特別研究報告

題目

ソースコードの静的特性を用いた Java プログラム間類似度測定ツールの試作

指導教官
井上 克郎 教授

報告者
小堀 一雄

平成 15 年 2 月 19 日

大阪大学 基礎工学部 情報科学科

平成 14 年度 特別研究報告

ソースコードの静的特性を用いた Java プログラム間類似度測定ツールの試作

小堀 一雄

内容梗概

近年のプログラム開発現場においては，再利用を用いた開発が頻繁に行われている．その際，プログラム部品は単にコピーされて用いられる場合もあれば，一部を改変して用いられる場合も存在する．この時，ソフトウェアに含まれる似た部品を抽出することで部品の再利用に関する情報を得ることができる．ソフトウェアプログラムの類似度を測定する手法として，最も単純かつ有効な手法として，ソースコードそのものの文字列比較があるが，これは大量のソースコードを比較対象とした場合に膨大な時間がかかってしまうため，そのような場合でも類似度を高速かつ低コストに抽出する手法が必要になる．

本研究ではソースコードに出現する予約語や識別子などのトークンの数を利用した類似度メトリクスを用いた類似度測定手法の提案を行う．本手法の場合，ソースコードの文字列情報を数値に抽象化することで比較に要する時間を大幅に削減することができる．そのため，各部品間の類似度を高速に測定することが可能となる．また，研究グループで開発しているソフトウェアプロダクトの収集・解析・検索システム SPARS において，類似度測定ツールを試作した．本ツールは，Java プログラムを解析した結果を受けて差分が 3 %未満の部品同士を一つの部品群へとグループ化を行う機能を実現している．本ツールを利用することにより，各部品の利用関係をその部品が所属する部品群に継承することが可能となる．

主な用語

メトリクス (Metrics)
類似度 (Similarity)
トークン (Token)
ハッシュ(Hash)

目 次

1 まえがき

ソフトウェアの大規模化と複雑化に伴い，高品質なソフトウェアを一定期間内に効率良く開発することが重要になってきている．これを実現するために近年のソフトウェア開発において，再利用を用いた開発がよく行われている．再利用とは，既存のソフトウェア部品を同一システム内や他のシステムで利用することを指し，開発期間の短縮や品質向上を期待できるといわれている [1, 2, 5, 7]．ソフトウェアの再利用による効果を最大限に引き出すためには，開発者が開発しようとするソフトウェアに必要な部品およびライブラリに関する知識を持つことが重要になってくるが，知識の共有が満足になされていないために，同種のプログラムが別々の場所で，独立して開発されている事も多い．

一方でインターネットの普及により，SourceForge [10] などのソフトウェアに関する情報を交換するコミュニティが誕生し，大量のプログラムソースコードが簡単に入手できるようになった．これらの公開されている大量の部品の中から，開発者の必要としている機能を持つ部品，その機能の使い方を示している部品のような，再利用に有益な情報を提供する検索システムを実現する事で，知識の共有が実現でき，再利用を促進する事ができると考えられる．

そこで我々の研究チームでは利用実績に基づくソフトウェア部品の収集, 検索システム *SPARS*(Software Product Archiving, analyzing, and Retrieving System) を研究しており，Java プログラムを対象として SPARS-J という検索システムを開発している．SPARS-J においては，継承やメソッド呼び出しによる利用関係だけでなく，ソースコードをコピーして転用することによる利用関係も補足するために，コピーされたと推測される部品を判定し，部品群としてグループ化するという特徴がある．しかし大量のソースコードに対して類似度計算を行う場合，既存の文字列比較による類似度測定法では，解析コストが膨大になる問題が発生するため，それを考慮にいれた手法が必要である．

そこで本研究では，類似度メトリクスを用いた Java プログラム間の類似度測定手法を提案する．この手法では，ソースコードの静的特性の数値に抽象化したメトリクス値を比較することで類似部品の抽出が可能となるので，解析コストを低く抑えることが可能となる．また，提案した手法を類似度測定ツール *Luigi* として実現し，SPARS-J における部品間の類似度測定部として実装した．さらに *Luigi* を用いた適用実験を行うことで，その有効性を検証する．

以下，2 節で SPARS-J について説明した上で，ソースコードの文字列比較を用いた類似度測定手法についての考察を行う．3 節では文字列比較を用いた既存の類似度比較システムの説明を行う．それを踏まえ 4 節では Java ソースコードの部品情報を数値化した類似度メトリクスの比較を用いた類似度測定方法を提案する．さらに，部品間の類似度を測定し，類

似部品対を部品群としてまとめる機能を持つ Java プログラム間類似度測定ツール *Luigi* の説明をする．5 節で適用実験を行い，その有効性を検証する．最後に 6 節でまとめと今後の課題について述べる．

2 SPARS-J

2.1 SPARS-J とは

現在，我々のチームでは利用実績に基づくソフトウェア部品検索システム *SPARS-J* (Software Product Archiving, analyzing, and Retrieving System for Java) を研究している．

SPARS-J は Java プログラムを対象とした部品検索システムで，収集された部品に対して解析およびインデックス付けを行うことで検索機能を提供する．SPARS-J は現在のところキーワード検索に対応しており，検索はクラス単位で行われる．検索結果を表示する際，検索された部品を評価し順位付けし，選別して表示する仕組みが必要となるが，SPARS-J においては，部品間の利用関係を元に定められた Component Rank[4] に基づいて順位付けを行うことで，利用実績の高い汎用的な部品の取得を容易にしている．

SPARS-J は，入力されたソースコードに対してデータベースを構築するデータベース構築部と，データベースから情報を取得することで部品の検索を行う部品検索部から構成されている．概念図を図 1 に示す．

以下，それぞれについて説明を行う．

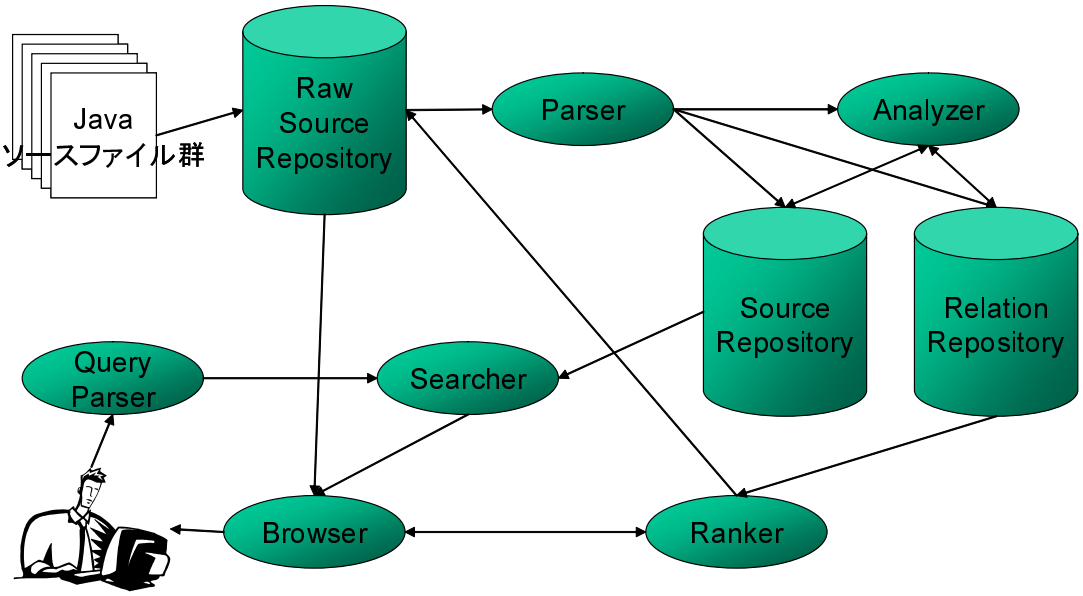

図 1: SPARS-J の概観図

2.2 データベース構築部

データベース構築部では，検索時に必要なデータを提供するために，次の手順で入力された Java ソースコードからデータベースの構築を行う．部品のグループ化，Component Rank の計算については後述する．

1. 入力された Java ソースコードに対して構文解析を行う事で，利用しているクラス名や，LOC，トークン数などのメトリクスを抽出し，部品情報としてデータベースに保存する．

2. （1）と同時に，ソースコード中で出現する全ての単語に対して，インデックス付けを行い，検索用のデータベースを構築する．その際，どこでその単語が出現した（コメント中，メソッド宣言中，文中など）かを出現した単語と共に保存する．

3. 入力された Java ソースコードに対して意味解析を行い，利用しているクラス名が実際にどのクラスを指すのかを判定し，利用関係情報としてデータベースに保存する．SPARS-J では，クラス継承，メソッド呼び出し，フィールド変数参照，抽象クラスの実装などを利用関係とみなしている．

4. メトリクス値を用いて部品のグループ化を行い，類似した部品を一つの部品として扱う．

5. 利用関係をもとに，Component Rank を求め，それを部品の評価値として保存する．

2.2.1 部品のグループ化

一般に，部品の集合には，コピーした部品や，コピーして一部を変更しただけの部品が数多く存在する．異なるシステムをまたいで全く同じ，もしくはほとんど似た部品が現れる場合，それらの部品が再利用されたのではないかと推測する事ができる．しかし，コピーしたという利用関係に関してはコピー元の特定が難しく，部品グラフ上で利用関係として定義するのは難しい．そこで，類似した部品をまとめて一つの部品群としてみなし，グループ化を行う．その際，それぞれの部品への辺が一つの部品群への辺とみなされるため，コピーされた部品への評価を高くすることができる．

部品間の類似度を評価する指標として，当初は 2 つのソースコードファイル間で一致する行の割合 [12] を求めていたが，現在はソースコードから LOC，トークン数，利用されている変数などのメトリクスを用いている．

2.2.2 Component Rank の計算

一般にソフトウェア部品の間には互いに利用する，利用されるという利用関係が存在する．本システムにおいては，この利用関係をもとに全ての部品を対象に評価値（Component Rank）を計算する．計算の際には，各部品を頂点，部品間の利用関係を利用する側からされる側への有向辺として，部品間の関係を部品グラフとして表現する．開発者はある部品を参照した後に，グラフ中の辺に沿って利用関係のある部品の一つを見ると仮定することで，この部品グラフを開発者の閲覧行動に関するマルコフ連鎖モデルとみなし，定常状態において各部品が参照されている確率を求め，それを Component Rank としている．この Component Rank により，ただ単に利用数が多い部品だけでなく，利用数が多い部品が利用している部品も重要であると評価する事ができる．

2.3 部品検索部

部品検索部では，構築されたデータベースを用いて，部品の検索を行う．現在のシステムでは，基本検索として全てのソースコードからのキーワード検索を実現しているが，検索条件を指定する事で，コメントのみにキーワードが現れる部品を省く，クラスやメソッド定義にキーワードが出現する部品のみを表示するなどの詳細な検索を行うことができる．検索の手順は以下のようになっている．

1. ユーザーはブラウザを通じてキーワードを入力する事で，部品検索部にクエリーを投げる．
2. 部品検索部はクエリーをキーワードの集合に分解し，データベースに対してそれぞれのキーワードが出現する部品を照会する．キーワードが複数ある場合には，結果を統合する．
3. ユーザーが指定した検索条件に基づいて，与えられたクエリーにマッチした部品を Component Rank の順に並べ，それを検索結果として出力する．
4. ユーザーはブラウザを通じて検索結果を受け取る．ユーザーはさらにキーワードを追加する事でより詳しい検索を行ったり，検索結果として表示された部品に関して，ソースコードや，その部品のメトリクスなどの部品の詳しい情報を取得する事が出来る．

2.4 本研究との接点

2.2 節で述べたように現在我々のチームで開発している SPARS-J では, コピー転用による利用関係も補足するために，大量の部品の中から類似した部品をまとめて一つの部品群への

グループ化を行う必要がある．SPARS-J のプロトタイプシステムではこの類似度比較部に文字列比較を用いた類似度測定手法を採用していた．しかし，この手法にはコストの面での問題がある．

そこで本研究では，この文字列比較を用いた手法の内容を考察した上で，よりコストの低い，メトリクスを用いた類似度比較手法の実装を本研究で試みた．

3 関連研究

本節では，文字列比較を用いた既存の類似度測定システム SMMT[12] について説明する．

3.1 類似度の定義

ソフトウェアシステム P はそれを構成する要素の集合と考え，$\mathrm{P} = \{p_1, \cdots, p_m\}$ と書く．二つのソフトウェアシステム $P = \{p_1, \cdots, p_m\}$, $Q = \{q_1, \cdots, q_n\}$ に対し，等価な要素の対応 $R_s \subseteq P \times Q$ が得られるとする．P と Q の R_s に対する類似度 $S(0 \leq S \leq 1)$ は次のように定義されている

$$S(P,Q) \equiv \frac{|\{p_i|(p_i,q_j) \in R_s\}| + |\{q_j|(p_i,q_j) \in R_s\}|}{|P| + |Q|}$$

これは，図 2 のように対応 R_s に含まれる P, Q の要素数を P と Q の総要素数で割ったものである．R_s に関係しない P, Q の要素が増えることによって S は下がる．$R_s = \phi$ では，$S = 0$ となる．また，P と Q が同じものの時，$\forall i(p_i, q_i) \in R_s$ となり $S = 1$ となる．

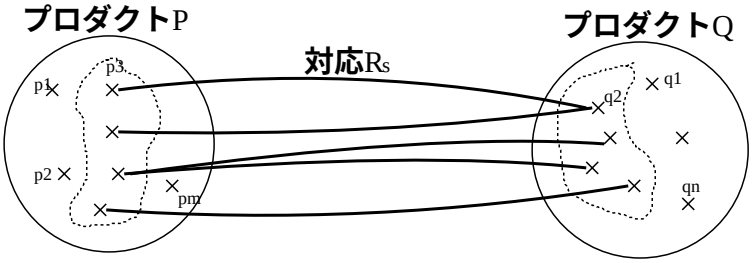

図 2: 要素の対応 R_s

3.2 類似度のメトリクス

等価な行を用いたメトリクス S_{line}

ソフトウェアシステム P, Q に対し，p_i, q_j を P, Q それぞれの各ファイルの各行とする．直感的には P，Q それぞれ各ファイルを連結した一つのファイルを考え，その各行を構成要素とする．等価な行の対応を調べることによって対応 R_s が与えられる．この類似度メトリクスを S_{line} とする．これは，ファイル名やファイルの大きさに影響されず，直感的に近い値が得られることが期待される．

その他に，いろいろ類似度のメトリクスを考えることが出来るが，求める手間やその効果を考え，ここでは S_{line} について議論する．

3.3 類似度メトリクスの計算手法

二つのソフトウェアシステム P，Q に対し S_{line} を求めるためには，R_s を得ること，すなわち P の各ファイルの各行が Q のどのファイルのどの行に対応するか，又は対応する行がないか，を知る必要がある．

このための簡単な方法としては，P の各ファイルを連結したファイル p_{all} と Q の各ファイルを連結したファイル q_{all} を作り，p_{all} と q_{all} の間の共通部分を求めて，そこに含まれる各行を R_s とすることが考えられる．共通部分の発見には，通常，最長共通部分列を発見するアルゴリズム LCS[8, 9, 11] を用いた diff[3] 等のツールが便利であるが，ファイル名が変わるなどしてファイルの連結順が変わった場合には，共通部分列として検出が出来なくなる．

そこで，ここではコードの重複（クローンと呼ぶ）を求めるためのツール CCFinder[6] と diff とを組み合わせて R_s を求める．

CCFinder は，与えられたプログラムファイルの内に存在する同じプログラム文の系列を効率よく検出し，出力する．ただし，コメントや改行，空白の違い，また，変数や関数の名前（識別子）の違いがあっても同じものとして検出する．

まず CCFinder を用いて p_{all} と q_{all} の間に存在するクローンを検出する．クローン中の各行は R_s の要素とする．CCFinder は，後述のように保守作業にとって無意味なクローンは除去されて出力される．しかし，除去されるものの中には類似度における対応としては有用なものもある．

そこで，検出されたクローンを持つファイル間に対し，共通部分列を diff によって求め，そこに含まれる各行も R_s の要素とする．二種類のツールを組み合わせることで，意味のある行の対応を効率よく求めることができる．

CCFinder では，プリプロセッサ命令（C言語の#include 行など）は検出対象から除外される．そのため，diff を用いた差分情報を加えることにより，同一行と判断できる行が増加し，より有効な行の対応が求められると考える．後節で述べる適用実験の結果，対応をもつ行は一割程度増加する．

また，diff だけを用いた対応の抽出の場合，ディレクトリ構造を保ったまま二つのソフトウェアシステムを入力とすると，二つのソフトウェアシステムの間で同一の構造を持つ必要があり，ファイル名の変更やディレクトリ構造の変化に追従できない．そのため，ここでは CCFinder と diff を組み合わせて対応を求める方法を採用する．

3.4 類似コードの対応関係の計算方法

CCFinder と diff を用いた対応の求め方の例を述べる．図 3 に示すようにソフトウェアシステム P, Q があり，P は 2 つのファイルから構成され, Q は 4 つのファイルから構成されているとする．また，Q は P の後継バージョンとする．Q を構成するファイルの中でファイル A とファイル B は，P のファイル A とファイル B を基にしており，ファイル C はファイル A を基に新しく作成されたファイルである．さらに，ファイル D は新規に作成されたファイルとする．

この二つのソフトウェアシステム P，Q の R_s を求めるために，まず CCFinder を用いて P と Q の間に存在するクローンの検出を行う．検出されたクローンからクローンを含む行の間に対応を結び R_s の要素とする．さらに，クローンが一つでも存在するファイルの組を探す．図 3 の場合，矢印で結ばれた P のファイル A と Q のファイル A，P のファイル B と Q のファイル B，P のファイル A と Q のファイル C の間にクローンが一つ以上検出されたとする．

次に，これらの 3 つの組に対してのみ diff を用いてファイル間の差分を求める．差分の結果から共通行と判断された各行も R_s の要素とする．全てのファイルの組み合わせに対して diff を実行しないため，処理時間は短くなる．また，C 言語の#include 行といった多くのファイルに存在するようなプリプロセッサ命令行の対応もクローンが検出されたファイルの組に対してのみ付けられる．そのため，単純に共通行であるからといって対応はせず，意味のある行のみが対応する．

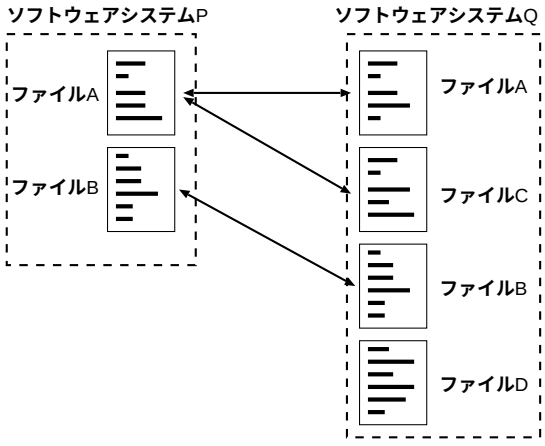

図 3: 対応の求め方

3.5 類似度測定ツール SMMT

S_{line} を求めるためのアプローチに基づき，S_{line} を計測するツールを作成している．以下に，ツールの入出力と処理の流れを述べる．図 4 に処理の流れを表す.

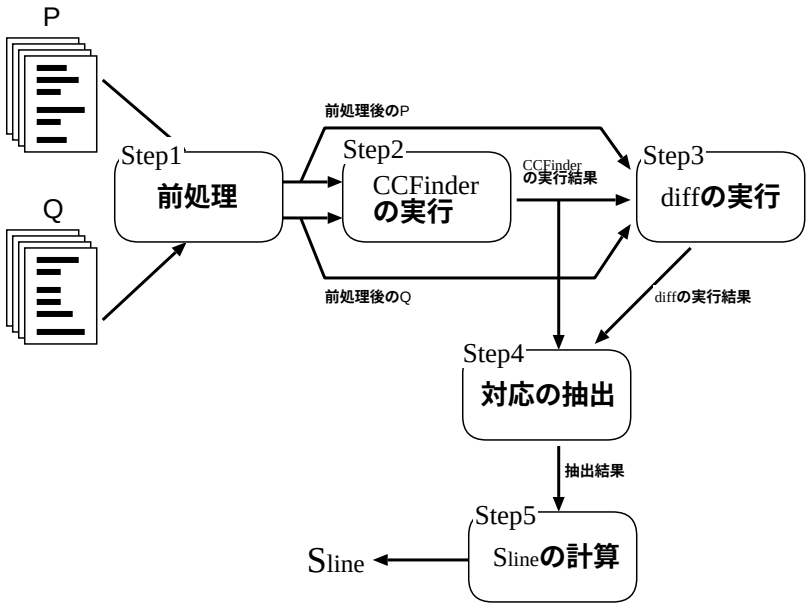

図 4: 処理の流れ

入力：二つのソフトウェアシステム P と Q
出力：P と Q の類似度 S_{line} $(0 \leq S_{line} \leq 1)$

Step1: 前処理

生成されるプログラムの機能に影響を与えない部分を取り除く．この処理は，用いられているプログラミング言語によって異なる．たとえば，C 言語で記述されたファイルの場合，コメント部分，空行をすべて取り除く．これにより，diff を実行した時の類似度の精度を向上させる.

Step2: CCFinder の実行

与えられた二つのソフトウェアシステム P と Q を入力として CCFinder を実行させる．CCFinder の実行にあたって，最低一致トークン数を 20 とした．最低一致トークン数とは，出力すべき一致するクローンが含むトークンの長さの最低値を表す．ツールのオプションとして，この値は変更可能である.

Step3: diff の実行

CCFinder の実行の結果，一つでもクローンが発見された P と Q のファイルの組の全てに対して diff を実行する．

Step4: 対応の抽出

CCFinder で検出されたクローンのトークン列から，一致している行同士を求める．さらに，diff で求まった差分情報から一致している行同士を計算する．CCFinder か diff のどちらかで一致している判断された行の組を R_s に入れる．

Step5: S_{line} の計算

S_{line} の定義より計算する．ただし，二つのソフトウェアシステムの全行数は前処理後の行数を用いる．

ツール SMMT は Windows2000 上で稼働し，その対象言語は C，C++，Java，COBOL である．例えば，二つのソフトウェアシステムの総行数が 503884 行の S_{line} を計測する場合，PentiumIII 1GHz，メモリ 2GBytes のシステムで 329 秒を要する．

4　メトリクスを用いた類似度測定手法

4.1　現状の課題

前節で述べた類似度測定手法は，一回あたりの計算コストが高い上に，新しく類似度を比較したいソースコードが現れた場合に，毎回全ソースコードに対して文字列比較するため，非常に時間がかかり，大量の比較を行うのには不向きである．

そのため，現在我々の研究チームが開発しているプロジェクト SPARS-J は，類似度を比較する対象が大量のコード群になるので，大量のソースコード間の類似度測定に特化した，高速かつ低コストな類似度検出方法の実装が必要となる．

そこで本研究では，SPARS-J が対象としている Java ソースコードについて，プログラム間の類似度を計測する手法を提案する．本手法では，構文解析時に得られる静的性質メトリクスを数値に抽象化し，それを比較して類似度を求めているため，解析コストを低く抑えることが可能となる．

以降，提案手法について説明を行う．

4.2　類似度の定義

本研究では，プログラムの類似度を比較するのに，構文解析時に予め取得しておいた静的性質メトリクスのみを用いる．プログラムの類似度を，以下の 2 つの視点から分析してそれぞれの類似度を算出し，それを掛け合わせたものを最終的な類似度とする．

- **構成トークン類似度**

  J プログラムの表層的特徴の一つとして，そのソースコードに記述されているトークンの種類別使用頻度が考えられる．ソースコードとはトークンの集合であると考えらるので，構成しているトークンの数と種類が良く似ているプログラム同士は類似度が高いといえる．

  トークンの構成類似度の分析方法として，分析プログラムの言語仕様に基づく全ての予約語，演算子，に加え，識別子の各数をメトリクスとして用いる．つまり，変数名やメソッド名は全て一つの識別子トークンという範疇に集約される．具体的な名前は変更されてもプログラムの動作に影響しないので考慮しない．

- **制御構造類似度**

  プログラムソースコードの構造的特徴を表すために，サイクロマチック数，クラス当たりのメソッド宣言数，ネストの深さなどをメトリクスとして採用し，２部品間のこれらのメトリクス値の差分が設定した閾値以内であれば類似と判定する．

4.3 類似度のメトリクス

4.2 節で定義した 2 つ類似度を算出するため，Java プログラムにおける類似度メトリクスとして，Java ソースコードのトークンの出現回数を用いたメトリクス，Java ソースコードの複雑度メトリクスを定義した．以降，この 2 つのメトリクスについて説明を行う．

- **Java ソースコードのトークンの出現回数を用いたメトリクス**
  本手法におけるトークンとは，ソースコード上での意味のある単語のこと指す．Java ソースコードのトークンは，4 種類に分けることができ，それぞれのメトリクスを以下の 1〜4 に定義する．そして，各メトリクスの値の合計を，トークンの出現回数を用いたメトリクス **Ttotal** として保存する．この時，メトリクス **Ttoken** は，プログラムのサイズのメトリクスとして捉えることができる．

  既存のメトリクス測定手法では，サイズのメトリクスに LOC(Line Of Code) が使われるのが一般的であったが，本手法ではコメントや空行，改行位置などのプログラマによる違いは，プログラムの動作には何の影響も及ぼさないノイズであるとみなし，サイズの指標としてそれらのノイズの影響を受けないトークンの総出現回数を採用している．

  これにより，性質上小さなサイズのプログラムが多く存在する，オブジェクト指向言語プログラムに対しても，本来のサイズに対するノイズの影響を最小限に抑えることが可能となる．

  1. **Java ソースコードの予約語の出現回数を用いたメトリクス**
     Java ソースコードに対して，49 種類の予約語の各出現回数を，以下の表 1〜表 5 に示すメトリクスとして記録する．
     なお，Java の予約語の中にはこれ以外に const と goto があるが，この二つには機能が実装されていないため，本手法においてはスペースや改行記号などと同じくノイズとみなし類似度メトリクスには含まない．
  2. **Java ソースコードの記号の出現回数を用いたメトリクス**
     Java ソースコードに対して，9 種類の記号の各出現回数を，以下の表 6 に示すメトリクスとして記録する．
  3. **Java ソースコードの演算子の出現回数を用いたメトリクス**
     Java ソースコードに対し，36 種類の演算子の各出現回数を，以下の表 7 に示すメトリクスとして保存する
  4. **Java ソースコードの識別子の出現回数を用いたメトリクス**

Java プログラムに対し，識別子（変数名やメソッド名など）の延べ出現回数を，メトリクス **NOidentifier** として保存する

- **Java ソースコードの複雑度メトリクス**
  Java プログラムに対し，以下の表 8 に示すメトリクスを，ソースコードの複雑さの指標として保存する．さらにそれぞれのメトリクスに対して，類似判定の際の値の差分の許容限界値である閾値を表 8 のように設定する．

表 1: 類似度メトリクス (データ型関連予約語)

メトリクス名	説明
NOvoid	予約語 void の出現数
NOnull	予約語 null の出現数
NOchar	予約語 char の出現数
NObyte	予約語 byte の出現数
NOshort	予約語 short の出現数
NOint	予約語 int の出現数
NOlong	予約語 long の出現数
NOfloat	予約語 float の出現数
NOdouble	予約語 double の出現数
NOboolean	予約語 boolean の出現数
NOtrue	予約語 true の出現数
NOfalse	予約語 false の出現数

表 2: 類似度メトリクス (クラス関連予約語)

メトリクス名	説明
NOimport	予約語 import の出現数
NOpackage	予約語 package の出現数
NOclass	予約語 class の出現数
NOinterface	予約語 interface の出現数
NOextends	予約語 extends の出現数
NOimplements	予約語 implements の出現数
NOthis	予約語 this の出現数
NOsuper	予約語 super の出現数
NOnew	予約語 new の出現数
NOinstanceof	予約語 instanceof の出現数

表 3: 類似度メトリクス (修飾子関連予約語)

メトリクス名	説明
NOprivate	予約語 private の出現数
NOpublic	予約語 public の出現数
NOprotected	予約語 protected の出現数
NOfinal	予約語 final の出現数
NOstatic	予約語 static の出現数
NOabstract	予約語 abstract の出現数
NOnative	予約語 native の出現数
NOsynchronized	予約語 synchronized の出現数
NOvolatile	予約語 volatile の出現数
NOtransient	予約語 transient の出現数
NOstrictfp	予約語 strictfp の出現数

表 4: 類似度メトリクス (制御構造関連予約語)

メトリクス名	説明
NOfor	予約語 for の出現数
NOwhile	予約語 while の出現数
NOdo	予約語 do の出現数
NOif	予約語 if の出現数
NOelse	予約語 else の出現数
NOswitch	予約語 switch の出現数
NOcase	予約語 case の出現数
NOdefault	予約語 default の出現数
NObreak	予約語 break の出現数
NOcontinue	予約語 continue の出現数
NOreturn	予約語 return の出現数

表 5: 類似度メトリクス (例外関連予約語)

メトリクス名	説明
NOtry	予約語 try の出現数
NOcatch	予約語 catch の出現数
NOfinally	予約語 finally の出現数
NOthrow	予約語 throw の出現数
NOthrows	予約語 throws の出現数

表 6: 類似度メトリクス (記号)

メトリクス名	説明
NOLPAR	記号"("の出現数
NORPAR	記号")"の出現数
NOLBRACE	記号"{"の出現数
NORBRACE	記号"}"の出現数
NOLBRACK	記号"["の出現数
NORBRACK	記号"]"の出現数
NODOT	記号"."の出現数
NOSEMICOLON	記号";"の出現数
NOCOMMA	記号","の出現数

表 7: 類似度メトリクス (演算子)

メトリクス名	説明	メトリクス名	説明
NOASSIGN	演算子"="の出現数	NOBAND_ASSIGN	演算子"&="の出現数
NOEQUAL	演算子"="の出現数	NOCONDITIONAL	演算子"?"の出現数
NOPLUS	演算子"+"の出現数	NOOR	演算子"||"の出現数
NOPLUS_ASSIGN	演算子"+="の出現数	NOBOR	演算子"|"の出現数
NOGT	演算子">"の出現数	NOBOR_ASSIGN	演算子"—="の出現数
NOLET	演算子"<="の出現数	NOCOLON	演算子":"の出現数
NOMINUS	演算子"-"の出現数	NOINCREMENT	演算子"++"の出現数
NOMINUS_ASSIGN	演算子"-="の出現数	NOBXOR	演算子"^"の 出現数
NOLT	演算子"<"の出現数	NOBXOR_ASSIGN	演算子"^= "の出現数
NOGET	演算子">="の出現数	NODECREMENT	演算子"--"の出現数
NOTIMES	演算子"*"の出現数	NOMODULO	演算子"%"の出現数
NOTIMES_ASSIGN	演算子"*="の出現数	NOMODULO_ASSIGN	演算子"%="の出現数
NOCOMPLEMENT	演算子"!"の出現数	NOSLEFT	演算子"<<"の出現数
NONOT_EQUAL	演算子"!="の出現数	NOSLEFT_ASSIGN	演算子"<<="の出現数
NODIVIDE	演算子"/"の出現数	NOSRIGHT	演算子">>"の出現数
NODIVIDE_ASSIGN	演算子"/="の出現数	NOSRIGHT_ASSIGN	演算子">>="の出現数
NOBNOT	演算子"~"の出現数	NOUSRIGHT	演算子">>>"の出現数
NOAND	演算子"&&"の出現数	NOUSRIGHT_ASSIGN	演算子">>>="の出現数
NOBAND	演算子"&"の出現数		

表 8: 類似度メトリクス (複雑度)

メトリクス名	説明	閾値
cyclomatic	サイクロマチック数	0
declamethodnbr	メソッド宣言の数	1
callmethodnbr	メソッド呼び出しの数	2
nestingdepth	ネストの深さ	0
NOclass	クラスの宣言数	0
NOinterface	インタフェースの宣言数	0

4.4 類似度の判定方法

ソフトウェアシステム P はそれを構成する要素の集合と考え，P $= \{p_1, \cdots, p_m\}$ と表現する．今 P をプログラムソースとすると，各 p_i はソースコードの各トークンの出現数となる．前節で定義した 96 種のメトリクスを基に，二つのソフトウェアシステム $P = \{p_1, \cdots, p_{96}\}$, $Q = \{q_1, \cdots, q_{96}\}$ に対し，類似の判定方法を説明する．

部品 P の全トークン数を Ttotal(P) とすると，

$$Ttotal(P) \equiv \sum_{k=1}^{96} p_k$$

と表せる．

さらに部品 P と部品 Q の各トークン数の差分の和を diff(P,Q) とすると，

$$diff(P,Q) \equiv \sum_{k=1}^{96} |p_k - q_k|$$

と表せる．

diff(P,Q) が 0 なら，全く同じ種類のトークンを，全く同じ回数用いてプログラムを構成していることになるので，コピーした部品である可能性が非常に高い．
また，diff(P,Q) が同じ 30 トークンだとしても，全トークンが 40 トークン中の差分が 30 トークンであるのと，10000 トークン中に差分が 30 トークンあるのとでは，その性質が全く違うため，非類似度を求める際には全トークン数に対する割合で求めるべきである．

そこで，部品 P と 部品 Q の非類似度 $D(P,Q)$ を次のように定義する．

$$D(P,Q) \equiv \frac{diff(P,Q)}{min(Ttotal(P), Ttotal(Q))}$$

本手法では，$D(P,Q) < 0.03$，つまり部品 P,Q の各要素の差分の合計が全体の 3%以下であるものを類似部品候補とみなす．この 0.03 という値を設定した理由は，SPARS －J において，試行錯誤の結果この値に設定することにより，希望する類似部品判定ができたという経験則からであり，類似という概念は直感的なものであるので，プロジェクトごとにこの値は随時変更されるべきものであると考える．

さらに，$D(P,Q)$ が 0.03 未満であった類似部品候補 P,Q 間の複雑度メトリクスの差分が全て閾値以内であれば，最終的に P,G は類似部品であると判定する．

4.5 主メトリクスを用いた効率化手法

前節では類似度判定の方法を述べたが，新しい判定をする度に，毎回全ソースコードに対して類似度判定を行うのはコストが高くなる．そこで，まず全メトリクスの中から，重要なメトリクスを主メトリクスとして採用する．そして，図 5 のように各部品と，その部品が持つ主メトリクス値のハッシュ値とを対応づけたデータベースを構築する．そのハッシュ値が閾値以内である部品だけを対象として，前節の類似度判定を行うようにすれば，判定結果を変えることなく測定の効率のみが上がる．

主メトリクスの候補として，まず閾値をもつメトリクスである複雑度メトリクスが挙げられる．複雑度メトリクスの値の差分が閾値以内であるかどうかは，類似判定の絶対条件であるので，複雑度メトリクスはその閾値とともに主メトリクスとして採用する．

しかし，複雑度メトリクスだけを主メトリクスにすると，単純な構造の部品がすべて同じハッシュ値をもってしまい，分類ができずに大きな偏りができる．そこで，部品のサイズを表すメトリクス **Ttotal** を加工して主メトリクスとして採用することにより，効率的に部品を分類できる．

そこで，まず **Ttotal** が関係する類似判定の条件について考えてみる．前節の類似度判定条

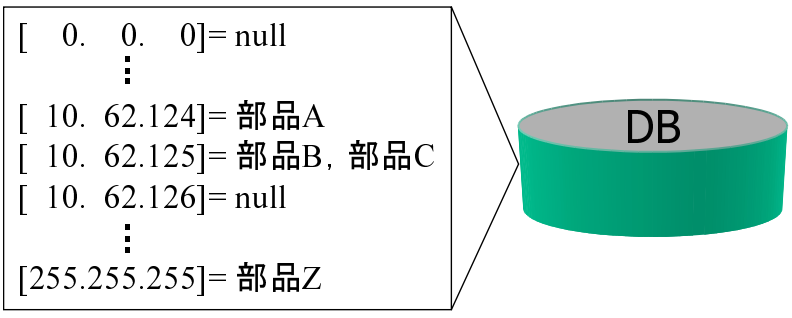

図 5: ハッシュ値と部品の対応づけデータベース

件の一つに相違トークン数の合計が**Ttotal**の3%以内であると定義した．この時，Ttotal(P)とTtotal(Q)の絶対値の差が3%を超えるもの(ここではTtotal(P) $\geqq$ Ttotal(Q)と仮定する)は，たとえQの全トークンの各値がPの各値と同じか小さかったとしても，Ttotalの差の数は全て差分となるので，最終的に差分は全体の3%を超え，類似度判定をしても類似でないと判定されるはずである．

このことから，主メトリクスを用いてまずTtotalの差が3%以内の部品だけを抽出すれば良いと考えられる．

具体的な実現方法として，図6のようにTtotalを関数 $f(n+1) = \lceil f(n) * 1.04 \rceil$ (ただし $f(0) = 0, f(1) = 1$) を満たす関数f(n)の値ごとに区切っていき，それぞれの範囲を群に分割する．この時f(n)とf(n+1)に挟まれた群を群番号nの群と呼ぶ．

閾値が3%というような可変値である場合にはそのままの値でなく，加工した値を主メトリクスとして使わなくてはならない．今回の場合はTtotalを群番号に加工して閾値が1である主メトリクスとして使用している．

4.6 類似度測定ツール

提案手法を基にツールを実装した．ツールは図7のような構成になっている．

類似度を判定したいJavaのソースコードを入力する．構文解析部は入力された各ソースコードの類似度メトリクスを計測し，部品のIDとともにデータベースに保存する．同時に，主メトリクスによるハッシュ値も計算し部品IDと対応させて記録しておく．

メトリクス値の計測が終わると，類似度計測部ではハッシュ値が閾値以内である部品を抽出し，それらのメトリクス値から類似度測定を行って類似部品とされた部品対を部品群としてまとめる．この際，類似部品が無かったものは単独部品群として扱う．そして，最終的に部品群のデータを出力する．

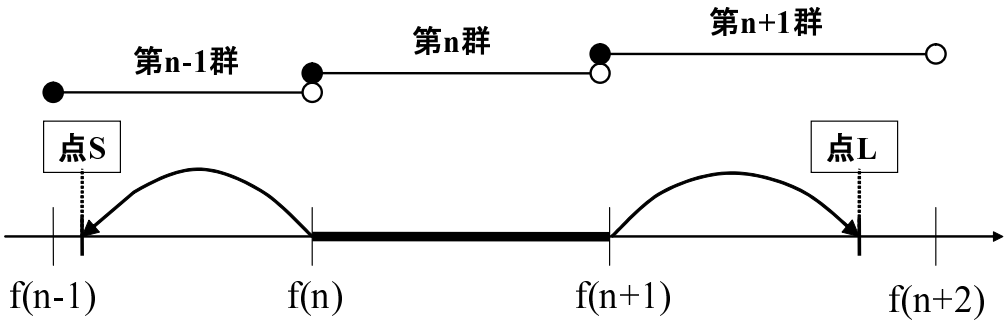

図 6: Ttotalの群分割

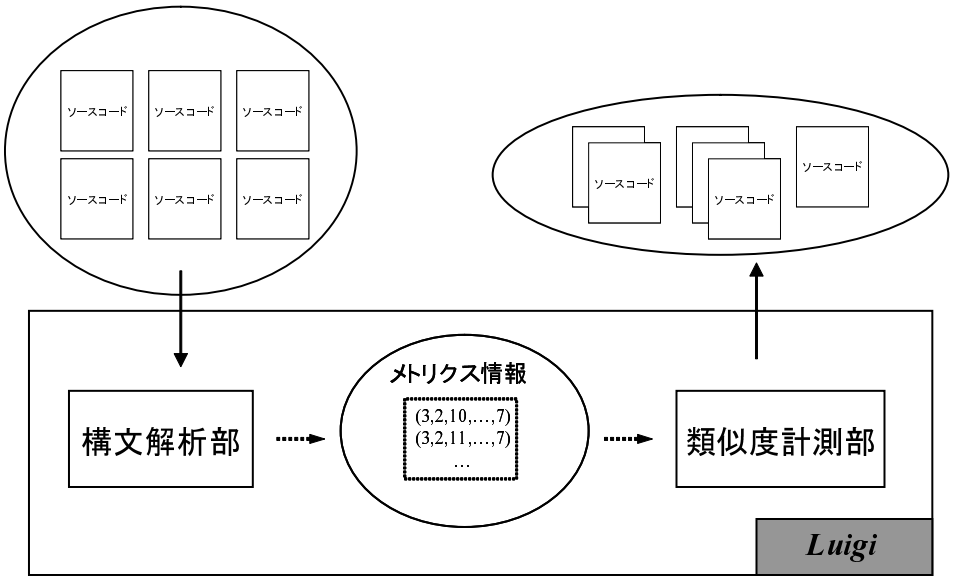
ソースコード
ソースコード
ソースコード
ソースコード
ソースコード
ソースコード
ソースコード
ソースコード
ソースコード
メトリクス情報
(3,2,10,…,7)
(3,2,11,…,7)
…
構文解析部
類似度計測部
Luigi

図 7: ツール構成

5 検証結果とその分析

本節では，提案手法を実現したLuigiを用いて適用実験を行い，提案手法の有効性を評価する．実験においては，Sun JDK 1.3内の431クラスに対してLuigiを適用する．主メトリクスを定義する事で適用対象を絞り込む事ができるが，実験においては，次のよう4通りの定義を用意し，それぞれに条件においてLuigiを適用した．

1. 主メトリクスを定義しない

   この場合，一つの部品について全てのクラスに対して判定を行う．

2. サイクロマチック数による分類

3. サイクロマチック数およびメソッド数による分類

   これらの場合，制御構造類似度の一部を満たしている部品のみが抽出される．

4. サイクロマチック数およびメソッド数およびクラスタによる分類

   この場合，制御構造類似度による基準に加えて，構成トークン類似度を満たす可能性がある部品のみが抽出される．

適用実験の結果を表9にしめす．表中の主メトリクスCはサイクロマチック数，Mは宣言メソッド数，Tはトークン数による分類が行われたことを指し，それぞれ前述した分類方法に対応している．

クラスタ分類数とは主メトリクスを用いて分類したときにいくつの部品群に分かれたかを指す．未使用の場合は，主メトリクスにより分類されなかったとし，1としている．Cの場合は，431のクラスが21のグループに分類され，各グループは平均21個の部品から構成されている．以下，分類条件が細かくなるほど，細かく分類されることがわかる．

最終クラスタ数は，本ツールを用いて解析を行った結果，いくつの類似部品群に分かれたかを示している．主メトリクスにおいて採用された分類条件が制御構造類似度および構成

表 9: 適用実験結果

主メトリクス	クラスタ分類数	最終クラスタ数	計算コスト
未使用	1	278	05.02
[C]	21	278	00.56
[C,M]	85	278	00.29
[C,M,T]	232	278	00.16

トークン類似度の判定において，類似であるとみなすのに必要な条件を破綻させるものではなかったので，最終クラスタ数は全て同じとなった．計算コストは，本ツールの開始から終了までの時間を計ったもので，構文解析におけるメトリクス抽出時間は含んでいない．

以下では，解析結果の妥当性および解析コストの考察を行う．

5.1 妥当性の考察

本節では，解析結果の妥当性について考察する．

まず，最終クラスタ 278 個を手作業で調べたところ，今回要求に満たすようなコピーして一部変更した部品が群としてまとまっていることが確認された．また，文字列比較を用いた類似度測定ツール SMMT との結果比較においても，精度の差は多少あるものの，高い相関が得られた．これにより，ソースコード部品がコピーされたかを判定するためには，文字列を直接比較する手法だけではなく，ソースコード中の構成トークンの類似度および制御構造の類似度を測ることでも十分であることが確認できた．

また，主メトリクスを細かくしていくにつれて，類似度クラスタは増加していくが最終クラスタ数は変化していない．さらに，各パターンにおいてそれぞれ 278 個のクラスタの構成部品を調べたが全て同じであった．これは，主メトリクスにおいて採用された分類条件が制御構造類似度および構成トークン類似度の判定において，類似であるとみなすのに必要な条件を破綻させるものではなかったからであると考えられる．実際には，主メトリクスを細かくしていくことは，主メトリクスじの絞込みにおいて前倒しで判定を行っていることに他ならない．主メトリクスの判定条件を十分考慮する事で計算コストを下げる事ができ，効率的に判定を行うことができることを確認した．

5.2 解析コストの考察

本節では，解析コストについて考察する．主メトリクスを細かく分類する事で，解析コストを大きく削減できる事がわかる．実際の判定回数が，絞込みを行った場合の主メトリクスを用いて分類したときの各グループの平均部品数にほぼ比例しているという事がわかる．実際には，各グループの部品数にばらつきがあり，きれいな比例関係とはならないが，主メトリクスの分類によって必要な実行時間が大体予想できる．

また，JDK の 431 クラスに対して文字列比較を用いた類似度測定ツール (SMMT) を用いて解析した時のコストが 24.35(sec) であった．この手法の場合，構文解析を前もって行う必要は無く，メトリクス抽出のコストを加味しないと正確な比較を行うことはできない．しかし，SPARS-J においては，類似度測定ツールの種類に関わらず利用関係を抽出するために必ず構文および意味解析が必要となる．さらに，構文解析ルーチンにおけるメトリクス抽出

の時間は微々たる物であるため，メトリクス抽出のコストを考慮する必要はないと考えられる．

以上の事を考えると，主メトリクスによる分類を行った後にメトリクス比較により類似度を判定する手法は，文字列比較を用いた類似度測定ツールに比べて，時間コストを 1/150 にまで下げることができているといえる．さらにこの手法の場合，分類を行うべきソフトウェア部品の数が多くなった場合，主メトリクスにおいて分類条件を追加する事で，解析対象をより狭くする事ができる．これにより，更なる効率化を図る事ができ，大規模な部品の集合にも対応が比較的容易である．このことから，SPARS-J においては，本提案手法は文字列比較を用いた類似度測定手法よりも大きな威力を発揮すると期待できる．

主メトリクスにおいて，前倒しで判定をし絞込みを行う事で，効率的に判定を行うことができることを確認した．

以上のことから，本手法は文字列比較を用いた類似度測定手法よりも十分低コストかつ効率的な手法で，主メトリクスにおいて，前倒しで判定をし絞込みを行う事で，効率的に判定を行うことができることを確認した．

6　まとめと今後の課題

本研究では，Java ソースコードの部品情報を数値化した類似度メトリクスの比較を用いた類似度測定方法を提案した．さらに，ソフトウェア部品の中から類似した部品を検出し，部品群にまとめる機能を実装したツールの試作を行った．その中で，主メトリクスによる事前分類を導入し，測定回数の効率化も行った．最後に JDK 1.3 のソースコードから 431 クラスを対象とした適用実験により既存の文字列比較を用いた測定手法よりも十分低コストかつ効率的な手法であることを確認した．

今後の課題としては，さらに多くの視点から類似度を測定し，その精度を上げていくこと，Java 以外の言語への対応，さらには SPARS-J に実装した類似度測定ツール *Luigi* の保守などがある．

謝辞

本研究において, 常に適切な御指導および御助言を頂きました大阪大学情報科学研究科コンピュータサイエンス専攻 井上 克郎 教授に深く感謝致します.

本研究において, 適切な御指導および御助言を頂きました 同 楠本 真二 助教授に深く感謝致します.

本研究において, 適切な御指導および御助言を頂きました 同 松下 誠 助手に深く感謝致します.

本研究において, 多大な御指導および御助言を頂きました科学技術振興事業団計算科学技術研究所 山本哲男 研究員に深く感謝致します.

本研究において, 多大な御指導および御助言を頂きました 大阪大学大学院基礎工学研究科情報数理系専攻博士後期課程 2 年 横森 励士 氏に深く感謝致します.

最後に, その他様々な御指導, 御助言を頂いた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 井上研究室の皆様に深く感謝いたします.

参考文献

[1] V. R. Basili, G. Caldiera, F. McGarry, R. Pajerski, G. Page and S. Waligora: "The software engineering laboratory - an operational software experience", in Proceedings of 14th International Conference on Software Engineering (ICSE14), pp. 370-381, Melbourne, Australia, 1992.

[2] C. Braun: Reuse, in John J. Marciniak, editor, *Encyclopedia of Software Engineering*, Vol. 2, John Wiley & Sons, pp. 1055-1069, 1994.

[3] Diffutls: "http://www.gnu.org/software/diffutils/diffutils.html".

[4] K. Inoue, R. Yokomori, H. Fujiwara, T. Yamamoto, M. Matsusita and S. Kusumoto: "Component Rank: Relative Significance Rank for Software Component Search", to be appeared in Proceedings of 25th International Conference on Software Engineering (ICSE 2003), Portland, Oregon, 2003.

[5] S. Isoda: "Experience report on a software reuse project: Its structure, activities, and statistical results", in Proceedings of 14th International Conference on Software Engineering (ICSE14), pp.320-326, Melbourne, Australia, 1992.

[6] T. Kamiya, S. Kusumoto and K. Inoue: "CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code", IEEE Transaction on Software Engineering vol. 28, no. 7, pp. 654-670, 2002.

[7] B. Keepence and M. Mannion: "Using patterns to model variability in product families", *IEEE Software*, Vol. 16, No. 4, pp. 102-108, 1999.

[8] W. Miller and E. W. Myers: A file comparison program. *Software- Practice and Experience*, Vol. 15, No. 11, pp. 1025–1040, November 1985.

[9] E. W. Myers: An $O(ND)$ difference algorithm and its variations. *Algorithmica*, Vol. 1, pp. 251–256, 1986.

[10] SourceForge: "http://sourceforge.net/".

[11] E. Ukkonen: Algorithms for approximate string matching. *INFCTRL: Information and Computation (formerly Information and Control)*, Vol. 64, pp. 100–118, 1985.

[12] 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 D-I, Vol. J85-D-I, No.6, pp.503-511, 2002.