オブジェクト指向メトリクスを用いた
開発支援法に関する研究

神谷 年洋

2001 年 2 月

内容梗概

本論文は，筆者が大阪大学大学院基礎工学研究科に在学中に行った，ソフトウェアの複雑度メトリクスを用いた開発支援に関する研究をまとめたものである．本研究は，近年のソフトウェア開発で用いられるオブジェクト指向技術やオブジェクト指向開発プロセスの特徴を考慮して，複雑度メトリクスを適用する手法を提案し評価することを目標としている．

近年，ソフトウェアの応用分野の拡大と共に，ソフトウェアが大規模・複雑化してきている．それに伴い，開発期間の短縮やコストの削減・品質の向上が求められている．これらの要求を実現するために数多くのソフトウェア開発支援に関する研究が行われてきている．複雑度メトリクスはソフトウェアの複雑さを評価する尺度であり，複雑度メトリクスよるプロダクトの品質評価や，複雑度メトリクスによるフォールト発生予測手法が，現在盛んに研究されている．オブジェクト指向プロダクトに特化した複雑度メトリクスも多数提案されているが，オブジェクト指向に特有の（あるいはオブジェクト指向とともに一般的となった）開発技術やプロセスが，複雑度メトリクスの計測や評価に与える影響については考慮されていなかった．

本研究では，再利用技術，設計，開発プロセスの 3 点に注目し，オブジェクト指向複雑度メトリクスの新しい適用手法を提案し，実験的な評価を行った．また，従来の（オブジェクト指向ではない）ソフトウェア向けの重複コード検出技術を，オブジェクト指向ソフトウェアに適用するための手法を提案し，実験的な評価を行った．

まず，再利用による影響を調べるため，著者は従来の複雑度メトリクスを修正し，再利用部分と新規開発部分を区別して評価する手法を開発した．この修正手法は，代表的なオブジェクト指向複雑度メトリクスである Chidamber と Kemerer のメトリクス（CK メトリクス）を始めとする多くのプロダクトメトリクスに適用可能である．これらのオブジェクト指向複雑度メトリクスは，プロダクトの構造のみから複雑さを計測するため，構造に表れないような質的な違いは無視される．提案する手法では，そのような質的な違いを考慮して複雑度メトリクスを修正する．実験により，修正されたメトリクスによってフォールト予測精度が改善されることを評価した．

次に，クラス階層に基づいて新規開発クラスの分類を行う手法を提案した．本手法は，オブジェクト指向開発において，多くの新規開発クラスがライブラリのクラスから導出(derivation)によって作られるという観察に基づき，再利用されたクラスによって新規開発クラスを分類する．手法を評価する実験においては，分類によってメトリクス計測値の分布が異なり，フォールト予測精度が向上するなど，統計的にも有効性が確認された．

次に，開発の早期段階でオブジェクト指向複雑度メトリクスを適用する手法を提案した．オブジェクト指向ソフトウェア開発プロセスとして代表的な OMT 法においては，漸増的に分

析/設計モデルが詳細化される．CK メトリクスなどは詳細な設計モデルを要求するため，そのままでは詳細設計が終了するまでは適用できない．本手法では，OMT 開発プロセスの各ステップで分析/設計モデルに付け加わる情報によって計測可能なメトリクス集合を定義し，各メトリクス集合によってフォールト発生予測を行う．実験によって，開発の早期における予測は，後期における予測と比較して，完全性では劣るが，ある程度の正確性を持つことが確認された．

最後に，ソースコードの品質を劣化させる要因である重複コードに関して，オブジェクト指向プログラミング言語の構造化されたスコープ規則／名前空間を考慮した重複コード検出法を提案し，実験による評価を行った．実験により，スコープ規則／名前空間を考慮することで検出可能になるような重複コードの存在が確認された．

以上，オブジェクト指向ソフトウェア開発において，大規模な再利用や，オブジェクト指向開発プロセスの特性を考慮した，複雑度メトリクスの適用方法を提案し，実験的な評価を行った．また，オブジェクト指向プログラミング言語向けの重複コード検出法を提案し，その効果を実験によって評価した．本研究で得られた知見は，オブジェクト指向ソフトウェア開発において，プロダクトの品質評価に貢献すると考えられる．

関連発表論文

(1) 学術論文誌

[1-1] 神谷 年洋, 別府 明, 楠本 真二, 井上 克郎, 毛利 幸雄: “オブジェクト指向プログラムを対象とした複雑度メトリクスの実験的評価,” 電気学会論文誌 C, Vol.117-C, No.11, pp.1586-1591 (1997).

[1-2] Toshihiro Kamiya, Shinji Kusumoto, Katsuro Inoue and Yukio Mohri: “Empirical evaluation of reuse sensitiveness of complexity metrics,” *Information and Software Technology*, Vol. 41, No. 5, pp.297-305 (1999).

[1-3] Motoyasu Takehara, Toshihiro Kamiya, Shinji Kusumoto and Katsuro Inoue: “Empirical Evaluation of Method Complexity for C++ Program,” *IEICE Trans. on Information and Systems*, Vol. E83-D, No. 8, pp.1698-1700 (2000).

[1-4] 神谷 年洋, 楠本 真二, 井上 克郎, 毛利 幸雄: “複雑度メトリクスを用いたエラー予測の一手法 －アプリケーションフレームワークを用いた開発への適用－,” 情報処理学会論文誌. (条件付採録)

[1-5] Toshihiro Kamiya, Shinji Kusumoto and Katsuro Inoue: “A Token-based Code Clone Detection Tool - CCFinder and Its Empirical Evaluation,” *IEEE Trans. on Software Eng.*(投稿中)

[1-6] 神谷 年洋, 楠本 真二, 井上 克郎: “OMT に基づく開発プロセスでのフォールト発生クラス予測,” 情報処理学会論文誌. (投稿中)

(2) 国際会議(査読あり)

[2-1] Toshihiro Kamiya, Shinji Kusumoto and Katsuro Inoue: “Prediction of Fault-Proneness at Early Phase in Object-Oriented Development,” *Proc. of The 2nd IEEE Int'l Sympo. on Object-Oriented Real-Time Distributed Computing (ISORC'99)*, pp.253-258, Saint Malo, (May, 1999).

[2-2] Toshihiro Kamiya, “On the Prediction of Fault-proneness in Object-Oriented Development”, *Proc. of Empirical Studies of Software Development and Evolution*, pp. 1-3. Los Angeles, (May, 1999).

[2-3] Toshihiro Kamiya, Fumiaki Ohata, Kazuhiro Kondou, Shinji Kusumoto, and Katuro Inoue: “Maintenance support tools for Java programs: CCFinder and JAAT”,

Proc. of The IEEE 23rd Int'l Conf. on Software Eng. (ICSE'2001), Toronto, Canada, (May, 2001). (to appear)

(3) 研究会(査読あり)

[3-1] 神谷 年洋, 別府 明, 楠本 真二, 井上 克郎, 毛利 幸雄: "オブジェクト指向プログラムを対象とした複雑度メトリクスの実験的評価 ～Chidamber らのメトリクスを対象として～," ソフトウェア工学の基礎 IV 日本ソフトウェア科学会 FOSE'97, pp.159-166 (1997-12).

[3-2] 神谷 年洋, 高林 修司, 楠本 真二, 井上 克郎: "C++プログラムを対象とした複雑度メトリクス計測ツールとその評価," オブジェクト指向最前線'98 情報処理学会オブジェクト指向'98 シンポジウム, pp.37-44 (1998-9).

[3-3] 神谷 年洋: "ソフトウェアを取りまく環境の変化がメトリクスに及ぼす影響について," 情報処理学会シンポジウムシリーズ ウィンターワークショップ・イン・高知論文集, pp.55-56 (1999-1).

[3-4] 神谷 年洋, 楠本 真二, 井上 克郎: "オブジェクト指向開発におけるフォールト発生早期予測手法の一提案," 情報処理学会オブジェクト指向'99 シンポジウム論文集, pp.145-152 (1999-7).

[3-5] 神谷 年洋, 楠本 真二, 井上 克郎: "コードクローン検出における新手法の提案および評価実験," 電子情報通信学会 情報・システムソサイエティ ソフトウェアサイエンス 2001 年 1 月研究会 (2001-1). (採録予定)

(4) その他

[4-1] Toshihiro Kamiya, Takuya Uemura, Fumiaki Ohata, Shinji Kusumoto, and Katsuro Inoue: "SMART --- A Complexity Metrics Tool for C++," Poster Session of The 21st Int'l Conf. on Software Eng. (ICSE'99), Los Angeles, U.S.A., (May, 1999).

[4-2] 神谷 年洋, 楠本真二, 井上 克郎, 毛利 幸雄: "C++プログラムを対象とした複雑度メトリクス計測ツールと新人卒業演習への適用", ユニシス技法, Vol. 19. No. 2, pp. 138-151 (1999).

[4-3] 神谷 年洋, 楠本 真二, 井上 克郎, 毛利 幸雄: "オブジェクト指向に基づくソフトウェア開発プロセスの分析 ～教育環境における評価実験～," 電子情報通信学会 情報・システムソサイエティ 大会講演論文集, p. 41 (1996-9).

[4-4] 神谷 年洋: "コードクローンの検出手法," 情報処理学会 ソフトウェア工学研究会

ウィンターワークショップ・イン・金沢 (2001-1). (採録予定)

目次

図目次

表目次

1. 緒論

近年，ソフトウェアの応用分野の拡大と共に，ソフトウェアが大規模・複雑化してきている．それに伴い，開発期間の短縮やコストの削減・品質の向上が求められている．これらの要求を実現するために数多くのソフトウェア開発支援に関する研究が行われてきている．開発支援のアプローチの１つはソフトウェア開発における各作業の効率化である．開発作業の効率化を目指してこれまでに多くのソフトウェア開発手法や CASE(computer aided software engineering)ツールが開発されてきた．最近では，オブジェクト指向パラダイムが注目され，それに基づいた分析法，設計法，プログラミング言語等が数多く提案されており，実際の開発現場でも使われている．オブジェクト指向技術により，ソフトウェア部品の再利用が効率よく行え，結果として生産性や品質の向上が実現されている．

品質改善のためのもうひとつのアプローチとして，「プロセス改善」，すなわち，開発プロセスの品質を改善することで生産性やプロダクトの品質を向上させるという手法がある．プロセス改善の枠組みとしては，CMM(Capability Maturity Model)[41]や ISO9000[23]が良く知られている．また，「ソフトウェア計測」，すなわち，プロダクトを定量的または定性的に評価する手法も，品質改善に用いられている．定量的にソフトウェアの品質を評価するために，ソフトウェアを定量的に計測するための尺度であるソフトウェアメトリクスが用いられる．定性的にプロダクトの品質を評価する手段としては，レビューにおけるチェックリストや，重複コードの検出がある．

オブジェクト指向技術の発展や普及により，あるいはそれと時期を同じくして，フレームワークなどの大規模な再利用技術や，オブジェクト指向開発プロセスが用いられるようになった．このような状況のもとで，オブジェクト指向複雑度メトリクス(Chidamber と Kemerer のメトリクス[14]，Briand のメトリクス[9]，Li のメトリクス[32]など)は，再利用されるソースコードの品質や，再利用部分が新規開発部分の設計に与える影響を十分に反映しているとはいえない．また，従来の重複コード検出技術においては，オブジェクト指向プログラミング言語を前提とした検出を行っていなかった．そこで，本研究では以下の 4 点について研究を行った．

(a)再利用技術

ソフトウェアの再利用は，開発コストの削減，開発期間の短縮，およびプロダクト品質の向上に効果的であることが知られている[26]．また，オブジェクト指向ソフトウェア開発においては，特定のドメインに特化したライブラリであるフレームワークを用いて，ソフトウェアの大部分を再利用部品によって開発する手法が用いられる．このような再利用部品には十分にテストされた高品質なものも存在する(IBM の zero-defect components など)．しかしながら，CK メトリクスなどの従来の複雑度メトリクスは，個々のソフトウェア部

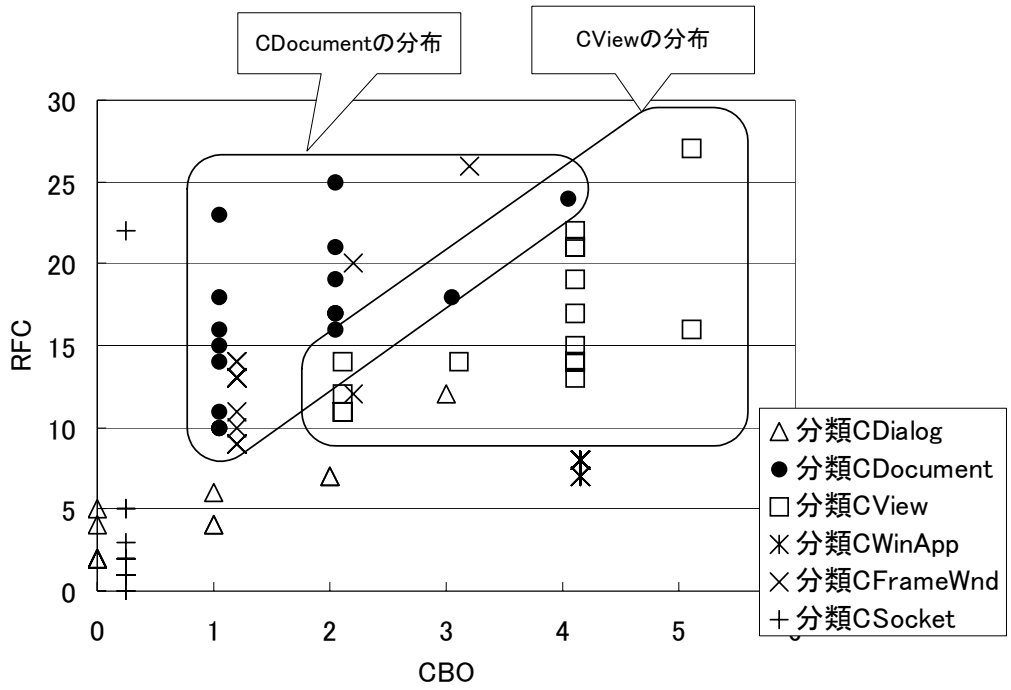


図 1.1 124のクラスの CBO と RFC

品がどの程度テストされているかなどの要因は考慮せず，ソフトウェア部品の構造だけから複雑度を算出する．そのため，再利用部分と新規開発部分の品質の差はメトリクスの計測値に反映されず，その結果メトリクスによる予測モデルが不十分なものになる可能性がある．本論文では，再利用部分と新規開発部分の品質の差を考慮したメトリクスの適用法を議論する．

(b)設計

CK メトリクスなどいくつかのオブジェクト指向複雑度メトリクスは，クラスを計測対象として，その設計の複雑度を計測する．個々のクラスの設計は，そのクラスが果たす機能によって大きく異なる．機能によってクラスを分類した場合，クラスの種類によりメトリクス計測値の分布が異なることが指摘されている[38]．一例として，図 1.1 は，後述する実験（4.5 参照）で開発されたクラスについて，2 つのメトリクス CBO と RFC の計測値をプロットしたものである．たとえば分類 CDocument と CView では，計測されたメトリクス値の分布が分離していることがわかる．このような統計的な偏りはメトリクスによる予測の精度を悪化させると考えられる．本論文では，オブジェクト指向開発において，クラス分類により予測精度を改善する方法を議論する．

(c)開発プロセス

メトリクスによる予測を開発プロセスの計画に用いる(典型的には，ファンクションポイント[1]から算出したソフトウェアの開発期間やコスト見積りを，ソフトウェア開発契約に用いる)場合には，開発プロセスのなるべく早期に，より正確に予測が行えることが望ましい．エラーは早期に発見するほうがその修正に必要なコストが低いことを考えると，複雑度メトリクスによるエラーの予測も，なるべく早期に行うべきである．OMT などのオブジェクト指向開発プロセスにおいては，開発の初期から一貫してオブジェクトを用いて分析/設計を行う．プロセスの進行に従って，オブジェクトあるいはクラスの関係や属性が漸増的に詳細化されるため，CK メトリクスなどの複雑度メトリクスは通常，設計の後期，コーディングの直前に適用されている．本論文では，オブジェクト指向開発プロセスの特性を考慮して，複雑度メトリクスを開発のより早期，設計フェーズの早期に適用する方法について議論する．

(d)コードの重複

重複したコードは，カット＆ペーストによるソースコードの再利用などにより作られる．重複したコードによってメトリクスの計測が不正確になる恐れがあり，また，重複したコードの存在によってプロダクトの保守が困難になることが指摘されている．本論文では，オブジェクト指向プログラミング言語で記述されたソースコードから重複コードを検出する手法について議論する．

以下，2 では本論文の基礎となる開発技術や手法の説明を行う．3 では，オブジェクト指向ソフトウェア向けの複雑度メトリクスのひとつである CK メトリクスを，再利用部分と新規開発部分の品質の差を考慮して修正する手法を提案する．CK メトリクスと修正された CK メトリクスを，フォールトとの相関によって比較・評価する．4 では，フレームワークの再利用に着目して，クラスを分類し，メトリクスによる予測の精度を高める方法を提案する．実験により，フォールト予測精度を評価する．5 では，オブジェクト指向開発プロセスの特性を考慮して，複雑度メトリクスを開発プロセスの早期に適用する方法を提案し，フォールト予測精度を実験的に評価する．6 では，オブジェクト指向プログラミング言語で取り入れられたクラススコープや名前空間を考慮した，重複コード検出手法を提案し，実験によってその効果を評価する．7 で結論と今後の展望を述べる．

2. 諸準備

2.1. オブジェクト指向開発技術

現在，オブジェクト指向およびオブジェクト指向を基礎とした，さまざまな開発手法・技術が存在する．ソフトウェア開発に関しては，オブジェクトを直接表現できる記述言語として，モデリング言語(要求仕様書および設計書を記述するための言語)UML(Unified Modeling Language)[54]が提案されている．また，JavaやC++を筆頭する数多くのオブジェクト指向プログラミング言語が用いられている．これらの言語は，オブジェクトの型を表現する「クラス」という概念を持ち，さまざまなツール(エディタや処理系，リバースエンジニアリングツール)によってサポートされている．**アプリケーションフレームワーク**と呼ばれるライブラリは，特定ドメインのアプリケーションの「半完成品」であり，必要な部分を補うことでアプリケーションを作成することができる．現在，複数のアプリケーションフレームワークが，オブジェクト指向プログラミング言語向けに実用化されている．デザインパターン[21]は，実際の設計に繰り返し現れる解法を，オブジェクト指向で構造化したカタログであり，オブジェクト指向ソフトウェアの設計を行う際に用いられる．デザインパターンはまた，いくつかのアプリケーションフレームワークの設計にも取り入れられている[46]．

2.2. オブジェクト指向と開発プロセス

オブジェクト指向技術に特化した開発プロセス(以下，**オブジェクト指向開発プロセス**)としては，Booch 法[7]や Jacobson の OOSE[24]，Rambaugh らの OMT 法[42]，Shlaer-Mellor 法[43]など，数多く提案されている．OMT 法においては，開発プロセス全体を通じて，オブジェクトを用いてシステムの構造を表現する．システムの詳細化が進むにつれ，扱うオブジェクトの粒度も小さくなっていく．最初はユーザー/システムといった抽象度の高いオブジェクトを用い，最後はプログラミング言語のクラスといった抽象度の低いオブジェクトを扱う．後に，Booch, Jacobson, Rambaughの3者は彼らの手法を統一した，Unified Development Process とよばれる開発プロセスを提案した[25]．また，近年では，オブジェクト指向開発技術に特化したテスト方法も提案されてきている[6]．

2.3. ソフトウェアの品質改善技術

ソフトウェアの品質には，機能性，信頼性，ユーザビリティ，効率，保守容易性，移植性などさまざまな側面がある[45]が，本論文では，信頼性および保守性に限定して議論する．

ソフトウェアの品質を改善するために，さまざまな(検査，検証)の技術が開発されている．フォーマルアプローチ[18]は，ソフトウェアの仕様を形式的に表現し，代数的な検証を可能

にする技術である. レビューはプロダクト(仕様書・設計書・ソースコードなど)を複数の関係者の間で調査し, フォールトを発見するための技術である. テストはソフトウェアが要求仕様どおりに正しく機能するか否かを検証し, 欠陥を検出する技法である. CMM(Capablity Maturity Model)[41]は, ソフトウェア開発組織の開発能力がどの程度成熟しているかを5段階で評価し, 具体的な改善点を示す手法である.

メトリクスは, 開発プロセスやプロダクトのさまざまな特性を計測する尺度である. ソフトウェアの行数, 開発工数の人月など, 比較的直感的なメトリクスがある一方で, ファンクションポイントなどの複雑な計算を必要とするメトリクスも存在する. メトリクスはプロセスの評価[44][45], 統計的モデルに基づく予測にも用いられる. たとえば, ソフトウェア信頼度成長モデルでは, 「ソフトウェアに含まれるフォールトの総数は有限であり, テストによって順次発見される」というモデルに基づき, テスト時間とそれまでに発見されたフォールト数から残存するフォールトを予測する[50]. プロダクトメトリクスによる評価をレビューに取り込んだ開発プロセスも提案されている[47].

ソースコードの重複とは, 「カット&ペースト」プログラミングや意図的な繰り返しなどにより生じた, ソースコード中の同一あるいは類似した部分である. このような, 生成されたソースコード中の同一あるいは類似したコードの断片のことを, **コードクローン**と呼ぶ. コードクローンの存在はソースコードの一貫した修正を困難にするため, XP(extreme programming)[55]などの開発プロセスにおいては, リファクタリングにおいてコードクローンを取り除くことが奨励されている. クローンの存在は, メトリクスの計測にも影響を及ぼす. たとえば, 保守プロセスにおいて, 機能の追加とともにコードクローンの除去を行った時に, 追加されたコードよりも除去されたコードのほうが多ければ, 機能が増えてコードが小さくなるという, 一見矛盾した事態が生じる.

2.4. プロダクトメトリクスによるソフトウェア計測および予測

プロダクトを計測対象とするメトリクス(以下, **プロダクトメトリクス**)には, SLOC(ソースコードの行数)のような**規模メトリクス**と, McCabe のサイクロマチック数[37]のような**複雑度メトリクス**がある. 規模メトリクスのひとつであるファンクションポイント[1]は, 要求仕様書を計測対象とし, ソフトウェアの機能量を数値化する. ファンクションポイントの計測値は, ソフトウェアの規模(行数)や開発工数の予測に用いられる[34]. オブジェクト指向ソフトウェアを対象とした複雑度メトリクスとして, Chidamber と Kemerer のメトリクス(以下, **CK メトリクス**)[14], Brianらのメトリクス[9], Liのメトリクス[32], などがある. これらのメトリクスはソフトウェアの静的な構造の複雑さを評価するものである. 近年, Yacoub らが提案したメトリクス[49]は, プロ

グラムの動的な複雑さ(特定の実行における振る舞いの複雑さ)を計測する.

プロダクトメトリクスには, 規模メトリクスと複雑度メトリクスという分類以外にも, 計測対象(ソースコード, 設計書, 仕様書)による分類がある. さらに, 特定のプロダクトから特定のメトリクスを計測する際には, 計測コスト(たとえば, あるメトリクスを計測する際に, 人間による評価が不可欠であれば, 人件費が計測コストとなる)や, 予測精度など, さまざまな要因を考慮する必要がある.

プロダクトメトリクスを用いて, 規模や工数, エラーを予測する際には, 統計的分析が用いられる. その手順は, 一般的には以下の 3 つのステップを順に行う.

(1)基準となるデータの収集

統計モデルの基礎となるデータを収集する. メトリクスの計測対象となるプロダクト(要求仕様書, 設計仕様書, ソースコードなど)から, メトリクス値を計測する. 予測したい変数(規模, 工数, フォールトの有無, フォールト数, フォールト修正労力など)の実測値も収集する.

(2)予測モデルの作成

メトリクスの計測値から統計分析によって, クラスにエラーが含まれるかどうか, あるいは, 含まれる数, エラー修正労力などを予測する統計的な予測モデルを作る.

(3)予測モデルの適用

予測モデルを実際のプロダクトに適用して, 予測を行う.

メトリクスに関する研究の現状として,「特定の予測ために必要十分なメトリクスの集合」といった標準はいまだに確立されておらず, 重要な新規メトリクスが提案・評価されているところである. したがって, 長期間にわたるメトリクスのデータを蓄積して, 後の評価や予測に役立てようとする向きには, メトリクスの計測値だけをデータとして残しておくのではなく, メトリクスを計測可能なプロダクトも併せて残しておくほうが現実的であろう.

統計的な予測モデルの選択に関しては, 予測されるもの(従属変数)が真偽値であるか, 分類であるか, 連続値であるかによって, ロジスティック回帰分析, 決定木, 線形回帰分析など, さまざまな統計分析が用いられる.

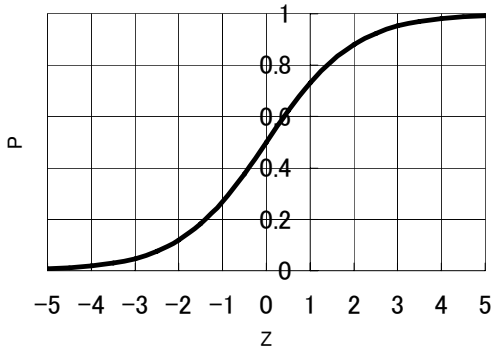


図 2.1 ロジスティック曲線

2.4.1. 多変量ロジスティック回帰分析

予測モデル作成の具体例として，多変量ロジスティック回帰分析について説明する．多変量ロジスティック回帰分析は，複数の複雑度メトリクスを用いて，プロダクトにフォールトが含まれるかどうかを予測する際に，文献[4][9][10]を始めとする多くの論文で用いられている．多変量ロジスティック回帰分析で用いられる予測式は，プロダクトのメトリクス計測値を入力とし，プロダクトにフォールトが含まれるかどうか（真偽値）を出力する関数である．関数の一般形を以下に示す．

$$P(X_1,\Lambda\ ,X_n)=\frac{1}{1+\exp(-(C_o+C_1\cdot X_1+\Lambda\ +C_n\cdot X_n))}$$

ここで，Pは計測対象のプロダクトにフォールトが含まれる確率であり，X_iはプロダクトの各メトリクスの計測値である．後述する手続きによって，係数 C_0, C_1, ..., C_n を決定する．係数が決定された後，「もし与えられたメトリクス計測値が Pを 0.5 以上にするなら，プロダクトはフォールトを持つ」と予測する．この式において，$Z = C_0 + C_1 \cdot X_1 + \ldots + C_n \cdot X_n$とおけば，$P(Z) = 1 / (1 + \exp(Z))$となる．この Pと Zの関係は S 字カーブ(図 2.1 参照)になる．このような S 字カーブは X_iと Pの関係が単調であれば，2 値の分類に適用可能である．

各係数 C_iは，収集された基礎データから，最尤度(maximum-likelihood)基準によって，すなわち，観測された結果をもっともよく反映するように，その値を決定する．ただし，メトリクス(変数)X_1, ..., X_nの相関が強い(独立性が低い)場合，冗長な変数が含まれると，導かれる係数を不適切あるいは誤解を招くようなものにしてしまう．このような，意味がないあるいは有害な変数は，段階的に変数を選択することによって取り除く[35]．

2.4.2. 複雑度メトリクスの例：CK メトリクス

オブジェクト指向ソフトウェアに対する複雑度メトリクスとしては，Chidamber と Kemerer が提案した 6 種のメトリクス(CK メトリクス)がもっとも著名である[14]．CK メトリクスは，クラスの構造に基づいて，その複雑度を静的に評価する．CK メトリクスは，Weyuker が提案した複雑度メトリクスが満たすべき数学的性質[48]をおおむね満たすことが確認されている．CK メトリクスはまた，複数の実験によって，エラーの発生を予測する精度が評価されている[4][10][13]．CK メトリクスの変種も多数提案されており，文献[12]では，CK メトリクスおよびその変種を含む多くの複雑度メトリクスを同一のデータによって比較している．特定のプログラミング言語で記述されたソースプログラムから CK メトリクスを抽出するツールも開発されている[52]．以下に，文献[4]からの引用した，CK メトリクスの定義を示す．

WMC(クラスの重み付きメソッド数;Weighted methods per class):

計測対象クラス C_1 が，メソッド $M_1, \ldots, M_n$ を持つとする．これらのメソッドの複雑さをそれぞれ $c_1, \ldots, c_n$ とする．このとき，$WMC = \Sigma c_i$ である．適切な間隔尺度 f を選択して $c_i = f(M_i)$ によりメソッドを重み付けする．すべてのメソッドが同じ複雑度であると仮定した場合，WMC はメソッドの数となる(以下では，特に断らない限り，WMC はメソッドの数とする)．

DIT(継承木における深さ;Depth of inheritance tree):

DIT は計測対象クラスの継承の深さである．多重継承が許される場合は，DIT は継承木におけるそのクラス(を表す節点)からそれ以上基底クラスが存在しないクラス(根)に至る最長パスの長さとなる．

NOC(子クラスの数;Number of children):

NOC は計測対象クラスから直接導出されているサブクラスの数である．

CBO(クラス間の結合;Coupling between object classes):

CBO は，計測対象クラスが結合しているクラスの数である．あるクラスが他のクラスのメソッドやインスタンス変数を参照しているとき，結合しているという．

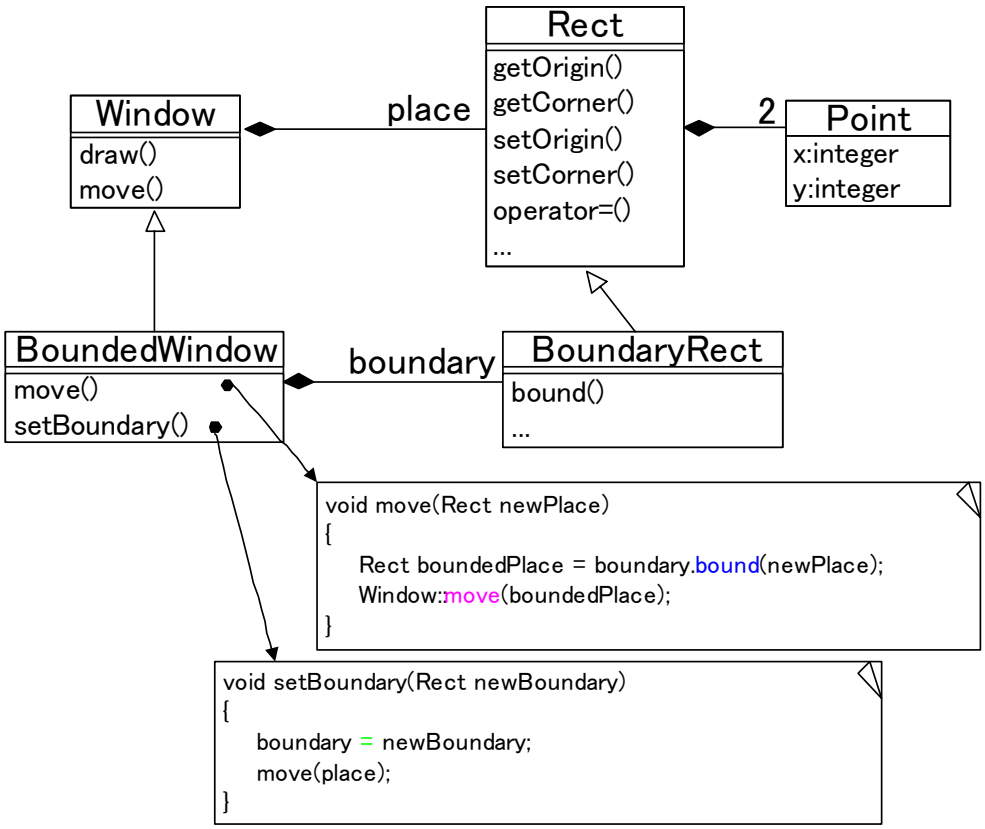


図 2.2 CBO と RFC の計測方法を説明するための例

RFC(クラスの反応;Response for a class):

計測対象のクラスのメソッドと，それらのメソッドから呼び出されるメソッドの数の和として定義される(すなわち，メッセージに反応して潜在的に実行されるメッセージの数となる).

LCOM(メソッドの凝集の欠如;Lack of cohesion in methods):

計測対象クラス C_i が n 個のメソッド $M_1, \ldots, M_n$ を持つとする. I_i ($i = 1, \ldots, n$)を，それぞれメソッド M_i によって用いられるインスタンス変数の集合とする. $P = \{(I_i, I_j) \mid I_i \cap I_j = \phi\}$ と定義し，$Q = \{(I_i, I_j) \mid I_i \cap I_j \neq \phi\}$ と定義する. もし $I_1, \ldots, I_n$ がすべて ϕ の時は，$P = \phi$ とする. このとき，$LCOM = |P| - |Q|$，ただし，値が 0 より小さくなるときは 0，と定義する.

いずれのメトリクスも，計測値は 0 以上になり，計測対象のクラスが複雑になるほど，その計測値が大きくなる. 図 2.2 は CBO と RFC の計測方法を示すための例である. 図中のクラス `BoundedWindow` は，クラス `Window` から導出され，インスタンス変数 `boundary`(型はクラス `BoudaryRect`)，メソッド `move()` と `setBoudary()` を持っている. メソッド `move()` の定義中で，クラス `BoundaryRect` のメソッド `bound()` と，クラス `Window` のメソッド

move()を呼び出している. メソッド setBoudary()の定義中で, クラス Rect のメソッド operator=()を呼び出している. BoundedWindow は 2 つのメソッドを持っており, 他のクラスの 3 つのメソッドを参照しているので, RFC は 5 となる. BoundedWindow は, クラス BoundaryRect, Rect, Window の 3 つのクラスを(いずれもメソッド呼び出しによって)参照しているので CBO は 3 となる.

3. 再利用を考慮した構造メトリクス計測法

3.1. 緒言

近年，ソフトウェアが大規模・複雑化してきたことに伴い，開発期間の短縮やコストの削減・品質向上の要求が高まっている．そのような要求に応えるためには，ソフトウェアの全ライフサイクル(ソフトウェアの開発・保守)にわたる管理が必要である．「ソフトウェア開発プロセスの品質」という概念は，ソフトウェアを開発するプロセスを改善し，そのプロセスで生産されるソフトウェアの品質を安定させ管理可能にするために生まれたものである[28]．

開発プロセスの品質改善のひとつの方法は，開発プロセスの各フェーズで開発されるプロダクトの状態を測定し，分析して，プロセスにフィードバックすることである．ソフトウェアメトリクスは，ソフトウェアのさまざまな特性(複雑度，信頼性，効率等)を判別する客観的な数学的尺度である．そのなかでも，ソフトウェアの複雑度メトリクスは，ソフトウェアの品質や保守の容易さを評価／予測するために用いられる．測定の結果，ソフトウェアが複雑であればあるほど，エラーが含まれている可能性が高く(品質が低く)，保守が困難であると評価される．

これまでに提案された代表的なソフトウェア複雑度メトリクスには，Halstead のメトリクス[22]，McCabe のサイクロマチック数[37]などがある．Chidamber と Kemerer は，これらのメトリクスは従来の(非オブジェクト指向の)プログラミング言語で開発されたソフトウェアに対する複雑度メトリクスであり，オブジェクト指向設計を用いて開発されたソフトウェアの複雑度を評価するには不十分であると指摘し，オブジェクト指向設計で開発されたソフトウェアを対象とする 6 つの複雑度メトリクスを提案した[14]．

一方，オブジェクト指向ソフトウェア開発では，独立性が高く，組み合わせの容易な部品を利用してソフトウェアを開発することが，効率の良い開発の鍵であるとされている[26]．すでに存在する高品質のソフトウェアを再利用することで品質の向上を実現し，また，再利用によって開発するソフトウェアの分量を減少させることで開発期間の短縮を目指している．しかし，CK メトリクスは，そのような，積極的な再利用を用いて作成されたソフトウェアに対しては，有効性の評価が十分に行われていない．本章では，積極的な再利用を用いて作成されたソフトウェア中のクラスに，複雑度メトリクスを適用する際の問題について議論し，再利用による影響を反映するようにメトリクスを修正する．修正されたメトリクスを計測の理論に基づいて評価し，次に，実験により統計的な評価を行う．

3.2. 再利用によってメトリクスが受ける影響

従来の研究では，CK メトリクスを始めとする複雑度メトリクスを，再利用されるクラスと，新

規に開発されるクラスに，同じように適用している．しかし，複雑度の評価においては，再利用クラスと新規開発クラスを区別して扱うべき理由が存在する．まず，再利用される部品は通常，新規開発の部品よりも品質が高く，含まれるフォールトも少ない[20][26]．再利用されるクラスと新規開発されるクラスとでは，複雑さを押し上げる要因が異なっていると考えられる(表 3.1)．

3.3. 再利用を考慮した修正 CK メトリクス

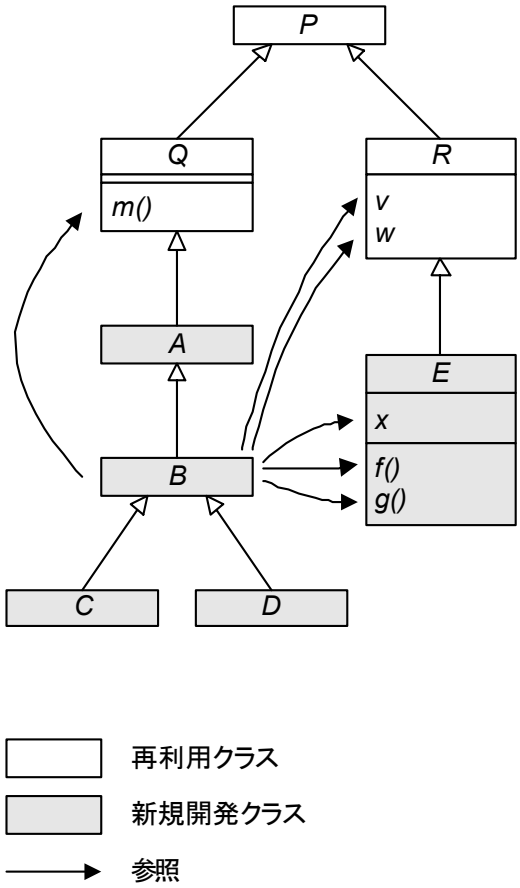
本研究では，3.2 の議論に基づいて，「新規開発部品と再利用部品はプログラムの複雑さに異なった影響を与える」という仮説をおく．CK メトリクス(2.4.2 参照)のうち，DIT, NOC, CBO, RFC はクラスの外部複雑度，すなわち，計測対象クラスとそれ以外のクラス間の関係の複雑さを計測する．DIT と NOC は導出の複雑さを計測する．CBO と RFC は他のクラスへの結合，参照の複雑さを計測する．仮説に基づいて，これら 4 つのメトリクスを以下のように修正する．

DITN, DITR(Depth of inheritance tree)：

DITN(C)は，クラス階層木における，クラス C から根にいたるパスの中に現れる，新規

表 3.1 新規開発部品と再利用部品が与える影響

新規開発部品	再利用部品
• 新規開発部品はテストフェーズを経るまで未テスト状態であり，再利用部品よりエラーが含まれることが多い． • トップダウンに設計されるため，仕様はシステムに適合している． • 開発中に，仕様の変更，エラー修正などによって変更されやすい． • 文書化が不足しがち(あるいはあっても不十分になりがち)であり，開発者が誤解をしている可能性がある．	• 少なくとも一度テストを経てきている． • 部品が開発されるシステムに適合していない，あるいは，一般性を持たせるために過度に複雑なインターフェイスを持つことがある． • 部品の供給者の意向により，再利用する開発者が修正を行うことができないかもしれない．このような場合には，部品に含まれるフォールトが深刻な影響を及ぼすことがある． • 開発者が再利用部品に関する知識をもたない場合，学習に時間を割く必要がある．



計測値	関係
DIT(B) = 3	{ A, P, Q }
DITN(B) = 1	{ A }
DITR(B) = 2	{ P, Q }
NOC(B) = 2	{ C, D }
NOCN(B) = 2	{ C, D }
NOCR(B) = 0	ϕ
CBO(B) = 3	{ E, Q, R }
CBON(B) = 1	{ E }
CBOR(B) = 2	{ Q, R }
RFC(B) = 3	{ f of E, g of E, m of Q }
RFCN(B) = 2	{ f of E, g of E }
RFCR(B) = 1	{ m of Q }

図 3.1 再利用クラスと導出によって作られた新規開発クラス

開発クラスの数である. DITR(*C*)は同パス中に現れる, 再利用クラスの数である. 定義より, DITN(*C*) + DITR(*C*) = DIT(*C*).

NOCN, NOCR(Number of children):

NOCN(*C*)はクラス *C*から直接導出されている新規開発クラスの数. NOCR(*C*)はクラス *C* から直接導出されている再利用クラスの数. 定義より, NOCN(*C*) + NOCR(*C*) = NOC(*C*). NOCR は新規開発クラスに対しては常に 0 となる(新規開発クラスから再利用クラスが派生することはないため).

CBON, CBOR(Coupling between object-class):

CBON(*C*)は, クラス *C*が結合している新規開発クラスの数である. CBOR(*C*)は, クラス C が結合している再利用クラスの数. 定義より, CBON(*C*) + CBOR(*C*) = CBO(*C*)となる.

RFCN, RFCR(Response for a class):

Ms(C)をクラス C のすべてのメソッドの集合， Mr(C) = { M_j | M_j は，$M_i \in$ Ms(C) に呼び出されるメソッド }とする．さらに，M_N は新規開発クラスに属するメソッドの集合，M_R は再利用クラスに属するメソッドの集合とする．このとき，RFCN(C) = | (Ms(C) ∪ Mr(C)) ∩ M_N |，また，RFCR(C) = | (Ms(C) ∪ Mr(C)) ∩ M_R |となる．定義より，RFCN(C) + RFCR(C) = RFC(C)となる．

CK メトリクスの残る 2 つ，WMC と LCOM はクラスの内部複雑度を計測する．WMC はメソッドの複雑度を計測し，LCOM はメソッドの凝集度を計測する．これら 2 つのメトリクスはクラス間の関係を計測するものではないため，その修正版は定義されない．

図 3.1 は修正されたメトリクスの計測法を示すためのクラス階層の例である．左側に，UML で記述されたクラス階層，右側に，計測されるメトリクス値が示されている．クラス A から E は新規開発クラス，P, Q, R は再利用クラスである．クラス B は A から導出されていて，2 つの子クラス，C と D を持つ．B のメソッド（図示されていない）は Q のメソッド m と E のメソッド f および g を呼び出す．B のメソッドはまた，R のインスタンス変数，v と w を参照する．それぞれのメトリクスによって数え上げられた関係は，括弧の中に示されている．たとえば，クラス B の CBO は 3，すなわち，B はクラス E, Q, R と結合している．

Briand らは文献[11]において，結合を数えるようなメトリクスには一般に「不安定なクラスへの結合だけを数えるオプションがある」と記している．ただし，本研究においては，Briand らの指摘しているような新規開発クラスへの結合だけを数える方法だけではなく，再利用クラスへの結合だけを数える方法も提示している．再利用クラスへの結合だけを数えるメトリクス自体の有効性は，後の 4, 5 における実験において示される．

3.4. 修正されたメトリクスの Weyuker の性質による評価

CK メトリクスは，Weyuker が提案した複雑度メトリクスが満たすべき数学的性質[48]をほぼ満たすことが，Chidamber と Kemerer によって確認されている．ここでは，修正メトリクスがこれらの性質を満たしていることを，数学的に検証する．

以下に示す Weyuker の性質は，Chidamber と Kemerer によってオブジェクト指向複雑度メトリクス向けに修正されたものである[14]．

ここで，$\mu(c)$ はクラス c に対するメトリクス μ の計測値を表し，$p + q$ はクラス p と q を合成してできたクラスを表わすとして，

W1 $\exists p \exists q, \mu(p) \neq \mu(q)$.

W2 $\exists\, p \exists\, q,\ \mu(p) = \mu(q)$, ただし, p と q は異なる.

W3 $\exists\, p \exists\, q,\ \mu(p) \neq \mu(q)$, ただし, p と q は同機能であり, 設計は異なる.

W4 $\forall\, p \forall\, q,\ \mu(p) \leq \mu(p + q)$,かつ$\mu(q) \leq \mu(p + q)$.

W5 $\exists\, p \exists\, q \exists$ r, $\mu(p) = \mu(q)$,かつ$\mu(p + r) \neq \mu(q + r)$.

¬W6[*] $\forall\, p \forall\, q,\ \mu(p) + \mu(q) \geq \mu(p + q)$.

ChidamberとKemererは, メトリクスDIT, NOC, LCOMがW4を満たさないという例外を除いて, WMC, DIT, NOC, CBO, RFC, LCOMのそれぞれが, 性質W1, ..., ¬W6を満たすことを証明した.

8種の修正されたメトリクス（DITN, DITR, NOCN, NOCR, CBON, CBOR, RFCN, RFCR）が, DITNとDITRがW4を満たさないという例外を除いて, それぞれ性質W1, ..., ¬W6を満たすことを示す. まず, W1, W2, W3, W5に対する評価を行い, 次に, W4と¬W6に対する評価を行う.

W1, W2, W3, W5 に対する評価

性質W1, W2, W3, W5に関しては(存在限量子つきの命題であるため), それぞれを満たす例をあげる. Chidamberと Kemererの証明に基づいて, DITが W1を満たすような2つのプログラム P_1 と P_2 が存在する. 一般性を失うことなく, P_1 のすべてのクラスが新規開発クラスであったと仮定する. すると, P_1 においてDITNはDITに等しくなり, それゆえDITNもW1を満たす. 次に, 一般性を失うことなく, P_2 のすべてのクラスが再利用クラスであると仮定する. P_2において, DITRはDITに等しくなり, それゆえDITRもW1を満たす. これを直積{DIT, NOC, CBO, RFC} × {W1, W2, W3, W5}のそれぞれの要素について繰り返すことにより, 8つのメトリクスDITN, DITR, NOCN, NOCR, CBON, CBOR, RFCN, RFCRはW1, W2, W3, W5を満たすことが証明される.

W4 と¬W6 に対する評価

修正されたメトリクスを W4 と¬W6 に対して評価するにあたって, メトリクスμに対して形式的な定義を導入する.

---

[*] ¬W6はWeyukerが提案した性質 W6 $\exists\, p \exists\, q,\ \mu(p) + \mu(q) < \mu(p + q)$の否定になっている. ChidamberとKemererは6種のメトリクスが¬W6を満たすことを証明した.

定義:

あるメトリクスμ が有限集合 X_μ と関係 R_μによって以下のように定義されるとする.

(C1) 任意のクラス c に対して, $M(c) = \{ x \mid x \in X_\mu$,かつ$(c\ R_\mu\ x) \}$とし, $\mu(c) = \mid M(c) \mid$と定義する.

(C2)任意のクラス pと q に対し, 任意の $x \in X_\mu$に対して, $(p + q)\ R_\mu\ x$となるのは$(p\ R_\mu\ x)$ または $(q\ R_\mu\ x)$が成立するとき, かつそのときに限る. 言い換えれば, 任意のクラス pと q に対して, $M(p + q) = M(p) \cup M(q)$が成立する.

定理 1:

もしあるメトリクスμが上記の定義に従うなら, μ は W4 を満たす. なぜなら

$\mu(p) = \mid M(p) \mid$ (C1)より

$\leq \mid M(p) \cup M(q) \mid$ なぜなら, $M(p) \subseteq M(p) \cup M(q)$

$= \mid M(p + q) \mid$ (C2)より

$= \mu(p + q)$ (C1)より

定理 2

もしあるメトリクスμが上記の定義に従うなら, μ は¬W6 を満たす. なぜなら

$\mu(p) + \mu(q) = \mid M(p) \mid + \mid M(q) \mid$ (C1)より

$\geq \mid M(p) \cup M(q) \mid$

なぜなら $\mid M(p) \cup M(q) \mid = \mid M(p) \mid + \mid M(q) \mid - \mid M(p) \cap M(q) \mid$

$= \mid M(p + q) \mid$ (C2)より

$= \mu(p + q)$ (C1)より

NOC, CBO, RFC を上記の形式に従って定義し, これらのメトリクスが W4 と¬W6 を満たすことを示す. 集合 X_{NOC}, X_{CBO}, X_{RFC} はプログラム中のすべてのクラスとする. 関係 $c\ R_{NOC}\ x$は「の親クラスである」(クラス xはクラス cから導出される), 関係 $c\ R_{CBO}\ x$は「結合する」(クラス cはクラス x に結合する), 関係 $c\ R_{RFC}\ x$ は「参照する」(クラス c のあるメソッドが, メソッド xを呼び出す)[1], とする.

NOCN, NOCR, CBON, CBOR, RFCN, RFCR を(NOC, CBO, FRC の定義を修正

[1] 厳密に言うと, CBO は必ずしも W4 を満たさない. なぜなら, X_{CBO} はクラスの合成によって変化する可能性があるから. クラスpと q が合成されたとき, p と q は X_{CBO} から取り除かれ, クラス $p + q$ が X_{CBO}に付け加えられる. たとえば, $M(p) = \{ q \}$かつ $M(q) = \{ p \}$とすれば, $M(p + q) = \phi$となり, $\{ p, q \}$とはならない. $CBO(p) = 1$, $CBO(q) = 1$ かつ $CBO(p + q) = 0$ であるから, $\mid CBO(p) \mid > \mid CBO(p + q) \mid$となる.

することにより), 上記の形式に従って定義して, これらのメトリクスが W4 と¬W6 を満たすことを示す. X_{NOCN}, X_{CBON}, X_{RFCN} はプログラム中のすべての新規開発クラスの集合とする. X_{NOCR}, X_{CBOR}, X_{FCR} はプログラム中のすべての再利用クラスの集合とする. R_{NOCN} と R_{NOCR} は R_{NOC} と等価な関係とする. R_{CBON} と R_{CBOR} は R_{CBO} と等価な関係とする. R_{RFCN} と R_{RFCR} は R_{RFC}と等価な関係とする.

メトリクス DIT(depth of inheritance tree of a class) は W4 を満たさないので, DITN と DITR が W4 を満たすかどうかは評価しない. DIT が¬W6 を満たすことを示すために, Chidamber と Kemerer は 2 つの仮定を置いた:(1)クラスの合成によって, 合成されるクラスの先祖クラスは変更されない. (2)合成されてできたクラスは, クラス階層木の中で, 元のクラスのどちらか一方があった場所に位置する. 結果として, $DIT(p + q) = DIT(p)$または $= DIT(q)$となり, $DIT(p + q) \leq DIT(p) + DIT(q)$となる, すなわち, ¬W6 を満たす. DITN と DITR もまた計測対象の先祖クラスによって決定されるため, DITN と DITR もまた, 先ほどの等式を満たし, 従って¬W6 を満たす.

3.5. 実験概要

次に, CK メトリクスと修正された CK メトリクスの違いを実験データにより統計的に評価する.

実験データは, 日本ユニシス株式会社の 1996 年度新人研修における C++プログラム開発演習から収集された. 研修生(被験者)は演習の前に, オブジェクト指向設計, オブジェクト指向言語について講習を受けている. この演習では, 6 つのチームが独立に同じ課題を行った. 各チームは 4～5 名の被験者で構成されている.

開発プロセスはウォーターフォールモデルで行われた. すなわち, 要求仕様定義, 設計, コーディング, レビュー, 単体テスト, 結合テストのフェーズを経た. 課題プログラムはいわゆる酒屋問題[51]を拡張したもので, データベースを用いた在庫管理, パスワードによるオペレータ認証, 売上データのグラフィカルな表示, 売上予測等の機能を持つ. 課題が渡された時点で, データベースの構造, 入出力ファイルフォーマット, および被験者が開発すべきサブシステム(パスワード管理サブシステム, 等)が決定されている. つまり, 要求仕様定義フェーズと設計フェーズの一部が終了していることになる. 開発期間は 5 日間である. 開発されたプログラムは, インストラクターによってテストされ, 要求仕様を満たすことが確認される.

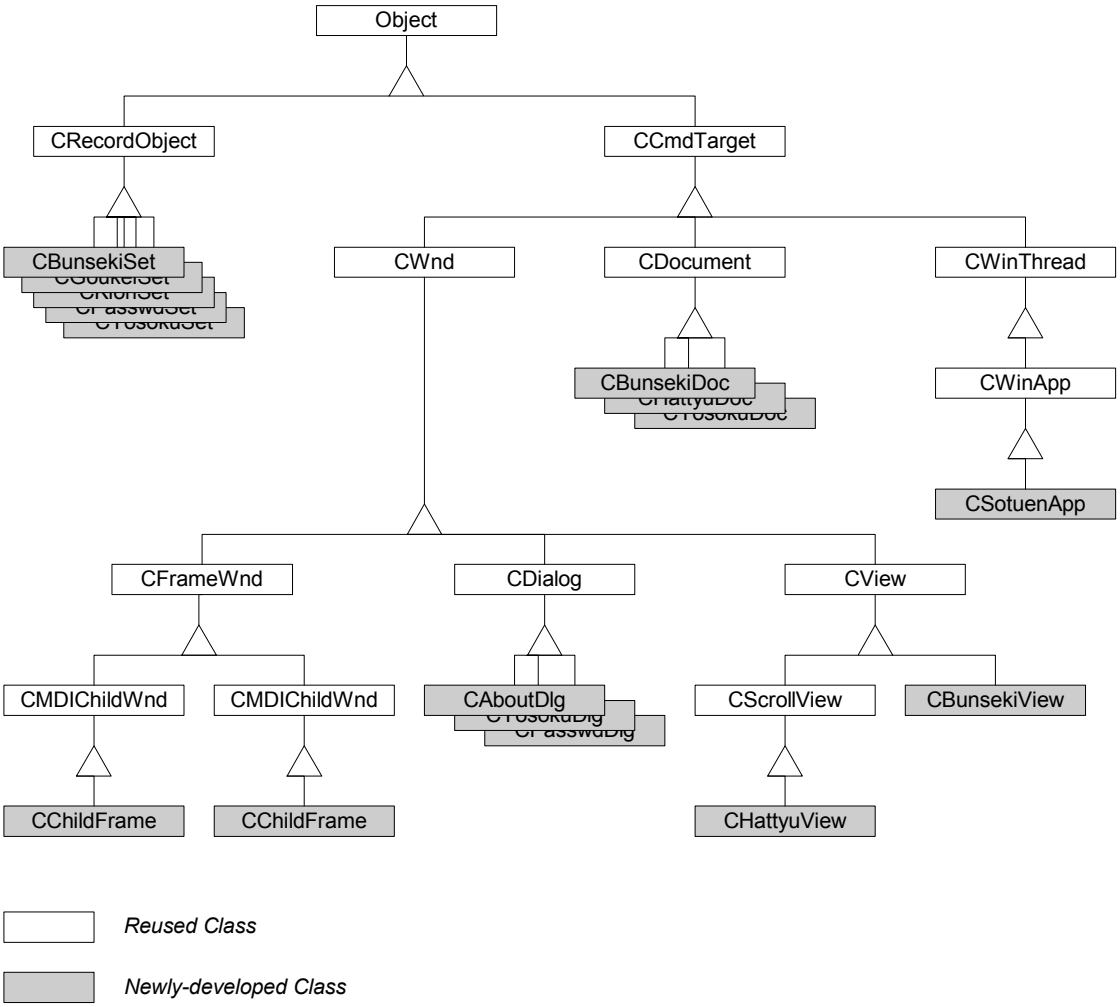


図 3.2 開発されたあるプログラムのクラス階層

プログラミング言語はC++であり，処理系はMicrosoft Visual C++である．フレームワークとしてMicrosoft Foundation Class(MFC)を用いた．MFCを用いることは課題の重要な要件であり，ユーザーインターフェイスとデータベースインターフェイスはすべてMFCのクラスを用いて実装される．図 3.2 に，本実験において，あるチームによって開発されたアプリケーションのクラス階層を示す．図では，新規開発されたクラスは網掛けで示されている．新規開発のクラスはすべてクラスライブラリのクラスから派生していることがわかる．

被験者はそれぞれ，割り当てられたパーソナルコンピュータ(PC)上で作業を行う．各PCおよびサーバーは同一のネットワークに接続されている．サーバーは1時間おきに被験者の作業ディレクトリをバックアップすることで自動的にプログラムソースファイルを収集する．

発見されたエラーは，レビュー報告書，単体テスト報告書，結合テスト報告書に記入される．それぞれのエラーについて，エラーの修正までの作業を記入する報告書がある．メトリクスの

値は，上記の報告書に記載されるエラーが含まれる時点，コードレビュー直前のプログラムソースファイルから算出した．

今回の開発では大規模な再利用が行われている．新規開発部分については，行数でチーム当たり3000行程度であり，これには空白行やツールによって生成された行が含まれる．また，開発されたクラスは，すべてクラスライブラリから派生したものである．一方，再利用した部分については，行数でチーム当たり1万行程度である．

3.6. 分析

開発はチームを構成して行われているが，課題プログラムは独立した部分プログラムに分割され，チームのメンバーに割り当てられる．実際に，部分プログラム間に渡るようなエラーはほとんど発見されておらず，各開発者は同じチームに属する他のメンバーの開発による影響を受けていない．従って，以降の分析は被験者単位で行っている．

なお，収集されたデータに不備のあった被験者は分析の対象から除いた．結果的には，19人のデータが分析対象となった．

表 3.2 各開発者のメトリクス計測値

Developer	Cc	WMC	DIT	DITN	DITR	NOC	NOCN	NOCR	CBO	CBON	CBOR	RFC	RFCN	RFCR	LCOM	Ec	Et(min.)
t1	6	33	17	0	17	0	0	0	38	8	30	75	31	44	73	7	112
t2	3	19	10	0	10	0	0	0	16	3	13	38	17	21	47	2	50
t3	4	22	14	0	14	0	0	0	21	3	18	58	19	39	46	5	315
t4	2	7	6	0	6	0	0	0	8	0	8	14	9	5	16	0	0
t5	3	19	10	0	10	0	0	0	17	3	14	41	17	24	47	2	390
t6	2	8	6	0	6	0	0	0	8	0	8	13	9	4	14	2	114
t7	3	19	10	0	10	0	0	0	17	2	15	40	17	23	47	3	21
t8	4	20	14	0	14	0	0	0	18	0	18	32	24	8	49	7	891
t9	9	8	6	0	6	0	0	0	9	0	9	16	8	8	13	0	0
t10	4	25	16	0	16	0	0	0	24	2	22	58	24	34	62	5	530
t11	3	21	12	0	12	0	0	0	18	1	17	52	19	33	52	8	576
t12	4	24	16	0	16	0	0	0	20	1	19	50	22	28	59	8	100
t13	2	8	6	0	6	0	0	0	9	0	9	16	8	8	13	1	60
t14	6	38	20	0	20	0	0	0	37	3	34	90	35	55	88	4	850
t15	3	22	12	0	12	0	0	0	20	3	17	55	20	35	55	3	154
t16	4	26	16	0	16	0	0	0	23	3	20	67	24	43	57	1	94
t17	2	11	6	0	6	0	0	0	10	0	10	24	10	14	11	1	90
t18	3	17	10	0	10	0	0	0	13	0	13	24	16	8	47	3	75
t19	2	8	6	0	6	0	0	0	9	0	9	15	5	10	13	1	25

3.6.1. 実験データ

実験によって収集されたデータを表 3.2 に示す. それぞれのメトリクス値は, その開発者についての合計である[29]. Ec(フォールト数), Et(フォールト修正時間)はそれぞれの開発者の報告書に基づいて算出されている.

表中ではそれぞれの数値は開発者ごとに集計されているが, その理由は, 開発されるシステムは 4 つのサブシステム(モジュール)として開発チームに手渡され, ぞれぞれのメンバーが開発したためである. さらに, サブプログラムにまたがったフォールトは観察されなかった. それゆえ, メンバー個人の開発が他のメンバーからあまり影響を受けなかった(少なくともフォールトの発生に関しては)と考え, それぞれのメンバーを別々に分析することにした. 報告書に欠落, あるいは明らかな間違いがあるものは分析対象から外された. 結果として, 19 人のデータが分析対象となった.

メトリクスはコードレビュー直前のソースコード, すなわちフォールトを含むソースコードから収集した. 図 3.2 はこの実験においてあるチームが開発したプログラムのクラス階層である. すべての新規開発クラスはクラス階層の葉(末端)であり, したがって, NOC(そして NOCR, NOCN)は 0 になる. DITN の値もすべて 0 となった.

本実験においては再利用が積極的に行われていた. 新規開発のコードの量は約 3 千行(コメントを含む)になった. 新規開発クラスはすべて再利用クラスから導出されていた. 一方で, 再利用されたソースコードは 1 万行程度となった. 多くのクラスが再利用されたため, CBON, CBOR, CBO の間には大きな違いが見られた. 同様に, RFCN, RFCR, RFC の間にも大きな違いが見られた.

3.6.2. CK メトリクスとフォールトの相関

修正前の CK メトリクス(WMC, DIT, NOC, CBO, RFC, LCOM) とフォールト数(Ec)の相関, フォールト修正時間(Et)の相関を表 3.3 に示す. フォールト数と修正時間がともにメトリクス値と高い相関を持っていることが示されている. 高い相関係数を持つことは, CK メトリクスはフォールト数や修正時間を予測するために用いることができることを意味する.

表 3.3 メトリクスとフォールト数および修正時間の相関係数

メトリクス	Ec		Et	
WMC	0.622	**	0.721	**
DIT	0.684	**	0.767	**
NOC	-		-	
CBO	0.579	**	0.744	**
RFC	0.543	*	0.632	**
LCOM	0.652	**	0.699	**
DITN	-		-	
DITR	0.684	**	0.767	**
NOCN	-		-	
NOCR	-		-	
CBON	0.340		0.470	
CBOR	0.610	**	0.774	**
RFCN	0.653	**	0.772	**
RFCR	0.453		0.523	*

3.6.3. 修正 CK メトリクスと CK メトリクスの比較

表 3.2 には修正後のメトリクス(DITN, DITR, NOCN, NOCR, CBON, CBOR, RFCN, RFCR)とフォールト数，修正時間も示されている．CBO，CBON，CBOR のうちでは，CBOR が Ec と Et の両方について，最も高い相関を示している．図 3.3 は CBO と Et，図 3.4 と CBOR と Et の分布を示す．CBOR はフォールト数，修正時間との相関が CBO よりも大きく，より精度の高い予測が可能である．したがって，CBO に関しては，フレームワークのクラスのほうが，新規開発のクラスよりも複雑度に寄与していると考えられる．

一方では，RFC,RFN,RFCR に関しては RFCN が，Ec と Et の両方にもっとも高い相関を示す．図 3.5 は RFC と Et，図 3.6 は RFCR と Et の分布を示す．RFCN はフォールト数，修正時間との相関が RFC よりも大きく，より精度の高い予測が可能である．したがって，RFC に関しては，新規開発のクラスのほうが，フレームワークのクラスよりも複雑度に寄与していると考えられる．

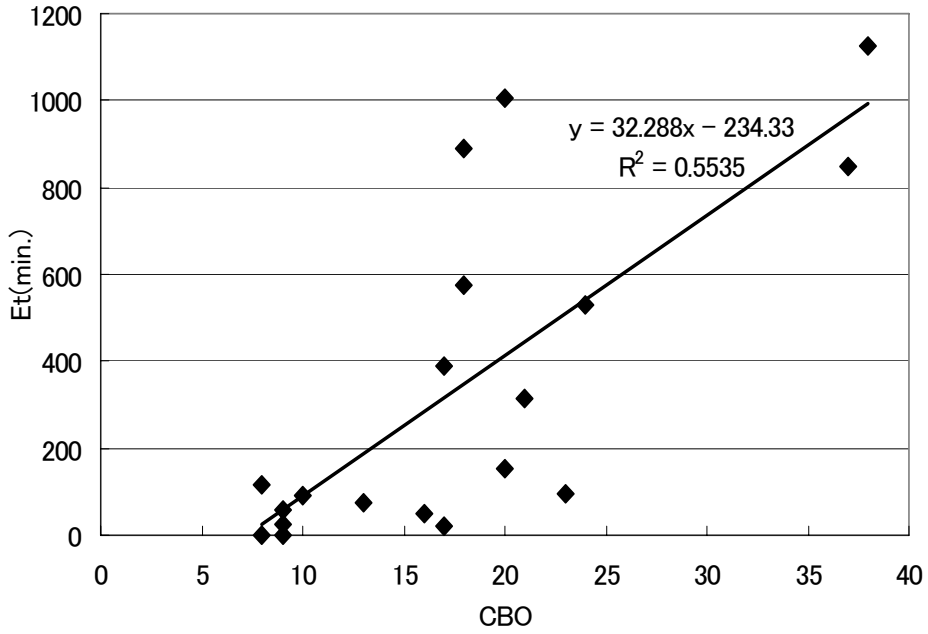


図 3.3 CBO と Et の分布図

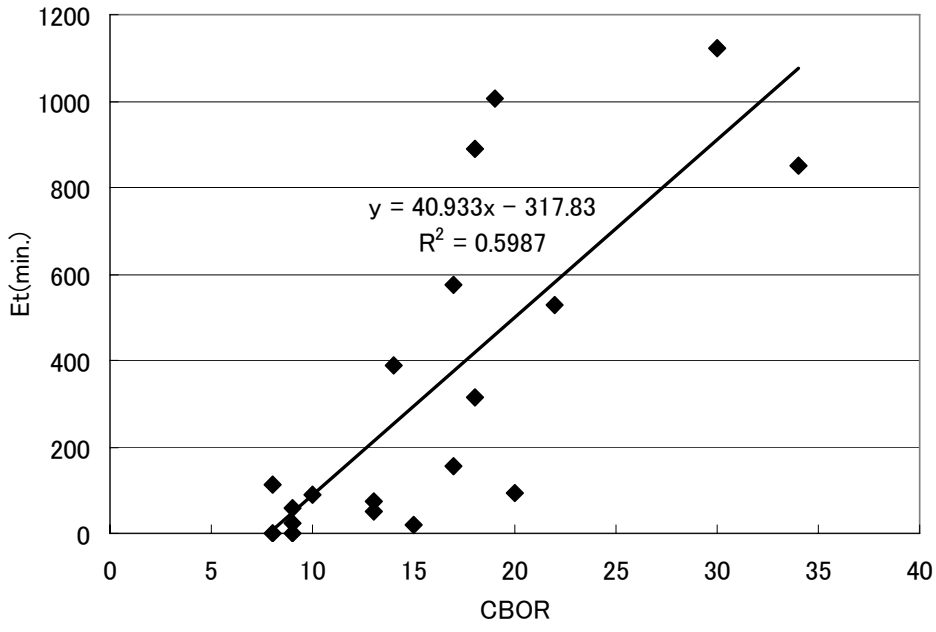


図 3.4 CBOR と Et の分布図

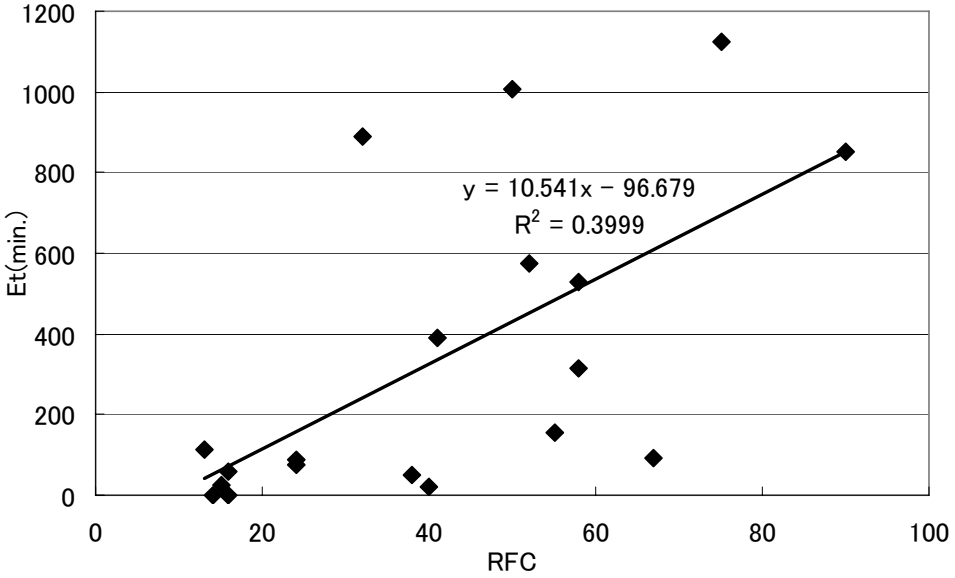


図 3.5 RFC と Et の分布図

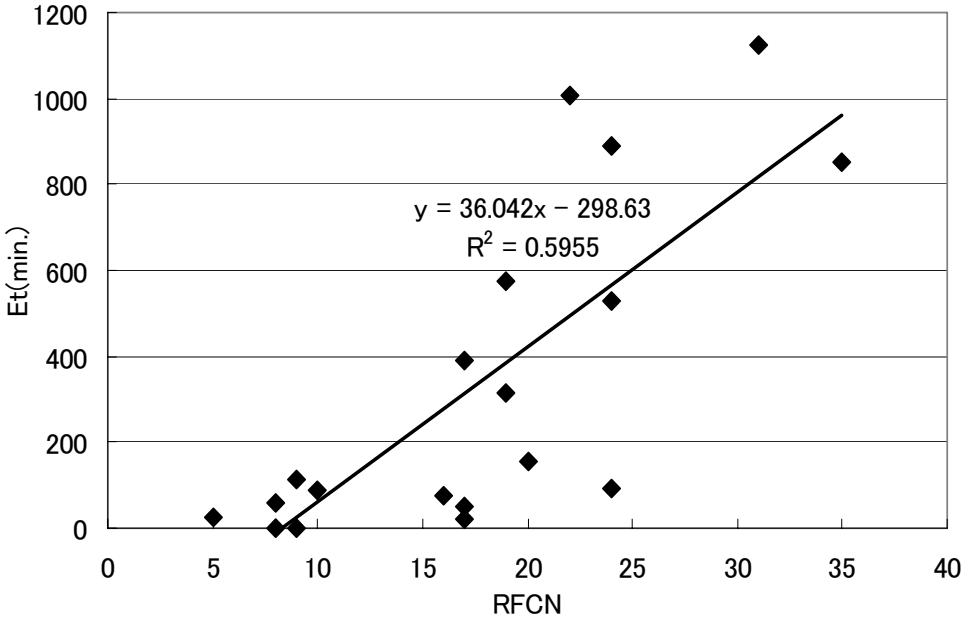


図 3.6 RFCR と Et の分布図

3.7. 結論と課題

本研究によって，複雑度メトリクスによってソフトウェアの複雑度を計測する際には，新規開発部分と再利用部分を区別して扱うべきであることが明らかにされた．

たとえば，本実験の場合には，再利用されるクラスに対する（メソッドを介した）参照は，新規開発クラスに対する参照と比較して，あまり複雑度を増やさない．一方で，再利用されるクラスに対して（インスタンス変数を介して）結合することは，新規開発クラスに対する結合と比較して，複雑度を増やす．これは，一般に言われている以下のような経験則の裏付けにもなっている．

(1) 開発者は必要な機能を持つクラスがフレームワークに存在するのであれば，フレームワークのクラスを利用すべきである．しかし，フレームワークのクラスが公開インスタンス変数を通じたアクセスを要求する場合には，より注意を払うべきである．開発者はそのようなクラスを再利用するには，新規開発クラスと同程度に，内部の詳細を知っていなければならない．

(2) 同じ理由により，開発者は，導出と合成（composition）の両方を使える場合には，合成を使うべきである．導出では，子クラスは親クラスの「限定公開（protected）」インスタンス変数を参照することができる（C++や Java で）．そのような参照は「公開（public）」インスタンス変数と同様に，情報隠蔽の原則に反する．合成では，他のクラスを部品として使うことになり，情報隠蔽の原則を破ることはない．

今回の実験においては，CK メトリクスを評価の対象にしたが，これら以外のいくつかの構造メトリクスに対しても，再利用部分と新規開発部分を区別して扱う手法は適用可能である．

4. フレームワークを用いたクラス分類法

4.1. 緒言

近年，ソフトウェアの応用分野の拡大と共に，ソフトウェアが大規模・複雑化してきている．それに伴い，開発期間の短縮やコストの削減・品質の向上が求められている．これらの要求を実現するために数多くのソフトウェア開発支援に関する研究が行われてきている．

開発支援のアプローチの 1 つはソフトウェア開発における各作業の効率化である．開発作業の効率化を目指して，これまでに多くのソフトウェア開発手法や CASE ツールが開発されてきた．最近では，オブジェクト指向パラダイムに基づいた分析・設計法，プログラミング言語等が数多く提案され，実際の開発現場でも使われている[2]．

オブジェクト指向開発の普及に伴い，設計やドメインに関する知識が蓄積されてきており，ドメインを限定することで大規模な再利用を可能とするフレームワークやコンポーネントのようなライブラリが登場してきた．また，デザインパターン[21]などの分析・設計を補助するための手法も提案されてきている．

一方，ソフトウェアメトリクスを用いた生産性や品質向上のアプローチも広く受け入れられている．ソフトウェアメトリクス[40]は，ソフトウェアプロダクトのさまざまな特性(複雑度，信頼性，効率など)を判別する客観的な数学的尺度である．メトリクスを用いてソフトウェアの状態を評価することで，問題の含まれる部分に対する変更を行う，あるいは，その部分に対するレビュー・テスト工数の割当を増やすという対処が施される．メトリクスを用いたアプローチを実行する際には，メトリクスによるフォールト予測手法を確立する必要がある．

これまでに，ソフトウェアメトリクスを用いて，エラー数やエラー修正工数等を予測する手法が提案されている[17][39]．オブジェクト指向ソフトウェアのエラー予測を行う場合には，クラスの種類に応じて予測式を別々に作成するほうがよいという指摘[4]もされているが，実際にそのような方法を用いてエラー予測精度を向上させるという報告はない．本論文では，アプリケーションフレームワークを用いた開発に限定することで，クラスを分類するための方法を示し，複雑度メトリクスを用いてクラスに含まれるエラーを予測するための手法を提案する．提案した手法をある企業で行われた C++プログラム開発に適用し，有効性を実験的に評価した．

4.2. クラス分類とフォールト予測

Basili のオブジェクト指向ソフトウェアのフォールトを予測する研究においては，多変量ロジスティック回帰分析を用い，クラスのメトリクス値を入力とし，クラスに作りこまれるフォールト

の有無(真偽値)を予測する統計的なモデルを作っている[4]. その実験において, クラスの種類(例えば, ユーザーインターフェイスを受け持つクラスか, データベースにアクセスするクラスか)によって, メトリクス値の分布やエラー予測の有効性に大きな違いがあることを指摘した. したがって, クラスの種類毎に異なる予測式を用いることで予測精度が向上することが期待されるが, 任意の開発においてクラスの種類分けをソースコードから自動的に行うことは困難である.

4.3. CK メトリクスによる一般的なフォールト予測手法

プロダクトメトリクスを用いて予測を行う一般的な手順は 2.4 に示した. ここでは, 具体的に, CK メトリクスを用いてクラスのフォールト修正時間を予測する手順を説明する.

(1)基準となるデータの収集

CK メトリクスを計測するためのソースコードあるいはクラスの設計書, 発見された個々のフォールトによって修正されたクラス, 修正に要した時間の記録を収集する.

(2)予測式の作成

メトリクスデータとフォールトデータから, 回帰分析によって, クラスのフォールト修正時間を予測する式を作る. 予測式の入力(独立変数)はクラスのメトリクス値, 出力(従属変数)はクラスで発見されたすべてのフォールトについて修正時間を合計したものとなる. 修正時間を予測する場合には, 通常の(線形)回帰分析が適用される.

(3)エラーの予測

予測を行いたいクラスに対して, メトリクスを計測し, 予測式によってフォールトの修正時間を予測する.

4.4. クラス分類の手法

アプリケーションフレームワークを用いた開発においては, フレームワークのクラス階層は(厳密ではないにしても)クラスの種類を反映すると考えられる. また, フレームワークに含まれるクラスから導出によって新しいクラスを作成することが多く行われる. そのような開発において, クラスの種類の代わりに, そのクラスが導出されたフレームワークの親(または先祖)クラスを用いる. クラス分類によって予測式のパラメータ(係数)を変更することにより, エラー予測精度の向上が期待できる. クラス分類はフレームワークのクラス階層に依存するため, フレームワークごとに係数を用意する必要がある. クラス分類と CK メトリクスを用いてクラスのエラーを予測する手順は次のようになる.

(1)代表クラスの選出

フレームワークのクラスから，分類に適したクラス(以下「代表クラス」)を選出する．代表クラスは，導出によって新規開発クラスが作られると期待できるクラスから，フレームワークのアーキテクチャを考慮して，代表的なものを選出する．

(2)基準となるデータの収集

CK メトリクスを計測するためのソースコードあるいはクラスの設計書，発見された個々のフォールトによって修正されたクラス，修正に要した時間の記録を収集する．

(3)予測式の作成

クラスの親(または先祖)クラスがどの代表クラスであるかによって，収集データの新規開発クラスを分類する．分類ごとに，メトリクスデータとフォールトデータによって，フォールト修正労力を予測する式を作る．

(4)エラーの予測

予測を行いたいクラスに対して，分類にしたがって予測式を適用し，フォールト修正労力を予測する．

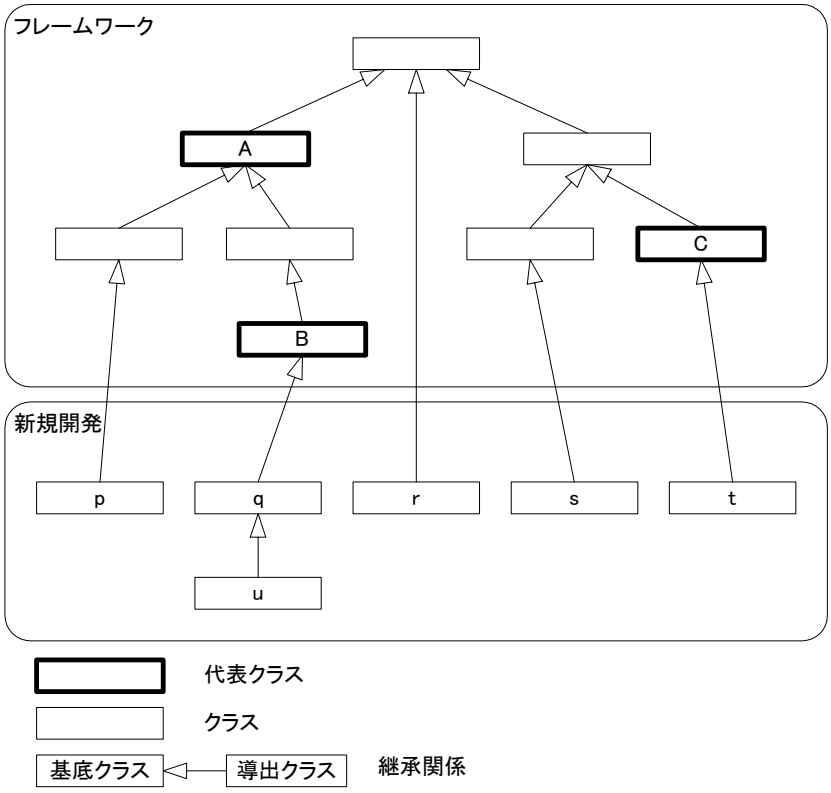


図 4.1 代表クラスの例

一般に，フレームワークのクラス間にも継承関係があるため，代表クラス間にも継承関係が生じる．子クラスは親クラスを特化したクラスであると考えられるため，分類としては子(あるいは子孫)側のクラスを用いるべきである．たとえば，図 4.1 において，代表クラス B は A の子孫であるため，新規開発クラス q と u の分類は B となる．

また，C++などの多重継承を許すプログラミング言語を使用した場合には，代表クラス間で継承関係がなくても，あるクラスの代表クラスが複数になる可能性がある．もしこのような状況が発生すると想定される場合には，あらかじめ代表クラス間の優先順位を指定する，あるいは，多重継承しているクラスは別の分類とする，などの対策が必要である．

可能であれば，すべてのクラスが何らかの分類に所属するような分類を設ける．ただし，分類が細かくなり，サンプル数が少ない(少数のクラスしか所属しない)分類ができた場合，有意な予測式の係数を導くことができない(予測を行うことができない)．従って，一部のクラスに関して予測をしないようにする，あるいは，「その他」分類を設ける，などの手段を講じる必要がある．

4.5. 評価実験

4.5.1. 実験概要

提案した手法を，ある企業で行われた新人研修におけるプログラム開発プロジェクトで得られたデータに適用した．このプロジェクトでは，GUI(Graphical User Interface)を備えた電子メールの配送システムを開発する．システムは 5 つのサブシステム(SMTP サーバー，POP サーバー，DELIVER サーバー，SMTP クライアント，POP クライアント)から構成されている．開発チームは 4 から 5 人の開発者で構成され，開発者は各サブシステムを開発する．プロジェクト開始時に開発チームに各サブシステムの仕様が渡され，6 日間で，設計，実装，テストを行う．最終的にインストラクターによる受け入れテストが実施される．プログラミング言語として C++，コンパイラとしては Visual C++を用い，フレームワークとして MFC(Microsoft Foundation Class)を用いる．開発規模はチームあたり 3000 行程度(再利用分を含まない) である．

4.5.2. クラス分類

今回の開発におけるドメインおよびフレームワーク MFC のアーキテクチャを考慮し，代表クラスとして以下の 6 つを選定した．

(a)CDocument

派生クラスにはプログラムのデータを処理する部分が記述される．

(b)CView

派生クラスにはユーザーに対してデータを表示する部分が主に記述される.

(c)CDialog

派生クラスには, ユーザーからデータを受け取る部分と, ユーザーに対してエラーメッセージを出す部分が主に記述される.

(d)CWinApp

派生クラスには, アプリケーションの設定に関する処理(「アプリケーションが前回実行された時のウィンドウの位置と大きさを覚えておく」など)が記述される.

(e)CFrameWnd

複数のビューを持つプログラムの場合, それらを管理するためのコードが CFrameWnd 派生クラスに記述される. (ユーザーインターフェイスが複雑になると, 複数のビューを切り替える方法は良く用いられる.)

(f)CSocket

CSocket は, ネットワーク通信を行う「ソケット」を実装しているクラスである.

これらのいずれからも派生しないクラスは, **(g)その他**に分類される. 今回の実験では, 選出された代表クラス間に継承関係は存在せず, 複数の代表クラスを継承するクラスが定義されることもなかった.

4.5.3. 実験データ

実験において収集されたクラスのソースコードとエラーのデータから, 記録に不備があるもの, および, 開発ツールによって生成されたあと一切変更されていないクラスに関するデータを取り除いた. 最終的には, 17 人分, 124 のクラスに関するデータが利用できた. 本実験では, 複雑度メトリクスとして CK メトリクス[14]およびその修正されたメトリクス(3.3 参照), NIV(計測対象クラスのインスタンス変数の数)[33], SLOC を用いた. クラス分類ごとに, 抽出したメトリクス値, およびエラー個数, 修正時間の統計量を表 4.1 から表 4.3 に示す. 全クラスについての同統計量を表 4.8 に示す.

表 4.1 分類 CDialog のメトリクスの統計量（サンプル数 15）

	CBO	CBOR	CBON	RFC	RFCR	RFCN	WMC	LCOM	DIT	NOC	NIV	SLOC	Ec	Et (min.)
Min.	0	0	0	2	2	0	2	0	4	0	2	44	0	0
Max.	3	0	3	12	7	5	7	19	4	0	6	204	2	83
Ave.	0.67	0.00	0.67	4.20	3.20	1.00	3.20	2.93	4.00	0	4.33	71.13	0.13	5.51
Std. Dev.	0.98	0.00	0.98	2.86	1.47	1.56	1.47	4.73	0.00	0	1.11	41.53	0.52	21.33

表 4.2 分類 CDocument のメトリクスの統計量（サンプル数 19）

	CBO	CBOR	CBON	RFC	RFCR	RFCN	WMC	LCOM	DIT	NOC	NIV	SLOC	Ec	Et (min.)
Min.	1	0	1	10	7	3	7	21	3	0	3	77	0	0
Max.	4	1	3	25	20	10	20	148	3	0	14	420	6	255
Ave.	1.63	0.37	1.26	16.37	11.53	4.84	11.53	54.95	3.00	0	8.26	204.00	1.26	37.00
Std. Dev.	0.83	0.50	0.56	4.78	3.45	1.95	3.45	30.29	0.00	0	3.75	98.67	1.73	71.58

表 4.3 分類 CView のメトリクスの統計量 （サンプル数 17）

	CBO	CBOR	CBON	RFC	RFCR	RFCN	WMC	LCOM	DIT	NOC	NIV	SLOC	Ec	Et (min.)
Min.	2	1	1	11	8	2	8	28	4	0	3	76	0	0
Max.	5	1	4	27	20	8	20	190	6	0	7	300	7	86
Ave.	3.59	1.00	2.59	16.59	12.47	4.12	12.47	77.35	5.53	0	3.82	137.94	0.94	10.03
Std. Dev.	1.00	0.00	1.00	4.53	3.79	1.54	3.79	48.61	0.87	0	1.33	60.17	2.30	24.02

表 4.4 分類 CWinApp のメトリクスの統計量(サンプル数 17)

	CBO	CBOR	CBON	RFC	RFCR	RFCN	WMC	LCOM	DIT	NOC	NIV	SLOC	Ec	Et (min.)
Min.	4	3	1	7	3	4	3	3	4	0	2	66	0	0
Max.	4	3	1	8	3	5	3	3	4	0	2	78	0	0
Ave.	4.00	3.00	1.00	7.71	3.00	4.71	3.00	3.00	4.00	0	2.00	72.12	0.00	0.00
Std. Dev.	0.00	0.00	0.00	0.47	0.00	0.47	0.00	0.00	0.00	0	0.00	3.28	0.00	0.00

表 4.5 分類 CFrameWnd のメトリクスの統計量(サンプル数 17)

	CBO	CBOR	CBON	RFC	RFCR	RFCN	WMC	LCOM	DIT	NOC	NIV	SLOC	Ec	Et (min.)
Min.	1	0	1	9	6	3	6	15	4	0	3	60	0	0
Max.	3	0	3	26	16	10	16	116	4	0	11	302	17	600
Ave.	1.24	0.00	1.24	13.00	7.59	5.41	7.59	27.29	4.00	0.00	4.71	107.59	1.24	38.74
Std. Dev.	0.56	0.00	0.56	4.32	2.37	2.18	2.37	23.82	0.00	0.00	1.86	59.51	4.13	145.32

表 4.6 分類 CSocket のメトリクスの統計量(サンプル数 19)

	CBO	CBOR	CBON	RFC	RFCR	RFCN	WMC	LCOM	DIT	NOC	NIV	SLOC	Ec	Et (min.)
Min.	0	0	0	0	0	0	0	0	3	0	0	31	0	0
Max.	0	0	0	22	22	0	22	157	3	0	10	361	1	1
Ave.	0.00	0.00	0.00	2.74	2.74	0.00	2.74	8.84	3.00	0.00	3.26	65.21	0.16	0.12
Std. Dev.	0.00	0.00	0.00	4.86	4.86	0.00	4.86	35.90	0.00	0.00	2.33	72.91	0.37	0.32

表 4.7 分類その他のメトリクスの統計量(サンプル数 20)

	CBO	CBOR	CBON	RFC	RFCR	RFCN	WMC	LCOM	DIT	NOC	NIV	SLOC	Ec	Et (min.)
Min.	0	0	0	0	0	0	0	0	0	0	1	5	0	0
Max.	2	0	2	9	8	1	8	16	2	0	5	416	8	78
Ave.	0.25	0.00	0.25	3.35	3.15	0.20	3.15	2.90	0.65	0	3.35	70.95	0.70	8.54
Std. Dev.	0.55	0.00	0.55	2.43	2.11	0.41	2.11	3.63	0.67	0	1.35	86.58	1.87	20.78

表 4.8 全体のメトリクスの統計量 (サンプル数 124)

	CBO	CBOR	CBON	RFC	RFCR	RFCN	WMC	LCOM	DIT	NOC	NIV	SLOC	Ec	Et (min.)
Min.	0	0	0	0	0	0	0	0	0	0	0	5	0	0
Max.	5	3	4	27	22	10	22	190	6	0	14	420	17	600
Ave.	1.58	0.60	0.98	9.09	6.24	2.85	6.24	25.35	3.36	0	4.27	104.85	0.65	14.42
Std. Dev.	1.61	1.03	1.00	6.83	4.97	2.62	4.97	38.38	1.49	0	2.74	82.93	2.04	62.68

クラス分類ごとの，メトリクス値のレーダーチャートを図 4.2 から図 4.8 に示す．各グラフの，細い線で描かれた一つの多角形が，一つのクラスについてのメトリクス値を表わす．メトリクスの値は，すべてのクラスについての平均が 1.0 となるように正規化されている．太い線で描かれた多角形は，その分類に属するクラスすべてのメトリクス値の平均である．CBO, RFC, WMC, LCOM, DIT は CK メトリクス，NIV はクラスのインスタンス変数の数，SLOC はクラスのソースコードの行数である．CBOR と RFCR はそれぞれ，CBO，RFC を修正したメトリクスである．メトリクス NOC，CBON，CBOR はグラフには描かれていない(NOC はすべてのクラスについて 0 であったため，CBON(および RFCN)は CBO と CBOR(RFC と RFCR)の差に常に等しくなるため)．

分類毎の平均値(太い線) のメトリクス値の傾向は，CDocument，CView，CWinApp，CFrameWnd で大きく異なっている．たとえば，CDocument と CView は多くのメソッドを備えており(WMC が大きく)，他のクラスのメソッドも多く呼び出す(RFC が大きい)点は共通である．しかし，CDocument はアプリケーションのデータを格納するため多くのインスタンス変数を備えている(NIV が大きい)のに対して，CView はあまり多くのインスタンス変数を持たない．CWinApp はスレッドや例外処理などのライブラリクラスに多く結合する(CBOR が大きい)．CFrame は CBOR を除いて平均的な値となっている．これに対して，分類 CDialog，CSocket，その他はいずれも，計測されたメトリクス値が小さいため，差が出にくくなっている．また，分類毎の平均値と，分類に属する個々のクラス(細い線)のメトリクスは互いに似た傾向を示しており，クラス分類が適切であったことの傍証となっている．

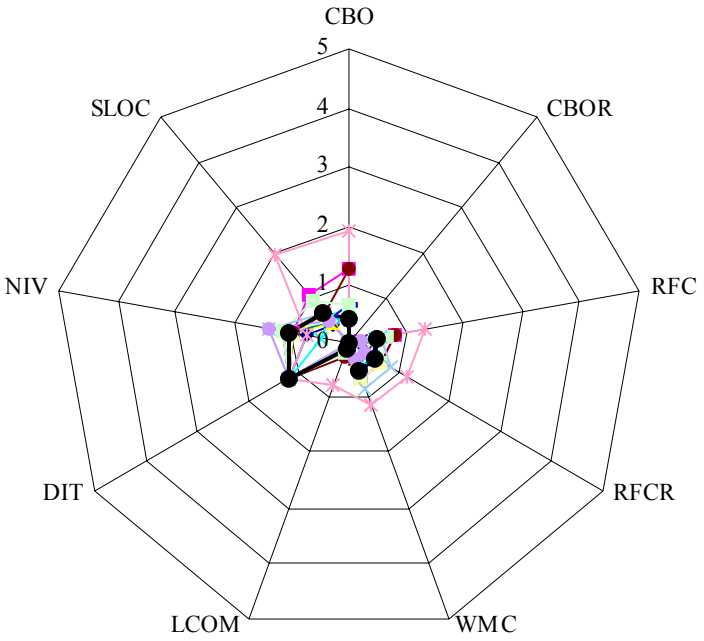


図 4.2 分類 CDialog のメトリクス値

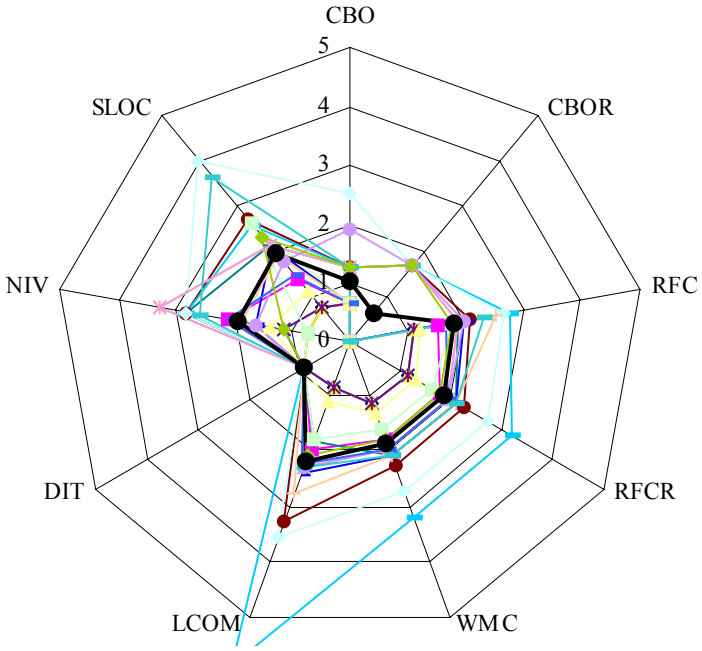


図 4.3 分類 CDocument のメトリクス値

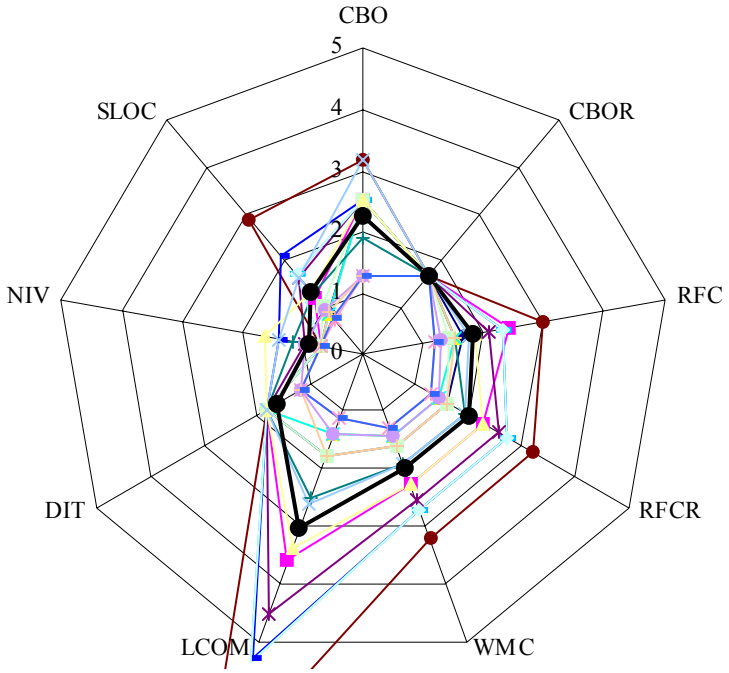


図 4.4 分類 CView のメトリクス値

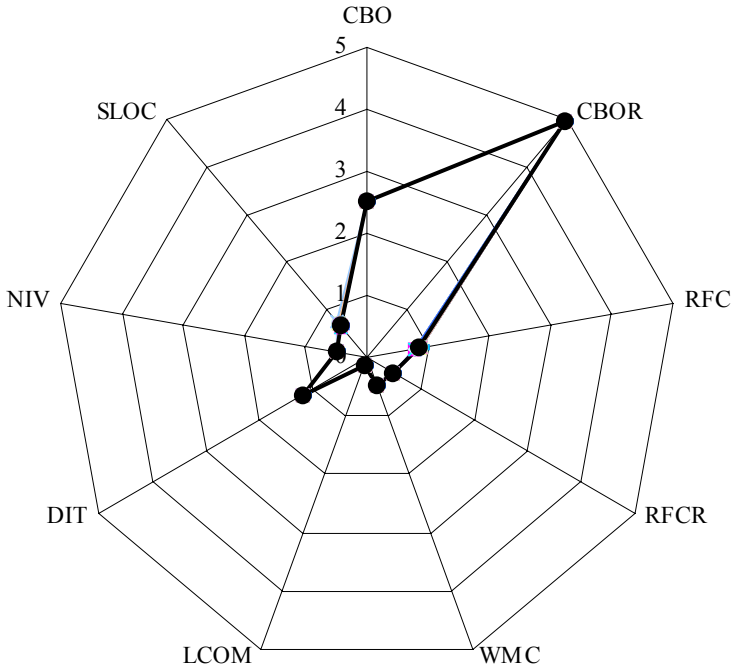


図 4.5 分類 CWinApp のメトリクス値

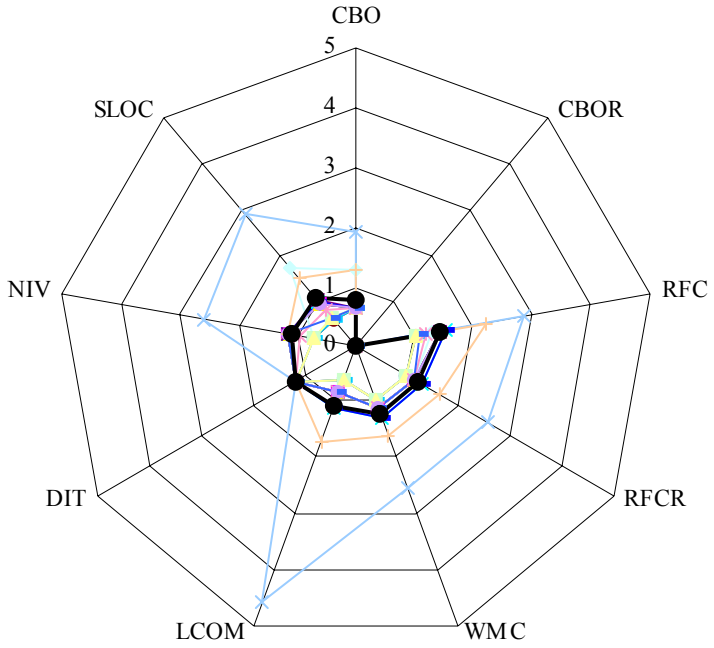


図 4.6 分類 CMainFrame のメトリクス値

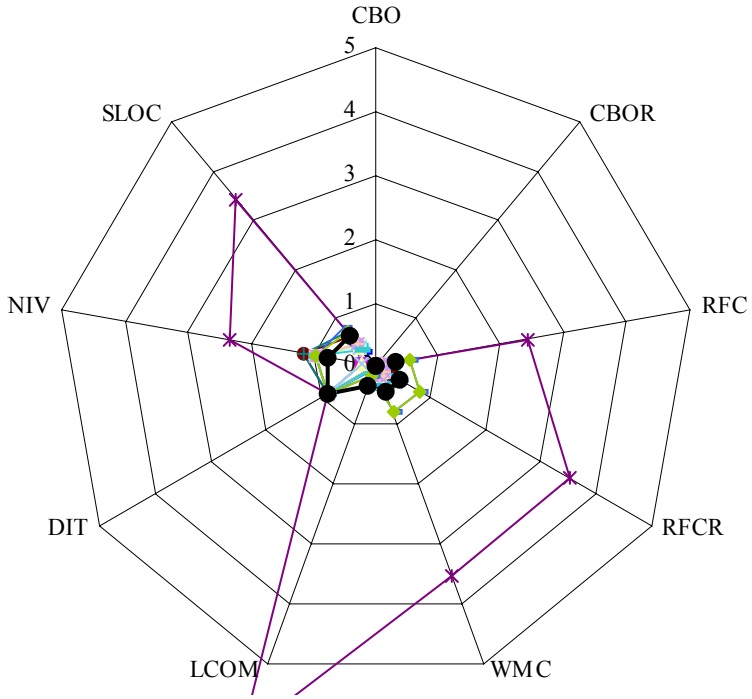


図 4.7 分類 CSocket のメトリクス値

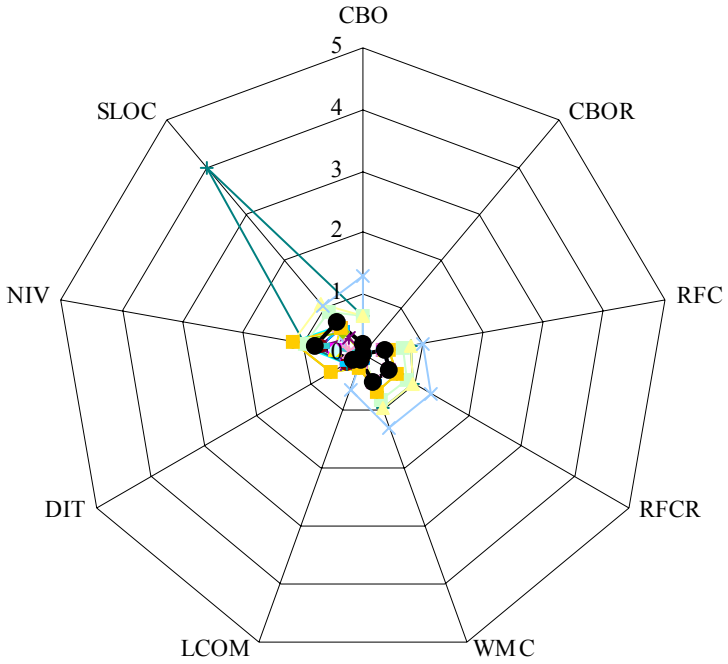


図 4.8 分類その他のメトリクス値

4.5.4. 分析

クラス分類の効果を調べるため，クラス分類を行わない場合と行った場合について，フォールト修正時間の予測精度を比較する．

2.4 で示した手順に従い，メトリクスの計測値を独立変数，フォールト修正時間を従属変数とする回帰式を，重回帰分析[30][35]によって求めた．変数減少法を用い，サンプル数や寄与率に照らして統計的に有意とならない独立変数は取り除いてある．たとえば，CBON，CBOR，CBO の間には CBON + CBOR = CBO という関係が成り立つため，3 変数がともにひとつの回帰式に含まれることはない．回帰式の一般形は次のようになる(*ET* は予測フォールト修正時間，*Const* は定数，C_{CBO}, …, C_{SLOC} は各メトリクスの係数である)．

$$ET = Const + c_{CBO} \cdot CBO + \Lambda + c_{SLOC} \cdot SLOC$$

クラス分類ごとの回帰式の係数を表 4.9 に示す．比較のために，分類せずに求めた回帰式の係数も示した(All の欄)．エラー修正時間の予測値と実測値をプロットしたものを図 4.9(分類せず)および図 4.10(分類を用いる)に示す．グラフ中の点がクラスであり，横軸がそのクラスの予測フォールト修正時間，縦軸が実測フォールト修正時間を表す．分類せずに予測した場合の決定係数(R^2)は 0.11，クラス分類を行った場合の R^2 は 0.89 であり，フォールト予測精度は，クラス分類を行ったほうが向上していることがわかる．

次に，突出した修正時間を持つ 2 つのクラス(Et=255，Et=600)を外れ値とみなして，データから取り除いた上で分析を行った．回帰式の係数を表 4.10 に，フォールト修正時間の予測値と実測値をプロットしたものを図 4.11（分類せず）および図 4.12（分類を用いる）に示す．分類せずに予測した場合の R^2 は 0.28，クラス分類を行った場合の R^2 は 0.55 であり，予測精度は，クラス分類を行ったほうが向上していることがわかる(相関係数で比較すると分類を行わない場合に 0.53，分類を行った場合には 0.74 となる)．

4.5.5. クラス分類の統計的な意味

クラス分類を行うことによって予測精度が改善する理由を考察する．クラス分類は，分類間に順序や間隔が定義されないため，名義尺度と呼ばれるメトリクスである．クラス分類ごとに異なった予測式を用いるということは，すなわち，クラス分類をダミー変数として取り入れたような単一の予測式を用いることと等価である．今回の実験では，クラス分類ごとにメトリクスの計測値の分布が大きく異なっていたため，クラス分類を取り入れることによって予測精度が改善している．

また，クラス分類もひとつのメトリクスであるため，予測式に取り入れるかどうかは統計的判断(クラス分類が統計的予測にどの程度寄与するか)によるべきである．本稿では，クラス分類の数がそれほど多くなく，クラス分類ごとにメトリクスの分布が大きく異なり，また，どの分類についても適当な数のクラスが属していたため，クラス分類は無条件で予測式に取り入れられることとした(すなわち，クラス分類ごとに異なった予測式を立てることとした)．このような条件が成立しない場合，たとえば，クラス分類の数が多い，あるいはある分類に含まれるクラスの数が少ない，クラス分類間にメトリクスの分布の差が見られない，などの場合には，統計的検定によってクラス分類を取り入れるべきかどうかを決定する必要がある．

表 4.9 全データによるフォールト修正時間予測式の係数

Coefficient	All	CDialog	CDocument	CView	CWinApp	CFrameWnd	CSocket	Others
(Constant)	-11.4	-22.1	-98.4	-22.9	0	-614	-0.13	20.9
C_{CBO}								
C_{CBOR}								
C_{CBON}			47.9			70.6		-169
C_{RFC}								-11.2
C_{RFCR}								
C_{RFCN}		-6.76				-67.8		264
C_{WMC}						119		
C_{LCOM}		3.52	1.36			-21.5		14.8
C_{DIT}								-18.7
C_{NOC}								
C_{NIV}				8.61				-4.83
C_{SLOC}	0.246	0.338				5.76	0.00379	

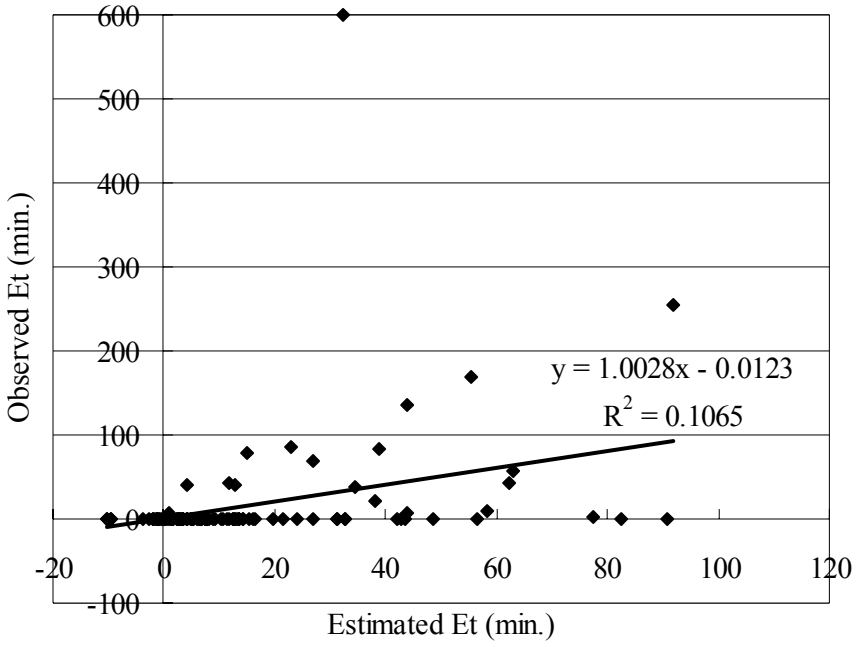


図 4.9 クラス分類を用いないフォールト修正時間予測(全データ)

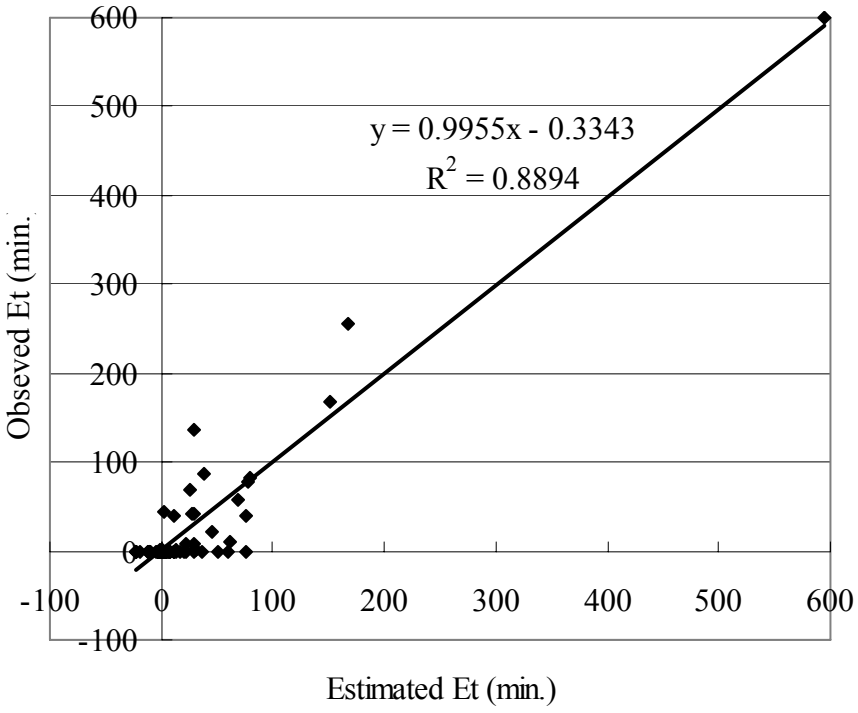


図 4.10 クラス分類を用いたフォールト修正時間予測(全データ)

表 4.10 外れ値を除いたデータによるフォールト修正時間予測式の係数

Coefficient	All	CDialog	CDocument	CView	CWinApp	CFrameWnd	CSocket	Others
(Constant)	5.46	-22.1	-36.7	-22.9	0	39.6	-0.13	20.9
C_{CBO}								
C_{CBOR}								
C_{CBON}	6.04					-1.51		-169
C_{RFC}								-11.2
C_{RFCR}						-10.8		
C_{RFCN}		-6.76				-0.657		264
C_{LCOM}	0.161	3.52	1.16			1.53		14.8
C_{DIT}	-5.37							-18.7
C_{NOC}								
C_{NIV}	2.49			8.61		0.758		-4.83
C_{SLOC}		0.338				0.00521	0.000379	

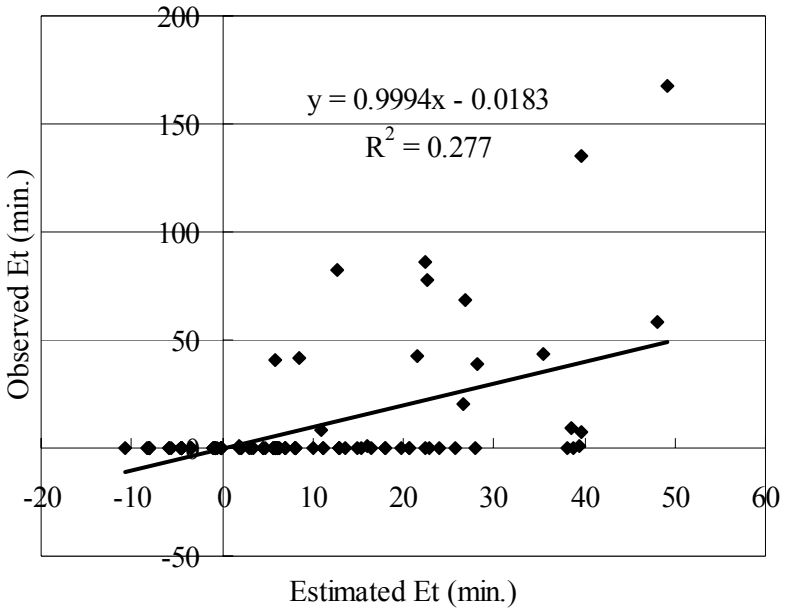


図 4.11 クラス分類を用いないフォールト修正時間予測(外れ値を除く)

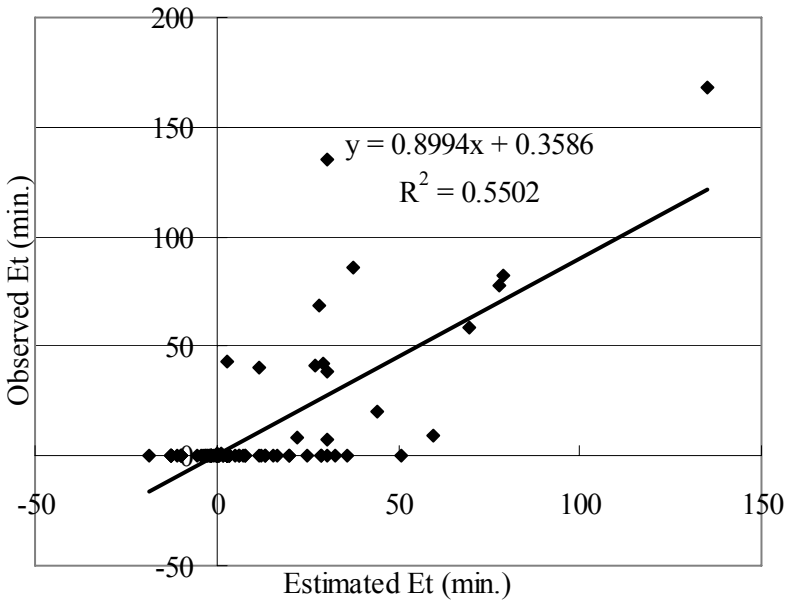


図 4.12 クラス分類を用いたフォールト修正時間予測(外れ値を除く)

4.6. 結論と課題

本研究では, C++言語およびアプリケーションフレームワークを用いた開発において複雑度メトリクスを用いてフォールト修正時間の予測を行う際に, クラス分類を行って予測精度を向上させる手法を提案し, 実験によってその有効性を評価した.

今後の課題としては, 以下の 3 点があげられる.

(a)クラス分類の精密化

例えば, Java 言語を用いて開発を行った場合には, インターフェイスや匿名クラスといった, 単なる継承とはみなせない機能がある. また, クラス階層以外の関係(委譲(delegation)やコンポジション(composition)など)もよく用いられるようになってきている. それらを考慮することで, より適切にクラスを分類できる可能性がある.

(b)代表クラス選出の自動化

本研究ではフレームワークに関する知識によって利用者が分類に用いられるクラスを選出する. クラス階層の構造やクラス間の関係, 統計的手段を用いてクラス分類を自動化できれば, 手法の利用がより簡単になると考えられる.

(c)適用事例の追加

より多くのプロジェクトに対してメトリクスの収集を行い, 手法の有効性を評価する. 例えば, 本稿における実験では, プログラムは開発されるだけで保守はされていない. より実際的な繰り返し型の開発プロセスに提案する手法を適用し, 有効性を確認するべきであると考えている.

5. CK メトリクスを開発プロセスの初期に計測する手法

5.1. 緒言

ソフトウェアのテストは，出荷するソフトウェアに含まれるフォールトを発見・除去するために必要な作業である．テストに費やされる労力は開発コストの50～80%に上るという報告もある[16]．従って，テスト労力の削減は，ソフトウェア開発の生産性を改善する重要な手段となる．レビューはテスト労力を削減するためのもっとも効果的な手段の一つである．レビューやテストを効率よく行うためには，フォールトが含まれると予測されるモジュールを特定することにより，レビューやテストの労力をそのモジュールに集中させることが効果的である[4]．ChidamberとKemererはCKメトリクスを提案し，2つのソフトウェア開発組織を観察し，オブジェクト指向プログラミング言語(C++とSmalltalk)で記述されたプログラムからCKメトリクスを収集した[14]．Basiliらも，CKメトリクスを用いてクラスにエラーが発生するか(fault-prone)を予測する実験を行った[4]．CKメトリクスは，オブジェクト指向設計を計測対象とする複雑度メトリクスであるにも関わらず，これらの実験においては，ソースコードを計測対象としている．CKメトリクスはソースコードからも，すなわち，コーディングフェーズにおいても計測することができるが，フォールトを修正するための労力を効果的に配分するためには，フォールトが発生する個所を早期のフェーズにおいて予測することが望まれる．

本研究では，設計の初期段階において，オブジェクト指向ソフトウェア向けのいくつかの複雑度メトリクス(主として CK メトリクス)を用いて，クラスにフォールトが発生するか(fault-prone)を予測する新しい手法を提案する．提案する手法では，OMT 法[42]に基づく分析/設計/実装フェーズに 4 つのチェックポイントを設ける．各チェックポイントにおいて，入手可能なプロダクトに関する情報(設計仕様書やソースコード)を用いて，複雑度メトリクスのうち適用可能なもののみを適用する．次に，多変量ロジスティック回帰分析を用いて，fault-prone なクラスを予測する．さらに，提案する手法の有効性を評価するために，あるコンピュータ会社で行われた開発プロジェクトで実験を行った．実験の結果，提案する手法を用いて，早期の段階でクラスのフォールト発生をある程度予測できることが確認された．

5.2. オブジェクト指向開発プロセスにおける段階的詳細化

これまでに，多くのオブジェクト指向設計手法が提案されてきている[7][15][42][43]．なかでも，Booch 法[7]と Rumbaugh の OMT(Object Modeling Technique)法[42]がよく知られたオブジェクト設計手法である．OMT はオブジェクト指向の分析・設計の技術的な側面のほとんどを取り込んだ手法であり，実際のプロジェクトの経験も取り入れたものになって

いる. 本論文では, オブジェクト指向設計手法としては OMT を対象とする.

OMT は, 分析, システム設計, オブジェクト設計の 2 つのフェーズから構成される. 分析フェーズでは, アプリケーションとアプリケーションが動作するドメインを理解しモデル化する. 分析フェーズの出力はシステムの基本的な側面を捉えた次の 3 つの形式的なモデルである.

(a)オブジェクトモデル:オブジェクトとそれらの関係

(b)動的モデル:制御の動的なフロー

(c)機能モデル:データ加工の制約

システム設計フェーズでは, まず, システムの全体的なアーキテクチャを決定する. 次に, オブジェクトモデルを参照しながら, システムをサブシステムに分割する. このとき, オブジェクトを並列に実行できるタスクにグループ分けすることで, 並列性を取り出す. また, プロセス間の通信, データの格納, 動的モデルの実装に関する全体的な決定を行う. 設計のトレードオフに関する優先順位も確立する. オブジェクト設計フェーズでは, 分析モデル(オブジェクトモデル, 動的モデル, 機能モデル)を洗練することで最適化を行い, 現実の設計を生成する. クラス内で用いられるアルゴリズムやデータ構造が決定される.

クラスの構造の観点からは, 以下の 6 つのステップで開発が進行する.

1. ターゲットシステムに含まれるクラスを特定する.
2. クラス間の参照関係を特定する.
3. クラスの属性を特定する.
4. クラス間の継承関係を特定する.
5. 機能モデルに基づいて操作を定義する.
6. 操作を実装するアルゴリズムを設計する.

5.3. 開発プロセス OMT とプロダクト

Chidamber と Kemerer の実験[4], Basili らの実験[14]のいずれにおいても, メトリクス値はソースコードから収集されている. その理由は次の 2 つである. (1)オブジェクト指向設計仕様書の標準的な記述法が確立されていなかったため, 実験のためだけに設計仕様書を作るのは不適当である. (2)ソースコードは設計仕様書を実装したものであり, メトリクス収集のために必要な情報はソースコードに含まれる.

設計仕様書が生成されるフェーズは, ソースコードが実装されるフェーズよりも早期である. 従って, メトリクスを設計仕様書に適用することで, より効果的にフォールト発生の予測を用いることが望ましい. フォールト発生の予測に基づいて, 開発プロセスにおける資源割り当て

やスケジューリングを行うことで，フォールトの効果的な検出を行うことが期待される．しかし，CK メトリクスのすべてを設計仕様書に適用するのは非常に困難である．たとえば，RFC と LCOM の計算をするためには，メソッド内部で用いられているアルゴリズムやメソッド間の呼び出し関係といったクラス内部の詳細な情報が必要である．これらの情報が記述されるのは，通常，設計フェーズの後期，実装フェーズの直前である．

5.4. 設計仕様書にメトリクスを適用する具体的な手法

本論文においては，分析/設計/実装フェーズを，進行するにつれてプロダクトに関する知識が増大していく一連のプロセスであると捉える．5.3 で述べたように，OMT に基づく分析・設計・実装のフェーズにおいて，メトリクスの中には設計フェーズの早い段階で適用可能なものと，設計フェーズの遅い段階，実装フェーズの直前でしか適用できないものがある．この事実に基づき，設計フェーズの各段階で，設計仕様書に対して適用可能なメトリクスを用いてフォールト発生の予測を行う．

まず，計測の観点から開発プロセスに 4 つのチェックポイントを導入し，各チェックポイントでどのような情報が設計仕様書に付け加わるかを明らかにする．次に，各チェックポイントで開発される設計仕様書で適用可能なメトリクスの部分集合を定義する．最後に，各チェックポイントにおいて，フォールト発生を，適用可能なメトリクスの多変量ロジスティック回帰分析を用いて予測する．

5.5. チェックポイントと適用可能なメトリクス

5.2 では，OMT の分析・設計フェーズを，クラスの構造を詳細化していく過程と見立て，6 つのステップに分割した．この分割に基づいて，設計仕様書にメトリクスを適用するために，分析/設計/実装フェーズに 4 つのチェックポイントを設ける．

(CP1)実体と関係(entity and relation)

CP1 はステップ 1, 2, 3 が完了した時点である．クラス間の参照関係とクラスの属性が決定されている．参照関係はクラス間の結合(coupling)に対応し，属性はインスタンス変数に対応する．CP1 においては，導出関係は決定されておらず，クラスライブラリ中のどのクラスが再利用されるかも設計仕様書には記述されていない．他方で，NIV は属性の情報から計算される．CBO は参照関係から計算されるが，クラスライブラリ中の再利用されるクラスへの参照は明確に記述されていないので，CBO の値は正確ではない．

(CP2)構造と継承

CP2 はステップ 4 と 5 が完了した時点である. すなわち, クラスの導出関係とクラスのメソッドが決定されている. 導出関係を決定するために, クラスの継承木が明確に記述される. 従って, DIT が導出関係から計算される. NIM はメソッドに関する情報から計算できる. 再利用されるクラスが決定されているので, CBO は正しく計算される.

(CP3)アルゴリズム

CP3 はステップ 6 が完了した時点である. すなわち, 各メソッドのアルゴリズムとメソッド間の呼び出し関係が決定されている. この情報から, LCOM と RFC が計算される.

(CP4)実装

CP4 はソースコードが実装された時点である. 各クラスについて, SLOC(ソースの行数)が計算される.

CBO は CP1 と CP2 で計算される. CBO は対象となるクラスとそれ以外のクラスとの結合の数を数えるメトリクスである. CP1 の CBO は計測対象クラスと新規に開発されたクラスとの結合のみを数えるため, 実質 CBON(3.3 参照)であると考えられる. チェックポイントと, チェックポイントにおいて計算可能なメトリクスをまとめたものを 表 5.1 に示す.

5.6. 実験的評価

5.6.1. 実験の概要

実験データは, 1997 年 8 月にある企業の新人研修で行われた C++プログラム開発演習

表 5.1 チェックポイントと適用可能なメトリクス

チェックポイント	付け加えられる情報	適用可能なメトリクス
(CP1)実体と関係	クラス間の関係, クラスの属性	NIV, CBON
(CP2)構造と継承	クラスの継承構造, メソッド, 再利用されるライブラリ	NIV, CBON, CBOR, CBO, NIM, DIT, NOC
(CP3)アルゴリズム	メソッドのアルゴリズム	NIV, CBON, CBOR, CBO, NIM, DIT, NOC, RFC, LCOM
(CP4)実装	ソースコード	NIV, CBON, CBOR, CBO, NIM, DIT, NOC, RFC, LCOM, SLOC

から収集された. 演習の概要は以下の通りである:

(a)開発者は会社の新入社員であり, 大学あるいは大学院を卒業し, 1997 年 4 月に入社した. 事前に行われた講義と演習により, オブジェクト指向設計と C++言語によるプログラミングを修得している.

(b)16 の開発チームが, 同一の要求仕様書に基づいてメール配送システムを作成する. このシステムは分散ネットワーク環境で動作し, ASCII エンコードされたメールを送受信する. 開発開始時点で, 要求仕様書, サブシステム(それぞれ SMTP サーバー, POP サーバー, DELIVER サーバー, SMTP クライアント, POP クライアント)への分割, サブシステムのインターフェイス設計がチームに与えられる. それぞれのチームのリーダーが, チームのメンバーに開発すべきサブシステムを割り当てる.

(c)チームは 4 から 5 人の開発者で構成される. インストラクターが, 開発者の能力を考慮して, 開発能力のチーム格差が小さくなるように, 開発者をチームに編成する.

(d)チームがシステムの完成を通知すると, インストラクターが受け入れテストを行う.

(e)システムは C++で実装される. 開発環境は Visual C++であり, 開発には Microsoft Foundation Class(MFC)がアプリケーションフレームワークとして用いられる. ユーザー

表 5.2 実験における各メトリクスの統計量

メトリクス	最小	最大	中央	平均	標準偏差
NIV	0	14	3	4.00	2.67
CBO	0	5	1	1.39	1.59
CBON	0	3	0	0.53	0.99
CBOR	0	4	1	0.86	0.99
NIM	0	22	3	5.73	4.86
DIT	0	6	4	3.44	1.41
NOC	0	0	0	0.00	0.00
RFC	0	27	7	8.23	6.81
LCOM	0	190	3	22.42	36.84
SLOC	0	420	71	96.43	81.01
エラー数	0	17	0	0.57	1.93
Et(分)	0	599	0	12.68	58.94

インターフェイスとソケットサービスが MFC のクラスから派生したクラスとして実装された.

5.6.2. 実験データ

開発者ごとに, メトリクスとフォールトデータを収集した. 本実験では OMT による設計仕様書は収集できなかった. そこで, ソースコードは設計仕様書を実装したものであるから, 設計仕様書のすべての情報を含んでいるという仮定に基づいて, 各チェックポイントにおけるメトリクスの値をソースコードから収集した計測値で代用した. 開発者は各々割り当てられた PC 上で作業を行い, ネットワーク経由でサーバーが 1 時間ごとに, ソースコードを収集した. メトリクス値の算出には, C++プログラムから 9 種のメトリクスを抽出するツールを用いた. 本実験では開発作業を記録するためのツールも準備され, フォールトデータの収集に用いられた. 収集されたフォールトデータは, (a)コードレビューのフェーズとテストフェーズで発見されたフォールト, (b)これらのフォールトを修正するために変更されたクラス, (c)フォールトを修正するために費やされた労力(時間), である. フォールトデータを記録していなかった, あるいはデータが欠落している開発者は, 分析の対象から除外した. 結果として, 17 人のデータ(141 個のクラス, 80 個のフォールト)が分析対象になった. 表 5.2 はメトリクス計測値の統計

表 5.3 各チェックポイントにおける係数

メトリクス	係数			
	CP1	CP2	CP3	CP4
定数 C_0	-3.37	-1.23	-1.31	-2.69
NIV	0.420	EL	EL	EL
CBON	EL	EL	EL	EL
CBOR	-	0.934	0.890	EL
CBO	-	EL	EL	EL
NIM	-	0.336	EL	EL
DIT	-	-1.16	-1.28	-0.663
NOC	-	-	EL	EL
RFC	-	-	0.284	EL
LCOM	-	-	-	EL
SLOC	-	-	-	0.0302

「EL」はそのメトリクスが変数減少法によって予測式から取り除かれたことを示す.「-」はそのメトリクスがそのチェックポイントでは適用できないことを示す.

量である. 開発されたクラスはおおむね小規模なものであったことがわかる. NIVとNIMがともに 0 のクラスがあったが, このクラスは実装のすべてを親クラスから継承していた.

5.6.3. 分析

表 5.3 に, 多変量ロジスティック回帰分析(2.4.1 参照)によって算出された予測モデルの係数を示す. CBO, CBOR, CBON には依存関係があるため(CBO = CBOR + CBON), 3 つがともに予測式に含まれることはない. DIT は複雑さに対する負の要因となった. この原因は, 本実験では多くの「ダイアログ」クラスが作られたが, 機能が単純であったにも関わらず

表 5.4 CP1 におけるフォールト予測

予測		フォールト無	フォールト有
実測	フォールト無	112	2
	フォールト有	18(43)	9(37)

括弧の外の数字はクラスの数. 括弧内の数字はクラスで発見されたフォールトの数.

表 5.5 CP2 におけるフォールト予測

予測		フォールト無	フォールト有
実測	フォールト無	109	5
	フォールト有	11(20)	16(60)

表 5.6 CP3 におけるフォールト予測

予測		フォールト無	フォールト有
実測	フォールト無	111	3
	フォールト有	9(18)	18(62)

表 5.7 CP4 におけるフォールト予測

予測		フォールト無	フォールト有
実測	フォールト無	111	3
	フォールト有	8(14)	19(66)

比較的大きな DIT を持ったことである(ダイアログクラスの DIT 値はすべて 4 であった). 観測された NOC はすべて 0 であった(表 5.2 参照)ため, NOC は正しく予測式から取り除かれている. LCOM は CP4 において予測式から取り除かれているが, これは[4][10]の結果と合致する. 表 5.4 から表 5.7 は各チェックポイントで収集されたデータを多変量ロジスティック回帰分析することで得られた予測モデルを示している. たとえば, 表 5.4 では, 112 個のクラスがフォールトを持たないと予測され, 実際にフォールトが発見されなかった. 2 個のクラスはフォールトがあると予測され, 実際にはフォールトが発見されなかった. 18 個のクラスはフォールトを持たないと予測されたが, 実際にはフォールトが発見された(43 個のフォールトを含んでいた). 9 個のクラスはフォールトを持つと予測され, 実際にフォールトが発見された(37 個のフォールトを含んでいた).

ここで, 予測式の精度を評価するために, 3 つの指標を導入する:

正確性(Correctness): 正しくフォールトがあると予測されたクラスの割合(%)

完全性(Completeness): フォールトがあるクラスが検出された割合(%)

フォールトベースの完全性: フォールトがあると予測されたクラスで実際に検出されたフォールトの割合(%).

これらの指標はそれぞれ, 以下の式によって定義される.

$Correctness = C_{PFAF} / (C_{PFAF} + C_{PFAN})$

$Completeness = C_{PFAF} / (C_{PFAF} + C_{PNAF})$

$Completeness_{faultbased} = E_{PFAF} / (E_{PFAF} + E_{PNAF})$

ここで, C_{PFAF} はフォールトがあると予測され実際にフォールトがあったクラスの数, C_{PFAN} はフォールトがあると予測されたが実際にはフォールトがなかったクラスの数, C_{PNAF} はフォールトがないと予測されたが実際にはフォールトがあったクラスの数, E_i は対応する C_i のクラスで発見されたフォールトの数である.

チェックポイント CP1 から CP4 での fault-prone 予測精度を表 5.8 に示す. 後期のチェックポイントほど, より正しく予測を行える. CP4 は開発プロセスの最終フェーズであり, 従って, CP4における予測は本実験における予測精度の上限である.

CP1 においては，完全性は低く(33%)，正確性は高い(82%)．つまり，CP1 での予測を，品質が悪いクラスをすべて列挙する目的に用いることはできないが，フォールトが発生しそうなクラスを「シード」する目的で用いることができる．シードされたクラスは重点的にレビューされテストされるクラスの候補になる．また，シードされたクラスの分布が設計レビューの判断基準になる．たとえば，シードされたクラスが設計仕様書の重要な部分に集中していて，かつ，テストが困難な部分であるなら，再設計を行うということが考えられる．

CP2 では，CK メトリクスのメソッドのアルゴリズムに関するものは用いられていないにも関わらず，CP4 を予測精度の上限と比較して，かなりよい予測精度となっている(「完全性」ではほかのチェックポイントよりも低くなっているが，「フォールトベースの完全性」ではよい成績を収めているので，フォールトを予測するという当初の目的に照らせば問題はないと考えられる)．この結果は，設計フェーズにおいて，アルゴリズムが決定していない段階で(当然ソースコードも用いず)，設計仕様書からエラーの発生を予測する可能性を示唆している．

CP3 での予測は CP2 での予測に比べて，予測精度がそれほど向上していない． 我々は，CP3 における予測精度は，「細粒度」C++設計メトリクス[9]を援用することで改善できると考えている．Chidamber らも，WMC の値は，計測されるメソッドの実装に依存すると述べている．たとえば，サイクロマチック数等を用いてメソッドの複雑さを適正に重み付けする WMC を用いることで，予測精度は改善されると考えられる．

5.7. 結論と課題

本実験では，オブジェクト指向複雑度メトリクスを用いて，設計フェーズの早期にフォールトの発生を予測する方法を提案した．この手法では，分析/設計/実装フェーズに 4 つのチェックポイントを導入した．これらのチェックポイントでは，特定の部分メトリクスのみが計測可能であった．さらに，この手法を実験的なプロジェクトに対して適用した．分析結果は提案する手法の評価と有効性を示している．

今後の課題として，以下の 3 点について研究を行う必要がある．

表 5.8 各チェックポイントにおけるフォールト予測の精度

チェックポイント	CP1	CP2	CP3	CP4
正確性(%)	82	76	86	86
完全性(%)	33	59	67	70
フォールトベースの完全性(%)	46	75	78	83

(a)提案する手法の拡張

今回利用したメトリクス以外のメトリクスに対して，提案した手法を用いることで，より正確な予測を行う．

(b)動的な複雑度メトリクスの適用

CK メトリクスはオブジェクト指向ソフトウェアの静的な複雑さを評価する．しかし，オブジェクト指向設計仕様書は動的な情報も含む（たとえば，UML(Unified Modeling Language)[54]の状態遷移図，シーケンス図，コラボレーション図）．そのような動的な複雑さを評価する方法が必要である．

(c)設計仕様書に対するメトリクスツールの開発

現在，UML がオブジェクト指向設計仕様書の標準記述言語として普及しつつある．UML をサポートする CASE ツールは，現在，独自形式のファイルフォーマットを用いている．将来的に，UML の標準的ファイルフォーマットが策定されれば，単一のメトリクスツールによって，さまざまな設計仕様書からメトリクスを計測することが可能になる．

6. オブジェクト指向プログラミング言語向けのコードクローン検出手法

6.1. 緒言

コードクローンとは，ソースファイル中の，まったく同じあるいは類似したソースコード断片のことである．クローンは「カット＆ペースト」によるコードの再利用や，実行時の性能を向上させるための意図的な繰り返しなど，さまざまな理由で作られる[5]．クローンの存在はソースファイルの首尾一貫した変更を困難にする．たとえば，あるソフトウェアがクローンによって作られた複数のサブシステムを含むとする．もしそのようなサブシステムのひとつにフォールトが発見された場合，開発者はそれ以外のクローンサブシステムをすべて修正する必要がある．巨大で複雑なシステムでは，多くの開発者が別々のサブシステムを保守することがあり，修正はより困難になる．また，クローンが存在した場合，重複したコードが規模メトリクスや複雑度メトリクスの計測に影響を及ぼす可能性がある．本章では，オブジェクト指向プログラミング言語で記述されたソースコードから，より正確にクローンを検出するための手法を提案する．提案した手法はツールに実装され，実験により，JDK のソースコードからクローンを抽出することができた．提案した手法により，従来の検出方法では見逃されてしまうようなクローンが発見できた．

6.2. クローン検出ツール edup と pdup

現在までに，さまざまなクローン検出ツールが実装され，さまざまなクローン検出アルゴリズムが用いられている[3][5][19][27][31][36]．Baker はソースファイルからクローンを検出するツール，edup と pdup を開発した[3]．ツールの入力はソースファイル（複数）であり，出力はそれらのソースファイルで発見されたクローンである．Edup と pdup においては，クローンは類似したコード断片のペアと定義されている．つまり，ある連続した行の並びが，別の連続した行の並びと同一か類似しているとき，このペアがクローンとして抽出される．Edup は 2 つの行が同じ文字を同じ順序で含む場合に等価であると判断する．Pdup は parameterized matching と呼ばれるアルゴリズムにより行を比較する．このアルゴリズムでは，2 つの行は，変数や関数の名前が変更されていても等価であると判断される．

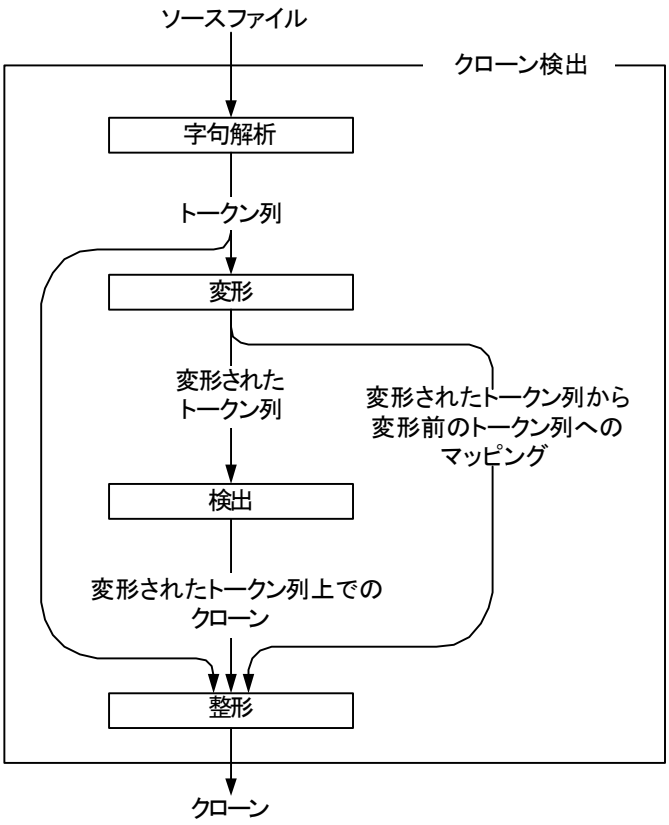


図 6.1 提案するコードクローン検出手法の概要

6.3. 問題点

C++や Java といったオブジェクト指向プログラミング言語は，クラスによるスコープや名前空間，汎用型をサポートする．結果として，識別子は名前空間やテンプレート引数に修飾されて出現することがある．クローン検出においては，そのような複雑な名前を，しばしば単純な名前と等価であると判定する必要がある．コードクローンを検出し，それらを共通のソースコードとして書き換え，整理することを考えた場合には，配列の初期化データの並びなど，書き換えに適さないクローンをフィルタリングすることが望ましい．

構造の識別や名前の変形を行うには，プログラミング言語の文法知識が必要である．したがって，そのようなクローン検出手法は，入力となるソースファイルの記述言語に依存することになるが，多くのプログラミング言語に対する移植性も確保する必要がある．次節で，クローン検出プロセスの詳細について述べる．

6.4. 提案するクローン検出手法

本研究においては，名前の変形や構造の識別を可能にするために，ソースファイルをトー

クン列として表現することとした. 従って, ソースファイル中の構造(文や関数の定義)はトークン列の部分列として表現される. 名前の変形や構造の識別は, トークン列の変形ルールによって実現することとした. 提案するトークン単位の比較によるのクローン検出プロセスの全体を図 6.1 に示す. プロセスは以下の 4 つのステップから構成される.

(1) 字句解析

ソースファイルの各行がプログラミング言語の字句ルールに従ってトークンに切り分けられる. すべてのソースファイルのトークンを連結してひとつのトークン列にするので, 単一のソースファイルからクローンを検出するのとまったく同じ方法で, 複数のソースファイルからクローンを検出することができる.

(2) 変形

トークン列は以下の(2－1),(2－2)を経て変形される. 同時に, 変形されたトークン列から変形前のトークン列へのマッピングが保存され, 後の整形ステップで用いられる.

(2－1) 変形ルールによる変形

変形ルールによって, トークン列が変形され, トークンが付け加えられたり, 削除されたり, 変更されたりする. 表 6.1 と表 6.2 に示す変形ルールは, 識別子の正規化(RC1, RC2, RJ1, RJ2)および構造の識別 RC3, RC4, RJ3, and RJ4)を行う.

表 6.1 C++向けの変形ルール

#	ルール
RC1	**(Name '::')+ Name2 → Name2** ここで, +演算子は正規表現の後置演算子であり, 1 回以上の繰り返しを意味する.
RC2	**Name '<' ParameterList '>' → Name** ここで, ParameterList は名前, 数値, 文字列, 演算子, ',', および式の並び. 式はトークンの並びであり, '('で始まって, 対応する')'で終了し, ';'を含まない.
RC3	**'=' '{' InitalizationList, '}'** **→ '=' '{' UniqueIdentifier '}'** ここで, InitalizationList は名前, 数値, 文字列, 演算子, ',', '(', ')', '{', および'}'の並び. UniqueIdentifier はユニークなトークンであり, ほかの場所には出現しない.
RC4	トップレベルの定義や宣言の終わりに UniqueIdentifier を挿入する.

表 6.2 Java 向けの変形ルール

#	Rule
RJ1	**(PackageName '.')+ ClassName** **→ ClassName** ここで, PackageName は小文字で始まる語. ClassName は大文字で始まる語.
RJ2	**NDotOrNew NClassName '('→ NDotOrNew CalleeIdentifier '.' NClassName '('** ここで, NDotOrNew は'.'や'new'以外で始まるトークン. NClassName は小文字で始まる語. CalleeIdentifier は省略されている callee をあらわす語.
RJ3	**'=' '{' InitalizationList, '}'** **→ '=' '{' UniqueIdentifier '}'** **']' '{' InitalizationList, '}'** **→ ']' '{' UniqueIdentifier '}'** ここで, InitalizationList は名前, 数値, 文字列, 演算子, ',', '(', ')', '{',および'}'の並び.UniqueIdentifier はユニークなトークンであり, 他の場所には出現しない.
RJ4	トップレベルの定義や宣言の後に UniqueIdentifier を挿入する.

(2－2)パラメータ置換

次に, 型, 変数, 定数に関係するすべての識別子が, 単一の特殊なトークンに置き換えられる(この置換は「parameterized match」[3]の前処理である). この置換により, 変数名が付け替えられたコード断片を等価とみなすことができる.

(3) 検出

変形されたトークン列のすべての部分列のうち, 等価なペアがクローンとして検出される. 各クローンは, 4 つ組 (*cp*, *cl*, *op*, *ol*)として表現される. ここで, *cp* と *op* は それぞれ最初のコード断片ともうひとつのコード断片の位置であり, *cl* と *ol* はそれらの長さである.

(4) 整形

クローンの位置が入力ソースファイル上での行番号に変換され, 整形されて出力される.

図 6.2 に, コードクローン検出プロセスを説明するための例となる C++コードを示す. 左側の数字は行番号である. この入力はトークンに切り分けられる. 切り分けられ, 変形ルールによって変形されたトークン列を図 6.3 に示す. 行 1, 3, 11, および 13 は短くなっている. 次にパラメータ置換によって再び変形される. パラメータ置換を受けた後のトークン列を図 6.4 に示す. この例では, 識別子が単一のトークン *$p* に置き換えられている.

```
1   void print_lines(const set<string>& s) {
2       int c = 0;
3       set<string>::const_iterator i
4           = s.begin();
5       for (; i != s.end(); ++i) {
6           cout << c << ", "
7               << *i << endl;
8           ++c;
9       }
10  }
11  void print_table(const map<string, string>& m) {
12      int c = 0;
13      map<string, string>::const_iterator i
14          = m.begin();
15      for (; i != m.end(); ++i) {
16          cout << c << ", "
17              << i->first << " "
18              << i->second << endl;
19          ++c;
20      }
21  }
```

図 6.2 コードクローン検出プロセスを説明するための例題コード

```
1   void print_lines ( const set & s ) {
2   int c = 0 ;
3   const_iterator I
4   = s . begin ( ) ;
5   for ( ; i != s . end ( ) ; ++ i ) {
6   cout << c << ", "
7   << * i << endl ;
8   ++ c ;
9   }
10  }
11  void print_table ( const map & m ) {
12  int c = 0 ;
13  const_iterator I
14  = m . begin ( ) ;
15  for ( ; i != m . end ( ) ; ++ i ) {
16  cout << c << ", "
17  << i -> first << " "
18  << i -> second << endl ;
19  ++ c ;
20  }
21  }
```

図 6.3 変形ルールによって変形されたトークン列

```
1   $p $p ( $p $p & $p ) {
2   $p $p = $p ;
3   $p $p
4   = $p . $p ( ) ;
5   for ( ; $p != $p . $p ( ) ; ++ $p ) {
6   $p << $p << $p
7   << * $p << $p ;
8   ++ $p ;
9   }
10  }
11  $p $p ($p $p & $p ) {
12  $p $p = $p ;
13  $p $p
14  = $p . $p ( ) ;
15  for ( ; $p != $p . $p ( ) ; ++ $p ) {
16  $p << $p << $p
17  << $p -> $p << $p
18  << $p -> $p << $p ;
19  ++ $p ;
20  }
21  }
```

図 6.4 パラメータ置換を行った後のトークン列

最終的に, クローン, すなわち, トークン列内の等価な部分列が検出される. ここで t_i を i 番目のトークン (1 <= i <= 114)とする. さらに, 行列{ d_{xy} }を, d_{xy} = 1 if t_x is equal to t_y, 0 otherwise, と定義する.行列の一部を図 6.5 に示す. 図で, d_{xy} = 1 かつ $x > y$ の部分は'*'で示した. 対称性より, $d_{xy} = d_{yx}$ であり, また, 明らかに d_{xx} = 1 であるので, x <= y の部分には何も置かない. クローンは, 行列の主対角線に平行な(右下がりの) '*'の線分として検出される. 行 1 から 7 のコード断片と, 行 11 から 17 のコード断片[2] がクローンとして検出される. 行 8 から 10 までのコード断片と, 行 19 から 21 までのコード断片がもうひとつのクローンとなる. 行 9, 10, 20, 21 は互いにクローンとなるが, 非常に短くて自明なクローンであり, クローン検出時に検出するクローンの最小行数でフィルタリングすることにより取り除くことができる.

[2] より厳密には,「行 11 から始まって行 17 の最初のトークンまでのコード断片と, 行 1 から始まって行 7 の最初のトークンまでのコード断片・・・」である. ツールの出力の中では, クローンの位置は行番号で示される.

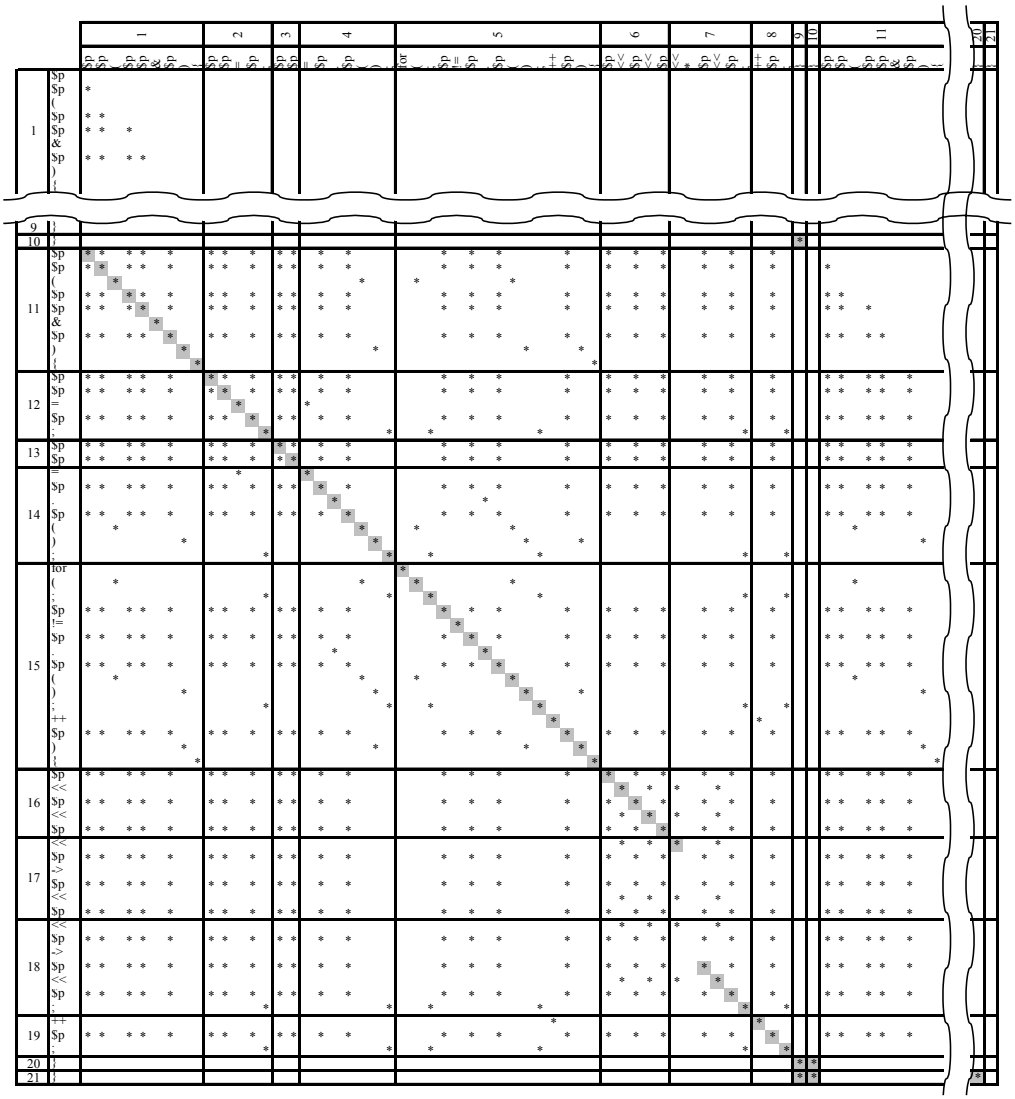


図 6.5 トークン単位の散布図

6.5. JDK への適用実験

提案するクローン検出手法の有効性を評価するため，手法を実装するツールを開発し，JDK 1.2.2[53]に対して適用した．JDK は広く用いられている Java のライブラリであり，ソースファイルも公開されている．クローン検出ツールは，サンプルとデモプログラムを除く，すべての JDK ソースファイルに適用された．入力ソースの規模は 50 万行，ファイル数で 1648 となった．ツールの実行には，Pentium III 650MHz および 1GB の RAM を持つ PC で約 3 分を要した．

図 6.6 は 20 行以上のクローンの散布図を示す. グラフの両軸はソースファイルの行を表現する. ソースファイルはパスの辞書順に並べられているので, 同じディレクトリにあるソースファイルは軸上でも近くに存在する. クローンは右下がりの線分で表現される. 主対角線の下側にだけクローンを図示する. 図では, 線分はほとんど点にしか見えないが, クローンの長さが数十行であり, 軸のスケールと比べて小さいためである. ほとんどの線分は主対角線のすぐ近くに位置しており, これは, 単一のファイルの中か, あるいは(ファイルシステム内で)近傍のファイルの間でクローンが発生していることを意味する. 図中 29A で示される込み合った部分は, `src/javax/swing/plaf/multi/*.java` の 29 のソースファイルに対応する. これらのファイルは互いに類似しており, それらのいくつかは親クラスを除いてまったく同じクラスの定義を含んでいた.

そのような類似したファイルの例として, 図 6.7 に `MultiButtonUI.java` と `MultiColorChooserUI.java` を示す. 2 つのファイルの違いはわずかに 3 行(行 32,

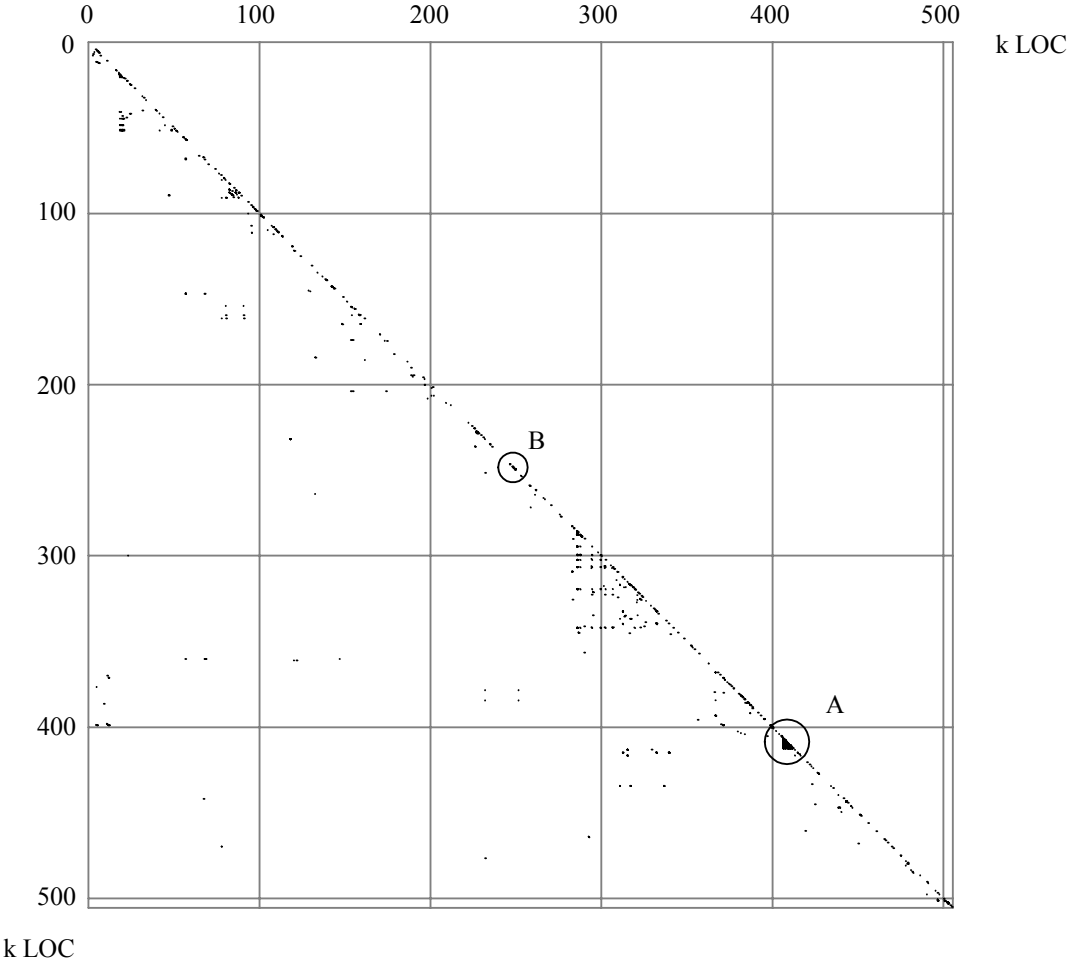


図 6.6 JDK ライブラリで発見されたコードクローンの散布図

```
 31|  */
 32| public class MultiButtonUI extends ButtonUI {
 33|

160|     public static ComponentUI createUI(JComponent a) {
161|         ComponentUI mui = new MultiButtonUI();
162|         return MultiLookAndFeel.createUIs(mui,
163|                 ((MultiButtonUI) mui).uis,
164|                 a);
165|     }
```

(a) MultiButtonUI.java

```
 31|  */
 32| public class MultiColorChooserUI extends ColorChooserUI {
 33|

160|     public static ComponentUI createUI(JComponent a) {
161|         ComponentUI mui = new MultiColorChooserUI();
162|         return MultiLookAndFeel.createUIs(mui,
163|                 ((MultiColorChooserUI) mui).uis,
164|                 a);
165|     }
```

(b) MultiColorChooserUI.java

これら 2 つのファイルは太字で示されている 3 ヶ所を除いてはまったく同一である.

図 6.7 JDK のソースファイルで発見された類似コードの一例

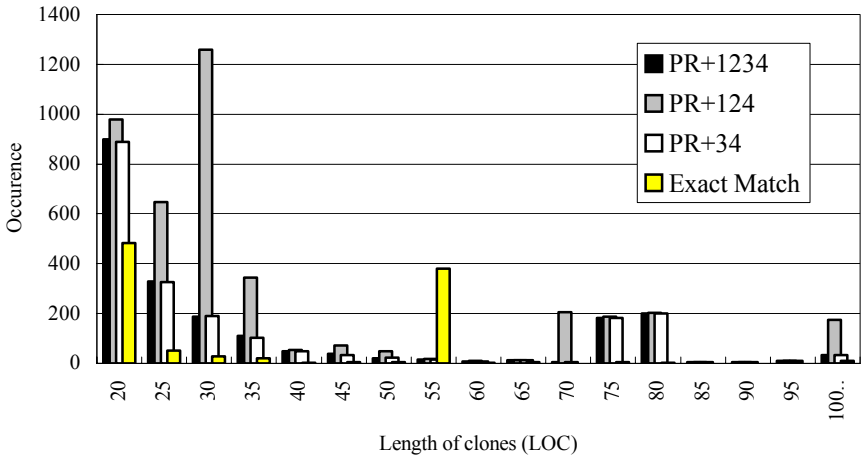
161, 163)だけである. ソースファイルのコメントによると, これらのファイルは AutoMulti と呼ばれるコード生成器によって作成された. これらのファイルを修正するためには, 開発者はそのツールを何らかの方法で入手し(ツールは JDK には含まれていなかった), 修正し, 正しく適用しなければならない. ツールを用いないとすれば, 手作業ですべてのファイルを注意深く修正しなければならない. この例が示すように, クローンの修正は余分な作業を必要とする. これらのクローンを共通化コードとして書き換えるには, 汎用型[8]を用いることが考えられる(ただし, 現在の Java は汎用型をサポートしてない). 最長のクローン(349 行)は src/java/util/Arrays.java で発見された(図中では B で示される部分). このクローンは, "sort"と言う名前を持つメソッドを含んでいた. sort はシグネチャ(引数の型と数)が異なる 18 のバリエーションがあり, それらはソーティングを行う同一のアルゴリズム／ルーチンを用いていた.

6.6. 変形ルールの評価

提案した Java 向けの変形ルールの効果を評価するために, クローン検出ツールを, 変形ルールの一部を無効にして適用した. 図 6.8 は変形ルールの一部を適用しなかった場合に発見されるクローンの長さの度数分布である. PR+1234 はパラメータ置換とすべての変形ルール(RJ1, RJ2, RJ3, RJ4)が適用された場合(すなわち, 検出ツールのデフォルト)を表す.「Exact Match」はパラメータ置換も変形ルールも適用されない場合を表す. 全体的な

傾向として，より長いクローンはより少ないことが分かる．80 行の付近のピークは，AutoMulti によって生成されたコードであり，Exact Match では検出されない．

この実験では，PR+1234 で発見されたクローンは，PR+124 で発見されたクローンよりも少数である．これは，RJ3 が多くのテーブル初期化コードを取り除いたことを意味している．PR+1234 は 2111 個のクローンを検出し，PR+34 は 2093 のクローンを検出している．少数ではあるが，RJ1 と RJ2 の導入によって検出可能になったクローンが存在することがわかる．図 6.9 はそのようなクローンの一部である．下段のコード断片はメソッドをクラス名つきで(Utility.arranRegiionMatches)呼び出している一方で，上段のコード断片はメソッ



"PR+1234"はパラメータ置換と変形ルール RJ1 RJ1, RJ2, RJ3, RJ4 が適用されていることを意味する.

図 6.8 JDK ライブラリで発見されたクローンの長さの度数分布

```
if (hashes[i] == hashes[j] &&
        arrayRegionMatches(values, iBlockStart,
        values, jBlockStart, BLOCKCOUNT)) {
    indices[i] = (short)jBlockStart;
    break;
}
```

```
if (hashes[i] == hashes[j] &&
        Utility.arrayRegionMatches(values, iBlockStart,
        values, jBlockStart, BLOCKCOUNT)) {
    indices[i] = (short)jBlockStart;
    break;
}
```

図 6.9 RJ2 によって検出されたクローンコードの一部

ドをクラス名なしで(arrayRegionMatches)呼び出している. Exact Match では, 少数のクローンしか検出されていない. Exact Match で発見されるクローンはまったく同じトークンの並びを意味しており, 容易に共通化コードとして書き換えることができるクローンである. しかしながら, 変形ルールとパラメータ置換を用いて始めて発見されるようなクローンは, Exact Match によって発見されるクローンのほぼ 2 倍に達しており, 書き換えによる再構造化の機会を増すことになる.

6.7. 結論と課題

本章では, オブジェクト指向プログラミング言語向けのコードクローン検出手法を提案した. 本手法は, 変形ルールとトークンごとの比較を用いて, オブジェクト指向プログラミング言語の機能であるクラススコープや名前空間による, 複雑な名前が用いられているソースコードから, コードクローンを検出することを目的としている. 実験により, 提案する手法が JDK のライブラリから効果的にコードクローンを検出することが確認された.

今後の課題としては, 以下のような問題に取り組むことを予定している.

(a) 多くの言語に対応したクローンの検出

筆者は現在, 複数のプログラミング言語で記述されるソフトウェアのソースコードから効果的にクローンを検出する方法を研究中である.

(b) クローンに関するメトリクス

クローンに関するメトリクスとしては, 基本的なメトリクス(大きさ, 数)以外にも, クローンを除去した場合に減少するソースコードの量が提案されている. 筆者が所属するグループにおいても, クローンに関するメトリクスを研究しているところである. クローンに関するメトリクスは, 発見されたクローン全体によるによる品質の劣化を評価するもの, 個々のクローンの影響を評価するもの, クローンを選択するためのものなど, さまざまなものが考えられる.

(c) クローンと他のメトリクスの関係

クローンの存在が他のメトリクスの計測値に影響するのは明らかであるが, クローンを除去する前と除去した後で複雑度メトリクスなど計測し比較する事例や, クローンの影響を考慮しつつメトリクスを計測する具体的な手法はいまだ提案されていない. また, クローンと他のメトリクスの相関などについては, 筆者が所属するグループが参加している共同プロジェクトにおいて研究しているところである.

7. むすび

7.1. 結論

本研究においては，再利用技術，設計，開発プロセス，の 3 つの観点に注目し，オブジェクト指向複雑度メトリクスの新しい適用手法を提案し，実験的な評価を行った．また，従来の（オブジェクト指向ではない）ソフトウェア向けの重複コード検出技術をオブジェクト指向ソフトウェアに適用するための手法を提案し，実験的な評価を行った．

まず，再利用部分と新規開発部分を区別するメトリクス修正法を提案した．実験により，修正された CK メトリクスは，修正前の CK メトリクスよりも，フォールトの個数や修正時間との相関が高いことが示された．次に，クラス階層を利用して新規開発クラスの分類を行う手法を提案した．実験により，クラス分類によってフォールト修正時間の予測精度が向上することが確認された．この実験は同時に，フォールト修正時間の予測においても，修正 CK メトリクスが有効であることの評価にもなっている．次に，OMT 開発プロセスにおいて，開発の早期段階でオブジェクト指向複雑度メトリクスを適用する手法を提案した．実験によって，開発の早期における予測が，ある程度の正確性をもって，fault-prone なクラスを指摘することが確認された．この実験は同時に，ロジスティック回帰分析による fault-prone クラスの予測においても，修正 CK メトリクスが有効であることの評価にもなっている．次に，オブジェクト指向プログラミング言語で記述されたソースコードから，より正確にコードクローンを検出するための手法を提案した．実験によって，従来のコードクローン検出手法では検出されないクローンを検出できることが確認された．

7.2. 課題と展望

メトリクスの修正に関する短期的な課題としては，CK メトリクス以外のメトリクスを提案した方法により修正し，さまざまな再利用ライブラリに対して適用することがある．現在，筆者が所属する研究グループにおいて CK メトリクス以外のメトリクスを修正し，適用実験を行っているところである．本研究における実験は，教育環境における実験であったため，一度開発したソフトウェアを保守したり，何度も開発を繰り返したりする事例は観察されていない．より実際的な繰り返し型ソフトウェア開発プロセスにメトリクスを適用する実験を行うことで，新たな知見と問題が得られるはずである．本論文で議論したコードクローン検出技術についての短期的課題としては，XP（extreme programming）に代表される，リエンジニアリングによる継続的な設計の変更を前提とした開発プロセスが提案されている．本論文で議論したコードクローンやクローンに関するメトリクスがそのような設計の変更のための判断基準となるか，ある

いは，コードクローンと設計変更の関係などについて，定量的な評価が必要である．

中期的な課題としては，現在も発展し続けるソフトウェア開発技術に対して，従来のメトリクスあるいはコードクローン検出手法やその適用方法は適切か，あるいは適切ではないならどのようにすればよいか，という問題がある．ソフトウェア開発技術については，UML，デザインパターン，汎用型による（静的な型チェックを損なわない）汎用コンテナや汎用アルゴリズム，プログラミング言語間の相互運用性の向上による複数言語プログラミングなど，多くの技術が登場してきている．これらの技術が複雑度メトリクスやコードクローンの利用に影響を及ぼすと考えられる．

UML により，要求仕様書（あるいはシステムレベルの分析）が構造化された文書として記述されるようになり，その振る舞いの（動的な）複雑度を計測するメトリクスが登場した．筆者が所属する研究チームにおいても，動的なメトリクスと静的なメトリクスを比較する研究が始まっている．コードクローンを発生させやすい設計の特徴を調べたり，設計書からコードクローンを予測する方法を探るなど，多くの研究課題がある．従来のオブジェクト指向複雑度メトリクスは，クラス（あるいはシステム全体）を計測対象とすることが多かったが，デザインパターンなどの，複数のクラスを（それぞれのクラスに一定の役割を持たせ）組み合わせる考え方が一般化すれば，複数のクラスの組み合わせを計測対象とする，あるいはクラスがパターンの中で果たしている役割を考慮する複雑度メトリクスも考えられる．汎用型を用いて定義されたクラスは，実際に型パラメータが与えられるまで，どのようなクラスと関係があるかを決定できない．従来の複雑度メトリクスは，クラスの定義だけからクラス間の関係が決定できると仮定しているため，汎用型を用いたソースコードにそのまま適用することはできない．汎用型によって共通化コードとして書き直せるようなコードクローンも存在するため，汎用型はコードクローンの書き直しにおいて無視できない問題である．複数言語による開発においては，言語によって扱うオブジェクトの粒度が異なる，あるいは，オブジェクト指向言語と関数型言語の併用も考えられるため，オブジェクト指向複雑度メトリクスの大前提であるオブジェクトやメソッドの概念が揺らいでしまい，適用が困難になるかもしれない．コードクローン検出においては，構文が大幅に異なるプログラミング言語で記述されたソースファイルから，意味的に似ているコード断片を探し出すことが必要とされるかもしれない．

長期的な課題としては，複雑度メトリクスの構造に関する研究がある．現在乱立している複雑度メトリクスを，一定の構造化により比較する研究が行われつつある．また，より根本的な問題として，現在の複雑度メトリクスは，人間が起こすエラーを予測するのに使われるが，人間の認識に関するモデルを内包していない．多くの複雑度メトリクスは，「ソフトウェアを何らかのグラフで表現したときに，頂点や辺が多いほど複雑である」という定義を用いている

（本論文においても，メトリクスの性質を議論する際に，このような形式の定義を用いた．3.4参照）．複雑度メトリクスは，いずれ，人間の記憶力の限界や，人間はどのようにエラーを起こすかといった，人間の認識に関する研究と統合される必要があるだろう．

謝辞

本研究を行うにあたり，常日頃より適切なご指導を賜りました井上克郎教授に深く感謝いたします．講義などを通じて，さまざまなご指導とご教示を賜った，都倉信樹教授に深く感謝いたします．講義や国際会議などを通じて，さまざまなご指導とご教示を賜った，菊野亨教授に深く感謝いたします．

講義などを通じて，さまざまなご指導とご教示を賜った，今川正治教授，柏原敏伸教授，北橋忠宏教授，谷口健一教授，萩原兼一教授，橘本昭洋教授，東野輝夫教授，藤原融教授，増澤利光教授，宮原秀夫教授，村田正幸教授，首藤勝元教授(現大阪工業大学教授)，故西川清史教授，故藤井護教授に心から感謝いたします．

本研究の遂行にあたり，さまざまなご配慮とご協力をいただいた，日本ユニシス株式会社の毛利幸雄氏，齋藤滋氏，高橋優亮氏，小谷野圭司氏，尾畑祐一氏，川南理恵氏に深く感謝いたします．

研究会や会議における質疑応答を通じて有益なご意見を賜った鳥居宏次副学長（奈良先端科学技術大学院大学）に心から感謝いたします．

特に 6 の内容に関して，ミーティングなどを通じて有益なご意見をいただきました佐藤慎一氏（株式会社 NTT データ），加藤裕史（株式会社 NTT データ），門田暁人助手（奈良先端科学技術大学院大学），中江大海氏（奈良先端科学技術大学院大学）に深く感謝いたします．

本論文をまとめるにあたり，楠本真二助教授には，常に適切なご指導を賜り，また細部にわたる議論に応じていただきました．ここに深く感謝いたします．松下誠助手には，ミーティングにおける議論などを通じてご助言を賜りました．心から感謝いたします．高林修司氏（現松下通信工業株式会社），柏本隆志氏（現株式会社日立製作所），上村拓也氏（現ソニー株式会社パーソナル IT ネットワークカンパニー），竹原元康氏，藤井邦浩氏（現奈良先端科学技術大学院大学），ならびに大阪大学大学院基礎工学研究科情報数理系専攻ソフトウェア科学分野井上研究室の方々には，ミーティングなどを通じてさまざまなご助言をいただきました．深く感謝いたします．

参考文献

[1] A. Abran and P. N. Robillad, Function Point Analysis: “An Empirical Study of Its Measurement Process,” *IEEE Trans. on Software Eng.*, Vol. 22, No. 12, pp. 895-909 (Dec., 1996).

[2] 青木淳: オブジェクト指向システム分析設計入門, 株式会社ソフト･リサーチ･センター (1993).

[3] B. S. Baker: “On finding Duplication and Near-Duplication in Large Software System,” *Proc. of The Second IEEE Working Conf. on Reverse Eng.*, pp. 86-95, Tronto, Canada (Jul., 1995).

[4] V. R. Basili, L. C. Briand, W. L. Mélo: “A validation of object-oriented design metrics as quality indicators,” *IEEE Trans. on Software Eng.*, Vol. 20, No. 22, pp. 751-761 (1996).

[5] I. D. Baxter, A. Yahin, L. Moura, M. Sant’Anna, and L. Bier: “Clone Detection Using Abstract Syntax Trees,” *Proc. of The IEEE Int’l Conf. on Software Maintenance (ICSM) ’98*, pp. 368-377, Bethesda, Maryland (Nov., 1998).

[6] R. V. Binder: *Testing Object-Oriented Systems - Models, Patterns, and Tools* -, Addison-Wesley Longman, Inc., (2000).

[7] G. Booch: *Object Oriented Analysis and Design with Applications*, The Benjamin / Cummings (1994).

[8] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler: “GJ Specification.” <http://cm.bell-labs.com/cm/cs/who/wadler/pizza/gj/>

[9] L. C. Briand, P. Devanbu, and W. Mélo: “An Investigation into Coupling Measures for C++,” *Proc. of The IEEE 19th Int'l Conference on Software Eng.*, pp.412-421, Boston, USA (May, 1997).

[10] L. C. Briand, J. Daly, V. Porter and J. Wüst: “Predicting Fault-Prone Classes with Design Measures in Object-Oriented Systems,” *Proc. of the IEEE 9th Int'l Symposium on Software Reliability Eng.*, pp.334-343, Paderborn, Germany (Nov., 1998).

[11] L. C. Briand, J. W. Daly, and J. Wüst: “A Unified Framework for Coupling Measurement in Object-Oriented Systems,” *IEEE Trans. on Software Eng.*, Vol. 25, No. 1. pp. 91-121 (1999).

[12] L. C. Briand, J. Wüst, J. W. Daly, D. V. Porter: “Exploring the relationships between design measures and software quality in object-oriented systems,” *The Journal of Systems*

and Software, No. 51, pp. 245-273 (2000).

[13] S. R. Chidamber, D. P. Darcy, and C. F. Kemerer: “Managerial Use of Metrics for Object-Oriented Software: An Exploratory Analysis,” *IEEE Trans. on Software Eng.*, Vol. 24, No. 8, pp. 629-639 (1998).

[14] S. R. Chidamber and C. F. Kemerer: “A metrics suite for object-oriented design,” *IEEE Trans. on Software Eng.*, Vol. 20, No.6, pp.476-493 (1994).

[15] P. Coad and E. Yourdon: *Object Oriented Analysis, 2nd ed.*, Yourdon Press (1991).

[16] J. S. Collofello and S. N. Woodfield: “Evaluating the effectiveness of reliability-assurance techniques,” *Journal of Systems and Software*, Vol.9, No.3, pp.191-195, Berlin, Germany (Mar., 1989).

[17] P. Devanbu, S. Kartsu, W. Melo, and W. Thomas: “Analytical and Emprical Evaluation of Software Reuse Metrics,” *Proc. of the IEEE 18 th Int’l Conf. of Software Eng.*, pp. 189-199 (1996).

[18] A. Diller: *Z - An Introduction to Formal Methods, 2nd Ed.*, John Wiley & Sons, Inc. (1994).

[19] S. Ducasse, M. Rieger, and S. Demeyer: “A Language Independent Approach for Detecting Duplicated Code,” *Proc. of IEEE Int'l Conf. on Software Maintenance (ICSM) '99*, pp. 109-118, Oxford, England (Aug., 1999).

[20] N. E. Fenton, and S. L. Pfleeger, *Software Metrics, A Rigorous & Practical Approach. 2nd ed.,* Thomson, London (1996).

[21] E. Gamma, R. Helen, R. Johnson, and J. Vlissides, *Design Patterns*, Addison-Wesley Publishing Co., (1995).

[22] M. H. Halstead: *Element of software science*, New York, Elsevier North-Holland (1977).

[23] 飯塚悦功 監修,「ソフトウェア分野における ISO9000」研究会 編: ソフトウェア品質システム審査登録ガイド ソフトウェア ISO9000, 日科技連(1996).

[24] I. Jacobson: *Object-Oriented Software Engineering - A Use Case Driven Approach* -, Addison-Weslay Publishing Co., (1992).

[25] I. Jacobson, G. Booch, and J. Rambaugh: “The Unified Process,” *IEEE Software*, May/June 1999, pp. 96-102 (1999).

[26] I. Jacobson, M. Griss, and P. Johnson: *Software Reuse - Architecture Process and Organization for Business Success* -, ACM Press (1997)

[27] J. H. Johnson: “Identifying Redundancy in Source Code using Fingerprints,” *Proc. of IBM*

Centre for Advanced Studies Conference (CAS CON) '93, pp. 171-183, Toronto, Ontario (Oct., 1993).

[28] S. H. Kan: *Metrics and Models in Software Quality Engineering*, Addison-Wesley (1995).

[29] E. M. Kim, O. B. Chang, S. Kusumoto, and T. Kikuno, Analysis of metrics for object-oriented program complexity, *Proc. of The 18th IEEE Computer Software & Application Conference (COMPSAC) '94*, pp. 201-207, Taipei (Nov., 1994).

[30] 久米 均, 飯塚 悦功: 回帰分析 シリーズ入門 統計的方法 2, 岩波書店 (1987).

[31] B. Laüge, E. M. Merlo, J. Mayrand, and J. Hudepohl: "Assessing the Benefits of Incorporating Function Clone Detection in a Development Process," *Proc. of IEEE Int'l Conf. on Software Maintenance (ICSM) '97*, pp. 314-321, Bari, Italy (Oct., 1997).

[32] W. Li: "Another metric suite for object-oriented programing," *The Journal of Systems and Software*, No. 44. pp. 155-162 (1998).

[33] M. Lorenz and J. Kidd: *Object-Oriented software metrics*, New Jersey, Prentice Hall (1994).

[34] G. C. Low and D. R. Jeffery: "Function Points in the Estimation and Evaluation of the Software Process," *IEEE Trans. on Software Eng.*, Vol. 16, No. 1, pp. 64-71 (1990).

[35] 前谷 俊三: 臨床生存分析 生存データと予後因子の解析, 南江堂 (1996).

[36] J. Mayland, C. Leblanc, and E. M. Merlo: "Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics," *Proc. of IEEE Int'l Conf. on Software Maintenance (ICSM) '96*, pp. 244-253, Monterey, California (Nov., 1996).

[37] T. J. McCabe: "A complexity measure," *IEEE Trans. on Software Eng.,* Vol. SE-2, No.4, pp.308-320 (1976).

[38] 中谷多哉子, 玉井哲雄, : "オブジェクトの進化モデル構築に向けて," 情報処理学会研究報告 (SE-115), Vol. 97, No. 74, pp.133-140 (1997)

[39] P. Nesi and T. Qurci: "Effort estimation and prediction of object-oriented systems," *The Journal of Systems and Software*, pp. 89-102 (1998).

[40] P. Oman and S. L. Pfleeger: *Applying Software Metrics*, IEEE Computer Society Press (1997).

[41] M. C. Paulk, et al: *The Capability Maturity Model: Guidelines for Improving the Software Process*, Addison Wesley Publishing Co., Inc. (1995).

[42] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy and W. Lorensen: *Object Oriented Modeling and Design*, Prentice Hall (1991).

[43] S. Schlaer and S. Mellor: *Object Oriented System Analysis*, Prentice Hall (1988).

[44] N. F. Schneidewind: “Measuring and Evaluating Maintenance Process Using Reliability, Risk, and Test Metrics,” *IEEE Trans. on Software Eng.*, Vol. 25, No. 6. pp. 769-781 (1999).

[45] R. van Solingen and E. Berghout: *The Goal/Question/Metric Method: a practical guide for quality improvement of software development*, McGraw-Hill Int’l (UK) Ltd. (1999).

[46] S. Srinivasan: “Design Patterns in Object-Oriented Frameworks,” *IEEE Computer, Feb., 1999*, pp. 24-32 (1999).

[47] G. V. Subramaniam and E. J. Byrne: “Reengineering the Class - An Object Oriented Maintenance Activity,” *Proc. of The 18th IEEE Computer Software & Application Conference (COMPSAC) ’98*, pp. 39-44, Vienna, Austria (Aug., 1998).

[48] E. J. Weyuker: “Evaluating software complexity measures,” *IEEE Trans. on Software Eng.*, Vol.14, No.9, pp.1357-1365 (1988).

[49] S. M. Yacoub, H. H. Ammar, and T. Robinson: “Dynamic Metrics for Object-Oriented Designes,” *Proc. of the 6th IEEE Int’l Software Metrics Sympo.*, pp. 50-61 (Nov., 1999).

[50] 山田茂, 高橋宗雄: ソフトウェアマネジメントモデル入門, 共立出版 (1993).

[51] 山崎利治: “共通問題によるプログラム設計技法解説”, 情報処理学会誌, 25, 9, p. 934 (1984).

[52] Metamata Metrics. <http://www.metamata.com/>.

[53] The source for Java Technology. <http://java.sun.com/>.

[54] *UML Modeling Language, Standard Software Notation*. <http://www.rational.com/>.

[55] XProgramming.com <http://www.xprogramming.com/>.