記事の種類：解説
表題：
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
著者，勤務先：
神谷 年洋
科学技術振興機構さきがけ「機能と構成」領域研究者
正員（入会申請中）
〒560-8531
大阪府豊中市待兼山町１番３号
大阪大学　大学院情報科学研究科 井上研究室

kamiya@ist.osaka-u.ac.jp

Title:
What Code Clone is, What Problem It Causes, and How to Cope With It
Author:
Toshihiro Kamiya
Presto, Japan Science and Technology Agency
Software Engineering Laboratory, Department of Computer Science
Graduate School of Information Science and Technology, Osaka University
1-3, Machikaneyama-cho, Toyonaka, Osaka, 560-8531, Japan

1. はじめに

この記事では，コードクローンとは何であるか，どのように問題であるのか，どのような対策があるのかを解説する．基本的には，コードクローンとはソースコードの中で全く同じかあるいは類似している部分である．コードクローンは，典型的には，開発者がソースコードをテキストエディタでコピー＆ペーストすることによって生じ，コードクローンが含まれるとソフトウェアの保守が困難になるといわれている．本記事では，これまでにコードクローンを検出するために提案されている手法について述べ，さらに，ソフトウェアのライフサイクルを通じて，コードクローンが発生するさまざまな原因があることを説明する．

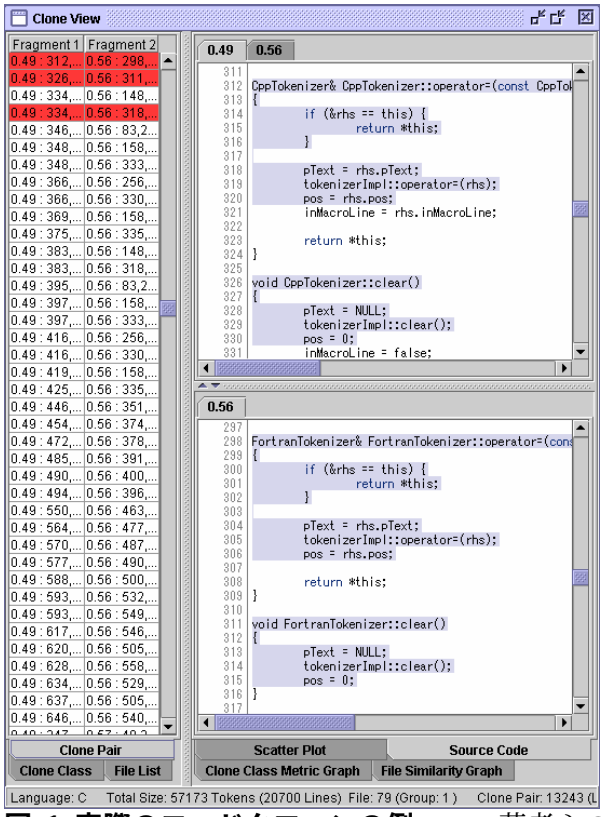

図 1 実際のコードクローンの例　著者らの研究グループで開発されたコードクローン検出ツール[36]により，コードクローンを検出したところ．上下に分かれたウィンドウに表示されているソースコードの中で，ハイライトで表示された部分がコードクローンになっている．

2. コードクローンとは

コードクローンとは，ソースコードの中で全く同じかまたは類似している部分である（以降，混乱のおそれがない限り「クローン」とも呼ぶ）．クローンが発生する典型的な理由は，開発者がソースコードを書く（コーディングする）ときに，ソースコードをテキストエディタでコピー＆ペーストすることである．図 1 に実際のクローンの例を示す．

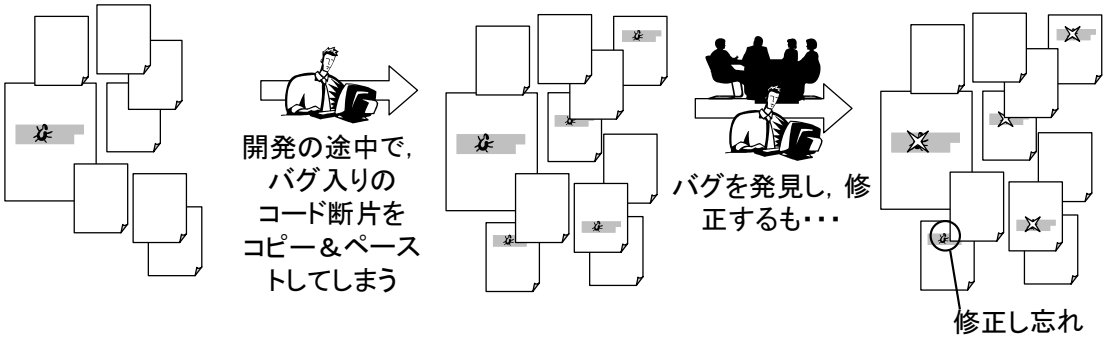

図 2 コードクローンが引き起こす問題　数百万行のソースコードからなる大規模なソフトウェアの場合，コードクローンが含まれていても，そのコード断片を手作業で見つけ出すのは困難である．

クローンがソフトウェアの保守作業において問題になるのは，クローンを持つソースコードは修正が困難なためである(“Programs that have duplicated logic are hard to modify” [13])．例えば，ソースコード中の間違い（バグ）を発見して修正する場合に，もし修正すべき部分がクローンのコード断片になっていれば，そのすべてのコード断片について，修正を行う必要がある（図 2)．これは，バグの修正ばかりではなく，ソフトウェアの機能拡張や移植など，ソースコードの修正を伴うすべての作業について当てはまる．作り込まれたクローンは，数ヶ月もするとその存在が忘れられてしまい，特に大規模なソースコードであれば，容易には発見できなくなる．そのため，ソフトウェアを修正するときになって，クローンのコード断片のいくつかを修正し損ない，「修正したはずのバグが特定の条件下で再現する」といった事態を引き起こす．このような理由により，一般には，ソースコード中のクローンは少ない方がよいとされている．

クローンへの抜本的な対策としては，クローンを作り込まないように予防すること，および，クローンを除去することが挙げられる．予防する手段としては，設計段階で類似した機能を持つモジュールをあらかじめ一覧にしておき，ソースコードを書くときにその一覧を参考にするという方法が提案されている[11]．一方，後者に関しては，クローンの除去を行うために，特に大規模なソフトウェアを対象としたクローン検出ツールが開発されてきている．

3. コードクローン検出

3.1. コードクローン検出技術

クローンを検出するためには，どのような一致または類似を検出したいのかを明確にする必要がある．例えば，開発者は様々なソースコード断片をコピー＆ペーストするが，このうち，定数の初期化テーブルなどは，たとえクローンであっても検出したくない場合が多い．また，開発者は，コード断片をコピー＆ペーストした後，何らかの修正を加えることも多く，結果として，クローンのコード断片は少しずつ異なった字面をもつ．クローン検出手法の多くは，このような違いをある程度許容するようなアルゴリズムを用いることで，通常の文字列検索よりも精度良くクローンを検出することができる．

コードクローン検出技術にすこし似ていて，よりよく知られているものとして，遺伝子の DNA の配列を比較する技術がある[1][21]．DNA の配列は４つのアルファベット A, T, C, G から構成される文字列であり，DNA 配列の比較技術においてももちろん，完全な一致やある程度の違いを許容した一致を検出することができる．この技術により，DNA を手がかりに生物の系統樹を作ったり（遺伝生物学)，特定の DNA の配列がどんな機能を持っているか類推したりする．発見的手法として，DNA のアルファベット３つで一つのタンパク質の合成を指示することを利用して，アルファベットの区切りをずらしながら比較したり，DNA の配列により合成されるタンパク質の列での比較を行うといった手法が用いられている．これまでに提案されているコードクローン検出技術においても同様に，ソースコードの構造を利用することで，人間が見て類似していると思うであろうソースコード断片をコードクローンとして検出することを目指している．

3.2. コードクローン検出の目的

クローンを検出する目的としては，クローンの除去，クローンのチェック，ソースコードの評価，ソースコードの比較といったものがあり，一概にクローン検出といっても，それぞれの目的によって少しずつ異なった検出方法が望まれる．本章では，これらの目的について説明し，それぞれの目的に照らした研究の現状について述べる．

コードクローンの除去

ソースコードを修正し，クローンを除去する．クローンを除去するためには，クローンになっている部分を人手で修正する[6]（リファクタリング[13]と呼ばれる，ソフトウェアの機能を修正することなくソースコードを整理するプロセスを実施する）か，あるいは，ソースコードを自動的に修正するツールを利用する．このような，クローンを検出し，クローンになっている部分を共通のルーチンとしてまとめ，個々のコード断片をその共通ルーチンへの呼び出しに置き換えるようなツールを開発することは，クローン研究が始まった当初からの目標であるが，現状でめざましい進歩はないようである．技術的に困難な点として，共通ルーチンを作成するために必要とされる，ソースコード修正技術が挙げられる．実際のソースコードに適用することを考えた場合，プログラミング言語には方言を持つものもあり，それらのサポートも考慮しなければならない．また，プログラミング言語は現在も進化しつつあり，オブジェクトや汎用型，アスペクトといった技術が導入されているため，共通ルーチンを生成する際にも，ソフトウェアの設計の意図に応じた適切な技術の選択[10]が望まれる．

クローンの除去から派生した目的として，ソフトウェア部品の抽出や改良にクローン検出を用いようとするアプローチ[33]もある．例えば，２つの部品が組み合わせて用いるようなクローンが多数検出されるなら，それらをまとめた部品を提供する，あるいは，ある部品を利用する場合にいつも必要とされるコード断片があるなら，そのコード断片を部品に取り込んで，新たなインターフェイスを提供する，といった具合である．

コードクローンのチェック

バグ修正やソフトウェアの機能変更のためにソースコードを修正する際，修正しようとしている部分にクローンがあるかを調べ，それらのコード断片に同じ修正を加える．前述の「コードクローンの除去」に比べると対処療法的な感があるが，実用上はきわめて重要である．特に，クローンがあっても何らかの理由（5 章で後述）により共通ルーチンに書き換えられない場合には，これが唯一の対策となる．

この目的のためにクローンを検出する際には，除去の場合とは異なり，コピー＆ペースト後に少し修正されて一部動作が異なるようなコード断片であっても，クローンとして検出することが求められる．また，クローンではないコード断片を検出してしまう誤りより，検出漏れの方が大きな問題となるため，クローンを検出する際の同一性，類似性の判定基準は除去の場合より緩いものが用いられる．

コードクローンによるソースコードの評価

クローンが多く含まれるソフトウェアは修正が困難であるため，保守作業でバグを作り込んでしまう可能性も高いと考えられる．従って，クローンの含有率等をソフトウェアの複雑度メトリクスとして用いる，というアプローチである[16]．含有率が高い部分はリファクタリングを適用すべき候補とする，あるいは，クローンを含むルーチンを捨てて新たに書き直すべきといった判断を下す．ケーススタディ[29]では，ある事務処理ソフトウェアについて，長いコード断片を持つクローンが含まれるモジュールは，そうではないモジュールよりも頻繁に改版されることが観察されている．

コードクローンによるソースコードの比較

複数のソフトウェア，あるいは一つのソフトウェアの複数のバージョンについて，それらの間に含まれるクローンを調べる．これにより，ソースコードの変遷[19]や系統樹[35]，剽窃を調べる[2][32]．ソースコードは著作権によって保護される対象でもあるため，これらはクローン検出の重要な応用の一つである．ただし，クローン検出の結果だけから，ソースコードの剽窃を立証することは不可能である．例えば，最近，Linux[26]が自社の著作権を侵害していると主張して耳目を集めた SCO の事例では，該当するソースコード（の少なくとも一部）は，彼らの主張とは逆に，最初オープンソースとして開発されたソースコードを，商用ソフトウェアに取り込んだものであることが判明した．ソースコードの剽窃を立証するには，クローン検出だけではなく，オリジナルの出所もはっきりさせる必要がある．

図 3 は著者らのグループで開発したコードクローン検出ツール CCFinder[20]を用いて，３つのオペレーティングシステム FreeBSD[14]，Linux[26]，NetBSD[30]のソースコードからクローンを検出した結果の散布図である．おおざっぱには，グラフの軸はソースファイルの行，それぞれの点がクローンの場所を表している（散布図の詳しい利用法は文献[9][36]を参照されたい）．左上の原点から伸びる対角線上の３つの正方形にはそれぞれのオペレーティングシステム内のクローンが，それ以外の長方形にはオペレーティングシステム間のクローンが示されている．点の分布から，それぞれのオペレーティングシステム内には数多くのクローンがあることがわかる．また，FreeBSD と NetBSD の間にはクローンが数多く検出され，これら二者と Linux との間にはクローンはほとんど無いことがわかる．これは，FreeBSD と NetBSD が元々同じソースコードから分岐してできており，Linux が新たに書き起こされたソースコードでできているという経緯をよく説明する．

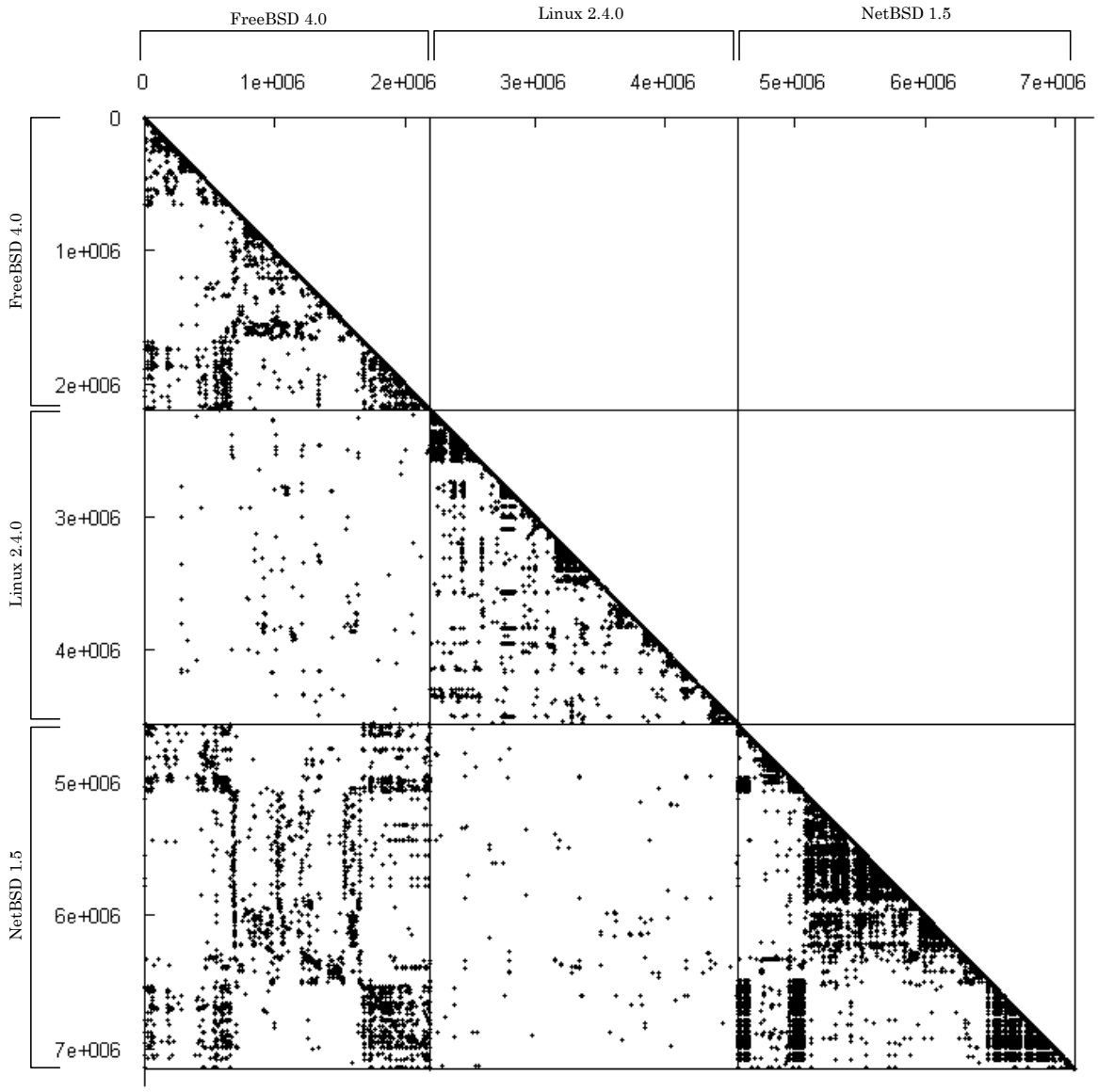

図 3 三種類のオペレーティングシステムから検出されたコードクローンの散布図　両軸の目盛りはソースコードの行数である．垂直および水平の実線は，それぞれのオペレーティングシステムの境界を示す．点はコードクローンの位置を示す．対角線を対称軸として線対称になるため，右上半分の点は省略されている．

4. コードクローン検出手法・ツール

3 章で述べた目的，すなわち，クローンの除去や品質評価といった目的のために，クローンを検出する手法・ツールがこれまでに提案されてきている[3][4][6][7][8][12][17][18][20][22][23][24][25][27][28][33]（剽窃検出に目的を絞ったツール[2][32]もある）．一般にクローン検出ツールの入力はソースコードである．ソースコードはプログラミング言語の構文や意味に従って記述されるため，その構文や意味に応じた構造を持つ．プログラミング言語によっても異なるが，これらの構造は階層を持ち，微視的なものから順に，文字，トークン，式や文，宣言や定義，パッケージ等となる．これらの階層に応じて，ツール内部でのソースコードの表現としては，文字列，トークンの並び，AST（抽象構文木）といったデータ構造が用いられる．さらに意味解析やそれに続く解析によって，スコープ（名前の有効範囲．スコープが異なると同じ名前で違う実体を参照することになる）や，制御依存やデータ依存といった構造を抽出する（図 4）．

それぞれのクローン検出手法は，これらソースコードを表現する構造の一つあるいは複数について，一定の判定基準に従って，同一性，あるいは類似性を判定する．例えば，ソースコードの表現として文字列を用いた場合であれば「文字の並びが同じであるような行が３０行以上続けばクローンと見なす」，あるいは，ソースコードの表現として，ルーチン（C 言語の関数や Java のメソッド）の特徴メトリクス（ルーチンの入力，出力，制御の流れの複雑度）を用いた場合には，「ルーチンの入力の数の差が 0

または 1 つであり，それ以外が同じであるときはクローンと見なす」，といった具合である．図 5 に，これまでに提案されているクローン検出手法が比較対象としている構造と比較アルゴリズム[5][15]をまとめたものを示す．いくつかのツールは複数の構造やアルゴリズムを併用している．

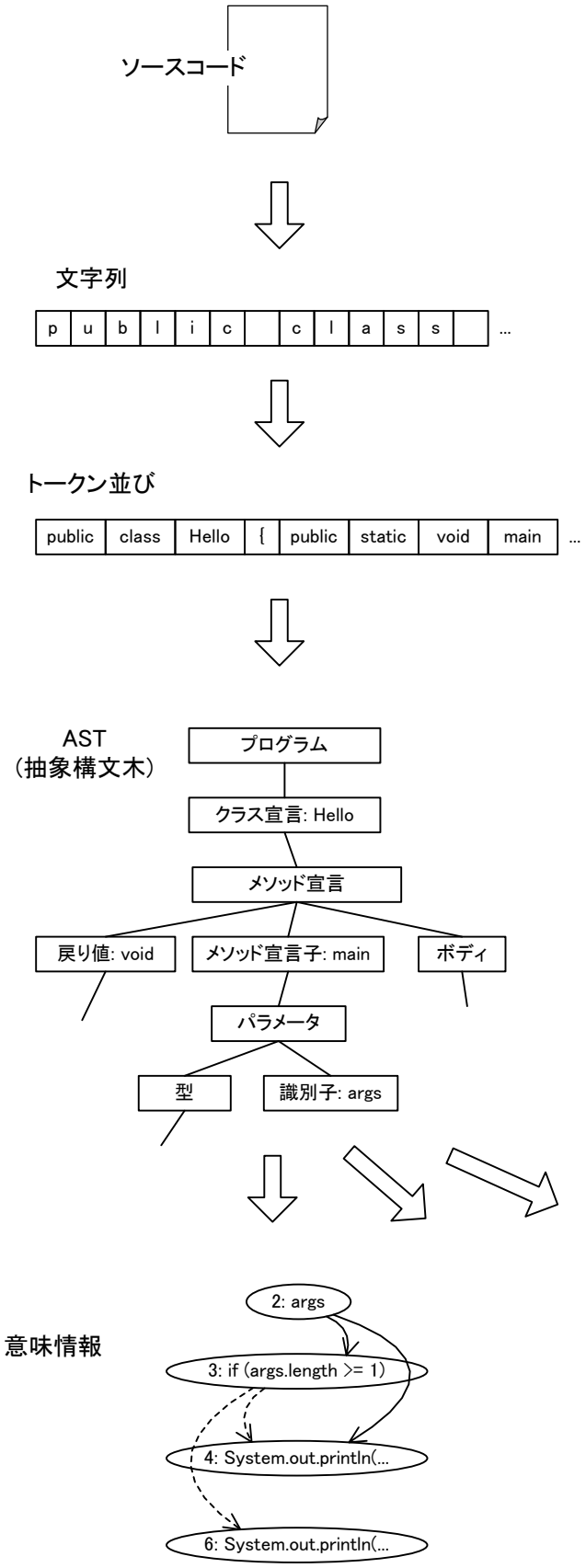

図 4 コンパイラなどのツールがソースコードを解析する順序

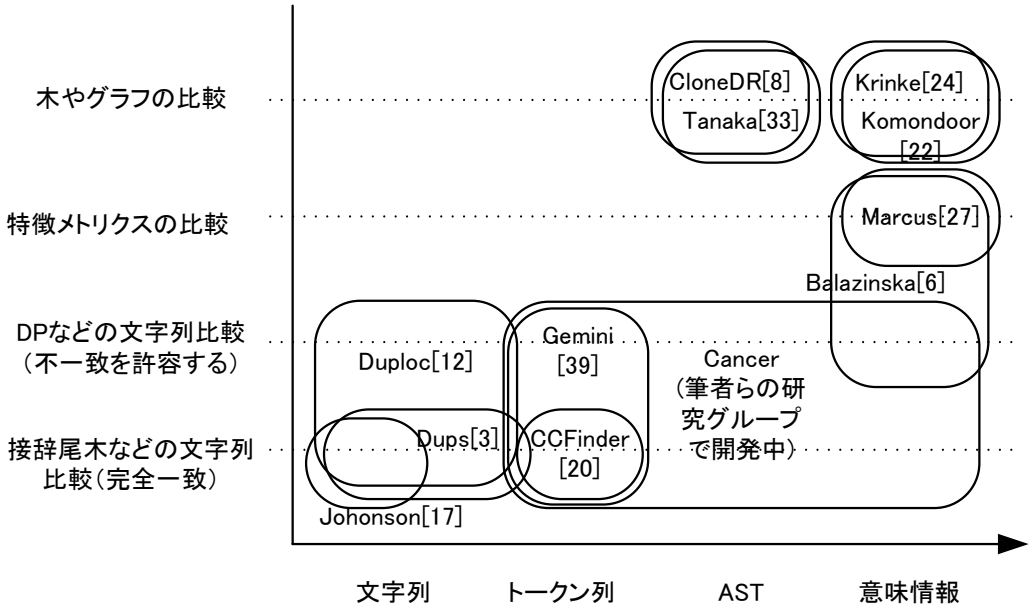

図 5 コードクローン検出ツールが扱う構造とアルゴリズム　横軸には構造が粒度の小さいものから順に並んでいる．縦軸は一致を判断するためのアルゴリズム．いくつかのツールは複数の粒度やアルゴリズムを併用している．

5. コードクローンが発生する理由

フトウェアのライフサイクルを順に見ていくと，以下のような原因によりクローンが発生すると考えられる．

(1)再利用のためのコピー＆ペースト

一般に，最初から再利用を前提としてソフトウェアを開発することは困難である[34]．再利用のための準備が整っていないソースコードを再利用するためには，その一部を修正して再利用することになる．

(2)意図的なコードクローンの作り込み

実行時の性能を稼ぐために，中身が小さくて繰り返し回数も少ないループを展開するなどの手段により，意図的にクローンを作り込むことがある．

(3)定型コード

先に, クローンとして検出したくないコードとして初期化テーブルを挙げた. このようなコードは，クローンであることがわかっても，うまくまとめる方法がないことがある．他の例としては，特定のオブジェクトを初期化するためのメソッド呼び出しの列（setter 呼び出しの列），データを一定の書式に従って印字するルーチン等が挙げられる．

(4)ソースコード生成ツール

ソースコード生成ツールによって生成されたソースコードには，しばしば多くのクローンが含まれる．このようなクローンは，何らかの理由によりコード生成器が使えなくなったり，生成されたソースコードと元の記述（ソースコード生成ツールに与えた入力）との対応が失われれば，直ちに問題となる．

(5)「動いているコードには触るな」方針

ソフトウェアの保守段階でクローンが作り込まれてしまうこともある．開発組織の中には，運用に入っているソフトウェアを修正して事故が起きることを避けるため，「動いているコードには触るな」[13]という方針を採っているものがある．このような方針の下で保守・運用されているソフトウェアは，時間がたつにつれてクローンが単調に増加し，そのうち，ソースコードを修正することが事実上不可能になってしまう．

(6)プログラミング言語の制約

プログラミング言語の制約により，例えば，昔の COBOL(COBOL74)[31]では，変数のスコープは大域変数のみであるために，クローンを作らざるを得ない場合がある．現在のオブジェクト指向 COBOL (COBOL2000)等を用いて書き直せば，共通ルーチンとすることができる．

このように，クローンが作り込まれてしまう原因は様々であり，単に開発者の責任とすることはできな

い．クローンへの対策を開発者だけに求めるのでは不十分であり，管理者も交え，ソフトウェアのライフサイクル全体を通じて行うべきである．

6. まとめと今後の展望

本稿では，コードクローンは開発者がソースコードをコピー＆ペーストをはじめ様々な原因により作り込まれること，コードクローンがソフトウェアの保守プロセスで問題になることを述べた．コードクローン対策のための基本的な手段である，いくつかのコードクローン検出手法・ツールを紹介した．また，コードクローンがソフトウェア開発プロセス中で作り込まれるさまざまな状況を説明した．

現状で，コードクローン検出のための基本的なアプローチはほぼ出揃った感がある（もちろん，著者には想像もつかない手法が登場する可能性もなしとはしない）．コードクローン検出ツールを比較する論文も現れ始めた．ただし，大規模ソフトウェアに適用可能なコードクローン検出ツールはまだ数種類しか登場しておらず，これらのツールも実用上はまだまだ改良の余地がある．これからも，コードクローンを検出する目的に応じてパラメータを調整したり，新たな発見的手法を取り入れるといった改良が続くだろう．また，ソースコードを修正するための技術が発達してくれば，コードクローンの検出と同時にその除去も自動的に行うツールが実現されるだろう．コードクローン検出ツールをすぐにでも使いたいという方は，筆者らの研究グループをはじめとして，いくつかの研究グループが配布している検出ツールを入手され，利用されると良い．また，コードクローン検出手法についてさらに詳しく知りたい方は，参考文献に当たられると良い．

コードクローン検出手法は，大きくは「たくさんのものの中から同じものを見つけ出す」手法の仲間であり，これまでに提案されたたくさんの比較アルゴリズムの応用であるとともに，ソフトウェアを比較するという，ソフトウェア開発プロセスのさまざまな場面で用いられる基礎的な技術でもある．実用面だけではなく，ソフトウェアの比較が必要な研究における手段としても用いられていくことだろう．

文献

[1] S.F. Altschul, W. Gish, W. Miller, E. Myers, D. Lipman, “Basic Local Alignment Tool,” *Journal of Molecular Biology*, 215, pp. 403-410, (1990).

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

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

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

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

[6] M. Balazinska, E. Merlo, M. Dagenais, B. Lagüe, and K.A. Kontogiannis, “Measuring Clone Based Reengineering Opportunities,” *Proc. 6th IEEE Int’l Symposium on Software Metrics* (METRICS ’99), pp. 292-303, Boca Raton, Florida, Nov. 1999.

[7] M. Balazinska, E. Merlo, M. Dagenais, B. Lagüe, and K.A. Kontogiannis, “Partial Redesign of Java Software Systems Based on Clone Analysis,” *Proc. 6th IEEE Working Conference on Reverse Engineering* (WCRE ’99), pp. 326-336, Atlanta, Georgia, Oct. 1999.

[8] I.D. Baxter, A. Yahin, L. Moura, M. Sant’Anna, and L. Bier, “Clone Detection Using Abstract Syntax Trees,” *Proc. IEEE International Conference on Software Maintenance* (ICSM) ’98, pp. 368-377, Bethesda, Maryland, Nov. 1998.

[9] K.W., Church, and J.I. Helfman, “Dotplot: A Program for Exploring Self-Similarity in Millions of Lines of Text and Code,” *Journal of Computational and Graphical Statistics*, V2, N2, pp. 153-174 (1993).

[10] James O. Coplien, *Multi-Paradigm Design for C++*, Pearson Education (2001). ジェームス・O・コプリン(著), 金沢 典子, 平鍋 健児, 羽生田 栄一(訳), マルチパラダイムデザイン, ピアソン・エデュケーション (2001).

[11] Elizabeth Burd, John Bailey, “Evaluating Clone Detection Tools for Use during Preventative Maintenance,” *Proc. 2nd IEEE International Workshop on Source Code Analysis and Manipulation* (SCAM) 2002, pp. 36-43. Montreal, Canada, Oct. 2002.

[12] S. Ducasse, M. Rieger, and S. Demeyer, “A Language Independent Approach for Detecting Duplicated Code,” *Proc. IEEE International Conference on Software Maintenance* (ICSM) ’99,

pp. 109-118. Oxford, England. Aug. 1999.
[13] M. Fowler, *Refactoring: improving the design of existing code*, Addison Wesley, 1999.
[14] FreeBSD. http://www.freebsd.org/.
[15] D. Gusfield, *Algorithms on Strings, Trees, and Sequences*, pp. 89-180. Cambridge University Press, 1997.
[16] T. Imai, Y. Kataoka, and T. Fukaya, "Evaluating Software Maintenance Cost Using Functional Redundancy Metrics," *Proc. 26th International Computer Software and Applications Conference* (Compsac 2002), pp.299-306, 2002.
[17] J.H. Johnson, "Identifying Redundancy in Source Code Using Fingerprints," *Proc. IBM Centre for Advanced Studies Conference* (CAS CON) '93, pp. 171-183, Toronto, Ontario. Oct. 1993.
[18] J.H. Johnson, "Substring Matching for Clone Detection and Change Tracking," *Proc. IEEE Int'l Conf. on Software Maintenance* (ICSM) '94, pp. 120-126. Victoria, British Columbia, Canada. Sep. 1994.
[19] J.H. Johnson, "Using Textual Redundancy to Understand Change," *Proc. IBM Centre for Advanced Studies Conference (CASCON) '95*, pp. 34-40, Nov. 1999.
[20] 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, (2002-7).
[21] 金久 實 (編), ヒューマンゲノム計画, 共立出版 (1997).
[22] R. Komondoor and S. Horwitz, "Using Slicing to Identify Duplication in Source Code," *Proc. of the 8th International Symposium on Static Analysis(SAS), LNCS 2126*, pp. 40-56, Springer (2001).
[23] K.A. Kontogiannis, R. De Mori, E. Merlo, M. Galler, and M. Bernstein, "Pattern Matching Techniques for Clone Detection and Concept Detection," *Journal of Automated Software Engineering* Kluwer Academic Publishers, vol. 3, pp. 770-108, 1996.
[24] Jens Krinke, "Identifying Similar Code with Program Dependence Graphs," *Proc. of the 8th Working Conference on Reverse Engineering*, 2001.
[25] B. Laguë, E.M. Merlo, J. Mayrand, and J. Hudepohl, "Assessing the Benefits of Incorporating Function Clone Detection in a Development Process," *Proc. IEEE Int'l Conf. on Software Maintenance* (ICSM) '97, pp. 314-321, Bari, Italy. Oct. 1997.
[26] Linux Online. http://www.linux.org/.
[27] A. Marcus, and J.I., Maletic, "Identification of High-Level Concept Clones in Source Code," Proc. Automated Software Engineering (ASE'01), San Diego, pp. 107-114, Nov. 2001,
[28] J. Mayrand, C. Leblanc, and E. M. Merlo, "Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics," *Proc. IEEE Int'l Conference on Software Maintenance* (ICSM) '96, pp. 244-253, Monterey, California, Nov. 1996.
[29] 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. 2178-2188 (2003-8).
[30] NetBSD Project. http://www.netbsd.org/.
[31] 大駒 誠一, COBOL 入門 改訂版, 培風館 (1986).
[32] L. Prechelt, G. Malpohl, M. Philippsen, "JPlag: Finding Plagiarisms Among a Set of Programs," Technical Report, University of Karlsruhe, Department of Informatics, 2000.
[33] 田中 哲, "データマイニングを利用したプログラムの改善," 日本ソフトウェア科学会 第 7 回プログラミングおよび応用のシステムに関するワークショップ(SPA 2004), pp. 1-8, (2004-3).
[34] Will Tracz, *Confessions of a Used Program Salesman --- Institutionalizing Software Reuse* ---, Pearson Education (1995). W. トレイツ (著), 畑崎 隆雄, 林 雅弘, 鈴木 博之 (訳), ソフトウェア再利用の神話 --- ソフトウェア再利用の制度化に向けて ---, ピアソン・エデュケーション (2001).
[35] 山本 哲男, 松下 誠, 神谷 年洋, 井上 克郎, "ソフトウェアシステムの類似度とその計測ツール SMMT," 電子情報通信学会論文誌, vol. J85-D-I, no. 6, pp. 503-511. (2002-6).
[36] 植田 泰士, 神谷 年洋, 楠本 真二, 井上 克郎, "開発保守支援を目指したコードクローン分析環境," 電子情報通信学会論文誌 D-I, Vol. J86-D-I, No. 12, pp. 863-871 (2003-12).