ソフトウェア工学 126－9
（2000．3．9）

類似度に基づくソフトウェア品質の評価

長橋 賢児

(株)富士通研究所 ソフトウェア研究部

業務アプリケーションの開発、メンテナンス時には類似の処理を実装する際に、既存のコードを複製流用する、いわゆるコピーペーストプログラミングが行われることがある。ソフトウェアにどの程度のコード複製が行われているかを知ることによって、メンテナンスコストの増大やソフトウェア品質の劣化の状況を知ることができる。本稿では、簡易な分析によってプログラム間の類似度を計算し、それに基づいてコード複製の状況を調査し、またコード複製を共通化によって除去することで可能なコード削減量を推計して、コード品質を評価する一方法を提案する。

Evaluating Software Quality with Code Similarity

Kenji Nagahashi

Software Laboratory, Fujitsu Laboratories Ltd.

Programmers often try to be productive by replicating existing program fragments, so called 'copy-paste programming', to implement similar application logic. As such replication may cause considerable impact on software maintenance, it is helpful to know how much such replication exists in the software suite, and to estimate possible improvement achieved by eliminating the replication. That will help managers make decision for re-engineering. This paper shows the techniques for estimating the amount of code replication in the software . These techniques define similarity between programs and estimate possible cut down achieved by eliminating code replication with the calculated similarity.

1. はじめに

ソフトウェアを調べると、ほとんど同一のコード記述が複数箇所に発見されることが多い。このようなコード記述の重複を意図的なもの、意図的でないものを問わず、コード重複と呼ぶ。コード重複が起こる理由は、第一にソフトウェア開発時に行われているコード複製(カットアンドペースト)プログラミングである。大規模なシステムを開発する際には、各所に分析設計時に気付かなかった類似した処理が必要になる。これに気付いた開発者はすでに作られたソフトウェアからコード断片を複製して、処理が異なる部分だけを手直しすることによって効率よく開発しようとする。類似した処理を含むプログラムを担当する開発者が異なる場合には、共通性に気付かずに独立してほとんど同じコードを実装することもある。

複製される部分は本来再利用部品になる可能性を持っていると考えられる[1]が、共通部品に発展させるためには、分析設計に多大な労力を割かねばならず、開発者は短期的に見て手間のかからない「複製による再利用」を選ぶ。COBOL 言語など部品化を十分に支援できない言語での開発の場合

はそもそも部品化が難しく、コードの重複の頻度が高まる傾向にある。大規模業務アプリケーション開発においては、設計者と実装者が異なるのが一般的であり、現実には分析設計を十分に行うことでコード重複を排除するということは難しい。しかし、このようなコード複製はコード量の増加させて見かけの生産コストを引き上げるだけでなく、プログラムの理解を妨げ、障害の影響範囲が広がる可能性があるなど、ソフトウェアのメンテナンスコストに大きな影響を与える[1]。例えば、複数プログラムに重複している箇所に障害が見つかれば、重複しているすべての箇所を調査して、同じように修正しなければならない。どこに重複したコードがあるかを調査するだけでも多大な労力がかかるうえ、重複したコードは完全に同じではないことが多いから、それぞれについて詳細に修正方法を検討しなければならない。

以上のような観察から、ソフトウェアに存在するコードの重複の程度を知ることができれば、共通部品や再利用部品の可能性を発見することや、同じように変更すべきプログラムの集団を知ることができ、メンテナンスに役立つと考えられる。またコードの重複を排除することにより、ソフトウェアにどのような効果があるかを知ることができれば、リエンジニアリングによってソフトウェアを整理すべきかどうかの判断材料とすることができるとも考えられる。

ソフトウェア中のコード重複を発見しようという試みにはすでにいくつかのアプローチがある。例えば、Baxter らは構文木間で比較を行うことにより、コード複製(code replication)を発見する手法を提案している[2]。また Ducasse らはプログラムの行を文字列比較して比較行列を作り、コード複製の状況を可視化する方法を提案している[3]。しかしいずれもコード複製を発見することに重点をおいており、そのコード複製を定量的に評価しようという試みはない。リエンジニアリングによってコード重複を解消すべきかどうかの意思決定は、コード複製を解消するために必要な労力と、それによって得られる効果を比較して評価することが必要である。コード重複を発見することはもちろん有用だが、それに基づいてリエンジニアリングすべきかどうかの判断を行うには、さらに踏み込んで、コード重複の程度を定量的に評価して、何らかの指標を提供する必要がある。本研究では COBOL 言語で記述されたソフトウェアを対象に、コードにどの程度の重複が存在するのか、またリエンジニアリングによってどの程度の改善効果が得られるのか、をコードの類似度を利用して評価する方法を検討した。

2. アプローチ

重複したコードがまったく同じであることはむしろ少ない。これは完全に同一ならば簡単に共通化できるからである。むしろほとんど同じだが一部異なる、という場合にコードの重複が起こりやすい。したがって単純に完全同一のコードを検索する方法は十分とは言えない。そこで、プログラムテキスト間から特徴を抽出して、その間に類似度を定義し、任意の2プログラム間の類似度を計算することによって、コード重複の量を評価する。また、コード重複の程度を直感的に把握できるように、コード重複を解消するための共通化を行ったと仮定した場合に、どの程度コード量が削減されるのかを推計するという方法を提案する。

2.1. 前処理

COBOL プログラムには各行に「一連番号領域」「識別番号領域」と呼ばれる文字領域があり、プログラムコードの内容とは直接関係しない文字列を含むので、これを取り除く。またコメント行も取り除く。

2.2. 類似度の計算

プログラム間の類似性の評価は、プログラムから特徴を抽出して、その特徴の間の類似度を計算することによって行う。抽出する特徴と、特徴間の類似度の計算方法は様々なものが考えられ、その選択によりプログラム間の類似性を評価する観点が変わる。コード複製の量とコード複製を解消す

ることによる効果を見積もることを目的とすることから、次のような特徴および類似度の計算方法を採用した。

1. 手続き部の各行を正規化したものを特徴とし、差分類似度(後述)を計算する
2. 手続き部を文単位に分解して正規化したものを特徴とし、文ベクトル類似度(後述)を計算する

ここで「特徴」と呼んでいるものは、いずれの場合も文字列の順序付集合である。類似度が計算されると、それをもとにクラスタリング分析を行い、プログラムを互いに似た集団に分類することができる。

2.2.1. 差分類似度

正規化を施したプログラムテキストをそのまま比較する簡単な方法である。手続き部(PROCEDURE DIVISION)以降の各行を正規化して、そのまま出現順に並べたものを特徴とする。正規化として、複数個の空白文字の連続を1個の空白文字に置き換える処理を行う(文字列定数中の空白文字は置き換えの対象にはしない)。

抽出した特徴に対しLCS(UNIXのdiffに使われている差分抽出のアルゴリズム)を利用して類似度を定義する。プログラムA、Bから抽出した各行に対してLCSを適用すると、AとBの行単位の差分が得られる。この差分は、AをB(またはその逆)に書き換えるための編集スクリプトとみなすことができる(図1)。得られた差分の行数をd_{AB}とし、A、Bの総行数l_A、l_Bとして、類似度sを

$$s = 1 - \frac{d_{AB}}{l_A + l_B}$$

と定義する。このsは、完全に一致するモジュールに対しては1、完全に相違するモジュールに対しては0となる。AとBの中で行の順序を維持した共通部分の割合を表している。

差分類似度は特に構文解析などが必要ないため、COBOL以外の言語にも容易に適用可能である。

2.2.2. 文ベクトル類似度

コード重複がコード複製によるものではない場合、同じ処理であっても、記述の順序が違うなどして、

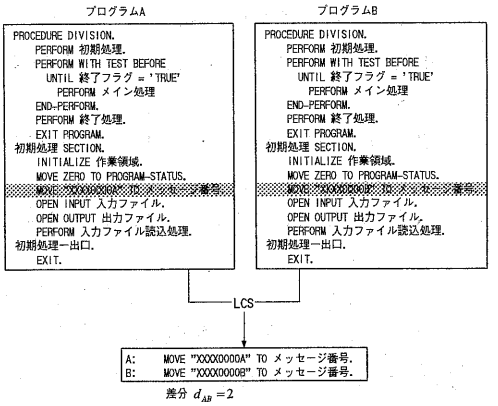


図1 プログラムの差分

表面的に大きな違いがある場合がある。差分類似度はプログラムテキストを順序を維持してそのまま比較するため、このような場合に類似性を検出できない。そこで、プログラム中に記述されている文にどのようなものがあるかを、順序に関係なく比較することによって類似性を検出する方法として、文ベクトル類似度を定義した。

文ベクトル類似度では、COBOLプログラムの手続き部を文単位に分解して正規化したものを特徴とする。この特徴抽出には構文解析が必要だが、COBOLの実行文は常に予約語の動詞で始まるので、それを目印に容易に文への分解が可能である。IF文やEVALUATE文など、文中に他の文を含むことができる文の場合は、内部に含まれる文はIF文、EVALUATE文には含めない。また、扱っているデータと、それに対する操作に絞って比較を行うために、データを扱わない文(フロー制御のみの文など)や、補助語などを取り除く。具体的には以下のように、各文の仕様に合わせた処理を行う。

- EVALUATE文:EVALUATE <選択主体>と、WHEN <選択対象>に分解してそれぞれ特徴要素とし、WHEN句にはEVALUATE <選択主体>を付け加える。WHEN OTHERは取り除く。
- SEARCH文:EVALUATE文同様にSEARCH部とWHEN句に分ける。
- 内PERFORM文:内部に含まれる文手前まで

を1つの特徴要素とする。

- IF文:IF部を1つの特徴要素とする。THEN、ELSEは除去する。
- END-IF、END-EVALUATE、END-PERFORMなど文の終わりを示す語を除去する。
- CONTINUE文、EXIT文、GO TO文、GOBACK文：取り除く。
- AT END、INVALID KEY指定：取り除く。ただし中に含まれる文は特徴要素に含める。

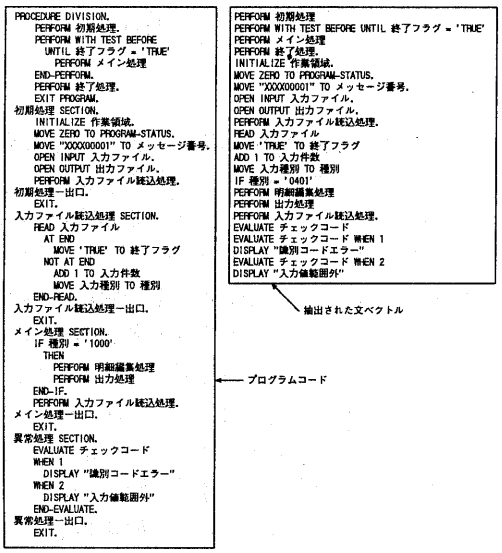


図2 プログラムコードから抽出された文ベクトル類似度用の特徴

図2にサンプルコードと、それから生成される文ベクトル類似度のための特徴を示す。

こうして抽出された特徴の要素として出現する文字列を集計し、それぞれが各プログラムの特徴中に何回出現するかを対応付けると、各プログラムの特徴ベクトルが定義できる。この特徴ベクトル間の類似度を特徴ベクトル間の距離として定義する。すなわちプログラムA、Bに対応する特徴ベクトルをv_A、v_Bとしたとき、

$$s = \frac{|v_A - v_B|^2}{|v_A||v_B|}$$

とする。この場合、特徴中の文字列の順序は類似度に影響しない。つまりこの類似度は処理の順序は無視して、大まかにプログラム中でどのような処理が存在するかを比較しているものと解釈できる。COBOL言語にはローカル変数の概念がなく、各変数にプログラム中で一意の名前が付ける慣習が一般的なので、このような比較方法でもプログラムの処理内容の特徴をとらえることができ、よい結果を得ることができる。逆にCやJavaなど現代的な言語への応用は難しい。

2.3. 削減可能量の推定

上記の類似度の定義に従って、任意のプログラムの間の類似度を計算して、コードの重複のある組み合わせを知ることができる。しかし類似度の表やそれを用いたクラスタリングの結果だけでは、コード複製がどの程度行われているかを直感的に理解しにくい、という指摘があった。そこで複製によるコストを直感的にとらえる方法として、共通化による削減可能量を推計する方法を考案した。

まず、重複したコードを持つプログラムを1つに共通化したとき、共通化後のコード量(行数)がどうなるか、を推定する方法について考える。差分類似度の計算と同じ方法で行を抽出したファイルA、Bに対してLCSを適用し、AとBの差分を得る。この差分は、AとBの内容をを共通化しようとした場合に共通化できない部分であるとみなせるので、AとBを1プログラムに共通化した結果のコード量l_{AB}を、

$$l_{AB} = (l_A + l_B) - \frac{l_A + l_B - d_{AB}}{2} = \frac{l_A + l_B + d_{AB}}{2}$$

と推計することができる。もちろん、これは+共通化した場合に必要になる条件分岐などの行数を考慮していないので、粗い近似である。これに差分類似度sの計算式からd_{AB}を逆算して代入すれば、

$$l_{AB} = \frac{l_A + l_B + (l_A + l_B)(1-s)}{2} = \left(1 - \frac{s}{2}\right)(l_A + l_B)$$

となる。この式によって類似度からも共通化後の行数を計算できる。またこの式を使って、文ベク

トル類似度など他の類似度から推計される削減後コード量を定義することもできる。

次にA, Bを1本にまとめたプログラムと、別のプログラムCを共通化することを考える。これはA、Bのうち Cに近い(類似度が高い)ものをCと共通化することと同等の効果を持つと考えて、A、B、Cの3プログラム共通化後の行数 l_{ABC} を、

$$l_{ABC} = l_{AB} + l_C + \frac{l_x + l_C + d_{xC}}{2}$$

※ x はA、Bのうち、d_{xC} が小さくなる方

と計算する。以下同様にプログラム D、E、…をさらに共通化した場合を計算していくことができる。

上記の計算は、どのプログラムを、どの順序で共通化していくかによって結果が異なる。共通化すべきプログラムの集団、および共通化の順序は次のような手順で決める。

1. まず、共通化下限類似度s*を指定する。これより大きな類似度のプログラムの組を共通化する。
2. 共通化されていないプログラムのうち、もっとも大きな類似度s (ただしs ≧ s*であること)の2プログラムの組 P_1、P_2 を求め、共通化後コード量を計算する。この2プログラムを要素とするグループGを定義する。
3. Gに属さないプログラムで、Gに属するプログラムとの間で最大の類似度s (ただしs ≧ s*であること)を持つプログラム P_i を選び、Gに属するプログラムと共通化した場合のコード量を計算する。P_i をGに加える。
4. 該当する P_i がある間3を繰り返す。
5. 該当する P_1、P_2 がある間、2~4を繰り返す。
6. 共通化されたプログラムについて計算された共通化後コード量とに、共通化されていないプログラムすべてのコード量を加え、ソフトウェアの共通化後コード量とする。

3. 実験

上記の計算アルゴリズムに基づき、ツールを実装し、実際に使われているCOBOL資産を対象にして分析を試行した。ツールはソースコードの解析を含め、ほとんどをインタプリタ言語Perlで実装し、差分類似度の計算に用いる差分抽出に GNUの diffユーティリティを使用している。実験に使用したサンプル資産のプロフィールを表1に示す。いずれも、実際に使われている資産から一部を選び出したものである。

3.1. 類似度と分類

3資産それぞれに対して差分類似度、文ベクトル類似度を計算した。

求められた類似度を用いてクラスタリング分析を行うと、プログラムを互いに似たグループに分類することができる。図4は資産Cを差分類似度を使ってクラスタリングした結果を樹状図(Webブラウザで見ることができるように、HTMLのテーブルになっている)で示したものである。資産Cにも、ほとんど同一のプログラムがあり、例えばプログラム yyyy0010 と yyyy0050 は類似度0.9であり、さらに yyyy0070、yyyy0100 とは類似度0.7であること、などが読み取れる。このような表示方法によって、コード重複の状況を直感的に理解することができる。分類の結果と、プログラムを目視で確認して比較したところ、類似度類似度1から0.5程度までが共通化の候補になりうるという印象であった。

表1 実験サンプルCOBOL資産のプロフィール

資産名	プログラム数 * 手続部平均行数	特性等
A	167 * 768	開発中に複製が多く行われているという指摘のあった資産。
B	132 * 4580	CASEツールでCOBOLを自動生成した資産。特に複製が行われているわけではないが、業務の類似性からコードの重複があるだろうと推測されている。
C	115 * 527	設計段階から共通化に努力したという資産。

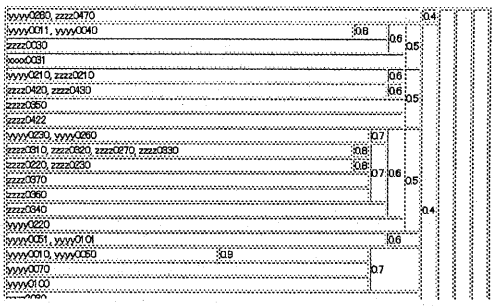


図 4　類似度をもとにプログラムを分類した樹状図(部分)

差分類似度と文ベクトル類似度の違いを調べるため、同じプログラムの組み合わせに対する差分類似度と文ベクトル類似度を対応づけて視覚化したのが図 3 である。横軸が差分類似度、縦軸が文ベクトル類似度で、あるプログラムの組み合わせに対するこの 2 つの類似度を、点によって表現している。図中の線分は(0,0)-(1,1)を結ぶ線で、点がこの線の左上にあるということは、その組み合わせでは、差分類似度よりも文ベクトル類似度のほうが大きな値を示した、という意味である。

資産 A は、全体として 2 種類の類似度がほぼ同じであるが、一部大きく異なるケースがある。これらを調べたところ、文ベクトル類似度が高くなるケースでは COBOL の節名、段落名など、文ベクトル類似度用の抽出には含まれない部分の違いが差分類似度を低くしていることがわかった。

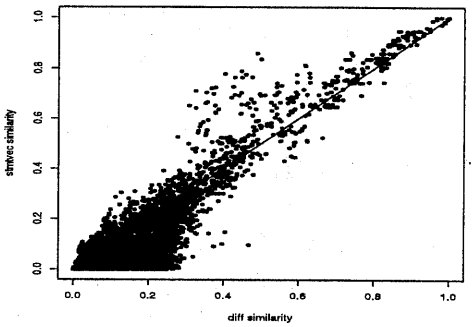
資産 C はどの組み合わせについても 2 種類の類似度がほぼ同じで、しかも資産 A よりも類似度が低い方に分布が集中している。設計段階から共通化に努力したことが表れていると言える。資産 B は特徴的で、どの組み合わせについても文ベクトル類似度は差分類似度より高い。これはコード複製がほとんどなく、類似処理を独立して記述していることの表れと考えられる。

3.2.　削減可能量

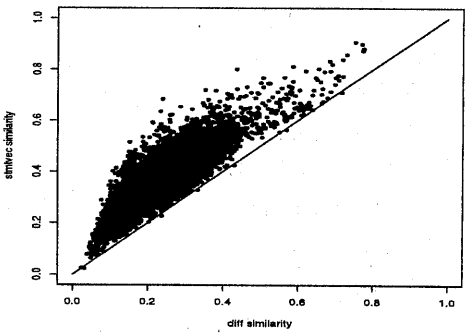
差分類似度を元に、削減可能量を推計した。図 6 は、共通化下限類似度として 1.0~0.2 の値を与えて、それぞれに対して推計された共通化後のコード量がどうなるかを、元のコード量を 100 とする割合で示したものである。

資産 A はコード複製が多いという観察の通り、差分類似度が大きな値を示す組み合わせが多く、共

資産 A



資産 B



資産 C

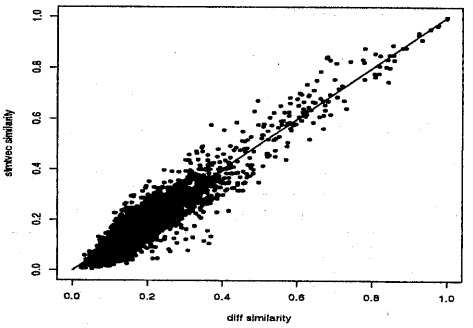


図 3 各資産の差分類似度と文ベクトル類似度の違い

通化下限類似度 0.7 付近まで急激にコード量が減っている。しかしそのあとは下限類似度を小さくしてもさほどの削減効果は得られない。したがって資産 A に対しては、類似度 0.7 以上の組み合わせに絞って整理を行うのが効率がよいと言うことができる。

資産 C は下限類似度の大きい範囲での急激な低下は見られず、共通化下限類似度の変化に対して直線的にコード量が減っていく。これは資産 A に比べてコード複製が少ないためであろう。この計算では考慮されていないが、下限類似度を小さくすると共通化されるプログラム数が増えることによって条件分岐が増え、実際のコード量の変化はこの計算結果より緩やかなものとなる。また関わるプログラム数の増加にともなって共通化にかかる手間も増えることを考えれば、資産 C に対する整理は全体として大きな効果があるとは言えない。

資産 B はさらに全体的に類似度が低く、下限類似度を 0.6 まで下げてもほとんど削減効果が見られない。ただし前述のとおり、資産 B は差分類似度よりも文ベクトル類似度が高い値を示す傾向があり、実際のコード削減可能量は差分類似度から計算される値よりは大きいと思われる。

3.3. 実験ツールの性能について

分析の実行環境は GP7000S model 25 (UltraSPARC 300MHz ×1、主記憶 512MB、OS は Solaris 2.6) を使用した。例えば A 資産 167 個のプログラムの類似度を計算するために要した時間は、およそ 1 時間であった。

類似度はプログラムのすべての組み合わせに対して計算されるので、類似度の計算回数はプログラム数 n に対して $O(n^2)$である。このため対象のプログラム数を増やすと処理時間が長くなり、1000 個のプログラムを対象にした場合には、同じ環境で 50 時間あまりかかり、実用上は 3000 個程度が限界であった。差分類似度の計算に使用する LCS アルゴリズムも、最適なものは $O(n^2)$なので、プログラムの行数が増えると急速に性能が悪化する。

コード削減可能量推定は類似度計算の結果を利用しているので高速に処理でき、100 個程度のプログラムに対し、処理時間は高々20 秒程度であった。

4. 結論

プログラム間に類似度を定義することによって、コード重複の存在をとらえる方法を示した。また類似度を利用してコード量削減可能量を推計することにより、共通化によってコード重複を取り除こうとしたときに、どの程度の効果が期待できるかという、コード重複の性質を知ることができることを示した。

差分類似度はコードの複製が多い場合には高い類似度を示すが、処理内容を維持したまま記述の順序が入れ替わった場合でも類似度が低下するため、コード複製によらない類似性を検出する能力は劣る。そのような類似性を検出する方法として、文ベクトル類似度を定義した。B 資産の例に見るとおり、コード複製によらない重複を検出できていると思われる結果は得られたが、それ以外の節名・段落名の違いを無視することなどによる効果も影響しており、明確には示せなかった。

コード削減可能量の推計は、粗い近似でありながらも、資産のコード重複の特性を反映した結果が得られ、ソフトウェアのメンテナンスの方針を決める判断材料を提供することができると考えられる。

5. 今後の課題

本稿ではプログラム間類似度として簡単な 2 種類のものを定義したが、この他にも様々な類似度を考えることができるだろう。それぞれの類似度は本稿のコード削減量推計のように、利用目的に合わせたものを考えるべきものなので、類似度の意味を考えながら、さまざまなものを検討する必要がある。現在の比較方法は同一データに対するデータ項目名の違いなど表面的な違いに左右されてしまうので、これに対処するメカニズムを導入することも必要であろう。

コード削減量の推計に関しては、残念ながら、現在は計算結果がどの程度信頼できるものなのかに

ついての評価は十分ではなく、直感的な妥当性の評価があるだけである。本来ならば実際の再構築の結果と比較して、信頼性を評価することが望ましいが、実際にそのような評価は難しい。今後何らかの形で評価を行い、より高度な分析を導入して推計精度を向上させるべきかを判断したい。「類似度」や「コード削減可能量」など明瞭な意味付けを持つ数値によってソフトウェアの状態を示すことは、実用上大きなインパクトを持ち、特にプロジェクト管理者に好評であった。それだけに、数値の信頼性は重要である。

また処理性能も課題である。現在は処理のほとんどがインタプリタ言語で実装されているため、これをより高速な実装に置き換えることによって高速化をはかる余地はあるものの、類似度の計算を必要最小限に限定するなどアルゴリズム上の工夫をしない限り本質的には解決しない。今後効率化を目指したい。

コード重複は共通部品、再利用部品の可能性を示唆するとされるが、本稿の分析では、まだ共通部品、再利用性部品の発見には有効な手段を提供できていない。これも今後重要な方向の一つである。

6. 参考文献

[1] E. Burd and M. Munro, "Investigating the Maintenance Implication of the Replication of Code", Proc. International Conference on Software Maintenance: ICSM'97, IEEE Press, 1997.

[2] I. D. Baxter et al, "Clone Detection Using Abstract Syntax Trees", Proc. International Conference on Software Maintenance: ICSM'98, IEEE Press, 1998.

[3] S. Ducasse, M. Rieger and S. Demeyer, "A Language Independent Approach for Detecting Duplicated Code", Proc. International Conference on Software Maintenance: ISCM'99, IEEE Press, 1999.