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

信学技報
IEICE Technical Report
SS2006-31 (2006-08)

コードクローンを対象としたリファクタリングの有効性に関する調査

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性を改善する手段としてリファクタリングが挙げられる．リファクタリングとは，ソフトウェアの外部的な振る舞いを保ったままで，内部構造を改善していくことである．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すべき箇所と指摘されているが，具体的な評価事例はほとんど報告されていない．本稿では，コードクローンに対してリファクタリングを行うコストとその効果を調査し，有効性を評価する．題材は15バージョンのソースコードが公開されているオープンソースソフトウェアであり，その最初のバージョンに対してリファクタリングを行った．コストの尺度としてソースコード修正と回帰テストに要した時間を用い，効果の尺度としてCKメトリクス値の変化と総行数の増減，集約箇所の後のバージョンでの修正回数を用いた．調査の結果，コードクローンのリファクタリングは，メトリクス値の改善についてはあまり効果が無いものの，その後の修正回数の削減については効果があることが分かった．また個々のリファクタリングに要したコストは多くても20分程度であるという結果であった．

キーワード リファクタリング，コードクローン，ソフトウェア保守

Report on Effectiveness of Refactoring for Code Clone

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 *Refactoring* is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior. It is said that duplicated code is one of bad implementations to be refactored, but usefulness of the refactoring hasn't been reported yet. In this paper, we investigate cost and effectiveness of the refactoring to evaluate the usefulness. The target is an open source software system, 15 versions of whose source code are published. We refactored code clones in the first version. We used the time that we spent to modify the source code and perform regression tests as a cost metric, and the changes of CK metrics and the number of modifying the refactored parts from the second version to the final version as effective metrics. As a result, we found that the code clone refactorings didn't improve CK metrics values, but reduced the number of modifications. The cost of each refactoring was at most 20 minutes.

Key words Refactoring, Code Clone, Software Maintenance

1. まえがき

近年，ソフトウェア開発においてリファクタリングが注目され始めている．リファクタリングとは，ソフトウェアの外部的な振る舞いを保ったままで，内部構造を改善していくことである [6]．リファクタリングを行うことでソフトウェアの保守性が増し，その後の保守作業効率の向上が見こまれる．しかし，大規模なソフトウェアに対してはリファクタリングを適用すべき箇所を手作業で探すのは非常に手間がかかり現実的ではない．このようなことからこれまでに多くのリファクタリング位置特定手法が提案されている [3], [7], [16], [17], [20]．

リファクタリングを最も優先して行うべきものの1つとしてコードクローンが指摘されている [6]．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 [10]．コードクローンが生成される原因として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1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．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対して修正の是非を考慮する必要がある．このような作業は，特に大規模ソフトウェアでは非常に手間のかかる作業である．従ってコードクローンを検出し，その情報を保守

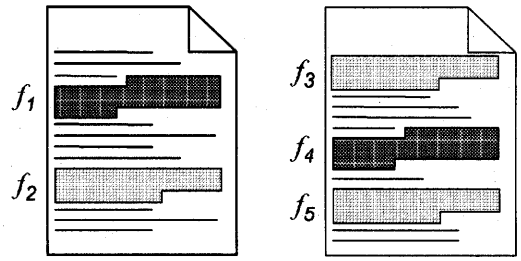


図 1　クローンペアとクローンセット

作業に用いることは有効である．これまでに，数多くのコードクローンを検出する手法 [1], [2], [11], [18]，コードクローンを管理する手法 [21], [22]，コードクローンに対するリファクタリング支援手法 [8], [13], [14] が提案されている．

しかし，これまでに提案されている手法はどれもその手法により検出したコードクローンのリファクタリングの可能性についての評価しか行っておらず，具体的な適用事例はほとんど報告されていない．本稿では，コードクローンに対してリファクタリングを行うコストとその効果を調査し，有効性を評価する．題材は 15 バージョンのソースコードが公開されているオープンソースソフトウェアであり，その最初のバージョンに対してリファクタリングを行った．コストの尺度としてソースコード修正と回帰テストに要した時間を用い，効果の尺度として CK メトリクス値の変化と総行数の増減，集約箇所の後のバージョンでの修正回数を用いた．調査の結果，コードクローンのリファクタリングは，メトリクス値の改善についてはあまり効果が無いものの，その後の修正回数の削減については効果があることがわかった．また，個々のリファクタリングに要したコストは多くても 20 分程度であるという結果であった．

2. 準　　備

2.1 コードクローン

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 [11] 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 [10], [11]．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．

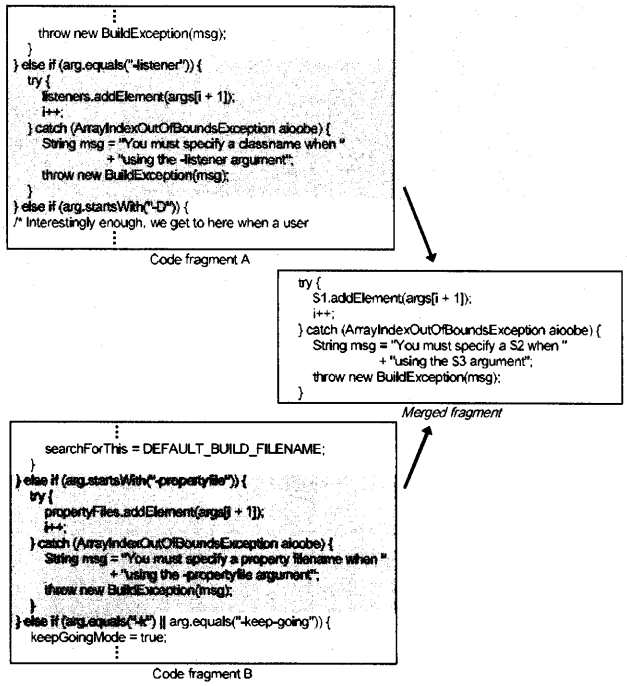


図 2　コードクローン抽出の例

2.1.1 コードクローン検出ツール: CCFinder

CCFinder [11] 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トークン列に変換する．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．

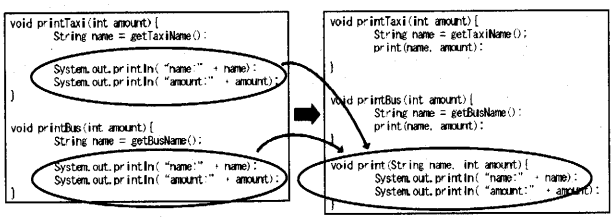
Step2（変換処理）：実用上意味を持たないコードクローン（テーブルの初期化部分や関数やモジュールをまたがったコードクローン等）を取り除くこと，及び，些細な表現上の違いを吸収することを目的とした変換ルールによりトークン列を変換する．例えば，変数名は同一のトークンに置換されるので，変数名が付け替えられたコード片もコードクローンであると判定することができる．

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．

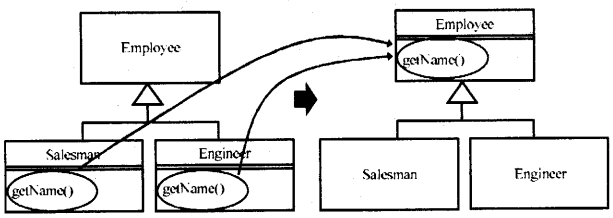
Step4（出力整形処理）：検出されたクローンペアのソースコード上での位置情報を出力する．

2.1.2 コードクローン抽出ツール: CCShaper

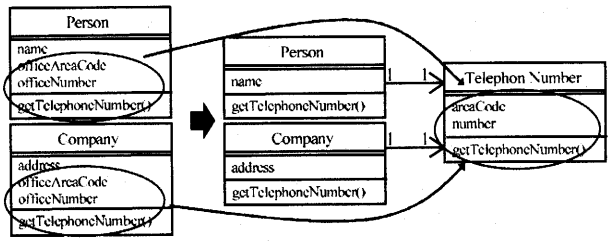
2.1.1 節で述べたように，CCFinder はコードクローンをトークンの列として検出するため，検出されたコードクローンはリファクタリングを行いやすい単位になっていない．本稿ではコードクローンに対してリファクタリングを行うため，リファクタリングの適用可能はコードクローンを得る必要がある．このようなコードクローンを得るために，コードクローン抽出ツール CCShaper を用いる．CCShaper [9] は CCFinder の検出したコードクローンの内部に含まれるプログラミング言語の構



(a) Extract Method の例



(b) Pull Up Method の例



(c) Extract Class の例

図 3　コードクローンに対するリファクタリング例

造的なまとまりを持った部分を抽出する．図 2 は抽出の例を表している．この例の 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，つまり CCFinder が検出したコードクローンであることを示している．リファクタリングを行う場合は，このコードクローン全体を対象とするよりは，内部に含まれる try 文のみを対象としたほうが望ましい．このように，CCFinder の検出したコードクローンはリファクタリングを行う単位としては適切ではない．CCShaper を使うことによって，よりリファクタリングに適した単位のコードクローンを抽出することができる．対象が Java 言語の場合，CCShaper は以下の単位でコードクローンを抽出する．

宣言	:	クラス, インターフェース
メソッド	:	メソッド本体, コンストラクタ, スタティックイニシャライザ
文	:	if, for, while, do, switch, try, synchronized

2.2 リファクタリング

リファクタリングとは，外部から見たときの振る舞いを保ちつつ，理解や修正が簡単になるように，ソフトウェアの内部構造を変化させることである [6]．最も優先してリファクタリングを行うべき箇所（不吉な匂い）として，重複コード（コードクローン）が指摘されている [6]．コードクローンをどのようにリファクタリングすべきかは，状況によりさまざまであろうが，文献 [6] では，次のように紹介されている．

Case1: 同一クラス内の複数箇所が重複している場合

この場合は，*Extract Method* を適用して，各メソッドの内部からコードクローンを抽出し，新たなメソッドとして定義する．抽出箇所は，そのメソッドの呼び出し文に変更すればよい．図 3(a) は *Extract Method* の適用例を示している．この例では，メソッド printTaxi と printBus がコードクローンを共有している．このコードクローンを新たなメソッドとして抽出し，そのメソッド呼び出し文に置換することにより，コードクローンを集約することができている．

Case2: 兄弟クラス間で重複コードが見られる場合

この場合は，まず各クラスに対して *Extract Method* を行い，次に *Pull Up Field* や *Pull Up Method* を適用すれば，解決する．場合によっては，さらに *Form Template Method* などを適用する場合もありえる．図 3(b) は *Pull Up Method* の適用例を示している．この例では，クラス Employee を親クラスとして持つクラス Salesman と Engineer がコードクローン（メソッド getName）を共有している．このメソッドを共通の親クラスである Employee に引き上げることによって，コードクローンを取り除くことができている．

Case3: まったく関係のないクラス間で重複コードが見られる場合

この場合は，*Extract Class* を使い，古いクラスから新たなクラスへと処理を委譲するように変更する．図 3(c) は *Extract Class* の適用例を示している．リファクタリング前は，クラス Person と Company が共に電話番号に関する処理を持っている．*Extract Class* を適用し，この部分の処理を新しいクラスとして抽出することによって，クラス Person と Company の間でのコードクローンが取り除かれている．

2.3 CK メトリクス

CK メトリクスとは，Chidamber と Kemerer が提案したメトリクスであり，オブジェクト指向ソフトウェアの複雑度を測定する [4]．オブジェクト指向ソフトウェアの性質を測定するメトリクスとして広く用いられている．各メトリクスは，クラス単位で計測される．以下，簡単に各メトリクスについて述べる．

WMC(Weighted Methods per Class)

そのクラスに定義されているメソッドに重み付けをして足し合わせた値である．重み付けの方法としてサイクロマチック数 [15] などが用いられる．この値が高いほどそのクラスが複雑であり，保守を行うことが困難であることを表している．

DIT(Depth of Inheritance Tree)

クラス階層における継承の深さを表す．深い位置にあるクラスは，多くのフィールドとメソッドを継承しているため，振る舞いを予測するのが困難である．

NOC(Number Of Children)

サブクラスの数を表す．この値が大きいほど，このクラスの変更が多くのクラスに影響を与えることを意味する．小さな変更も思わぬ影響を与える可能性があり，注意深く修正作業を行わなければならないクラスである．

CBO(Coupling Between Objects)

関係しているクラスの数を表す．関係しているとは，他のクラスのフィールドやメソッドにアクセスしたり，メソッド内で他のクラスのオブジェクトを受け取っていたりしていることをいう．この値が大きくなると，そのクラスは他の多くのクラスと結びついて動作していることになり，理解が困難になる．

RFC(Response For a Class)

クラスのレスポンスとは，そのクラスのオブジェクトにメッセージが送られた結果，実行される全てのメソッドの集合である．他のオブジェクトに対して実行されるメソッドも含まれる．この値が大きいクラスは，他のクラスから多くのメッセージを受信し，それに答えるために他のクラスに多くのメッセージを送信しており，複雑なクラスであることを表している．

LCOM(Lack of Cohesion Of Methods)

クラスの凝集とは，クラスのメソッドがお互いに関連している程度を表す．各メソッドがどのフィールドにアクセスするかを調べることで凝集度を求めることができる．このメトリクスは，クラスのメソッドの全ての組み合わせのうち，参照するフィールドに共通するものがない組み合わせの数から，共通するものがある組み合わせの数を引いたものである．ただし，0 よりも小さい場合は 0 とする．この値が大きいクラスは，各メソッドがあまり関連していないことを表し，1 つのクラスとして適切でないことを示唆している．

表 1 S_0 の CK メトリクス値の変化

クラス名	WMC	DIT	NOC	CBO	RFC	LCOM
org.apache.tools.ant.DirectoryScatter	21(+1)	1(±0)	0(±0)	0(±0)	47(+1)	158(+20)

表 2 S_1 の CK メトリクス値の変化

クラス名	WMC	DIT	NOC	CBO	RFC	LCOM
org.apache.tools.ant.DirectoryScatter	21(+1)	1(±0)	0(±0)	0(±0)	47(+1)	158(+20)

表 3 S_3 の CK メトリクス値の変化

クラス名	WMC	DIT	NOC	CBO	RFC	LCOM
org.apache.tools.ant.Task	17(+1)	1(±0)	23(±0)	4(±0)	24(+6)	98(+16)
org.apache.tools.ant.ProjectHelper$NestedElementHandler	4(±0)	2(±0)	0(±0)	4(±0)	14(−3)	0(±0)
org.apache.tools.ant.ProjectHelper$TaskHandler	4(±0)	2(±0)	0(±0)	7(−2)	21(−7)	0(±0)

表 4 S_4 の CK メトリクス値の変化

クラス名	WMC	DIT	NOC	CBO	RFC	LCOM
org.apache.tools.ant.ProjectHelper	15(+2)	1(±0)	0(±0)	7(+1)	66(+7)	79(+27)
org.apache.tools.ant.taskdefs.Expand	4(±0)	2(±0)	0(±0)	5(±0)	35(−3)	2(±0)
org.apache.tools.ant.taskdefs.Untar	4(±0)	2(±0)	0(±0)	8(±0)	36(−3)	2(±0)

表 5 S_5 の CK メトリクス値の変化

クラス名	WMC	DIT	NOC	CBO	RFC	LCOM
org.apache.tools.ant.taskdefs.Javac	23(−1)	3(±0)	1(±0)	11(±0)	93(±0)	128(−105)
org.apache.tools.ant.taskdefs.MatchingTask	16(+1)	2(±0)	11(±0)	5(+1)	56(+4)	94(+79)
org.apache.tools.ant.taskdefs.Rmic	17(−1)	3(±0)	0(±0)	8(±0)	66(−2)	98(−17)

3. 実　　験

3.1 概　　要

本実験の目的は，リファクタリングのコストと効果を測定することにより，その有効性を調査することである．実験対象として Ant のバージョン 1.1 ～ 1.6.5 の計 15 版を用いた．最も古いバージョン 1.1 に対して CCFinder と CCShaper を用いてコードクローン検出を行った．バージョン 1.1 の規模はソースファイル数は 83 個，総行数は 19,986 行である．本実験では，検出する最小のコードクローンの大きさを 50 トークンに設定し，結果として 7 個のクローンセットを検出した．検出に要した時間は約 1 分であった(注1)．検出したコードクローンの特徴を表 7 に示す．表 7 の分類は，2.2 節で述べた重複コード片間の関係を表している．これら全てのクローンセットに対してリファクタリングを試み，クローンセット S_2 を除く 6 個のクローンセットを集約することができた．また，ソースコードの修正後，外部的振る舞いが変化していないことを確認するために，Ant 付属のテストケースを用いて回帰テストを行った．

本実験では，リファクタリングのコストの尺度として，ソースコードの修正と回帰テストに要した時間を用いた．また，リファクタリングの効果の尺度として，CK メトリクスの値の変化と総行数の増減，リファクタリング箇所のバージョン 1.2 以降での修正回数を用いた．

表 7 検出されたクローンセット一覧

ID	コード片数	単位	分類	時間
S_0	2	メソッド	Case1	7 分
S_1	6	メソッド	Case1	19 分
S_2	2	メソッド	Case1	-
S_3	2	メソッド	Case1	15 分
S_4	2	if 文	Case2	14 分
S_5	2	メソッド	Case2	5 分
S_6	2	メソッド	Case3	17 分

(注1)：CPU:Pentium4 3.0GHz，メモリ:2GB，OS:WindowsXP

3.2 コスト：ソースコード修正と回帰テストに要した時間

表 7 の “時間” の項目は，ソースコード修正と回帰テストに要した時間を表している．最も時間を要したのはクローンセット S_1 であり，19 分必要であった．また，最も短時間でリファクタリングを完了できたのは，クローンセット S_5 であり，5 分しかかからなかった．リファクタリングを完了させるのに必要な時間は，適用したリファクタリングパターンの違いにはあまり左右されていなかった．

3.3 効果：CK メトリクス値の変化

本実験では，CK メトリクスを計算するために ckjm [5] を用いた．表 1 ～ 6 は各クローンセットのリファクタリングを行った後の CK メトリクス値を表している．リファクタリングを適用することによって，いずれかのメトリクス値が変化したクラスのみを表示している．括弧の中の値は，そのメトリクス値がリファクタリング前に比べてどの程度増減したのかを表している．例えば，表 1 では，クラス DirectoryScanner の WMC の値は 21 であり，これはリファクタリング前に比べて 1 増加したことを現している．これらの表を見てわかるように，コードクローンを集約することによって，総合的に CK メトリクスの値が改善されたとは言いがたい．特に，クローンセット S_0 と S_1 のリファクタリングでは，値が改善しているメトリクスは無く，WMC，RFC，LCOM の値が改悪しているのみであった．

3.4 効果：総行数の変化

表 8 はリファクタリング前後での Ant の総行数の変化を表している．クローンセット S_6 に対するリファクタリングでは，新たにクラスを 1 つ作成しているため，コードクローンは集約されているものの，リファクタリング前に比べ行数は多くなってしまっている．その他のクローンセットに対するリファクタリングでは，コードクローンは既存のクラス内に集約されるため，総行数は減っている．しかし，多くても 20 行程度しか減っておらず，リファクタリングによってサイズ面での改善が見られたとは言いがたい．

3.5 効果：バージョン 1.2 以降での修正回数

図 4 ～ 10 は各クローンセットのコード片に対するバージョン 1.1 ～ 1.6.5 間での修正の様子を表している．コード片は C_n で表され，修正が加わる度に，C'_n，C''_n と変化していく．各バージョン間でコード片に修正が加わっているか否かは UNIX コマ

表 6　S_6 の CK メトリクス値の変化

クラス名	WMC	DIT	NOC	CBO	RFC	LCOM
org.apache.tools.ant.taskdefs.NewClass	2	1	0	2	7	1
org.apache.tools.ant.taskdefs.JavacOutputStream	4(±0	1(±0)	0(±0)	2(+1)	10(−1)	0(±0)
org.apache.tools.ant.taskdefs.TaskOutputStream	3(±0)	1(±0)	0(±0)	2(+1)	8(−1)	0(±0)

表 8　リファクタリング前後での行数の変化

リファクタリング	S_0 後	S_1 後	S_3 後	S_4 後	S_5 後	S_6 後
行数	19,982(-4)	19,964(-22)	19,981(-5)	19,981(-5)	19,962(-24)	19,990(+4)

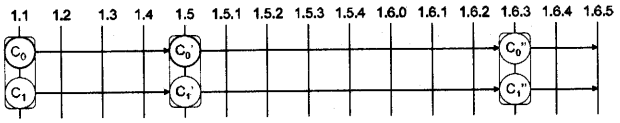


図 4　S_0 のコード片に対する修正の流れ

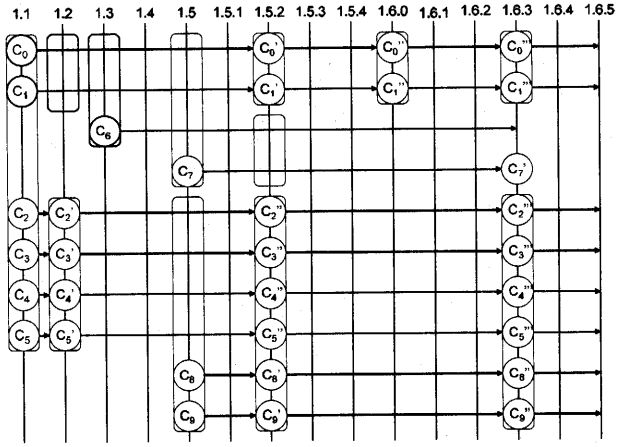


図 5　S_1 のコード片に対する修正の流れ

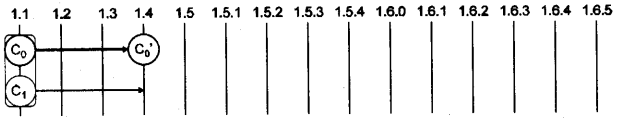


図 6　S_2 のコード片に対する修正の流れ

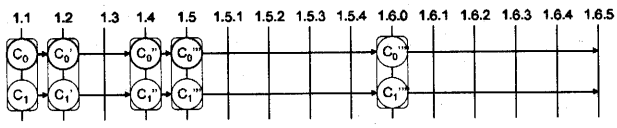


図 7　S_3 のコード片に対する修正の流れ

ンドの diff を用いて調査した．またバージョン 1.2 以降のソースコードに対しても CCFinder と CCShaper を実行し，バージョン 1.1 でコードクローンになっていた部分がそれ以降のバージョンでもコードクローンになっているかを調査した．コード片はクローンセットの要素となっている場合のみ表示されており，コードクローンでなくなった時点で図から消えている．同一のクローンセットに含まれるコード片は長方形の枠で囲まれている．例えば，図 4 では，コード片 C_0 と C_1 にバージョン 1.5 と 1.6.3 で修正が加わっており，バージョン 1.1 ～ 1.6.5 までコード片 C_0 と C_1 は重複関係にあることを表している．

これらの図から，クローンセット S_0 と S_3 はバージョン 1.1 ～ 1.6.5 まで繰り返し修正が加えられているが，重複関係を維持しているのがわかる．クローンセット S_1 は，バージョン 1.2 で 2 つのクローンセットに分裂してはいるものの，それらのクローンセットは修正が加わりながらもバージョン 1.6.5 まで存

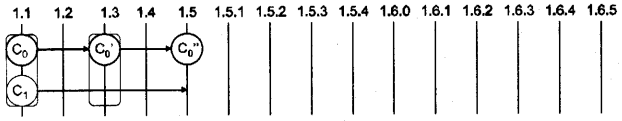


図 8　S_4 のコード片に対する修正の流れ

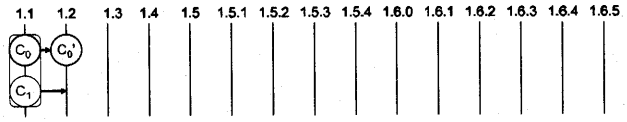


図 9　S_5 のコード片に対する修正の流れ

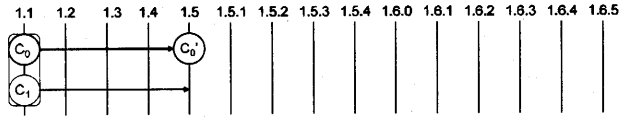


図 10　S_6 のコード片に対する修正の流れ

在している．

それに対して，クローンセット S_2，S_4，S_5，S_6 はいずれも途中で一部のコード片にのみ修正が加わり，コード片間の重複関係がなくなっていることがわかる．クローンセット S_4 のコード片 C_0 におけるバージョン 1.3 での修正は，コーディングスタイルの変更であったため，片方のコード片のみに修正が加わっているものの重複関係は維持されている．

3.6 考　　察

本実験では，CK メトリクスを測定するために ckjm [5] を用いている．このツールは，メトリクス WMC として単純にそのクラスに定義されたメソッドの数を用いている．そのため，*Extract Method* のように，メソッドの数を増やすリファクタリングパターンを適用した場合は，必ず WMC の値は悪化してしまう．サイクロマチック数などを用いて適切にメソッドを重み付けすれば，メソッド数を増やすリファクタリングであっても，WMC の改善がある場合があるかもしれない．

各々のリファクタリングの規模が小さいため，CK メトリクス値の増減は微細であったが，コードクローンを集約することによって必ずしもメトリクス値が改善するとはかぎらないことがわかった．中には，クローンセット S_0 や S_1 のようにメトリクス値を悪化させているだけのリファクタリングも存在した．CK メトリクスはオブジェクト指向ソフトウェアの複雑度を測定するためのソフトウェアである．つまり，リファクタリングの効果を CK メトリクスを用いて評価することは，複雑度の観点からリファクタリングを評価していることになる．秦野ら [7] もリファクタリングの効果を測定するための手段として CK メトリクスを用いている．しかし，複雑度の変化はリファクタリングの効果の 1 つの側面に過ぎず，他の面からの評価も必要である．

リファクタリング箇所の後のバージョンでの修正回数はクローンセットによってさまざまであった．全てのクローンセットがバージョンをまたいで存在しているわけではなく，各コー

ド片に対して異なった修正が加わった結果，コードクローンでなくなるという場合もあった．本実験では，7つのクローンセットのうち，4つがそのようなクローンセットであり，一度も機能追加やバグ修正の変更が加えられてはいなかった．しかし，同一のクローンセットに含まれる全てのコード片に一度でも同様の修正が加わった場合は，その後も繰り返し各コード片に同様の修正が加わる傾向であることが判明した．このことから，コードクローンであるという理由ですぐにリファクタリングを行うのではなく，全てのコード片に同様の修正が加わった場合にリファクタリングを行うことで，その後の修正回数を減らすことが可能であると考えられる．

リファクタリングが有効であるかどうかは，その効果がコストを上回っているかどうかで決まる．本実験の場合では，例えばクローンセット S_0 については，リファクタリングを行うために7分の時間が必要であったが，それを行うことによってバージョン 1.5 と 1.6.3 での修正箇所が2つから1つに減ったという効果が得られた．しかし，リファクタリングによって WMC, RFC, LCOM の値が若干増加してしまった．この効果とコストの大小関係は一意に決めることのできるものではなく，さまざまな開発・保守のコンテキストによって異なってくる．

また，本実験で行ったリファクタリングは基本的かつ汎用的であり，ソースコード修正を行う者のスキルや対象ソフトウェアのドメインを限定するものではないと考えられる．

4. 関連研究

門田ら [19] 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．彼らの研究では，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 [12] 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，今後も引き続き各コード片に同様の修正が加わると考えられ，リファクタリングをすることが有益であると述べている．しかし，彼らは修正の履歴からリファクタリングの是非を議論したにとどまっており，実際にリファクタリングを行いその有効性を評価しているわけではない．

5. まとめ

本稿では，リファクタリングを行うコストとその効果を調査し，有効性を評価した．題材は 15 バージョンのソースコードが公開されているオープンソースソフトウェアであり，その最初のバージョンに対してリファクタリングを行った．コストの尺度としてソースコード修正と回帰テストに要した時間を用い，効果の尺度として CK メトリクス値の変化と総行数の増減，集約箇所の後のバージョンでの修正回数を用いた．調査の結果，コードクローンのリファクタリングは，メトリクス値の改善についてはあまり効果が無いものの，その後の修正回数の削減については効果があることが分かった．また個々のリファクタリングに要したコストは多くても 20 分程度という結果であった．

謝辞 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，文部科学省 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．

文　献

[1] B. S. Baker. A program for identifying duplicated code. In *Proc. the 24th Symposium of Computing Science and Statistics*, pp. 49–57, Mar 1992.

[2] I. Baxter, A. Yahin, L. Moura, M. Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proc. International Conference on Software Maintenance 98*, pp. 368–377, Mar 1998.

[3] B. Bois, S. Memeyer, and J. Verelst. Refactoring - improving coupling and cohesion of existing code. In *Proc. the 11th IEEE Working Conference on Reverse Engineering*, pp. 144– 151, Nov 2004.

[4] S. Chidamber and C. Kemerer. A metric suite for object-oriented design. *IEEE Transactions on Software Engineering*, 25(5):476–493, Jun 1994.

[5] ckjm. http://www.spinellis.gr/sw/ckjm/.

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

[7] 秦野, 乃村, 谷口, 牛島. ソフトウェアメトリクスを利用したリファクタリングの自動化支援機構. 情報処理学会論文誌, 44(6):1548–1557, Jun 2003.

[8] 肥後, 神谷, 楠本, 井上. コードクローンを対象としたリファクタリング支援環境. 電子情報通信学会論文誌, J88-D-I(2):186–195, Feb 2005.

[9] 肥後, 植田, 神谷, 楠本, 井上. コードクローン解析に基づくリファクタリングの試み. 情報処理学会論文誌, 45(5):1357–1366, May 2004.

[10] 井上. 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, 4:1–4, Dec 2005.

[11] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, Jul 2002.

[12] M. Kim and D. Notkin. Using a clone genealogy extractor for understanding and supporting evolution of code clones. In *Proc. of the 2nd International Workshop on Mining Software Repositories*, pp. 17–21, May 2005.

[13] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *Proc. of the 8th International Symposium on Static Analysis*, pp. 40–56, Jul 2001.

[14] J. Krinke. Identifying similar code with program dependence graphs. In *Proc. the 8th Working Conference on Reverse Engineering*, pp. 301–309, Oct 2001.

[15] T. Macabe. A complexity measure. *IEEE Transaction on Software Engineering*, 2(4):308–320, Dec 1976.

[16] R. Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *Proc. the 20th IEEE International Conference on Software Maintenance*, pp. 350–359, Sep 2004.

[17] 丸山. 基本ブロックスライシングを用いたメソッド抽出リファクタリング. 情報処理学会論文誌, 43(6):1625–1637, Jun 2002.

[18] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Proc. of the International Conference on Software Maintenance 96*, pp. 244–253, Nov 1996.

[19] 門田, 佐藤, 神谷, 松本. 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, 44(8):2178–2187, Aug 2003.

[20] L. Tahvildaril and K. Kontogiannis. A metric-based approach to enhance design quality through meta-pattan transformations. In *Proc. the 7th European Conference on Software Maintenance and Reengineering*, pp. 183–192, Mar 2003.

[21] M. Toomim, A. Begel, and S. Graham. Managing duplicated code with linked editing. In *Proc. IEEE Symposium on Visual Languages and Human-Centric Computing*, pp. 173–180, Sep 2004.

[22] 植田, 神谷, 楠本, 井上. 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, 86-D-I(12):863–871, Dec 2003.