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

門田暁人[†] 佐藤慎一[†,††]
神谷年洋[†††] 松本健一[†]

ソフトウェアに含まれるコードクローン（重複するコード列）は，ソフトウェアの構造を複雑にし，ソフトウェア品質に悪影響を与えるといわれている．しかし，コードクローンとソフトウェア品質の関係はこれまで定量的に明らかにされていない．本論文では，20年以上前に開発され，拡張COBOL言語で記述されたある大規模なレガシーソフトウェアを題材とし，代表的なソフトウェア品質である信頼性・保守性とコードクローンとの関係を定量的に分析した．信頼性の尺度として保守工程で発見された「フォールト数/LOC（Lines of Code）」を用い，保守性の尺度として「モジュールの改版数」を用いた．分析の結果，コードクローンを含むモジュール（clone-included モジュール）は含まないモジュール（non-clone モジュール）よりも信頼性（平均値）が約40%高いが，200行を超える大きさのコードクローンを含むモジュールは逆に信頼性が低いことが分かった．また，clone-included モジュールは non-clone モジュールよりも改版数（平均値）が約40%大きく（すなわち，改版のためにより多くの保守コストが費やされてきた），さらに，モジュールに含まれるコードクローンのサイズが大きいほど改版数がより大きい傾向にあることが分かった．

Analyzing the Quality of Legacy Software Based on Code Clone

Akito Monden,[†] Shin-ichi Sato,[†,††] Toshihiro Kamiya[†††]
and Ken-ichi Matsumoto[†]

Existing researches suggest that the code clone (duplicated code section) is one of the factors that degrades the design and structure of software and lowers the various quality attributes. However, the influence of code clones on software quality has not been quantitatively clarified yet. In this paper, we quantitatively analyzed the relation between code clones and the software reliability and maintainability of twenty years old software, which is written in an extended COBOL language. We used the number of faults per LOC (Lines of Code) as a reliability metric, and the revision number of modules as a maintainability metric. As a result, we found that modules having code clones (clone-included modules) are 40% more reliable than modules having no code clone (non-clone modules) on average. Nevertheless, the modules having very large code clones (more than 200 SLOC) are less reliable than non-clone modules. We also found that clone-included modules are 40% less maintainable (having greater revision number on average) than non-clone modules; and, modules having larger code clone are less maintainable than modules having smaller code clone.

1. はじめに

今日，多くの企業において，レガシーソフトウェア（Legacy Software：遺産的ソフトウェア）の品質低下による保守コストの増大が問題となっている[22),25)~27)]．レガシーソフトウェアとは，ユーザからの新たな要求による機能追加やデバッグのための修正が長年にわたって繰り返された結果，ソフトウェアの規模や複雑さが増大し，さらなる修正や機能追加が難しくなっているソフトウェアのことである．

本論文では，レガシーソフトウェアの品質低下，および，保守コスト増大の具体的な要因の1つとして，コードクローン（Code Clone）に着目する．コードクローンとは，ソースコード中の重複したコード列のことであり，主にソースコードのコピー&ペーストを行うことにより生成される（**図1**）．コードクローンは，ソフトウェアの構造を複雑にし，ソフトウェア品質を低下させる一因であると言われており，大規模ソ

† 奈良先端科学技術大学院大学情報科学研究科
Graduate School of Information Science, Nara Institute of Science and Technology
†† 株式会社 NTT データ公共システム事業本部
Public Administration Systems Sector, NTT DATA Corporation
††† 学技術振興事業団若手個人研究推進事業（さきがけ研究21）「機能と構成」領域研究員
PRESTO, Japan Science and Technology Corporation

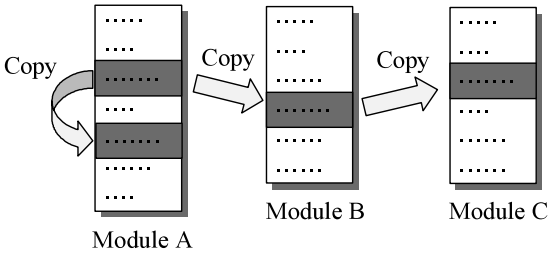

図 1 コードクローンの生成
Fig. 1 Production of code clone.

フトウェアのソースコードの 5〜60%の部分をコードクローンが占めていたことが報告されている[5),7),24)]．多数のコードクローンを含むソフトウェアでは，重複する多数のコード列の 1 つのコピーに変更を加える際に，他のすべてのコピーにも同様の変更を行わなければならないことが多く，保守コストを増大させる原因となる[7)]．さらに，変更し忘れや見逃しが生じた場合には，フォールトが混入し，信頼性を低下させる原因にもなる．しかし，コードクローンと保守性・信頼性の関係はこれまで定量的に明らかにされていない．

本論文では，20 年以上前に開発されたある大規模なレガシーソフトウェアを対象とし，コードクローンと保守性・信頼性の関係を定量的に分析する．ある種のコードクローンが保守性や信頼性に対して特に悪影響を与えることが明らかになれば，そのようなコードクローンを除去したり，できるだけ生じさせないように保守現場へフィードバックしたりすることで，保守作業の軽減，および，保守コストの削減に役立つと期待される．

関連研究としては，近年，Baker[1)] が大規模 C 言語プログラム中のクローンコードを現実的な時間内で検出する手法を提案したことがきっかけとなり，コードクローンの検出方法に関する研究が数多く行われるようになった[1)〜5),7),14)〜20)]．オブジェクト指向プログラムを対象として，クローンコードを除去するための系統的な方法も提案されている[10)]．これらの研究は，コードクローンを検出・除去する方法の開発が主目的であるのに対して，本研究では，検出したコードクローンの分析を主目的とする．

以降，2 章では，本論文の目的とアプローチを詳細に述べる．3 章では，本論文で採用したコードクローン検出方法，および，コードクローンに関する定義について述べる．4 章では，あるレガシーソフトウェアを題材とし，コードクローンと信頼性・保守性の関係を分析した実験について述べる．5 章では，実験結果と考察を述べる．6 章はまとめと今後の課題である．

2. 目的とアプローチ

本論文の目的は，ある 1 つの大規模レガシーソフトウェアを対象としたケーススタディによって，コードクローンと保守性・信頼性の関係を定量的に分析することである．コードクローンと保守性・信頼性の関係を議論するため，モジュール内の位置に着目したコードクローンの分類，および，コードクローンのサイズメトリクスを用いる．実験では，対象となるソフトウェアからコードクローンを検出し，分類およびメトリクスの計測を行い，独立に計測されたソフトウェアの保守性・信頼性メトリクスと比較することで，コードクローンと保守性・信頼性の関係を評価する．

この評価におけるソフトウェアの単位は，ソフトウェアを構成するモジュール（ファイル）とした．多くのソフトウェアシステムでは，保守プロセスデータがモジュールごとに計測されているため，モジュールを単位とすると都合がよい．たとえば，故障の発生と修正，機能追加，改版数，作成日時，更新日時等の情報は，モジュールごとに記録されていることが多い．ソースコードの行数や変数の個数をはじめとするプロダクトデータも，モジュール単位での計測が可能である．そこで，本論文では，それらのモジュール単位の情報を利用するとともに，コードクローンに関してもモジュール単位で計測可能なメトリクスを定義し，モジュール単位での分析を行うことにした．

分析に際しては，分析対象となるソフトウェアの各モジュールについて，信頼性と保守性をあらかじめ推定しておく必要がある．本論文では，信頼性を推定する 1 つの方法として，保守工程において過去に検出されたフォールト数を用いる．過去に多くのフォールトが検出されたソフトウェアモジュールは，フォールト検出数の少ないモジュールよりも信頼性が低かったといえる．

保守性は，保守に要した労力（人月や人日）を計測することで推定できるように思われる．しかし，本論文で対象としているような複数の開発者・保守作業者が関わるソフトウェアの場合には，ある保守作業者が特定の作業に要した時間を計測したとしても，保守性を計測したことにならない．保守に要した時間を計測することは，保守性の計測だけでなく，保守作業の量の計測，作業者の性質の計測，作業の能率の計測，作業環境の計測等を含むためである[26)]．

本論文では，単純ではあるが実践的な 1 つの解決策として，ソフトウェアに含まれる各モジュールの改版数（リビジョン数）を用いる．一般に，機能追加，機

能変更，修正等のためにモジュールに変更が加えられる（改版される）と，そのモジュールの改版数が 1 増加する．改版にはコストを要するため，改版数の大きなモジュールは小さなモジュールよりも，平均的により多くの保守コストが費やされてきたといえる．したがって，改版のための保守コストを要するか否かという観点でモジュールの保守性を評価する場合には，改版数はモジュールの保守性の指標となると考えられる．また，Eick らは 1,000 万行規模のレガシーソフトウェアを分析し，改版が進むにつれて Code Decay（コードの劣化）が進行することを示しており[9]，改版数はコードの劣化の指標にもなりうる．

保守性を推定するもう 1 つの方法は，ソフトウェアの複雑さを表すメトリクス（Complexity Metrics）を用いることである．一般には，複雑なソフトウェアほど保守性が低いといえる．従来より，McCabe のサイクロマティック数，Halsted のソフトウェアサイエンス尺度，オブジェクト指向言語における Chidamber と Kemerer の尺度をはじめとして，数多くのメトリクスが提案されている[7),10),13),22)]．しかし，これらの従来のメトリクスはその定義によって，コードクローンの有無とは本質的に独立であるため，コードクローンにより評価できる保守性とは異なる保守性の側面を表現している．例として，あるモジュールを計測した結果,「1 行あたりのサイクロマティック数」がきわめて大きく，保守性が低いと予測された場合を考える．この場合，このモジュール中にコードクローンが多く含まれていた，もしくは，含まれていなかった，のいずれの場合においても，コードクローンの有無と保守性の関係を明らかにすることはできない．コードクローンの有無とサイクロマティック数が独立であるためである．したがって，コードクローンと保守性の関係の分析においては，従来の Complexity Metrics を用いずに，モジュールの保守性を推定する必要がある．

3. コードクローンの定義と分類

3.1 コードクローン

コードクローンとは，ソースコード中の重複したコード列のことであり，ソフトウェアの開発者や保守作業者がコード列をコピー&ペーストすること等のいくつかの原因によって作られる．コピー&ペースト以外の原因としては，コードジェネレータによって生成されたコード，特定のコーディングスタイルによるもの，パフォーマンスを稼ぐための意図的な繰返し，偶然の一致等がある[5)]．コピー&ペーストによって作られたコードクローンの場合には，コード列に部分的な変更を加えることも多く，そのような部分的に異なる類似のコード列の組もコードクローンと見なすのが一般的である．ただし，どのようなコード列がコードクローンと見なされるかについての正確な定義は，これまで提案された数多くのクローン検出手法やツールごとに異なる[1)~5),8),15)~21)]．

本論文では，神谷ら[15),16)]が提案したトークンベースのクローン検出方法を採用する．プログラミング言語の構文規則に基づいた処理を行い，コード列中の空白やコメント，インデントが異なったり，あるいは，変数名等が書き換えられたりした場合も，コードクローンとして検出する．この手法を実装したツール CCFinder は，COBOL や PL/I 等のプログラミング言語で記述されたレガシーソフトウェアへの適用が可能である．このトークンベースの方法によるクローン検出手順の概略を次に示す．

(1) 字句解析
入力として与えられたソフトウェアに含まれるすべてのソースファイルを，プログラミング言語の字句規則に従ってトークンに分割する．ソースファイル中の空白やコメントは無視される．

(2) トークン変換
型，変数，定数に属するトークンは，同一のトークンに置き換えられる．この置き換えにより，たとえば，変数名だけが異なるコード列の組をコードクローンとして検出できるようになる．

(3) マッチング，および，フォーマッティング
変換後のトークン列に含まれるすべての部分列の集合から，同一の部分列の組を探し出してコードクローンとして検出する．検出にあたっては Suffix Tree マッチングアルゴリズム[12)] を用いることで，ソフトウェアのサイズ n に対して $O(n)$ の計算量で検出できる．最後に，検出されたトークン列の位置情報を元のソースファイル上の行番号に変換し，出力する．

3.2 コードクローンおよびモジュールの分類

本論文では，コードクローンとして検出された互いに等しい（もしくは類似する）2 つのコード列の組を「コードクローンペア」と呼ぶ．コードクローンペアを，含んでいる 2 つのコード列のモジュール内（間）での位置によって，次の 2 種類に分類する（**図 2** 参照）．

(1) In-module クローンペア
コードクローンペアを構成する 2 つのコード列の両方が同一のモジュールに存在する．

(2) Inter-module クローンペア
コードクローンペアを構成する 2 つのコード列がそ

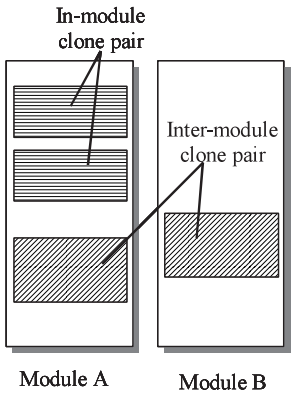

図 2 2 種類のコードクローンペア
Fig. 2 Two types of code clone pairs.

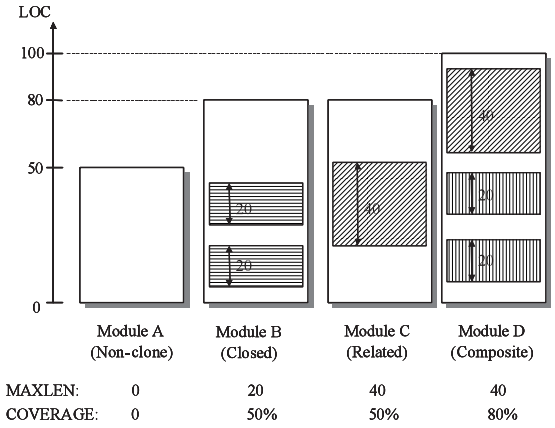

図 4 コードクローンメトリクス
Fig. 4 Code clone metrics.

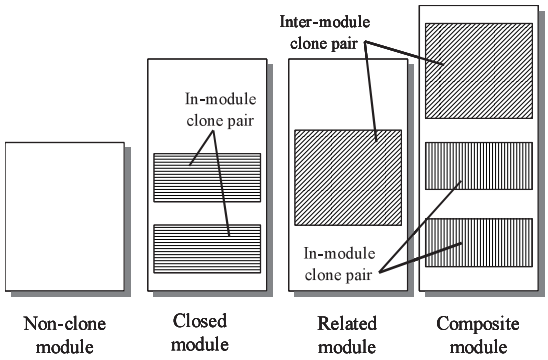

図 3 4 種類のモジュール
Fig. 3 Four types of modules.

れぞれ異なるモジュールに存在する．

それぞれの種類のコードクローンペアは，ソフトウェア品質に対する影響が異なると思われる．Inter-module クローンペアの場合は，類似の処理を行うコード断片が 2 つの複数のモジュールにまたがるため，モジュール間の結合度を増大させる可能性がある．一方，In-module クローンペアは，モジュール間の結合度にさほど影響を与えない可能性がある．

次に，上記の分類に基づいて，モジュールを以下の 4 つの種類に分類する（**図 3** 参照）．

(1) Non-clone モジュール
コードクローン列をまったく含まないモジュールである．

(2) Clone-included モジュール
1 つ以上のコードクローン列を含むモジュールである．さらに次の 3 つの種類に分類される．

(2a) Closed モジュール
コードクローン列として，In-module クローンペアに属するもののみを含むモジュールである．

(2b) Related モジュール
コードクローン列として，Inter-module クローンペアに属するもののみを含むモジュールである．

(2c) Composite モジュール
In-module クローンペアに属するコードクローン列と，Inter-module クローンペアに属するコードクローン列の両方を含むモジュールである．

3.3 コードクローンメトリクス

検出されたコードクローンに基づいて各モジュールを評価するための尺度（コードクローンメトリクス）を定義する．ここでは，2 つのコードクローンメトリクスを次のとおり定義する．

(1) MAXLEN（最大コードクローン行数：Maximum length of code clone）
モジュールに含まれるコードクローン列のうち，最大のものの行数．長いコード列のコピー&ペーストを行うと，この値が大きくなる．

(2) COVERAGE（コードクローンカバレッジ：Coverage of code clone）
モジュールに含まれる行のうち，コードクローン列に含まれる行の割合（パーセントで表す）．1 つのモジュール全体をコピー&ペーストした場合には，コピー元のモジュールの COVERAGE は 100%となる．

各メトリクスの例を**図 4** に示す．図中，モジュール B に含まれる 2 つのコードクローン列は 20 行であるため，モジュール B の MAXLEN は 20 となる．また，モジュール D に含まれる 3 つのコードクローン列のうち最大のものは 40 行であるため，モジュール D の MAXLEN は 40 となる．

一方，図 4 のモジュール B の大きさは 80 行であ

り，いずれかのコードクローン列に含まれる行は 40 行（＝ 20 + 20）であるため，モジュール B の COVERAGE は 50%（＝ 40 ÷ 80 × 100）となる．同様に，モジュール D の大きさは 100 行であり，いずれかのコードクローン列に含まれる行は 80 行（＝ 40 + 20 + 20）であるため，モジュール D の COVERAGE は 80%（＝ 80 ÷ 100 × 100）となる．

4. 実　　験

4.1 実験の目的

本実験の主目的は，ある大規模レガシーソフトウェアを題材として，次の 2 点について分析することである．

(1) モジュールの信頼性（フォールト数）とコードクローンの関係
(2) モジュールの保守性（改版数）とコードクローンの関係

本実験では，上記の 2 つの関係の分析に先立って，ソフトウェア中のコードクローンの分布についても調査した．

4.2 分析対象のソフトウェア

分析対象のソフトウェアは，ある公共機関の業務に用いられるアプリケーションソフトウェアであり，メインフレーム上で稼働している．本ソフトウェアは，Capers Jones による分類[14] に従うと，MIS（Management Information System）に該当する．MIS は，企業，官公庁，銀行等において業務の基盤となるアプリケーションソフトウェアであり，たとえば，給与計算システム，会計処理システム，銀行業務システム，保険業務システム，航空予約システム等が該当する．本ソフトウェアを含む多くの MIS の特徴は，大規模で，データベースを含み，メインフレーム上で稼働することである．本ソフトウェアは，20 年以上前に初期バージョンがリリースされた．今日までに，ユーザからの新たな要求による機能追加と変更，および，デバッグのための修正が続けられ，現在も稼動中である．このソフトウェアは，プラットフォームの変更にともなう大規模な書き換えが数回行われており，分析対象のバージョンは近年にリリースされたものである．

このソフトウェアは複数の言語で記述されているが，本論文ではそのうちの拡張 COBOL 言語で記述された部分についての分析を行った．この部分の規模は約 100 万行であり，約 2,000 個のモジュールから構成される．なお，拡張 COBOL 言語は，COBOL 言語にマクロ命令を追加した言語である（マクロ命令は，プリプロセッサによって COBOL 文に置き換えられてからコンパイルされるため，基本的な言語機能は COBOL と同じである）．

4.3 データの計測

本実験では，各モジュールのフォールト数として，分析対象のソフトウェアのリリース後 6 年間に運用中に検出された値を用いた．したがって，テスト工程において検出されたフォールトは含まれない．モジュールの信頼性の尺度としては，モジュールの規模の要因を排除するために，フォールト数をモジュールの行数で割った値「1 行あたりのフォールト数」を用いた．単なるフォールト数を信頼性の尺度として用いると，規模の大きなモジュールほど検出フォールト数が多くなる傾向にあり，コードクローンと信頼性の関係の分析が困難になると判断した．なお，6 年間にフォールトが検出されたモジュールは全体の約 14%であった．

各モジュールの改版数としては，リリース時に計測された数値を用いた．改版数の最小値は 0，最大値は 338 であった．ソフトウェアの改版としては，単なる仕様変更以外にも，データパッチ等で逃げていたものを正式にモジュール反映するもの，扱うデータ規模（データベース規模）の拡大によるもの，外部インタフェースの変更によるもの等が含まれる．

コードクローンの検出にあたっては，偶然に一致する（コピー&ペーストを原因としない）コード列の組がコードクローンとして検出されることをなるべく避けるために，30 行以上一致するコード列の組をコードクローンとして検出することとし，30 行未満の一致列は無視した．

5. 実験結果と考察

5.1 コードクローンの分布

COVERAGE とモジュール数の関係を**図 5** に示す．図に示されるように，約 50%のモジュールは COVERAGE > 0，すなわち，clone-included モジュールであった．類似するシステムの過去の事例では，COBOL で記述された Payroll システムが 59%のコードクローンを含んでいたという結果が報告されている[8]．

検出されたコードクローンに基づいてモジュールを分類した結果を**図 6** に示す．図に示されるように，コードクローンを含むモジュール（clone-included モジュール）のうち，closed モジュールの占める率は 7%であったのに対して，related モジュールは 34%を占めた．つまり，検出されたコードクローンの大部分は inter-module クローンであった．

5.2 コードクローンと信頼性の関係

Non-clone モジュールと clone-included モジュール

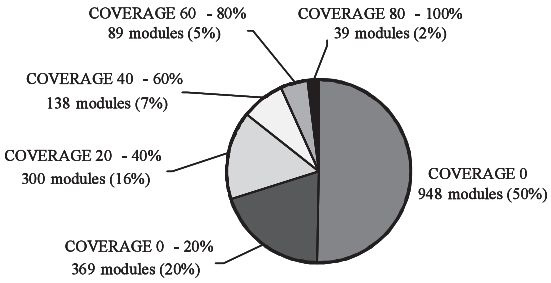

図 5 コードクローンカバレッジ
Fig. 5 Coverage of code clone.

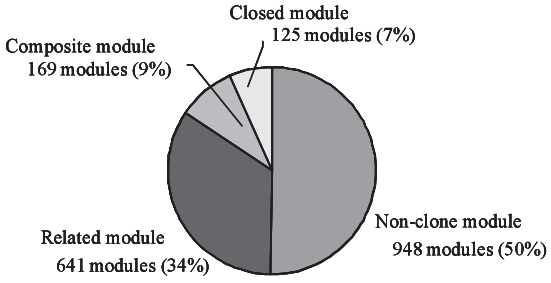

図 6 モジュールの分類
Fig. 6 Classification of modules.

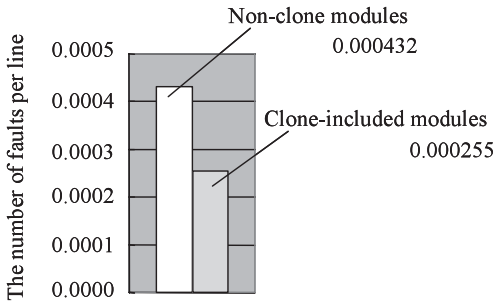

図 7 コードクローンと信頼性の関係
Fig. 7 Relation between code clone and reliability.

の 1 行あたりのフォールト数の平均値を**図 7** に示す．図より，clone-included モジュールは，non-clone モジュールよりも信頼性が平均的に約 40%高い（有意水準 5%で有意差あり）．1 章では，1 つのコードクローン列に変更を加える際に，同一のすべてのコードクローン列にも同様の変更を行わなければならないことが多く，変更し忘れや見逃しが生じるとフォールト混入の原因となりうると述べたが，その逆の結果が得られたことになる．考えられる 1 つの解釈は，このソフトウェアの開発者，および，保守作業者がコードクローンを生成した際に，信頼性の高い部分のコード列のコピーを行ったことである．このようなコピー&ペーストによるコード生成は，一からコードを記述するよりも，フォールト混入の可能性を減らせる可能性がある．

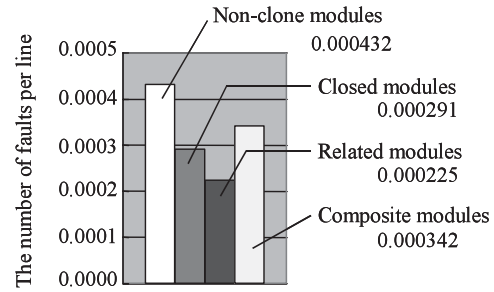

図 8 モジュール種別と信頼性の関係
Fig. 8 Relation between module type and reliability.

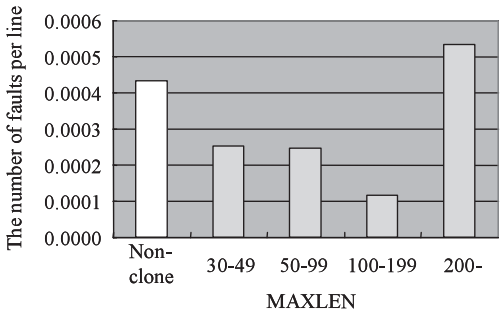

図 9 MAXLEN と信頼性の関係
Fig. 9 Relation between MAXLEN and reliability.

もう 1 つの解釈としては，このようなコードクローンは，新しい種類の機能を含まないため，未知の種類のフォールトを混入させる機会が少なかったことである．

さらに，clone-included モジュールの分類を行った場合の 1 行あたりのフォールト数を**図 8** に示す．コードクローンを含むモジュールのうちでは，related モジュールが最も信頼性が高かったが，統計的な有意差は認められなかった．

次に，コードクローンメトリクスと信頼性の関係の分析として，MAXLEN と 1 行あたりのフォールト数の関係を**図 9** に示す．MAXLEN の値が大きくなるに従って信頼性は次第に向上していくが，200 行以上の長さのコードクローン列を持つモジュール群（MAXLEN $\geqq$ 200）では，逆に，最も低い信頼性を示した（有意水準 5%で有意差あり）．このとこから，一定以上のサイズのコードクローンを生成することは，信頼性を低下させる原因となると予想される．なお，MAXLEN を行数で正規化した値（MAXLEN/LOC）と 1 行あたりのフォールト数の間には，明確な関係は見られなかった．

COVERAGE と 1 行あたりのフォールト数の関係を**図 10** に示す．COVERAGE が 0 より大きく 80%以下の場合，non-clone モジュールよりも信頼性が平均

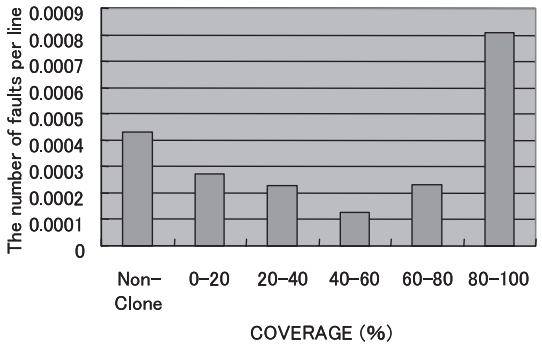

図 10 COVERAGE と信頼性の関係
Fig. 10 Relation between COVERAGE and reliability.

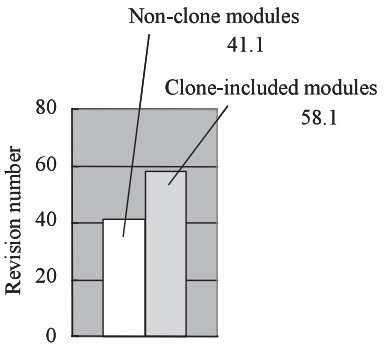

図 11 コードクローンと改版数の関係
Fig. 11 Relation between code clone and revision number.

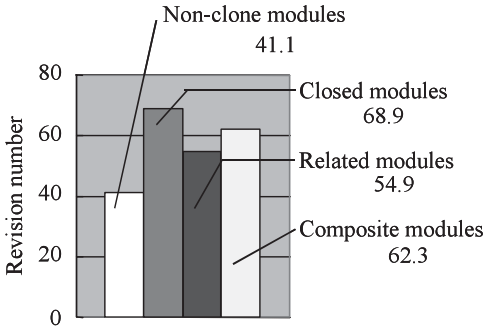

図 12 モジュール種別と保守性の関係
Fig. 12 Relation between module type and revision number.

的に高く（有意水準 5%で有意差あり），COVERAGE が 80%を超えると non-clone モジュールと同程度の信頼性となる（有意差なし）ことが分かった．

このソフトウェアの保守関係者へのインタビューで判明したこととして，コードクローンが生成される 1 つのケースに，変更のリスクが高い部分のコード列に対して機能追加を行う際，変更したい部分のコード列からコピー&ペーストしてコードクローンを生成し，コピー元のコード列からペースト後のコード列へと処理を分岐させたうえで，ペースト後のコード列に変更を加えるというものがあった．このようにコードクローンを生成してから変更を行うことは，ソフトウェアの構造を複雑にするというデメリットがある一方で，変更によるフォールト混入のリスクを低減できる可能性がある．特に，長年の保守の過程で保守作業者の入れ替わりが起こり，ソフトウェア全体を熟知した人間がいない場合には，コードクローンの生成はフォールト混入のリスク回避の 1 つの手段となりうる．

5.3 コードクローンと保守性の関係

Non-clone モジュールと clone-included モジュールの改版数の平均値を**図 11** に示す．図より，clone-included モジュールは，non-clone モジュールよりも改版数が平均的に約 40%大きいといえる（有意水準 5%で有意差あり）．したがって，このソフトウェアが 20 年以上前に初期バージョンがリリースされて以来，clone-included は non-clone モジュールと比べて平均的により多くの改版コストを要したと推測される（clone-included，non-clone の各モジュール群における 1 回あたりの改版コストの平均に偏りがないと仮定した場合）．

この結果に対しては，2 通りの解釈が考えられる．1 つは，コードクローンの存在が改版数の増大の原因となったという解釈である．もう 1 つは，改版数の大きいことがコードクローン生成の原因となったという解釈である．現時点では，後者の解釈には合理的な説明が与えられないが，前者については次のような説明が考えられる．1 章で述べたように，1 つのコードクローン列に変更を加える際には，同一のすべてのコードクローン列にも同様の変更を行わなければならないことが多い[8]．このとき，変更が加えられたすべてのモジュールにおいて改版数が増加するため，clone-included モジュールの集合全体の平均改版数を飛躍的に増大させる原因となる．したがって，改版に要する保守コストの観点からモジュールの保守性を評価すると，clone-included モジュールは non-clone モジュールよりも平均的に保守性が低いといえる．

次に，各種類のモジュールの改版数を**図 12** に示す．Closed モジュール，related モジュール，および，composite モジュールのどれもが，non-clone モジュールよりも平均的に改版数が大きいといえる．さらに，コードクローンを含むモジュールのうちでは，closed モジュールが最も改版数が大きかった（ただし，closed と related の間に有意水準 5%で有意差があるが，closed と composite，および，related と composite の間には有意差なし）．図 9 で示したように，closed モジュー

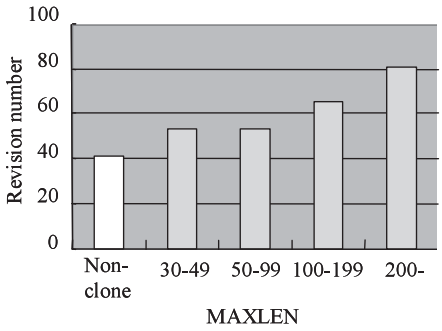

図 13 MAXLEN と改版数の関係
Fig. 13 Relation between MAXLEN and revision number.

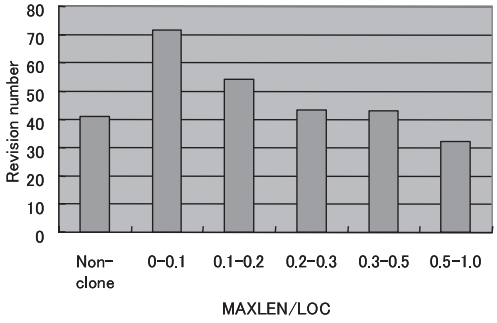

図 14 MAXLEN/LOC と改版数の関係
Fig. 14 Relation between MAXLEN/LOC and revision number.

ルは related モジュールと比較して，平均的により多くのフォールトを発生させたため，フォールト修正のための改版が多くなった可能性がある．

コードクローンメトリクスと保守性の関係の分析として，MAXLEN と改版数の関係を**図 13** に示す．MAXLEN の値が大きくなるに従って平均の改版数は次第に増大する傾向があった（Non-clone モジュールとその他のモジュール群との間に有意水準 5%で有意差あり．MAXLEN が 30〜49 と 50〜99 のモジュール群の間には有意差なし．MAXLEN が 100〜199 のモジュール群は，他のモジュール群との間に有意差あり．同様に，MAXLEN が 200〜のモジュール群についても，他のモジュール群との間に有意差あり）．このことから，サイズの大きなコードクローンを生成することは改版コストを著しく増大させる可能性がある．

MAXLEN をモジュールの行数で正規化した値（MAXLEN/LOC）と改版数の関係を**図 14** に示す．図より，MAXLEN/LOC が 0 より大きく 0.2 以下の場合，non-clone モジュールよりも改版数が平均的に大きい（有意水準 5%で有意差あり）ことが分かった．特に，「MAXLEN/LOC が 0〜0.1 の範囲にあり，MAXLEN が 100 以上」という条件を満たすモジュール群では改版数の平均値は 117 となり，non-clone モジュールにおける平均改版数 41 と比較して非常に大きな値をとった．なお，COVERAGE と改版数の間には，明確な関係は見られなかった．

このソフトウェアの保守関係者へのインタビューでは，機能追加，機能変更，および，修正等を行った際に，「横並びチェック」と呼ばれる作業を行っていることが分かった．この横並びチェックでは，フォールトが発生したり変更を加えた部分のコード列と類似の機能を持ったコード列を特定し，同様の変更が必要となるかどうかを判断する．たとえば，同じようなマクロ呼び出しはしていないか，同じようなデータベースアクセスをしている箇所はないか，同じような臨界値処置をしている部分はないか，同じロジックを作り込んでいないか等を調査する．このようなチェック対象となる類似の処理はコードクローンを形成していることも多く，コードクローンの存在が，横並びチェック，および，改版のためのコストを増大させることがうかがえる．

5.4 信頼性と保守性のトレードオフ

前節までの結果より，コードクローンを生成することは，信頼性の低下を抑える可能性があるが，一方では保守コスト増大の原因となりうることが分かった．したがって，コードクローンの生成においては，信頼性と保守性はトレードオフの関係にあるといえる．ただし，図 9 と図 13 に示されるように，200 行を超える大きなコードクローンを生成した場合には，信頼性と保守性の両方を低下させるため，そのような大きなコードクローンはできる限り生成しないことが望ましい．以上により，信頼性の確保と保守性の確保を両立させるためには，著しくフォールト混入のリスクの高いモジュールに変更を加える場合にのみコードクローン生成によってリスクを回避し，それ以外の変更時にはできる限りコードクローンを生成させないことが望ましいと考えられる．

6. ま と め

本論文では，20 年以上前に開発されたある大規模なレガシーソフトウェアを題材とし，コードクローンと信頼性の関係，および，コードクローンと保守性の関係を分析した．主な分析結果は次のとおりである．

信頼性に関する分析結果

- Clone-included モジュールは，non-clone モジュールに比べて，平均的に 40%信頼性が高い（1 行あたりのフォールト数が少ない）．
- ただし，200 行を超える大きさのコードクローン

を含むモジュール（MAXLEN ≧ 200）は，non-clone モジュールよりも平均的に信頼性が低い．
- COVERAGE が 0 より大きく 80%以下の場合，non-clone モジュールよりも平均的に信頼性が高く，COVERAGE が 80%を超えると non-clone モジュールと同程度の信頼性を示した．

保守性に関する分析結果
- Clone-included モジュールは，non-clone モジュールに比べて，平均的に 40%改版数が大きい．したがって，改版のためにより多くの保守コストが費やされてきたと考えられる．
- MAXLEN の値が大きいほど（すなわち，モジュールに含まれるコードクローンのサイズが大きいほど），改版数がより大きい傾向にある．
- MAXLEN/LOC が 0 より大きく 0.2 以下の場合，non-clone モジュールよりも平均的に改版数が大きいことが分かった．

以上の結果から，コードクローンの生成は，信頼性の低下を抑える可能性があるが，一方では保守コスト増大の原因となりうるため，信頼性と保守性はトレードオフの関係にあるといえる．ただし，200 行を超える大きなコードクローンを生成した場合には，信頼性と保守性の両方の低下の原因となりうる．信頼性の確保と保守性の確保を両立させるためには，大きなコードクローンを生成しないこと，および，著しくフォールト混入のリスクの高いモジュールに変更を加える場合にのみコードクローン生成によってリスクを回避し，それ以外の変更時にはできる限りコードクローンを生成させないことが望ましいと考えられる．

なお，本論文の結果は，COBOL および拡張 COBOL 言語で記述されたソフトウェアに限定した結果であり，異なる言語で記述されたソフトウェアに対しては，異なる結果が得られる可能性がある．COBOL もしくは拡張 COBOL で記述されたレガシーソフトウェアは数多く現存しており，また，新規に開発されるソフトウェアに COBOL 言語が用いられる場合も少なくなく[6]，本論文の結果は，それら COBOL システムの開発者，保守作業者にとって有用であると考える．今後同様の追実験を重ねていくことで，結果の信頼性を高めていくことが必要となる．

また，本論文で分析されたコードクローンは，コピー&ペーストで生成されたものであるが，コードクローンが生成された原因によって，信頼性や保守性に与える影響も異なると考えられる．そのようなコードクローンとして，たとえば，C 言語で開発されたソフトウェアは GUI ビルダツールによって自動生成されたプログラムコードを含む場合があり，それらはコードクローンを多く含む．同様に，Flex や Bison 等の字句・構文解析器生成ツールを用いた場合も，コードクローンを多く含むコードが生成される．今後，より多くの種類のソフトウェア，および，コードクローンに対して，分析を深めていく必要がある．

謝辞 本研究の一部は，新エネルギー・産業技術総合開発機構産業技術研究助成事業の援助によるものである．

参 考 文 献

1) Baker, B.S.: A program for identifying duplicated code, *Proc. 24th Symposium on the Interface: Computing Science and Statistics*, pp.49–57 (Mar. 1992).
2) Baker, B.S.: On finding duplication and near-duplication in large software system, *Proc. 2nd IEEE Working Conf. on Reverse Eng.* (*WCRE'95*), pp.86–95 (July 1995).
3) Balazinska, M., Merlo, E., Dagenais, M., Lagüe, B. and Kontogiannis, K.A.: Measuring clone based reengineering opportunities, *Proc. 6th IEEE Int'l Symposium on Software Metrics* (*METRICS '99*), Boca Raton, Florida, pp.292–303 (Nov. 1999).
4) Balazinska, M., Merlo, E., Dagenais, M., Lagüe, B. and Kontogiannis, K.A.: Partial redesign of Java software systems based on clone analysis, *Proc. 6th IEEE Working Conf. on Reverse Eng.* (*WCRE '99*), Atlanta, Georgia, pp.326–336 (Oct. 1999).
5) Baxter, I.D., Yahin, A., Moura, L., Sant'Anna, M. and Bier, L.: Clone detection using abstract syntax trees, *Proc.IEEE Int'l Conf.on Software Maintenance* (*ICSM'98*), Bethesda, Maryland, pp.368-377 (Nov. 1998).
6) 千葉 滋：プログラミング，これからの 10 年，オブジェクト指向最前線 2002，pp.213–218, 近代科学社 (2002).
7) Chidamber, S.R. and Kemerer, C.F.: A metrics suite for object oriented design, *IEEE Trans. Softw. Eng.*, Vol.20, No.6, pp.476–493 (1994).
8) Ducasse, S., Rieger, M. and Demeyer, S.: A language independent approach for detecting duplicated code, *Proc. IEEE Int'l Conf. on Software Maintenance* (*ICSM'99*), Oxford, England, pp.109–118 (Aug. 1999).
9) Eick, S.G., Graves, T.L., Karr, A.F., Marron, J.S. and Mockus, A.: Does code decay? Assessing the evidence from change management data, *IEEE Trans. Softw. Eng.*, Vol.27, No.1,

pp.1–12 (2001).
10) Fenton, N.E.: *Software metrics: A rigorous approach*, Chapman & Hall, London (1991).
11) Fowler, M., Beck, K., Brant, J., Opdyke, W. and Roberts, D.: *Refactoring: Improving the design of existing code*, Addison-Wesley (1999).
12) Gusfield, D.: Algorithms on Strings, Trees and Sequences, pp.89–180, Cambridge University Press (1997).
13) Halstead, M.H.: *Elements of software science*, Elsevier, New York (1977).
14) Jones, C.: *Applied software measurement*, 2nd edition, McGraw-Hill, United States (1997).
15) Kamiya, T., Ohata, F., Kondou, K., Kusumoto, S. and Inoue, K.: Maintenance support tools for Java programs: CCFinder and JAAT, *Proc. 23rd Int'l Conf. on Software Engineering* (*ICSE2001*), pp.837-838, Toronto, Canada (May 2001).
16) Kamiya, T., Kusumoto, S. and Inoue, K.: CCFinder: A multi-linguistic token-based code clone detection system for large scale source code, *IEEE Trans. Softw. Eng.*, Vol.28, No.7, pp.654–670 (July 2002).
17) Johnson, J.H.: Identifying redundancy in source code using fingerprints, *Proc. IBM Centre for Advanced Studies Conference* (*CAS CON'93*), Toronto, Ontario, pp.171–183 (Oct. 1993).
18) Johnson, J.H.: Substring matching for clone detection and change tracking, *Proc. IEEE Int'l Conf. on Software Maintenance* (*ICSM'94*), Victoria, British Columbia, Canada, pp.120–126 (Sep. 1994).
19) Kontogiannis, K.A., De Mori, R., Merlo, E., Galler, M. and Bernstein, M.: Pattern matching techniques for clone detection and concept detection, *Journal of Automated Software Engineering*, Vol.3, pp.770–108, Kluwer Academic Publishers (1996).
20) Laguë, B., Merlo, E.M., Mayrand, J. and Hudepohl, J.: Assessing the benefits of incorporating function clone detection in a development process, *Proc. IEEE Int'l Conf. on Software Maintenance* (*ICSM'97*), Bari, Italy, pp.314–321 (Oct. 1997).
21) Mayland, J., Leblanc, C. and Merlo, E.M.: Experiment on the automatic detection of function clones in a software system using metrics, *Proc. IEEE Int'l Conf. on Software Maintenance* (*ICSM'96*), Monterey, California, pp.244–253 (Nov. 1996).
22) McCabe, T.J.: A complexity measure, *IEEE Trans. Softw. Eng.*, Vol.2, No.4, pp.308–320 (Dec. 1976).
23) Monden, A., Sato, S., Matsumoto, K. and Inoue, K.: Modeling and analysis of software aging process, *Proc. Int'l Conf. on Product Focused Software Process Improvement* (*Profes2000*), Lecture Notes in Computer Science, Vol.1840, pp.140–153, Springer-Verlag (June 2000).
24) Monden, A., Sato, S. and Matsumoto, K.: Capturing industrial experiences of software maintenance using product metrics, *Proc. 5th World Multi-Conference on Systemics, Cybernetics and Informatics* (*SCI2001*), Florida, USA, Vol.2, pp.394–399 (July 2001).
25) Monden, A., Nakae, D., Kamiya, T., Sato, S. and Matsumoto, K.: Software quality analysis by code clones in industrial legacy software, *Proc. 8th IEEE International Software Metrics Symposium* (*METRICS2002*), Ottawa, Canada, pp.87–94 (June 2002).
26) Schneidewind, N.F. and Ebert, C.: Preserve of redesign legacy systems?, *IEEE Software*, Vol.15, No.4, pp.14–17 (July/Aug. 1998).
27) Sneed, H.M.: Economics of software reengineering, *Journal of Software Maintenance: Research and Practice*, Vol.3, No.3, pp.163–182 (1991).
28) Sneed, H.M.: Planning the reengineering of legacy systems, *IEEE Software*, Vol.12, No.1, pp.24–34 (Jan. 1995).

(平成 14 年 8 月 23 日受付)
(平成 15 年 6 月 3 日採録)

門田 暁人（正会員）

平成 6 年名古屋大学工学部電気学科卒業．平成 10 年奈良先端科学技術大学院大学博士後期課程修了．同年同大学情報科学研究科助手．平成 15 年 Auckland 大学客員研究員．博士（工学）．電子情報通信学会，日本ソフトウェア科学会，IEEE，ACM 各会員．ソフトウェアメトリクス，ソフトウェアセキュリティ，ヒューマンインタフェースの研究の従事．

佐藤　慎一（正会員）

平成 6 年大阪大学基礎工学部情報工学科卒業．平成 8 年同大学大学院博士前期課程修了．同年株式会社 NTT データ入社．平成 12 年より奈良先端科学技術大学院大学博士後期課程在籍．ソフトウェアメトリクス，ソースコード分析の研究に従事．

神谷　年洋（正会員）

平成 8 年大阪大学基礎工学部情報工学科中退．平成 13 年同大学大学院博士後期課程修了．現在，科学技術振興事業団さきがけ研究 21 研究員．博士（工学）．IEEE 会員．

松本　健一（正会員）

昭和 60 年大阪大学基礎工学部情報工学科卒業．平成元年同大学大学院博士課程中退．同年同大学基礎工学部情報工学科助手．平成 5 年奈良先端科学技術大学院大学助教授．平成 13 年同大学教授．工学博士．収集データに基づくソフトウェア開発/利用支援，ウェブユーザビリティ，ソフトウェアプロセス等の研究に従事．電子情報通信学会，IEEE，ACM 各会員．