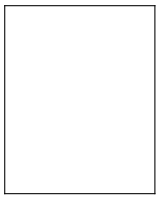
謝辞 旧 Gemini の実装を行った宇宙航空研究開発機構の植田泰士氏に感謝する[15]．本稿の適用実験を行うにあたり協力を頂いた，独立行政法人情報処理推進機構の松浦清氏，神谷芳樹氏，樋口登氏，奈良先端科学技術大学院大学の松村知子氏に感謝する．本研究は一部，文部科学省リーディングプロジェクト「e-Society 基盤ソフトウェアの総合開発」の委託に基づいて行われた．また，日本学術振興会 科学研究費補助金 基盤研究 (A)(課題番号:17200001)，特別研究員奨励費 (課題番号:16-8351) の助成を得た．

参 考 文 献

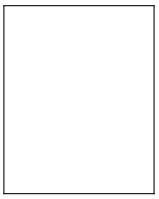
1) Baker, B. S.: On Finding Duplication and Near-Duplication in Large Software Systems, *Proc. of the 2nd Working Conference on Reverse Engineering*, pp.86–95 (1995).
2) Baxter, I., Yahin, A., Moura, L., Anna, M. and Bier, L.: Clone Detection Using Abstract Syntax Trees, *Proc. International Conference on Software Maintenance 98*, pp.368–377 (1998).
3) Ducasse, S., Rieger, M. and Demeyer, S.: A Language Independent Approach for Detecting Duplicated Code, *Proc. of the International Conference on Software Maintenance 99*, pp. 109–118 (1999).
4) Fowlor, M.: *Refactoring: improving the design of existing code*, Addison Wesley (1999).
5) 井上克郎：エンピリカルソフトウェア工学の研究と実践 —コードクローンを例に—，EASE プロジェクトニュースレター， Vol.4, pp.1–4 (2005).
6) Johnson, J.H.: Navigating the Textual Redundancy Web in Legacy Source, *Proc. of the 1996 Conference of Centre for Advanced Studies on Collaborative Research*, pp.7–16 (1996).
7) Kamiya, T., Kusumoto, S. and Inoue, K.: 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 (2002).
8) Kapser, C. and Godfrey, M.: Improved Tool Support for the Investication of Duplication in Software, *Proc. of the 21st International Conference on Software Maintenance*, pp.305–314 (2005).
9) 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 (2000).
10) Komondoor, R. and Horwitz, S.: Using slicing to identify duplication in source code, *Proc. of the 8th International Symposium on Static Analysis*, pp.40–56 (2001).
11) Lanza, M. and Ducasse, S.: Polymetric Views:

A Lightweight Visual Approach to Reverse Engineering, *IEEE Transactions on Software Engineering*, Vol.29, No.9, pp.782–795 (2003).
12) 松浦　清，神谷芳樹，樋口　登：先進ソフトウェア開発プロジェクト PartⅡ，*SEC journal*, Vol.5, pp.44–49 (2006).
13) Mayrand, J., Leblanc, C. and Merlo, E.: Experiment on the automatic detection of function clones in a software system using metrics, *Proc. of the International Conference on Software Maintenance 96*, pp.244–253 (1996).
14) Rieger, M., Ducasse, S. and Lanza, M.: Insights into System-Wide Code Duplication, *Proc. the 11th Working Conference on Reverse Engineering*, pp.100–109 (2004).
15) 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 (2003).
16) Yip, S. W.L. and Lam, T.: A Software Maintenance Survey, *Proc. of the 1st Asia-Pacific Software Engineering Conference*, pp. 70–79 (1994).
17) ソフトウェア工学工房（コードクローンセミナー）：. http://sel.ist.osaka-u.ac.jp/kobo/.

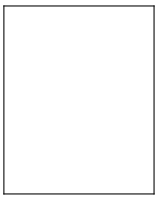
(平成？年？月？日受付)
(平成？年？月？日採録)



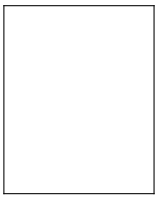
肥後　芳樹（正会員）
平 14 阪大・基礎工・情報中退．平 18 同大大学院博士後期課程修了．現在日本学術振興会特別研究員．コードクローン分析・リファクタリング支援の研究に従事．



吉田　則裕（学生会員）
平 16 九工大・情報工・知能情報卒．平 18 阪大大学院博士前期課程修了．現在同大大学院博士後期課程 1 年．コードクローン分析・リファクタリング支援に関する研究に従事．人工知能学会会員．



楠本　真二（正会員）
昭 63 阪大・基礎工・情報卒．平 3 同大学院博士課程中退．同年同大・基礎工・情報・助手．平 8 同学科講師．平 11 同学科助教授．平 14 阪大・情報・コンピュータサイエンス・助教授．平 17 同学科教授．博士 (工学)．ソフトウェアの生産性や品質の定量的評価，プロジェクト管理に関する研究に従事．IEEE，JFPUG 各会員．



井上　克郎（正会員）
昭 54 阪大・基礎工・情報卒．昭 59 同大大学院博士課程了． 同年同大・基礎工・情報・助手．昭 59〜昭 61 ハワイ大マノア校・情報工学科・助教授．平 1 阪大・基礎工・情報・講師．平 3 同学科・助教授．平 7 同学科・教授． 工学博士．平 14 阪大・情報・コンピュータサイエンス・教授．ソフトウェアエ 学の研究に従事．日本ソフトウェア科学会，IEEE，ACM 各会員．