ソフトウェア工学 118－15
（1998．3．11）

C++を対象としたオブジェクト指向メトリクスの検討

直田繁樹　　中島哲　　堀田勇次
（株）富士通研究所 マルチメディア研究所
{suguta,nakasima,hotta}@flab.fujitsu.co.jp

C++言語によって記述されたオブジェクト指向ソフトウェアを評価するために，従来，提案されているメトリクスに加えて，C++における非オブジェクト指向要素の比率を測定することを提案している．これにより，C言語からの移行において発生する手続き的設計の傾向を検出できると考えた．また，オブジェクト指向フレームワークの評価を行なうために，従来のクラスを単位としたメトリクスに加えて，モジュール単位の参照を測定するメトリクスも提案する．従来のメトリクスに合わせて，これらのメトリクスを測定するツールを開発し，大規模なシステムを含む複数のプログラムで計測実験を行なった．その結果，非オブジェクト指向要素の比率は開発者の習熟度の目安として有効であること，及びモジュール単位の参照のメトリクスはフレームワークの改善にデザインパターンを適用した場合に有効であることがわかった．

An Object Oriented Software Metrics Suite for C++

Shigeki Suguta　　Satoshi Nakashima　　Yuuji Hotta
Fujitsu Laboratories Ltd., Multimedia Systems Laboratories

We introduce a new object-oriented metrics suite for measuring C++ program. In addition to well-known OO metrics, we focused on the rate of non-OO features in C++ which often occur when traditional C programmers start writing C++ programs. We also introduce some metrics for measuring inter-module relationship to evaluate the design of an object-oriented framework . We developed a tool for measuring these metrics, and measured some programs, including large scale one. From the result, we show that non-OO rate is effective for estimating the skill level of the developer and that the inter-module metrics is effective when some design patterns are applied to a framework in order to improve it.

1　はじめに

近年，オブジェクト指向によるソフトウェア開発手法／環境の急速な普及に伴い，その評価を行なうためのオブジェクト指向のメトリクスの必要性が大きくなってきている．そのようなメトリクスとして，既に様々な提案がなされており [1], [2], Chidamber と Kemerer のメトリクス (以下 CK メトリクスと略す) [3], や Lorentz らの著書 [4] がよく知られている．これらのメトリクスは，オブジェクト指向一般を扱ったもので，実現するプログラミング言語を問わないものである．

しかし，Smalltalk, C++, Java といった代表的なオブジェクト指向プログラミング言語は，それぞれ独自の特徴を強く持っており，その結果，プログラムやクラスの構造も異る特徴を示すことが知られている．

こうしたことから，実際のソフトウェア開発の指針とするためには，プログラミング言語の特徴を考慮し

たメトリクスが必要であると考えられる.

また, 従来のメトリクスでは, 主としてクラスの構造やクラス間の関係のみを議論しているが, 複数のクラスからなるオブジェクト指向フレームワークの設計を評価する場合には, より大きな単位でのメトリクスがなければ不十分である.

そこで, 我々はこれらの点を補う新たなメトリクスを提案し, そのための測定ツールを開発した. そのツールを用いて, 大規模なシステムを含むプログラムについて測定を行ない, その有用性を確認した.

2 既存のオブジェクト指向メトリクス

先にも述べたように, 代表的なオブジェクト指向のメトリクスとして, CK メトリクスがある. これは以下の 6 つの項目からなっている.

- 重み付きメソッド数 WMC
  クラスの規模を表す. 文献 [3] では重みは付けず, 単にメソッド数をカウントしている.
- 継承の深さ DIT
- 子供の数 NOC
- クラス間結合度 CBO
  メッセージ送信しているクラスの数. クラス間の相互作用の程度を表す.
- メッセージ応答数 RFC
  メソッド数とそこから呼び出すメソッドの数の和.
- クラス凝集度の欠如度 LCOM
  インスタンス変数を共有しないメソッドペアの数 − インスタンス変数を共有するメソッドペアの数 (但し, 負になる場合は 0), として定義されており, クラスのまとまりのなさを示す尺度になっている.

これらのメトリクスのうち, LCOM を除く 5 つについてはエラー発生数の予測に役立つことを実験的に示した研究もある [5].

その他の提案も含め, 一般に, 既存のメトリクスは, 以下のように分類できると考える.

- クラスのサイズの情報
  メンバ数, WMC 等.
- クラスの内部構造の情報
  LCOM, メンバ関数の複雑度 等
- クラスの継承構造の情報
  DIT, NOC, 親の数, オーバライドの数, 多相度 等
- クラスのインタフェースの情報
  RFC , 公開メンバ関数の数 等
- クラス間の相互作用の情報
  CBO, 参照クラス数 等

我々は, C++言語においては, これらに加えて, 以下の点を考慮すべきであると考えている.

- 非オブジェクト指向部分を含む言語である.
  C 言語に対してほぼ上位互換であり, オブジェクト指向的でないプログラムを書くことが可能な言語仕様になっている. クラス以外のグローバル定義 (グローバル関数, グローバル変数, 列挙型, typedef) が任意に記述できる. このため, コンパイラのみ C++であっても, 実態は C 言語のプログラムになっている場合も多い. これらの存在を明らかにする必要がある.
- 独自の言語要素が存在する.
  一般的なオブジェクト指向の枠組からは外れるが, テンプレート, 演算子多重定義, デフォルトの型変換など, オブジェクト指向的特徴と密接に関連して, プログラムの生産性や複雑さに影響を与える言語要素がある. 特にクラステンプレートの利用は, C++のプログラムでは重要なキーになると思われる.

C++言語向けのメトリクスとして, 金らの提案 [7] がある. しかし, これは純粋なオブジェクト指向プログラミングが行なわれていることを前提しているものであり, 上記の点をカバーしてはいない.

また, 従来のメトリクスでは, クラス構造やクラス間の関係のみを議論しているが, 実際に有効な再利用を行なうためには, 複数のクラスからなるフレームワークの設計が重要である.

フレームワークの評価方法として, その成熟化を抽象化の過程としてとらえ, それを評価するメトリクスの提案がある [8]. しかし, これは, 従来のメトリクスの一部を選択して, 時間的変化を見ているものであり, 個々のクラスの構造やクラス間の関係を評価しているに留まっている.

我々は, フレームワークを論じるには, クラス間の関係のみでは不十分であると考え, フレームワークを構成するモジュールの相互作用を調べるメトリクスが必要であると考える.

3 メトリクスの提案

上記の議論を踏まえて, 我々は以下のメトリクスを測定することを提案する.

3.1 非オブジェクト指向部のメトリクス

非オブジェクト指向部 (以下非 OO 部と略す) のメトリクスとして, 以下のものを導入する.

- モジュール単位での非 OO 定義のメンバの比率
- モジュール単位での非 OO 定義の文の数の比率

ここで, 非 OO 定義とは, クラス以外のグローバル定義 (グローバル関数, グローバル変数, 列挙型, typedef) のこととした. これらは, その定義が所属するファイル毎にまとめた. サイズ等の情報は, このファイルの単位をクラスと同等の単位として計測を行なった.

非 OO 定義のメンバの比率は, これらの非 OO 定義の総数がクラス定義のメンバの総数の何%になるかを計算したものである.

非 OO 定義の文の数の比率は, グローバル関数の文の数の和がメンバ関数の文の数の和の何%になるかを計算したものである.

また, モジュールの単位は, 測定者が特定のモジュールに属するファイル (ヘッダ, ボディの両方) を指定し, その中に含まれる諸定義がそのモジュールに属するものとした. 実際の測定では, 同一のディレクトリに属するファイルをモジュールとした場合と Makefile のターゲットとしてまとめられているファイルをモジュールとした場合がある.

さらに, クラスの定義においても, 非オブジェクト指向的特徴をよく表すものとして, 以下の項目もメトリクスとする.

- 公開データメンバの数
- モジュール単位でのデータメンバのみのクラス (以下非ＯＯ構造体を称す) の比率

これらはデータ抽象化の考え方ができていないことの目安になる.

3.2 クラステンプレートのメトリクス

クラステンプレートの定義及び利用について, 以下の項目をメトリクスとする.

- モジュール単位でのクラステンプレートの定義数
- 通常のクラスと同じメトリクス
- クラステンプレートのインスタンス数

但し, これらについては, 計測ツールの開発が間に合わなかったため, 測定実験に含めることができていない.

3.3 モジュール間相互作用のメトリクス

モジュール間相互作用のメトリクスとして, 以下のものを個々のクラスについて測定する.

- 他モジュールクラスの参照数 (クラス数, 出現数)

参照数としては, 一つのクラス中で同じクラスが複数回参照された場合を区分しないクラス数と区分する出現数に分けて計測している.

また, 2 つのモジュールを特定して, 以下の項目を測定する.

- 相手モジュールクラスからの被参照数 (クラス数, 出現数)

フレームワークの評価に用いる場合, 被参照モジュールがフレームワークであることを意識している.

なお, モジュール間の参照については, 継承による参照も含めるべきであったが, 今回の測定では行なえなかった.

4 測定実験

4.1 測定した従来のメトリクス

測定実験では, 上記の提案に加えて, 従来のメトリクスから次のものを選び, 測定を行なった.

- メンバの数
- メンバ関数の数 (WMC)
- クラス中のメンバ関数の文の数の和
- メンバ関数の文の数の関数あたりの平均値
- メッセージ応答数 (RFC)
- 直接の派生クラスの数 (NOC)
- 継承の深さ (DIT)
- 参照クラス数

手続き部のサイズを知るため, 文の数を用いている. ループ数や関数呼出し回数などを考慮した複雑度は用いなかった. これは実際に測定したところ, ほぼ同様の傾向を示すことが分かったため, より直観的な文の数を採用した.

また, 相互作用の情報としては, CK メトリクスの CBO ではなく, 参照クラス数を用いた. ここでは, 直接メッセージ送信（メンバ関数呼出し）をしていなくても, パラメタや戻り値の型, クラステンプレートのクラス引数に現れるクラスを含めている.

さらに, CK メトリクスの LCOM は含めなかった. この理由は, このメトリクスの定義が他のものに比べて複雑で直観的でないことと, かつ, 我々の重点は, より大きなモジュール単位の評価にあり, クラス内部の結合度を調べることは重視しなかったためである.

4.2 非OO部に関する測定実験

大規模なシステムを含む 4 つのプログラムを対象として, 上記のメトリクスのうち, モジュール間相互作用のメトリクスを除いて測定を行なった.

4.2.1 測定対象

表 1 に測定を行なった対象の規模を示す. それぞれのプログラムの内容は次の通りである.

A は 大規模な設備管理システムの一部である. このシステムには, C++以外の言語 (主に K&R C) で書かれた部分も多数あったが, それは含んでいない. また, UI 部も測定対象には含まれていない. なお, これは, オブジェクト指向に不慣れな開発者を多数含むプロジェクトであった.

B は, 本実験のメトリクス測定ツールとそのベースになっている C++解析部である. これは yacc による文法記述を含んでいるが, その部分は, yacc で生成されたパーサの C++プログラムを測定の対象とした.

C は, COBOL 言語の解析ツールの一部である. これも, C++以外で書かれたプログラムや自動生成コードを含んでいるため, その部分は除外して測定した. これにも, yacc 文法を含んでいたが, その部分は, 他の自動生成処理と関係があったため, 測定対象にはしなかった.

ここで, B と C は, 主に研究所内で開発したもので, オブジェクト指向開発には比較的習熟した開発者によるものであった.

D は, 著名なフリーソフトの汎用クラスライブラリ, NIHCL3.0 である. このライブラリは, C++によるオブジェクト指向プログラムとしては, 洗練されたものといって良いと考えている. 但し, テンプレートが言語仕様に含まれる前に実装されたクラスライブラリであるため, やや特殊な構造をしていることには注意が必要である.

表 1 において, クラス数は, 非 OO 構造体を含めた数字になっている. 非 OO 定義数としたのは, クラス以外のグローバル定義をそれが所属するファイル毎にまとめたものの数である.

名称	行数	クラス数	非 OO 定義数	モジュール数
A	441K (一部重複)	333	208	75
B	28K	133	28	4
C	32K	239	87	6
D	14K	67	39	1

表 1: 測定対象の規模

4.2.2 測定結果

表 2 に測定結果を示す.

ここで,「文の数の和」はメンバ関数毎の文の数の和を,「関数の文の数」はメンバ関数当たりの文の数の平均値,「非OOメンバ比」は, 非 OO 定義の総数のクラスのメンバ総数に対する比率,「非OO文数の比」は, 非 OO 定義の文の数の総数とクラスのメンバ関数の文の数の総数の比率,「公開データ数」は公開データメンバの数,「非OO構造体比」は, クラスの総数中の非OO構造体の比率である.

なお, 測定はモジュールを単位として行なったため, 派生クラスが他のモジュールに含まれている場合は, そのクラスの子供の数としては数えられていない.

また, モジュール間のメトリクスについては, この実験では評価の対象とはしなかった. その理由は, D のように 1 モジュールしかないものも含めて, モジュールの意味がそれぞれで異っていたためである.

4.2.3 考察

まず, 個別のプログラムについて, 測定結果を検討する.

メトリクス	A	B	C	D
メンバ数	11.6	12.7	10.6	32.2
同最大	204	61	63	84
メンバ関数の数	6.0	9.9	8.4	29.3
同最大	108	55	58	80
文の数の和	218.6	34.1	27.8	72.8
同最大	7131	477	388	258
関数の文の数	36.4	3.5	3.3	2.5
子供の数	0.02	0.54	0.50	0.93
同最大	2	14	6	19
継承の深さ	0.58	0.86	1.09	2.16
同最大	3	3	5	6
RFC	82.5	29.0	19.3	66.4
同最大	2948	361	184	213
参照クラス数	5.7	4.6	3.2	5.5
同最大	84	24	33	14
非ＯＯメンバ比	28%	7%	10%	9%
非ＯＯ文数の比	14%	38%	16%	5%
公開データ数	3.0	0.1	0.5	0.3
同最大	55	3	33	4
非ＯＯ構造体比	42%	2%	9%	1%

表 2: 測定結果

A では, クラスの手続き部のサイズが他に比べて非常に大きい. また, 公開データメンバの数や非ＯＯ構造体の比率が非常に多い. 全体として, 手続き指向の作りになっており, データ抽象化が充分できていないことが伺える.

また, 継承については, 深さの平均値はあまり小さくないが, 子供の数が非常に小さい. これについて, 内容を調査してみたところ, システム全体で共通のパターンを採用しているため, 少数の共通クラスを継承しているクラスが多数あることがわかった. その共通クラスが個々の測定対象モジュールには含まれないことが原因であった.

B と C は, 共通する点が多い. 処理の対象が類似していることもあるが, オブジェクト指向開発に慣れた開発者が作成したソフトウェアが類似した結果を示したとも考えられる.

異る点としては, まず, B で, 非ＯＯ部の複雑度が非常に大きくなっている点がある. これは yacc が生成したプログラムを含めていることとそれに付随する部分の設計が手続き中心的なものになっていることに原因があった.

一方,C では公開データメンバの数がやや多い傾向がある. これは特定のモジュールに, 非オブジェクト指向的なプログラムが集中していたためであった. その部分は, オブジェクト指向開発に不慣れな開発者が担当していたことがわかった.

D は, メンバ関数の数は非常に大きい. これは汎用ライブラリであるため, 豊富なインタフェースをもたせる必要があったためと考えられる. 一方でメンバ関数の文の数の平均は最も小さい. しかし, B や C とは大きな差はない.

また, 継承が深く, 子供の数が多い. この理由は, 大部分のクラスが共通ベースクラスをもち, かつ, その共通クラスが測定単位のモジュール内に含まれていたからであった. なお, これは, テンプレート機能のない初期の C++において, リストなどのコンテナクラスを作成するための一つの手法であった.

全般には, オブジェクト指向の習熟度という観点から以下のことが言えると考えられる.

- 非ＯＯ部の比率や公開データメンバ数を計るのは習熟度の目安になる.
  但し, 非ＯＯ部は必然性がある場合もある. 例えば, yacc を使ったためなどである.
- クラス中の文の数に関するメトリクスも習熟度の目安になる.
  大きければ, 手続き中心の構造になっていることがわかる.
- 継承のメトリクスは評価が難しい.
  単純には, 継承が多ければ, ＯＯに習熟し, よく再利用が行なわれていると言えそうだが, 継承の使い方には, インタフェースの統一という側面もあるため, 内容抜きで再利用ができているとは言えない. 例えば, D は共通ベースクラスがあるため, 子供の数が多い. 一方, A は継承を使うパターンを使っているので, 実態よりも継承の深さは大きめの値になっていた. しかし, これらは直接には再利用には関係がなかった.

また, 従来のメトリクスについて, 以下のような特徴が見られた.

- RFC は文の数に比例するような傾向が見られた.
  図 1に示すように, はっきりした相関関係が見ら

れた. この図では, 傾向を明らかに示すために, 極端に手続きが大きい A の結果を除いているが, A の場合もほぼ同じような傾向が出ている. RFC は, 設計初期の段階でも概数を計測できるので, サイズを表すために, その文の数の代替として使えると言えるだろう.

- 参照クラス数の評価も難しい.
  この実験の範囲では, サイズとの比較の上で参照クラス数に異常な値を示すものは見られなかった. このため, このメトリクス独自の有効性が確認できなかった. また, B において比較的大きな値を示したクラスは, 後述する Visitor パターンや Builder パターンを用いている場合であった. これは, クラス間の依存関係を弱めるために用いたものであるが, 参照クラス数の点では逆に大きくなってしまっている.

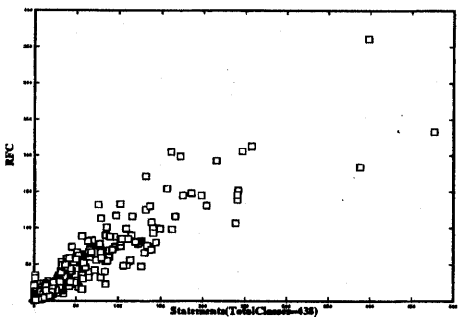

図 1: 文の数と RFC の相関関係 (A を除く)

なお, これらの結果を, A のシステムの開発担当者に説明したところ, 以下のような意見を得た.

- メトリクスが異常な値を示しているクラスは保守時に注意して対処する.
- 問題は分かったが, 測定結果を見て作り直すのは大規模プロジェクトでは現実的には難しい. プログラムができてから計るのでは間に合わない.

4.3 デザインパターン適用とフレームワーク評価の実験

デザインパターンを適用することで, フレームワークの構造を改善した場合に生じるメトリクスの変化について測定実験を行なった.

4.3.1 測定対象

先の実験で B としたメトリクス測定ツールを対象とした.

この測定ツールは, C++の構文解析をするパーサ部, その結果を保持してツール開発のための API を提供する内部モデル (フレームワークに相当), 及びそのアプリケーションであるメトリクス測定部 (MET) から成る. 内部モデル部は, さらに, 手続き部を表現する抽象文モデル部 (STM), と, シンボル, スコープ, 構文木等を表す基本部とに分かれている. 以上, 全体で 4 つのモジュールで構成されている.

STM 部は MET 部作成のために内部モデルに追加したものであり, MET 部と同一の開発者が並行して開発した. そのため, 初期の段階では, それらの機能分担やクラス構成の整理が不十分であった. STM 部をフレームワークとして利用していくためには, その構造を改善する必要があった. その手段として, デザインパターンを適用した.

ここで, 適用したパターンは Gamma らの著書 [6] における Builder と Visitor である.

Builder は, 一連のクラス群の生成処理をそのクラス群の構造から独立させる効果がある. これは, 基本部に依存する STM 部の生成処理を, STM 部のクラス構造から独立させ, ひいては, 基本部と STM 部の独立性を高めることを目的として採用した.

一方, Visitor には, 一連のクラス群に適用される, 応用に依存する処理を分離し, まとめる手段である. これは, 最初, STM 部の各クラスのメンバ関数として組み込んでいた手続き部の複雑度計算処理を, MET 部を移行するために用いた.

表 3に, まず, 全体及び各部の規模の変化を示す. ここで, B0 がこのツールの初期版であり, B はその約 4ヶ月後の版である.

4.3.2 測定結果

まず, 表 4 に全体のメトリクスの変化を示す.

この表で,「外部参照比率」とは, モジュール外部のクラスの参照が全体の参照クラス数に占める割合である.

表 5 には, STM 部のメトリクスの変化を示す.

また, 図 2には, STM 部での, MET 部から参照されるクラス出現数の分布を B0,B, それぞれの場合について示している.

名称	行数	クラス数	非OO定義数
B0	27.1K	120	26
B	28.2K	133	28
STM(B0)	4.3K	24	3
STM(B)	3.7K	28	3
MET(B0)	3.9K	13	4
MET(B)	5.2K	23	4

表 3: B とそのモジュールの規模の変化

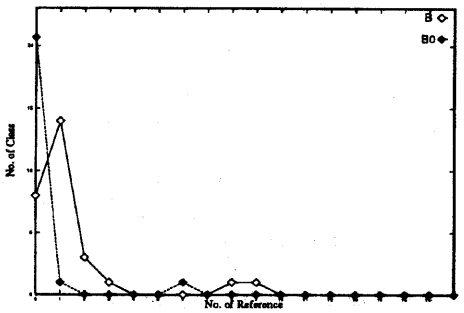

図 2: STM 部の被参照出現数の分布

メトリクス	B0	B
メンバ数	12.5	12.7
同最大	143	61
メンバ関数の数	9.6	9.9
同最大	103	55
文の数の和	38.4	34.1
同最大	470	477
関数の文の数	4.0	3.5
子供の数	0.57	0.54
同最大	14	14
継承の深さ	0.92	0.86
同最大	3	3
RFC	30.4	29.0
同最大	385	359
参照クラス数	4.1	4.6
同最大	22	24
モジュール外参照	1.5	1.5
同最大	16	19
外部参照比率	37%	33%

表 4: B 全体のメトリクスの変化

4.3.3 考察

参照クラス数を見ると, 全体でも STM でも増加している. 特に, 参照クラス数の最大値は大幅に増加している. これに対応するクラスは, 先の実験 1 の結果でも述べたように, Builder として導入したクラスであった. このメトリクスだけを見ていると, デザインパターン適用は複雑度を増しただけのように見える.

しかし, モジュール外参照やその比率を見ると, B 全体ではそれほど変化していないが, STM では減少していることがわかる. これは Builder パターンの適用が期待した効果を上げたことを示している.

また, Visitor パターンにより, 本来 MET 部に属するべき処理を移動させたことによって, クラスサイズが減少している. Builder に生成処理を集中させたことも個々のクラスサイズには影響しているが, 平均値では Builder クラスのサイズの増大で相殺されている. また, このため, 文の数の最大値は大幅に増加している.

図 2の MET 部からの被参照の状況を見ると, 被参照数が増加していることがわかる. また, 少数のクラスが集中的に参照されていた状態から, 多くのクラスが参照されるように変化した. これは, 従来, STM 部で行なっていた処理を, Visitor パターンの形式で MET 部に移動した結果である. このことは Visitor パターンが必ずしも良い効果ばかりをもたらすのではなく, その適用には充分な検討が必要であることを物語っていると考えられる.

5 まとめ

従来のメトリクスに加えて, 以下のメトリクスが必要であることを提案した.

- C++の非オブジェクト指向要素の比率
- フレームワークの評価のためのモジュール間の参照数

実用規模のプログラムについて, それらのメトリクスの測定実験を行なった. その結果, 非オブジェクト指向要素の割合は開発者のオブジェクト指向開発の習熟度の目安として, モジュール間の参照数はフレー

メトリクス	STM(B0)	STM(B)
メンバ数	9.6	8.9
同最大	20	30
メンバ関数の数	8.1	7.5
同最大	16	28
文の数の和	40.5	28.8
同最大	129	397
関数の文の数	5.0	3.8
子供の数	0.71	0.68
同最大	14	14
継承の深さ	1.96	1.79
同最大	3	3
RFC	29.5	25.5
同最大	89	359
参照クラス数	3.3	4.1
同最大	6	18
モジュール外参照	2.0	1.8
同最大	4	4
外部参照比率	61%	44%

表 5: STM のメトリクスの変化

ムワークの改善の目安として, それぞれ利用できることを示した.

また, 従来のメトリクスについても実測定の結果, いくつかの知見が得られた. さらに, 計測実験の過程で, メトリクスについては, 実感として以下のようなこともわかった.

- メトリクスの平均値はあまり意味がない.
  通常, 測定できるクラス数は多くないので, 分布は偏る傾向がある. 単純に平均のみ見ていると, 結論を誤る恐れが高い.
- メトリクスの最も有効な利用方法は異常値を検出することである.

今後は, 今回測定できなかった, クラステンプレートに関するメトリクスやモジュール間の継承情報も合わせて, より総合的に C++プログラムを評価する手段を検討していきたい.

また, 我々は, メトリクス測定ツールを C++のコーディングレビュー作業をガイドするシステム [9] の一部として位置付け, メトリクスの有効な利用形態を検討していきたいと考えている.

6 謝辞

本研究に, 種々の御協力を頂いた関係者の皆様に感謝致します.

参考文献

[1] 本位田, 青山, 深沢, 中谷 (編). オブジェクト指向分析・設計 開発現場に見る実践の秘訣. 共立出版, 1995.

[2] B. Henderson-Sellers. *Object-Oriented Metrics.* Prentice Hall, 1996.

[3] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE trans on SE*, Vol. 20, No. 6, pp. 476–493, 1994.

[4] M. Lorentz and J. Kidd. *Object-Oriented Software Metrics.* Prentice Hall, 1994. 羽生田 訳 オブジェクト指向ソフトウェアメトリクス.

[5] V. R. Basili, L. C. Briand, and W. L. Melo. A validation of object oriented design metrics as quality indicators. *IEEE trans on SE*, Vol. 22, No. 10, pp. 751–761, 1996.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[7] 金, 椿, 楠本, 菊野. C++プログラムに対する複雑さメトリクスの提案と大学環境での実験的評価. 信学論 D-I, Vol. J79-D-I, No. 10, pp. 729–737, 1996.

[8] 荒野, 中西, 藤崎. フレームワーク成熟化段階におけるクラス抽象化の評価メトリクスと評価方法論. 信学論 D-I, Vol. J79-D-I, No. 10, pp. 719–728, 1996.

[9] 堀田, 中島, 直田. C++レビュー支援システム. 情報処理学会第 56 回全国大会 発表予定, 1998.