OUKA
Osaka University Knowledge Archive

Title	類似コード片を利用したリファクタリングの試み
Author(s)	肥後, 芳樹; 神谷, 年洋; 楠本, 真二 他
Citation	情報処理学会研究報告. ソフトウェア工学研究会報告. 2003, 2003-SE-143(73), p. 29-36
Version Type	VoR
URL	https://hdl.handle.net/11094/50148
rights	ここに掲載した著作物の利用に関する注意 本著作物の著作権は情報処理学会に帰属します。本著作物は著作権者である情報処理学会の許可のもとに掲載するものです。ご利用に当たっては「著作権法」ならびに「情報処理学会倫理綱領」に従うことをお願いいたします。
Note	

Osaka University Knowledge Archive : OUKA

https://ir.library.osaka-u.ac.jp/

Osaka University

社団法人 情報処理学会　研究報告
IPSJ SIG Technical Report
2003－SE－143　(5)
2003／7／17

類似コード片を利用したリファクタリングの試み

肥後芳樹 † 神谷年洋 ‡ 楠本真二 † 井上克郎 †

† 大阪大学大学院情報科学研究科
〒 560-8531 大阪府豊中市待兼山町 1-3

‡ 科学技術振興事業団 若手個人研究推進事業
〒 332-0012 埼玉県川口市本町 4-1-8

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．ソフトウェアの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同一，または類似したコード片のことである．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．コードクローンに対する問題に対処するために，我々はコードクローン分析環境 Gemini を開発してきた．これまでに Gemini をさまざまなプロジェクトに適用する中で，いくつかの問題点が指摘された．その一つは，リファクタリングをクローン検出の目的とした時に，Gemini によってユーザに示されるクローンがリファクタリングに適していないということであった．本論文ではこの問題を解決するための手法を提案し，その手法を Gemini の機能拡張として実装した．また，オープンソースのソフトウェアに対して適用実験を行ない，本手法の有用性を確認した．

On Program Refactoring Using Code Clone Information

Yoshiki Higo[†] Toshihiro Kamiya[‡] Shinji Kusumoto[†] Katsuro Inoue[†]

[†] Graduate School of Information Science and Technology
Osaka University
1-3 Machikaneyama-cho, Toyonaka,
Osaka 560-8531, Japan

[‡] PRESTO, Japan Science and Technology Corporation
4-1-8 hon-machi, Kawaguchi,
Saitama 332-0012, Japan

Maintaining software systems is getting more complex and difficult task. Code clone is one of the factors that make software maintenance more difficult. A code clone is a code portion in source files that is identical or similar to another. If some faults are found in a code clone, it is necessary to correct the faults in its all code clones. We have developed a maintenance support environment, **Gemini**, which provides the user with the useful functions to analyze the code clones and modify them. However, through case studies, several problems were reported. That is, the clones provided by Gemini were not appropriate for refactoring. In this paper, we intend to extend the functionality of Gemini to cope with the problems. Finally, we apply the extended Gemini to several software and evaluate the applicability of the new functions.

1　はじめに

ソフトウェアの大規模化・複雑化に伴い，高品質なソフトウェアを効率的に開発する手法が重要となっている．ソフトウェアプロセスの改善はその手法の一つといえる．近年，保守工程はソフトウェア開発で最もコストを要する過程であると指摘されている．また多くのソフトウェア会社が既存のシステムの保守に非常に多くの人的，時間的コストをかけているとも報告されている [22]．ソフトウェアシステムの保守とは，そのシステムが顧客に渡された後に，バグの修正や性能の改善のために，システムを修正することを意味する [19]．

コードクローンはソフトウェア保守を困難にしている一つの要因といわれている [7]．コードクローンとはソースコード中に存在する同一，または類似したコード片のことである．コードクローンが生成される原因はさまざまな理由が考えられるが，その最も大きな原因の一つとしてコピーアンドペーストによる修正，拡張作業があげられる．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となっている部分全てに対して修正の是非を考慮する必要がある．特に大規模ソフトウェアにおいては，これは非常に手間のかかる作業であるので，コードクローン検出の効率化はソフトウェア保守工程の改善において有効である．これまでにコードクローン自動的にを発見するためのさまざまな手法が提案されている．

その手法の一つとして，我々はコードクローン検出ツール CCFinder[12] と分析環境 **Gemini**[20] を開発してきている．Gemini の一コンポーネントに CCFinder が用いられている．ユーザは Gemini を用いることによりコードクローンの解析，ソースコードの修正を容易に行なうことができる [21]．Gemini は主に，クローン散布図とメトリクスグラフをユーザインターフェースとして提供する．クローン散布図はソースコード中のコードクローンの分布状態を俯瞰的に表示する．またメトリクスグラフは各々のコードクローンについての定量的な情報を提供し，その値を用いることによって保守を阻害するコードクローンの選択をすることが可能である．選択されたコードクローンのソースコードは容易に閲覧することができる．ユーザはこれらの機能を用いることによってソフトウェアの保守作業を改善することができると期待できる．

我々は Gemini を数十のソフトウェア会社に配布し，さまざまなプロジェクトに用いることによって評価した．その結果，ソフトウェア会社からのフィードバックよりいくつかの問題点が発見された．最も多く指摘された問題は，Gemini をリファクタリング [7] に利用する際に発生する問題であった．一般的に，コードクローンを除去することを目的として，コードクローンになっている部分を一つのメソッドやクラスにまとめるリファクタリングが適用される．しかし Gemini によって検出されたコードクローンは，必ずしも一つのモジュールとしてまとめるのに適していない．

本論文では，この問題を解決するために Gemini のコードクローン検出部に対して行なった拡張について論ずる．そして最後に，提案した機能の有用性を確認するために行なった適用実験の結果について述べる．

2　コードクローン解析

2.1　コードクローンの定義 [10]

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 と β は互いにクローンであるという．またペア（α，β）をクローンペアと呼ぶ．α，β それぞれを真に包含する如何なるトークン列も等価でないとき，α，β 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クラスと呼ぶ．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　コードクローン分析環境 Gemini

文献 [20] において我々はコードクローン分析環境 Gemini を開発した．図 1 はシステムのアーキテクチャを示している．Gemini は内部の CCFinder にソースコードを渡し，CCFinder の解析結果をさまざまなユーザインターフェースを通してユーザに提供する機能を有する．

本章では，簡単に Gemini と CCFinder の特徴を説明する．

2.2.1　CCFinder

CCFinder 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

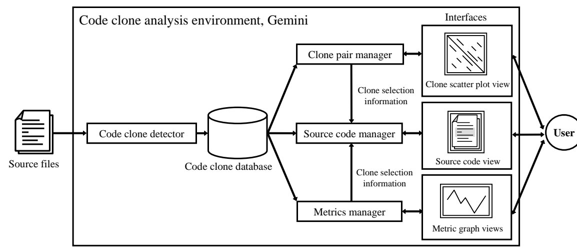

図 1: Gemini 全体図

ドクローンの最小トークン数はユーザが前もって設定することができる．

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）は大きく 4 つの過程から成り立っている．

ステップ 1（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．
ステップ 2（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．
ステップ 3（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．
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．

CCFinder の詳細については文献 [12] を参照されたい．

2.2.2 Gemini

Gemini は GUI ベースのコードクローン分析環境であり，内部的にコードクローン検出部として CCFinder を用いている．Gemini はユーザに以下のユーザインターフェースを提供し，対話的な解析を可能としている．

- クローン散布図,
- メトリクスグラフ,

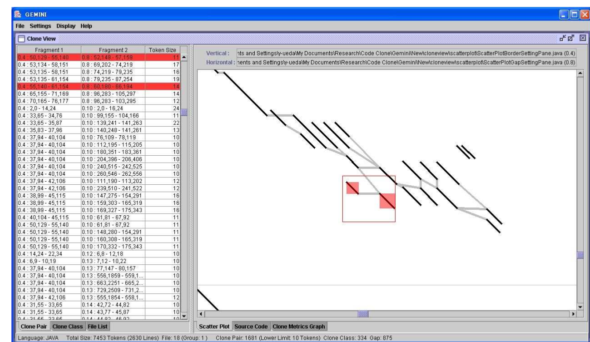

図 2: クローン散布図表示例

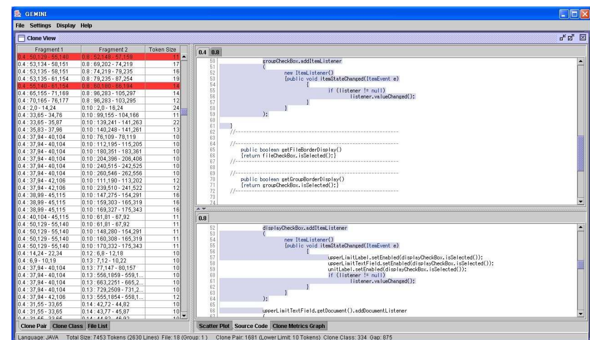

図 3: ソースコードビュー表示例

- ソースコードビュー．

クローン散布図はソースコードのどの部分にクローンペアが存在するのかを示す図である．一目でソースコード中のコードクローンの分布状況がわかるので，コードクローン解析の初期段階では非常に有効な解析手段となりうる．図上でユーザはマウスを用いて任意のクローンペアを選択することが可能であり，その例を図 2 に示す．クローン散布図の詳細については後ほど論ずる．

またメトリクスグラフを用いることにより，ユーザはコードクローンを定量的な特性に基づいて選択することができる．それぞれのクローンクラスについて複数のメトリクス値が示されているので，ユーザは長いコードクローンや，出現数の多いコードクローンを選択できる．

ソースコードビューはクローン散布図やメトリクスグラフと組み合わせて用いられる．ユーザはクローン散布図やメトリクスグラフで選択されたクローンのソースコードをソースコードビューを用いることにより閲覧することができる．図 3 では，図 2 において選択されたコードクローンのソース

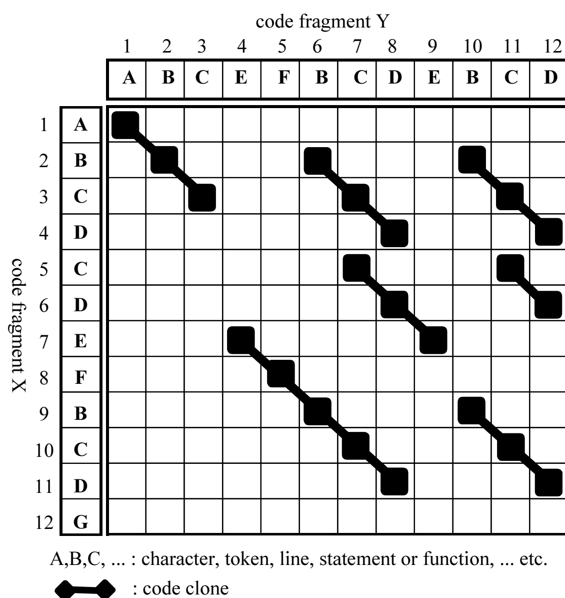

図 4: クローン散布図モデル

コードを表示している．

2.2.3 クローン散布図

図4はクローン散布図の例を示している．クローン散布図の縦軸と横軸にはソースコード中のトークンが出現順に配置される．ここではクローン散布図を説明するために以下の文字列を用いる．

コード片 X: “ABCDCDEFBCDG”,
コード片 Y: “ABCEFBCDEBCD”

ここでは，“A” や “B” などは文字や，トークン，行，文などのある一定の単位を表すとする．図4の格子内の黒色の矩形はその縦軸の要素と横軸の要素が等しいことを意味している．このことからクローンペアはクローン散布図においてある一定以上の長さを持った線分として出現することとなる．もし縦軸と横軸に配置される要素が同じファイルである場合は，主対角線上に黒色の矩形がプロットされる．またこの時はこの対角線に対してクローン散布図は線対称となる．

3 提案手法

3.1 これまでの問題点

我々は，Gemini（と CCFinder）をさまざまな商用，及び非商用ソフトウェアに適用してきた．そしてフィードバックとしていくつかの問題点が指摘された．その一つはコードクローンを用いたリファクタリングにおける問題であった．

検出したコードクローンをリファクタリング [7] に用いる場合の問題点として，単に最大長のクローンを検出してもその部分を一つの関数，メソッドなどにまとめることは難しい．我々はこれまでに多くの実験を行なってきたが，その実験で検出された大部分のコードクローンは，特に意味的なまとまりを持たないものであった．図5に例を示す．図5では，A と B の二つのコード片が示されている．A と B それぞれの灰色の部分は，その部分が A と B の間の最大長のコードクローンであることを示している．コード片 A ではいくつかのデータがリスト構造の先頭から順に連続して格納されている．一方コード片 B では，リスト構造の後方から順に連続してデータが格納されている．これら二つのコード片間には共にリスト構造を扱っているという点において論理的に共通している．しかしながらこれらのコード片の for 文の前後には偶然クローンとして含まれてしまった部分が存在している．リファクタリングの視点からは，灰色の部分全体よりも for 文のみをコードクローンとして抽出する方が望ましい．

コードクローンをリファクタリングするための研究はいくつか行なわれている．文献 [13] と文献 [14] では，プログラム依存グラフを用いてコードクローンの凝集度を測定し，関数やメソッドの抽出に用いる方法が述べられている．しかしそれらの方法では，プログラム依存グラフの構築に非常にコストがかかることから大規模なソフトウェアへの適用が難しく，スケーラビリティの面から見て問題がある．一方 CCFinder では，検出過程において字句解析のみにとどめているため，その検出処理は非常に高速である．しかし，CCFinder によって検出されるコードクローンは単に最大一致トークン数によるものであり，コードクローンの凝集度は考慮されていない．つまり CCFinder を用いたコードクローン検出をリファクタリングに適用しようとした場合，ユーザは自ら CCFinder の検出結果から意味的なまとまりのある部分を抽出する必要がある．

この問題を解決するため，我々はまず最大長のコードクローンを検出し，さらにそのクローンに

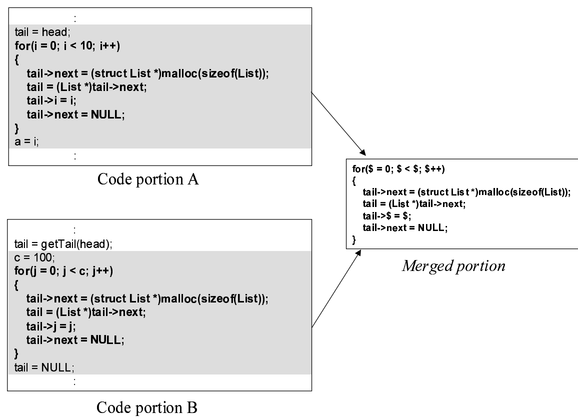

図 5: コードクローンのリファクタリング例

対して意味的なまとまりのある部分のみを抽出するという二段階のアプローチを提案する．この提案手法では，例え対象が大規模なソフトウェアであっても実用的な時間でリファクタリングの容易なコードクローンを検出することができる．

3.2 アプローチ

CCFinder によって検出されたコードクローンから一つの関数やメソッドとしてまとめるのに適した部分のみを取り出したコードクローンを**シェイプドクローン**と定義する．ソフトウェアからシェイプドクローンを取り出す過程は以下の三つからなる．

STEP1: 対象のソフトウェアに対し CCFinder を実行し，コードクローンを検出する．
STEP2: 対象のソフトウェアを解析し，関数や for 文，if 文などの意味的なまとまりをもったブロックの位置情報を抽出する．
STEP3: コードクローンの情報と，ブロックの位置情報を突き合わせ，意味的なまとまりのある部分のみを抽出する．
ここで,「意味的にまとまりのある」とは，直観的には一つの関数などにまとめるのが容易にできるような部分を指している．

3.3 実装

ソースコードからシェイプドクローンを取り出す機能を Gemini に実装した（図 6 参照）．この部分のプログラムは約 4000 行であり C++によって実装されている．現在のところ，対象言語は Java，C，

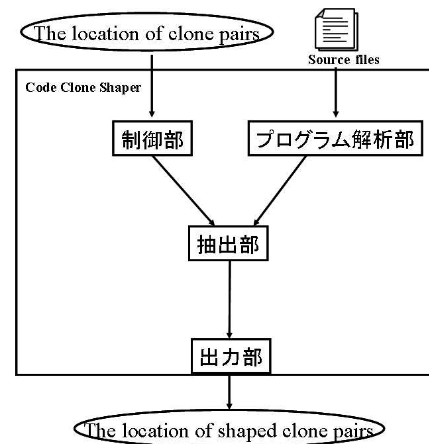

図 6: Code Clone Shaper 概要

C++の三種類である．ここでは，実装したシェイプドクローン検出プログラム（Code Clone Shaper 以後 CCShaper と略す）について説明する．実装したプログラムは図 6 にも示されているように大きく分けて四つのモジュールから構成されている．

- 制御部
- プログラム解析部
- 抽出部
- 出力部

制御部

制御部は CCFinder の出力したコードクローン情報を解析し，字句解析部や，ブロック抽出部，ブロック管理部を呼び出す役割を担っている．

プログラム解析部

プログラム解析部では対象のソースコードを字句解析，構文解析して，プログラム中のブロックの位置情報を取得する．ここではブロックを中括弧 (‘{’,‘}’) で囲まれた範囲と定義する．現段階では字句解析によって抽出された情報しか用いておらず，構文解析は今後の解析で用いる予定である．

抽出部

プログラム解析部で抽出したブロックの位置情報と，CCFinder の出力結果であるコードクローン情報を突き合わせ，コードクローンである部分から意味的にまとまりのある部分のみを抽出する．

表 1: ソースコードサイズ

	ファイル数	行数	トークン数
Ant	508	141254	221203

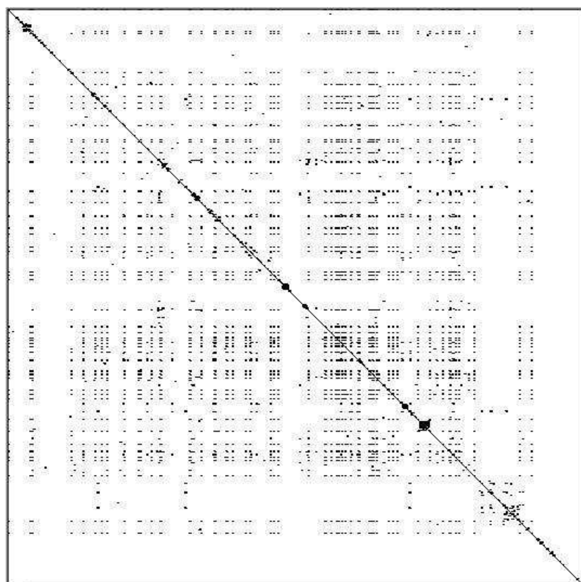

図 7: Ant のクローン散布図（CCShaper 無）

出力部

ブロック抽出部で取り出した意味的にまとまりのあるコードクローンを適切な順番で並び変え，Gemini に与えるデータとして一貫性のあるものに変換する．

4　適用実験

4.1　実験概要

前章で提案したシェイプドクローンの検出方法の有用性を確認するためにオープンソースの Java ソフトウェアである Ant[1] に対して適用実験を行なった．Ant は Java 用のビルドツールである．ビルド手順は XML で記述される．

提案手法の評価をするために，CCShaper を用いた Gemini と用いていない Gemini を用いてそれぞれ対象のソースファイルを解析した．そしてその後結果を比較した．この適用実験では，CCFinder が検出するコードクローンの最小一致トークン数は 50 に設定した．

4.2　Ant

次に Ant に対しての実験結果を示す．Ant は 508 個のファイルから成り立っており，総行数は 14 万 1 千行であった．

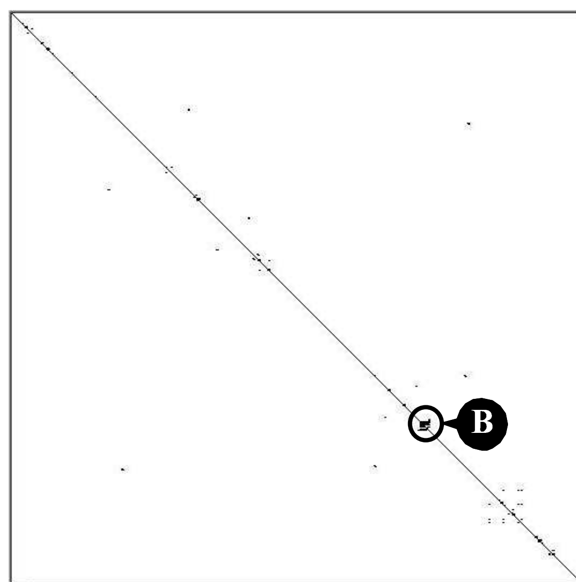

図 8: Ant のクローン散布図（CCShaper 有）

Code Clone Shaper を適用する前、適用後のクローン散布図をそれぞれ図 7、図 8 に示す．また、適用前後のコードクローン数の比較を表 2 に示す．CCShaper を用いない場合には、Ant 内には一様にコードクローンが散布しており，どのコードクローンを取り除くべきか、取り除くことができるかを判断するのに、非常に手間がかかってしまうと考えられる．一方で、CCShaper を用いた場合には、リファクタリングが難しいコードクローンを取り除くことにより、クローンペアの 99 %以上（1 万個以上)、クローンクラスの約 94 %（803 個）を省くことができている．

実際に検出されたコードクローンのソースコードを調べるために，最もクローンペアが密集しているラベル B で示す部分をソースコードビューで閲覧する．9 はこの部分のコードクローンのソースコードを示している．このメソッド単位のコードクローンは 7 つのクラスに，各々一つずつ存在した．またこれら 7 つのクラスは全て同一のクラスを親クラスとして定義していた．これらのことからこのコードクローンは親クラスに引き上げる

表 2: Ant から検出されたコードクローン数の比較

	CCShaper 有	CCShaper 無
クローンペア数	12033	103
クローンクラス数	856	53

```
public void getAutoresponse(Commandline cmd) {
  if (m_AutoResponse == null) {
    cmd.createArgument().setValue(FLAG_AUTORESPONSE_DEF);
  } else if (m_AutoResponse.equalsIgnoreCase("Y")) {
    cmd.createArgument().setValue(FLAG_AUTORESPONSE_YES);
  } else if (m_AutoResponse.equalsIgnoreCase("N")) {
    cmd.createArgument().setValue(FLAG_AUTORESPONSE_NO);
  } else {
    cmd.createArgument().setValue(FLAG_AUTORESPONSE_DEF);
  } // end of else
}
```

図 9: ラベル B のコードクローンのソースコード

ことによって取り除くことができる．

5 まとめ

本論文では，既存のコードクローン分析環境 Gemini を拡張してリファクタリングの候補となりうるコードクローンのみを抽出する機能（Code Clone Shaper）を提案した．そして提案手法をツールとして実装し，オープンソースのソフトウェアである Ant に適用した．Code Clone Shaper を適用することでコードクローンをフィルタリングすることに成功し，抽出されたコードクローンはいずれもリファクタリングの容易なものであった．これらのことから本論文で提案したコードクローン抽出手法は非常に有用であると考えることができる．

また，今後はこのツールを実際のソフトウェア保守の場面で利用し，そのプロセス改善に役立てたいと考えている．

参考文献

[1] Ant, `http://jakarta.apache.org/ant/`, 2002.

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

[3] B. S. Baker, *On Finding Duplication and Near-Duplication in Large Software Systems*, IN Proc. IEEE Working Conf. on Reverse Engineering, pages 86-95, July 1995.

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

[5] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, *Clone Detection Using Abstract Syntax Trees*, Proc. IEEE Int'l Conf. on Software Maintenance (ICSM) '98, pages 368-377, Bethesda, Maryland, Nov. 1998.

[6] 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, pages 109-118, Oxford, England, Aug. 1999.

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

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

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

[10] 井上克郎, 神谷年洋, 楠本真二, "コードクローン検出法", **コンピュータソフトウェア**, vol.18, no.5, pp.47-54, 2001.

[11] J. H. Johnson, *Identifying Redundancy in Source Code using Fingerprints*, Proc. of CASCON '93, pages 171-183, Toronto, Ontario, 1993.

[12] 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, (to appear).

[13] R. Komondoor and S. Horwitz, *Using slicing to identify duplication in source code*, In Proc. of the 8th International Symposium on Static Analysis, Paris, France, July 16-18, 2001.

[14] Jens Krinke, *Identifying Similar Code with Program Dependence Graphs* , In Proc. of the 8th Working Conference on Reverse Engineering, 2001.

[15] J. Mayland, C. Leblanc, and E. M. Merlo *Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics*, Proc. of IEEE Int'l Conf. on Software Maintenance (ICSM) '96, pages 244-253, Monterey, California, Nov. 1996.

[16] L. Prechelt, G. Malpohl, M. Philippsen, *Finding plagiarisms among a set of programs with JPlag*, submitted to Journal of Universal Computer Science, Nov. 2001, taken from `http://wwwipd.ira.uka.de/~prechelt/Biblio/`

[17] M. Rieger, S. Ducasse, *Visual Detection of Duplicated Code*, 1998.

[18] Duploc, `http://www.iam.unibe.ch/~rieger/duploc/`, 1999.

[19] Pigoski T. M, *Maintenance*, Encyclopedia of Software Engineering, 1, John Wiley & Sons, 1994.

[20] Y. Ueda, T. Kamiya, S. Kusumoto, K. Inoue, *Gemini: Maintenance Support Environment Based on Code Clone Analysis*, 8th International Symposium on Software Metrics, June 4-7, 2002.

[21] Y. Ueda, T. Kamiya, S. Kusumoto, K. Inoue, *On Detection of Gapped Code Clones using Gap Locations*, Submitted to 9th Asia-Pacific Software Engineering Conference, 2002.

[22] S. W. L. Yip and T. Lam, *A software maintenance survey*, Proc. of APSEC '94, pages 70-79, 1994.

[23] E. J. Wegman and Q. Luo, *High Dimensional Clustering Using Parallel Coordinates and the Grand Tour*, Proc. 28th Symp. Interface of Computing Science and Statistics, 1996.